Software Requirements Specification (SRS)

Automotive Onboard Diagnostic System

Authors: Reed Fielstra, Riti Adhi, Christopher Marsh, Louis Bodnar, Collin Lotus

Customer: Dr. Edward C. Nelson, Ford Motor Company, Innovation Center

Instructor: Dr. Betty H.C. Cheng

1 Introduction

This document outlines the software requirements for the Automotive Onboard Diagnostic System. It covers the overall description of the system, including general requirements, modeling requirements, associated diagrams, and a description of a prototype to be built to illustrate the system's functionality.

1.1 Purpose

This document is intended to inform those who have an interest in the Automotive Onboard Diagnostic System of its purpose and design. It should be useful to the customer as well as any members of a software development team who are tasked to build and/or maintain the system itself. Users or any other parties may find the documentation of the interactive online prototype useful.

1.2 Scope

The software system emphasized in this document is the Automotive Onboard Diagnostic System. The objective is to design an automotive diagnostic layer implementation to run on the Controller Area Network (CAN) bus of passenger vehicles. The application domain is an embedded system for an automotive system. The system will be able to detect and record any errors that devices report to the bus, and then send the proper diagnostic command messages to the instrument cluster and/or device. Service technicians will be able to diagnose malfunctioning equipment by connecting a laptop computer to the CAN bus to communicate with individual controllers. The laptop software will be able to initiate diagnostic self-tests and read from a log of error messages generated by the malfunctioning sensor. Although there are many devices operating on the CAN bus, our product will only focus on the onboard collision-deterrent radar and parking camera systems.

1.3 Definitions, acronyms, and abbreviations

This section consists of key definitions, acronyms, and abbreviations for this document.

Automotive Controllers – The processing unit in each module. Used to link sensors to the CAN bus.

Boot-Loader – An external program which can be used through a testing device that can handle startup functionality, limited diagnostics, and for the purpose of the diagnostic system, handles the process of upgrading the software in a module. The Boot-Loader is not considered part of the diagnostic communication system, as it is software that uses the system to communicate.

Camera System – The camera system is a module. The vehicle will make use of the camera system in order to assist with parking and backing up a vehicle. It is also referred to as the parking camera system.

CAN bus – Controller Area Network bus. The CAN bus is the standard communication network for automotive controllers. One module can communicate with another module through the CAN bus. It allows sensors to work together to improve their accuracy and functionality. It allows the instrument cluster to monitor the CAN bus for diagnostic messages and display sensor malfunctions using tell-tale lights and text display.

Checksum – A Checksum is used to verify that the message makes it to the correct destination and no information is lost. It is contained within the 6^{th} byte of each message.

CPU – Central Processing Unit. This is referred to as the Instrument Cluster.

CSMA – Carrier Sense Multiple Access. It is a protocol for communication over a shared medium.

DTC – Diagnostic Trouble Code. It is used to identify the fault type of an error and the time the error occurred.

ECU – Electronic Control Unit. This refers to an embedded system which controls one or more of the systems or subsystems in a vehicle.

HUD – Head's Up Display. The elements of the Instrument Cluster responsible for communicating with the driver. It includes the tell-tale lights and the text display located on the dashboard for alerting the driver to diagnostic issues.

Instrument Cluster – Also known as the Central Processing Unit (CPU) for this project, it is comprised of a set of gauges and tell-tales, including those of the HUD. It plays a significant role in the diagnostic layer. It is responsible for monitoring the CAN bus for diagnostic messages and interpreting those messages for the driver using tell-tale lights and the text display.

Module – A device that is attached to the CAN bus.

OBD System – Onboard Diagnostics System. The system is referred to as the vehicle's self-diagnostic and reporting capability. It is responsible for lighting up tell-tale lights to inform the driver that their car needs to be serviced by a service technician.

OBD2 – Onboard Diagnostics 2.

Power Cycle – When a system is turned off and then on again.

Radar System – The radar system is a module. The vehicle will make use of the radar system in order to recognize oncoming objects, also referred to as the onboard collision-deterrent radar system.

Sensor –A device used to monitor the status of a module as a method of error detection.

Tell-Tale light –When toggled, the light communicates a malfunction to the driver. The tell-tale light will stay on until the failure does not exist anymore.

1.4 Organization

This document is organized in seven sections with subsections describing various aspects of the Automotive Onboard Diagnostics System. Section 2 gives the overall description of the product, which includes the perspective, function, user characteristics, constraints, assumptions, dependencies, and future requirements. Section 3 lists specific requirements that were taken into account when developing the Onboard Diagnostics System. Section 4 contains models outlining the design and use of the system, which includes a use-case diagram, class diagram, sequence diagrams depicting scenarios of the system, and state diagrams. Section 5 describes the product prototype, Section 6 lists references, and Section 7 gives the point of contact for questions about the product.

2 Overall Description

This section gives an overall description of our product. It covers the product perspective, product functions, and characteristics of the expected user.

2.1 Product Perspective

As seen in Figure 2.1, the diagnostic layer is software that can run on automotive controllers. These controllers are the processing unit in each module and they are used to link sensors to the CAN bus. The diagnostic layer monitors the sensors for failures and reports them on the CAN bus.

A Block Diagram shows the layers of a system. The blocks at the bottom represent low level layers (i.e. physical layers) and the blocks at the top represent high level layers (i.e. software or user interface). Blocks at the same level represent different elements of the same layer (i.e. radar software and camera software).

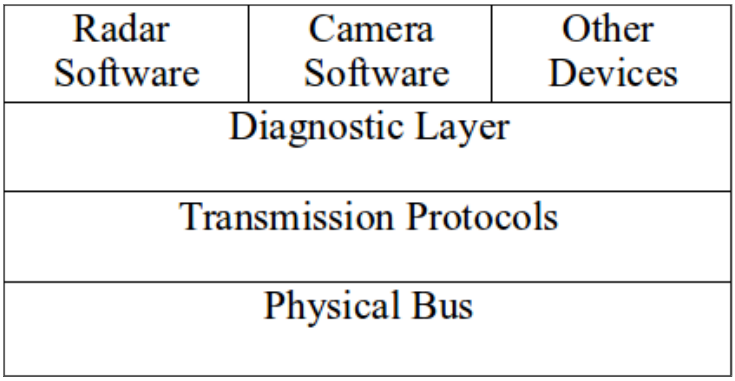

Figure 2.1: Block Diagram Showing the Diagnostic Layer in a Module

The diagnostic layer in each module has the ability to broadcast messages over the CAN bus. Other modules and external testing devices can read these messages. It communicates on this medium using a transmission protocol layer that is able to translate the internal data of the controller into bus-appropriate messages. The internal data of the module is gathered from sensors attached to the module controller. These sensors provide information on the state of their hardware and the data that they have gathered.

If a sensor malfunctions, the module broadcasts a message on the CAN bus alerting other devices of the malfunction. If the malfunction needs to be brought to the driver's attention, the instrument cluster either toggles a tell-tale light or displays a message on the text display. If the driver wants to get rid of the text display warning, then they may press a reset button located near the text display to clear the message. The driver is unable to hide a tell-tale light. The tell-tale light stays on until the malfunction is no longer detected.

When the driver takes the automobile into a repair shop, the service technician is able to connect a laptop to the CAN bus via an access port on the automobile. The service technician has the ability to query each module individually. If queried, a module performs a self check and logs any errors; it then sends the error log back to the service technician. The service technician also has the ability to update the software on any of the modules by sending a special message that puts the controller in software upgrade mode.

2.2 Product Functions

Send and receive diagnostic messages on the CAN bus - The CAN bus is where the module communicates with other devices. Modules have the ability to both send and receive messages on this bus.
Log failures - In the event of a sensor failure, the module is able to internally record the type of failure, time stamp of the first failure of that type, time stamp of the most recent failure of that type, and total number of failures of that type. It uses persistent memory to retain the error log between power cycles.
Emergency shutdown mode - When a critical failure occurs, the module's controller is able to shut down individual sensors without taking the entire module off-line.
Software upgrade mode - Service technicians are able to put the module in software upgrade mode, allowing it to accept a new firmware.

Broadcast failures - When a sensor has a failure, its controller broadcasts the failure on the CAN bus to alert other modules that there is a malfunction.
Self-check - Each module is able to run a self-check routine to detect problems with the sensors and controller unit.
Individual module query - An external testing device is able to individually query each module on the CAN bus. The queried module is able to provide the external testing device with error logs.
Tell-tale lights and text display - The diagnostic layer is able to communicate with the driver by toggling tell-tale lights and displaying messages on the text display.

2.3 User Characteristics

The driver is expected to have basic training in vehicular operation. They have enough knowledge to know that tell-tale lights indicate a problem and know that the automobile should be taken to a repair shop. They are able to understand common text display messages and be able to tell when a text display warning warrants a trip to the repair shop. It is expected that the driver has no knowledge of the inner workings of the automobile.

The service technician is expected to know how to connect an external testing device to the CAN bus and use it to diagnose problems. They are expected to understand messages returned by the sensors and know how to handle the problem (i.e. know which sensor failed, replace or repair the sensor).

2.4 Constraints

There are a number of factors that will limit development options. Any errors that are detected by a module must be logged internally by the module where the error has occurred. Modules receive diagnostic messages from an external testing device and will transmit its error log to the external testing device for audit by a service technician. There will be no network handshaking protocol in message transmission. Modules will not establish direct connections with each other when communicating. Messages will be assumed to have been received correctly. Messages will be broadcast by a module over the CAN bus. Messages will adhere to the CAN specifications. Each message will be eight bytes in size with the first byte consisting of a CAN ID. The CAN ID indicates which module the message originated from. Modules will assume that messages have been delivered successfully unless it receives a retransmission request message, in which case the original message will be re-broadcasted.

The majority of the software's functions will be used when the vehicle is off-road. The software will be used primarily by service technicians to help with repairs, audits, and possibly claims requests by drivers. This software will not interfere with drivers while they are on the road. It will only notify them of an error that has occurred. The customer has not defined any security concerns with the diagnostic system.

2.5 Assumptions and dependencies

There are some assumptions in the software, hardware, and about users of this system that, if changed, can affect requirements.

Hardware and Software:

It is assumed that the communication network uses the CAN bus architecture. It is also assumed that each individual module that is plugged into the CAN bus has its own diagnostics software. Each module has a different set of potential errors to diagnose, but they read and compose messages in a standard format that will be transmitted over the bus.

Users:

It is assumed that the service technician will only interact with the software in two ways. First will be by sending diagnostic messages to modules plugged into the CAN bus to transmit their error logs to an external testing device. The second way will be to make software updates. It is also assumed that drivers will only interact with the system by viewing lights and messages on the Heads Up Display (HUD). Drivers can remove messages from the HUD by pressing a reset button.

2.6 Apportioning of requirements

The system may need to be extended to support additional diagnostic capabilities, as required by service technicians. Currently, the system supports diagnostic commands for a particular module or all modules to transmit their error logs to an external testing unit. While security is a concern, the ability to address security issues through a diagnostic layer implementation is severely hampered by the On Board Diagnostics II, which is the standard specifications for automotive on board diagnostics, and CAN bus specifications. As weaknesses in CAN security become apparent, those risks will have to be mitigated and the system may change to reflect that.

3 Specific Requirements

The following is a list of requirements gathered for the Automotive Onboard Diagnostic System. These requirements have been split into communication, module capability, and error logging categories.

1. Communication

1.1: Data will be transmitted on a central CAN bus.

1.2: Messages will be read by a service technician, CPU, or other device connected to the bus.

1.3: Specially formatted CAN bus messages will be put on the bus.

1.4: Radar will use identification values of: Receive: 0x250, Transmit: 0x260

1.5: Camera will use identification values of: Receive:0x350, Transmit:0x360

1.6: Checksums are contained within the 6th byte of each message.

1.7: If received checksum does not match calculated checksum, ignore the received data, and request a re-transmission of the message if it is not one that is periodically sent.

1.8: The Instrument Cluster (also known as the CPU for this project) is required to interpret diagnostic error broadcasts and determine which errors need to be displayed to the driver.

1.9: If an error is to be reported to the driver, the tell-tale light (such as Adaptive Cruise Control or Lane Assist) that is associated with the malfunctioning module is lit up. A message indicating the failure is displayed on the text display of the HUD.

1.10: The driver may clear the messages on the message center by pushing a reset button. It will not affect the tell-tale lights, which may only be turned off by a technician or as a result of the problem no longer being detected.

2. Module Capability

2.1: Every module has manufacturer installed diagnostic capability to detect its own errors.

2.2: The self-diagnostic check (see 2.1) is run at the ignition sequence of the automobile for every module. If the check detects an error, it will act accordingly (see 2.5).

2.3: All modules run diagnostic checks at periodic intervals relative to the time required by other modules or actors to react in a safe manner.

2.4: When a module has run its initial self-test, it broadcasts to the CAN that it is ready to receive messages.

2.5: Modules that detect any errors will log the errors in their internal error logs, and broadcast a diagnostic message with the associated code onto the CAN bus.

2.6: If a module needs to turn off its main functionality, it still listens for messages on the CAN bus. For example, the driver may not receive a signal from the camera, but the device is still listening to the CAN bus.

2.7: In extreme cases, a module that is disconnected from the CAN bus is guaranteed to never respond to CAN messages or queries.

2.8: Modules can be put into a software upgrade mode, triggered by a diagnostic message. This will then allow the boot-loader, or other external software, to run.

3. Error Logging

3.1: All errors will be logged internally on the module it was detected on.

3.2: Logged errors will include the associated Diagnostic Trouble Code (DTC) or fault type, and timestamp.

3.3: Internally logged errors must be able to be read by a service technician.

3.4: Errors may occur more than once - when an error occurs, an internal flag must be set so the error is not logged again.

3.5: If an error resolves itself then reoccurs at a later time, a new error log is generated at each event, along with all other associated diagnostic procedures.

3.6: The error log will be a circular buffer for each device, able to hold 24-100 messages. Newer unique error log entries will push the oldest off.

4 Modeling Requirements

This section covers the modeling requirements, which includes use case diagram and documentation, domain/class diagram and data dictionary, sequence diagrams and documentation, and state charts and documentation.

4.1 Use Case Diagram and Documentation

A use case diagram (Figure 4.1) is used to model how different entities act with the whole system. The use case diagram consists of a system boundary, actors, and use cases. The large box represents the system boundary. The system boundary is the Diagnostic Communication System. Stick figures outside the box portray actors. The actors are the module, testing device, and CPU/Instrument Cluster. Each oval inside the box is a use case that represents some major required functionality and its variants. A line between an actor and use case indicates that the actor participates in the use case. All use cases for this system are documented below Figure 4.1 in Table 1.

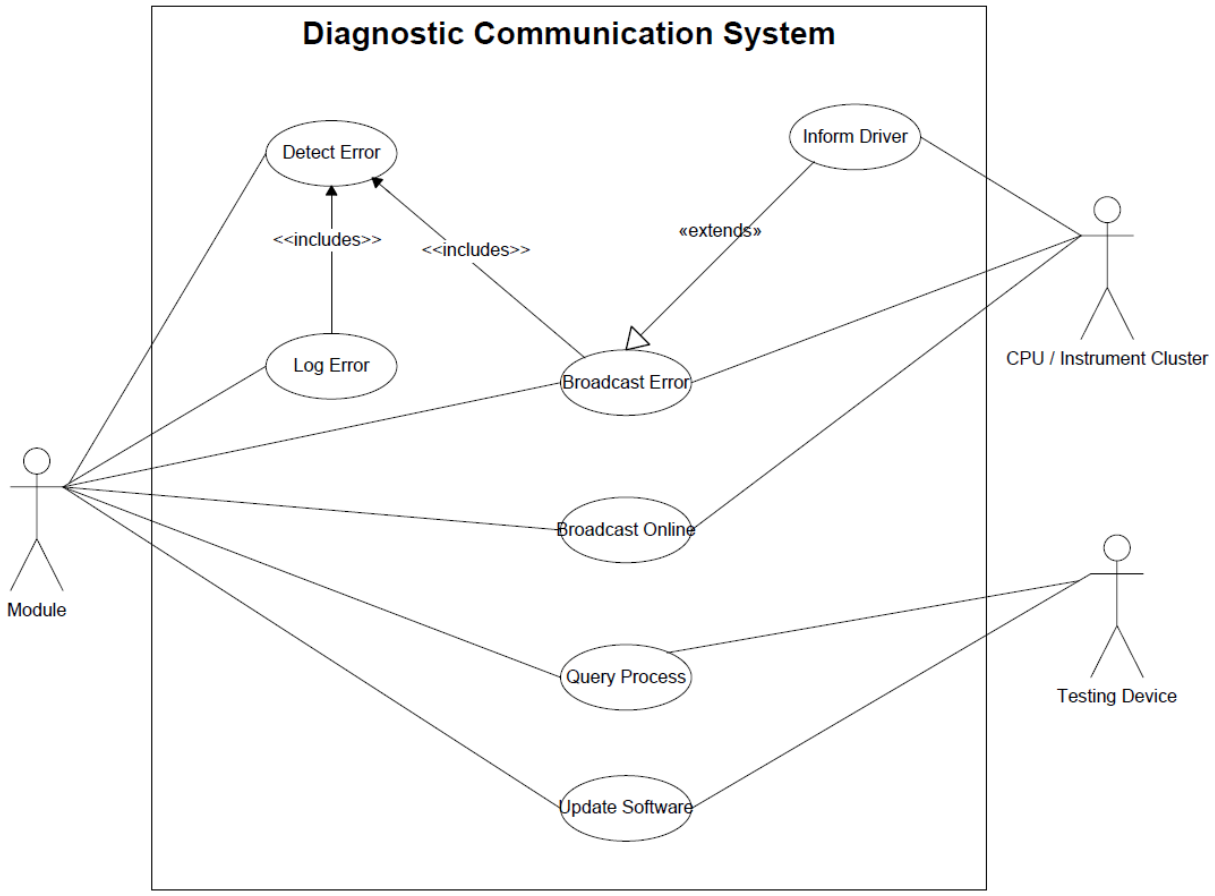

Figure 4.1: Use case diagram of the Automotive Onboard Diagnostic System.

Use Case:	Broadcast Error
Actors:	Module, CPU / Instrument Cluster
Description:	After an error is discovered, the error will be broadcast on the CAN bus so all devices that rely on that particular device may act accordingly. It will encode the message in the proper diagnostic format, which will include the specific CAN ID tag, and a checksum, along with additional diagnostic information.
Type:	Primary
Includes:	Detect Error
Extends:	
Cross-refs:	1.1-1.6
Dependent Use Cases:	

Use Case:	Broadcast Online
Actors:	Module, CPU / Instrument Cluster
Description:	As part of standard procedure for all devices, a power-on self-test is performed on every device when the car is started. If a device has completed a test without any errors detected, it broadcasts a specific message that informs all other devices on the BUS that it is ready to receive messages and perform its functionality.
Type:	Primary
Includes:	
Extends:	
Cross-refs:	2.1, 2.2, 2.4
Dependent Use Cases:	

Use Case:	Detect Error
Actors:	Module
Description:	On startup, as well as periodically, modules run self-diagnostic checks. The functionality for detecting errors is assumed to be provided by the manufacturer.
Type:	Primary
Includes:	
Extends:	
Cross-refs:	2.1-2.3
Dependent Use Cases:	Log Error, Broadcast Error

Use Case:	Inform Driver
Actors:	CPU / Instrument Cluster
Description:	After receiving an error message broadcast on the CAN bus, the instrument cluster determines in what way to inform the driver, which can vary between toggling tell-tale lights and/or displaying a text message in the message center.
Type:	Primary
Includes:	
Extends:	Broadcast Error
Cross-refs:	1.8-1.10
Dependent Use Cases:	

Use Case:	Log Error
Actors:	Module
Description:	After a device recognizes an error, it registers a Diagnostic Trouble Code (DTC) which will be stored on the ECU of the associated module for later diagnostic use. The time the error occurred will also be stored. When an error occurs, a flag that the error has been logged will be set, preventing similar errors from filling the error buffer, excepting errors that resolve themselves then reappear (intermittent errors).
Type:	Primary
Includes:	Detect Error
Extends:	
Cross-refs:	3.1-3.2, 3.4-3.6
Dependent Use Cases:	

Use Case:	Query Process
Actors:	Testing Device, Module
Description:	An external testing device is attached to the CAN bus, and sends a message to a device to ascertain what DTCs were stored on it. The device then sends a reply with the codes to the testing device for diagnosis by a technician.
Type:	Primary
Includes:	
Extends:	
Cross-refs:	1.10, 3.3
Dependent Use Cases:	

Use Case:	Update Software
Actors:	Testing Device, Module
Description:	An external testing device is attached to the CAN bus, and a message is sent to a module to set a “Re-flash” mode, at which point the loader software on the external testing device takes over.
Type:	Secondary
Includes:	
Extends:	
Cross-refs:	2.8
Dependent Use Cases:	

Table 1: Use case documentation for the Automotive Onboard Diagnostic System

4.2 Domain/Class Diagram and Data Dictionary

A class diagram (Figure 4.2) is used to model how classes in the system interact with one another. A class has a name, a set of attributes, which are simple data variables whose values can vary over time and among different entities of the class, and a set of operations on the class’s attributes. Each class has attributes and operations it can use to interact with other classes. A line between two classes is called an association, indicating a relationship between the classes’ entities. Associations can have labels, role names and/or multiplicities that describe the relationship between entities. Role names specify the context of an entity with respect to a particular association. Multiplicities specify constraints on the number of entities and the number of links between associated entities. Classes for this system are recorded in a data dictionary below Figure 4.2 in Table 2.

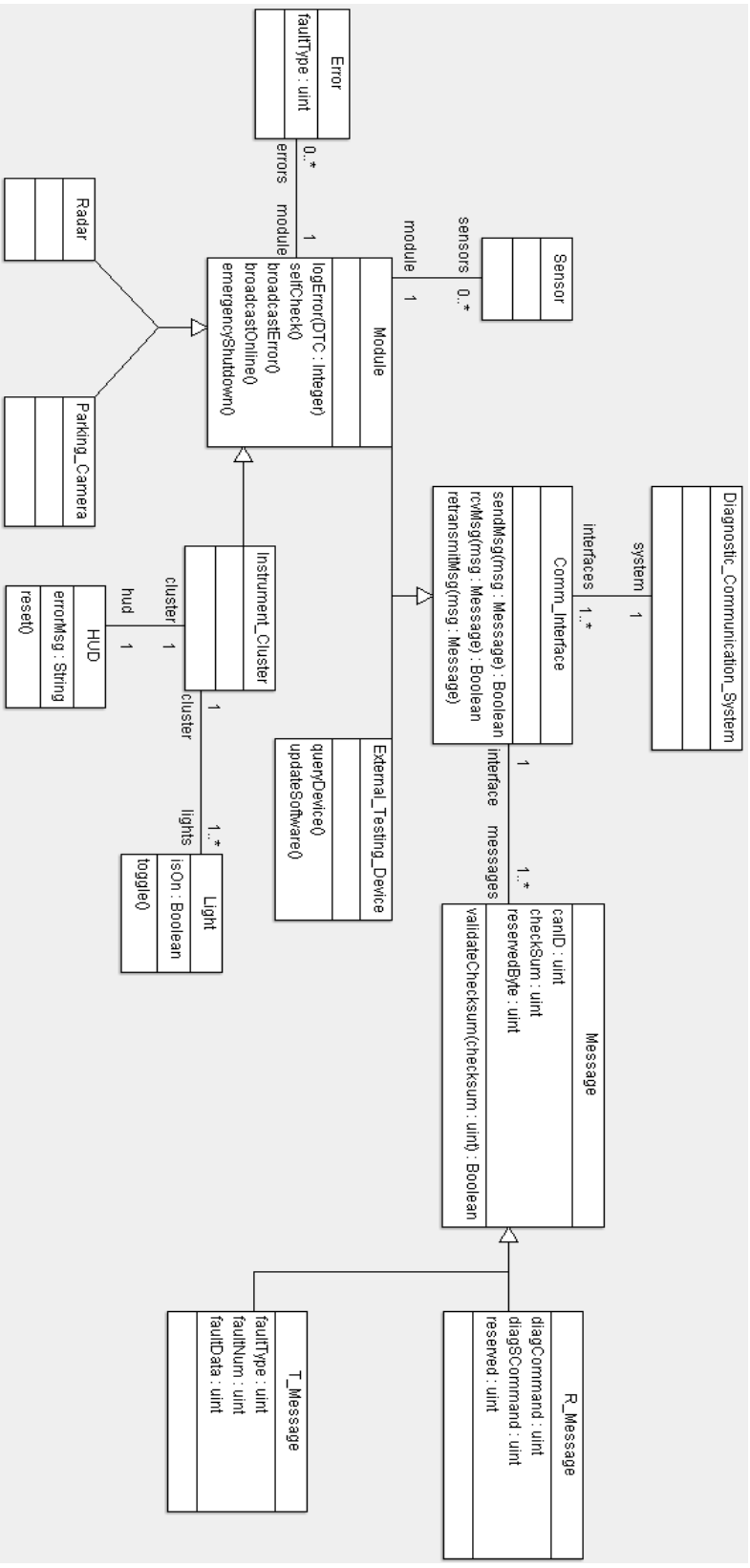

Figure 4.2: Class diagram for the Automotive Onboard Diagnostic System

Comm_Interface	Abstract class which will provide the send and receive capability of messages to all the devices on the CAN bus.	
Attributes		
Operations		
	`rcvMsg` (msg:Message): boolean	Allows the interface to receive a message. Message "msg" is the message being received.
	`retransmitMsg` (msg:Message): void	Retransmits a message if the original is corrupt in some way. Message "msg" is the message being retransmitted.
	`sendMsg` (msg:Message): boolean	Allows the interface to send a message through the CAN bus. Message "msg" is the message being sent.
Relationships	All devices, including the external testing device, CPU, and any others that are attached to the bus that require the ability to send or receive diagnostic messages will use the inherited functions from this class. Specific implementations of the send/receive functions may vary in the subclasses.	
UML Extensions	N/A	

Diagnostic_Communication_ System	Abstraction of the communication system that will handle the transfer of diagnostic messages from one entity to another.	
Attributes		
Operations		
Relationships	Many messages will be sent over the CAN bus within the system. The system will run on the bus, and facilitate communication between many devices that share the same bus.	
UML Extensions	N/A	

Error	A defined error in a module	
Attributes		
	`faultType`: unsigned integer	The type of error that has occurred.
Operations		
Relationships	A module can have many possible errors with specific error codes. Once an error occurs, the appropriate light must illuminate in the instrument cluster and a readable message must be displayed on the HUD.	
UML Extensions	N/A	

External_Testing_Device	Any device used by a trained technician that is connected to the CAN bus and used to diagnose problems. It sends queries or other diagnostic commands to devices internal to the automobile and also connected to the CAN bus.	
Attributes		
Operations		
	`queryDevice` (): void	Sends a query to a module to receive information about any errors.
	`updateSoftware` (): void	Allows the software of the testing device to be updated at any time.
Relationships	Utilizes the comm_interface to be able to send and receive messages to internal devices on the bus.	
UML Extensions	N/A	

HUD	The “Head’s Up Display”, or interface elements responsible for alerting the driver to diagnostic issues. Traditionally a Malfunction Indicator Light, but becoming much more sophisticated in recent automobile generations.	
Attributes		
	`errorMsg`: string	The error message that will be displayed in the message center.
Operations		
	`reset` (): void	Resets the display on the message center.
Relationships	The HUD is controlled by the CPU.	
UML Extensions	N/A	

Instrument_Cluster	The vehicle's central processing unit; in charge of coordinating other devices and the interface/dashboard elements with the driver.	
Attributes		
Operations		
Relationships	Utilizes the comm_interface to be able to send and receive messages to internal devices on the bus.	
UML Extensions	N/A	

Light	A light in the instrument cluster of the vehicle	
Attributes		
	`isOn`: boolean	An indicator to determine whether the light is on or off.
Operations		
	`toggle` (): void	Changes the value of isOn from true to false or false to true.
Relationships	The instrument cluster has multiple lights to indicate which errors have occurred.	
UML Extensions	N/A	

Message	A message formatted specifically to run over the CAN bus that will be heard by all devices on the bus. It will only by read and decoded by the device matching the receiving ID.	
Attributes		
	`canID`: unsigned integer	The ID of the CAN bus. Used to verify the authenticity of messages.
	`checkSum`: unsigned integer	Used to verify that the message made it to the correct destination and no information was lost.
	`reservedByte`: unsigned integer	The reserved byte that is found in both types of messages (default to 0x00).
Operations		
	`validateChecksum` (checksum: unsigned integer): boolean	Validates the checksum to ensure that the message made it to the correct destination. Unsigned integer "checksum" is the checksum the system is trying to validate.
Relationships	The Diagnostic Communication System will manage the transfer of messages between devices.	
UML Extensions	N/A	

Module	A class describing any general purpose device hooked into the CAN bus. Modules provide a general interface for logging or diagnosing errors.	
Attributes		
Operations		
	`broadcastError` (): void	Sends a message over the CAN bus to the CPU to notify that an error has occurred.
	`broadcastOnline` (): void	Sends a message over the CAN bus to the CPU confirming that the module is online and running correctly.
	`emergencyShutdown` (): void	Sends the module into emergency shutdown mode in case a fatal error occurs.
	`logError` (DTC: integer): void	Logs an error within the module. Integer "DTC" is the diagnostic trouble code, which is used in identifying the fault type.
	`selfCheck` (): void	The module will check to make sure that all components are operating correctly; Occurs every few milliseconds.
Relationships	Receives capability to send and receive messages from the comm_interface, and is superclass to the Parking_Camera and Radar – along with any other device not mentioned.	
UML Extensions	N/A	

Parking_Camera	On-board rear-view camera used to assist drivers in accurately parking and avoiding any objects that may be in the automobile's blind spot.	
Attributes		
Operations		
Relationships	The camera is a device connected to the bus, and has the capability to send and receive diagnostic messages.	
UML Extensions	N/A	

R_Message	A message that is received	
Attributes		
	`diagCommand`: unsigned integer	The diagnostic command in received messages.
	`diagSCommand`: unsigned integer	The diagnostic sub-command in received messages.
	`reservedByte`: unsigned integer	The reserved byte in received messages (default to 0x00).
Operations		
Relationships	Derived from the Message class. Modules contain their own messages for logging errors. Messages are then sent through the CAN bus to other modules.	
UML Extensions	N/A	

Radar	On-board radar used in newer automobiles in adaptive cruise control, object detection, or collision avoidance.	
Attributes		
Operations		
Relationships	The radar is a device connected to the bus, and has the capability to send and receive diagnostic messages.	
UML Extensions	N/A	

Sensor	The sensor used to monitor the status of a module	
Attributes		
Operations		
Relationships	Modules have multiple sensors that monitor the status of the module and are primarily used in error detection.	
UML Extensions	N/A	

T_Message	A message that is transmitted	
Attributes		
	`faultData`: unsigned integer	Fault data sent out in transmitted messages.
	`faultNum`: unsigned integer	The fault number of the error that occurred.
	`faultType`: unsigned integer	The fault type of the error that occurred.
Operations		
Relationships	Derived from the Message class. Modules contain their own messages for logging errors. Messages are then sent through the CAN bus to other modules.	
UML Extensions	N/A	

Table 2: Data dictionary for the automotive onboard diagnostic system

4.3 Sequence Diagrams and Documentation

Sequence diagrams depict specific scenarios of the system. The boxes at the top represent objects in the system. The vertical lines represent object lifelines. Time is the y-axis. The solid arrows represent the invocations of the object methods, while the dotted arrows signify a return value.

In Figure 4.3, a service technician uses an external testing device to query a module. During transmission, the message becomes corrupted. When the module receives the message, it attempts to validate the checksum, but the calculated checksum does not match the message checksum. The module then sends a retransmit message to the external testing device. The external testing device resends the query message and the module successfully validates the checksum. The module performs a self-check and sends the results back to the external testing device.

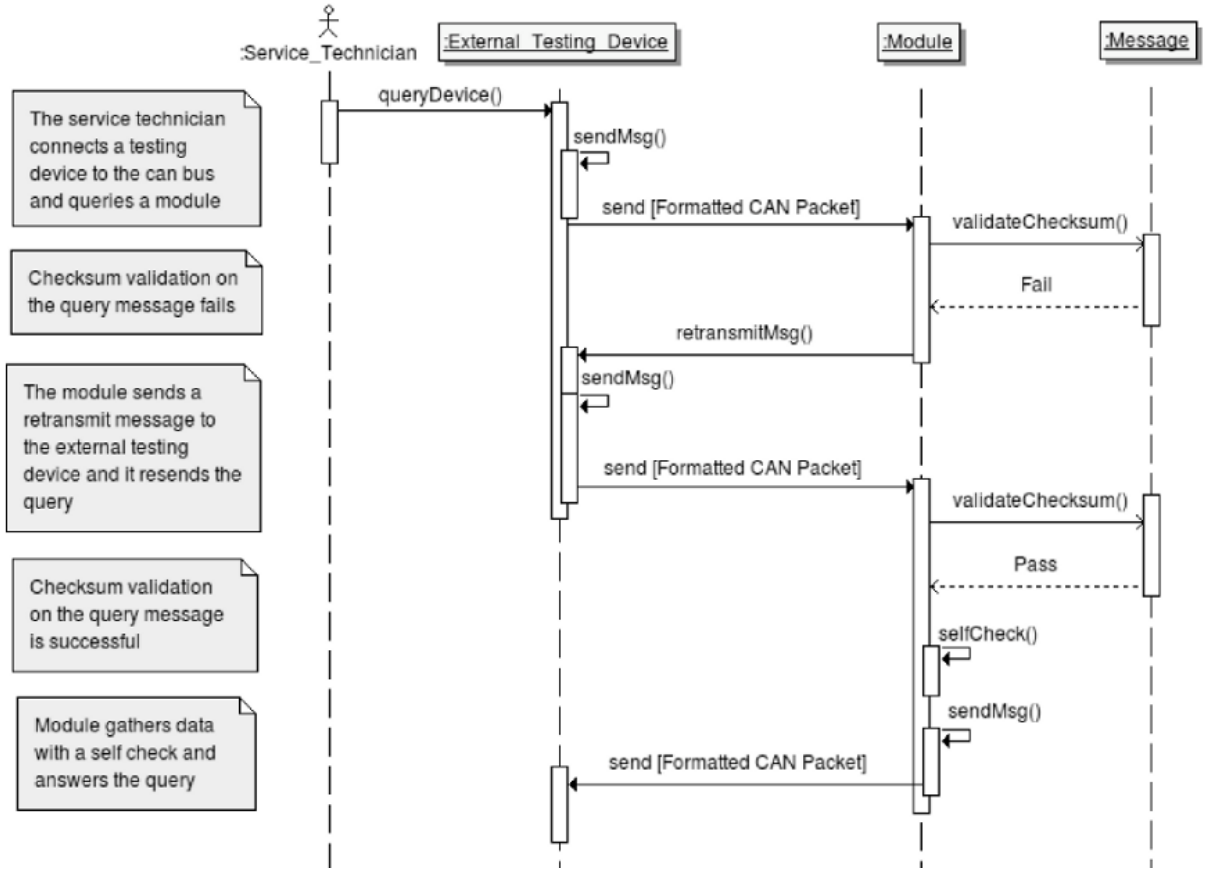

Figure 4.3: Sequence Diagram 1: Data from a query becomes corrupted and must be re-sent.

The sequence diagram in Figure 4.4 illustrates a scenario in which the camera system detects an error with its image processing unit and needs to go into an emergency shutdown mode. The camera system's self-check mechanism detects an error with its image processing unit. The camera system logs an internal error for later reading by a service technician. The camera system notifies the instrument cluster of the error by calling the sendMsg function. The instrument cluster receives the message using the rcvMsg function. The camera system goes into emergency shutdown mode. The service technician reads the error by querying the device.

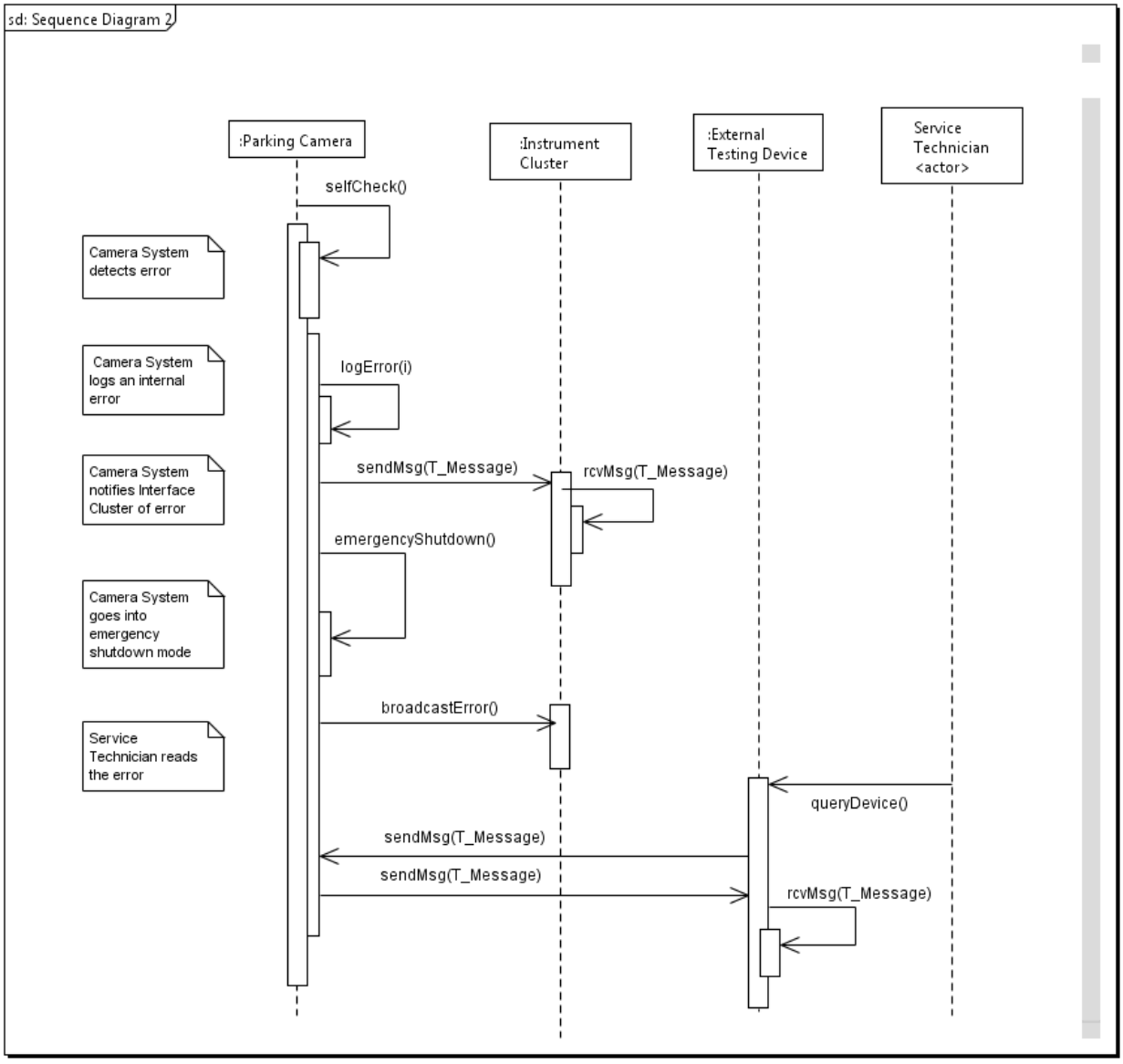

Figure 4.4: Sequence Diagram 2: Camera System Detecting an Error

In Figure 4.5, a mechanism in the radar has malfunctioned and has gone out of specification. The radar module must notify both the driver and the vehicle's instrument cluster that it must go offline. The radar module calls the 'broadcastError' function internally which in turn calls the SendMessage function using the newly created 'message' as a parameter. The message is broadcast over the CAN bus as a packet and is received by the instrument cluster and the HUD using the rcvMessage function. Once the message has been sent by the radar module, it is destroyed.

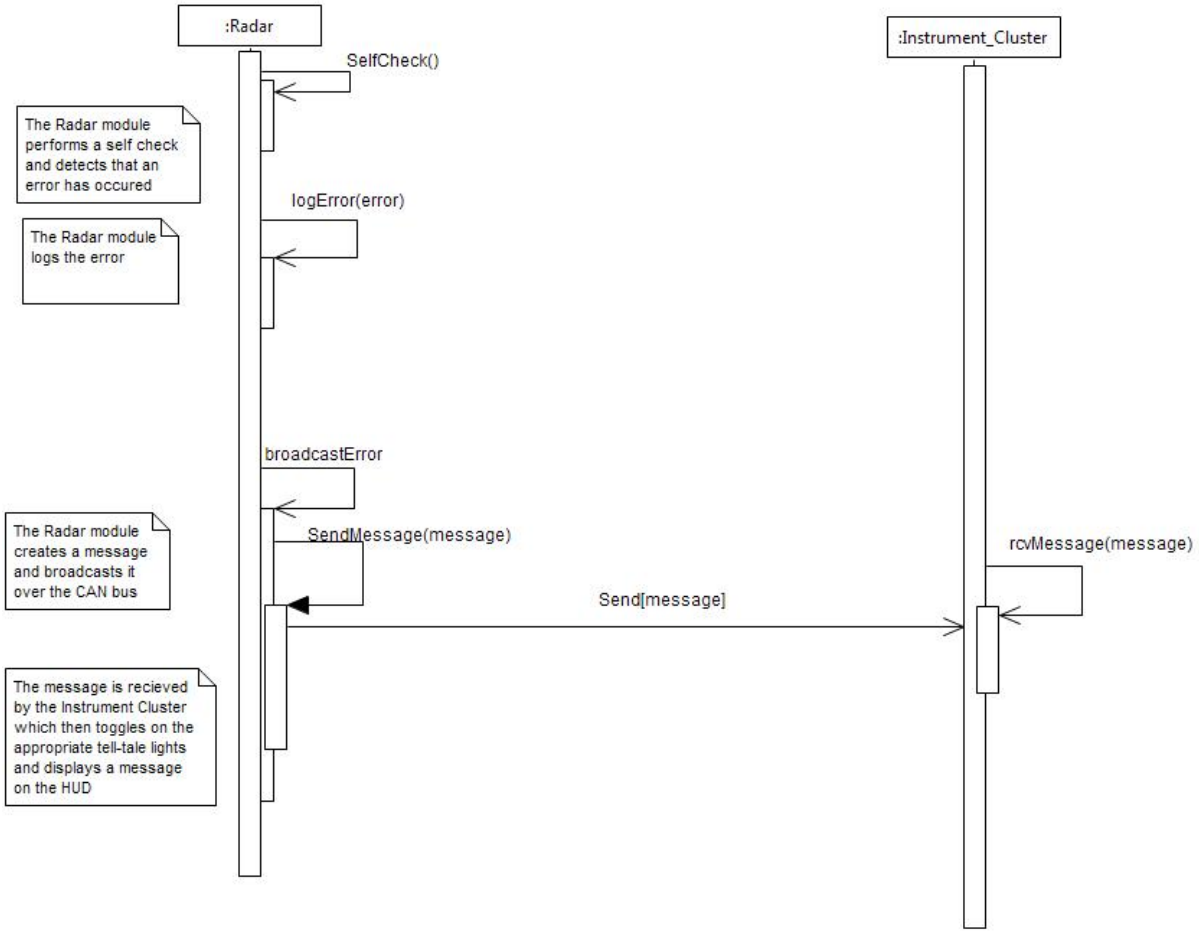

Figure 4.5: Sequence Diagram 3: The radar has malfunctioned and must notify the Instrument Cluster and the Driver

4.4 State Chart Diagrams and Documentation

Statecharts illustrate the operational behavior of a system as a collection of states and the conditions that occur in order to move the system from one state to another. Each state is an abstraction of the system as a whole. The current values of all of its variables, conditions and methods are captured into each box in the statechart. Transitions are the arrows that indicate the flow of control from one state to another. Each transition starts with the event that causes the transition. It can then be followed by a slash and an action, or list of actions that will occur with that transition. Additionally, bracketed elements in a transition indicate conditions that must be true in order for that transition to occur. Statecharts are used mainly to analyze and explain the behavior of a class or object (as depicted in the class diagram) as the system operates, or to detect any possible aberrant situations (such as the ability to enter a state but not exit, or ambiguous transition logic).

Figure 4.6 shows the behavior expected of any module (which includes the radar or parking camera). The states relative to communication have been emphasized (as these states occur quickly), indicating the state of the module during the actual process of broadcasting or listening to messages from the CAN bus.

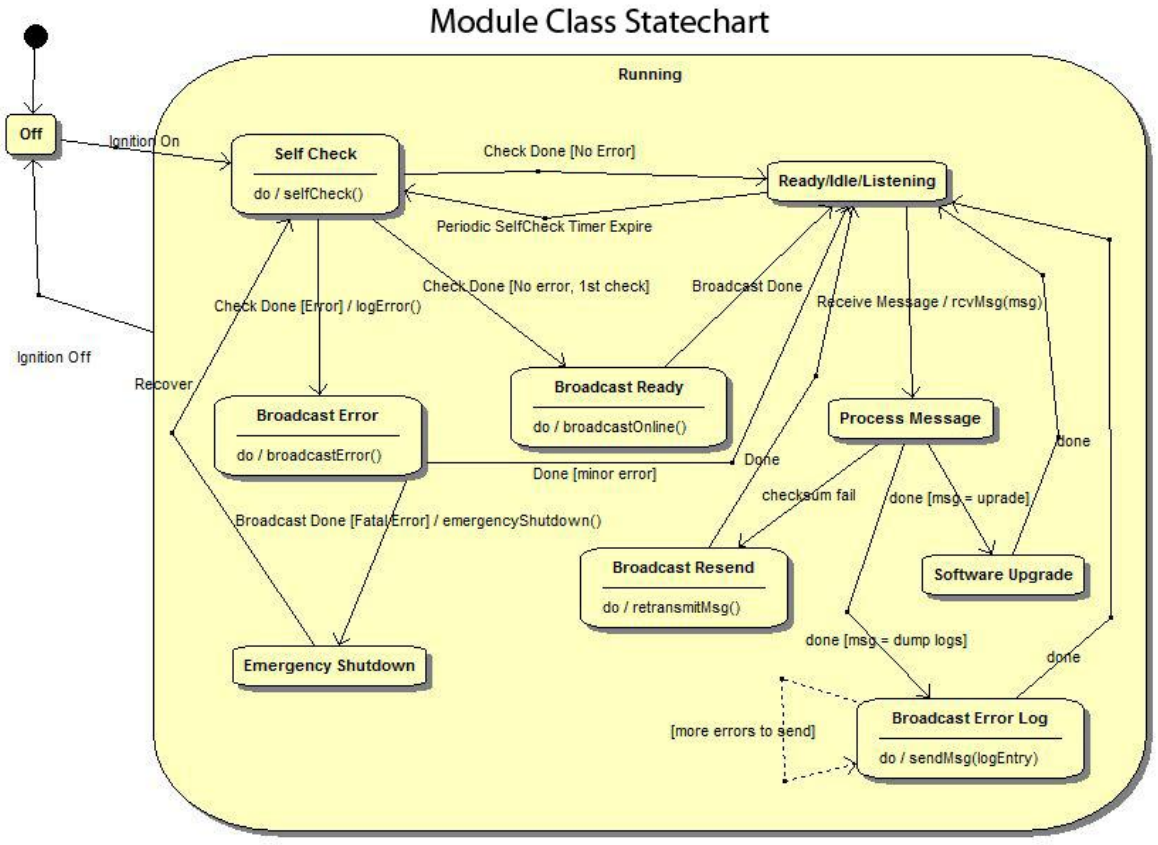

Figure 4.6: The Module Class State Chart describes the typical behavior of a Module as an abstraction of states and events.

Figure 4.7 describes the state-based behavior of the diagnostic layer within the instrument cluster. While many of its states and transitions are the same for modules, extended capability and assurance requirements prevent it from going offline, and add additional conditional states for processing messages from the CAN bus.

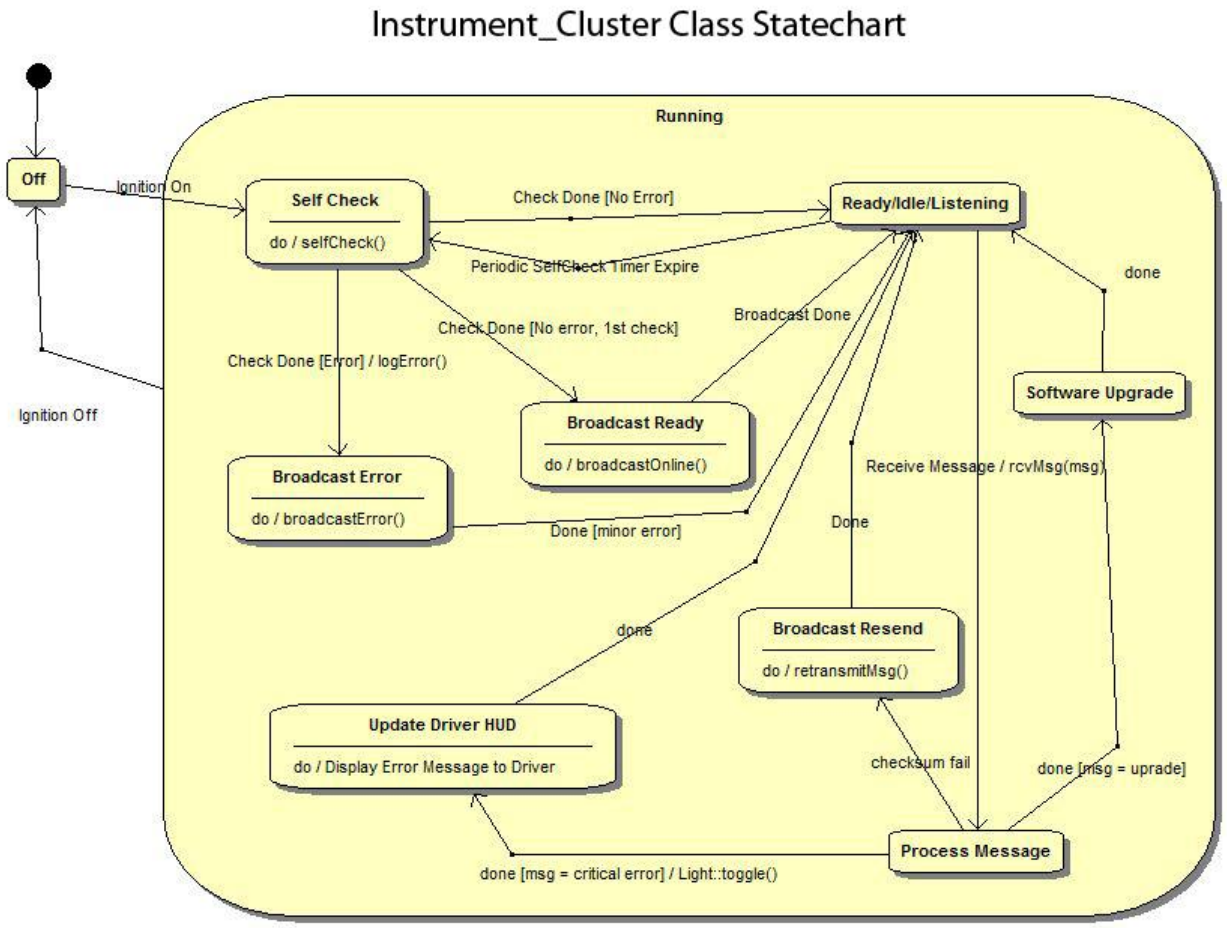

Figure 4.7: The Instrument_Cluster Class State Chart describes the behavior of the instrument cluster as an abstraction of states and events. Its behavior is similar to a more general module.

5 Prototype

The prototype demonstrates the flow of diagnostic messages between modules in the vehicle and an external testing device, all connected to the CAN bus. Based on events that could occur during the typical operation of a vehicle, the prototype will simulate the flow of diagnostic messages. Additionally, all interfaces with the driver of the vehicle will also be displayed and will react properly. Certain diagnostic commands can be entered into the prototype as well to simulate the interaction with an external testing device. Certain capabilities of the CAN bus specification itself will also be demonstrated, such as: collision resolution, ensuring message data integrity, and a simplified version of the Carrier Sense Multiple Access (CSMA) protocol for communication over a shared medium.

5.1 How to Run Prototype

The prototype has been created as a Java Applet that is embedded on the D-Cubed project website, using Java 1.4.2. The prototype is viewable with most any modern web browser. Web browsers developed prior to 3-10-2003 may not support Java applets. Any equal or higher version of the Java platform is required to view the applet. If the java platform is not present or is not the correct version, a message will be displayed on the web page in lieu of the applet.

To obtain the latest version of Java, Sun Microsystems has extensive support:
Automatic version and platform detection: http://java.com/en/download/installed.jsp
Manual download and installation: http://java.com/en/download/manual.jsp

Two versions of the prototype exist, version '1' and version '2'.

Version 1 refers to the very first conceptual draft for the user interface of the prototype; a development milestone used for design feedback. It has no functionality, and shall not undergo any further development.

Version 2 refers to the current and incremental builds of the prototype that will be periodically updated as design of the diagnostic system progresses. At such a time as the developers choose (Nov. 21) the prototype will be considered 'complete', and will demonstrate the capability of the diagnostic system as a whole to the best of its and its developer’s ability. Until the final release date of Nov. 21, Version 2 may be accessible from the developer's website, but is not representative of the final version of the product - functionality may be limited or incomplete, bugs may be present, and capability may change.

The current build of the Version 2 of the Prototype can be seen at:
http://www.cse.msu.edu/~435diag3/proto2.html

The first draft of the Prototype, Version 1, can be seen online at:
http://www.cse.msu.edu/~435diag3/proto1.html

5.2 Sample Scenarios

Every scenario involving the Diagnostic Communication System revolves around the sending and receiving of diagnostic messages over the CAN bus. Illustrating the simple scenario of an issue with the Radar module will demonstrate the general usage of the Prototype, which can then be applied to all other scenarios.

At any time, mousing over elements in the Prototype interface will display a pop-up tool tip detailing specifics of what that control does.

Figure 5.1 shows and describes the basic elements of the interface necessary to trigger an event and monitor the diagnostic response.

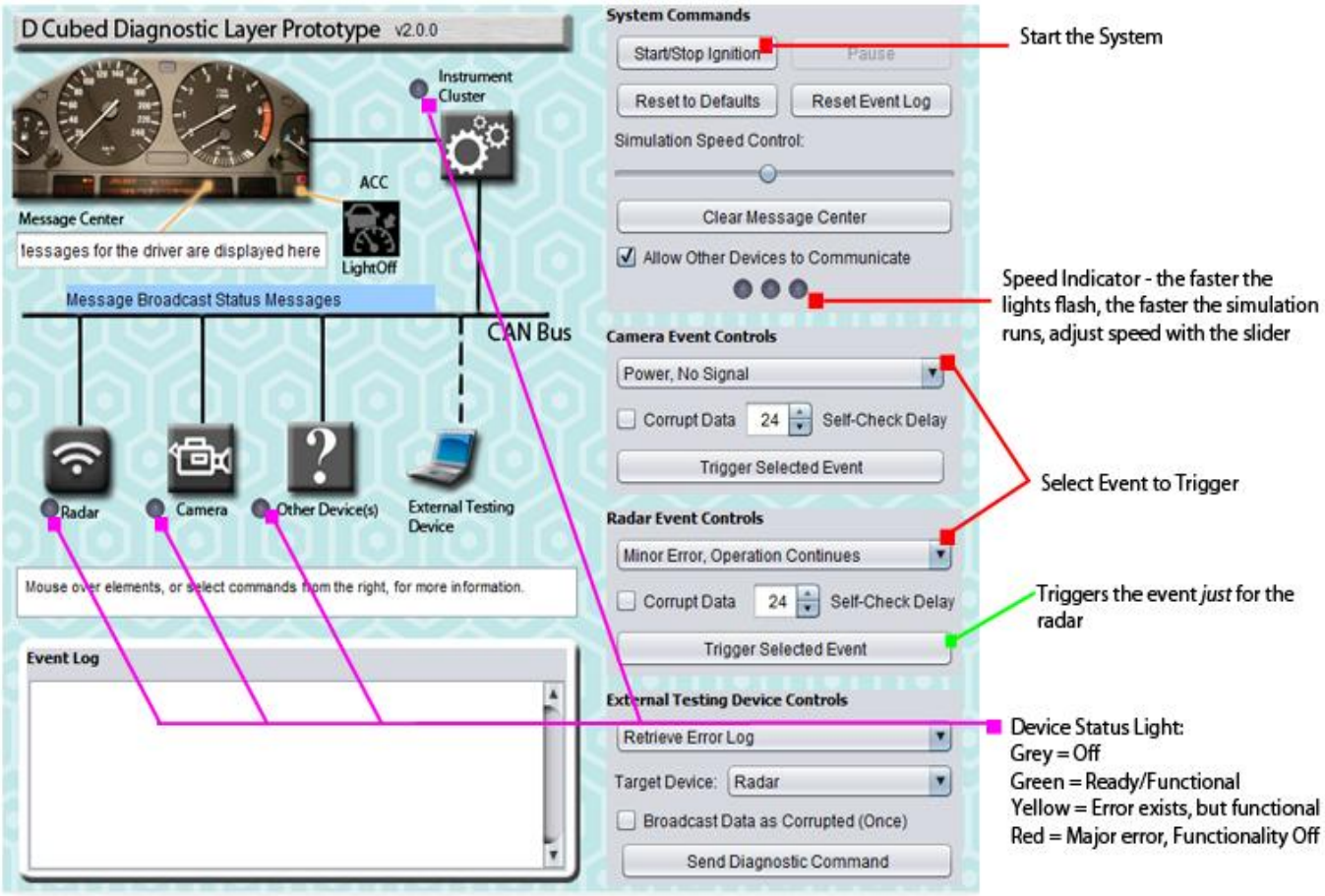

Figure 5.1: Visual outline of the basic, key components to the Prototype's interface.

The first step is to press the "Start/Stop Ignition" button under System Controls on the right-hand panel. This is the equivalent of turning on the ignition in the vehicle. The simulation will start, and a number of different icons and events will appear as each module begins its own initial self-check and broadcasts that it is ready to handle any new messages. Figure 5.2 shows the Radar device after it has performed its self-check and no errors have been detected. The blue icon

superimposed on the radar icon indicates its current state (broadcasting), and a summary of the broadcasted message is shown over the CAN bus graphic. The broadcast registers an event which is shown in the log, along with the specifics of the message. From the event log, it is evident that the Camera already broadcasted that it is ready as well. The green lights that have become lit next to the module icons indicate the status of the physical device: green for no errors, yellow for functional but minor errors, and red for no functionality (but can still communicate). The packets, represented by tiny teal squares, will appear as an animation in the prototype. They will visually indicate which module is sending to the BUS, and if any device is processing any of the broadcasted messages. If a collision occurs on the CAN bus, it will be similarly noted over the CAN bus graphic, as well as logged with an event as to its resolution.

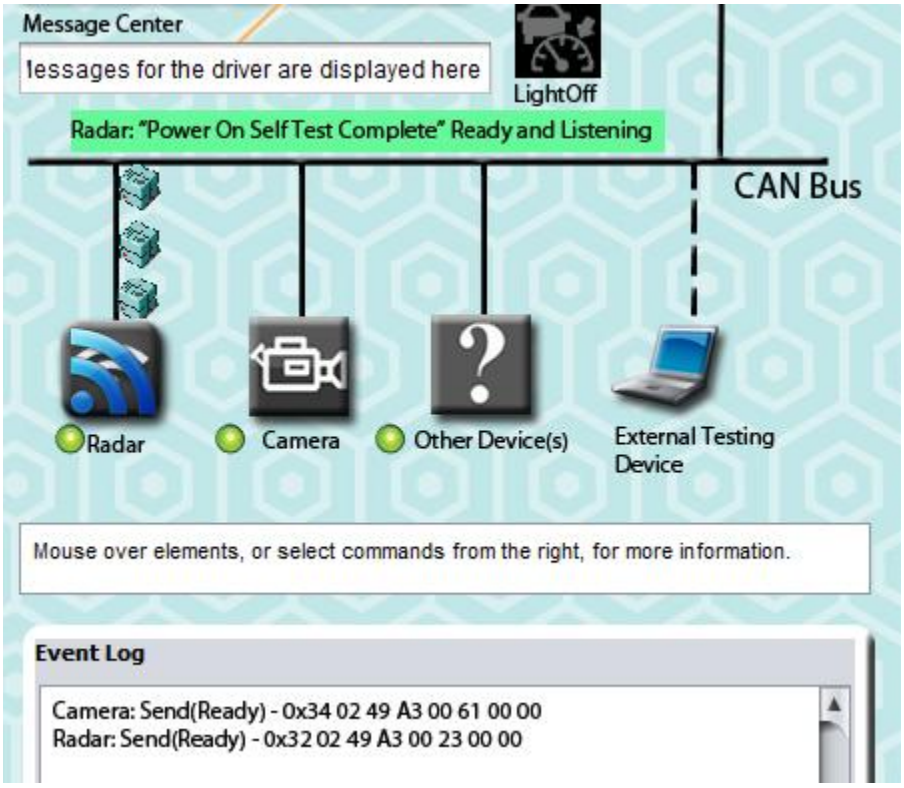

Figure 5.2: A snapshot of the Prototype in action during the broadcast of the results of the radar's diagnostic test.

There are three different blue icons representing the current state of each module that can appear throughout the simulation. No icon means the device is idle, meaning there are no diagnostic actions to take. Figure 5.3 summarizes these icons.

Figure 5.3: Icons used in the prototype.

At any time, the user of the prototype may select an event from a drop-down menu for a device on the right side, and then press “Trigger Selected Event”. This will simulate the particular event. For example, if a “Major Error, Operation Ceases” event is selected for the radar, the radar will act as if it has encountered this event. Note that many errors and events will not be detected until each module performs its periodic self check. The interval delay for these can be modified under the control panel for each device. Alternatively, the simulation can be sped up to expedite matters.

Figure 5.4 shows the results of triggering a fatal event for the Radar. It will broadcast its error message, the specific contents noted in the event log. Note that some values are unknown or beyond the scope of this product, and will be represented with 'X', or another explicit method of annotation. This particular message is being processed by the Instrument Cluster (CPU). The CPU has decided to inform the driver with a message in the message center and by turning on the ACC light, as the adaptive cruise control cannot function without the radar. Note the status light for the radar has turned red, indicating its failure. This failure can be 'fixed' by sending a recovery event from the Radar Event Controls. This represents the error resolving itself, and is needed to demonstrate intermittent failure and error logging policies. Alternatively, using the “Reset to Defaults” command under the System Commands will reset the entire simulation back to its default values.

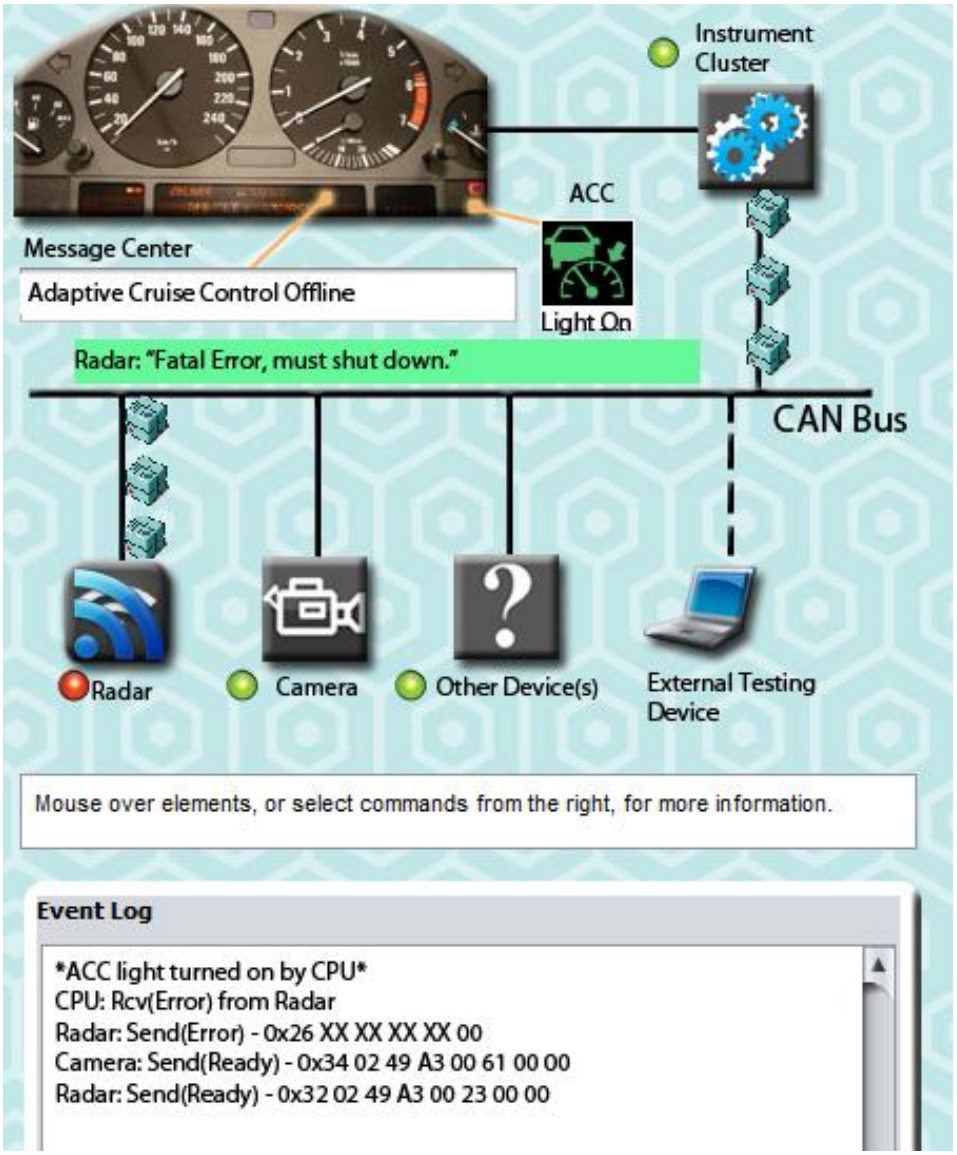

Figure 5.4 Screen capture of the Prototype, demonstrating the driver interface representations and logging capability.

All other devices and commands follow a similar pattern of triggering the event, and then allowing the diagnostic system to resolve itself. The error logging and messaging required to inform each device, or the driver, of the error is generated automatically. Note that multiple events may be triggered at once, though by requirements the same type of error will not be logged more than once.

6 References

- Almeida, Luis. “Safety-critical automotive systems: New developments in CAN”. http://www.artist-embedded.org/docs/Events/Artist%20WS%20at%20DATE06_Munich/2-ARTIST-DATE06-Almeida.pdf
- “CAN Basics Part-1”, Digi-Key Corporation: Product Training Modules. http://dkc1.digikey.com/us/en/tod/Renesas/CANBasicsPart-1_NoAudio/CANBasicsPart-1_NoAudio.html
- “Controller Area Network”, Infineon Technologies: microcontroller.com, http://microcontroller.com/learn-embedded/CAN1_sie/CAN1big.htm . Aug. 1999
- Murphy, Niall. “A short Trip on the CAN Bus”, EETimes. http://www.eetimes.com/discussion/murphy-s-law/4024614/A-short-trip-on-the-CAN-bus. Aug 11, 2003
- “Vehicle Networks: CAN-based Higher Layer Protocols”, Dr. Strang, Thomas. Deutches Zentrum fuer Luft-und Raumfahrt. http://www.sti-innsbruck.at/fileadmin/documents/vn-ws0809/03-vn-CAN-HLP.pdf”

7 Point of Contact

For further information regarding this document and project, please contact Prof. Betty H.C. Cheng at Michigan State University (chengb at cse.msu.edu). All materials in this document have been sanitized for proprietary data. The students and the instructor gratefully acknowledge the participation of our industrial collaborators.