Testing CAN-based Safety-Critical Systems using Fault Injection

Dr. Marcus Rimén, Dr. Jörgen Christmansson

Today, many Distributed Control Systems (DCS) communicate via a Controller Area Network (CAN). A failure in a DCS controlling a safety-critical application, *e.g.*, an automotive brake system, can lead to significant economic losses or even the loss of human lives. Therefore, such DCS are usually designed to avoid failures in the service they provide to the user by including capabilities to handle the faults that could cause a failure. However, the testing of a safety-critical DCS' capability to handle faults is a complicated task. A common test method is called fault injection. Using fault injection, realistic artificially generated faults are inserted into the system under test while the performance of the system's fault-handling capabilities is measured. This paper presents some conceptually simple fault injection techniques that makes it possible to assess the handling of realistic node-level faults in a CAN-based distributed control system.

1 Introduction

Today, many Distributed Control Systems (DCS) communicate via a Controller Area Network (CAN). Many of these DCS are being built such that manual intervention is no longer a feasible backup measure in the event of a system failure. The control of vehicle dynamics in future automobiles, for instance, will be totally dependent on computers, as these systems will be built without any mechanical backup devices. A failure in a computer system that controls such an application can thus lead to significant economic losses or even the loss of human lives. Fault tolerance techniques are therefore deployed to obtain a dependable DCS.

Three fundamental terms in fault-tolerant design are fault, error and failure [1]. There is a cause-effect relationship between these terms; faults are the cause of errors, and errors are the cause of failures. These three terms can be described by the three universe model.

The first universe is the *Physical universe*. This is where faults occur. A fault is a defect that occurs within some hardware or software component.

The second universe is the *Informational universe*. This is where errors occur. An error is the manifestation of a fault in a unit of information, such as the contents of a memory location or a CPU register.

The third universe is the *External universe*. This is where failures occur. A failure is any deviation that occurs from the desired or expected behavior of a system.

One of the most crucial issues in the development of a fault-tolerant computer system is the verification and validation (V&V) of its fault-handling mechanisms [2][3]. These mechanisms include those parts of the system that perform such functions as error detection, error masking, recovery and system reconfiguration. Ineffective or unintended operation of these mechanisms can significantly impair the dependability of the system. Even a small reduction of, for instance, the error detection coverage, *i.e.*, the probability that an error is detected given that one has occurred, may significantly degrade system dependability.

Analytical dependability modeling and formal design verification are important tools for the verification and validation of a fault-tolerant computer system. However, when verifying and validating a fault-handling mechanism it is important to deal with both faults and the propagation of errors in the system. These aspects are very hard to model analytically and therefore an experimental technique to verifying and validating the system is also needed. The obvious way to experimentally verify and validate a fault-handling mechanism is to inject faults into the system and study the system's response during fault injection tests.

This paper is organized as follows. Section 2 presents the main goals of fault injection testing. A fault model suitable for CAN-based DCS is presented in section 3, and section 4 outlines how the faults are injected. Section 5 concludes the paper.

2 Goals of Fault Injection tests

The main goals of most fault injection tests are threefold:

- ***verification***
- ***calibration***, and
- ***validation***.

First of all, as fault-handling mechanisms are not exercised during normal operation under fault-free conditions, fault injection must be used to ***verify*** that they are implemented according to the design specification. This is usually the simplest task of the three.

Furthermore, many error detection mechanisms make use of threshold constants which must be carefully ***calibrated*** using fault injection so that the mechanisms always signal a detected error when a fault has occurred but never erroneously signal a detected error during fault-free operation. This task is more difficult than it might seem, as there usually is a trade-off to make between the conflicting interests of high availability and high safety in a system. High availability needs low-sensitive error detection mechanisms so that false detection never occurs, while high safety calls for high-sensitive mechanisms so that an error is never undetected. The problem is usually to distinguish ordinary noise on sensor signals from sensor faults.

Finally, it is important to ***validate*** the efficiency of the specified fault-handling mechanisms. The mechanisms should be validated after calibration by means of fault injection. By efficiency we can here mean any of the following non-exhaustive list of things:

- detection coverage,
- detection latency, *i.e.*, the length of time between the occurrence of a fault and its detection, and
- reconfiguration latency, *i.e.*, the length of time between the detection of an error and the resulting reconfiguration of the system.

The validation activity aims at validating that the system detects and handles faults (*e.g.,* by reconfiguration) with the coverage and latency specified in top-level requirements. In many cases, dependability requirements are specified instead of coverage and latency requirements. In that case, the validation activity investigates if the system meets the specified dependability requirements, *e.g.*, the appropriate level of availability or safety. This task is by far the most difficult of the three.

The test cases will differ depending on the goal of the test. A test case describes what fault to inject when to inject it, the use-case of the system at the time of injection, what to measure, etc.

When verification and calibration is the goal, each test case is focused on testing some well-defined part of the system that is specified in the design requirement specification, *e.g.*, a fault-handling mechanism. The fault to inject must in this case be a fault that the fault-handling mechanism is intended to detect. There is no point in injecting a fault that it cannot detect.

When validation is the goal, the test cases are focused on injecting all possible kinds of faults for all possible use-cases of the system. This is done to determine the probability that any fault is detected and handled [4]. Let us assume that a fault causes an error that alters the value of a variable in the system. In many cases the effect on the system will be different depending on when the variable was corrupted and also on the amount of corruption, e.g., did the variable increase by just 1 or by 1000? For such fault types, exhaustive testing is intractable. Therefore, statistical inference is often employed to extract meaningful information from the system, such as point estimates of the detection probability. To get a high statistical confidence in the point estimate, vast numbers of faults might have to be injected. This fact contributes to the difficulty in doing validation by testing.

Independent of the goal of the fault injection test activity, a fault model is always needed to limit and unambiguously define the faults to inject. The next section proposes a fault model that is suitable for CAN-based DCS.

3 Fault model

A general problem when injecting faults is that most real faults are hard or even impossible to inject artificially. Instead, a set of 'injectable' faults must be selected which are believed to represent real faults in the sense that they will cause errors that are similar to those caused by real faults. The fault selection procedure is, as previously discussed, dependent on the goals of the fault injection testing. When, for instance, validation is the goal, the selection procedure aims at a large variation in the error patterns.

In practice, different faults are used depending on the goal of the testing (i.e., verification, calibration or validation) and the 'level' of the error detection capabilities of the system. For example, when evaluating a DCS that contains basic components such as nodes and communication links, node and communication link faults are relevant. There are often tools available that test the communication links for electrical-level faults. Our concern is therefore how to test for node-level faults. To do this, we must have a realistic node-level fault model.

To understand the node-level fault model, we must go inside the node and view it as a separate system that may fail due to errors caused by faults. The failure modes of the node will become our node-level faults when viewing a complete DCS.

An erroneous state in the CPU of a node of a DCS can be caused by a hardware fault or by a software fault. Hardware faults are caused by, for instance, cosmic particle intervention, fluctuations in the power supply, electromagnetic pulses and manufacturing flaws. Examples of software faults are missing initialization, omitted logic, incorrect timing and wrong algorithm. The effects of these faults on a CPU can be emulated by means of a technique called SoftWare Implemented Fault Injection (SWIFI). SWIFI mimics the effect of a fault via the modification of the information content of memory locations or CPU registers [5]. That is, SWIFI is actually a technique for injecting errors into the node.

SWIFI can be used to verify and calibrate fault-handling mechanisms in a node. Furthermore, SWIFI can be used to validate the dependability of the node. Consequently, SWIFI is a powerful node-testing tool. However, the deployment of SWIFI requires a detailed knowledge of the node and the insertion of extra software in the node. More specifically, the modification of memory locations and CPU-registers are carried out by special purpose software that must be downloaded to the node. This software must be activated in a controlled way. The activation is often done via external interrupts, and hence impose that extra wiring must be added to the node. A complementary, less intrusive, technique would thus be handy.

An erroneous state might propagate from a component in the node to the border of the node, e.g., to the CAN bus. The node will in this case deliver a service that deviates from the expected service and such a deviation is labeled node failure. The failure occurred because the node was erroneous. The adjudged or hypothesized cause is a fault [6]. A failure of a node in a distributed system can be classified as one of the following failure categories [7]:

- *Crash failure*. A crash failure occurs when a node loses its internal state or halts for more than one cycle. The bus-level effect is that the node stops sending messages.

- *Omission failure*. An omission failure occurs when a node omits to send a message.

- *Timing failure*. A timing failure occurs when a node's message is functionally correct but untimely - the message is sent outside the specified real-time interval.

- *Incorrect computation failure*. An incorrect computation failure occurs when a node sends an incorrect message.

- *Byzantine failure*. A Byzantine failure occurs when the node neighbors receive different messages during a broadcast.

All of these failure categories can be emulated via the modification of the messages that are sent between the nodes in a DCS. Such a modification can be done without the connection of additional wires to the nodes and without the insertion of extra software in the node.

By replacing the word ‘failure’ by ‘fault’ for each failure category above, we have arrived at the node-level fault model we propose to use for a DCS. The next section describes how the fault model can be implemented in a CAN-based DCS.

4 CAN Fault Injection

Fault injection, which affect a CAN frame, can be used to implement the fault model proposed in the previous section. We have developed a CAN fault injection (CANFI) tool with the product name VeriCan-Interactive. Basically, the tool has two CAN ports and it acts as a repeater for CAN messages between the ports. One has to partition the system under test's CAN bus into two buses and insert the tool between the buses (see Figure 1). This result in a delay of the amount of time it takes to receive one message.

This delay is typically small. Assuming that the message is not repeated until it has been completely received, the delay depends on the message bit-length and the communication bit rate. For example, the delay is approximately 0.5 ms when the bit rate is 250kbit/s. For most systems the message delay has no serious impact on the system’s function and performance.

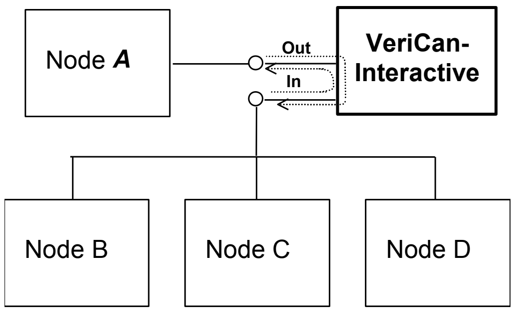

Figure 1. Partitioning of the system under test.

VeriCan-Interactive implements the fault model by means of CANFI. More specifically:

- A node-level *crash fault* in node ***A*** is simulated by simply not repeating frames sent from node ***A***.
- An *omission fault* is mimicked by not repeating a specified frame sent by node ***A***.
- A *timing fault* is simulated by delaying the repetition of a specified frame sent by node ***A***.
- An *incorrect computation fault* is emulated via modification of the content of a specified frame sent by node ***A***. Note that the modified frame is always sent with the correct low-level checksum specified by the CAN protocol.
- A *Byzantine fault* is mimicked by modifying the content of a specified frame meant for node ***A***. Note that the node will receive a frame with correct checksum.

A controlled fault injection test requires that a particular fault can be injected when a specific event occurs. Examples of events that can trigger a fault injection are:

- Arrival of a CAN frame with a specific identity.
- A timer elapses.
- The value of a particular signal goes above a threshold.

5 Summary

To summarize, we have in this paper pointed out the need for using fault injection techniques for verifying, calibrating and validating safety-critical distributed control systems.

All producers of safety-critical systems should be potential users of fault injection techniques. Examples of such producers are found in the automotive, medical equipment and machine control industry.

Finally, we presented a fault model containing a simple set of five practical faults that can be used when performing fault injection testing of a CAN-based distributed control system.

6 References

[1] D. K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice Hall PTR, 1996.

[2] J. Karlsson *et al.*, “Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms”, *IEEE Micro*, pp. 8-23, February 1994.

[3] R. Chillarege *et al.*, “Understanding Large System Failures - A Fault Injection Experiment”, in *Proc. FTCS-19*, Chicago, MI, pp. 356-363, June 1989.

[4] J. Arlat *et al.*, "Fault Injection for Dependability Validation: A Methodology and Some Applications", *IEEE Transactions on Software Engineering*, Vol. 16, No. 2, pp. 166-182, February 1990.

[5] J. Christmansson *et al.*, “An Experimental Comparison of Fault and Error Injection”. in *Proc. Ninth IEEE Int. Symp. On Software Reliability Engineering (ISSRE´98)*, pp. 369-378, Paderborn, Germany, November 1998.

[6] J.C.Laprie (ed.), *Dependability: Basic Concepts and Terminology*, Dependable Computing and Fault-Tolerant Systems series, Vol.5, Spring-Verlag, 1992.

[7] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Communications of ACM*, Vol.34, No.2, pp. 56-78, 1991.

Carlstedt Research & Technology AB
Stora Badhusgatan 18-20
S-411 21 Göteborg
Sweden

Phone +46 31 701 42 00
Fax +46 31 10 19 87

E-mail [rimen/joc]@crt.se
Web http://www.crt.se