Mocking an Integrated Clinical Environment with JavaScript

Charlie Meyer
University of Illinois Urbana-Champaign, cemeyer2@illinois.edu

Abstract – **Currently, no standards exist for the safe operation and communication between medical devices. By developing these standards, the Medical Device Plug n Play project aims to improve patient safety. Our group was tasked with gathering all of the requirements of the project and developing a mock system that demonstrates the possibilities of having medical devices function in coordination with each other. From initial design to final testing, we utilized a variety of technologies to implement a successful mock environment that integrated actual hardware in several scenarios to demonstrate the potential of the system.**

BACKGROUND

Every year in hospitals across America, there are thousands of accidents that occur due to unsafe or improper interactions between medical devices. Medical devices are highly complex cyber physical systems each having their own operating constraints, safety certifications and levels of reliability. Each device must work properly in the context of the operation being performed on the patient by all the devices connected to that patient simultaneously while still managing all of the requirements for each device [1]. This high degree of complexity raises many questions and challenges:

- How do we ensure that all of the data coming from each device attached to the patient is interpreted properly?
- How can the data collected from each device be reassembled into a format the each other device can consume reliably?
- Since each device has different reliability and safety levels, how can we verify that a system composed of many devices is safe and reliable from one state to the next?

Our group was tasked with taking all of the above questions into consideration and designing a mock system that was able to simulate an actual integrated clinical environment with many devices connected to one patient. We balanced the needs of safety and reliability with the desire to have a fully functioning system that accurately mocks what one may expect to see in a hospital room in the future.

SYSTEM DESIGN

From the onset of the project, we were given several use cases that described how the system should perform under varying conditions. Each use case reads like a story and describes what devices are attached to the system, the operational constraints of each device, how each device should interact with the other devices in the system, and what input the system will be receiving from both the patients and the clinical staff.

To accomplish our goals, we chose to work with the JavaScript programming language. This lightweight and dynamic language allowed us to leverage many of its best features to create an environment that accurately mocks each of the use cases. Many traditionally think that JavaScript is a language that was created simply for enhancing functionality of internet sites, but in recent years JavaScript has matured and today is used for many more applications beyond glorifying websites. For instance, many desktop widgets are built using pure JavaScript [2] and Palm's new webOS handles applications that are written in JavaScript [3].

Using JavaScript, however, did present several challenges to us initially. Primarily, we were unfamiliar with the language and it took several weeks of learning to become familiar with proper syntax, use of features, and popular design patterns. Once we were comfortable with the language, we were able to begin to design the system utilizing some of the best features that JavaScript has to offer.

To assist in our design of the environment, we turned to the Unified Modeling Language to design diagrams of how we thought the system should be laid out. After several iterations of design and discussion, we had decomposed the design into several main components that each served its own purpose:

- Controller – the central node of the system. All devices interact with each other through the controller and all rules get processed through the controller.
- Devices – each device object in the system represents one hardware device attached to the patient. Devices inform the controller of their capabilities and the controller dynamically adjusts to accommodate each device.
- Events – any object in the system can fire an event. When an event has one or more handlers attached to it, each handler is executed.
- Rules – rules govern how the system works when events are fired. Each rule processes one

or more conditions and takes action based on the evaluation of the set of conditions it contains.

- Hardware interface – a static interface that allows the JavaScript objects to communicate through it to access the hardware.
- User Interface – a simple way to interact with the system through a web browser

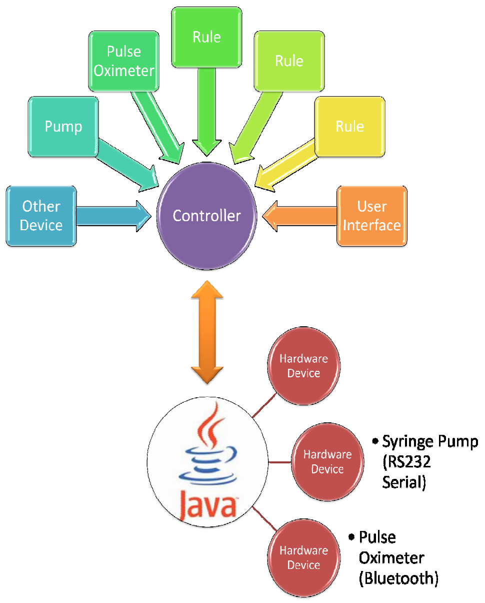


FIGURE 1
BASIC DESIGN OF THE INTEGRATED CLINICAL ENVIRONMENT

IMPLEMENTATION

To assist in our implementation of the system, we turned to a mixture of software development methodologies to guide us. We took some elements from Extreme Programming and several elements from Scrum to combine into a lightweight process that suited our needs. We used the customer/client approach from Extreme Programming and adopted a version of Scrum's product backlog to decompose the actual implementation into tasks and iterations with set tasks at each step.

I. Server Side Components

The first component to be implemented was the hardware interface to allow the JavaScript code to communicate with the actual hardware that we used for our experiments. Since JavaScript runs in a sandbox environment, this was a necessary step to allow us to use the hardware and JavaScript in combination. We chose to use Java to implement the hardware bridge due to our existing familiarity with the language and ease of getting a functional interface running quickly.

The Java component of our environment was tasked with communicating with the hardware and translating each individual device's communications protocol into one unified language that our JavaScript components could process. One of the emerging formats on the internet for communication between server side software and client side JavaScript is JavaScript Object Notation or JSON. JSON is a lightweight key/value format for transmitting sets of data. Since libraries exist in both JavaScript and Java to create and parse JSON objects, it was a natural fit to be the vehicle for all of our communications.

The interface exposed by the Java component was a HTTP server that used the URL as the UI design pattern to effectively communicate to clients what operations each device exposed and how to pass in the proper parameters for each operation.

II. Controller

After the hardware interface was complete, we moved on to implementing the JavaScript infrastructure. The first component we tackled was the centralized controller of the system. The controller was designed so that all communication between any of the other components passed through it, allowing for fine grained constraints to be placed on all aspects of operation of the system.

The design of the controller allows for a collection of devices to be kept, with each device identified by a unique identifier supplied by that device. This way, the controller is able to effectively keep track of each device in the system.

The controller also acts as the mechanism through which each of the rules in the system is evaluated. Even though each rule could be evaluated autonomously, by providing an execution environment for the rules in the controller, the system is able to more gracefully handle error conditions and communicate results of rule evaluations to other components of the environment.

III. Devices

The next step in implementing our system was to create software object models of each of the hardware devices in the system. Where we did not have a physical hardware device that we needed, we created mock software objects designed to mimic the functionality of what the actual hardware would have done had it been connected. Each device we implemented inherited from abstract models of classes of devices, such as sensors and actuators. These classes inherited functions and properties from an abstract device model. Using this hierarchy, we were able to provide a consistent interface for addressing each device, while still allowing each device model to expose its own unique functionality.

We initially concentrated on creating two software device models, one for an infusion pump and one for a pulse oximeter. The infusion pump model inherited from the actuator class of devices and contained all of the functionality to operate the pump. This required the model to read from and send commands to the hardware interface. The pulse oximeter model was a read only design, since its sole purpose was to take readings from a patient and pass the data on to the patient model.

To accomplish communication, we had to modify the way that native JavaScript accomplishes network communication. Since JavaScript runs in a browser sandbox, any network communication methods that JavaScript can utilize must be provided by the underlying engine that the JavaScript is running on top of. Due to security concerns, JavaScript is limited to accessing only resources that are on the same host and port as the page containing the JavaScript was loaded from. This posed a serious problem to us, since we wanted to be able to have a more flexible system than have everything run from a single host and port.

To get around this limitation, we used an existing Flash communications library that provides objects in JavaScript to access a Flash object to handle network communication. Since Flash does not have any of the limitations that native JavaScript networking does, this alternative was a natural fit for our project. Since the flash transport was a drop in replacement for the native communication, only several small changes were required to modify the code to support it. This change also allowed several developers to work with the same hardware simultaneously, since Flash could communicate to hosts other than the one where the JavaScript was being served from. We were able to have all the hardware connected to one computer running the Java interface software, with multiple clients connected to it. This allowed for more rapid development since hardware resources could be shared.

In the spirit of making the internet feel more like a desktop application, communication is by default asynchronous. This presented several challenges for us, but we were able to leverage some of the features of JavaScript to overcome this. For instance, if a client requested to one of our infusion pump models that it begin pumping fluid, the request would be sent asynchronously to the hardware interface and execution would continue. There is no traditional way to notify back to the requestor if the request was successful or what the results of the request were. To enable this functionality, we relied on the traditional JavaScript method of providing a mechanism for the caller of a function to provide a callback function that would be executed after the request completed.

These callback functions would by definition take in parameters as documented in the function that they were provided to and could take whatever action the caller wanted to do with the data. Referring to the infusion pump example, the requestor of the pump to run could provide a function that would take in the updated status of the infusion pump and update the patient status given the new data in the pump status, so when the request completed that action would be taken. Since in JavaScript functions are first class variables, they can be passed in just like any other type of variable. This flexibility allowed us to have a system that was actually more flexible than one that would have been implemented with synchronous communication due to the fact that the callback function could be as dynamic as the caller desired it to be.

IV. Events

The event subsystem is a critical component of the system because it is what powers the rules framework and the user interface to be interactive. The central idea of the events subsystem is that there exists a set of events maintained in a central repository, and clients can attach handlers to each of the events known to the system. When a component fires an event, the events subsystem accesses which handlers are attached to that event and safely executes them asynchronously, passing each of the parameters the component that fired the event passed into the event subsystem to each handler. Events can be fired due to new input from a hardware device, input from a user, or from conditions arising such as an error in communication.

Once the system loads, the events subsystem automatically scans all loaded objects in the environment to see if they fire events and if they do, registers each of the events with it. Events and handlers are stored internally in a two dimensional associative array, which allows keying objects in the array by the name of the event. The first level of the array stores each of the events and for each event there is an array of zero or more functions that act as the handlers for that particular event. This allows for a simple and quick implementation. When an event is fired, it is one access to the array to access the handlers for the event, then N accesses through the array of handlers to run each handler.

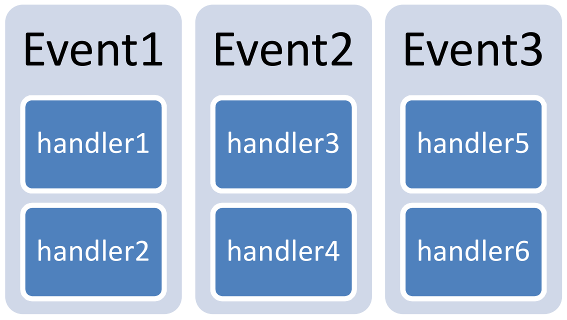


FIGURE 2
MODEL OF EVENT AND HANDLER STORAGE

Each handler is executed safely and asynchronously. This is done to ensure reliability and safety of the system. If one handler locks up or causes an error to be thrown, it does not cause an impact on the guarantee that any other handler does not begin execution within a reasonable amount of time from when the event was fired. Any errors that handlers might throw are captured and do not impact the other handlers from executing properly. The only possible side effect that one handler can have on another with out system is a competition for resources. This is due to the fact that handlers are run nearly parallel, so computing resources such as processor cycles must be shared.

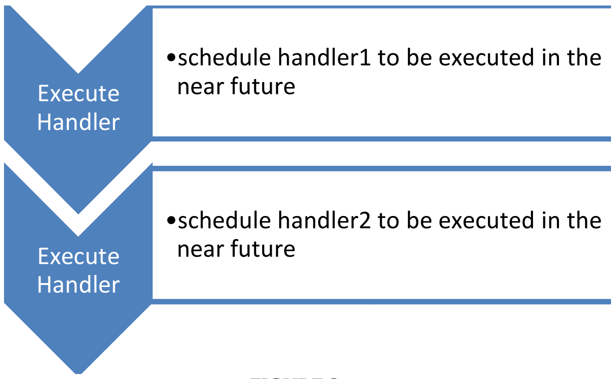

Execute Handler
•schedule handler1 to be executed in the near future
Execute Handler
•schedule handler2 to be executed in the near future

FIGURE 3
ASYNCHRONOUS EXECUTION OF EVENT HANDLERS

The event subsystem is heavily used by the rules subsystem and by the user interface. In the rules subsystem, each rule gets attached to a specific event and is evaluated only when that event is fired. In the user interface, different components of the user interface appear, change, and remove themselves based on events fired from the environment.

V. Rules

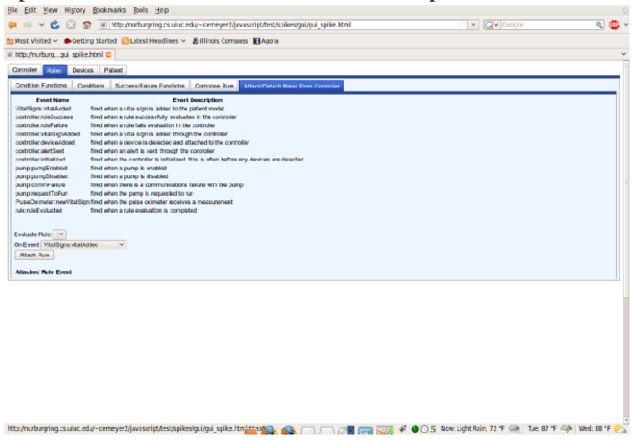
The last core component of the environment that was implemented was the rules framework. Rules govern how the system behaves when different events are fired. For example, when a user requests that an infusion pump connected to the system infuses liquid, an event is fired by the software model of the pump that notifies any attached handlers to that event that a pump is requesting to run. A rule can be attached as a handler to that event that then evaluates the conditions in the environment and proceeds to allow the pump to infuse if the conditions are met and disables the pump and notifies the clinical staff if the conditions are not acceptable.

Originally, we planned on having an iterative system of evaluating rules. Under this system, the controller would have kept a collection of all the rules in the system and constantly iterated through the collection, evaluating each one in sequence. Although this idea was simpler and worked effectively, it caused an excessive load on the system resources and it became impractical as the number of rules in the system grew.

VI. User Interface

The final component implemented was the user interface. We decided to make a simple and clean interface that exposed all functionality of the system, while accurately representing the state of the underlying environment. As a convenience, we used off the shelf widgets and modules from the Yahoo User Interface library.

Each of the main components of the system was divided into its own main tab, with some of the main tabs containing several children tabs below it to represent each of the sections of that component.


File Edit View History Bookmarks Tools Help
http://nurburgring.cs.uiuc.edu/~cemeyer2/javascript/test/spikes/gui/gui_spike.html
Most Visited Getting Started Latest Headlines Illinois Compass Agora
http://nurburg...gui_spike.html
Controller Rules Devices Patient
Condition Functions Conditions Success/Failure Functions Compose Rule Attach/Detach Rules From Controller

Event Name	Event Description
VitalSigns:vitalAdded	fired when a vital sign is added to the patient model
controller:ruleSuccess	fired when a rule successfully evaluates in the controller
controller:ruleFailure	fired when a rule fails evaluation in the controller
controller:vitalSignAdded	fired when a vital sign is added through the controller
controller:deviceAdded	fired when a device is detected and attached to the controller
controller:alertSent	fired when an alert is sent through the controller
controller:initialized	fired when the controller is initialized. this is often before any devices are detected
pump:pumpEnabled	fired when a pump is enabled
pump:pumpDisabled	fired when a pump is disabled
pump:commFailure	fired when there is a communications failure with the pump
pump:requestToRun	fired when the pump is requested to run
PulseOximeter:newVitalSign	fired when the pulse oximeter receives a measurement
rule:ruleEvaluated	fired when a rule evaluation is completed

Evaluate Rule:
On Event: VitalSigns:vitalAdded
Attach Rule

Attached Rule Event

FIGURE 4
TABBED USER INTERFACE

When the user interface is first loaded, it attaches several handlers to various events in the system that modifies the interface when the events are fired. For example, when a new device is detected by the controller, the controller fires an event and passes a reference to the software object model of the newly detected device. The user interface has a handler attached to that event, and when that handler is run, the interface examines the new device and dynamically creates a tab for that device that exposes that device’s functionality.

TESTING

To verify that our implemented system performed as expected, we tested it in a variety of ways. First, using the Jasmine behavior driven development testing framework we were able to unit test each of the components individually. Second, using the use cases that influenced our initial design of the system, we input the constraints for several of them and observed the system under the varying conditions that the use cases dictated to see if the behavior of the system corresponded to what the use cases prescribed the system should do.

I. Behavior Driven Development Testing

We utilized the Jasmine framework for testing each component of our system individually. The Jasmine framework is unique among JavaScript testing frameworks in that it does not require the Document Object Model provided by the browser to function properly. This feature makes our testing extremely portable between different JavaScript engines that may be providing differing execution environments.

Using Jasmine, we created a test suite for each of the main components of the system, which was composed of multiple specs that each tested a particular function of that component in a particular context. Each spec is composed of whatever initializations are necessary to get the component into the context we wish to test it in and several expects, which each compare if a particular property is in the state that we expect it to be in given the context.

II. Use Case Testing

To verify that all of the components worked together as a system correctly, we used the use cases that influenced our design to test the system. Each use case laid out all of the necessary conditions for the use case to take place, the rules about how the system should act when events take place and the input from the users of the system. We input all of the data from several of the use cases and were able to successfully demonstrate that our system acts as it should in varying situations.

CONCLUSIONS

During the course of the summer, we were able to successfully produce a system that accomplishes the goals that we set out with. From initial design to completion, we worked with the given use cases to create a dynamic system that illustrates that the overall concepts of the Medical Device Plug n Play project are in fact feasible. We leveraged all of the best features of JavaScript and worked around its shortcomings with unique solutions that solved the problems that we faced.

Each component was specifically designed to best allow it to operate under the constraints dictated by the use cases that we used throughout the project. We used leading edge technologies such as the Yahoo User Interface framework and the Jasmine testing framework along with accepted software design patterns to create a system that not only performs the tasks we set out to have it perform, but is flexible enough to be modified to support new features, devices, and input in the future.

REFERENCES

[1] https://agora.cs.illinois.edu/display/mdpnp/Home

[2] http://manual.widgets.yahoo.com/

[3] http://developer.palm.com/