EDA122 Fault-Tolerant Computer Systems

Laboratory Class 1

2016

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
GÖTEBORG 2016

1. Introduction

The aim of this laboratory class is to show how dependability modelling can be used to compare different design solutions for a fault-tolerant system. We will compare two candidate architectures for a brake-by-wire system.

Brake-by-wire systems are expected to replace hydraulic brake systems in future road vehicles. In a brake-by-wire system, the driver's brake intention is transmitted electronically from the brake pedal to electro-hydraulic or electro-mechanical brake actuators positioned at each wheel. Potential advantages of brake-by-wire systems compared to traditional hydraulic systems include lower cost, lower weight and simpler integration with stability control and active safety systems.

Figure 1 shows an overview of the brake-by-wire system. The brake pedal is connected to a central unit (CU). When the driver presses the brake pedal, the central unit sends messages containing brake commands to each of the four wheel units (WU). To ensure the safety of the system, the central unit and the wheel units must be fault-tolerant. Both the central unit and the wheel units are therefore implemented using redundant computer modules (see Figure 3 and 4). Two serial busses (SB1 and SB2) are provided to ensure fault-tolerance for the data communication.

To optimize brake performance, the system executes a closed loop anti-lock braking control algorithm for each wheel. The inputs to the anti-lock control program consist of the current wheel speed and brake force commands generated by a stability control program, which is executed in the central unit. The wheel speed is measured by a wheel speed sensor included in each wheel unit. The input to the stability control program consists of data from several sensors. These sensors measures the position of the brake-pedal (the driver's brake intention), the angle of the steering wheel (the driver's intended direction), the yaw rate (the rotation rate of the vehicle around its y-axis), the roll rate, and the vehicle's lateral and longitudinal acceleration. In this laboratory class, we will focus our attention on where to execute the anti-lock-braking algorithms. We will *not* consider the reliability of the sensors used by stability control program.

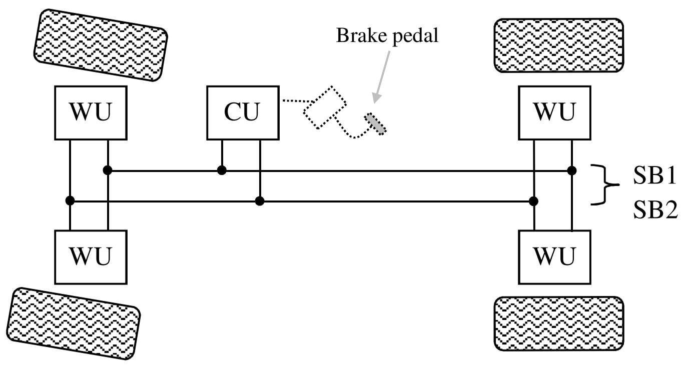

Figure 1. Brake-by-wire system

We will investigate two possible implementations of the system. In the first approach, the anti-lock control algorithms are executed locally in the wheel units. The complexity of the wheel units is in this design approximately the same as that of the central unit. However, the failure rate of the wheel units is higher than for the central unit, as they are more exposed to vibrations, moisture and temperature cycling. We call this design approach *the distributed architecture.*

The second approach is to execute the control algorithm for each wheel in the central unit, and let the wheel units consist of simple interfaces to the actuators and sensors. In this case, the control loops for anti-lock braking are closed over the communication network. The advantage with this approach is that the wheel units contain less hardware since they essentially consist of a communication interface, which sends data from the wheel speed sensor and receives commands to the brake actuator. Thus, the failure rate of the wheel units is lower compared to the other design. On the other hand, the failure rate of the central unit is higher, since it requires more processing power and more memory. We call this design approach *the centralized architecture*, since all control law calculations are performed by the central unit.

2. System description

The brake-by-wire system consists of one central unit (CU), four wheel units (WU 1 to 4) and two system buses (SB1 and SB2), as shown Figure 1. The system has two modes of operation: *full functionality* and *degraded functionality*. To provide full functionality, the central unit, all four wheel units, and at least one system bus must be working. In degraded mode, the system has lost the ability to issue brake commands to one wheel. This means that the central unit, three wheel units, and at least one system bus must be working in order for the system to provide degraded functionality. To simplify the analysis, we consider it to be a catastrophic failure if the system is unable to send brake commands to two or more wheels.

2.1 Wheel units (WU)

The wheel units consist of the following subunits: two fail-silent computer modules (CMs), two sensors (S), one actuator (A) and four bus interfaces (BI), see Figure 2. We assume that the coverage is 100% for all errors affecting the wheel unit. The failure rates for the sensors, the actuator, and the computer modules are given below in Table 1 for the distributed architecture and in Table 2 for the centralized architecture. The computer modules used in the distributed architecture have higher complexity than those used in the centralized architecture.

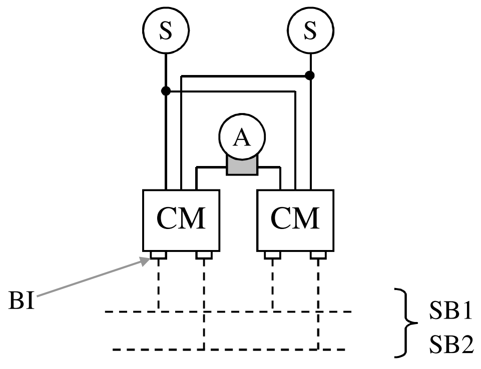

Figure 2. Wheel unit

The failure rate of the computer modules in the distributed architecture is therefore higher than the failure rate of the computer modules in the centralized architecture. The failure rates of the sensors and the actuator do not vary between the two architectures, since they use the same sensors and the same actuator.

2.2 Central unit (CU)

We will consider two different fault-tolerant configurations for the central unit: a *Duplex* configuration for *the distributed architecture* and a *Triplex/Duplex* configuration for *the centralized architecture*.

Duplex: This configuration consists of two fail-silent computer modules that operate in active redundancy, see Figure 3. If an error is detected in a computer module it stops producing results. The central unit then maintains its operation by using the results from the remaining non-faulty module. Undetected errors in the CMs may lead to violations of the fail-silent property. We consider such fail-silent violations to constitute a failure of the central unit. The coverage factor for the fail-silent property is 99%.

Triplex/Duplex: This configuration consists of three fail-silent computer modules operating in active redundancy, see Figure 4. The computer modules send redundant messages to the wheel units, which perform a majority vote on the messages to mask errors. We assume that the coverage for errors occurring in the computer modules is 100% as long as three computer modules are operational. The computer modules execute a so called *redundancy management protocol*, which ensures that any computer module which produces a result that deviates from the results produced by the other two computer modules is shut down. After the first failure of a computer module, the system is reconfigured to a duplex system. The error detection coverage is 99% in the duplex mode. The Triplex/Duplex system can thus tolerate the loss of up to two computer modules.

Each computer module contains a CPU, memory, I/O circuits and other hardware components. We assume that the failure rate of the bus interface (BI) is negligible compared to the failure rate of the other hardware components. For both architectures, the occurrence of a non-covered fault in a computer module leads to a failure of the central unit, and hence to a system failure.

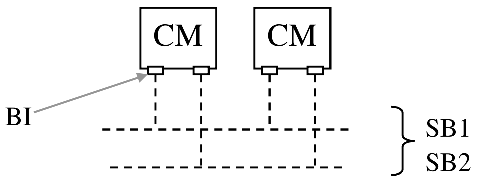

Figure 3. Duplex configuration

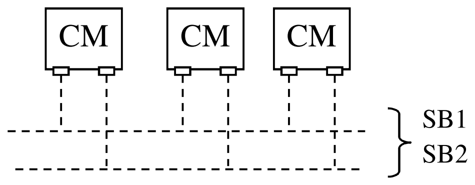

Figure 4. Triplex/Duplex configuration

3. Analysis

Use the SHARPE program to analyse and compare the two architectures. SHARPE provides a command language for building models and performing calculations. This command language will be explained at one of the exercises. You can also read about it in the SHARPE Manual and the SHARPE User Guide, which are available on the course homepage.

3.1 Problem A

Compare the reliability of the distributed and the centralized architectures after two years. Compare also their MTTF. Use failure rates and coverage factors from Table 1 and Table 2. Discuss within the group the pros and cons of the two designs. Which architecture is the best one? Where are the reliability bottlenecks?

Compare the architectures for two levels of functionality:

1. Reliability with respect to full functionality (four wheels operational).
2. Reliability with respect to degraded functionality (three wheels operational).

Solve the problem in the following order:

1. Analyze the reliability of the wheel unit subsystem.
    a. Create a reliability block diagram for one wheel unit in SHARPE. Calculate and compare the reliability of the wheel units for the two architectures. (The same model can be used in both cases, but the failure rates are different for the computer modules.)
    b. Create a fault tree to calculate the reliability and the MTTF for the wheel unit subsystem, i.e. all four wheel units, with respect to both full and degraded functionality. Compare the reliability and the MTTF of the wheel unit subsystem for the two architectures.
2. Create Markov models of the central unit for the two architectures. Calculate and compare the reliability and the MTTF of the central unit.
   (*continued*
3. Use a fault tree to calculate the reliability and the MTTF for the entire system, and compare the two architectures with respect to both full and degraded functionality.

Hints:

Create two SHARPE files, one for each architecture, and gradually develop the models in these files following the procedure described above. Use the gnuplot program to plot the reliability curves. Plotting the curves is easier if the models are in separate files.

To print the reliability after 2 years use **expr 1-value(17520; system)** in SHARPE. To plot the reliability use the SHARPE command **eval (system) 0 35040 730**. This command lists the *failure* probability of **system** over 4 years (4x8760= 35040 hours) in one month (= 730 hours) steps. (We approximate a year to be 365x24 = 8760 hours disregarding the leap years.) To convert the failure probability into reliability use the utility program **complement**, as shown in Appendix A. Then use gnuplot to plot the data file produced by **complement**.

Note that SHARPE does not allow underscore characters ('_') in the names of constants and that there must be a carriage return after the final 'end' command given to SHARPE.

3.2 Problem B

Compare the operation time for the two configurations for a reliability of 0.92. Consider only the degraded mode. How much higher can the failure rate of the CMs be in the centralized architecture in order to achieve the same operation time as for the distributed architecture?

Only the failure rate of the CMs in the central unit for the centralized architecture should be varied. (Do not change the failure rate of the wheel-nodes.)

3.3 Modelling parameters

Table 1 and 2 show the failure rates and coverage factors to be used for the distributed architecture and the centralized architecture, respectively.

Table1: Failure rates and coverage factors for the Distributed Architecture

Subsystem	**Part**	**Failure rate (λ)**	**Coverage**
System bus	Serial bus	$5 \bullet 10^{-7}$ [f/h]	1
Wheel unit	Computer module	$15 \bullet 10^{-6}$ [f/h]	1
	Sensor	$2 \bullet 10^{-6}$ [f/h]	1
	Actuator	$1 \bullet 10^{-6}$ [f/h]	1
Central unit	Computer module	$8 \bullet 10^{-6}$ [f/h]	0.99

Table 2: Failure rates and coverage factors for the Centralized Architecture

Subsystem	**Part**	**Failure rate (λ)**	**Coverage**
System bus	Serial bus	$5 \bullet 10^{-7}$ [f/h]	1
Wheel unit	Computer module	$10 \bullet 10^{-6}$ [f/h]	1
	Sensor	$2 \bullet 10^{-6}$ [f/h]	1
	Actuator	$1 \bullet 10^{-6}$ [f/h]	1
Central unit	Computer modules	$10 \bullet 10^{-6}$ [f/h]	First CM failure:1 Second CM failure: 0.99

4. Laboratory approval

You must show and explain your results to a teaching assistant in order to have the lab approved. Make sure that a teaching assistant have noted your approval before you leave the lab session.

Appendix A - Formatting the output from SHARPE

Command	Description
eval_filter	Input from standard input, output to standard output. Used for formatting the output from SHARPE, when output is the result of the **eval** command.
complement	Input from standard input, output to standard output. Assumes that input consists of tuples, e.g., (x,y), plots x and 1-y.

Example:

Use the following command to format the output from SHARPE so that gnuplot can be used to plot the reliability, R(t).

```
sharpe  file.sharpe | eval_filter | complement > file.dat
```

Appendix B - Summary of gnuplot

The gnuplot program for Windows is called **wgnuplot.exe**.

The following text will be printed when the program starts:

```
G N U P L O T
Version 4.1 patchlevel 0
last modified Sat Jul  3 00:04:32 CEST 2004
System: MS-Windows 32 bit

Copyright (C) 1986 - 1993, 1998, 2004
Thomas Williams, Colin Kelley and many others

Type `help` to access the on-line reference manual.
The gnuplot FAQ is available from
http://www.gnuplot.info/faq/

Send comments and requests for help to
      <gnuplot-beta@lists.sourceforge.net>
Send bugs, suggestions and mods to
      <gnuplot-beta@lists.sourceforge.net>

Terminal type set to 'windows'
gnuplot>
```

Commands can be given to gnuplot at the “gnuplot>” prompt:

plot is the base command for the program. The command plots data from <data file>, plot data in the file is given as a tuple of coordinates (X Y) on each row. The coordinates must be separated with a "space". The name of the data file must be enclosed by quotation marks. Curly braces, {...}, are used to specify optional parts of the command.

plot {*ranges*} <*data file*> **with lines** {**,** <*data file*> **with lines**...}

ranges ::= [*xmin***:***xmax*] [*ymin***:***ymax*]
    | [*xmin***:***xmax*]
    | [] [*ymin***:***ymax*]
    | [*xmin***:***xmax*]

Ex.

```
plot ”file.dat” with lines
plot [0:35040] ”file1.dat” with lines, ”file2.dat” with lines
```

The command **replot** without any parameters replots the most recent **plot** command. This is useful when you want to plot something to a file that you just plotted on the screen. If parameters are passed to **replot** they will be put at the end of the most recent **plot** command before it is replotted.

E.g.

```
plot [0: 35040] ”file1.dat” with lines
replot ”file2.dat” with lines
```

the same as

```
plot [0: 35040] ”file1.dat” with lines, ”file2.dat” with lines
```

The command **set** controls a lot of settings. With the command **show** one can find out the current value of a setting. The command **show all** shows all settings.

set grid - Plot with grid.

set xtics - Number categories between tick-marks (e.g. years: set xtics 8760)

set nogrid - Plot without grid (default).

set output - The plot is sent to standard output (default).

set output <*plot file*> - The plot is sent to the file <*plot file*>. The name of the file must be surrounded by quotation marks (").

set terminal x11 - The plot is generated for a x11 terminal (default).

set terminal postscript - The plot is generated in postscript. Used when printing to laser printers.

set xrange [*xmin***:***xmax***]** **-** The interval at the x-axis that is plotted (e.g. 0-4 years: set xrange [0 : 35040])

set yrange [*ymin***:***ymax***]** **-** The interval at the y-axis that is plotted.

The command **quit** exits gnuplot.

Use the following commands to send the plot on the screen to a postscript file.

```
set output "file1.ps"
set terminal postscript
replot
```

The postscript file can be printed using the **gsview** program.