EURASIP Journal on Embedded Systems
a SpringerOpen Journal

RESEARCH Open Access

A router for the containment of timing and value failures in CAN

Roland Kammerer[1*], Roman Obermaisser[2] and Bernhard Frömel[1]

Abstract

The dependability deficiencies and bandwidth constraints of the controller area network (CAN) can prevent its use in safety-relevant and performance-demanding applications. This paper introduces mechanisms for fault detection and fault isolation based on an intelligent CAN router, which exploits a priori knowledge about the permitted behavior of attached electronic control units (ECUs) in order to detect and contain failures. Experiments using an FPGA-based implementation of the CAN router evaluate these mechanisms under different failure modes (e.g., timing failures, masquerading failures). Due to its compatibility to the CAN standard, the router can improve the dependability and performance of systems with existing ECUs. In addition, we extend the application areas of CAN to systems with higher performance and dependability requirements than can be supported with a conventional bus-based network.

Keywords: CAN, CAN router, Fault detection, Fault location, Fault isolation, MPSoC design, ACROSS MPSoC

Introduction

The communication protocol controller area network (CAN) is used for asynchronous fieldbus networks in many application domains including the automotive industry, the avionic industry and factory automation. For example, cars typically contain several CAN buses for powertrain, infotainment and comfort functions.

The benefits of CAN include its simplicity, the decentral structure and the low cost for CAN controllers and wiring. However, severe limitations concerning reliability have been identified in literature such as the ability of a single faulty node to cause a global communication failure by monopolizing the bus [1], the susceptibility to bus short-circuits [2] or the absence of an atomic broadcast in case of asymmetric bit flips [3,4]. In addition, CAN exhibits diagnostic deficiencies such as the inability to trace back faulty message identifications to the sender nodes [5].

Therefore, a single CAN bus does not support the construction of embedded systems where the correct operation of the communication system is required to ensure safety. As a result, new communication protocols are introduced in different application domains to address the reliability issues such as FlexRay [6] in the automotive area.

This paper provides improvements of CAN w.r.t. fault-tolerance, which can provide an alternative to the replacement of CAN in many applications. Thereby, system developers can benefit from the low cost, the high numbers of existing CAN-based applications and the widespread expertise in CAN hardware and software.

We replace the CAN bus with a star topology based on an intelligent CAN router. *Fault isolation* is one of the primary objectives of the router and our main focus of this paper. The CAN router eliminates the hazard of medium failures of an individual CAN bus leading to a global communication failure. Furthermore, the router exploits a priori knowledge about the permitted behavior of CAN nodes in the time and value domains for the containment of node failures. In the time domain interarrival times are monitored and enforced. In the value domain, permitted message identifiers and constraints on the application data within a message are enforced.

As CAN is widely used in different domains, *legacy CAN-interface support* is of utmost importance. Modifying the legacy CAN-interface would result in tremendous cost for redeveloping existing systems. Therefore, the CAN router provides interfaces that are compatible to standard CAN. This includes electrical compatibility and standard-conforming services of the data link layer (e.g., arbitration mechanism, message ordering) [7].

*Correspondence: kammerer@vmars.tuwien.ac.at
[1]Vienna University of Technology, Vienna, Austria
Full list of author information is available at the end of the article

Further, the router overcomes limitations of existing CAN networks concerning overall cable length and overall bandwidth (40 m at 1 Mbit/s) by its star topology. Naming incoherences are solved by a CAN identifier translation. As today's breakdown logs do not assist the technician in a proper way [8,9], the router also provides new diagnostic services for the detection of timing failures (e.g., crash failure of a node, babbling idiot) and value failures (e.g., invalid CAN identifiers, implausible message contents).

Major contributions of the paper are the introduction of a system model of a fault-tolerant CAN-based system using the CAN router, as well as the explanation of the basic services of the CAN router for fault isolation. Furthermore, we provide an experimental evaluation of the effectiveness of the fault isolation mechanisms.

The paper is organized as follows: Section Controller area network provides an overview about CAN and its limitations. In Section CAN router we define the system model of the CAN router, present failure modes of CAN, and state the basic services of the router. In Section Fault detection and isolation we describe how the router detects and isolates previous mentioned failure modes. In Section Implementation of fault detection and isolation we concentrate on the implementation of the router and describe how the means of fault detection and isolation are realized. Section Test framework gives an overview about the test framework we used for our evaluations of the CAN router, where Section Experiments presents the experiments we conducted and their results. Section Discussion discusses the gathered results, and finally Section Conclusion concludes the paper.

Controller area network

CAN belongs to the class of event-triggered communication protocols. It uses a broadcast bus with "carrier sense, multiple access with collision avoidance" (CSMA/CA) for medium access control [7]. The bit transmission takes two possible representations. The recessive state appears only on the bus when all nodes send recessive bits. The dominant state occurs, if at least one node sends a dominant bit. A given bit-stream is transmitted using the "Non-Return-to-Zero" (NRZ) code. Bit stuffing prevents that more than five consecutive bits of identical polarity are transmitted. A node delays its transmission if the bus-line is busy. If the bus is idle the node can start sending. Bus access conflicts are resolved by observing the message identifier bits on the bus-line. While transmitting a communication message identifier, each node monitors the serial bus-line. If the transmitted bit is recessive and a dominant bit is monitored, the node gives up transmitting and starts to receive incoming data. The node sending the object with the highest priority will succeed and acquire bus access. The information exchange occurs using four types of protocol data frames:

1. *Data frames* are used for the transmission of CAN message objects. A data frame contains a unique identifier, which identifies the message object and denotes the message priority.
2. By transmitting a *remote frame* the dissemination of a communication object is explicitly requested. For the same identifier, the data frame takes precedence over the remote transmission request.
3. An *error frame* is used for error signaling. It contains an error flag.
4. The *overload frame* serves the purpose of extending the interframe space to handle overload conditions.

After a loss in the arbitration process or the reception of an error frame, the sender automatically performs a retransmission of the corresponding communication object. The integrity of data and remote frames are checked by a 15-bit cyclic redundancy code (CRC).

CAN was originally developed for non safety-critical applications and exhibits the following limitations w.r.t. predictability, dependability and performance:

- A CAN communication system possesses a large variability in the transmission latency. A message's transmission latency depends on the network load. This latency jitter causes an error in the temporal domain and introduces an additional measurement error if there is no global notion of time.
- The arbitration logic of CAN limits the throughput, because the propagation delay of the channel must be smaller than the length of a bitcell. A CAN network of 40 m results in a maximum bandwidth of 1 Mbit/s.
- CAN does not prevent babbling idiot failures. A node can continuously send highest priority messages and thereby prevent communication of other nodes.
- The CAN protocol does not include a clock synchronization service. If a global notion of time is required, it must be implemented at the host level.
- Communication errors are handled with immediate message retries. Errors cause increased latencies.
- Handling of station failures is performed with error counters by recording the receive and transmit errors. A threshold is defined for entering the error passive mode. In this mode, a node must wait for a minimum idle time on the bus before starting a transmission. If bus contention is low, this results in interleaving of correct and invalid messages. Exceeding of another error counter threshold results in entering the bus-off state. Under the assumption that failed nodes reach the bus off state, the worst-case inaccessibility time at 1 Mbit/s is bounded by 2.5 ms [10-12].

- Since the temporal properties of a CAN system are changed during the integration of the system, CAN does not support temporal composability [13]. The transmission of a message is triggered explicitly by a transmission request from the host. The temporal coordination of the communication activities is a global issue and depends on the host software in all nodes.
- CAN error recovery mechanisms are unable to ensure a consistent state, if an error is detected in the last but one bit of a frame. Possible consequences are an inconsistent message duplication or an inconsistent message ordering. Establishing consistency requires modifications to the host software [4,14] or a dedicated hardware component [3].

CAN router

This section describes the system model and states the terminology that is used in the rest of the paper. Figure 1 depicts a CAN system using the CAN router.

CAN segments, consisting of a CAN bus with at least one node are connected to the router via a CAN port. The router is implemented as a multi-processor-system-on-a-chip (MPSoC), where every CAN port is served by its own CAN interface subsystem (CIS). Each CIS consists of a CPU, local memory, and a CAN controller. The CPU executes software used for message processing. Every CIS contains a routing configuration that allows the router to forward messages from a source CAN segment to one or more destination segment(s). We use a time-triggered network-on-chip (TTNoC) [15] for the message transport between CISes. All processing in the router is time-triggered. Additionally, the router possesses a management port, served by the the management unit (MU), which is used for diagnosis and configuration (e.g., update of the routing configuration).

Fault hypothesis

In this section we define the fault containment regions of the CAN router and describe the failure modes we expect to occur.

A *FCR* is a region of the system that operates correctly regardless of faults outside of this region [16]. In our case this includes arbitrary logical or electrical faults. We distinguish two types of fault containment regions: The CAN router itself and the individual CAN segments. We assume the router to be free of faults. In case compatibility to standard CAN is not a strict requirement, a setup with two redundant CAN routers can be used to tolerate a single failure in one FCR.

A definition of *failure modes* is fundamental for the design of the CAN router as well as for error handling and containment of these failures. The following failure modes, which are subcategories of the arbitrary failure mode, are assumed for the CAN segments:

Stuck at dominant/recessive failures

If a node or the bus is affected by a stuck at dominant failure, the state of the bus becomes dominant (e.g., a node constantly sends '0'). In the bus-based CAN segment this means that no further communication is possible. In case of a stuck at recessive failure we have to distinguish between the node and the bus. If a node is affected by this kind of failure this node is not able to participate in the communication on the bus. If the bus itself is affected by a stuck at recessive failure, no nodes can communicate.

Crash/omission failures

Crash/omission failures are one of the most frequent failure types in CAN. The CAN standard also defines mechanisms (e.g., error counters) for mapping different types of faults into crash/omission failures ([7], p. 42). FCRs affected by this kind of failure either provide the specified

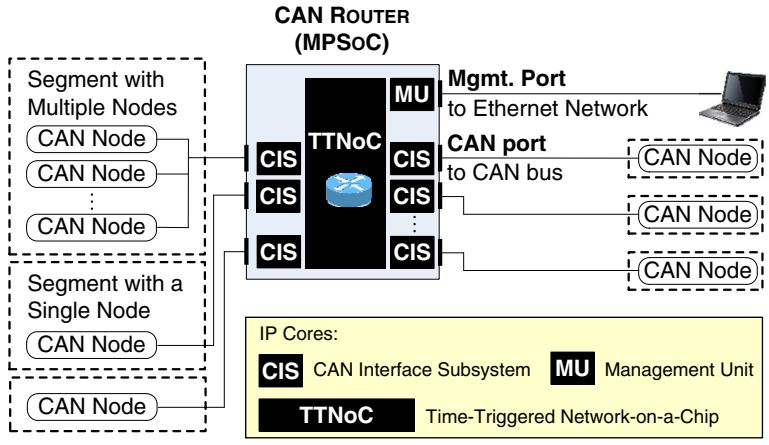

Figure 1 CAN router.

service (i.e., sending CAN messages), or they do not provide the service at all. Crash/omission failures are extreme scenarios of late message failures.

Asymmetric bit-flip failures

Even though in literature it is sometimes assumed that CAN provides an atomic broadcast, this is not the case. If the last but one bit of the end of frame (EOF) delimiter is affected by an asymmetric bit flip, the nodes on the bus can be split in two sets, one accepting the message, and one not. In this case the sender retransmits the message, which leads to a duplication of that message in the set of nodes that previously accepted the message [4,14]. A subsequent crash failure can lead to an inconsistent message omission.

Babbling idiot failures

As the former mentioned crash/omission failures, babbling idiot failures are in the category of message timing failures [17]. A babbling idiot failure is an extreme case of an early timing failure. Since conventional CAN does not provide effective mechanisms for handling babbling idiot failures [1], a node that sends high priority messages can disrupt the communication of all other nodes which share the same bus.

Masquerading failures

Masquerading failures, which are part of the value domain, occur, if one node impersonates the identity of another node. In case of CAN, one node could send its messages with the CAN identifier reserved for another node, which might lead to severe consequences. For example consider an environment where node A sends a temperature value, node B sends a velocity value, and node C opens a valve according to the temperature value from node A. If node B sends velocity values with the identifier of node A, node C misinterprets the velocity value as a temperature value. Masquerading failures in combination with diagnostic deficiencies of CAN are one of the reasons why today's automotive breakdown logs do not assist the technician adequately in the identification of faulty Electronic Control Units (ECUs) [9,18].

Basic services of the CAN router

The CAN router provides basic services which will be explained in the rest of this section.

Message rate control

In classic CAN a node k sends messages with a specific identifier i in the overall set of CAN identifiers $I \subset \{0, \ldots, 2^z{-}1\}$. A basic CAN identifier contains 11 bits (i.e., $z = 11$), whereas extended identifiers contain 29 bits (i.e., $z = 29$). As CAN is an event-triggered protocol, the interval between two messages with the same identifier is a stochastic variable. In order to provide fault isolation and enhanced diagnosis, we constrain the rate of messages. The behavior of a CAN node k is defined by the set M_k:

$$M_k = \{\langle i, d, e\rangle \mid \text{where } i \in I \text{ and } d, e \in \mathbb{Q}^+\}$$

where the positive rational numbers d and e are the *minimum* and *maximum interarrival times* associated with the identifier i. The router contains an entry that specifies these interarrival times for every CAN identifier that is valid on the corresponding CAN segment. In case of a violation, the message is discarded and the violation is reported to the MU. Blocking of untimely messages is one of the key aspects of fault isolation.

Properties for minimum and maximum interarrival times are often known and in the automotive domain they can be for example extracted from a fieldbus exchange format (FIBEX) [19] specification. Tools which generate a routing configuration from a FIBEX specification are described in [20].

Message multicasting

In conventional CAN a message that is successfully sent on the bus gets broadcasted to all other nodes. In order to overcome the limitations of CAN, the router supports selective multicasting, which uses the existing bandwidth more efficiently. The router contains knowledge about the destination CAN segment(s) of a message and forwards the message exactly to these segment(s). Broadcasting, which is a special case of multicasting, is supported by the router (i.e., the router forwards a message to all other CAN segments).

Message scheduling

On a CAN bus every transmitting node monitors the state of the bus. Whenever a node tries to send a recessive bit, and reads back a dominant bit, it backs off and retries to send the interrupted message at a later point in time. As high-priority message identifiers contain more leading dominant bits than low-priority identifiers, a message with a high priority is sent first (i.e., it wins the arbitration). In order to reproduce this behavior, the CAN router maintains a priority queue at every destination CIS and transmits the messages to the destination CAN bus according to their priority.

Identifier validation and translation

In a classic CAN setup a faulty node A could send messages with an identifier reserved for another node B. Therefore, the router contains knowledge about valid identifiers of a CAN segment in its routing configuration. If a node sends a message with an identifier not specified in the routing configuration, the router discards this message and sends a report to the MU. In addition to identifier validation, the router supports identifier translation. This is important in the case of legacy system integration,

where two legacy systems use the same CAN identifiers. It is possible to specify a translation ID for every CAN identifier used in the system.

Message checks

For every CAN identifier it is possible to specify a function that gets called with the content of the message. If, and only if, the message passes the check, the message gets forwarded to its destination. For example a message that contains an engine coolant temperature variable can be checked if the temperature is within a meaningful specified range.

Message content translation

As legacy system integration is one of the goals of our approach, the router supports the translation of the content of CAN messages. It is possible to specify a *translation function* per valid CAN ID. One practical example is the conversion between different measurement units (e.g., converting between degrees Celsius and Fahrenheit).

Diagnosis and management

The router contains a dedicated MU which is capable of collecting violations in the time and value domain. This includes violations of the minimum and maximum interarrival time, as well as invalid message identifiers. This information can then be used as input for further analysis. Additionally, the MU can be used to change the routing configuration at run-time. This includes the addition and removal of valid CAN identifiers, the modification of minimum and maximum interarrival times, as well as changing multicast patterns (i.e., the destination of CAN messages).

Fault detection and isolation

The purpose of this section is to describe how the failure modes mentioned in Section Fault hypothesis are detected and contained by the CAN router.

Asymmetric bit-flip failures

Based on the star topology, if the CAN router is correct, it is ensured that messages are consistently received by all correct CAN segments. If a node in a CAN segment transmits a message and the CIS does not accept this message, then the node has to retransmit the message. However, no destination CIS is influenced, therefore, the state is consistent over all CISes. If the source CIS accepts the message, it will forward the message to the destination CIS(es). The CAN router will subsequently send the messages on all correct destination CAN segments, thus ensuring a consistent overall state of all correct CISes. If a CAN segment exhibits a transient fault, the CAN controller at the destination segment will retransmit and try to eventually deliver the message.

Stuck at dominant/recessive failures

By the use of a star topology, stuck at dominant and stuck at recessive failures are contained at their corresponding source CAN bus. Therefore, a faulty node can only disrupt the communication of all the other nodes that share the same CAN segment, but it is not possible to influence the communication of other, separated CAN segments.

Crash/omission failures

As described in Section Basic services of the CAN router, the router contains knowledge about maximum interarrival times of messages for each valid CAN identifier. Further, the source CIS stores a timestamp of the last successful reception per CAN ID in its internal data structure. The router checks if the difference between the current time and the time of the last reception exceeds the maximum interarrival time. If this is the case, this temporal violation is reported to the MU. As the router cannot enforce messages from a potentially failed node, reporting the violation is the next best thing to do. On the MU this knowledge can be used to initiate a reconfiguration of the system. For example an α-count can be increased every time the maximum interarrival time is violated in order to discriminate between transient and permanent faults as discussed in [21]. In case of a permanent fault, the MU could initiate a reconfiguration that uses a spare node instead of the apparently failed one.

Babbling idiot failures

The router provides means for fault isolation when messages are sent too fast, e.g., in case of a babbling idiot failure. Whenever the difference between the current timestamp and the timestamp of the last reception is less than the specified minimum interarrival time, it is considered a temporal violation. In that case the router blocks this message and does not forward it to the destination CISes. Therefore, the fault is contained in the specific source CAN segment. In addition to that, the violation is reported to the MU, which then takes further actions (e.g., log the violation for later analysis conducted by a maintenance engineer).

Masquerading failures

The CAN router isolates masquerading failures and contains them in their source CAN segment. The routing configuration contains entries for every valid CAN identifier. Whenever a faulty node tries to send a CAN message with an identifier that is not specified in the routing configuration, the source CIS blocks this message and sends a violation report to the MU. This prevents the propagation of messages with an identifier not specified for the given CAN segment. Nodes sharing the same CAN segment can sill masquerade IDs of nodes on the

same segment. The router provides the best fault detection and isolation capability if CISes consist of a single CAN node.

Implementation of fault detection and isolation

Figure 2 shows the internal structure of the CAN router. The router itself is realized as a MPSoC, where every CAN port is served by its own CIS based on a Nios II softcore CPU. We use the ACROSS MPSoC [22] as our underlying platform. Our design allows us to cleanly decouple the temporal behavior of each CIS. Additionally, higher scalability and fault isolation is achieved. From a scalability point of view it is much easier to add additional CISes, compared to software or single-core designs where the addition of every new message requires extra processing power at one processing core that is shared among all CAN segments. The presented MPSoC design also provides better fault isolation compared to single core solutions because the TTNoC [15] ensures that a transient or permanent fault of a core does not affect the operation of other cores.

The operation of the router is strictly time-triggered and divided into *rounds of activity*. The underlaying ACROSS MPSoC [22] provides a generic timer service which is used to synchronously trigger activity rounds in all CISes. The trigger-period (2^{-15} s $\equiv$ 30.52 μs) is faster than the minimum interarrival time of CAN messages at 1 Mbps. Within one activity cycle each CIS checks if a newly arrived CAN message at the CAN port has to be processed, then it processes this message according to the associated routing configuration, and forwards the message to the TTNoC. Additionally, the CIS checks if it received a message from the TTNoC (i.e., this message was originally sent in the last activity cycle), and if this is the case, it processes the message and finally sends the message with the highest priority to the destination CAN bus.

The router introduces a delay of one CAN message due to its store and forward behavior and a maximum of three activity cycles from the instant a message is successfully sent on the source CAN bus, until it is stored in the priority queue at the destination CIS. This includes finishing the current activity cycle (i.e., which started before the new message arrived), and one additional activity cycle for processing the message and forwarding it to the TTNoC. As the message transport of the TTNoC is triggered by the system frequency, it is guaranteed that the message is available at the destination CIS until the next activity cycle starts. The third and last activity cycle is consumed at the destination CIS for processing and storing the message to the priority queue. If the queue is empty or there are no other higher priority messages, the newly arrived message is sent to the CAN bus attached to the destination CIS in the same activity cycle.

As shown in Figure 2, every CIS contains a local *routing configuration*. This configuration contains an entry for every CAN identifier that is valid on the given CIS. We use the configuration to store properties important for routing messages to their destination as well as to specify temporal properties important for fault detection. A sample entry is shown in Listing 1.

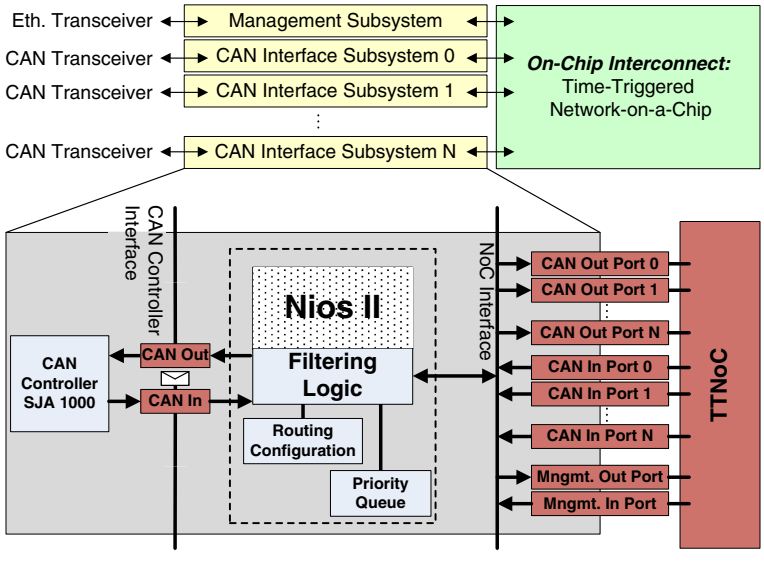

Figure 2 Overview of the CAN router implementation.

Listing 1 An example routing entry

```
typedef struct {
    time64 min;
    time64 max;
    time64 last_arrived_min;
    time64 last_arrived_max;
    uint16_t message_check;
    uint16_t message_translation;
    uint16_t forward_to;
} routing_entry;
```

The variables `min` and `max` contain the specified minimum and maximum interarrival time, `forward_to` is a bit array that specifies the destination CIS(es). Variables starting with `last_arrived` are used to store timestamps of the last successful message reception. The variables `message_check` and `message_translation` contain a position in an array of function pointers that can be used to validate the content of a message, respectively translate the content.

As a direct mapping from CAN ID to a position in the routing configuration is not feasible (i.e., this would require 2^{29} entries in the routing configuration), we look up the configuration with the help of a binary search. This has the advantage that the memory footprint can be kept small and that the lookup is bounded by $O(log(n))$. For the lookup we use a sorted array of CAN identifies, which is generated off-line by configuration tools and checked during start-up by the MU. The structure of a search entry is shown in Listing 2.

Listing 2 Structure of an entry used for configuration lookup

```
typedef struct {
    uint32_t can_id;
    uint32_t pos; /* in the routing_config */
} searchstruct;
```

Temporal domain

A CIS checks for violations of the minimum interarrival time whenever it receives a message from its CAN controller and has to forward the message to one or more destination CAN segments. In Figure 2, this is represented by a message flow from the left to the right. In order to get precise timestamps of message receptions, we store the timestamp as soon as the CAN message arrives (i.e., even between two activity cycles). As the activity cycles are shorter than the minimum interarrival times of CAN messages, we have to store at most one timestamp. In every activity cycle the CPU checks if a new message has to be processed. If this is the case, the CPU transfers the message over the CAN Controller Interface to its own memory and does further processing. In the first step the router has to look up the configuration that is associated with the CAN identifier of the newly arrived message. If the ID is found, the routing configuration associated with the CAN ID gets evaluated.

The router checks the properties that are important for fault isolation. For minimum interarrival time validation the router builds the difference between the timestamp of the current reception and the timestamp stored for the last successful reception (`last_arrived_min`). If the specified minimum is violated, the router sends a message to the MU. In case of a violation of the minimum interarrival time, the message gets blocked. If there was no violation we update `last_arrived_min` and send the message to the destination CIS(es). The timestamp for `last_arrived_max` is updated on every reception, whether it was a minimum interarrival time violation or not, which is the reason we have to store two timestamps (i.e., one for `last_arrived_min` and one for `last_arrived_max`).

As previously described, it is sufficient to check the minimum interarrival time on every reception of a CAN message. Checking the maximum interarrival time is different because it has to be done in every activity cycle. If we would check it on reception of a new message, we would potentially detect the violation later than specified, and crash failures would remain even undetected. If we detect a violation of the maximum interarrival time, we report it and update `last_arrived_max`. This variable contains either the timestamp of the last successful reception or the timestamp of the last reported violation.

Value domain

As described in the previous section, whenever a new CAN message arrives from a CAN segment, the respective CIS tries to lookup the routing configuration associated with the CAN ID of the newly arrived message. If the search is not successful, a CAN node sent a message with an identifier which is not valid for the given CAN segment. In that case the router does not further process the message and sends an error report to the MU. If the lookup is successful, the specified check function for the message content is called. Whenever the message does not pass the check, the violation will be reported and the message is not further processed.

Test framework

We developed a test framework to validate the fault detection and fault containment mechanisms of the CAN router. The framework consists of a 4-port router, 4 independent CAN buses, a MU and several CAN test nodes (CTNs) that generate CAN messages according to a predefined CAN traffic pattern. We designed the MU in a way that it not only takes care about configuration and collecting violations, but also initiates experiments, monitors all CAN buses individually, stores the test results and performs a preliminary analysis. Tools take the preliminary analysis results and visualize [23] the output data off-line.

Test setup

We use an Altera Stratix III Devkit to host the complete test framework, i.e., the CAN router design, the CTN and CAN buses. Figure 3 gives an overview of the test framework.

- *CAN router design:* The CAN router design consists of four CISes and a MU which are connected by a TTNoC in a configuration that allows each CIS to send to and receive from all other CISes and the MU. The TTNoC communication schedule guarantees timely delivery of all CAN messages and minimum interarrival time and maximum interarrival time violation messages within the router, especially during maximum load. Additionally, each CIS is able to handle the maximum load arriving from either the TTNoC or the CAN bus. Specifically for the test framework, the MU contains four independent CAN controllers for monitoring all CAN buses within the design and a DDR2 controller which is attached to a 1 GB memory module for storing test results. All CISes and the MU are accessible over serial communication (UART), and the MU is also accessible by 100 Mbit/s Ethernet for high volume data up/download.
- *CTN:* A CTN emulates a CAN-based device. Each of the CTN is realized by a small Nios II softcore CPU system and a CAN controller to generate or consume CAN messages. The 8 byte payload of each generated CAN message contains a timestamp that is set shortly before the CAN message is transferred to the CAN controller. In case a CAN message transmit attempt fails (i.e., another CAN message with a higher priority is sent simultaneously), the CAN controller will retry sending indefinitely until it succeeds. Each CAN controller has a transmit FIFO buffer with a length of 30 messages. In case the FIFO buffer is overrun, new messages are lost. All CTNs are also connected to the TTNoC; specifically they can receive configuration messages from the MU.
- *CAN buses:* All CAN buses in the test framework operate independently from each other at a baud rate of 256 Kbit/s. While each node on a specific CAN bus can send and receive CAN messages, the MU is only allowed to receive them.

We use the TTNoC global time [24] as the time base for all timestamps. In our design the granularity of this time base is 2^{-21} s.

Test application

The test application conducts experiments using the described setup. An experiment is defined by *experiment parameters* that consist of the router configuration (see Section Implementation of fault detection and isolation, Listing 1) for the CISes and CAN traffic patterns for each CTN. A CAN traffic pattern describes CAN message contents and message rates over the duration of an experiment (i.e., message rates may change during the progression of an experiment). A single execution of an

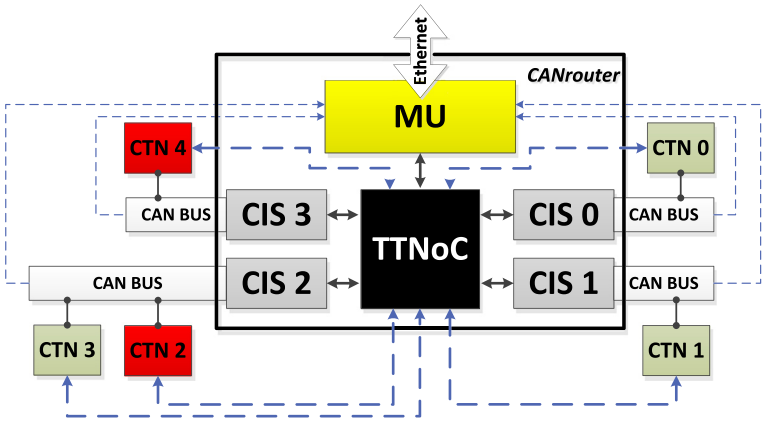

Figure 3 Test framework.

experiment is called *experiment run*. By design, the MU controls the hardware reset of all CISes and takes care of (re)configuring them. After all components of the test framework are configured and operational, the MU logs minimum interarrival time and maximum interarrival time violations from the CISes. For the test framework, we realized the MU software to

- control experiments: The MU autonomously controls the execution of a large number (i.e., several thousands) of experiment runs.
- monitor all CAN buses: Any observed CAN message is timestamped and stored for analysis in the main memory.
- log all minimum interarrival time and maximum interarrival time violations: Any observed violation emitted from a CIS is timestamped and stored for analysis.
- perform preliminary analysis of collected data: The MU evaluates the collected data by periodically calculating message rates according to a specified observation time (e.g., each 0.25 s). Also the MU logs dropped CAN messages (i.e., messages that appear on the source CIS but do not appear on the destination CIS), for example, caused by the router's fault isolation with respect to minimum interarrival time violations or masquerading failures.

Those additions to the MU software are usually not present in the CAN router. However, they do not influence any of the router's characteristics that we want to evaluate: After the CISes are configured, the MU does not take part in the actual service of the router and only collects experiment relevant data.

We divide an experiment into several consecutive phases as depicted in Figure 4. In the *startup phase* all CISes and CTNs remain in the reset state until the MU is configured for the experiment. Configuration is done by commands issued over Ethernet to the MU. Following the configuration, the MU releases the reset of the CISes and CTNs. Then an experiment run starts at the *sign-on phase* where the CISes and CTNs boot and start to send periodically alive messages to the MU. The MU realizes startup synchronization by waiting until all nodes have sent at least one alive message. Eventually all CISes and CTNs have indicated their readiness to the MU and the *configure phase* starts. There the MU sends routing and rate constraint information to all CISes. After all configuration messages are sent, the MU begins the *experiment phase* by sending a start message to all CISes and CTNs which immediately get operational at the same global time instant. During the experiment phase the MU

- collects minimum interarrival time and maximum interarrival time violations reported by the originating CISes.
- calculates CAN message end-to-end latencies. Latency measurements start when a CAN message is queued for transfer in a CTN (i.e., before bus arbitration on the source CAN bus) and stop when the MU observes the message on the destination CAN bus.
- counts dropped CAN messages.

Each CTN notifies the MU after it has finished its CAN traffic pattern. The MU waits until all CTNs have completed which also marks the end of an experiment run. In case the last experiment run is complete, the *analysis and presentation phase* concludes the evaluation of an experiment. Otherwise, the MU starts the next experiment run and issues a reset of the CISes and CTNs. Following to the reset, execution continues at the sign-on phase. The analysis and presentation phase is the last phase of an experiment, where statistical parameters (e.g., mean message throughput, mean end-to-end latencies, ...) are calculated from the individual experiment runs and presented to the user. All of the collected data can be downloaded over Ethernet or serial communication (UART) for further analysis.

Experiments

For the evaluation of the CAN router and its proposed means for fault detection and isolation we conducted experiments. In the following we define the hypotheses for the router, define the individual experiments, and present the gathered results.

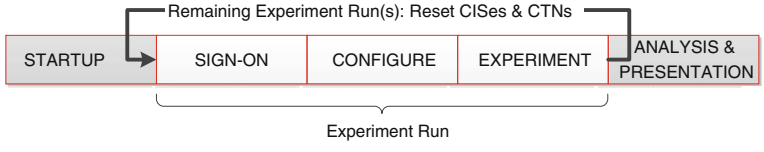

Figure 4 Test framework timeline.

Hypotheses

The following hypotheses were evaluated by the experiments:

Hypothesis 1. *Fault Detection. The router detects the faults in the value and time domain that are specified in Section Fault hypothesis. These include stuck at dominant/recessive failures, asymmetric bit-flip failures, crash/omission failures, babbling idiot failures, and masquerading failures. Additionally, the router detects failures that that are covered by standard CAN controllers (e.g., cyclic redundancy check (CRC) failures, bit-stuffing failures). Detected faults are reported to the MU.*

Hypothesis 2. *Fault Containment. With the exception of late message failures the router contains detected faults in the value domain as well as in the temporal domain. A violation of the maximum interarrival time can only be detected and reported. Thus the late messages will be delivered, while all other messages violating their specified properties are discarded at their source CIS.*

Hypothesis 3. *Latencies. By means of the minimal interarrival time it is possible to specify upper bounds for message latencies. As the number of messages is constrained, there is a bounded influence a potentially faulty node can have on messages from nodes connected to a separated CIS.*

Experiments

In order to evaluate the hypotheses, we conducted two experiments by using the previously described test framework. Each of the experiments is executed 10,000 times to minimize stochastic effects: Each experiment run (i.e., a single execution of an experiment) is carried out with the exact same experiment parameters. Even though CAN messages are event-triggered and occurrences are sporadic, we decided for our experiments to use periodic CAN messages only. By setting and controlling a specific CAN message rate, we are able to evaluate corner cases (i.e., right before and right after minimum interarrival time or maximum interarrival time violations occur) and investigate on interference effects among different prioritized CAN messages.

Experiment I

The experiment lasts 12 s and uses the following routing configuration: CTN 1, 2, and 4 send CAN messages to CTN 0. Concerning the value domain, error-free CTNs send only CAN messages where the CAN ID matches the CTN node number: e.g., CTN 1 is only allowed to send messages with ID 1. CTN 0 is only an receiver and does not produce any CAN messages.

Regarding the temporal domain, messages originating from CAN buses on CIS 0, 1 and 3 are not constrained, while messages with CAN ID 2 originating from the CAN bus on CIS 2 have a minimum interarrival time of 125 Kbit/s and a maximum interarrival time of 15 Kbit/s specified: i.e., according to the router's configuration only rates between 15 Kbit/s and 125 Kbit/s are valid.

CTN 1 generates CAN messages at a constant rate of 37.5 Kbit/s during the whole experiment phase. CTN 4 also sends at a constant rate of 37.5 Kbit/s CAN messages, but with alternating CAN IDs 1 and 4. This results in two 18.75 Kbit/s CAN message streams where only the one with CAN ID 4 is valid according to the router's configuration. Further, CTN 2 sends at first at a constant rate of approximately 8 Kbit/s. Starting from second 2.25 the traffic pattern rate function passes over into a linear ramp where the message generation rate is gradually increased each 0.25 s until it reaches 220 Kbit/s at the end of an experiment run. The following function describes the CAN message rate of CTN 2:

$$r_{CTN=2,\ CAN\_ID=2}(t) = \begin{cases} \min_{\text{rate}} & t \leq a \\ k \cdot t + \min_{\text{rate}} & a < t < b \\ k \cdot b + \min_{\text{rate}} & t \geq b \end{cases}$$
$$t \in \{0, 0.25, 0.5, \ldots, 12\}$$

CTN 3 is unused in this experiment and does not generate any CAN messages. This experiment contains in total two erroneous CTNs: CTN 4 violates the value domain and CTN 2 violates the time domain. Those two nodes are also marked red in Figure 3.

Experiment II

Experiment II is the same as Experiment I with the only difference that CTN 3 is active and generates CAN messages with CAN ID 3 at a constant rate of 37.5 Kbit/s. This additional node neither violates the time nor the value domain according to the router's configuration, but there is now a CAN segment with two senders CTN 2 and CTN 3) attached to CIS 2. There will be interference effects between CAN ID 2 and CAN ID 3 messages. This experiment compares classic CAN bus with segmented CAN bus behavior (as established by the CAN router) in terms of fault containment and end-to-end message latencies during temporal violations. Referring to CAN message interference effects, the two experiments are well comparable, because they only differ in a single experiment parameter.

Results

Here we present the results we gathered from our experiments. A detailed interpretation and discussion of the results follows in Section Discussion. The following figures present the arithmetic mean of all experiment runs for a specific experiment.

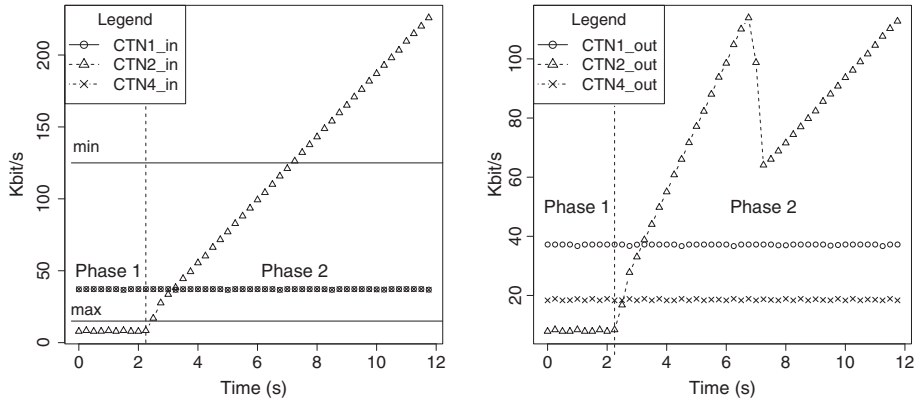

Figure 5 **Exp. I: Router input and its corresponding output.**

Experiment I

Figure 5a depicts the input traffic applied by the CTNs to their CAN buses. CTN 1 and 4 send at a constant rate, while the message generation of CTN 2 is divided into two phases. In the first phase, it sends too slow and violates the maximum interarrival time. In the second phase, it constantly increases its CAN message generation rate which leads to minimum interarrival time violations. Note that the traffic pattern of CTN 4 consists of messages with the CAN ID 1 and 4, though CAN ID 1 is not in the specified set of CAN IDs for the corresponding CIS.

Figure 5b shows the output on the CAN bus attached to CIS 0. The output of CTN 1 is constant and not influenced by any erroneous node. The router forwards only half of the total CAN messages generated by the erroneous node CTN 4 to the CAN bus on CIS 0: Every second message of CTN 4 has the valid CAN ID 4, while the other half has the invalid CAN ID 1 (i.e., only CTN 1 is allowed to send with CAN ID 1).

The send rate of CTN 2 at the source CIS 2 corresponds to the rate at the destination CIS 0 until CTN 2 starts to violate its specified minimum interarrival time. Starting from that instant a pattern occurs that oscillates to an upper bound.

Figure 6a shows CTN 2's number of messages violating the specified minimum interarrival time. In Experiment I the first violations start at the same instant where the input pattern (cf. Figure 5a) of CTN 2 crosses the horizontal line of the bandwidth limit defined by the minimum interarrival time. As there is no difference between the experiments concerning maximum interarrival time violations, they will be shown for Experiment II.

The oscillating output of CTN 2 can be explained as follows: In our test setup messages are sent periodically with different period lengths l. Figure 6b gives an illustrative example where period lengths of four, three, two, and one are examined. The interval in which messages are blocked due to violations of the minimum interarrival time are pictured as gray boxes. Successful send instants are denoted by arrows with an arrowhead, whereas blocked messages contain only an arrowtail.

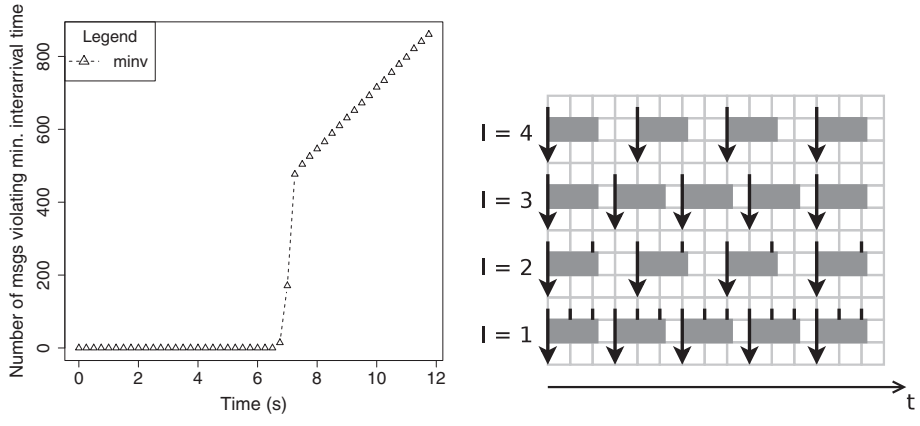

Figure 6 **Exp. I: Temporal violation and oscillating behavior.**

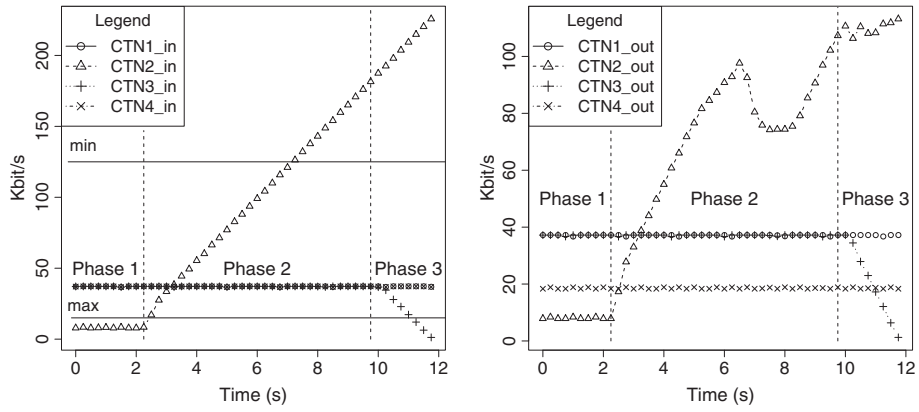

Figure 7 Exp. II: Router input and its corresponding output.

In the period with the length 4, messages are sent too slow to achieve a decent throughput. If messages are sent faster in the period with length 3, a higher number of messages is successfully sent, because successfully sent messages get closer to the gray boxes. In the output (cf. Figure 5b) this can be seen as a rising edge. In the period with the length 2, messages are sent a bit too fast and every second message gets blocked. Although we are trying to send more messages in the period with the length 2 than in the period of the length 3, less messages pass the temporal validity check. This corresponds to the falling edges of the oscillating output. If messages are sent even faster, 2 out of 3 messages get blocked in the period with the length 1, but as messages are sent that fast, a spot shortly after a gray box gets hit and therefore the throughput increases. This continues and we see again a rising edge until the message coming closer to the gray box gets blocked, which leads to a falling edge again. As the periods get shorter and shorter over time, the influence of blocked messages gets smaller and smaller.

Experiment II

The input pattern (cf. Figure 7a) of Experiment II is very similar to the previous experiment with the exception that CTN 3 also generates messages at a constant rate. The experiment is now divided into three phases. The first phase is equivalent to Experiment I. In the second phase CTN 2 increases its bus load and starts to violate its minimum interarrival time. In the third phase, CTN 2 keeps increasing the send rate and therefore starts to influence CTN 3 on the shared CAN bus.

Figure 7b shows the output on CIS 0. The output of CTN 1 is constant and not influenced by any erroneous node. The same applies for CTN 3 until its messages get blocked by the erroneous node CTN 2. Again, the router forwards only half of CTN 4's messages.

Figure 8a shows that, contrary to Experiment I, there are minimum interarrival time violations caused by messages from CTN 2 before the actual configured minimum interarrival time. Those violations increase slowly starting from approximately second 5 until they sharply

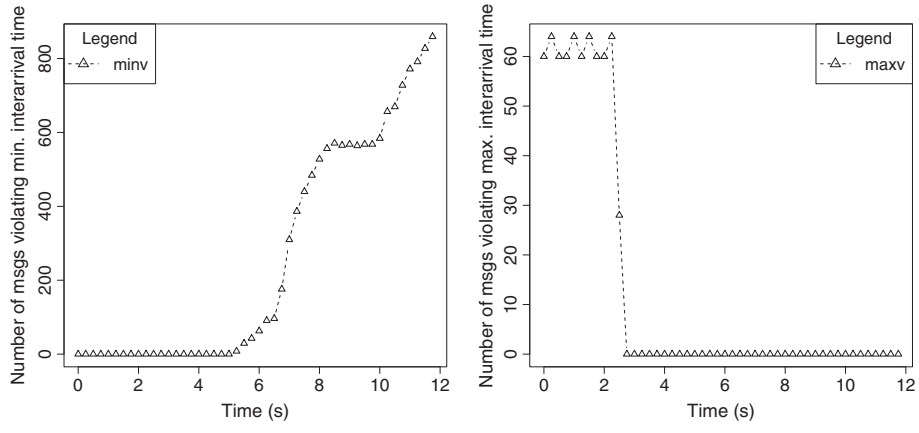

Figure 8 Exp. II: Temporal violations of CTN2.

go up when the specified minimum interarrival time is undershot. This difference of the two experiments will be discussed in Section Discussion.

Figure 8b gives an overview about the number of messages violating the maximum interarrival time. In the first phase of Figure 7a the number of messages from CTN 2 is below the specified maximum interarrival time. Therefore, every missing message in that phase gets reported. In the second and third phase the erroneous node CTN 2 sends messages in the specified range, or even faster. Therefore, the number of violations of the maximum interarrival time drops to zero.

Figure 9 depicts the end-to-end latencies of CAN messages measured from the time the message was queued for transmission on the source CAN segment until it gets successfully received on the destination bus. As in phase 3 messages from CTN 3 get blocked by the erroneous node CTN 2 on the source bus, the latencies of messages with CAN ID 3 dramatically increase. At the same time the latencies of messages with a lower priority drop to the next lower level (i.e., CTN 4's latencies drop to the previous level of CTN 3).

In all our experiment runs we did not encounter a single experiment run where the router lost valid CAN messages.

Discussion

Hypothesis 1 concerning fault detection was confirmed by the experiments. In the temporal domain, the router has detected violations of the maximum interarrival times (Phase 1 in Figure 7b) and violations of the minimum interarrival times (late Phase 2, and Phase 3 starting approx. at second 7 in Figure 7b).

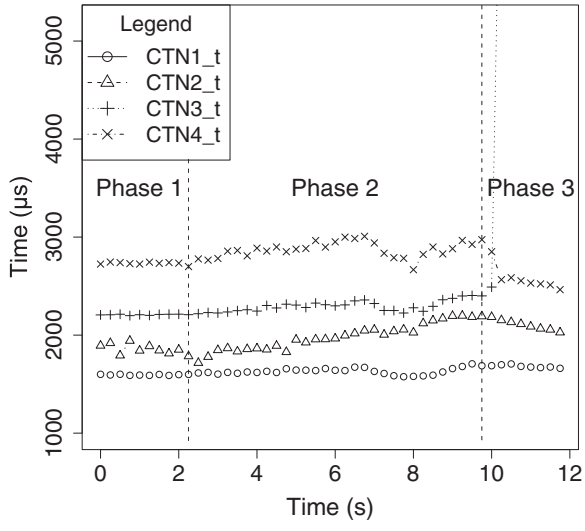

Figure 9 Exp. II: End-to-end latencies.

In the value domain, the routing configuration permits the detection of invalid message identifiers, while the recognition of errors in the user data occurs using message checks. In addition, the CAN controllers [25] detect errors (e.g., stuffing error, CRC error, acknowledgment error) that are reported to the MU.

The experiments also provide evidence for hypothesis 2 and the fault containment of the CAN router. At each CAN port messages are blocked where value or timing failures are detected by the CAN controller or the CIS' CPU. In contrast to a bus-based system, no faulty messages or error frames are relayed by the CAN router. Thus, faulty messages do not cause inaccessibility times (e.g., due to bit errors, stuffing errors, CRC errors, form errors and acknowledgment errors) as described for bus-base systems in [26].

Besides eliminating these inaccessibility times, the fault containment coverage of the CAN router is significantly higher compared to bus-based systems. In a bus-based system, local error detection mechanisms are assumed to shutdown a CAN node that is affected by a fault. However, error detection mechanisms should be part of separate fault-containment regions in order to ensure that the error detection mechanisms are not impacted by the same fault that caused the message failure [27].

In addition to the containment of errors detected by the CAN controller at a CAN port, the CAN router blocks messages with invalid message identifiers and messages that violate the minimum interarrival time. The experiments show the impact of timing failures on messages on the same CAN segment and messages from other CAN segments. In the same CAN segment, message timing failures with a given priority affect all messages with lower priorities (cf. Figure 7). In contrast, the effect on messages from other CAN segments is bounded by the minimum interarrival times. The worst-case delay for a given message occurs when the bus of a CAN segment is not idle at the time of the transmission request and all higher priority messages are sent according to their minimum interarrival times. The experiments show that even lower priority CAN messages can introduce additional latencies for higher-priority ones (i.e., when a high priority message is queued for sending, but a lower priority message is currently being transmitted on the non-preemptive bus). The additional delay, regardless whether caused by a lower or higher priority message, can push two messages with the same CAN ID from a sender closer together. This behavior occurs when the last CAN message was delayed and the current one is not. In case those messages come too close, a minimum interarrival time violation occurs, even though the actual message rate (assuming a sufficiently large observation time) is not violated. For example, Figure 8a shows this behavior for messages sent by CTN 2, where violations

of the minimum interarrival time occur before CTN 2 actually exceeds its allotted message rate. Consequently, the worst case delay (beside message size and CAN bus bitrate) must also be considered for the minimum interarrival time, if a maximum message rate should be guaranteed. This worst case delay can be computed using existing approaches for response time analysis in CAN [28].

For the given scenario of Experiment II, Figure 9 shows experimental results for these bounded effects of timing failures on the end-to-end latencies. Thus, hypothesis 3 was confirmed by the experimental results.

Due to its benefits concerning error detection and fault isolation, the CAN router can improve the reliability of existing CAN-based systems. Furthermore, for certain safety-relevant applications the CAN router can provide an alternative to more costly protocols (such as FlexRay in the automotive industry). If compatibility to existing CAN nodes is not required, two redundant CAN routers can be employed to tolerate an arbitrary single failure of a fault-containment region.

Conclusion

In this paper we showed that standard CAN exhibits limitations with respect to reliability, diagnosis and scalability. We presented an intelligent CAN router based on a star topology that allows to overcome these existing limits. We experimentally validated the proposed capabilities and showed fault detection and isolation in the temporal as well as in the value domain. In the time domain this includes monitoring of minimum and maximum interarrival times and enforcing minimum interarrival times. In the value domain the router successfully enforced the permitted CAN identifiers of given CAN segments. We showed that the router enables the extension of application areas of CAN to systems with higher dependability and performance requirements, while still providing legacy CAN interface support.

Competing interests

The authors declare that they have no competing interests.

Acknowledgements

This work has been supported in part by the European research project INDEXYS under the Grant Agreement ARTEMIS-2008-1-100021 and in part by the European research project ACROSS under the funding ID ARTEMIS-2009-1-100208. The responsibility for the content rests with the authors.

Author details

[1]Vienna University of Technology, Vienna, Austria. [2]University of Siegen, Siegen, Germany.

Received: 13 October 2011 Accepted: 23 May 2012
Published: 3 July 2012

References

1. K Tindell, H Hansson, in *Proceedings of the 1st International CAN Conference*, (Mainz, Germany, 1995), pp. 722–728
2. J Rufino, P Verissimo, G Arroz, in *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, (Madison, USA, 1999), pp. 286–293
3. M Livani, in *Proceedings of 6th International CAN Conference (ICC6)*, (Torino, Italy, 1999)
4. J Rufino, P Veríssimo, G Arroz, C Almeida, L Rodrigues, in *Proceedings of the 28th International Symposium on Fault-Tolerant Computing Systems*, (Munich, Germany, 1998), pp. 150–159
5. H Salmani, SG Miremadi, in *Proceedings of the 11th, Pacific Rim International Symposium on Dependable Computing* (IEEE Computer Society, Washington, DC, USA, 2005), pp. 310–316
6. FlexRay Consortium, BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, Volkswagen AG, *FlexRay Communications System Protocol Specification Version 2.1*, (2005)
7. ISO11898 International Standardization Organisation, *Road vehicles—Interchange of Digital Information—Controller Area Network (CAN) for High-Speed Communication*, (1993)
8. M Mateos, P Robin, S Sauvage, V Joloboff, G Madhusudan, Y Bennani, in *Convergence International Congress & Exposition On Transportation Electronics*, (Detroit, USA, 2002)
9. J Barkai, in *Proceedings of Automotive & Transportation Technology (ATT) Congress & Exhibition,* vol. 5, (Barcelona, Spain, 2001)
10. P Veríssimo, J Rufino, L Rodrigues, in *Proceedings of the 10th IFAC Workshop on Distributed Computer Control Systems*, (Semmering, Austria, 1991)
11. P Verissimo, J Rufino, L Ming, in *Proceedings of Symposium on Fault-Tolerant Computing*, (Seattle, USA, 1997), pp. 112–121
12. J Rufino, *Dual-media redundancy mechanisms for CAN. Technical Report CSTC RT-97-01*. (Centro de Sistemas Telemáticos e Computacionais do Instituto Superior Técnico, Lisboa, Portugal, 1997)
13. H Kopetz, R Obermaisser. Computing and Control Engineering Journal. **13**, 156–162 (2002)
14. J Kaiser, M Livani, in *Dependable Computing – EDCC – 3*, (1999), pp. 351–363
15. R Obermaisser, H Kopetz, C Paukovits, in *IEEE Transactions on Industrial Informatics*, (2012), pp. 548–567
16. J Lala, R Harper, in *Proceedings of the IEEE,* vol. 82, issue 1, (1994), pp. 25–40
17. F Cristian. Commun. ACM Journal. **34**(2), 56–78 (1991)
18. J Suwatthikul, R McMurran, in *Proceedings of the International Symposium on Industrial Embedded Systems*, (Antibes, France, 2006), pp. 1–4
19. Association for Standardisation of Automation and Measuring Systems, *ASAM MCD-2 NET Data Model for ECU Network Systems (Field Bus Data Exchange Format) Version 3.1.0*, (2009)
20. R Obermaisser, R Kammerer, A Kasper, in *Proceedings of AmE 2011—Automotive meets Electronics*, (Dortmund, Germany, 2011)
21. A Bondavalli, S Chiaradonna, FD Giandomenico, F Grandoni. IEEE Tansactions on Computers Journal. **49**(3), 230–245
22. CE Salloum, M Elshuber, O Höftberger, H Isakovic, A Wasicek, in *15th Euromicro Symposium on Digital System Design (DSD)*, (Izmir, Turkey, 2012)
23. R Development Core Team, *R: A Language and Environment for Statistical Computing*. (R Foundation for Statistical Computing, Vienna, Austria, 2011). http://www.R-project.org/. ISBN 3-900051-07-0
24. C Paukovits, *The Time-Triggered System-on-Chip Architecture. Ph.D. thesis, TU Vienna*, (2008)
25. P Riekert, F Sprenger, *IFI NIOSII Advanced CAN module. Tech. rep., Ingenieurbüro für IC-Technologie*, (2010)
26. J Rufino, P Veríssimo, in *2nd International, CAN Conference*, (London, UK, 1995), pp. 7.12–7.21
27. H Kopetz, in *Proceedings of the International Symposium on Autonomous Decentralized Systems*, (Pisa, Italy, 2003), pp. 139–146
28. K Tindell, A Burns, A Wellings. Control Engineering Practice Journal. **3**, 1163–1169 (1995)

doi:10.1186/1687-3963-2012-4
Cite this article as: Kammerer *et al.*: A router for the containment of timing and value failures in CAN. *EURASIP Journal on Embedded Systems* 2012 **2012**:4.