Dynamic Configuration of a Time-Triggered Router for Controller Area Network

Roland Kammerer
Vienna University of Technology, Austria
kammerer@vmars.tuwien.ac.at

Roman Obermaisser
University of Siegen
roman.obermaisser@uni-siegen.de

Bernhard Frömel
Vienna University of Technology, Austria
froemel@vmars.tuwien.ac.at

***Abstract*—The time-triggered router for CAN has as its goal to improves the dependability, performance and timeliness of CAN communication. The configuration of the CAN router includes knowledge about the permitted behavior of nodes in the time and value domains. Based on this configuration the CAN router performs fault isolation, diagnosis, message multicasting and message transformations. This paper presents architectural elements and algorithms for the modification of the router configuration at run-time. The router is realized as a Multi-Processor-System-on-a-Chip (MPSoC) and contains a dedicated hardware core for controlling the reconfiguration process. The configuration information is transferred to the cores responsible for the individual CAN segments through a time-triggered network-on-a-chip. We provide a solution for assured reconfiguration with predictable timing and continuity of service during the reconfiguration. An experimental evaluation demonstrates the bounded time for reconfiguration, as well as the seamless and consistent switching to new configurations.**

I. Introduction

Dynamic reconfiguration occurs when the configuration of an application is modified while it is running. Dynamic configuration enables a system to dynamically adapt to changes in the environment and to changes in resource demands or resource availability (e.g., power, time/scheduling, communication bandwidth, memory). Dynamic configuration enables better resource utilization, improved dependability, and the enabling of power-aware system behavior. In a distributed system where node computers are interconnected by a communication network, dynamic configuration involves changes in the use of computational resources (i.e., the allocation of functions to node computers) and changes in the use of communication resources (i.e., the use of communication network for interaction between node computers).

This paper investigates the adoption of a new configuration of the communication resources for a Controller Area Network (CAN) [7] based on a router. As presented in previous work [8], [11], a time-triggered router for CAN exploits a priori knowledge about the communication behavior of nodes in order to improve the dependability, performance and composability in CAN-based systems. The disadvantage of this approach is that a given configuration of the router restricts modifications of the system such as the replacement, addition or removal of nodes and messages.

Therefore, the goal is to modify at run-time the router's a priori knowledge about the communication behavior of nodes such as the temporal properties of messages (e.g., maximum and minimum interarrival times), valid identifiers, recipients of messages, conversion functions and message checks. At the same time, the following fundamental challenges are addressed in the reconfiguration mechanisms:

- *Predictable Reconfiguration Results.* Predictability of the service availability after reconfiguration is required. In particular, the aftereffects of reconfiguration to critical services should be known beforehand. This challenge is also known as the problem of assured reconfiguration [16].
- *Bounded Time for Reconfiguration.* The dynamics of the environment and the system requirements determine the available time for the new configuration. Hence, the time needed to adopt a new configuration needs to be bounded.
- *Continuity of Service during Reconfiguration.* Reconfiguration activities should not disrupt the behavior of subsystems that are not subject to the reconfiguration activities. Also, resource requirements should be satisfied when reconfiguration is in progress. The architecture enables that resource requirements are satisfied during reconfiguration activities and prevents disruptions of service.
- *Consistent Switch to new Configuration.* It is necessary to provide a way to consistently switch between different quality levels and operational modes. Intermediate configurations need to be avoided where subsets of nodes are in the new configuration, while subsets of nodes are in the old configuration.
- *Robust Reconfiguration Mechanisms.* The reconfiguration mechanisms are critical in the sense that a failure of the reconfiguration process has the potential of causing a global system failure.
- *State Preservation [3].* Relevant internal state must be preserved when switching between configurations. For example this includes already stored, but not yet delivered CAN messages, if these messages are still valid in the new configuration.

The paper is organized as follows. Section II gives an overview of the time-triggered CAN router. In Section III we present related work. The explanation of the configuration data structures and algorithms for reconfiguration is the focus of Section IV. Section V discusses the implementation of the CAN router using an FPGA. Section VI is dedicated to the presentation of the experimental evaluation. The paper finishes with a discussion and a conclusion in Section VII

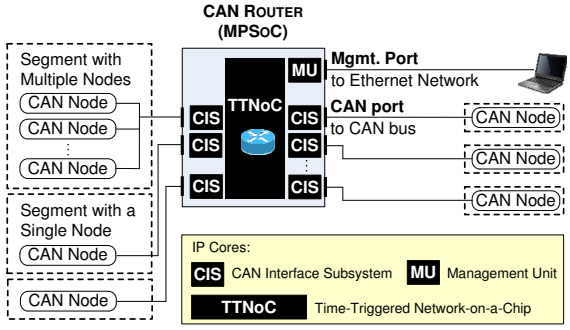


Fig. 1: CAN router

and Section VIII.

II. Time-Triggered CAN Router

Figure 1 shows a system using the CAN router. The CAN router implements a star topology and forwards messages between CAN segments. A CAN segment, consisting of a CAN bus and at least one node, is connected to the router via a router port. Every router port is served by its own hardware component, which consists of a CPU, local memory and a CAN controller. The CPU of such a hardware element executes software used for message processing. We call the combination of hardware and software for a CAN segment a CAN Interface Subsystem (CIS). We store a *local routing configuration* in the memory of each CIS which contains important properties like the set of valid CAN identifiers for a CAN segment and the permitted temporal behavior of allowed CAN messages. Additionally, the router features a Management Unit (MU) which serves an Ethernet based management port. The MU is used for diagnosis (i.e., it stores violations of the specified behavior reported by the CISes) and for management purposes like updating the routing configuration.

A. Basic Services

In the following we provide a short overview of the basic services of the CAN router [8], [11]:

1) Message Rate Control: The CAN router specifies for every message on a given CAN segment two properties defining the permitted temporal behavior. This are the *minimum* and *maximum interarrival time*. The first one defines the maximum rate of messages with a given CAN ID which are allowed to arrive. The second property defines how often a message has to arrive at least. The second property cannot be enforced, but it is valuable for fault detection (e.g., if a faulty node stops sending messages). The system engineer still needs to ensure a bounded jitter of all temporally critical CAN messages on every CAN segment.

2) Message Multicasting: In contrast to a CAN bus where every message is broadcasted, the router allows multicasting. This allows us to use the limited overall bandwidth of 1 Mbit/s [7] more efficiently. With multicasting a message is selectively routed to a defined subset of the CAN segments.

3) Message Scheduling: In order to establish the temporal message order of a CAN bus, every CIS contains a priority queue for outgoing CAN messages. This queue is used by a CIS for buffering and scheduling messages that shall be sent on the respective CAN segment. In case the CAN bus is idle the first element in the queue (i.e., the message with the highest priority) is sent first.

4) Identifier Validation and Translation: The configuration of the router contains an entry for every permitted CAN ID. If a node sends messages with IDs not in the set of permitted IDs (e.g., masquerading as a CAN node from a different CAN segment), the router detects this faulty behavior and reports it to the MU. Additionally, the configuration can contain an entry used for identifier translation. If such an entry exists, the router replaces the original CAN ID with the specified one, before sending it to the destination segments. Translation from basic to extended identifier CAN IDs and vice versa is also supported. Identifier translation is mainly used for the integration of legacy CAN systems.

5) Message Checks: The router provides a predefined library of check functions which can be utilized to validate the content of a CAN message (e.g., to check if a temperature value is in a meaningful range). Application-specific functions can be added to the library pre runtime.

6) Diagnosis and Management: The router contains a dedicated MU for diagnosis and management. If one of the specified properties of a message is violated (e.g., a node sends with an ID not in the set of valid IDs), the source CIS sends a report to the MU. This data is stored and can be used by an engineer for further analysis. The second purpose, and the main focus of this paper, are its management capabilities. With the help of the MU it is possible to update the configuration during run-time.

B. CAN Interface Subsystem (CIS)

The CIS implements the described basic services (with the exception of the diagnosis and management service, which is implemented by the MU). A CIS typically has two purposes, namely the role as a *source CIS* and the role as a *destination CIS*. Usually, a CIS has both roles.

In its role as a source CIS it polls its local CAN controller and checks if a new CAN message has arrived. If this is the case, the source CIS processes the newly arrived message according to its local routing configuration (e.g., checking the minimum interarrival time). After finishing this step, the source CIS hands over the message to the Time-Triggered Network-on-a-Chip (TTNoC).

The TTNoC [12] is an architectural component of the router that interconnects CISes. Every CIS has a dedicated communication channel to every other CIS. The TTNoC uses a static communication schedule, which partitions time into slots that are specified by periods and phases. Within one period all CISes are able to communicate to all other CISes. In order to align the communication activities within one period, a period is divided into phases. A phase is an offset with respect to the associated period. Table I depicts an exemplary communication schedule for four CISes. A sending slot is denoted by S and a receiving slot by R. The subscript denotes the destination for sending slots and the source for receiving slots (e.g., in the first phase CIS 1 sends to CIS 2). The TTNoC

	TTNoC Period					
	Phase 0	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5
CIS1	S_2	S_3	S_4	R_2	R_3	R_4
CIS2	R_1	S_4	S_3	S_1	R_4	R_3
CIS3	S_4	R_1	R_2	R_4	S_1	S_2
CIS4	R_3	R_2	R_1	S_3	S_2	S_1

TABLE I: Exemplary communication schedule

is implemented in a mesh topology that consists of so called fragment switches [9]. These switches forward data according to a routing header that precedes the actual data and specifies the direction to take in the mesh (i.e., worm-hole routing [9]). Consequently, multiple sending and receiving actions within one phase are permitted, as long as the routes in the mesh do not collide.

In its role as a destination CIS, the CIS checks if there are any new messages from the TTNoC. If this is the case, the destination CIS inserts the newly arrived messages in a priority queue. This step is required to establish the message order of a CAN bus (i.e., CAN messages with a higher priority are sent before messages with a lower priority). The final step for a destination CIS is to extract the first element of the priority queue and hand it over to the local CAN controller which sends this message on the bus of the CAN segment.

C. Temporal Sequence of Interactions

As CAN is an event-triggered protocol, messages arrive at the router ports at sporadic points in time. Every CIS polls its CAN controller faster than the minimum possible interarrival time of CAN messages at 1 Mbit/s. Internal processing of the CAN router is strictly time-triggered. This includes the processing of CAN messages as well as sending messages in the TTNoC. Advantages of a time-triggered behavior include bounded delays introduced by the router, timely message forwarding (i.e., without collisions in the TTNoC), and a global time base that can be used by the basic services of the router (e.g., checking for minimum interarrival time violations). For dynamic reconfiguration a time-triggered system provides the advantage of supporting consistent points in time when configurations can be switched.

We denote the cycle of periodic message processing in a CIS as a *round of activity*. All CISes are synchronously triggered with a frequency that is higher than the maximum rate of arriving CAN messages at 1 Mbit/s. This guarantees that no CAN message is lost due to the time-triggered operation.

Figure 2 depicts the temporal alignment of rounds of activities to the actions within these rounds and the communication periods of the TTNoC. Within one round of activity the communication schedule of the TTNoC is executed once. The TTNoC period length is equal to the length of a round of activity. By guaranteeing that the TTNoC interactions (i.e., sending and receiving messages to/from the TTNoC) are finished within the first half of a round of activity, messages to be sent will be delivered in the second half of the TTNoC schedule. The TTNoC operates at system frequency, which is fast enough to guarantee that messages are delivered to the destination(s) before the next round of activity starts. Nevertheless, a CAN message processed in round of activity n at a source CIS, is received at round of activity $n+1$ at the destination CIS(es).

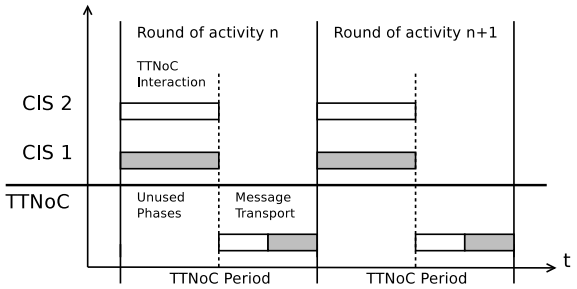


Fig. 2: Temporal alignment of activities

III. Related Work

As CAN is widely used in many application domains, there have been academic publications as well as industrial solutions to overcome the existing limitations of CAN. The advantage of these solutions is that they do not need a priori knowledge about permitted CAN message properties. Existing work ranges from simple CAN bus guardians [4], [6], to replicated CAN buses [14], [15], and reconfigurable transmission media [5]. CANcentrate [1], which is a solution that shares an important design decision with the CAN router, namely the use of a star topology, implements a logical AND function and distributes the calculated result to all connected CAN buses. In order to discriminate signals from individual CAN nodes from the calculated result, every node needs two CAN controllers, which increases the hardware costs and makes legacy system integration more difficult. While the proposed solutions increase the dependability of CAN to a certain degree, none of them sufficiently deals with fault detection and isolation in the temporal domain. For example, a faulty node that sends high priority messages too fast (i.e., a babbling idiot) can still drag down a whole CAN bus or the whole system. Increased fault detection and isolation by the CAN router requires an efficient mechanism for updating a priori knowledge of the router. To our best knowledge none of the existing work combines the high degree of fault detection and isolation with an efficient and temporally predictable reconfiguration mechanism proposed in this paper.

IV. Configuration and Reconfiguration

In this section we first discuss the configuration structures used for the CAN router, then we focus on the system startup, give insights on how we handle on-the-fly reconfiguration and state which reconfiguration scenarios we expect during runtime.

A. Configuration

The CAN router consists of n CISes ($n \geq 1$) where every CIS contains two configurations that define its behavior. In its role as a source CIS it uses the routing configuration and in the role as a destination CIS it uses the queue configuration to establish the message order of CAN messages.

1) Routing Configuration: The routing configuration of a given CIS c ($0 \leq c \leq n-1$) is a set R_c, that contains a tuple E_m (i.e., routing entry) for every permitted CAN message m with a CAN ID i in the overall set of CAN identifiers $I \subseteq \{0, \ldots, 2^z - 1\}$. The number z defines the number of CAN identifier bits (i.e., 11 for basic identifiers and 29 for extended identifiers). A tuple E_m is defined as follows:

$E_m = \langle id, min, max, chk, tran, to \rangle$, where
$id \in I$ and $min, max \in \mathbb{Q}, chk \in \mathbb{N}$ and $tran \in I \cup \{inval\}$
and $to \subseteq \{0, \cdots, n-1\}$

The semantics of a tuple E_m are given as follows:

- *id:* CAN identifier associated with the routing configuration entry.
- *min,max:* minimum and maximum interarrival times.
- *chk:* An index in an array of check functions that will be used to validate the content of a CAN message.
- *tran:* CAN identifier used for name translation. An invalid CAN ID is used in case a translation is not required.
- *to:* A set that contains the destination CIS(es) of a message m, where $c \notin to$ (i.e., messages are not forwarded to the CIS itself).

2) Queue Configuration: In order to reestablish the message order of a CAN bus, every CIS contains a priority queue. The queue configuration for a CIS c is a set Q_c, that contains for every CAN message m that can be received from a source CIS a tuple P_m (i.e., a priority queue entry). A tuple P_m contains the following entries:

$P_m = \langle id, migrate \rangle$, where $id \in I$ and $migrate \in \{0, 1\}$

The fields in P_m are defined as follows:

- *id:* CAN identifier associated with the queue configuration entry.
- *migrate:* A flag that is set if the corresponding CAN message is syntactically and semantically equivalent to the message in the previous configuration with the same CAN ID. This flag is used to accomplish state preservation between the current configuration and the new one to be deployed. For example consider a scenario where a queue already contains a message from the power train of a car and that these messages do not change syntactically or semantically between reconfigurations. Then the router has to provide means to migrate these messages between the old and the new configuration. A detailed discussion about switching configurations and the role of the migrate flag follows in Section IV-C3.

In the following we show how to determine the queue entries for a given destination CIS. This is on one hand important to generate a queue configuration for every CIS and on the other hand to bound the number of queue entries per CIS. The latter is a necessity to dimension the required queue length for an actual implementation of the CAN router. Note that newer messages overwrite older messages in the queue if they share the same CAN ID, whereas there is at most one message per configured CAN ID in the queue. Therefore, we have to compute the number of distinct CAN IDs for dimensioning the queue, but do not suffer from queuing effects of messages with the same CAN ID. At the time when all routing configurations from all CISes are known, it is possible to calculate the maximum number of entries in the priority queue for a given CIS d.

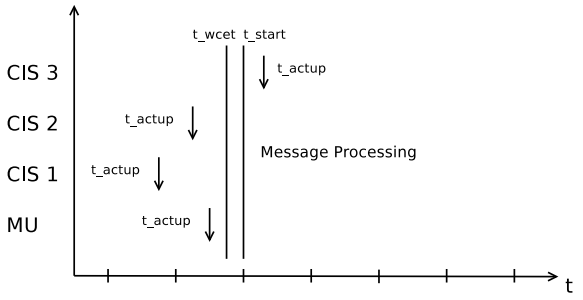


Fig. 3: Startup time line

The set M_d contains all CAN IDs for which the CIS d has to provide space in its queue configuration. The cardinality $|M_d|$ gives the number of slots in the priority queue. The function $transid()$ extracts the translation ID $tran$ (if one is specified, otherwise the id) and the function $getto()$ extracts the set containing the destination(s) of a CAN message for a given routing entry E_m. In order to simplify the generation of M_d, we use the helping set X_d that contains all routing entries from all CISes except the ones from d itself because CISes do not route to themselves (i.e., no loopback).

a) Helping Functions:

$$E_m = \langle id, min, max, chk, tran, to \rangle \text{ , then}$$
$$getto(E_m) = to$$
$$transid(E_m) = \begin{cases} tran & tran \in I, \\ id & otherwise \end{cases}$$

b) Set Definitions:

$$X_d = \{E_m \mid E_m \in R_c, 0 \leq c \leq n-1, c \neq d\}$$
$$M_d = \{transid(E_m) \mid E_m \in X_d, d \in getto(E_m)\}$$

B. System Startup

Guaranteeing a consistent point in time where all operable system components are successfully started is a prerequisite for further message processing, configuration and dynamic reconfiguration. A second essential property is the robustness of the startup. Failed components should not inhibit the startup of correct components.

After the router is turned on, as well as after a hardware reset, the global time service of the underlying platform [13] provides a consistent time base, that always starts at zero, for all components. For reasons of simplicity we do not distinguish between a CIS and the MU if it makes no difference and denote them as *components* in the rest of this section. During startup all components initialize their hardware (e.g., by setting up communication channels or initializing the local CAN controller or Ethernet controller). As the time for this initialization phase is known for all components, the instant measured in

global time ticks where all correctly operating components are up and running is known as well. This point in time, denoted as t_{wcet}, is the maximum of the worst case startup times for all components. We align that instant in time to the next multiple of a TTNoC period and store this value in t_{start} in the memory of every component. We denote the instant in time a component is actually up and running as t_{actup}. After the initialization phase has passed, every component checks if it did not miss the specified startup time (i.e., if $t_{actup} \leq t_{start}$). If a CIS missed that point in time, it does not participate in any further message processing. If a CIS started up in time it is ready to process messages starting from the next round of activity. In any case, all components check at the beginning of a round of activity if the current time is higher than t_{start}. Only if that is the case they actually process messages. The purpose of this check is that all components wait until the specified time before they send their first message. Without waiting for t_{start} very fast components would start sending messages before giving other components time to successfully start up. For example the software of the MU is more complex than a CIS' software and therefore takes longer to start up. Without waiting till t_{start} has expired, CISes would start sending messages before the MU has started up. Figure 3 summarizes the startup of a CAN router, where all components, with the exception of CIS 3, meet their startup time. We use the TTNoC's period as the time unit on the x-axis.

There are two options to deploy the initial system configuration. The first option is to store the initial configuration (e.g., broadcast every CAN message) in the memory of each CIS and set a flag that this local configuration is valid. The second option, if that flag is not set, is that all CISes wait for an initial configuration from the MU.

C. On-the-fly Reconfiguration

In the following we describe the activities of the CISes and the MU during reconfiguration and then put a focus on switching from a current configuration to a new one.

1) CAN Interface Subsystem (CIS): Every CIS maintains storage for two configurations. One buffer for the currently used configuration (*current*) and one for the new configuration (*shadow*) which is not currently used. When a CIS receives a *configuration request* message from the MU, it resets the state machine which manages the reconfiguration steps and waits for the actual entries. The configuration messages from the MU are then handled according to a protocol (cf. Section V for details) and written to the shadow buffer. Storing the new configuration does not influence the routing behavior (i.e., the router still uses the unaltered configuration from the current-buffer). The final step is to switch consistently to the new configuration. This is done when the CIS receives the *configuration commit* message from the MU which was broadcasted to all CISes in the same round of activity. Basically all CISes swap the current and the shadow buffer. Therefore, the old shadow configuration becomes active and the old active one becomes the new shadow configuration which will then be used for the next reconfiguration. A discussion on bounding

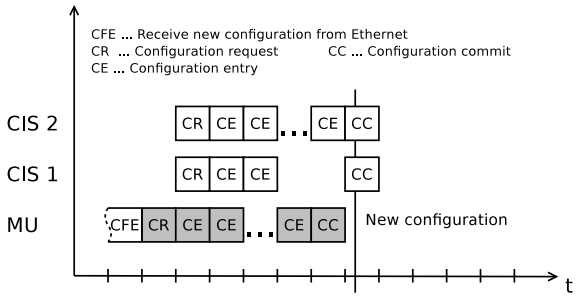


Fig. 4: Reconfiguration time line

the time from a configuration request to the actual switch depending on the number of configuration entries is given in Section VII.

2) Management Unit (MU): When used for reconfiguration the first step for the MU is to receive a new configuration via its management port over Ethernet and to temporarily store it. When the complete configuration is received, the MU broadcasts a *configuration request* message to all CISes. Then the MU broadcasts the individual configuration to every CIS. After the new configuration is deployed on every CIS, the MU broadcasts a *configuration commit* message and the new configuration gets active in all CISes consistently in the same round of activity. Figure 4 shows a reconfiguration sequence, where the MU sends a new configuration within an exemplary CAN router consisting of CIS 1, CIS 2 and the MU. The cut in the time line marks the instant when the new configuration becomes active. We use the duration of a round of activity as the unit of time in the figure. Sending a message is denoted by a shaded box, while unshaded boxes symbolize receiving messages.

3) Switching Configurations: The configuration switch is carried out consistently in the round of activity where all CISes receive the broadcasted *commit configuration* message from the MU. Before that, the CISes simply store the new configuration to their shadow buffers, while they keep processing CAN messages according to their current configuration. The old and the new configurations in all CISes are switched consistently after receiving the commit messages. This implies that messages from the MU are handled before CAN message processing. Switching to the new routing configuration requires to swap the current configuration with the shadow configuration. Switching queue configurations – while guaranteeing continuity of service – is more involving: Messages already in the priority queue that do not have an entry in the new queue configuration are obsolete and have to be removed. Messages which are already in the queue and have their *migrate* flag set in the new configuration are still valid and have to be kept in the queue. Additionally, in the current round of activity, destination CISes might receive messages from source CISes which have been processed according to the old configuration due to the delay of one round of activity discussed in Section II-C. Therefore, we insert only messages into the priority queue where the queue configuration contains a set *migrate* flag, and discard the rest. After this migration phase, which lasts one round of activity, the *migrate* flag is

not further used.

D. Reconfiguration Scenarios

For reconfiguration we identified the following typical scenarios:

- Adding new messages: Even if a single new message is added to the configuration of a CIS, this new configuration impacts the configuration of at least one – potentially more – destination CIS(es). Every new message needs an entry in the routing configuration of the source CIS, and one entry in the queue configuration of every destination CIS that receives this new message.
- Removing messages: Removal of valid messages also affects both the routing and queue configuration. By removing it from the queue configuration of the destination CIS it is guaranteed that old messages already in the priority queue are also removed.
- Changing properties of a message: Changed properties of a message like new destination CISes or changed minimum and maximum interarrival times are considered as a sequence of removing the old entry and adding a new entry. The resulting configuration can then be written to the CISes in one reconfiguration cycle. As for adding and removing messages the new configuration is only sent to affected CISes (i.e., selective reconfiguration).
- Switching to a spare CIS: Increased fault detection, isolation and tolerance are goals of the CAN router. To accomplish these goals one might deploy a redundant CAN segment connected to a separated router port. If the MU detects that the original CAN segment has a permanent failure, it has to switch to the spare CIS. The diagnosis capabilities of the MU provide the foundation for fault recovery. A concrete strategy is out of scope for this paper and depends on the application area of the CAN router. 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 [2]. Initially the MU would provide very similar configurations for the main CIS and the spare one. The only exception is that the spare one does not contain any specifications about the destination CISes. With that configuration the spare CIS would check all the specified properties and report violations to the MU, while simply not forwarding any message. These reports can then be used by the MU before switching to the spare CIS. If and only if the spare CIS is operational it switches to the spare one. Switching itself contains removing all entries of the failed CAN segment and applying a configuration with valid destinations for the spare CIS.

V. Implementation

A. Routing Configuration

The routing configuration is stored in every CIS. We call the memory where it is stored the *routing storage*. This routing storage is implemented as a sorted array consisting of C-structs. Every struct contains the configuration for exactly one CAN ID. Please refer to Listing 1 for an example routing entry.

Listing 1: An example routing entry

```
typedef struct {
   uint32_t can_id;
   time64 min;
   time64 max;
   ...
   uint16_t message_check;
   uint32_t translation_CANID;
   uint16_t forward_to;
} routing_entry;
```

All entries are sorted according to their CAN ID. The sorted array allows to look up an entry in $O(log(n))$ (i.e., a binary search). A direct mapping of CAN ID to a position in the routing storage is not feasible because that would require 2^{29} entries in the routing storage which would be too memory consuming. A linear search would consume too much time due to the complexity of $O(n)$.

B. Queue Configuration

When receiving a message from the TTNoC we use a two level approach. We first look up the CAN ID of the newly arrived message in the sorted *queue config* array (again with the help of a binary search) to find its fixed slot in the *queue storage*. The queue storage is an array of structs that provides space for the data field of the CAN message and a flag indicating if its CAN ID is already stored in the priority queue. After a successful lookup of a CAN ID, we check if an entry with the same CAN ID is already in the priority queue and overwrite the data fields in the queue storage. We always overwrite the data fields for two reasons. First, we consider newer data more important than old data and second, we do not have to enqueue multiple messages with the same CAN ID. This allows us to give simple guarantees for the size of the priority queue and the queue storage. If we did not already enqueue a message with the ID we looked up, we set the flag that indicates that it is in the queue and finally enqueue a new entry in the priority queue. These entries consist of the CAN ID, which is the key in the priority queue and the position of the CAN data in the queue storage. With that implementation we can keep the amount of data in the priority queue small and keep the costs for required time consuming memory moves as low as possible. Figure 5 shows an example where:

- The queue config contains entries for the CAN IDs 2, 5, 8 and 23.
- The queue storage provides space for the entries in the queue config.
- The priority queue contains entries with IDs 2 and 8.

With our implementation we find the position in the queue storage for a new message in $O(log(n))$ (i.e., a binary search), insert an element in $O(log(n))$ (i.e., heap implementation of the priority queue) and extract the element with the highest priority in $O(log(n))$. As we store the position of the data fields in the priority queue we can access the location of the data in the queue storage in $O(1)$. Without storing the location we would have to look up the position again in the queue config.

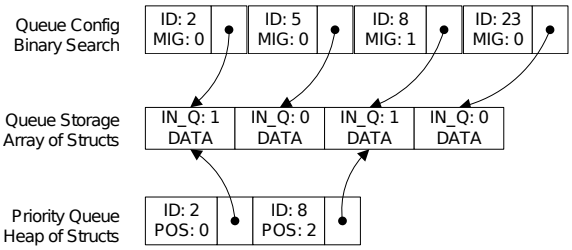


Fig. 5: Data structures for a receiving CIS

C. Switching Configurations

Switching to the new configuration requires a switch of the routing configuration and a switch of the queue configuration. Using pointers to refer to the current and the shadow configuration, a switch of the routing configuration is a simple swap of pointers. Switching the data structures for the priority queue involves the migration phase described in Section IV-C3. By utilizing the knowledge that a given index for a CAN message in the queue config corresponds to the index in the queue storage, we can implement the migration phase efficiently (see Algorithm 1 for details). We use the suffix *cur* as abbreviation for *current* and *shd* for *shadow*.

Algorithm 1 Efficient algorithm for priority queue migration

$idx_{cur} \leftarrow 0, idx_{shd} \leftarrow 0$
$PQ.empty()$
for $idx_{shd} = 0 \rightarrow QSIZE - 1$ **do**
  if $Qconfig_{shd}[idx_{shd}].migrate$ **then**
    $ID \leftarrow Qconfig_{shd}[idx_{shd}].id$
    while $ID \neq Qconfig_{cur}[idx_{cur}].id$ **do**
      $idx_{cur} \leftarrow idx_{cur} + 1$
    end while
    if $Qstorage_{cur}[idx_{cur}].in\_q$ **then**
      $Qstorage_{shd}[idx_{shd}] \leftarrow Qstorage_{cur}[idx_{cur}]$
      $PQ.add(id \leftarrow ID, pos \leftarrow idx_{shd})$
    end if
  end if
  $idx_{shd} \leftarrow idx_{shd} + 1$
end for

The algorithm is efficient for two reasons. First, it allows us to empty the priority queue in one step ($O(1)$) without losing important information. Finding specific elements in a priority queue is expensive. Processing the priority queue element by element would not be feasible because a possible re-insert would carry that element again to the front of the priority queue. Second, the queue configurations (i.e., current and shadow) are processed only once. The outer for-loop is limited to QSIZE iterations. The inner while-loop does not add extra complexity because it is guaranteed that it will be executed in total at maximum QSIZE times. Assuming that every message has to be migrated and is in the queue, this leads to an overall worst case complexity of $O(n)$. The MU has knowledge of the old and the new configuration to be deployed. The MU ensures that only configuration entries that are present in the old configuration have their migrate flag set in the new configuration.

D. Protocol

For dynamic reconfiguration we use a header payload protocol (5 bit header, 123 bit data). As we use the same message length for messages between the MU and the CISes as between CISes, the configuration has to be split up in several parts. On the one hand this increases the complexity of the protocol, on the other hand it tremendously decreases the effort for configuring the TTNoC, where different message lengths would have to be taken into account for scheduling and routing.

In the following we will discuss the meaning of the bits in the header (cf. Figure 6) and their influence on the data fields of the configuration messages.

- `Request`: This bit is set in the first message from the MU and is used by the CISes to reset their internal reconfiguration logic and to initialize the shadow buffers.
- `Commit`: This bit is set when the CIS should switch to a new configuration. This is the last message of a reconfiguration round.
- `Routing`: This bit indicates that the entry sent contains routing configuration data.
- `Queue`: If this bit is set, the message contains information for the queue configuration.
- `Next`: If this bit is set, the message is interpreted in the second format, else in the first one (cf. Figure 6). It also signals that an entry is complete and that the index in the shadow buffer has to be incremented.

Figure 6 shows the two different formats used for reconfiguration messages (cf. F1 and F2). Format one is used if the *Next* bit is unset in the header, else format two is used. Depending on whether *Routing* and/or *Queue* bits are set, the corresponding data fields are interpreted. For short, all fields contain routing configuration information with the exception of *CANIDQ* which is used for the queue configuration. Fields in reconfiguration messages have the following meaning:

- TSH_x: High bits for minimum and maximum interarrival times.
- TSL_x: Low bits for minimum and maximum interarrival times. Together with the high bits, a timestamp has a granularity of 2^{-31} seconds (about 465 ps) and a horizon of $2^{28} - 2$ seconds (about 8.5 years)[KO87].
- CANIDR: The CAN ID a routing entry belongs to (IDs are sent sorted in ascending order).
- TO: Bit array encoding the destination CIS(es) of a CAN message.
- CHK: Position in an array of function pointers used to validate the data field of a CAN message.
- TRID: A CAN ID used for name translation.
- CANIDQ: A CAN ID used for the queue configuration. IDs are sent sorted in ascending order. We use bit 30 (note that valid CAN IDs have at maximum 29 bits), to flag if messages with that CAN ID have to be migrated. In our implementation bit 30 acts as the *migrate* flag.

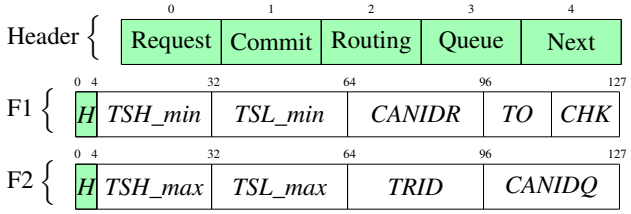


Fig. 6: Protocol used for reconfiguration

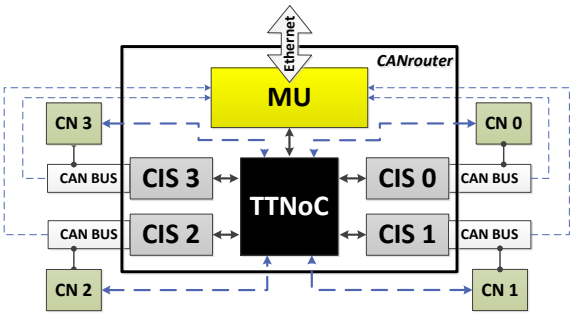


Fig. 7: Test framework overview

VI. Evaluation

In order to validate the dynamic configuration and real-time capabilities of the CAN router, we developed a test framework which executes a set of test cases.

A. Test Framework

Our test framework consists of an FPGA prototype implementation (Altera Stratix III Devkit) that hosts a four port CAN router and four CAN Nodes (CNs). Figure 7 depicts our setup. The CNs are used to generate predefined CAN traffic patterns which have to be processed by the router. CAN segments operate at a speed of 256 Kbit/s which is a performance limitation of the used FPGA devkit and not the design itself: On ASIC or more recent FPGA technology, a nominal speed of 1 Mbit/s is achievable. A notable addition compared to a usual CAN router setup is that the MU features four CAN controllers to monitor all CAN segments individually.

B. Test Cases

All test cases have a common initial phase, where all CISes start with a configuration that does not forward CAN messages (i.e., the *to* field of the routing configuration is empty, but all message properties like the minimum interarrival time are checked). All CNs with the exception of CN 0 generate CAN traffic at 35% of the maximum CAN segment bandwidth. CN 0 serves as a sink and does not produce any messages. All sending CNs generate CAN messages where the CAN ID corresponds to the number of the CN (e.g., CN 1 sends messages with CAN ID 1). The CNs are all synchronized, such that they generate their CAN traffic synchronously at the same time instants: e.g., if CN 1, 2 and 3 were on the same CAN segment, all three messages would collide and the CAN arbitration mechanism would ensure that the CAN ID 1 message would win, then the CAN ID 2 message would follow and the CAN ID 3 message would be transmitted last. Further, all CNs embed an integer sequence number in the data field of the generated CAN message. After every generated CAN message the sequence number is increased by one, such that any two consecutively sent CAN messages differ in their sequence number by exactly one. In case the router operates as expected, every CIS receives CAN messages where the embedded sequence number increases by exactly one for every CAN ID. If a CIS detects that the sequence number is not increased by one, CAN messages are lost or duplicated and the respective CIS reports this violation to the MU. The sending behavior of the CNs does not change during the experiment.

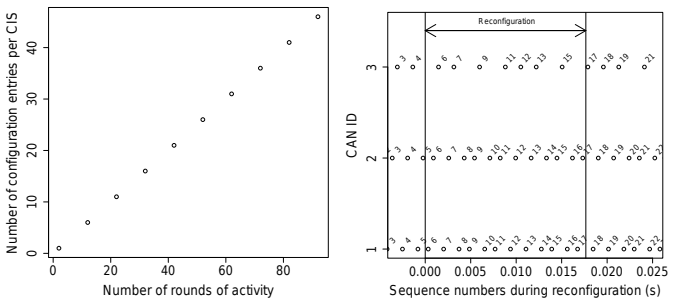
a) Test Case 1 (TC1): In this test case we validate *(1)* the duration of reconfiguration, and *(2)* if reconfiguration influences message processing (i.e., do source CISes lose CAN messages during reconfiguration).

After the described common initial phase (i.e., non-forwarding initial configuration), the MU repeatedly sends new configurations to the CISes starting with one routing configuration and one queue configuration per CIS. For every new configuration the number of entries in the configuration (routing as well as queue) is increased by five entries until the defined maximum number of configuration entries is reached. All configurations include entries to forward messages from CN 1, CN 2 and CN 3 to CIS 0. The CISes measure the time duration between the *configuration request* and the *configuration commit* and report it to the MU. Additionally, the MU logs all CAN messages sent on the destination CAN bus connected to CIS 0 w.r.t. their time of occurrence (timestamp) and sequence number.

b) Test Case 2a (TC2a): This test case validates that the router processes all messages timely during reconfiguration.

In the first phase of this test case, after the common initial phase, the MU sends a configuration where CAN messages from CN 1, CN 2 and CN 3 are forwarded to CIS 0. As all messages arrive at the same CIS within the same round of activity, the priority queue will contain entries for the CAN IDs 1, 2 and 3. In the second phase of the test case the MU deploys a new configuration where only messages from CN 2 are forwarded to CIS 0. The new configuration for CIS 0 does not require the migration of any ID 2 message which is possibly still in the queue. While the earlier described synchronization of the CNs helps in finding the worst case (i.e., an unprocessed ID 2 message in the queue), there is still the possibility that the queue does not contain an ID 2 message (e.g., in case the configuration commit occurs exactly after the ID 2 message has been processed and before a new one arrives). For our evaluation we present a case where the queue is non-empty. After the configuration commit, we expect that the MU observes only messages with ID 2 at the CAN segment connected to CIS 0. As the ID 2 message in the queue of CIS 0 is not going to be migrated, we expect that it will be lost during the configuration commit.

c) Test Case 2b (TC2b): In order to confirm the benefits of migration, we execute *Test Case 2a* again, but this time the new configuration in the second phase requires migration. Again, we only present a case where the queue of CIS 0 actu-



(a) TC1: Rounds of activity required for reconfiguration (e.g., 1 round $\equiv 2^{-13}s$ for a 256 Kbit/s CAN network)

(b) TC1: Continuity of service (at CIS0)

Fig. 8: TC1: Duration and continuity of service

ally contains an ID 2 message. During both phases we expect that the MU observes ID 2 messages at the CAN segment connected to CIS 0 and that all messages are in sequence (i.e., the sequence number in those messages increases by exactly one).

C. Results

1) Test Case 1: Figure 8a shows the number of required rounds of activity from a reconfiguration request until the new configuration is deployed on all CISes (i.e., configuration commit). The number of required rounds of activity is twice the number of configuration entries sent: every routing configuration entry has to be split in two TTNoC messages, and only a single message is transmitted per round of activity. None of the source CISes reported a message loss or duplication during reconfiguration. Figure 8b shows continuity of service during the last reconfiguration cycle with 46 configuration entries. Sequence numbers for CAN messages with the ID 1 and 2 are monitored with an increase of exactly one. The total required bandwidth to transmit all CAN messages on the CAN segment connected to CIS 0 exceeds the available bandwidth (35%+35%+35%=105%). Hence, some CAN ID 3 messages (lowest priority) originating from CN 3 will be overwritten in the priority queue of CIS 0. This intentional overload behavior is reflected by an occasional skip in sequence numbers of CAN ID 3 messages in the figures.

2) Test Case 2: Figure 9 and Figure 10 show the sequence numbers of the last messages that are observed on the destination CAN segment of CIS 0 before and after the new configuration was deployed. The significant difference between the two test case variations is that in test case TC2a message migration for messages with the CAN ID 2 was disabled, while in TC2b it was enabled. Therefore, the message with sequence number 103 was not migrated during reconfiguration in test case TC2a, while it was migrated in test case TC2b.

After the configuration commit, we observe a single apparently late CAN ID 1 and 3 message. These messages have been already transfered to the CAN controller of CIS 0 before the configuration commit (CAN controller processing delay).

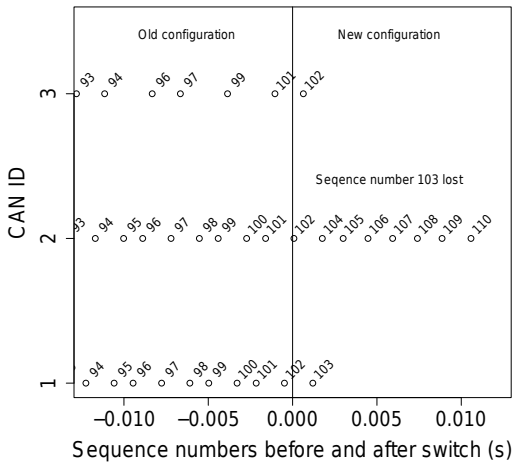


Fig. 9: TC2a, Reconfiguration without migration

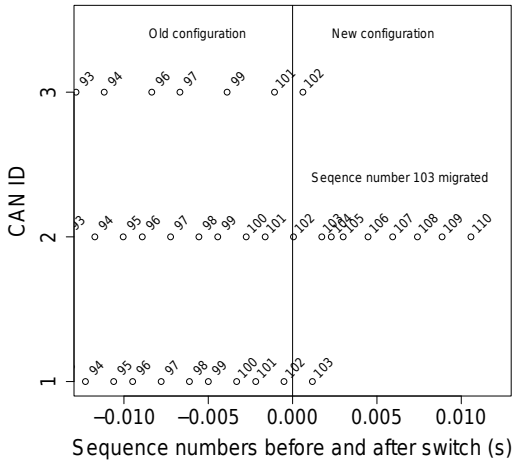


Fig. 10: TC2b, Reconfiguration with enabled migration

In test case TC2b, three CAN ID 2 messages are pushed together right after the configuration commit (sequence numbers 103, 104 and 105). This effect occurs, because before the configuration commit, CAN messages of CN 1, 2 and 3 arrived in the same round of activity (synchronized CNs) at CIS 0. This lead to a 'filled' priority queue during migration and a slight delay of the ID 2 message with the sequence number 103. In the new configuration there was no higher priority ID 1 message that needed delivery before the ID 2 message with sequence number 104. This gave room for said ID 2 message 104 and ultimately lead to a short burst of three messages.

VII. Discussion

The experimental evaluation has provided evidence for the claim that the router solves the reconfiguration challenges identified in Section I:

- *Predictable Reconfiguration Results.* The configuration of the CAN router as described in Section IV determines its behavior. Aftereffects of the new configuration are known before the configuration is actually deployed (e.g.,

the minimum and maximum interarrival times facilitate a predictable temporal behavior in the new configuration).

- *Bounded Time for Reconfiguration.* The reconfiguration requires a bounded number of rounds of activity, which results from the size of the new configuration (i.e., the number of configuration entries). The number of configuration entries has been formally described in Section IV. Required rounds of activity for reconfiguring a given CIS c are defined by $max(2*|R_c|, |M_c|)$. The factor of 2 is introduced by the need of two rounds of activity in our current implementation to send one routing configuration entry (i.e., the entry has to be split up into two messages). The overall duration of a reconfiguration is bounded by the largest of all CIS configurations, because the MU is able to send all CISes one (individual) configuration message per round of activity. Hence, all the individual CIS configurations are distributed in parallel.
  The scalability of our reconfiguration approach is therefore not determined by the number of CISes, but by the size of the largest configuration. In addition, test case 1 provides experimental evidence for the bounded reconfiguration time in a given scenario.
- *Continuity of Service during Reconfiguration.* Due to the shadow buffers, an ongoing reconfiguration does not affect the redirection of messages by the CAN router. The continuity of service during reconfiguration has also been experimentally evaluated in test case 2.
- *Consistent Switch to new Configuration.* The CAN router uses an MPSoC with a time-triggered NoC. The global time base of this MPSoC platform allows to define a consistent reconfiguration instant, at which all CISes adopt the new configuration from their shadow buffers.
- *Robust Reconfiguration Mechanisms.* The TTNoC provides inherent fault isolation based on a time-triggered communication schedule [9]. Thus, a faulty CIS cannot interfere with the reconfiguration or exchange of CAN messages at other CISes. The MU and the TTNoC represent a single-point-of-failure. It is expected that CIS failures are dominant due to the small chip area of the MU and TTNoC compared to the CISes. In addition, a safety-critical application would involve the need to replicate the complete router anyway due to the inability of achieving ultra-reliability with a single chip [10].
- *State Preservation.* The CAN router satisfies the requirement of state preservation by allowing to migrate messages between configurations (cf. Figure 10).

VIII. Conclusion

In this paper we introduced a mechanism to dynamically configure and reconfigure an earlier presented star coupling, time-triggered CAN router during run-time. We believe the router provides increased dependability compared to existing solutions. In order to achieve increased fault detection and isolation in the time as well as in the value domain this router utilizes a priori knowledge about the allowed behavior of connected CAN nodes (e.g., set of valid CAN IDs per CIS, minimum and maximum interarrival times). The increased dependability comes at the cost that the router has to provide mechanisms for reconfiguration in case the a priori knowledge has to be updated. In order to bound the time required for dynamic reconfiguration we provided a formal description of the configuration structures used by the CAN router. We then elaborated on an efficient implementation of these structures and a protocol used for reconfiguration. In the evaluation and discussion sections we showed that the router supports fundamental reconfiguration challenges including temporal predictability, state preservation and continuity of service during reconfiguration. We plan to invest further work in eliminating the router as a single point of failure (i.e. by replication) and carefully compare our solution to existing ones w.r.t. dependability and benefits.

Acknowledgments

This work has been supported in part by the European research project INDEXYS under the Grant Agreement ARTEMIS-2008-1-100021.

References

[1] M. Barranco, J. Proenza, G. Rodriguez-Navas, and L. Almeida. An active star topology for improving fault confinement in can networks. *Industrial Informatics, IEEE Transactions on*, 2(2):78 – 85, may 2006.

[2] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *IEEE Transactions on Computers*, 49(3):230–245, March 2000.

[3] U. Brinkschulte, E. Schneider, and F. Picioroaga. Dynamic real-time reconfiguration in distributed systems: timing issues and solutions. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 174 – 181, may 2005.

[4] I. Broster and A. Burns. An analysable bus-guardian for event-triggered communication. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 410 – 419, dec. 2003.

[5] L.B. Fredriksson. Can for critical embedded automotive networks. *Micro, IEEE*, 22(4):28–35, 2002.

[6] Infineon. *MultiCAN - CAN-Gateway Functionality without CPU Interaction*.

[7] Int. Standardization Organisation, ISO 11898. *Road vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication*, 1993.

[8] R. Obermaisser, R. Kammerer, and A. Kasper. Sternkoppler für Controller Area Network (CAN) auf Basis eines Multi-Processor System-on-a-Chip (MPSoC). In *Proceedings of AmE 2011 – Automotive meets Electronics*, 2011.

[9] R. Obermaisser, H. Kopetz, and C. Paukovits. A cross-domain multiprocessor system-on-a-chip for embedded real-time systems. *IEEE Transactions on Industrial Informatics*, 6(4):pp. 548–567, 2010.

[10] R. Obermaisser, H. Kraut, and C. Salloum. A transient-resilient system-on-a-chip architecture with support for on-chip and off-chip tmr. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 123 –134, may 2008.

[11] Roman Obermaisser and Roland Kammerer. A router for improved fault isolation, scalability and diagnosis in can. *INDIN 2010*, Jul. 2010.

[12] C. Paukovits and H. Kopetz. Concepts of switching in the time-triggered network-on-chip. In *Proc. of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 120–129, 2008.

[13] Christian Paukovits. *The Time-Triggered System-on-Chip Architecture*. PhD thesis, TU Vienna, 2008.

[14] Jos Rufino, Paulo Verssimo, and Guilherme Arroz. A columbus' egg idea for can media redundancy. In *FTCS'99*, pages 286–293, 1999.

[15] J. Rushby. Bus architectures for safety-critical embedded systems. In *Embedded Software*, pages 306–323. Springer, 2001.

[16] E.A. Strunk and J.C. Knight. Dependability through assured reconfiguration in embedded system software. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):172 –187, july-sept. 2006.