Requirements Analysis Document

Christian Balnies, Jonathan Brown, Zain Kazim, Brian Lawnichak, Mark Pulver

Anti-Lock Brake System
Group ABS 2

November 16, 2001
# Contents

1. **Introduction** ................................................................. 4
   1.1 Problem Description ....................................................... 4
   1.2 Motivation ................................................................... 4

2. **Overview** .................................................................. 4

3. **Requirements** ............................................................ 5
   3.1 Perform self-diagnostic test sequence .............................. 5
   3.2 Check the input from the individual wheel sensors and compute their deceleration ................................. 6
   3.3 Check for wheel lock-up and send signals to the actuators .......................................................... 6
   3.4 Reset the System ............................................................. 7

4. **UML Analysis** .............................................................. 7
   4.1 Use Cases .................................................................. 7
   4.2 Object Model ................................................................. 12
      4.2.1 Class Diagram .......................................................... 12
      4.2.2 Data Dictionary ......................................................... 14
   4.3 Dynamic Model ............................................................... 17
      4.3.1 High Level System Behavior ........................................ 17
      4.3.2 Controller Class ........................................................ 17
      4.3.3 Manager Class .......................................................... 18
      4.3.4 Timer Class ............................................................... 20
      4.3.5 Wheel Monitor Class ................................................... 21
      4.3.6 Dash Light Class ........................................................ 23
      4.3.7 Hard Reset Class ......................................................... 24
      4.3.8 Wheel Cylinder Class .................................................. 25
      4.3.9 Main Brake Valve Class ................................................. 25
      4.3.10 Relief Valve Class ...................................................... 26
      4.3.11 Gnome Class ............................................................ 26
      4.3.12 Wheel Speed Sensor Class .......................................... 28
      4.3.13 Normal (Non-Skid) Braking Operation Sequence Diagram One ..................................................... 28
      4.3.14 Normal (Non-Skid) Braking Operation Sequence Diagram Two ..................................................... 31
      4.3.15 Wheel Skid Braking Operation Sequence Diagram ................................................................. 33

5. **Promela Specification for System** ................................. 36
   5.1 Model Checking and Promela Overview ............................ 36
   5.2 Promela Model ............................................................... 37
      5.2.1 Macros, Channels, and Booleans .................................. 37
      5.2.2 Broadcast Proctypes ..................................................... 38
      5.2.3 Manager Proctype ........................................................ 40
      5.2.4 Wheel Monitor Proctype ................................................. 43
      5.2.5 Wheel Cylinder Gnome Proctype .................................... 47
      5.2.6 Brake Valve Proctype ...................................................... 50
      5.2.7 Relief Valve Proctype .................................................... 51
      5.2.8 Dash Light Proctype ...................................................... 53
      5.2.9 Timer Proctype ............................................................ 54
1 Introduction

The Anti-Lock Brake System (ABS) was first developed for use in aircraft. Since its creation, the applications of ABS have expanded into other domains. For this project, we are considering the automotive domain.

When an automobile experiences wheel lock-up due to rapid deceleration, the result is skidding and the loss of driver control. ABS prevents this situation by controlling the brake pressure applied to the wheel, thereby allowing the car to stop (or slow down) in a controlled fashion.

This document analyzes the requirements for the ABS. The purpose of this analysis is to describe the components, performance and functionality of the system. The requirements document is divided into five sections:

- Introduction to the domain
- Overview
- Specific requirements
- Unified Modeling Language (UML) analysis of the ABS
- Encoding of the state diagrams in Promela

1.1 Problem Description

ABS brakes prevent lock-up of the wheels in hard braking situations where the traction is low. Wheel lock-up can cause loss of steering control and can lead to uncontrolled skids.[1] ABS works by momentarily releasing brake fluid pressure from the caliper just before the wheel locks up. The impending lock-up is sensed by measuring the deceleration rate at the wheel. This allows the driver to apply maximum braking pressure while only the pressure required is transmitted to the brakes.[1] This solves the problem of uncontrolled skidding and creates a safer driving environment.

1.2 Motivation

The purpose of the ABS is to maximize the amount of pressure applied to the wheel in order to prevent wheel lock-up during rapid deceleration. This prevents uncontrolled skidding, creating a much safer automobile in sub-optimal situations. Once the skidding has ceased, the ABS is deactivated and normal braking resumes. The motivation of this analysis is to give an in-depth view of the functionality of our ABS.

2 Overview

ABS is an automotive embedded system. In a general embedded system, information about the current state of the system is obtained from sensor readings. From this information, the controller can designate certain actions within the system to occur. These actions are carried out by the actuators.

The ABS is a subsystem of the braking system. Its functionality relies on sensor readings obtained by sensors placed at each wheel. ABS performs a system check immediately after automobile ignition and after each application of the brake. In the (rare) case of error detection in the
sensors, an ABS error light is illuminated in the dashboard and ABS enters its error state. While ABS is in its error state, the automobile is limited to mechanical braking. In order to exit this state, the system can be reset by a certified mechanic after the necessary repairs have been made.

In the functioning ABS system, the current state is defined by the sensors. While the brakes are being applied, the sensors evaluate the rate of deceleration. If extreme pressure is applied by the brake to the wheel (also known as rapid deceleration) and the sensors detect skidding, the ABS relieves the pressure by opening the relief valve. This allows brake fluid to escape (in a controlled manner) to a reservoir (similar to “pumping” the brakes), thereby relieving brake pressure. During this time, the sensors continue to take readings. Once the sensors no longer detect a skid, the brake pressure is held constant. When braking ceases, the fluid that collected in the reservoir is returned and the relief valve is closed. Mechanical braking resumes.

3 Requirements

The Anti-lock Braking System is required to perform the following tasks:

1. Perform test sequences on the ABS.
2. Monitor the input from the individual wheel sensors and compute wheel deceleration while braking.
3. Check for wheel lock-up.
4. Send signals to the actuators that will adjust brake pressure on the wheels to respond to rapid deceleration.
5. Send the system failure notification to the driver.
6. Reset the system.
7. ABS Controller will also respond to prevent a vehicle-spin situation, with equations for this to be provided by the customer at design phase.

3.1 Perform self-diagnostic test sequence

1. When the car is initially started, the ABS goes through a sequence of tests on itself to check if the system is fully functional.
2. The tests involve checking the operation of the ABS controller, which receives information from the individual wheel sensors and the state of the communication lines.
3. A similar kind of testing is also performed every time the driver applies the brake and the system’s performance is evaluated.
4. These tests encompass the performance of ABS on each of the individual wheels: the performance of the ABS controller, actuators, and sensors. These components are described below:

   • ABS Controller: The central computer of the ABS that processes the information from the individual wheel sensors and sends signals to the actuators for action.
• ABS Actuators (Wheel Cylinders): A component of the ABS that performs the actual mechanical work of releasing the pressure from the caliper to modulate the brakes.

• ABS Sensors: Each wheel has an ABS sensor associated with it. These sensors compute the speed of each of the wheels, which is then sent to the main ABS computer (controller) to calculate the deceleration of the entire car and the individual wheels.

5. A signal in the form of a light appears on the driver’s console in case of system error and the braking mechanism is switched to the normal, non-ABS managed mode so mechanical brakes will still function correctly.

6. The system error also occurs if the car speed is over 10 m/s and one or more of the wheels is at 0 m/s.

3.2 Check the input from the individual wheel sensors and compute their deceleration

1. The ABS receives input speeds from the wheel sensors mounted on each of the individual wheels.

2. The speed is measured in meters per second.

3. The speed for each wheel is passed to the system’s controller.

4. The wheel speeds are used to compute the speed of the entire system and the deceleration of the individual wheels.

5. If the system speed is above 15 mph or roughly 6.7 m/s, rapid deceleration of a wheel once the brakes are applied activates the ABS, and the ABS controller checks for wheel skidding every 10 milliseconds.

6. Rapid deceleration is defined as a deceleration of more than 8.82 m/s² (0.9 g).

3.3 Check for wheel lock-up and send signals to the actuators

1. Rapid deceleration detected on any of the wheels is a sign of an impending wheel lock-up.

2. Upon detecting the impending lock-up, the controller sends signals to the wheel actuators, or monitors, to perform certain actions on individual wheels that will prevent them from locking up.

3. Even though all the wheels have separate speed sensors but their deceleration and lockup isn’t completely independent from each other. An example of this can be seen in the case where one side of the vehicle is on a completely different surface than the other. For instance, one side is on the icy surface while the other side is on the dry surface. In this case two of the wheels would have different deceleration rates than the other two. The controller needs keep track of such a situation. If left unchecked, this will put the vehicle into an unrecoverable spin.

4. From direct client input, detection of and response to rapid deceleration must occur within 0.015 seconds.
5. The wheel monitors avoid lock-up by releasing the brake fluid from within the wheel cylinders to a reservoir.

6. This leads to the application of a constant pressure on the wheel once the skid has stabilized.

7. The wheel monitors pump the brakes for 1/10 of a second.

8. Pumping the brakes is accomplished by alternating between having the main brake valve or the relief valve open and the other valve closed every 5 ms.

9. This allows the driver to apply maximum braking pressure without being worried about wheel lock-up, as only the pressure required is transmitted to the wheels.

10. If rapid deceleration is detected again while holding the pressure, more fluid is transferred.

11. The brake fluid is returned back to the main fluid reservoir once the rapidly decelerating wheel is brought back to the non-lock-up state.

12. The ABS is not engaged while the speed of the car is below 15 mph, even if the wheel is rapidly decelerating.

3.4 Reset the System

1. When the ABS is in the error state, only a mechanic can reset the system, by signal from the main on-board vehicle computer.

2. The car retains mechanical braking functionality in the error state.

4 UML Analysis

4.1 Use Cases

Use cases represent high-level functional goals for the Anti-Lock Brake System. Please see Figure 1 for a graphical diagram of use cases for the system.

<table>
<thead>
<tr>
<th>Use case: PowerOn</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors:</strong> Driver</td>
</tr>
<tr>
<td><strong>Type:</strong> Primary, Essential</td>
</tr>
<tr>
<td><strong>Description:</strong> Power On executes Power-On Self Test to ensure processor and communication lines are functioning correctly. Power On also uses the Brake Open / Relief Closed use case to ensure all wheel brakes are in a state for normal mechanical brake functioning, regardless of the result of Power-On Self Test.</td>
</tr>
<tr>
<td><strong>Cross-reference:</strong> 3.1.1, 3.1.5, 3.3.10</td>
</tr>
<tr>
<td><strong>Uses:</strong> Power-On Self Test, Brake Open / Relief Closed</td>
</tr>
<tr>
<td><strong>Extended by:</strong></td>
</tr>
<tr>
<td><strong>Extends:</strong></td>
</tr>
</tbody>
</table>
Use case: Power-On Self Test

Actors:

Type: Secondary

Description: Power-On Self Test executes a suite of processor and memory tests in addition to the tests in the System Check. If any of the tests fail, the system is moved into an error state and the Dash Light is signaled to light.

Cross-reference: 3.1.1, 3.1.3, 3.1.5

Uses:

Extended by:

Extends: System Check
Use case: System Check

Actors: Dash Light, Wheel Cylinder

Type: Primary, Tertiary

Description: Checks communication lines between controller and Wheel Cylinders. Also uses Get Speed on all Wheel Speed Sensors to determine if any of the wheels is returning impossible values, such as 0 m/s on one wheel while 10 m/s on all other wheels and not in the braking state. If an error is sensed, the system is moved into an error state and the Dash Light is lit.

Cross-reference: 3.1.1, 3.1.2, 3.1.4.2, 3.1.4.3, 3.1.5, 3.2.1

Uses: Get Speed

Extended by: Power-On Self Test

Extends:

Use case: Get Speed

Actors: Wheel Speed Sensor

Type: Primary

Description: Gets current speed from Wheel Speed Sensor.

Cross-reference: 3.1.4.3

Uses:

Extended by:

Extends:
Use case: Braking

Actors: Driver

Type: Primary

Description: Braking mode uses monitor wheel state to keep track of state of each of the wheels and adjust the brake pressure on any wheel for which rapid deceleration is sensed and the system is above the minimum speed for ABS engagement.

Sequence of events for responding to sensed rapid deceleration is as follows:

First, the pressure will be decreased using Relief Open / Brake Closed to open the relief valve and close the main brake valve until rapid deceleration is no longer sensed on the wheel.

Then the pressure will be held constant by switching rapidly between having the relief valve open and having the main brake valve open for the specified interval.

Finally, the pressure is allowed to increase again using Brake Open / Relief Closed to close the relief valve and open the main brake valve.

At steps two or three in the response sequence, a sensed rapid deceleration will cause the response sequence to be restarted.

Terminated by Release Brake use case.

Cross-reference: 3.1.3, 3.1.5, 3.3.2, 3.3.4, 3.3.7, 3.3.10

Uses: Relief Open / Brake Closed, Brake Open / Relief Closed

Extended by:

Extends:

Use case: Relief Open / Brake Closed

Actors: Wheel Cylinder

Type: Primary

Description: For a specific wheel, Relief Open / Brake Closed signals to open the relief valve on the Wheel Cylinder to drain brake fluid into a secondary reservoir while signaling to close the main brake valve on the Wheel Cylinder. Thus, this action lowers the brake pressure on the wheel.

Cross-reference: 3.1.4.2, 3.3.4

Uses:

Extended by:

Extends:
<table>
<thead>
<tr>
<th>Use case:</th>
<th>Brake Open / Relief Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors:</strong></td>
<td>Wheel Cylinder</td>
</tr>
<tr>
<td><strong>Type:</strong></td>
<td>Primary</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>For a specific wheel, Brake Open / Relief Closed signals to open the main brake valve on the Wheel Cylinder and signals to close the relief valve that drains to the secondary reservoir. Thus, this action raises the brake pressure on the wheel to that exerted by the Driver.</td>
</tr>
<tr>
<td><strong>Cross-reference:</strong></td>
<td>3.1.4.2, 3.3.10</td>
</tr>
<tr>
<td><strong>Uses:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Extended by:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Extends:</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case:</th>
<th>Power Off</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors:</strong></td>
<td>Driver</td>
</tr>
<tr>
<td><strong>Type:</strong></td>
<td>Primary</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Driver powers off system. It is assumed that actuators will not respond after a Power Off, thus no action is taken by the ABS system.</td>
</tr>
<tr>
<td><strong>Cross-reference:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Uses:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Extended by:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Extends:</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case:</th>
<th>Release Brake</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actors:</strong></td>
<td>Driver</td>
</tr>
<tr>
<td><strong>Type:</strong></td>
<td>Primary</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Signals for end of Braking use case. Uses Brake Open / Relief Closed to open all main brake valves and close all relief valves.</td>
</tr>
<tr>
<td><strong>Cross-reference:</strong></td>
<td>3.3.10, 3.4.2</td>
</tr>
<tr>
<td><strong>Uses:</strong></td>
<td>Brake Open / Relief Closed</td>
</tr>
<tr>
<td><strong>Extended by:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Extends:</strong></td>
<td></td>
</tr>
</tbody>
</table>
Use case: Reset

Actors: Technician
Type: Secondary
Description: Resets system and thus moves it out of an error state.
Cross-reference: 3.4.1
Uses:
Extended by:
Extends:

4.2 Object Model

4.2.1 Class Diagram

The center of the Object Model is the Class Diagram (Figure 2), which details attributes and methods of the classes in the Anti-Lock Braking System.
Figure 2: Class Diagram for Anti-Lock Braking System
4.2.2 Data Dictionary

- Vehicle Class - Model of relevant external stimuli from vehicle.
  - Methods
    * void powerOn( void ): Sends power on signal to Anti-Lock Braking System
    * void powerOff( void ): Sends power off signal to Anti-Lock Braking System
    * void setBrake( void ): Sends signal to Anti-Lock Braking System when brake is pushed by the driver
    * void releaseBrake( void ): Sends signal to Anti-Lock Braking System when driver releases the brake.
    * void resetABS( void ): Used by the mechanic or technician when ABS is serviced, external signal is a hard reset to the system that causes everything to return to a default state.

- Anti-Lock Braking System Class - Container class for Anti-Lock Braking System components
  - Attributes
    * bool isError: stored persistently, default false: Set by selfCheck() and brakeCheck() methods of Manager. Used to indicate whether an error has been detected. This is stored persistently and can only be changed from true to false by the resetABS() signal, although it can be changed from false to true by any of the Manager’s test suites.

- Dash Light Class - (Actuator) Interface to console light that indicates ABS error to driver.
  - Methods
    * void lightOn( void ): Turns console light on (Signal to main computer)
    * void lightOff( void ): Turns console light off (Signal to main computer)

- Controller Class - Container class for major software components of ABS
  - Attributes
    * int UPDATE_INTERVAL: default 10 ms: Time between updates of system status
    * real IMPOSSIBLE_DIFF: default value 10.0 m/s: If zero velocity is detected on a speed sensor but the system speed meets or exceeds this value, this causes an error condition.
    * real ACTIVATION_SPEED: default 15.0 mph or 6.7 m/s: If the system speed is below this value, or drops below this value, all ABS-controlled braking activity stops.
    * int HOLD_INTERVAL: default 5 ms: When system holds pressure constant on a wheel, time between opening one valve and opening the other valve.
    * int HOLD_THRESHOLD: default 100 ms: Time to hold pressure once rapid deceleration behavior on a wheel has stabilized.
    * real ACCEL_THRESHOLD: default -0.9g (-8.82) m/s²: Acceleration threshold for wheels that, below which, wheel monitors can begin a decrease of pressure to stop rapid deceleration (upper bound on rapid deceleration).
- Manager Class - Software thread that responds to external signals, activates and pauses wheel
  monitor threads, and manages signals to the driver via the dash light.

  - Attributes
    * real systemSpeed: default 0.0: Calculated system speed (upper bound of speed on
      the four wheels). Uses the currSpeed() method of each wheel monitor.

  - Methods
    * void dim( void ): Signals the dash light to turn off.
    * void light( void ): Signals the dash light to turn on.
    * bool selfCheck( void ): Executes a processor diagnostic to ensure no processor faults,
      and then executes a communication test with the wheel cylinders via the wheel
      monitor monitorCheck() method. Returns true if no errors were detected.
    * bool brakeCheck( void ): Executes a communication test with the wheel cylinders via
      the wheel monitor monitorCheck() method. Returns true if no errors were detected.
    * void timerOn( void ): Sends signal to the timer to begin signaling the manager to
      update every UPDATE_INTERVAL.
    * void timerOff( void ): Sends signal to the timer to stop signaling the manager to
      update.
    * void errorBroadcast( void ): Sends signal to all wheel monitor objects that an error
      has occurred and all wheel cylinders should be reverted to non-ABS braking state.
    * void wheelMonitorActive( void ): Sends signal to all wheel monitor objects to acti-
      vate monitoring of wheel speed and respond to rapid deceleration conditions.
    * void wheelMonitorIdle( void ): Sends signal to all wheel monitor objects to stop
      monitoring wheel speed and to return wheel cylinders to a non-ABS braking state.
    * void wheelMonitorUpdate( void ): Signals all wheel monitor objects to update their
      wheel speed value and acceleration and to take any action needed to handle a de-
      tected rapid deceleration condition.
    * void calcSystemSpeed(): Causes the manager to poll all wheel monitors with currSpeed() and to update systemSpeed attribute to be an upper bound of speeds on all wheels.

- Timer Class - Update timer that can signal the manager every UPDATE_INTERVAL to
  update.

  - Methods
    * void managerUpdate( void ): Signals the manager to update

- Wheel Monitor Class - Logical control module for each wheel. A wheel monitor object handles
  per-wheel state information and manages per-wheel ABS braking activity.

  - Attributes
    * real currentAccel: default 0.0 m/s²: Current acceleration of wheel, calculated as
      a discrete approximation to the acceleration of the difference of currentSpeed and
      previousSpeed over the last UPDATE_INTERVAL.
* int timeInHold: default 0 ms: Used to keep track of the amount of time pressure has
  been held constant on a wheel after end of rapid deceleration on a wheel. This value
  is incremented on each update of the wheel monitor if the wheel monitor is holding
  pressure constant on a wheel, and once this value exceeds HOLD_INTERVAL, it is
  zeroed and pressure is restored to normal.
* real currentSpeed: default 0.0 m/s: Current speed reading from the wheel speed
  sensor after the most recent update.
* real previousSpeed: default 0.0 m/s: Speed reading from the wheel speed sensor
  before last update. This should be UPDATE_INTERVAL older than currentSpeed.

- Methods
  * void normalPressure( void ): Signals wheel cylinder to enter a “normal braking
    pressure” mode — i.e., the state the wheel cylinders would be in without ABS-
    managed braking.
  * void decreasePressure( void ): Signals wheel cylinder to decrease brake pressure on
    the wheel.
  * void holdPressure( void ): Signals wheel cylinder to hold brake pressure constant
    on the wheel.
  * real calcAccel( void ): Calculates acceleration on the wheel by a discrete approxima-
    tion to the derivative — i.e., (currentSpeed - previousSpeed) / UPDATE_INTERVAL.
  * real currSpeed( void ): Returns the value of the currentSpeed variable
  * bool monitorCheck( void ): Causes the monitor to call and return the value of
    gnomeLive() for its wheel cylinder to test the communication line.

- Wheel Speed Sensor Class - Model of the wheel speed sensor input from each wheel. (Sensor)
  - Methods
    * real getSpeed( void ): Returns the current speed from the wheel in m/s.

- Wheel Cylinder Class - Container for main brake valve, relief valve, and gnome at each wheel.

- Valve Class - General valve class. (Actuator)
  - Methods
    * void open( void ): Opens valve.
    * void close( void ): Closes valve.

- Main Brake Valve Class - (Actuator) Subclass of Valve class. The Main Brake Valve models
  the valve that connects to the main reservoir, and is open during normal (non-ABS-managed)
  braking. The Main Brake Valve differs only in starting state. See state diagram below.

- Relief Valve Class - (Actuator) Subclass of Valve class. The Relief Valve models the valve that
  connects to the secondary reservoir on the wheel cylinder, and is opened to decrease pressure
  on a wheel to recover from rapid deceleration. The Relief Valve differs only in starting state.
  See state diagram below.

- Gnome Class - Interface to Wheel Cylinder for Wheel Monitor, the Gnome responds to
  requests to decrease, hold, or normalize brake pressure on the wheel by managing the Main
  Brake Valve and the Relief Valve, enforcing the behavior that when one is opened the other
  is closed.
- Methods
  * void openBrakeValve( void ): Signals Main Brake Valve to open.
  * void closeBrakeValve( void ): Signals Main Brake Valve to close.
  * void openReliefValve( void ): Signals Relief Valve to open.
  * void closeReliefValve( void ): Signals Relief Valve to close.
  * bool gnomeLive( void ): Function called by Wheel Monitor communication line test.
    Returns true to indicate successful receipt of Wheel Monitor command.

- Hard Reset Class - Clears all error flags within the system to return ABS functionality back to operating condition.

  - Methods
    * void clearError( void ): Set by the Technician to reset or clear all error flags in the system so that bool isError in the Anti-Lock Braking System Class is set to false.

4.3 Dynamic Model

The dynamic model describes the behavior of the system. The state diagrams provide detailed behavioral information for important classes in the class hierarchy (Figure 2). Sequence diagrams provide a detailed view of specific scenarios or traces through the state diagrams. As a convention, for a state diagram of a specific class, if events or actions originate from a different class, this is shown with C++-style class member notation.

4.3.1 High Level System Behavior

The high-level behavior of the ABS is as follows:

- The system starts in the Vehicle Off state.
- Receipt of the powerOn() signal from the Vehicle causes the system to transition to the ABS On state.
- In the ABS On state, there are eleven concurrent processes: the Controller, the Dash Light, the Hard Reset, each of the four Wheel Cylinders, and each of the four Wheel Speed Sensors.
- A powerOff() signal from the Vehicle at any time causes the system to return to the Vehicle Off state.
- Please see specific state diagrams for the Controller, Dash Light, Hard Reset, Wheel Cylinder, and Wheel Speed Sensor classes for further details.

4.3.2 Controller Class

The Controller behaves as follows:

- The Controller consists of six concurrent processes: the Manager, the Timer, and each of the four Wheel Monitors.
- Please see specific state diagrams for the Manager, Timer, and Wheel Monitor classes for further details.
4.3.3 Manager Class

The Manager behaves as follows:

- On activation of the Manager, following a powerOn() signal from the Vehicle, the Manager enters a Testing state. On entry, the Dash Light is signaled on by light() and selfCheck() is executed.
- If the selfCheck() detects an error, it will call the setError() function which will set isError to true.
- If there is a persistent error, isError remains true.
- On completion of the self check, the transitions guarded by isError are checked.
- If there is no error condition, the Manager transitions into the Watching state. It also signals for the Dash Light to dim.
- If there is an error condition, the Manager transitions into the Error state. An errorBroadcast() sends messages to all Wheel Monitors to revert to a non-ABS-braking configuration for mechanical brake operation. Mechanical braking is still operational in the Error state.
- In the Watching state, a setBrake() event causes a transition to the Brake Test state.
- On entry to the Brake Test state, brakeCheck() is executed, which tests communication lines to the Wheel Cylinders. If this test fails, isError is set to true by setError().
• If there is an error, the Dash Light is signaled to light, an errorBroadcast() event signals all Wheel Monitors to revert to a non-ABS-braking configuration for mechanical brake operation, and the Manager enters the Error state.

• If there is no error after brakeCheck(), the Timer is turned on to begin signaling to update, the wheelMonitorActive() event causes the Wheel Monitors to activate and update their current speed, and then calcSystemSpeed() updates the Manager’s record of the system speed. The Braking state is entered.

• If the systemSpeed is found to be below the ACTIVATION_SPEED threshold while the Manager is in the Braking state, the Wheel Monitors are returned to an idle state and the Manager returns to the Watching state. The Timer is turned off.

• If the brake is released while the Manager is in the Braking state, the Timer is turned off and the Wheel Monitors are returned to an idle state. The Manager returns to the Watching state.

• While in the Braking state, every UPDATE_INTERVAL the Manager receives managerUpdate() signals from the Timer. This causes the Manager to signal the Wheel Monitors to
update and to recalculate the system speed variable systemSpeed.

- When the system enters the Error state, it continues to poll the isError boolean, and when that becomes false, i.e., when the Hard Reset receives the resetABS() signal, the Manager transitions back to the Testing state and repeats the selfCheck() behavior.

- Calls to setError() and clearError() are not shown as setError() is, by definition above, part of selfCheck() and brakeCheck(), and clearError() appears in the Hard Reset diagram.

4.3.4 Timer Class

The Timer is used by the Manager for timed updates of system speed and the Wheel Monitors. Its behavior is as follows:

- The Timer starts in the Idle state.
• A timerOn() message from the Manager causes the Timer to transition to the Active state, where every UPDATE_INTERVAL it issues a managerUpdate() signal.

• If the Timer receives a timerOff() messages while in the Active state, it transitions to the Idle state and stops sending signals to the Manager.

4.3.5 Wheel Monitor Class
Figure 7: State Diagram for Wheel Monitor Class
The Wheel Monitor is responsible for managing the per-wheel information and status. Its behavior is as follows:

- The Wheel Monitor begins in the *Idle* state. If it receives a `wheelMonitorActive()` signal, it updates its `currentSpeed` attribute by calling `getSpeed()` on its Wheel Speed Sensor, then transitions to the *Active* state.
- As a safeguard to ensure that the brake valves are in the right configuration for normal mechanical braking, in the *Idle* state, receipt of an `errorBroadcast()` signal issues a normal-Pressure signal to the Gnome for its Wheel Cylinder.
- In the *Active* state, receipt of either a `wheelMonitorIdle()` or `errorBroadcast()` message causes the Wheel Monitor to issue a `normalPressure()` signal to its Gnome and then set `timeInHold` to zero. The Wheel Monitor then transitions to the *Idle* state.
- Within the *Active* complex state, there are three simple states, *Monitor, Skid*, and *Hold*.
  - In each of *Monitor, Skid* and *Hold*, receipt of a `wheelMonitorUpdate()` signal causes the `currentSpeed` to be updated with `getSpeed()` from its Wheel Speed Sensor and the `currentAccel` to be recalculated with `calcAccel()`. This also causes re-entry of the state. In the *Hold* state, the `wheelMonitorUpdate` causes `timeInHold` to be incremented by `UPDATE_INTERVAL`.
  - On entry to the *Active* state, the Wheel Monitor enters the *Monitor* simple state. If currentAccel drops below `ACCEL_THRESHOLD`, i.e., large and negative, rapid deceleration is detected and the Wheel Monitor transitions to the *Skid* state and issues a `decreasePressure()` command to the Gnome.
  - In the *Skid* state, if the `currentAccel` is greater than or equal to `ACCEL_THRESHOLD`, the Wheel Monitor transitions to the *Hold* state and issues a `holdPressure()` command to the Gnome. It also causes `timeInHold` to be zeroed.
  - In the *Hold* state, if the `currentAccel` is less than `ACCEL_THRESHOLD`, the Wheel Monitor transitions to the *Skid* state and issues a `decreasePressure()` command to the Gnome. If `timeInHold` exceeds `HOLD_THRESHOLD`, the Wheel Monitor transitions to the *Monitor* state and the Gnome is issued a `normalPressure()` signal.

### 4.3.6 Dash Light Class

The Dash Light is an actuator that signals to the main computer to turn the dashboard display light on or off. Its behavior is as follows:

- The Dash Light starts in the *Off* state, with the dashboard light assumed to be off.
- On receipt of the `light()` signal, the Dash Light signals the main computer to turn the light on with the `lightOn()` event, and then transitions to the *On* state.
- From the *On* state, a `dim()` signal causes the Dash Light to signal the main computer to turn the light off with the `lightOff()` event, and then the Dash Light transitions to the *Off* state.
4.3.7 Hard Reset Class

The Hard Reset is an electronic relay that responds to a signal from the main computer to clear the error flag in the ABS. Its behavior is as follows:

- If the power is off, the Hard Reset is in an Off state.
- When the power is turned on by a powerOn() event, the Hard Reset is in the Ready state and is ready to clear the error condition.
- On receipt of the resetABS() signal, the Hard Reset does a clearError() which sets the isError variable to false, then returns to the Ready state.
- A powerOff() event returns the Hard Reset to the Off state.
4.3.8 Wheel Cylinder Class

The Wheel Cylinder behaves as follows:

- The Wheel Cylinder consists of three concurrent processes: the Main Brake Valve, the Relief Valve, and the Wheel Cylinder Gnome.
- Please see specific state diagrams for the Main Brake Valve, Relief Valve, and Gnome classes for further details.

4.3.9 Main Brake Valve Class

The Main Brake Valve is an actuator which opens or closes the valve that connects the fluid line from the main reservoir. Its behavior is as follows:

- The Main Brake Valve starts in the Open state.
• When a closeBrakeValve() signal is received, it signals to close the valve with the close() method and transitions to the Closed state.

• In the Closed state, the openBrakeValve() event causes the Main Brake Valve to open the valve with the open() method and transitions to the Open state.

4.3.10 Relief Valve Class

```
Gnome.closeReliefValve() / Valve.close()
Gnome.openReliefValve() / Valve.open()
```

Figure 12: State Diagram for the Relief Valve Class

The Relief Valve Class signals to open or close the valve on the secondary reservoir on each wheel cylinder that is used to bleed off brake fluid and thus decrease pressure on the wheel. Its behavior is as follows:

• The Relief Valve starts in the Closed state.

• On receipt of the openReliefValve() signal, it signals to open the secondary reservoir’s valve and transitions to the Open state.

• In the Open state, receipt of the closeReliefValve() signal causes the Relief Valve to close the secondary reservoir’s valve and transitions to the Closed state.

4.3.11 Gnome Class
Figure 13: State Diagram for Gnome Class
The Gnome responds to signals from the Wheel Monitor, and coordinates activity of the Main Brake Valve and Relief Valve in its Wheel Cylinder. Its behavior is as follows:

- The Gnome starts in the *Normal* state.
- In the *Normal* state, receipt of a `decreasePressure()` signal causes the Gnome to signal the Relief Valve to open with `openReliefValve()` and the Main Brake Valve to close with `closeBrakeValve()`, and then the Gnome transitions to the *Decreasing* state. Receipt of a `holdPressure()` signal causes the Gnome to transition to the *Holding* state.
- In the *Decreasing* state, receipt of a `normalPressure()` signal causes the Gnome to signal the Main Brake Valve to open with `openBrakeValve()` and the Relief Valve to close with `closeReliefValve()`, and then the Gnome transitions to the *Normal* state. Receipt of a `holdPressure()` signal causes the Gnome to transition to the *Holding* state.
- In the *Holding* state, receipt of a `decreasePressure()` signal causes the gnome to signal the Relief Valve to open with `openReliefValve()` and the Main Brake Valve to close with `closeBrakeValve()`, and then the Gnome transitions to the *Decreasing* state. Receipt of the `normalPressure()` signal causes the gnome to signal the Main Brake Valve to open with `openBrakeValve()` and the Relief Valve to close with `closeReliefValve()`, and then the Gnome transitions to the *Normal* state.
- While in the *Holding* state, the Gnome transitions quickly between the *Holding 1* and *Holding 2* states, both upon completion of do/ events. In the *Holding 1* state, the Main Brake Valve is opened with `openBrakeValve()` and the Relief Valve is closed with `closeReliefValve()`. In *Holding 2*, the Relief Valve is opened with `openReliefValve()` and the Main Brake Valve is closed with `closeBrakeValve()`. Arbitrarily, *Holding* starts in the *Holding* state.

### 4.3.12 Wheel Speed Sensor Class

The Wheel Speed Sensor has one state, *Sense Speed*, and when it receives a `getSpeed()` signal, it returns the current wheel speed.

### 4.3.13 Normal (Non-Skid) Braking Operation Sequence Diagram One

This sequence diagram details the sequence of events in normal (non-skip) braking when the system speed remains above the `ACTIVATION_SPEED`. The events can be grouped into four general sequences:

- **Power-On Sequence** - power-on ABS diagnostic with no persistent error flag set.
  - The Vehicle issues the `powerOn()` signal to the Manager.
  - The Manager signals for the Dash Light to be turned on and begins its `selfCheck()` diagnostic.
  - The `selfCheck()` finds no error and `isError` remains false, so the Manager signals the Dash Light to be turned off.
- **Brake Set Sequence** - the ABS receives a signal that the brake has been set and executes at-brake-time diagnostic, which succeeds, then Wheel Monitors and Timer are activated.
- The Vehicle issues the setBrake() signal to the Manager.
- The Manager executes the brakeCheck() diagnostic, which finds no error.
- The Manager signals the Timer to turn on.
- The Manager signals all four Wheel Monitors to activate, which causes each Wheel Monitor to poll its Wheel Speed Sensor to update its internal speed variable.
- The Manager recalculates the system speed.

**Active Monitoring Sequence** - Since the system speed remains above ACTIVATION SPEED, the Timer continues to signal the Manager to update, which in turn signals the Wheel Monitors to update.

- The Timer issues a managerUpdate() signal to the Manager.
- The Manager issues a wheelMonitorUpdate() signal to all four Wheel Monitors.
- Each Wheel Monitor polls its Wheel Speed Sensor to update its internal speed variable and recalculates the wheel acceleration.
- The Manager recalculates the system speed.
- There is no additional action as no rapid deceleration is detected.

**Brake Release Sequence** - The Manager receives a signal from the Vehicle that the brake has been released, which causes both the Wheel Monitors and the Timer to be idled.

- The Vehicle issues the releaseBrake() signal to the Manager.
- The Manager issues a timerOff() signal to the Timer, stopping it from issuing further managerUpdate() signals.
Figure 15: Sequence Diagram Of Normal (Non-Skid) Braking Operation Above ACTIVA-
TION_SPEED
- The Manager issues a wheelMonitorIdle() signal to all four Wheel Monitors.
- Each Wheel Monitor signals its Gnome to return its wheel cylinder to a “normal” state
  with normalPressure().
- Each Gnome signals the Main Brake Valve in its wheel cylinder to open and the Relief
  Valve to close.

4.3.14 Normal (Non-Skid) Braking Operation Sequence Diagram Two

This sequence diagram details the sequence of events in normal (non-skid) braking when the
system speed drops below the ACTIVATION_SPEED. The events can be grouped into five general
sequences:

- **Power-On Sequence -** power-on ABS diagnostic with no persistent error flag set.
  - The Vehicle issues the powerOn() signal to the Manager.
  - The Manager signals for the Dash Light to be turned on and begins its selfCheck()
    diagnostic.
  - The selfCheck() finds no error and isError remains false, so the Manager signals the
    Dash Light to be turned off.

- **Brake Set Sequence -** the ABS receives a signal that the brake has been set and executes
  at-brake-time diagnostic, which succeeds, then Wheel Monitors and Timer are activated.
  - The Vehicle issues the setBrake() signal to the Manager.
  - The Manager executes the brakeCheck() diagnostic, which finds no error.
  - The Manager signals the Timer to turn on.
  - The Manager signals all four Wheel Monitors to activate, which causes each Wheel
    Monitor to poll its Wheel Speed Sensor to update its internal speed variable.
  - The Manager recalculates the system speed.

- **Active Monitoring Sequence -** Since the system speed remains above ACTIVATION_SPEED,
  the Timer continues to signal the Manager to update, which in turn signals the Wheel Mon-
  itors to update.
  - The Timer issues a managerUpdate() signal to the Manager.
  - The Manager issues a wheelMonitorUpdate() signal to all four Wheel Monitors.
  - Each Wheel Monitor polls its Wheel Speed Sensor to update its internal speed variable
    and recalculates the wheel acceleration.
  - The Manager recalculates the system speed.
  - There is no additional action as no rapid deceleration is detected.

- **Speed Threshold Reached Sequence -** When the system speed drops below ACTIVATION_SPEED,
  the Manager signals the Timer and Wheel Monitors to idle.
  - The Timer issues a managerUpdate() signal to the Manager.
  - The Manager issues a wheelMonitorUpdate() signal to all four Wheel Monitors.
Figure 16: Sequence Diagram of Normal (Non-Skid) Braking Leading To Speed Below ACTIVATION_SPEED
- Each Wheel Monitor Polls its Wheel Speed Sensor to update its internal speed variable.
- The Manager calculates that the system speed has dipped below ACTIVATION_SPEED, and thus issues a wheelMonitorIdle() signal to all four Wheel Monitors. Manager also turns Timer off with a timerOff() signal.
- Each Wheel Monitor signals its Gnome to return to normalPressure() — i.e., mechanical braking conditions.
- Each Gnome signals to its associated Main Brake Valve to open and its associated Relief Valve to close.

- **Brake Release Sequence** - The Manager receives a signal from the Vehicle that the brake has been released, but takes no action as the ABS has already idled.
  - Vehicle issues releaseBrake() signal to the Manager.

### 4.3.15 Wheel Skid Braking Operation Sequence Diagram

This sequence diagram details the sequence of events in braking following the detection of rapid deceleration on a wheel. The events can be grouped into several general sequences:

- **Power-On Sequence** - power-on ABS diagnostic with no persistent error flag set.
  - The Vehicle issues the powerOn() signal to the Manager.
  - The Manager signals for the Dash Light to be turned on and begins its selfCheck() diagnostic.
  - The selfCheck() finds no error and isError remains false, so the Manager signals the Dash Light to be turned off.

- **Brake Set Sequence** - the ABS receives a signal that the brake has been set and executes at-brake-time diagnostic, which succeeds, then Wheel Monitors and Timer are activated.
  - The Vehicle issues the setBrake() signal to the Manager.
  - The Manager executes the brakeCheck() diagnostic, which finds no error.
  - The Manager signals the Timer to turn on.
  - The Manager signals all four Wheel Monitors to activate, which causes each Wheel Monitor to poll its Wheel Speed Sensor to update its internal speed variable.
  - The Manager recalculates the system speed.

- **Active Monitoring Sequence** - Since the system speed remains above ACTIVATION_SPEED, the Timer continues to signal the Manager to update, which in turn signals the Wheel Monitors to update.
  - The Timer issues a managerUpdate() signal to the Manager.
  - The Manager issues a wheelMonitorUpdate() signal to all four Wheel Monitors.
  - Each Wheel Monitor polls its Wheel Speed Sensor to update its internal speed variable and recalculates the wheel acceleration.
Figure 17: Sequence Diagram of Rapid Deceleration Detection and Recovery Behavior
- The Manager recalculates the system speed.
- There is no additional action as no rapid deceleration is detected.

• Rapid Deceleration Sequence - Immediate system response to detection of rapid deceleration.
  - The Timer Issues a managerUpdate() signal to the Manager.
  - The Manager issues a wheelMonitorUpdate() signal to all four Wheel Monitors.
  - Each Wheel Monitor polls its Wheel Speed Sensor to update its internal speed variable and recalculates the wheel acceleration.
  - This Wheel Monitor detects rapid deceleration on the wheel and issues a decreasePressure() signal to its Gnome.
  - The Gnome issues an openReliefValve() signal to the Relief Valve causing it to open and a closeBrakeValve() signal to the Main Brake Valve causing it to close.

• The system then resumes the Active Monitoring Sequence, with the relief valve now open on the wheel, causing pressure to decrease on that wheel.

• Stabilized Pressure, Begin Holding Behavior - When rapid deceleration is no longer detected on the wheel, the Wheel Monitor signals to stabilize pressure on the wheel to “ride” the maximum allowable brake pressure.
  - The Timer signals the Manager to update with managerUpdate().
  - The Manager signals each Wheel Monitor to update with wheelMonitorUpdate().
  - The Wheel Monitors update their internal current speed variable by calling getSpeed() on their Wheel Speed Sensors.
  - The Manager recalculates the system speed.
  - Each Wheel Monitor updates its current acceleration by executing calcAccel().
  - Since the wheel is no longer undergoing rapid deceleration, the Wheel Monitor signals the Gnome with holdPressure().
  - The Gnome signals the Relief Valve to close with closeReliefValve() and the Main Brake Valve to open with openBrakeValve().

• Holding Behavior - Executed every HOLD_INTERVAL until timeInHold exceeds HOLD_THRESHOLD, the Gnome signals the valves to quickly open and close to stabilize the pressure on the wheel.
  Other objects continued with the Active Monitoring Sequence at this time.
  - The Gnome signals the Main Brake Valve to close with closeBrakeValve() and the Relief Valve to open with openReliefValve().
  - The Gnome signals the Relief Valve to close with closeReliefValve() and the Main Brake Valve to open with openBrakeValve().
  - It repeats this behavior every HOLD_INTERVAL until timeInHold exceeds HOLD_THRESHOLD (see next sequence).

• Return to Normal Braking - Once the Wheel Monitor determines pressure has been held long enough on the wheel (timeInHold meets or exceeds HOLD_THRESHOLD), and there is no rapid deceleration detected, normal braking is restored.
- The Timer signals the Manager to update with managerUpdate().
- The Manager signals the Wheel Monitors to update with wheelMonitorUpdate().
- Each Wheel Monitor updates its internal speed variable with getSpeed() called on its Wheel Speed Sensor.
- The Manager recalculates the system speed.
- Each Wheel Monitor recalculates its acceleration, and rapid deceleration is not detected.
- The Wheel Monitor in question determines that timeInHold has reached or passed HOLD_THRESHOLD, and thus it signals the Gnome to return to normal operation with normalPressure().
- The Gnome signals the Main Brake Valve to open with openBrakeValve() and the Relief Valve to close with closeReliefValve().

5 Promela Specification for System

5.1 Model Checking and Promela Overview

Promela is a mark-up language used to input a model to the Spin model verifier.[2] Because of Promela’s history in modeling communication networks, the logical units of a Promela description are processes (“proctypes”) which communicate by passing messages across data channels.[3] By encoding the state diagrams for the system in Promela, Spin can be used to check liveness and safety properties of the UML model expressed with linear temporal logic by analyzing all possible execution paths through the model. Thus, Promela and Spin allow for verification of specific essential properties of the system. It is thus that the Promela encoding of the ABS state diagrams has been done with the intention of accurately encoding the state diagrams without optimization of the Promela code.

The Promela model presented below is an encoding of all major state diagrams of the ABS. The following conventions were used in this encoding:

- Each class with major behavior has been implemented as a proctype. States have been implemented as labeled loops within the proctype.
- Complex states have been fully decomposed, with all boundary-transitions applied to each of the simple states contained within the complex state. This was repeated recursively as needed.
- Instead of direct computation of wheel speed, this is simulated through through boolean flags for “rapid deceleration” and whether the system has reached the activation speed. Thus, the Wheel Speed Sensor class is not encoded here.
- Error conditions are simulated by boolean flags. The guarded functions to set and clear the isError boolean are not implemented here; the variable is global.
- Signals from actuators to actual automobile hardware, such as the open and close signals from the valves, are not encoded here.
5.2 Promela Model

5.2.1 Macros, Channels, and Booleans

The following excerpt shows the macros, communication channels, and globally-defined booleans that are used throughout the following proctype definitions.

```c
/* declarations, definitions and symbolic constants */
#define true 1
#define false 0

/* general signals */
mtype = {POWERON,POWEROFF};

/* new events associated with the timer */
mtype = {timerOn, timerOff, managerUpdate, timeout};

/* new events associated with the dash light */
mtype = {light, dim};

/* new events associated with the hard reset */
mtype = {resetABS};

/* new events associated with the brake valves */
mtype = {closeBrakeValve, openBrakeValve};

/* new events associated with the relief valves */
mtype = {closeReliefValve, openReliefValve};

/* new events associated with the manager */
mtype = {errorBroadcast, setBrake, releaseBrake, wheelMonitorActive, wheelMonitorIdle, wheelMonitorUpdate};

/* new events associated with the wheel gnome */
mtype = {normalPressure, decreasePressure, holdPressure};

/* boolean flags, to be set by init to exercise the model */
bool rapidDecelerationWheel1 = false;
bool rapidDecelerationWheel2 = false;
bool rapidDecelerationWheel3 = false;
bool rapidDecelerationWheel4 = false;
bool impossibleDifference = false;
bool someErrorCondition = false;
bool aboveActivationSpeed = false;

/* boolean flags used for monitoring activity */
bool isPowerOn = false;
```
bool isTimerOn = false;
bool isDashLightOn = false;
bool brakeValveOneOpen = true;
bool brakeValveTwoOpen = true;
bool brakeValveThreeOpen = true;
bool brakeValveFourOpen = true;
bool reliefValveOneOpen = false;
bool reliefValveTwoOpen = false;
bool reliefValveThreeOpen = false;
bool reliefValveFourOpen = false;
bool isError = false;
bool isBrakeSet = false;

/* input channels to all of our 'classes' */
chan timer_in = [0] of {mtype};
chan dash_light_in = [0] of {mtype};
chan hard_reset_in = [0] of {mtype};
chan brake_valve_1_in = [0] of {mtype};
chan brake_valve_2_in = [0] of {mtype};
chan brake_valve_3_in = [0] of {mtype};
chan brake_valve_4_in = [0] of {mtype};
chan relief_valve_1_in = [0] of {mtype};
chan relief_valve_2_in = [0] of {mtype};
chan relief_valve_3_in = [0] of {mtype};
chan relief_valve_4_in = [0] of {mtype};
chan wheel_monitor_1_in = [0] of {mtype};
chan wheel_monitor_2_in = [0] of {mtype};
chan wheel_monitor_3_in = [0] of {mtype};
chan wheel_monitor_4_in = [0] of {mtype};
chan wheel_gnome_1_in = [0] of {mtype};
chan wheel_gnome_2_in = [0] of {mtype};
chan wheel_gnome_3_in = [0] of {mtype};
chan wheel_gnome_4_in = [0] of {mtype};
chan manager_in = [0] of {mtype};
chan broadcast_in = [0] of {mtype};
chan wheel_broadcast_in = [0] of {mtype};

5.2.2 Broadcast Protypes

To simulate behavior of a broadcast message, the allBroadcast and wheelMonitorBroadcast proctype echo any input they receive to all proctypes and to all wheel monitor proctypes, respectively.

/* broadcast process -- it replicates a message it receives to all communication channels */
active proctype allBroadcast()
{
    /* our input variable */
mtype channelInput;

do
::broadcast_in?channelInput; /* get the message */
    /* now, send it out on every channel */
    manager_in!channelInput;
    timer_in!channelInput;
    dash_light_in!channelInput;
    hard_reset_in!channelInput;
    brake_valve_1_in!channelInput;
    brake_valve_2_in!channelInput;
    brake_valve_3_in!channelInput;
    brake_valve_4_in!channelInput;
    relief_valve_1_in!channelInput;
    relief_valve_2_in!channelInput;
    relief_valve_3_in!channelInput;
    relief_valve_4_in!channelInput;
    wheel_monitor_1_in!channelInput;
    wheel_monitor_2_in!channelInput;
    wheel_monitor_3_in!channelInput;
    wheel_monitor_4_in!channelInput;
    wheel_gnome_1_in!channelInput;
    wheel_gnome_2_in!channelInput;
    wheel_gnome_3_in!channelInput;
    wheel_gnome_4_in!channelInput;
    od
}

/* wheel monitor broadcast process --
 it sends a message it receives to all four of the wheel monitors */
active proctype wheelMonitorBroadcast()
{
    /* our input variable */
    mtype channelInput;

do
::wheel_broadcast_in?channelInput; /* get the message */
    /* now, send it out to every wheel monitor */
    wheel_monitor_1_in!channelInput;
    wheel_monitor_2_in!channelInput;
    wheel_monitor_3_in!channelInput;
    wheel_monitor_4_in!channelInput;
    od
}
5.2.3 Manager Prototype

The manager proctype, based on the Manager class state diagram (Figure 5), is included below.

/* the manager */
/* the manager receives signals on the manager_in channel: 
POWERON - the power was turned on
POWEROFF - the power was turned off
managerUpdate - signal from timer to update
setBrake - signal that brake has been set
releaseBrake -- signal that brake has been released.

the manager sends the following signals:
light to dash light by dash_light_in
dim to dash light by dash_light_in
errorBroadcast to wheelMonitorBroadcast by wheel_broadcast_in
timerOn to timer by timer_in
wheelMonitorActive to wheelMonitorBroadcast
wheelMonitorIdle to wheelMonitorBroadcast
wheelMonitorUpdate to wheelMonitorBroadcast

the manager uses the following global variables:
someErrorCondition
isError
impossibleDifference
aboveActivationSpeed

*/

active proctype manager()
{
/* defense technique: always take what's in the channel --
ignore it if there's junk, otherwise act appropriately */
imtype channelInput;

/* power off pseudostate */
power_off:
printf("Manager in Power Off state.\n");
do
::manager_in?channelInput;
  if
    ::(channelInput == POWERON); goto testing
    ::(channelInput != POWERON);
  fi;
 OD;
/* Testing state */
testing:
printf("Manager in Testing state.\n");
/* turn the dash light on */
dash_light_in!light;
/* do the self check */
if
    /* if there’s some error condition, we have an error */
    ::(someErrorCondition == true); isError = true;
    /* if there’s an impossible difference, we have an error */
    ::(impossibleDifference == true); isError = true;
    /* if neither is true, do nothing */
    ::(someErrorCondition != true) && (impossibleDifference != true);
fi;
/* now, branch, based on whether isError is true */
if
    /* yes, there’s an error -- send the wheel monitors and error broadcast 
    and go to the error state */
    ::(isError == true); wheel_broadcast_in!errorBroadcast; goto error;
    /* no, there’s no error -- turn the dash light off and start watching */
    ::(isError != true); dash_light_in!dim; goto watching;
fi;
/* error state */
error:
printf("Manager has entered the error state.\n");
/* we’ve simplified the loop here because the busy wait 
   would tie up spin... the transition back to error is 
   ignored in favor of just a transition out on !isError*/
do
    /* if the error clears up, we transition out */
    :: (isError != true); goto testing;
    /* if the power turns off, we leave */
    :: manager_in?channelInput;
    if
        /* power is off */
        ::(channelInput == POWEROFF); goto power_off
        /* otherwise, it falls on the floor */
        ::(channelInput != POWEROFF);
    fi;
od;
/* watching state */
watching:
printf("Manager is in the Watching state.\n");
do
::manager_in?channelInput;
  if
    /* power is turned off ... */
    ::(channelInput == POWEROFF); goto power_off
    /* brake is set */
    ::(channelInput == setBrake); goto brakeTest
    /* otherwise, it falls on the floor */
    ::((channelInput != POWEROFF) && (channelInput != setBrake));
  fi;
  od;

/* Brake Test State */
brakeTest:
  printf("Manager is in the Brake Test state.\n");
  /* see if we have an impossible difference or an error condition. */
  /* if yes, we go to error, if not, we go to Braking */
  if
    /* if there's some error condition, we have an error */
    ::(someErrorCondition == true); isError = true;
    /* if there's an impossible difference, we have an error */
    ::(impossibleDifference == true); isError = true;
    /* if neither is true, do nothing */
    ::((someErrorCondition != true) && (impossibleDifference != true));
  fi;

  /* now, branch, based on whether isError is true */
  if
    /* yes, there's an error -- send the wheel monitors and error broadcast
       and go to the error state */
    ::(isError == true); wheel_broadcast_in!errorBroadcast; goto error;
    /* no, there's no error -- turn the dash light off and start braking */
    ::(isError != true); dash_light_in!dim; timer_in!timerOn;
    wheel_broadcast_in!wheelMonitorActive; goto braking;
  fi;

/* Braking State */
braking:
  printf("Manager is in the Braking state.\n");
  /* first, if we drop below activation speed, we're done */
  do
    /* if we are below the activation speed, we have nothing to monitor */
    /* turn the wheel monitors off */
    ::(aboveActivationSpeed == false); timer_in!timerOff;
    wheel_broadcast_in!wheelMonitorIdle; goto watching
    /* if we're above activation speed, wait for some input... */
    ::(aboveActivationSpeed == true); manager_in?channelInput;
  if
/* if the power is turned off, there isn’t much to do */
(channelInput == POWEROFF); goto power_off
/* if the brake is released, turn the wheel monitors and timer off
and go back to watching */
(channelInput == releaseBreak); timer_in!timerOff;
wheel_broadcast_in!wheelMonitorIdle; goto watching
/* if the update signal is received, signal all the wheel monitors
to update in turn */
(channelInput == managerUpdate);
wheel_broadcast_in!wheelMonitorUpdate
/* otherwise, it falls on the floor */
((channelInput != POWEROFF) && (channelInput != releaseBreak) &&
(channelInput != managerUpdate));
fi;
o;
}

5.2.4 Wheel Monitor Proctype

There are four wheel monitor proctypes instantiated in the model, one for each wheel. Below
is the encoding for “wheel_monitor_1.” This is based on the Wheel Monitor class state diagram
(Figure 7).

/* wheel monitor 1 */
/* receives these signals on wheel_monitor_1_in:
POWEROFF -- the power is turned off
wheelMonitorUpdate -- time to update
wheelMonitorActive -- transition to active monitoring
wheelMonitorIdle -- stop active monitoring, return to idle
errorBroadcast -- return to idle state, set brake valves to default
(all-ABS)
sends these signals:
(all to the wheel cylinder gnome)
 normalPressure
decreasePressure
holdPressure

*/
active proctype wheel_monitor_1()
{
/* defense technique: always take what’s in the channel --
ignore it if there’s junk, otherwise act appropriately */
mttype channelInput;

43
/* we'll need to keep count of our time in hold like this */
/* we will only remain in hold for up to ten updates */
int timeInHold = 0;
int maxTimeInHold = 10;

/* power off pseudostate */
power_off:
printf("Wheel Monitor 1 in Power Off state.\n");
do
::wheel_monitor_1_in?channelInput;
if
/* turn it on */
::(channelInput == POWERON); goto Idle
/* anything else doesn't matter here */
::(channelInput != POWERON);
fi;

/* Idle state */
Idle:
printf("Wheel Monitor 1 in Idle state.\n");
do
::wheel_monitor_1_in?channelInput;
if
/* receive power off signal */
::(channelInput == POWEROFF); goto power_off
/* receive errorBroadcast signal -- normal pressure on the gnome */
::(channelInput == errorBroadcast); wheel_gnome_1_in!normalPressure
/* receive active signal -- goto monitor state*/
::(channelInput == wheelMonitorActive); goto Monitor
/* otherwise, doesn't matter */
::((channelInput != POWEROFF) && (channelInput != errorBroadcast) &&
   (channelInput != wheelMonitorActive))
fi;

/* Monitor state */
Monitor:
printf("Wheel Monitor 1 in Monitor state.\n");
do
::wheel_monitor_1_in?channelInput;
if
/* receive power off signal */
::(channelInput == POWEROFF); goto power_off
/* receive error broadcast -- set gnome to normal */
::(channelInput == errorBroadcast); wheel_gnome_1_in!normalPressure;
timeInHold = 0; goto Idle
/* receive idle message -- set gnome to normal */
:(channelInput == wheelMonitorIdle); wheel_gnome_1_in!normalPressure;
  timeInHold = 0; goto Idle
/* receive update signal */
:(channelInput == wheelMonitorUpdate);
  if
    /* rapid deceleration! */
    :(rapidDecelerationWheel1 == true);
    wheel_gnome_1_in!decreasePressure;
    goto Skid
  /* no rapid deceleration */
  :(rapidDecelerationWheel1 == false);
  fi;
/* any other signal? we don’t care. */
:(((channelInput != POWEROFF) && (channelInput != errorBroadcast)
  && (channelInput != wheelMonitorIdle) && (channelInput !=
  wheelMonitorUpdate));
  fi;
  od;
/* Skid state */
Skid:
printf("Wheel Monitor 1 in Skid state.\n");
do
  ::wheel_monitor_1_in?channelInput;
    if
      /* receive power off signal */
      :(channelInput == POWEROFF); goto power_off
/* receive error broadcast -- set gnome to normal */
:(channelInput == errorBroadcast); wheel_gnome_1_in!normalPressure;
  timeInHold = 0; goto Idle
/* receive idle message -- set gnome to normal */
:(channelInput == wheelMonitorIdle); wheel_gnome_1_in!normalPressure;
  timeInHold = 0; goto Idle
/* receive update signal */
:(channelInput == wheelMonitorUpdate);
  if
    /* rapid deceleration! */
    :(rapidDecelerationWheel1 == true);
    /* no rapid deceleration */
    :(rapidDecelerationWheel1 == false); wheel_gnome_1_in!holdPressure;
    timeInHold = 0; goto Hold
    fi;
/* any other signal? we don’t care. */
:(((channelInput != POWEROFF) && (channelInput != errorBroadcast)

&& (channelInput != wheelMonitorIdle) && (channelInput !=
wheelMonitorUpdate));
fi;

od;

/* Hold state */
Hold:
printf("Wheel Monitor 1 in Hold state.\n");
do
::wheel_monitor_1_in?channelInput;
if
/* receive power off signal */
::(channelInput == POWEROFF); goto power_off
/* receive error broadcast -- set gnome to normal */
::(channelInput == errorBroadcast); wheel_gnome_1_in!normalPressure;
timeInHold = 0; goto Idle
/* receive idle message -- set gnome to normal */
::(channelInput == wheelMonitorIdle); wheel_gnome_1_in!normalPressure;
timeInHold = 0; goto Idle
/* receive update signal */
::(channelInput == wheelMonitorUpdate); timeInHold = timeInHold + 1;
if
/* rapid deceleration! */
::(rapidDecelerationWheel1 == true);
wheel_gnome_1_in!decreasePressure;
goto Skid
/* no rapid deceleration */
::(rapidDecelerationWheel1 == false);
if
/* if we’ve been here to long, leave */
::(timeInHold >= maxTimeInHold); goto Monitor
/* otherwise, stick around */
::(timeInHold < maxTimeInHold);
fi;
fi;
/* any other signal? we don’t care. */
::((channelInput != POWEROFF) && (channelInput != errorBroadcast)
&& (channelInput != wheelMonitorIdle) && (channelInput !=
wheelMonitorUpdate));
fi;
odo;
5.2.5 Wheel Cylinder Gnome Prototype

There are four wheel cylinder gnome protypes instantiated in the model, one for each wheel.
Below is the encoding for “wheel_gnome_1.” This is based on the Gnome class state diagram
(Figure 13).

```c
/* wheel gnome 1 */
/* receives the following signals on wheel_gnome_1_in:
  POWERON -- power turned on
  POWEROFF -- power turned off
  normalPressure - change to normal pressure state
  holdPressure - change to holding state
  decreasePressure - change to decreasing state
  timeout -- timeout if we are in the holding state

  sends the following signals:
  openReliefValve, closeReliefValve to relief_valve_1 by
    relief_valve_1_in
  closeBrakeValve, openBrakeValve to brake_valve_1 by
    brake_valve_1_in

  */
active proctype wheel_gnome_1()
{
  /* defense technique: always take what’s in the channel --
  ignore it if there’s junk, otherwise act appropriately */
  mtype channelInput;

  /* power off pseudostate */
  power_off:
  printf("Gnome 1 in Power Off state.\n");
  do
    ::wheel_gnome_1_in?channelInput;
    if
      /* power’s on...normal operation transition */
      ::(channelInput == POWERON); goto normal
    /* other messages fall to the floor */
    ::(channelInput != POWERON);
    fi;
  od;

  /* normal state */
  normal:
  printf("Gnome 1 in Normal state.\n");
  do
    ::wheel_gnome_1_in?channelInput;
    if
```
/* power is turned off -- nothing we can do here*/
{: (channelInput == POWEROFF); goto power_off
/* decrease pressure signal -- open the relief valve,
close the brake valve*/
{: (channelInput == decreasePressure);
   relief_valve_1_in!openReliefValve;
   brake_valve_1_in!closeBrakeValve;
goto decreasing
/* hold pressure signal */
{: (channelInput == holdPressure); goto holding
/* normal pressure signal */
{: (channelInput == normalPressure); brake_valve_1_in!openBrakeValve;
   relief_valve_1_in!closeReliefValve;
/* anything else, we fall to the floor */
{: ((channelInput != POWEROFF) && (channelInput !=
   decreasePressure) && (channelInput != holdPressure) &&
   (channelInput != normalPressure));
   fi;
   od;
/* decreasing state */
decreasing:
   printf("Gnome 1 in Decreasing State.\n");
do
   ::wheel_gnome_1_in?channelInput;
   if
   /* power is turned off -- nothing we can do here */
   {: (channelInput == POWEROFF); goto power_off
   /* normal pressure signal -- close relief valve,
open brake valve */
   {: (channelInput == normalPressure);
      brake_valve_1_in!openBrakeValve;
      relief_valve_1_in!closeReliefValve;
goto normal
/* hold pressure signal */
{: (channelInput == holdPressure); goto holding
/* anything else doesn't matter to us now */
{: ((channelInput != POWEROFF) && (channelInput !=
   normalPressure) && (channelInput != holdPressure));
   fi;
   od;
/* holding state */
holding:
   printf("Gnome 1 in Holding State.\n");
/* holding state 1 */
holding1:
printf("Gnome 1 in Holding 1 State\n");
/* tell brake valve to open */
brake_valve_1_in!openBrakeValve;
/* tell relief valve too close */
relief_valve_1_in!closeReliefValve;
/* wait for the signal */
do
::wheel_gnome_1_in?channelInput;
if
    /* we got a timeout */
    ::(channelInput == timeout); goto holding2
    /* we might be told to normalize pressure */
    ::(channelInput == normalPressure);
    brake_valve_1_in!openBrakeValve;
    relief_valve_1_in!closeReliefValve; goto normal
    /* or perhaps to decrease the pressure */
    ::(channelInput == decreasePressure);
    relief_valve_1_in!openReliefValve;
    brake_valve_1_in!closeBrakeValve; goto decreasing;
    /* or, perhaps even power off */
    ::(channelInput == POWEROFF); goto power_off
    /* but those are the signals we care about */
    ::((channelInput != timeout) && (channelInput != POWEROFF) &&
    (channelInput != normalPressure)
    && (channelInput != decreasePressure));
fi;
od;
/* holding state 2 */
holding2:
printf("Gnome 1 in Holding 2 State\n");
/* tell relief valve to open */
relief_valve_1_in!openReliefValve;
/* tell brake valve to close */
brake_valve_1_in!closeBrakeValve;
/* wait for the signal */
do
::wheel_gnome_1_in?channelInput;
if
    /* we got a timeout */
    ::(channelInput == timeout); goto holding1
    /* we might be told to normalize pressure */
    ::(channelInput == normalPressure);
    brake_valve_1_in!openBrakeValve;
    relief_valve_1_in!closeReliefValve; goto normal
    /* or perhaps to decrease the pressure */
    ::(channelInput == decreasePressure);
relief_valve_1_in!openReliefValve;
brake_valve_1_in!closeBrakeValve; goto decreasing;
    /* or, perhaps even power off */
    :((channelInput == POWEROFF); goto power_off
    /* but those are the signals we care about */
    :((channelInput != timeout) && (channelInput != POWEROFF) &&
        (channelInput != normalPressure)
        && (channelInput != decreasePressure));
    fi;
    od;
}

5.2.6 Brake Valve Proctype

There are four main brake valve proctypes instantiated in the model, one for each wheel. Below is the encoding for “brake_valve_1.” This is based on the Main Brake Valve class state diagram (Figure 11).

/* brake valve 1 */
/* receives on brake_valve_1_in:
   POWERON - the power is on
   POWEROFF - the power is turned off
closeBrakeValve -- changes to closed state
openBrakeValve -- changes to open state
uses global brakeValveOneOpen
*/
active proctype brake_valve_1()
{
    /* defense technique: always take what’s in the channel --
       ignore it if there’s junk, otherwise act appropriately */
    mtype channelInput;

    /* power off pseudostate */
    power_off:
    printf("Brake Valve 1 in Power Off State.\n");
    do
        ::brake_valve_1_in?channelInput;
        if
            ::(channelInput == POWERON); brakeValveOneOpen = true; goto open
            ::(channelInput != POWERON);
            fi
    od;

    /* open state */
    open:
    printf("Brake Valve 1 in Open state.\n");
do
  ::brake_valve_1_in?channelInput;
  if
    /* brake valve close signal received */
    ::(channelInput == closeBrakeValve); brakeValveOneOpen = false;
    goto closed
    /* power off received. note we CANNOT change the valve now */
    ::(channelInput == POWEROFF); goto power_off
    /* ignore any other signal */
    ::((channelInput != closeBrakeValve) & (channelInput != POWEROFF));
    fi;
  od;

/* closed state */
closed:
  printf("Brake Valve 1 in Closed state.\n");
do
  ::brake_valve_1_in?channelInput;
  if
    /* brake valve open signal received */
    ::(channelInput == openBrakeValve); brakeValveOneOpen = true; goto open
    /* power off received. note we CANNOT change the valve now */
    ::(channelInput == POWEROFF); goto power_off
    /* ignore any other signal */
    ::((channelInput != openBrakeValve) & (channelInput != POWEROFF));
    fi;
  od;
}

5.2.7 Relief Valve Proctype

There are four relief valve proctypes instantiated in the model, one for each wheel. Below is the
encoding for "relief_valve_1." This is based on the Relief Valve class state diagram (Figure 12).

/* relief valve 1 */
/* receives on relief_valve_1_in:
  POWERON -- the power is turned on
  POWEROFF -- the power is turned off
  closeReliefValve -- changes to closed state
  openReliefValve -- changes to open state
  uses gloval reliefValveOneOpen */
active proctype relief_valve_1()
{
  /* defense technique: always take what’s in the channel --
  ignore it if there’s junk, otherwise act appropriately */
mtype channelInput;

/* power off pseudostate */
power_off:
printf("Relief Valve 1 in Power Off state\n");
do
::relief_valve_1_in?channelInput;
    if
        :(channelInput == POWERON); reliefValveOneOpen = false; goto closed
        :(channelInput != POWERON);
        fi;
    od;

/* closed */
closed:
printf("Relief Valve 1 in Closed state\n");
do
::relief_valve_1_in?channelInput;
    if
        /* relief valve open signal received */
        :(channelInput == openReliefValve); reliefValveOneOpen = true;
        goto open
        /* power off received. note we CANNOT change the valve now */
        :(channelInput == POWEROFF); goto power_off
        /* some other signal received -- ignore it */
        :(channelInput != openReliefValve) && (channelInput != POWEROFF));
        fi;
    od;

/* open state */
open:
printf("Relief Valve 1 in Open state\n");
do
::relief_valve_1_in?channelInput;
    if
        /* relief valve close signal received */
        :(channelInput == closeReliefValve); reliefValveOneOpen = false;
        goto closed
        /* power off received */
        :(channelInput == POWEROFF); goto power_off
        /* some other signal received -- ignore it */
        :(channelInput != closeReliefValve) && (channelInput != POWEROFF));
        fi;
    od;

}
5.2.8 Dash Light Proctype

The dash light proctype is based on the Dash Light class state diagram (Figure 8).

```c
/* dash light object */
/* receives as input on dash_light_in: */
  POWERON - power is turned on
  POWEROFF - power is turned off
  dim - turn dash light off
  light - turn dash light on
uses global: isDashLightOn
*/
active proctype dash_light()
{
  /* defense technique: always take what’s in the channel --
  ignore it if there’s junk, otherwise act appropriately */
  mtype channelInput;

  /* power off pseudostate */
  power_off:
  printf("Dash Light in Power Off state.\n");
do /* only leave power off state if we get power on */
  ::dash_light_in?channelInput;
    if
      /* power is on, but dash light is off... */
      (channelInput == POWERON); goto dash_light_off;
    /* throw away other messages */
    (channelInput != POWERON);
  fi;
  od;

  /* power on, dash light off */
  dash_light_off:
  printf("Dash Light in Off state.\n");
do
  ::dash_light_in?channelInput;
    if
      /* power is turned off */
      (channelInput == POWEROFF); goto power_off
    /* dash light is turned on */
      (channelInput == light); isDashLightOn = true; goto dash_light_on
    /* throw away other messages */
    (channelInput != light) && (channelInput != POWEROFF));
  fi;
  od;
```

53
5.2.9 Timer Proctype

The timer proctype is based on the Timer class state diagram (Figure 6).

/* timer object, there is one of these */
/* receives as input on timer_in: */
/* POWERON -- power is turned on for car */
/* POWEROFF -- power is turned off for car */
timerOn -- turns timer on

timerOff -- turns timer off

timeout -- timeout is reached

sends messages out:
managerUpdate on manager_in channel to update manager

uses global: isTimerOn */
active proctype timer()
{
    /* defense technique: always take what’s in the channel -- */
    /* ignore it if there’s junk, otherwise act appropriately */
    messageType channelInput;

    /* not yet received power on signal pseudostate */
timer_power_off:
    printf("Timer in Power Off state.\n");
do
      ::timer_in?channelInput;
      if
        /* power is on, continue to timer_off */
        ::(channelInput == POWERON); goto timer_off
        /* if the power isn’t on, why do we get a message ? */
```c
    :=(channelInput != POWERON);
    fi;
  od;

  /* timer off state */
timer_off:
  printf("Timer in Idle state\n");
do
    ::=timer_in?channelInput;
    if
      /* they've turned the timer on! */
      ::=channelInput == timerOn); isTimerOn = true; goto timer_on
      /* car is turned off */
      ::=channelInput == POWEROFF); goto timer_power_off;
      /* ignore junk in the channel */
      ::=((channelInput != timerOn) && (channelInput != POWEROFF))
        fi;
    od;

  /* timer on state */
timer_on:
  printf("Timer in Active state\n");
do
    ::=timer_in?channelInput;
    if
      /* need to signal the manager to update */
      ::=channelInput == stimeout); manager_in!managerUpdate
      /* they've turned me off! */
      ::=channelInput == timerOff); isTimerOn = false;
        goto timer_off
      /* they've turned the vehicle off */
      ::=channelInput == POWEROFF); isTimerOn = false;
        goto timer_power_off
      /* unknown signal -- throw away! */
      ::=((channelInput != stimeout) && (channelInput != timerOff)
        && (channelInput != POWEROFF))
        fi;
    od;
  }
```

5.2.10 Hard Reset Prototype

The hard reset proctype is based on the Hard Reset class state diagram (Figure 9).

/* hard reset switch for the system */
/* receives signals on hard_reset_in:
    POWERON - when power comes on

POWEROFF - when power goes off
resetABS - sets isError to false

uses global variable isError
*/
active proctype hard_reset()
{
    /* defense technique: always take what's in the channel --
       ignore it if there's junk, otherwise act appropriately */
    mtype channelInput;

    /* power off, nothing's going on */
    power_off:
    printf("Hard Reset in Power Off state.\n");
    do
        ::hard_reset_in?channelInput;
        if
            /* power is turned on */
            ::(channelInput == POWERON); goto reset_ready;
            /* let the other message fall on the floor */
            ::(channelInput != POWERON);
            fi;
        od;
    /* power is on, ready to reset */
    reset_ready:
    printf("Hard Reset is in Ready state.\n");
    do
        ::hard_reset_in?channelInput;
        if
            /* power is turned off */
            ::(channelInput == POWEROFF); goto power_off
            /* reset signal caught */
            ::(channelInput == resetABS); isError = false;
            /* otherwise let it fall on the floor */
            ::((channelInput != POWEROFF) && (channelInput != resetABS));
            fi;
        od;
}

5.3 Sequence Diagram Validation

Promela and Spin were used to validate the sequence diagrams presented above. First, the
sequence of events for normal braking above ABS activation speed (Figure 15), was simulated
with the following init procedure. The counter was used to ensure that the "Active Monitoring
Sequence" was executed for any given simulation.
/* initialization procedure for sequence 1 */
init {
    /* a counter */
    int count = 0;

    /* turn the power on */
broadcast_in!POWERON;
isPowerOn = true;

    /* we're above activation speed */
aboveActivationSpeed = true;

    /* set the brake */
broadcast_in!setBrake;
isBrakeSet = true;

    /* simulate timer signals -- undetermined number of times */
do
        ::timeout;
        if ::broadcast_in!timeout; count = count + 1;
        ::(count > 5); break
    fi;
    od;

    /* release the brake */
broadcast_in!releaseBrake;
isBrakeSet = false;

    /* turn the power off */
broadcast_in!POWEROFF;
isPowerOn = false;
}

Then, the sequence diagram for normal braking leading to a drop below activation speed (Figure 16) was simulated with the following init procedure. The counter was again used.

/* initialization procedure */
init {
    /* a counter */
    int count = 0;

    /* turn the power on */
broadcast_in!POWERON;
isPowerOn = true;
/* we're above activation speed */
aboveActivationSpeed = true;
/* set the brake */
broadcast_in!setBrake;
isBrakeSet = true;
/* simulate timer signals -- undetermined number of times */
do :
:timeout; broadcast_in!timeout; count = count + 1;
:(count > 5); break
od;
/* speed drops below the threshold */
aboveActivationSpeed = false;
/* we repeat the behavior of the active monitoring sequence,
   this time, with no events occurring at the wheel monitors */
count = 0;
do :
:timeout; broadcast_in!timeout; count = count + 1;
:(count > 5); break
od;
/* release the brake */
broadcast_in!releaseBrake;
isBrakeSet = false;
/* turn the power off */
broadcast_in!POWEROFF;
isPowerOn = false;
}

Finally, the sequence diagram for rapid deceleration detection and recovery on one wheel (Figure 17) was simulated with the following init procedure similarly as the previous two sequence diagrams.

/* initialization procedure */
init {
/* a counter */
int count = 0;
/* turn the power on */
broadcast_in!POWERON;
isPowerOn = true;

/* we're above activation speed */
aboveActivationSpeed = true;

/* set the brake */
broadcast_in!setBrake;
isBrakeSet = true;

/* simulate timer signals -- undetermined number of times */
do
  ::timeout; broadcast_in!timeout; count = count + 1;
  ::(count > 5); break
od;

/* rapid deceleration starts on wheel 1 */
rapidDecelerationWheel1 = true;

/* we repeat the behavior of the active monitoring sequence */
count = 0;
do
  ::timeout; broadcast_in!timeout; count = count + 1;
  ::(count > 5); break
od;

/* rapid deceleration ends */
rapidDecelerationWheel1 = false;

/* we repeat the behavior of the active monitoring sequence */
count = 0;
do
  ::timeout; broadcast_in!timeout; count = count + 1;
  ::(count > 5); break
od;

/* release the brake */
broadcast_in!releaseBrake;
isBrakeSet = false;

/* turn the power off */
broadcast_in!POWEROFF;
isPowerOn = false;

}
References

