

Designing *sfiCAN*: a star-based physical fault injector for CAN

David Gessner, Manuel Barranco, Alberto Ballesteros, Julián Proenza
Dpt. Matemàtiques i Informàtica, Universitat de les Illes Balears, Spain
{david.gessner, manuel.barranco, julian.proenza}@uib.es

Abstract

This paper presents the design and a preliminary implementation of *sfiCAN*: a physical fault injector for the CAN field-bus that allows the creation of a great variety of complex fault scenarios. The fault injector replaces the CAN bus topology with a star topology, whose central element is a hub with fault injection mechanisms. The fault injector is easily configured, with great flexibility, from a PC connected to a dedicated port of the hub. For this it uses a fault-injection specification, which is translated to a configuration protocol on top of CAN. This protocol is only used in-between fault injection tests and therefore does not interfere with the execution of any test. The purpose of the fault injector is to test the behavior of the nodes of a CAN network in the presence of channel errors, in particular, of the nodes' CAN controllers and the software executing on them, for which the star topology is transparent.

1. Introduction

Controller Area Network (CAN) [5] is a widely used field bus often deployed for distributed embedded systems (DES) in harsh environments. Although CAN includes several mechanisms to make it robust to such environments, faults in the network still occur. Thus, in critical applications, a precise study of the system response to such faults is needed. For this, fault injection, which consists in generating artificial faults, is particularly useful.

The artificial faults may be generated in a simulated model of the system or in a physical prototype. The former has the advantage that it can be tested before a prototype is available, but it is less realistic and accurate. Thus, for critical systems, it is highly recommended to evaluate the system with physical fault injection once a prototype is available. This evaluation should particularly include the nodes of the DES, which should be as similar as possible to the production ones. Specifically, the DES should be evaluated with the production software deployed to the nodes. Moreover, the behavior of the system should be evaluated under a variety of faults, which may include complex and/or unlikely fault scenarios, such as inconsistent faults, i.e., faults that affect some nodes but not others. As far as we know, at the time of this writing, no fault injector for CAN exists that satisfies these requirements. This paper presents the design and a preliminary implementation of a physical fault injector for CAN to satisfy them.

The Star-based physical Fault Injector for CAN, *sfiCAN*, is based on a star topology for CAN, which is transparent from the nodes' point of view. Figure 1 shows the high-

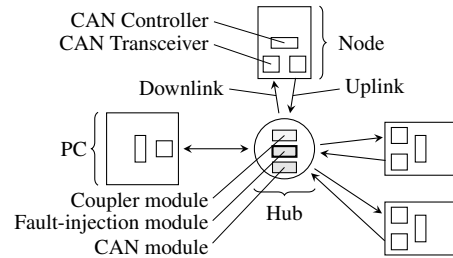


Figure 1. Fault-injection architecture.

level architecture of the fault-injection infrastructure. Each node is connected to a hub, where a fault-injection module resides, by means of a dedicated link comprised of an uplink and a downlink. A node's CAN controller is connected to the node's link by two CAN transceivers using a configuration identical to the one in CANcentrate [2]. Inside the hub a coupler module implements CAN's wired-AND function by performing the logical AND of the uplink signals and then broadcasting the result through all downlinks. Logically this implements a CAN bus, but with the advantage that local faults can be injected into the nodes with enough granularity to only affect an incoming or outgoing signal. The hub also contains a CAN Module. Its function is to observe the coupled signal, output by the coupler module, and to provide to the fault-injection module a set of signals indicating which bit within which bit field is currently broadcast. If other modules are added to the hub, the fault injector can also be used for more advanced CAN star topologies, e.g., if CANcentrate's fault-treatment module [2] is added, the fault injector can be used to inject faults into a CANcentrate network.

The hub has a dedicated port to which a personal computer (PC) is connected. The corresponding link is not separated into an up- and downlink since no faults are injected into the PC's link. Moreover, this allows connecting the PC to the hub using an off-the-shelf CAN controller board.

The purpose of connecting the PC to the hub is to easily configure the fault-injection module. This is possible since this module is an instance of a network configurable component (NCC), i.e., a component connected to the network that can receive its configuration from this network.

The remainder of the paper is organized as follows. Section 2 describes the operating modes of an NCC and Section 3 how the configuration to be applied to the fault-injection module is specified. Section 4 briefly overviews the implementation of this module. Section 5 describes an example fault injection and Section 6 related work. Finally, Section 7 concludes this paper and points to future work.

2. Operating modes of an NCC

A network configurable component can work in four modes: the *configuration mode*, the *idle mode*, the *wait-for-whistle mode*, and the *execution mode*.

An NCC enters the configuration mode when the PC broadcasts a *configuration-mode frame*. This frame has an identifier of the highest priority reserved to be transmitted exclusively by the PC.

During the configuration mode the PC typically transmits *configuration commands* encoded in CAN frames. Moreover, each NCC is identified uniquely by an *NCC identifier* (NCC ID). This ID is a regular CAN identifier, but during the configuration mode it is interpreted as an NCC ID. This allows configuration commands to be sent to a particular component without that command being interpreted by another one, i.e., unicast addressing. Moreover, the CAN IDs can be used by the nodes for the usual purposes during the normal operation of the network.

The only type of NCC currently implemented is the fault-injection module. It has its own set of configuration commands. If later on new types of NCCs are added, new sets of configuration commands need to be added for these.

The configuration of each NCC is terminated by a unicast *enter-idle-mode frame* or *wait-for-whistle frame*. The former instructs the destination NCC to enter the idle mode, the latter instructs it to enter the wait-for-whistle mode. A component in the idle mode ignores all frames except a configuration-mode frame; in the wait-for-whistle mode, it ignores all frames except a configuration-mode frame and a *starting-whistle frame*. A starting-whistle frame is broadcast by the PC to tell all NCCs in the wait-for-whistle mode to enter the execution mode. In the execution mode, a component starts to execute. For instance, a fault-injection module starts to inject faults according to its configuration.

Distinguishing between an idle and a wait-for-whistle mode allows some NCCs to be disabled for a given test.

3. Fault-injection specification

How the fault-injection module is configured during the configuration mode is specified in a text file: the *fault-injection specification*. Figure 1 shows the corresponding Backus-Naur Form (BNF) in ISO/IEC 14977 syntax [1]. The specification has a series of labeled *fault-injection configurations*, which are a set of key-value pairs, each of which we call a *configuration parameter*. Note that for brevity's sake we have omitted from the BNF the new line character that must follow each configuration parameter and the definition of a character *string*, a *natural* number, and a *boolean*, which have their usual definitions.

The configuration parameters indicate what, where, and when to inject. What to inject is given by a *fault-injection value*, e.g., a dominant bit (*stuckDominant*) or a given sequence of bits (*bitFlip*, with the sequence given by *value_bfvalue*). Where to inject is designated by a *target link*, e.g., the downlink of port 1 (*target_link* = *port1dw*). The specification of when to inject is more com-

```

specification = { '[' , string , ']' , fi_config }
fi_config = value , target_link , mode , aim , fire ,
           cease , [withdraw] ;
value = 'value_type' , '=' , value_type_value ,
       [ 'value_bfvalue' , '=' , { '0'|'1' } ] ;
target_link = 'target_link' , '=' , link - 'coupled' ;
aim = 'aim_count' , '=' , natural ,
     'aim_filter' , '=' , filter_value ,
     'aim_field' , '=' , field_value ,
     'aim_link' , '=' , link ,
     [ 'aim_role' , '=' , role_value ] ;
fire = 'fire_field' , '=' , field_value ,
     'fire_bit' , '=' , natural ,
     'fire_offset' , '=' , natural ;
cease = 'cease_bc' , '=' , natural
      | 'cease_field' , '=' , field_value ,
      'cease_bit' , '=' , natural ;
withdraw = 'withdraw_count' , '=' , natural ,
          'withdraw_filter' , '=' , filter_value ,
          'withdraw_field' , '=' , field_value ,
          'withdraw_link' , '=' , link ,
          [ 'withdraw_role' , '=' , role_value ] ;
mode = 'continuous' | 'iterative' | 'selective' ;
value_type_value = 'stuckDominant' | 'stuckRecessive'
                | 'bitFlip' | 'inverse' ;
filter_value = ( '0'|'1'|'x' ) , { '0'|'1'|'x' } ;
link = 'port0up' | 'port0dw' | 'port1up'
      | 'port1dw' | 'port2up' | 'port2dw'
      | 'port3up' | 'port3dw' | 'coupled' ;
field_value = 'idle' | 'id' | 'rtr' | 'res'
            | 'dlc' | 'data' | 'crc' | 'credelim'
            | 'ack' | 'ackdelim' | 'eof'
            | 'interfield' | 'errflag' | 'errdelim' ;
role_value = 'dont_care' | 'tr' | 're' ;

```

Listing 1. Fault-injection specification BNF.

plex. First, an *aiming condition* (*aim*) indicates a condition that must be satisfied before a fault is injected. For instance, the aiming condition may be the third reception of a CRC (*aim_field* = *crc*, *aim_count* = 3) that begins with the prefix '0101' or '0111' (*aim_filter* = 01x1) and is detected on the downlink of port 2 (*aim_link* = *port2dw*) when the node connected to the target link is the transmitter (*aim_role* = *tr*). The frame that satisfies the aiming condition is called the *start frame*. Next, the precise bits into which to inject are specified. For this two sets of key-value pairs, designated by *fire* and *cease* in the BNF, are used. The former indicates the *fault-injection-firing condition*, i.e., the first bit into which to inject a fault. The latter indicates the *cease-fault-injection condition*, i.e., how long to inject, either in terms of a bit count or in terms of a bit within a bit field up to which to inject. If it is specified as a bit plus a bit field, a *withdraw condition* can be specified. A withdraw condition is a condition that must be satisfied to stop injecting a fault-injection value. It is defined in the same way as an aiming condition, except for the fact that an undefined withdraw condition indicates an unlimited fault-injection, i.e., a permanent fault. The frame that satisfies the withdraw condition is called the *end frame*. Between the start and end frames several frames may have been transmitted. The *fault-injection mode* defines how the fault-injection value is injected in these frames. In the *continuous* mode, it is injected in all bits that range from the fire condition in the start frame up to the cease condition in the end frame. In the *iterative* mode, the fire and cease conditions indicate a range within each frame between the start and the stop frame where the fault-injection value is to be injected. The *selective* mode is similar to the iterative

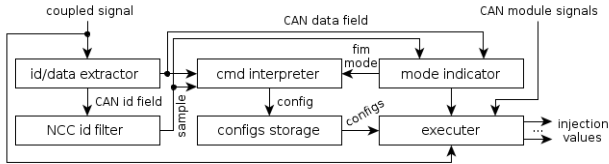


Figure 2. Fault-injection module diagram.

mode, but the fault-injection value is only injected in those frames that also satisfy the aiming trigger.

Not all syntactically correct specifications are valid semantically. For instance, in the arbitration (ID field) a node's role is invalid as the role is undetermined until the arbitration ends. Because of lack of space, we do not make these semantic restrictions explicit, but most can be determined by taking into account how the CAN protocol works.

Each configuration parameter is encoded by the PC as a configuration command into one or more CAN frames that have the NCC ID of the destination fault-injection module. To configure the fault-injection module the PC first broadcasts a configuration-mode frame and then the configuration commands. The fault-injection module decodes these commands and configures itself accordingly.

4. Implementation aspects

We implemented the hub's coupler, CAN, and fault-injection module in VHDL and synthesized them in a Xilinx Spartan-3 XC3S1000 FPGA. For the coupler and CAN module we used components from CANcentrate[2]; whereas the fault-injection module is new.

Figure 3 shows the fault-injection module internals. It contains the following submodules. (i) The *ID/data extractor* extracts the ID and data field from the currently transmitted frame. (ii) The *NCC ID filter* checks if the ID previously extracted matches the fault-injection module's NCC ID. If so, it generates a signal that causes the mode indicator and the command interpreter to read the data field from the ID/data extractor. (iii) The *mode indicator* checks if the data field indicates a mode switch (e.g. from configuration to idle mode) and, if so, signals the new mode to the command interpreter and the executer. (iv) The *command interpreter*, during the configuration mode, checks if each data field received contains a fault-injection configuration parameter, instead of a mode change. From all the received configuration parameters it builds a physical fault-injection configuration, which is passed to the configurations storage when an appropriate configuration command is received. (v) The *configurations storage* stores all the fault-injection configurations and makes them available to the executer. (vi) The *executer* contains a set of *programmable configuration executers*, which are modules that, during the configuration mode, are each programmed according to one of the stored configurations. During the execution mode these modules perform the fault injection according to how they have been programmed.

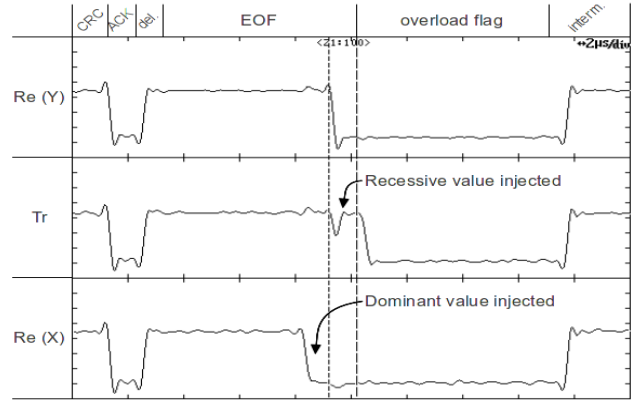


Figure 3. IMO scenario.

5. Example: inconsistent message omission

Figure 4 shows a screenshot taken with a Yokogawa DL7440 digital oscilloscope. The image was taken while the fault injector injected the faults that lead to the inconsistent message omission (IMO) scenario first described in [3]. We annotated the screenshot at its left with the role that the node corresponding to each of the shown signals played. $Re(Y)$ and $Re(X)$ are the signals on the downlink to a receiving node Y and X, respectively; Tr is the signal on the downlink to the transmitting node. Moreover, at the top we annotated the bit fields of the transmitted CAN frame as seen by the node labeled Y. Note that the tick marks shown on the X axes do not indicate bits, but $2 \mu s$ intervals. The two vertical dashed lines at the center of the image mark the limits of the last bit of the end of frame (EOF) field.

As shown in the screenshot, a dominant bit is injected in the last-but-one bit of the downlink of the node labeled X, which subsequently generates an error flag. The first bit of this error flag, however, is not detected by the transmitter, labeled Tr , because of a recessive bit injected into its downlink. No error is injected into the link to the node labeled Y, which, thus, detects the first bit of the error flag. The subsequent response of the network is the one described in [3]: node X rejects the frame, node Y accepts it, and node Tr does not retransmit it.

Listing ?? shows the fault-injection specification used to generate the IMO scenario. Because of space constraints, however, we obviate its description.

6. Related work

In 2003, a physical fault injector for CAN was proposed [6], claimed to be the only one able to generate complex fault scenarios. Thus, here we only consider said injector and the only other physical fault injector for CAN that has been published between 2003 and the date of this writing.

The fault injector presented in [6] uses a software tool, called CANfidant, and several *individual fault injectors* (IFI), which are hardware circuits inserted between the CAN transceiver and CAN controller of each node. This fault injector requires a complete a priori knowledge of the specific frame transmission order on the bus. Assuming this order, CANfidant assists the engineer in the design of

the error scenario by simulating the bit by bit behaviour and generating the instructions for the IFIs. The IFIs then inject each fault by counting the frames from the start of a test and the bits within the target frame. Thus, if the frames are not transmitted in the expected order, which could easily happen in the event-triggered CAN, the injection will be performed incorrectly. In contrast, *sfCAN* is also able to inject complex fault scenarios, but does not require a priori knowledge of the frame order. Nevertheless, it can also be used together with simulation. For instance, specific scenarios can be simulated with *CANfidant* and then an appropriate fault-injection specification for *sfCAN* can be created.

In [4] a fault injector is presented that can be attached to an existing CAN network as a node. Although this has the advantage that it does not require the modification of the network or the nodes' transceivers, it provides only low spatial resolution for the fault injection since all faults always affect the whole network. Thus, with such an approach inconsistency scenarios, such as those causing IMOs, cannot be injected. Moreover, that fault injector is far less configurable than *sfCAN*.

7. Conclusions and future work

In this paper we presented *sfCAN*: a network configurable fault injector based on a star topology that allows the creation of complex fault scenarios by injecting physical faults into a CAN network. The use of a star topology with a central hub, whose links are separated into an uplink and downlink, allows the faults to be injected with high spatial resolution; whereas the CAN module residing inside the hub, which keeps track of what bit within which bit field is currently being broadcast, allows fault injection with high temporal resolution. Moreover, the fact that several different fault-injection configurations can be stored simultaneously in the fault-injection module, and that these can be specified using many different configuration parameters (as illustrated by the BNF shown in Listing 1), makes *sfCAN* very flexible and potent. Also, it has a low impact on the system under test: it only introduces a short delay, which can be accounted for as an additional transmission delay, and only requires a single CAN ID to be reserved during the normal functioning of the application.

We see two main uses for the fault injector. First, to test how the CAN controllers and software of a CAN network respond to channel faults, in which case modifying the topology to be a star is not relevant since it is not the channel which is tested. Second, using it to test CAN star topologies—in particular the (Re)CANcentrate systems developed under the CANbids¹ project.

The paper also presented the approach of using NCCs for CAN. In general, this approach is very flexible. It allows such components to be added inside the nodes or other devices connected to the CAN network. Using several different types of NCCs a fully automated testing infrastruc-

```

1  [fault injection 1]
2  value_type = stuckDominant
3  target_link = port1dw
4  aim_count = 1
5  aim_filter = 10110101010
6  aim_field = id
7  aim_link = coupled
8  aim_role = re
9  fire_field = eof
10 fire_bit = 5
11 fire_offset = 0
12 cease_bc = 1
13 mode = continuous
14 [fault injection 2]
15 value_type = stuckRecessive
16 target_link = port0dw
17 aim_count = 1
18 aim_filter = 10110101010
19 aim_field = id
20 aim_link = coupled
21 aim_role = tr
22 fire_field = eof
23 fire_bit = 6
24 fire_offset = 0
25 cease_bc = 1
26 mode = continuous

```

Listing 2. Example fault-injection spec.

ture for CAN can be built.

A particular NCC we plan to add to the hub is a *logging module*. Its purpose is to capture relevant data during the execution mode of a test and to broadcast this data when polled for it during the configuration mode following the test. Since the logging module would reside inside the hub, it would have a privileged view of the system, and would be able to determine what has been transmitted to and from each node, as well as the internal status information from other modules residing inside the hub. Moreover, after a test, the transmission of these data to the PC would allow us to conveniently analyze them there.

Acknowledgement

This work was supported by the Spanish Science and Innovation Ministry with grant DPI2008-02195, FEDER funding, and the portuguese Fundação para Ciência e a Tecnologia with grant SFRH/BPD/70317/2010.

References

- [1] International Standard ISO/IEC 14977 - Information technology - Syntactic metalanguage - Extended BNF, 1996.
- [2] M. Barranco, J. Proenza, G. Rodríguez-Navas, and L. Almeida. An Active Star Topology for Improving Fault Confinement in CAN Networks. *IEEE Transactions on Industrial Informatics*, 2(2):78–85, May 2006.
- [3] J. Proenza and J. Miro-Julia. MajorCAN: A Modification to the Controller Area Network Protocol to Achieve Atomic Broadcast. *IEEE International Workshop on Group Communication and Computations, Taipei, Taiwan*, 2000.
- [4] M. S. Reorda and M. Violante. On-line analysis and perturbation of CAN networks. In *Proc. 19th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, 2004.
- [5] Robert Bosch GmbH. CAN Specification Version 2.0, 1991.
- [6] G. Rodríguez-Navas, J. Jiménez, and J. Proenza. An architecture for physical injection of complex fault scenarios in CAN networks. In *Proc. 9th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, volume 2, 2003.

¹<http://srv.uib.es/project/12>