Universitat de les Illes Balears

Departament de Ciències Matemàtiques i Informàtica

Màster en tecnologies de la informació i les comunicacions

MEMÒRIA DE TREBALL DE FI DE MÀSTER

Design and verification by means of model checking of reCANdrv: a media redundancy management driver for the nodes of a ReCANcentrate network

Data: 23 de setembre 2011

Autor,	**Directors,**	
David Gessner	Julián Proenza Arenas	Manuel Alejandro Barranco González
(signatura)	(signatura)	(signatura)

Resum

Controller Area Network (CAN) és un bus de camp àmpliament emprat en sistemes de control distribuïts. Però a pesar del seu ús estès, hi ha una controvèrsia sobre si CAN és adequat per sistemes que requereixen un nivell de fiabilitat elevat. Això és degut a una sèrie de limitacions de fiabilitat que tenen els bussos de camp, i en particular CAN. No obstant, moltes d'aquestes limitacions es poden solucionar reemplaçant la topologia en bus per una topologia en estrella. En particular, ReCANcentrate, una topologia en estrella replicada amb mecanismes avançats de contenció d'errors i tolerància a fallades, soluciona moltes d'aquestes limitacions. ReCANcentrate està compost per dos concentradors acoblats i que proporcionen un únic domini lògic de difusió. Això permet als nodes, que estan connectats a ambdós concentradors, gestionar fàcilment, mitjançant un driver software anomenat reCANdrv, l'estrella replicada. Aquest driver abstreu els detalls de la redundància proporcionada per ReCANcentrate i permet a una aplicació CAN comunicar-se amb altres nodes a la vegada que es toleren fallades i s'oculta de l'aplicació la gestió de la redundància. Aquest document descriu el disseny de reCANdrv i de les rutines que formen reCANdrv. A més, el document especifica les propietats que obté un node ReCANcentrate gràcies a reCANdrv i verifica, creant un model que empra autòmats amb temps (timed automata) implementats amb el model checker UPPAAL, que aquestes propietats són satisfetes per un node ReCANcentrate.

Paraules clau: tolerància a fallades, gestió de redundància de medi, topologia en estrella replicada, verificació formal, model checking, UPPAAL, bussos de camp, Controller Area Network

Design and verification by means of model checking of reCANdrv: a media redundancy management driver for the nodes of a ReCANcentrate network

David Gessner

davidges@gmail.com

Abstract

Controller Area Network (CAN) is a fieldbus widely used in distributed control systems. However, despite its wide use, it is controversial that CAN can be used for systems that require a high level of reliability. This is due to a series of severe dependability limitations of fieldbuses, and in particular of CAN. Nevertheless, many of these dependability limitations can be overcome by replacing their bus topology with a star topology. In particular, ReCANcentrate, a replicated star topology for CAN with advanced error-containment and fault-tolerance mechanisms, overcomes many of these limitations. ReCANcentrate is comprised of two hubs that are coupled with each other and create a single logical broadcast domain. This allows the nodes, which are connected to both hubs, to easily manage the replicated star by means of a software driver called reCANdrv. This driver abstracts away the details of the replication provided by ReCANcentrate and allows a CAN application to communicate with other ReCANcentrate nodes, while tolerating faults and hiding the management of the available redundancy from the application. This paper describes the overall design of reCANdrv and describes the routines that comprise it. Moreover, the paper specifies the properties that a ReCANcentrate node achieves thanks to reCANdrv, and, by creating a model using timed automata implemented using the UPPAAL model checker, verifies that these properties are satisfied by a ReCANcentrate node.

Key words: fault tolerance, media redundancy management, replicated star topology, formal verification, model checking, UPPAAL, field buses, Controller Area Network

1. Introduction

The Controller Area Network (CAN) protocol is widely used in industrial systems. However, it has severe dependability limitations [3] and it is thus not adequate for safety-critical applications. Several approaches have been proposed to improve CAN's dependability, e.g., [10, 15, 22–24]. Two particular approaches are based on the use of star topologies [4].

The first approach [3] attacks CAN's limited error containment by means of a simplex star topology for CAN, called CANcentrate. It solves CAN's error containment limitations by allowing the star's central element—the hub—to isolate faulty links. However, the hub is still a single point of failure. Thus, the second approach [2], which uses a replicated star topology for CAN, called ReCANcentrate, has been proposed. It overcomes CAN's limited fault tolerance, with its several single points of failure, and

CANcentrate's sole single point of failure. This second approach provides fault tolerance through media redundancy by accommodating an additional hub. With these star topologies most of CAN's dependability limitations can be overcome. In fact, their actual effectiveness to increase the dependability of the distributed systems that use them has recently been verified [1, 6, 7].

Figure 1 shows ReCANcentrate's basic architecture. It includes two hubs with nodes connected to them through dedicated *links* comprised each of an *uplink* and a *downlink*. The nodes contain a single microcontroller and two CAN controllers, each connected to one of the node's links through a pair of CAN transceivers. The hubs are interconnected by *interlinks*, which contain two independent *sublinks*, one for each direction. Each hub has mechanisms to contain errors coming from nodes, links, interlinks, or the other hub. Moreover, ReCANcentrate as a whole has mechanisms to tolerate faults at one of the hubs, at links/interlinks, and at the nodes' CAN controllers.

The hubs couple the signals from their uplinks in an internal AND-gate, whose result is then exchanged through the interlinks. Each hub then couples the incoming interlink traffic with the aforementioned AND-gate in a second AND-gate. The result of the second coupling is then signaled by each hub on its downlinks. All this is performed within a fraction of the bit time, thereby preserving CAN's *in-bit response* and implementing CAN's *wired-AND* while providing a network-wide single broadcast domain. In other words, in the absence of faults all CAN controllers sample the same bit-value for each bit received from the hubs. This allows a media redundancy management approach for the nodes that is, as opposed to other approaches, e.g. [25], compatible with traditional event-triggered CAN applications.

The media redundancy management is implemented by a software driver, called *reCANdrv*, that executes on the nodes. This driver allows a CAN application executing on a node to communicate with applications on other nodes, while the media redundancy and tolerance of faults are handled transparently by the driver. Specifically, the purpose of reCANdrv is to allow each node's application to exchange information through the channel as long as its node has one correct controller with a correct link, while tolerating permanent faults in one of the links, the failure of one of the controllers, as well as, to some extent, inconsistent message omission (IMO) scenarios such as those that have been identified for CAN [21, 22]. Transient channel faults are not handled by the driver, but through the CAN error handling mechanisms implemented in the CAN controllers [9].

The viability of a preliminary version of reCANdrv, which does not include all the features of the current design, has already been experimentally assessed by means of a prototype [5]. Nevertheless, since ReCANcentrate is intended for safety-critical applications, it is of utmost importance to additionally ensure that the driver has been designed correctly. To ensure the correctness of the design, it is appropriate to use formal methods, which in the context of industrial systems have recently gathered increased interest [11–13, 16–18].

This paper, after introducing the final design of reCANdrv, which is a significant improvement over the previous one, formalizes as a series of properties the features that reCANdrv provides to a ReCANcentrate node; describes a model of reCANdrv that was implemented as a series of timed automata; and formally verifies, by means of model checking, that the model satisfies the properties.

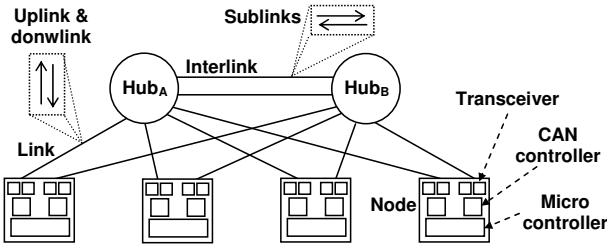


Figure 1: ReCANcentrate architecture.

2. Fault model

A node is comprised of two controllers that can diagnose their own failure. A single such *self-diagnosing CAN controller* can be built from two *individual* off-the-shelf CAN controllers and an electronic circuit that acts as a *comparator* of the individual controllers' outputs. The individual CAN controllers may suffer byzantine failures. The probability of both individual controllers generating the same erroneous output is assumed to be negligible. Thus, the comparator can detect when one of the individual controllers fails by noticing a discrepancy between their outputs. When such a discrepancy is detected, the comparator first generates an *alert interrupt*, which signals to the microcontroller that the self-diagnosing controller failed, and then triggers additional circuitry that ensures that the self-diagnosing controller can no longer generate traffic on the channel nor generate any interrupts to the microcontroller. Thus, in our fault model the self-diagnosing controllers are assumed to fail silently once they alerted of their failure. However, the microcontroller may still attempt to access buffers and registers of a failed controller. In that case arbitrary values may be read. Finally, note that in the remainder of this paper no further references are made to the individual controllers that make up a self-diagnosing controller and the term "controller" refers to a self-diagnosing controller.

Everything beyond the nodes' CAN controllers (on the transceivers' side) is part of the channel, which may fail in arbitrary ways, as long as no medium partitions occur, i.e., faults that partition the nodes into subsets that cannot communicate with each other. That is, we assume a *single broadcast domain* at all times.

We also assume that the implemented software routines are not affected by faults, e.g., that they have been implemented correctly and are not affected by memory corruptions.

Finally, we exclude from our fault model the failure of a node's microcontroller.

3. Media management

In reCANdrv's media management strategy, one of the CAN controllers of each node is the *transmission (tx) controller*. Its role is to transmit and receive frames. The other is the *non-transmission (non-tx) controller* and it exclusively receives frames—which may have been transmitted by its own node or by some other node.

When a frame is successfully exchanged through the network, i.e., when a *communication event* occurs, each node expects that its two controllers quasi-simultaneously notify of that event by means of an interrupt. Thus, when no faults occur, the node manages transmissions and receptions as follows. First, if the node successfully transmits a frame, the tx controller and the non-tx controller generate an interrupt to notify

of the transmission and reception of this frame respectively; thus, the node only needs to accept the transmission and release the reception buffer of the non-tx controller. Second, if the node receives a frame sent from another node, it is notified of this by its two CAN controllers. When this happens, the node simply consumes the frame received at one of the controllers and releases the reception buffers of both controllers.

When errors occur, the driver determines that it was not able to communicate through a link when it observes that the link's controller does not notify of a communication event while the other does, i.e., when an *omission discrepancy* occurs. To tolerate the fault, the driver accepts as valid the transmission or reception notified by one of the controllers. The notification is considered correct and the omission wrong because in CAN channel errors are converted into omissions and because the generation of spurious notifications is, thanks to the controllers being self-diagnosing, negligible.

If the controller that omits notifications is the non-tx controller, the driver does not need to diagnose it as faulty. This is so because the node can continue to correctly receive and transmit through the tx controller despite the non-tx controller omitting the notification of receptions.

If it is the tx controller that omits, the driver eventually diagnoses it as faulty by initiating a *transmission (tx) timer* when it requests a transmission. If the timer expires before the tx controller notifies of a successful transmission, the driver discards it, uses the other controller as the tx controller, and instructs a retransmission through the new tx controller. This ensures that if the old tx controller was not able to transmit, a transmission is ultimately performed through the other controller. Note that the tx timer must be set appropriately: it must not expire while the controller is correct, but only after it has suffered a permanent failure—otherwise a correct controller would be discarded. Specifically, the timer's value must be greater than the worst-case response time for a message with an error model that includes transient faults only [14].

A controller is also discarded when it generates an *error warning* interrupt, which indicates that the controller's error counters [9] reached a threshold. This prevents the controllers from entering the *error-passive state* [9], in which they could inconsistently exchange frames, leading to IMOs.

There is a further noteworthy detail. If the non-tx controller omitted a notification while the tx controller notified a transmission, and the number of consecutive omission discrepancies has not reached a threshold, the driver instructs a retransmission through the tx controller. It does this because an omission discrepancy may indicate an IMO scenario such as those that have been identified for CAN [21, 22]. Thus, the retransmissions are a best-effort attempt to prevent these IMOs. However, if the threshold is reached, no further retransmissions are performed because in that case it is more likely that the omission discrepancies are due to a link fault.

3.1. The architecture of reCANdrv

Figure 2 shows the basic architecture of reCANdrv. At the top is the application executing on the node's microcontroller, and at the bottom the two hardware CAN controllers and a hardware timer functioning as the tx timer. The driver is in the middle, providing a software layer between a node's application and its two CAN controllers, thereby abstracting from the application the existence and management of a redundant link.

Towards the application the driver provides an interface that includes primitives to allow the application to use the two CAN controllers as if there was only one, thereby implementing a *virtual CAN controller*.

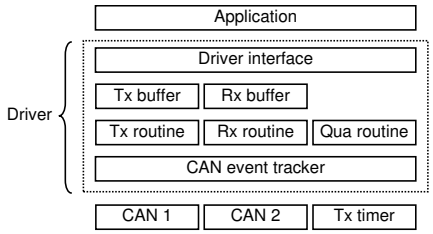


Figure 2: Basic reCANdrv architecture.

Below the interface are the driver's *transmission (tx) buffer* and *reception (rx) buffer*. When the application requests the transmission of a message, a copy of the message is stored in the driver's tx buffer in addition to being stored in the hardware transmission buffer of the tx controller. The driver uses this copy for different management operations. For instance, if the driver diagnoses the tx controller as faulty before that controller successfully transmits the message, it uses the copy to transfer the message to the other controller. Regarding the rx buffer, it is a buffer that allocates the last message received from the channel. By comparing the message in the rx buffer with the message, if any, in the tx buffer, the driver can determine whether it received a message from another node or a message that it itself requested to be transmitted. Moreover, since when the driver is notified of a reception it immediately copies the received message from one of the controllers to the driver's rx buffer, it is ensured that a copy of the received message is saved as soon as possible, which can prevent some message losses if a controller subsequently fails.

The driver has several *media management routines*, which cooperate with each other to implement the driver's main functionality. These are shown in Figure 2 and are the *transmission (tx) routine*, the *reception (rx) routine*, and the *quarantine (qua) routine*. These routines are invoked by the *media management ISRs* (not shown in the figure), which are ISRs that simply call with the appropriate parameters the corresponding media management routine. Specifically, the media management ISRs are the *ew0* and *ew1* ISR, which are invoked after an error warning interrupt (HW_INT_EW0 or HW_INT_EW1) and call the qua routine; the *rx0* and *rx1* ISR, which are invoked after a reception interrupt (HW_INT_RX0 or HW_INT_RX1) and call the rx routine; and the *tx0* and *tx1* ISR, which are invoked after a transmission interrupt (HW_INT_TX0 or HW_INT_TX1) and call the tx routine.

Although the media management routines are invoked after error warnings and CAN communication events (transmissions and receptions), none of the media management ISRs is directly triggered when such an event occurs. Instead, the media management ISRs are invoked through software interrupts generated by the *CAN event tracker* function (shown in Figure 2). Specifically, the ew0 and ew1 ISRs are invoked when the CAN event tracker function generates the SW_INT_EW0 and SW_INT_EW1 software interrupts, respectively; the tx0 and tx1 ISRs are invoked when the function generates the SW_INT_TX0 and SW_INT_TX1 software interrupts, respectively; and the rx0 and rx1 ISRs are invoked when the function generates the SW_INT_RX0 and SW_INT_RX1 software interrupts, respectively.

The CAN event tracker function is called by the *tracker ISRs*: *tracker0* and *tracker1* (not shown in Figure 2). These ISRs are the ones that are directly triggered by error warnings and CAN communication events. The tracker0 ISR is invoked when the first

Table 1: Interrupts, ISRs, and functions of reCANdrv.

Interrupt	Invoked ISR	Function called	Interrupt generated
HW_INT_FAIL0	alert0	—	—
HW_INT_FAIL1	alert1	—	—
HW_INT_EW0	tracker0	CAN event tracker	SW_INT_EW0
HW_INT_EW1	tracker1	CAN event tracker	SW_INT_EW1
HW_INT_TX0	tracker0	CAN event tracker	SW_INT_TX0
HW_INT_TX1	tracker1	CAN event tracker	SW_INT_TX1
HW_INT_TIMEOUT	timeout	qua routine	—
HW_INT_RX0	tracker0	CAN event tracker	SW_INT_RX0
HW_INT_RX1	tracker1	CAN event tracker	SW_INT_RX1
SW_INT_EW0	ew0	qua routine	—
SW_INT_EW1	ew1	qua routine	—
SW_INT_TX0	tx0	tx routine	—
SW_INT_TX1	tx1	tx routine	—
SW_INT_RX0	rx0	rx routine	—
SW_INT_RX1	rx1	rx routine	—

controller generates an interrupt indicating an error warning, transmission, or reception; whereas the tracker1 ISR is invoked when the second controller generates the equivalent interrupts.

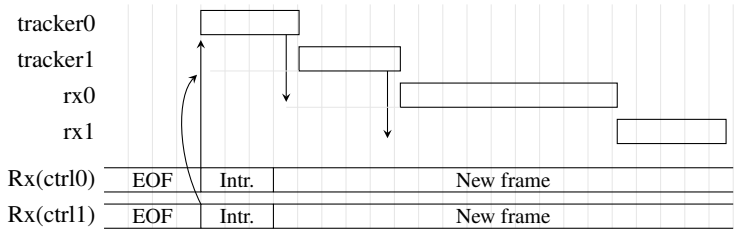
In the absence of faults, both tracker ISRs are invoked for each signaled frame—one for each controller. During a transmission, one tracker ISR will trigger the tx ISR and the other the rx ISR; during a reception, both tracker ISRs will trigger an rx ISR. In any case, two media management ISRs will be triggered and one of them executes before the other. In that case, the two executions must collaborate to handle the frame.

On the other hand, in the presence of faults, omission discrepancies can occur and, in consequence, there may only be a single tracker ISR that is invoked. This leads to only a single media management ISR being triggered, which will have to handle the communication event on its own.

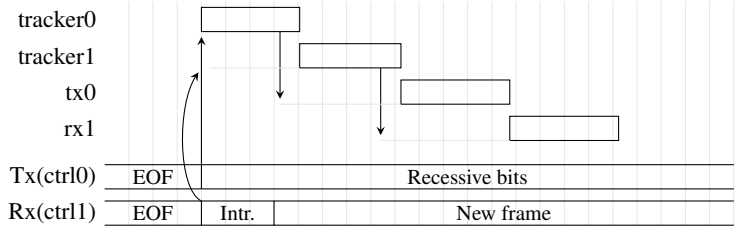
In addition to generating software interrupts to invoke the media management ISRs, and thus functioning as a dispatcher, the CAN event tracker sets a *tracking variable* to keep track of what interrupt occurred. These variables are then taken into account by the media management routines to correctly cooperate with each other.

Besides handling CAN communication events, the driver also needs to handle the expiration of the tx timer (which generates the HW_INT_TIMEOUT interrupt). For this the driver provides a *timeout* ISR, which is invoked when the tx timer expires and which also calls the qua routine. Moreover, there are two *alert ISRs* that handle the alert interrupt from the first and second controller (HW_INT_FAIL0 and HW_INT_FAIL1), respectively. These ISRs are the only ones that can interrupt other ISRs. They mark the corresponding controller as no longer trustworthy through variables that are checked by the rx routine before a message is passed on to the application. This ensures that no corrupted messages are delivered to the application.

Table 1 lists the different interrupt sources from higher priority (top) to lower (bottom). Moreover, the table summarizes the ISR invoked by each interrupt, the function called by each ISR, and, in case the CAN event tracker is called, which software interrupt is generated to invoke the appropriate media management ISR. Note that the



(a) Fault-free reception



(b) Fault-free transmission

Figure 3: Example driver executions.

priorities assigned to the interrupts are not arbitrary, but allow us to simplify the routines.

Figure 3 shows two chronograms to illustrate the interaction between the two tracker ISRs and the media management ISRs. The bottom of each of the two chronograms shows what each of the controllers receives or transmits, starting with the last bits of an end of frame (EOF), followed by the intermission [9]. The four top rows of each chronogram show when and how long each tracker and media management ISR executes. Upward arrows indicate hardware interrupts generated by the corresponding controller; downward arrows indicate software interrupts.

Figure 3a shows a fault-free reception. Both the EOF and the new frame shown at the bottom of that figure are transmitted by some remote node. Upon the reception of the last bit of the EOF, each controller generates a reception interrupt. The reception interrupt of controller ctrl0 triggers the tracker0 ISR; whereas the one from ctrl1 triggers the tracker1 ISR. Both tracker ISRs then determine that they have been invoked due to a reception. Consequently, each sets the tracking variable indicating that the corresponding controller notified of a reception and generates an interrupt by software to invoke the corresponding rx ISR. Both these rx ISRs call the rx routine, causing two invocations of the rx routine. These two invocations then collaborate to handle the reception.

Figure 3b shows a fault-free transmission. The EOF shown is transmitted by ctrl0, whereas the new frame shown is transmitted by another node. Once ctrl0 transmits the last bit of the EOF, it generates a hardware interrupt that invokes tracker0. The tracker0 ISR then sets a tracking variable that indicates that a transmission at ctrl0 occurred and invokes the tx0 ISR by means of an interrupt generated through software. Similarly, ctrl1 notifies a reception after the last bit of the EOF, which invokes tracker1. The tracker1 ISR then triggers the rx1 ISR. The tx0 ISR calls the tx routine and the rx1 ISR the rx routine. The two routines then collaborate to handle the transmission.

Note that since two rx ISRs, or an rx and a tx ISR, cooperate with each other

```
void mtx_request(message m)
{
  disable_interrupts();
  load_driver_tx_buffer(m);
  ftx_request(d.tx_controller, m);
  enable_tx_timer();
  d.has_tx_pending ← true;
  d.tx_success ← false;
  enable_interrupts();
}
```

Listing 1: The mtxreq routine.

to handle a given frame, the driver has real-time constraints. Specifically, it must be ensured that these ISRs have finished their execution before the next frame has been signaled. If not, an ISR handling one frame could incorrectly cooperate with another ISR that is already handling the next frame. The result of such an incorrect cooperation could be, for instance, that the reception of a new frame at a CAN controller of a node is incorrectly considered as having already been handled because it cooperates with an ISR that has handled the previous frame. The deadline for handling a frame is therefore the instant at which the next frame has been signaled.

3.2. The routines of reCANdrv

This section explains the main routines of reCANdrv: the *message transmission request* (mtxreq) routine and the media management routines. The mtxreq routine is part of the driver's interface. We cover it since it helps to understand the media management routines. The other primitives of the driver interface are not explained since they are either trivial, e.g., the one reading a received frame simply returns the contents of the driver's reception buffer, or they are out of the scope of this paper, e.g., the one initializing the different hardware registers of the CAN controllers.

3.2.1. The mtxreq routine

Listing 1 shows the mtxreq routine. The routine starts by temporarily disabling the handling of all interrupts in order to have exclusive access to the driver's variables and the controllers. Next, it loads the driver's tx buffer with the message that the application requested to be transmitted, which is passed as a parameter. Then it requests from the controller that is marked by the driver as the tx controller (`d.tx_controller`) the transmission of a frame that encapsulates the message. After the request, the transmission will be performed as soon as the channel becomes free. The next step of the routine is to enable the tx timer, to update the driver's status as having a transmission pending, and to reset the transmission success status to false. These two status variables (`d.has_tx_pending` and `d.tx_success`) are then updated by a tx ISR after a successful transmission, as described in Section 3.2.4. Finally, the mtxreq routine enables interrupts again.

3.2.2. The qua routine

Listing 2 shows the qua routine. The routine has two parameters of a datatype `t_ctrl_id` that is used to refer to the CAN controllers: `this_ctrl` and `other_ctrl`. It starts by checking whether the controller to be quarantined (`this_ctrl`) has already been marked as deactivated (for this it checks the driver variable `d.is_active[this_ctrl]`). If so, it simply returns. The check is necessary

because the qua routine can be invoked twice for the same controller, which could happen if an error warning and transmission timeout coincide. If the check returns false, the routine marks the controller as no longer active and requests it to be reset. This reset may not be satisfied immediately, but could take some time. For instance, if the controller was transmitting or receiving when the reset was requested, it will typically not be reset until finished (this is how, for instance, the CAN controller of a dsPIC30F microcontroller [20] behaves, which is the one used in the experimental prototype of the driver [5]).

Afterwards, the routine checks whether the quarantined controller is the tx controller. If not, it has finished since in that case the remaining, non-quarantined controller, is already marked as the tx controller and can be used for both future receptions and transmissions. Otherwise, the qua routine must reassign the tx controller role to the other controller, assuming that the other one is still marked as active. For that it continues as follows. It disables the tx timer to avoid its unnecessary expiration—in case the routine was invoked due to an error warning and not a transmission timeout. Next, it checks whether the other controller is still active. If not, it indicates (using the shared variable `d.controller_available`) that there are no controllers available anymore. Otherwise, it assigns the tx controller role to the other controller and, if necessary, instructs a retransmission through the new tx controller. To determine if it is necessary, it checks two conditions. First, whether the application has requested a transmission that has not yet been handled by a tx routine (`d.has_tx_pending`). Second, if the previous tx controller was not able to transmit before it was quarantined. This is indicated in the second condition of the if-statement, which checks that the tracking variable `d.tx_notification[this_ctrl]` was not set. This tracking variable is set by the tracker to indicate to the media management ISRs that the given controller has generated a transmission interrupt. The tracker sets it when it is invoked due to a transmission interrupt from the given controller. There is another equivalent tracking variable for the other controller. If both conditions are true, the tx routine performs the retransmission by requesting the message in the driver's transmission buffer to be transmitted in a frame by the new tx controller and by enabling the tx timer.

3.2.3. The rx routine

As shown in Listing 3, like the qua routine, the rx routine also has two parameters of type `t_ctrl_id`.

The routine begins by resetting `d.rx_notification[this_ctrl]`, which is a tracking variable that indicates that a reception interrupt occurred at `this_ctrl`. This tracking variable was set by the CAN event tracker and indicates that the rx routine has been triggered because of a reception interrupt by `this_ctrl`. Then the routine checks whether the driver variable `d.comm_event_handled` is true. If so, it means that the frame whose reception has launched the rx routine has either already been managed by the tx routine or that it has already been managed by a previous execution of the rx routine that was triggered by the other controller. In any case, if the `d.comm_event_handled` variable is true, all the rx routine must do is reset the variable and finish by emptying the reception buffer of `this_ctrl`.

If the `d.comm_event_handled` variable is false, the rx routine knows that neither the tx routine nor it itself has been called to handle the current frame. In that case the rx routine needs to handle the communication event. For this it starts by checking if it is handling a frame transmitted by the other controller of the node. It does this because even if the node was the one to transmit the frame, the other controller could omit a transmission interrupt due to a communication error, e.g., it could have crashed before

```
void qua_routine(
  t_ctrl_id this_ctrl,
  t_ctrl_id other_ctrl
)
{
  if (!d.is_active[this_ctrl]) {
    return;
  }
  d.is_active[this_ctrl] ← false;
  request_reset(this_ctrl);
  if (d.tx_controller ≠ this_ctrl) {
    return;
  }
  disable_tx_timer();
  if (!d.is_active[other_ctrl]) {
    d.controller_available ← false;
  } else {
    d.tx_controller ← other_ctrl;
    if (d.has_tx_pending and
      !d.tx_notification[this_ctrl]) {
      ftx_request(d.tx_controller, read_driver_tx_buffer());
      enable_tx_timer();
    }
  }
}
```

Listing 2: The qua routine.

```
void rx_routine(
  t_ctrl_id this_ctrl,
  t_ctrl_id other_ctrl
)
{
  d.rx_notification[this_ctrl] ← false;
  if (d.comm_event_handled) {
    d.comm_event_handled ← false;
  } else {
    load_driver_rx_buffer(read(this_ctrl));
    if (read_driver_rx_buffer() == read_driver_tx_buffer()) {
      /* Self-reception, but no tx notification
       * by other_ctrl */
      d.omission_count++;
    } else {
      if (d.is_trusted[this_ctrl]) {
        deliver(d.rx_ftuple);
        if (d.rx_notification[other_ctrl]) {
          d.comm_event_handled ← true;
        }
      }
    }
  }
  empty_receive_fbuf(this_ctrl);
}
```

Listing 3: The rx routine.

```
void tx_routine(
  t_ctrl_id this_ctrl,
  t_ctrl_id other_ctrl
)
{
  d.tx_notification[this_ctrl] ←
    false;
  disable_tx_timer();
  if (d.rx_notification[other_ctrl]) {
    /* Succesfull self-reception */
    d.comm_event_handled ← true;
    d.omission_count ← 0;
    d.tx_success ← true;
    d.has_tx_pending ← false;
  } else {
    /* Self-reception failed */

    if (d.omission_count < OMISSION_DEGREE) {
      d.omission_count++;
      ftx_request(d.tx_controller, read_driver_tx_buffer());
      enable_tx_timer();
    } else {
      d.tx_success ← true;
      d.has_tx_pending ← false;
    }
  }
}
```

Listing 4: The tx routine.

triggering its tracker ISR or it could omit the interrupt due to a CAN inconsistency scenario. Thus, in order to make sure that it is not managing a frame transmitted by the other controller, the rx routine reads into the driver's rx buffer the frame contained within the reception buffer of `this_ctrl`. Then it checks if that frame equals the one located in the driver's tx buffer.

If it does, the rx routine assumes that the other controller considered the transmission as failed and, thus, increases the omission counter (`d.omission_count`), which indicates the number of times that a controller omitted a notification. If the other controller omitted the notification of a transmission due to a transient fault, that controller will attempt a retransmission by itself, as specified by CAN [9]. In case the omission is due to a permanent fault, the tx timer will eventually expire and the qua routine will carry out the actions needed to tolerate the fault.

If `this_ctrl` did not receive the frame in the driver's tx buffer, then the controller received a frame from another node. In that case it checks whether one of the alert ISRs marked `this_ctrl` as having suffered a failure, which would be indicated by the boolean variable `d.is_trusted[this_ctrl]` being `false`. If it is not `false`, the frame, which is stored in the driver's rx buffer, has not been corrupted and can therefore be delivered to the application. In addition, if `this_ctrl` did not suffer a failure, the routine checks if the other controller has also received the frame. If it has, the rx ISR of the other controller will execute next and call the rx routine again. Therefore the current rx routine execution sets the driver variable `d.comm_event_handled` to `true` to indicate to the other rx routine execution that the reception of the frame has already been managed. Finally, the rx routine empties the reception buffer of `this_ctrl`.

3.2.4. The tx routine

Regarding the tx routine, shown in Listing 4, it has the same parameters as the other media management routines. The tx routine starts with the reset of both a track-

ing variable (`d.tx_notification[this_ctrl]`) and the tx timer. In contrast to the rx routine, the tx routine does not check whether the communication event has already been handled. This is so because a tx ISR does always execute before an also pending rx ISR since its interrupt is of higher priority than the one of the rx ISR. After resetting the tracking variable and the timer, the tx routine checks the tracking variable `d.rx_notification[other_ctrl]` to check if the other controller generated a reception interrupt.

If it has, the routine sets `d.comm_event_handled` to `true` and resets the driver's omission counter to zero. Setting `d.comm_event_handled` to true informs the rx routine, which is expected to execute next, that the tx routine has already verified that the non-tx controller received the transmitted frame. The rx routine will therefore not have to verify it again. Finally, the tx routine also indicates to the application that the frame has been successfully transmitted by setting to `true` the variable `d.tx_success` and, moreover, resets the driver variable `d.has_tx_pending`, which indicated that the frame was pending to be transmitted.

In contrast, if no reception interrupt was triggered, the tx routine checks if the omission counter is less than the maximum number of consecutive omission discrepancies that are assumed to occur due to non-permanent faults (`OMISSION_DEGREE`). If the omission counter has not reached that maximum, the tx routine attempts to prevent a potential IMO. For this it increases the omission counter and requests a retransmission of the frame through the tx controller. Otherwise, if the omission counter reached its maximum, the tx routine assumes that the omission by the non-tx controller is not due to an IMO scenario, but due to a permanent fault that prevents the non-tx controller from communicating; thus, the tx routine gives up its attempt to prevent an IMO and simply indicates to the application that the frame has been successfully transmitted and resets the driver variable `d.has_tx_pending`.

Note that the retransmissions are a best-effort attempt to avoid data inconsistencies between the nodes due to IMOs. When the tx routine detects that the non-tx controller did not notify the reception of the frame transmitted by the tx controller, it instructs a retransmission because of a possible IMO. This retransmission may cause the reception of duplicated frames at some nodes, i.e., *inconsistent message duplicates* (IMDs); however, it is known that IMDs can occur in CAN anyway and applications should take that into account, e.g., by only setting status variables upon the reception of a frame instead of toggling their values. The retransmissions are best-effort since an IMO may manifest itself in such a way that neither the driver nor the tx controller itself retransmits and a data inconsistency between the nodes still occurs. For instance, both controllers of a transmitting node may detect a communication event (the transmission or reception of a frame) while both controllers of a receiving node may detect a communication error (the abort of a frame).

4. Properties of a ReCANcentrate node

Having described the basic functioning of reCANdrv in the previous sections, we now proceed to explain the properties that reCANdrv provides to a ReCANcentrate node. For this, we start by introducing the concepts of a correct controller and a correct link.

A *correct controller* is one that is not affected by faults (transient or permanent) and is in the *error active state* [9], i.e., in a state in which it can fully participate in the communication—as opposed to the *error passive state*, in which it may not be able to signal errors to other controllers, and the *bus off state*, in which it does not

communicate at all. A *correct link* is a link that is not affected by permanent faults and whose transient faults have a frequency low enough to not trigger error warning at its controller.

The goal of reCANdrv is to correctly manage the redundancy provided by ReCANcentrate. Specifically, the goal is to allow an application using reCANdrv on a ReCANcentrate node to transmit information to other nodes and to correctly receive information originating at other nodes. Moreover, this should be done while tolerating all channel and controller faults as long as there is one correct controller with a correct link. This goal can be specified as a series of properties, which are listed at the end of this section. However, first a series of terms, notations, and assumptions need to be introduced.

A *message* is a unit of information that an application executing on a node deals with. A *frame* is a unit of information exchanged on the communication channel. It is assumed that each message can be encapsulated in a single frame, i.e., that messages are not fragmented into multiple frames. Nevertheless, multiple frames may contain the same message. This occurs in the case of retransmissions. Moreover, frames do not necessarily have to encapsulate messages because a frame may also serve control functions on the communication channel. Examples are *error frames*, *overload frames*, and *remote frames* [9]. Despite the fact that in reality these frames do not encapsulate any messages, the notation used in this paper, described shortly, can be made more consistent by reserving dedicated messages for these frames. For instance, it is assumed that an error frame always encapsulates an empty error-frame message.

The application and the driver deal with messages, not with frames; whereas the channel only carries frames, but not messages directly. The boundary between messages and frames are the CAN controllers. They encapsulate into frames messages whose transmission has been requested, and extract messages from frames received from the channel.

All messages and frames are assumed to be identified uniquely. Specifically, it is assumed that they have sequence numbers. The message sequence numbers are called *msns*, and the ones for frames *fsns*. Both are monotonically increasing natural numbers. Each time the application requests the transmission of a new message, that message gets assigned a new msn. Similarly, each time a controller signals a new frame on the channel, that frame gets a new fsn.

The following notation is used. $F_i(M_j)$ designates a frame with fsn i encapsulating a message M_j with msn j. The term *mtx-request* is used for a message transmission request performed by the application; whereas *ftx-request* is used for a frame transmission request from a CAN controller that is performed by the driver. When a frame $F_i(M_j)$ is signaled completely, i.e., without being corrupted, then it is *transmitted*. A frame is *received* by a controller when it is completely stored in that controller's reception buffer. When the message M_j encapsulated in a received frame $F_i(M_j)$ is stored in a buffer accessible by the application and the application is notified of this, the message M_j is *passed on*. When the driver notifies the application that a message has been transmitted, the driver notifies a *tx-success*. Figure 4 summarizes these terms. We also introduce the notion of *self-reception*, which occurs when the non-tx controller receives a frame transmitted by the tx controller. Finally, a *retransmission* of a frame $F_i(M_j)$ is defined as a new transmission of a frame $F_{i+1}(M_j)$ that encapsulates the same message M_j. Note that the retransmission does not need to be performed by the same controller that did the initial transmission.

The ReCANcentrate node properties are based on the following assumptions:

- the channel provides a single broadcast domain, i.e., it cannot be broken up into

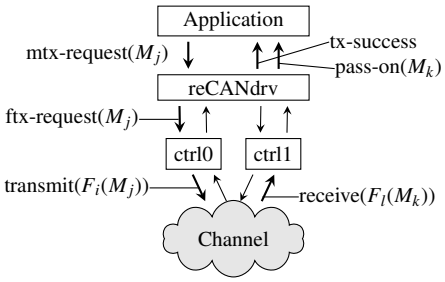


Figure 4: Terminology.

two or more independent subchannels and thus the two CAN controllers of each node cannot receive different frames simultaneously;

- there is no unicast or multicast addressing of messages, but only broadcast addressing to the applications, and thus any message from a remote node that was encapsulated in a received frame should be passed on to the application;
- the application does not mtx-request a new message until it was notified by the driver of a tx-success of the previous mtx-request;
- as soon as a controller is no longer correct, it generates an alert interrupt and immediately stops generating interrupts and traffic on the channel;
- at least one CAN controller remains correct and has a correct link at all times;
- the CAN controllers are BasicCAN [19], i.e., they only have a single transmission buffer.

Having introduced these terms, notations, and assumptions, the properties that reCANdrv provides to a ReCANcentrate node can be formalized as follows:

P1 - Pass on integrity: any message M_j passed on to the application was received in a frame $F_i(M_j)$ by at least one controller.

P2 - Double reception implies single pass on: each message M_j originated in a remote node and received in a frame $F_i(M_j)$ by both controllers is passed on to the application exactly once.

P3 - Pass on validity: each message M_j originated in a remote node and received in a frame $F_i(M_j)$ by at least one correct controller is passed on to the application, unless a single controller received $F_i(M_j)$ and that controller alerted of its failure.

P4 - No duplicate pass on: each message M_j originated in a remote node and received in a frame $F_i(M_j)$ by at least one controller is passed on to the application at most once.

P5 - No pass on of self-received messages: no message M_j is passed on to the application when it was self-received in a frame $F_i(M_j)$.

P6 - Ordered pass on: if messages M_j and M_k are passed on to the application, then M_j is passed on before M_k only if M_j was received by any one correct controller before M_k was received by any one correct controller.

P7 - Guaranteed transmission: if the application mtx-requests a message M_j, one of the controllers transmits a frame $F_i(M_j)$.

P8 - Bounded retransmissions: the controllers perform a bounded number of retransmissions of frames $F_i(M_j)$ encapsulating a message M_j.

P9 - FIFO transmission: if the application mtx-requests M_j and M_k, then M_j is transmitted before M_k only if the application mtx-requested M_j before M_k.

P10 - Bounded time to satisfy an mtx-request: if the application mtx-requests a message M_j, the driver notifies the application within a finite amount of time of a tx-success.

5. The UPPAAL model checker

To check whether reCANdrv actually provides the above properties to a ReCANcentrate node we used UPPAAL. UPPAAL [8] is a model checking tool that integrates three components: a graphical user interface (GUI) to design a model as a network of timed automata, a simulator that allows an interactive traversal of the state space, and a model checker for the automated verification of properties to be satisfied by the model. This section introduces the concepts necessary to understand how we implemented a model in UPPAAL to verify the ReCANcentrate node properties. Section 8.1 complements this section by introducing the UPPAAL query language, which is the language used to specify the properties as queries to be satisfied by the model. A more exhaustive introduction to UPPAAL can be found in the UPPAAL tutorial [8].

Timed automata are finite state machines with *clocks*. Clocks are variables to model the progress of time. They evaluate to a non-negative real number and progress synchronously. Their value can be tested or reset to a specific value. The timed automata in UPPAAL are comprised of *locations* and *edges* that connect the locations. Moreover, the timed automata are extended with discrete variables (such as integers and booleans), which can also be grouped into structures and arrays. The state of the network of timed automata is defined by the currently active location in each of the automata, and the values assigned to the clocks and variables.

The variables can be read or modified when an edge is taken. The reading and modification of variables can also be encapsulated into functions that are called when an edge is taken. The firing of an edge can depend on a *guard* and/or *synchronization*.

A guard is an expression that evaluates to a boolean value and that can use variables and clocks of the model, as well as side-effect free functions, i.e., functions that do not alter the state by, for instance, changing the value of a variable.

Synchronizations (also known as *channels*[1]) are a mechanism used to force different automata to take edges at the same time. There are different types of synchronizations. As described in the UPPAAL tutorial [8]

> *Binary synchronization* channels are declared as `chan c`. An edge labelled with `c!` synchronises with another labelled `c?`. A synchronisation pair is chosen non-deterministically if several combinations are enabled.
>
> *Broadcast channels* are declared as `broadcast chan c`. In a broadcast synchronisation one sender `c!` can synchronise with an arbitrary number of receivers `c?`. Any receiver that can synchronise in the current state must do so. If there are no receivers, then the sender can still execute the `c!` action, i.e. broadcast sending is never blocking.
>
> *Urgent synchronization* channels are declared by prefixing the channel declaration with the keyword `urgent`. Delays must not occur if a synchronisation transition on an urgent channel is enabled. Edges using urgent channels for synchronisation cannot have time constraints, i.e., no clock guards.

[1] Not to be confused with the CAN communication channel of our model, see Section 6.1.

Locations in UPPAAL can also be of different types. We will only describe the two that we used in our model: normal locations and committed locations. In *normal locations* the clocks of the model are allowed to progress and, unless there is an outgoing edge with an enabled synchronization, an automaton can stay in such a location indefinitely. In *committed locations* the clocks are not allowed to progress and the next state transition must involve at least one edge of the currently active committed locations. Normal locations are graphically represented as empty circles and committed locations as circles containing an uppercase 'C'.

Normal locations can have *invariants*. These are side-effect free expressions that compare a clock with an integer constant or variable. Valid comparison operators are less than (<), and less than or equals (<=). Invariants are used to assign an upper bound to the time that an automaton can stay at a given location.

Another feature of UPPAAL required to understand the implementation of our model is the concept of an *automata template*. This feature allows the definition of a parameterized automaton from which then particular automata can be instantiated. During the instantiation concrete values are assigned to the parameters.

6. A model of reCANdrv

This section introduces the model used to verify the properties that a ReCANcentrate node achieves thanks to reCANdrv. Although the model was implemented using the UPPAAL model checker [8], which uses timed automata, to make the model easier to understand, the model is described in a more abstract way first.

6.1. Model components

The model is basically comprised of a communication channel and a single node with two CAN controllers, an application executing on it, a tx timer, and the reCANdrv driver. We only consider a single node because the properties that the driver provides to a ReCANcentrate node are local properties of a node. Moreover, all the other nodes are represented abstractly by the channel because from a single node's perspective there is no difference between the channel itself transmitting and receiving frames, and other nodes transmitting and receiving frames through the channel. Figure 5 shows the model's components, which will be described shortly. Before that, however, it is necessary to understand how messages and frames are modeled.

Messages are modeled as non-zero positive integers, which correspond to the msns of the messages. Frames are modeled as tuples, called *f-tuples*, of two non-zero positive integers. One integer represents the encapsulated message and its value is the msn of that message. The other integer represents the frame and its value corresponds to the frame's fsn. For convenience at the time of verifying the model (see Section 8), the integers used for the msns take values from a subset S_m of consecutive non-zero positive integers, while the fsns take values from another non-overlapping subset S_f of consecutive non-zero positive integers, i.e., $S_m \cap S_f = \varnothing$.

The modeled CAN controllers generate f-tuples to model frames transmitted by the node. Similarly, the channel generates f-tuples to model frames received by the node. The fsns assigned to the f-tuples generated by the node's controllers take values of the non-empty integer set S_{f1}, whereas the fsns of f-tuples generated by the channel take values of the non-empty integer set S_{f2}, such that $S_f = S_{f1} \cup S_{f2}$ and $S_{f1} \cap S_{f2} = \varnothing$.

For f-tuples generated by one of the controllers, the msn is the one of the message encapsulated in a frame that is modeled as being transmitted. For example, if we

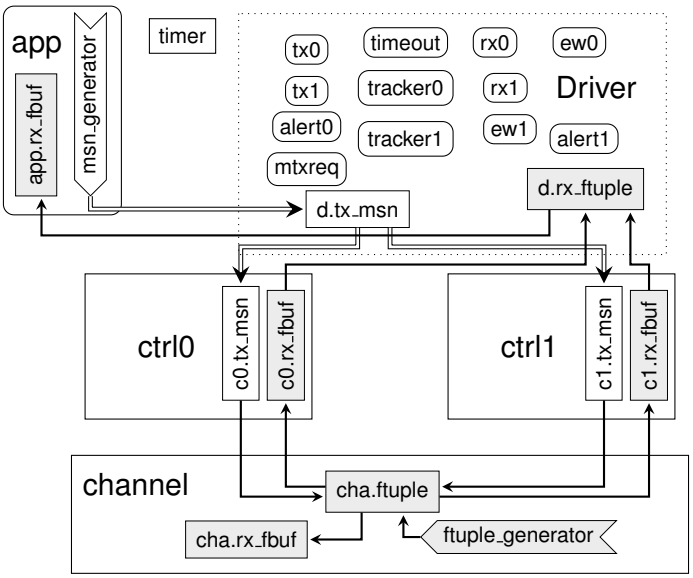


Figure 5: Model overview.

model a controller to transmit the frame $F_i(M_j)$, the f-tuple is (i, j). On the other hand, the msns used in the f-tuples created by the channel can have arbitrary values since the specific messages that are passed on to the application are irrelevant to verify the ReCANcentrate node properties. However, for convenience again, the msns used in the f-tuples generated by the channel do not overlap with the ones used in the f-tuples generated by the node's controllers.

As indicated at the beginning of this section, Figure 5 shows the main components of the model. The bottom of the figure shows the modeled communication channel, which corresponds to everything beyond the node's CAN controllers. The channel can generate new f-tuples and contains a single-slot f-tuple buffer (`cha.ftuple`) to store the f-tuple that models the currently being signaled frame and a multi-slot f-tuple buffer (`cha.rx_fbuf`) that stores all the f-tuples that have been transmitted. Everything above is part of the node.

The node is comprised of the following. A model of the node's application (labeled app in the figure), which can generate new msns and contains a multi-slot f-tuple buffer (`app.rx_fbuf`). A modelled hardware timer (shown to the right of app) that corresponds to the tx timer. A model of the reCANdrv driver (top-right of the figure), which contains models of the driver's software components (shown as boxes with rounded corners), i.e., the media management ISRs, the tracker ISRs, the alert ISRs, and the routine called during an mtx-request (`mtxreq` routine). Moreover, the model of the driver has a single-slot msn buffer (`d.tx_msn`) that models the driver's tx buffer and a single-slot f-tuple buffer (`d.rx_ftuple`) that models the driver's rx buffer. The CAN controllers are modeled by two entities (labeled ctrl0 and ctrl1), each of which contains a single-slot msn buffer (`c0.tx_msn` and `c1.tx_msn`) modeling a hardware transmission buffer, and a multi-slot f-tuple buffer (`c0.rx_fbuf` and `c1.rx_fbuf`) modeling a hardware reception buffer. By default ctrl0 is marked as the tx controller and ctrl1 as the non-tx controller. The reason for some buffers being multi-slot is to remember

what f-tuples had been inserted into them over time.

Although Section 4 claims that the boundary between messages and frames is at the CAN controllers, the model stores f-tuples, which correspond to frames, instead of msns, which correspond to messages, in the `app.rx_fbuf`. In other words, the msns received by one of the modeled controllers are left encapsulated in their f-tuples when they are passed on to the modeled application. This is convenient since it allows the model to relate each msn passed on to the application with the f-tuple in which it was received. Keeping track of this relationship is necessary in order to verify some of the properties introduced in Section 4.

Finally, there are a series of arrows between the components of the model. The thick white arrows between the msn buffers show the flow of msns representing messages; the thick black arrows between the f-tuple buffers show the flow of f-tuples; the dotted arrows incoming to the node's ISRs represent hardware interrupts generated by the timer and the controllers, or software interrupts generated by the tracker ISRs; and the dashed arrows represent notifications and requests between the different entities of the model.

6.2. Model behavior

Initially the channel is free. In the model this means that the channel's `cha.f_tuple` buffer initially contains a special empty f-tuple representing an idle channel. The signaling of a frame on the channel is modeled as the corresponding f-tuple being stored in the `cha.f_tuple` buffer. Once the signaling is finished, the channel overrides the `cha.f_tuple` buffer with a special f-tuple modeling the intermission between frames [9]. Shortly after that the channel becomes free again, that is, the channel overwrites the intermission f-tuple with the empty f-tuple.

The modeling of an mtx-request by the app begins with the app generating a new msn and invoking the model of the mtxreq routine with that msn as a parameter. The modeled mtxreq routine copies the msn to the `d.tx_msn` buffer. It then models the issuing of an ftx-request to the tx controller as follows. It copies the msn from the `d.tx_msn` buffer to the modeled tx controller's `tx_msn` buffer and sets a boolean variable to indicate to the modeled tx controller that a transmission is pending. Meanwhile, the app waits until it receives a signal that the mtx-request was completed successfully.

The subsequent transmission is modeled when the channel is free, which it indicates by overwriting the `cha.f_tuple` buffer with an empty f-tuple, as described above. When this occurs, and assuming that the channel does not initiate the signaling first by copying its own f-tuple to `cha.f_tuple`, the transmission is modeled by having the tx controller create an f-tuple from the msn in its `tx_msn` buffer and the next fsn from the integer subset S_{f1} shared with the other controller model. The modeled tx controller then writes into the `cha.f_tuple` buffer the created f-tuple and signals to the channel and the other controller that it is transmitting a message. This causes the channel to store a copy of the f-tuple in the `cha.rx_fbuf` buffer and both controllers to wait until the channel notifies that the signaling of the message is finished. This waiting models the controllers being busy receiving or transmitting the frame. Note that this means that the frames are modeled as always being transmitted fully, i.e., that they are not aborted by errors. Aborted frames are not modeled because the fact that a frame is aborted does not trigger the driver's execution. However, the errors that caused the frames to be aborted can generate error warning, alert, and timeout interrupts, all of which invoke the driver. Thus, errors need to be modeled at some level.

Specifically, the model allows errors to only occur before or after the signaling of a frame. This is justified because of two reasons. First, if the error is assumed to

have aborted a frame, it is as if the error had occurred when no frame is signaled at all. Second, if the error is assumed to not have aborted a frame, the error must have occurred during the signaling of the frame. The worst-case occurs at the two extremes: when the error occurs immediately before the signaling of the frame or if it occurs immediately after the signaling. These are the two cases that are modeled.

An f-tuple reception begins with the channel generating an f-tuple and then copying it to the `cha.f_tuple` buffer. Next, the channel indicates that it is signaling a frame, upon which the modeled controllers wait until the channel notifies that the signaling finished.

The channel's notification that the signaling of a frame finished can be of two types. It can be either a *communication event* or a *communication error* notification. For a transmitting controller a communication event means that the controller did not detect any error up to, and including, the last bit of the frame; whereas for a receiving controller it either means that the controller did not detect any error up to, and including, the last bit or, alternatively, that it detected an error at the last bit of the frame—in this latter case it would accept the frame due to CAN's last bit behavior [9]. Regarding a communication error, it means that the corresponding controller detected an error during the transmission or reception of the frame, but that, nevertheless, the error detection did not lead to the abort of the frame.

Note that a controller that detects a communication event accepts the transmission or reception of the frame; whereas a controller that suffers a communication error rejects it. This means that the communication event and error notifications allow to model omission discrepancies between the controllers of the modeled node. Modeling these omission discrepancies is important to test the driver's best-effort attempt to reduce the number of IMOs. The model does not signal a communication error to both controllers because that would represent the case where both controllers reject a frame, which does not invoke the driver.

The controllers that received a communication event notification from the channel each model the generation of an interrupt, which is then handled by the modeled tracker ISRs and media management ISRs such that they model the behavior described in Section 3.

Specifically, if the communication event notification occurs while a reception is modeled, the f-tuple is copied from the `cha.f_tuple` buffer to the `rx_fbuf` f-tuple buffer of each controller notified of the communication event. The result of the subsequent invocation of the modeled ISRs is that the received f-tuple ends up in the `d.rx_ftuple` buffer and that a message pass on is modeled next. The message pass on is modeled by having the modeled media management ISRs copy the f-tuple from the `d.rx_ftuple` buffer to the `app.rx_fbuf`.

If the notification occurs while a transmission is modeled, what happens next depends on the type of notification. If a communication event is signaled to each modeled controller, this represents a successful self-reception. In this case the modeled media management ISRs set the boolean variable `d.tx_success` to true. This represents the completion of a successful transmission and allows the modeled app to generate a new msn and to mtx-request it. If a communication error is signaled to one of the modeled controllers, a retransmission may need to be modeled. This is done by having the modeled tx controller create a new f-tuple using as the f-tuple's fsn the next integer of subset S_{f1} and using as the f-tuple's msn the one stored in the controller's `tx_msn` buffer. The created f-tuple is then written into the `cha.f_tuple` buffer and the modeled transmitting controller indicates to the channel and the other controller that it is transmitting.

7. Model implementation

This section gives an overview of how the abstract model described in the previous section has actually been implemented using timed automata. Note that the fact that the driver has real-time constraints means that it is important to not only model the system from a functional point of view, but also to model the timing of the system. This makes our choice to implement the model using a model checker based on timed automata, namely UPPAAL, especially appropriate.

7.1. The timed automata of the model

The communication channel is modeled by a single automaton. Specifically, the communication channel is modeled by the `channel` automaton, which is an instantiation of the `channelP` automata template (shown in Figure 6).

The node, on the other hand, is not modeled by a single automaton. Instead, it is modeled by several automata, each of which models a hardware or a software component of the node. Specifically, the automata are the following: the `app` automaton, which is an instantiation of the `applicationP` automata template (shown in Figure 7) and models the application running on the node; the `ctrl0` and `ctrl1` automata, which are instantiations of the `ctrlP` automata template (shown in Figure 8) and model the two CAN controllers of the representative node; the `tracker0` and `tracker1` automata, which are instantiations of the `trackerP0` and `trackerP1` automata templates respectively (shown in Figure 9 and 10) and model the two CAN event tracker ISRs; the `rx0` and `rx1` automata, which are instantiations of the `rx_ISR` automata template (shown in Figure 11) and model the two rx ISRs; the `tx0` and `tx1` automata, which are instantiations of the `tx_ISR` automata template (shown in Figure 12) and model the two tx ISRs; the `ew0` and `ew1` automata, which are instantiations of the `ew_ISR` automata template (shown in Figure 13) and model the two ew ISRs; the `timeout` automata, which is an instantiation of the `timeout_ISR` automata template (shown in Figure 14) and models the timeout ISR; the `alert0` and `alert1` automata, which are instantiations of the `alert_ISR` automata template (shown in Figure 15) and model the two alert ISRs; and the `timer` automaton, which is an instantiation of the transmission `timerP` automata template (shown in Figure 16) and models the hardware timer that is used as the tx timer. Note that there is no automata that corresponds to the mtxreq routine. Instead, that routine is directly called in an edge of the `app` automaton. In the following sections these automata will be described in more detail.

Apart from the automata mentioned above, there are also three helper automata that do not correspond to hardware or software entities. These are the `init_sys` automaton, which is an instantiation of the `init_systemP` automata template (shown in Figure 17) and is used to initialize the system; the `isr_scheduler` automaton, which is an instantiation of the `isr_schedulerP` automata template (shown in Figure 18) and is used to schedule the execution of the automata that model the node's ISRs (see Section 7.1.1); and the observer automata template `observerP` that has a single edge from its initial location to a location `finished`. The observer automaton transitions to `finished` once the model has finished modeling the transmission and reception of all frames, and the execution of the driver. An instantiation of it is used during the verification of the model, specifically, by the queries of Section 8.3.

Some of the automata templates have parameters. These are shown at the top of their respective figures. Specifically, the automata templates modeling media management ISRs (figures 11, 12, and 13) have as a first parameter the controller that originated

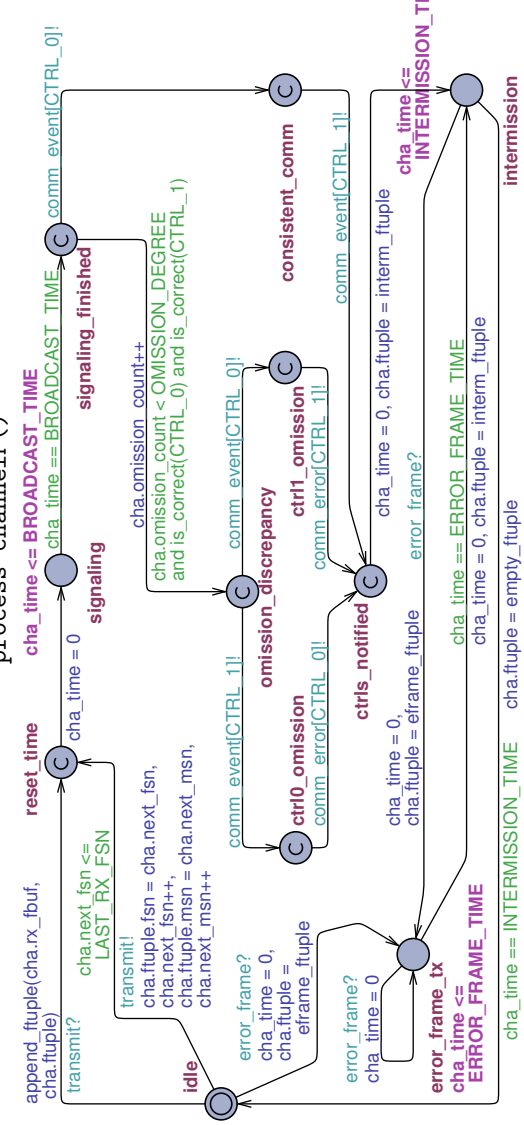


Figure 6: channel1P automata template.

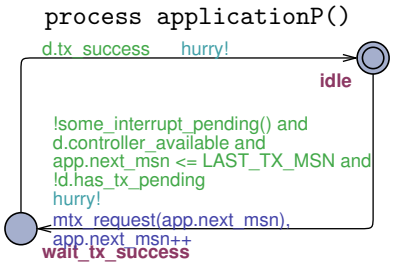


Figure 7: `applicationP` automata template.

the hardware interrupt that was dispatched to them by the tracker. As a second parameter they have the other controller. Finally, as the third parameter they have the software interrupt that they are handling and that was used by the tracker ISRs to dispatch the original hardware interrupt to them.

The `ctrlP` automata template (Figure 8) also has parameters. These are the following. First, the controller that a given instantiation of the template models. Second, the other controller of the modeled node. Finally, the four hardware interrupts that the modeled controller can generate: the error warning interrupt, the frame reception interrupt, the frame transmission interrupt, and the alert interrupt.

Listing 5 shows how the automata of our model are instantiated from the automata templates. The instantiations have the form `automaton` ← `template()`, where `automaton` specifies the name given to the automaton instantiation, '←' is the assignment operator, and `template` is the template used to create the automaton instantiation, which may have parameters.

Regarding the parameters passed to those automata templates that require them, they are globally defined constants. Specifically, the following constants are used: `CTRL_0` and `CTRL_1`, which are used to access data structures that store status variables and buffers corresponding to the first and second controller respectively; and a series of constants whose name matches the name of the different interrupts (see Table 1) and are used to model interrupts, as described in the next section.

Finally, the `system` keyword in Listing 5 is used to define that all of the instantiated automata should comprise the network of timed automata used for the model.

7.1.1. Modeling interrupts and ISRs in UPPAAL

A typical microprocessor implements interrupts using an interrupt vector table (IVT). In our model the IVT is an array data structure where each entry indicates whether a given interrupt is pending or not. The order of the entries reflects the priority of the interrupts: an interrupt with a lower index in the IVT has a higher priority than one with a higher index. When no ISR is executing, the next interrupt to be handled is always the one that is pending and has the highest priority. When an ISR is executing, the highest priority pending interrupt will be handled as soon as that ISR finishes. Except for one exception, namely the rx ISRs (as explained at the end of this section), there is no preemption between ISRs in our model.

We defined the following functions on the IVT: a function `is_next_interrupt` that returns true if a given interrupt is the next interrupt to be handled; a function `set_interrupt` that marks a given interrupt as pending; a function `clear_interrupt` that marks a given interrupt as no longer pending; and a function `some_interrupt_pending` that returns true if at least one interrupt is pending—this

process ctrlP(t_controller_id this_ctrl, t_controller_id other_ctrl, t_interrupt this_EW_interrupt, t_interrupt this_rx_interrupt, t_interrupt this_tx_interrupt, t_interrupt this_alert_interrupt)

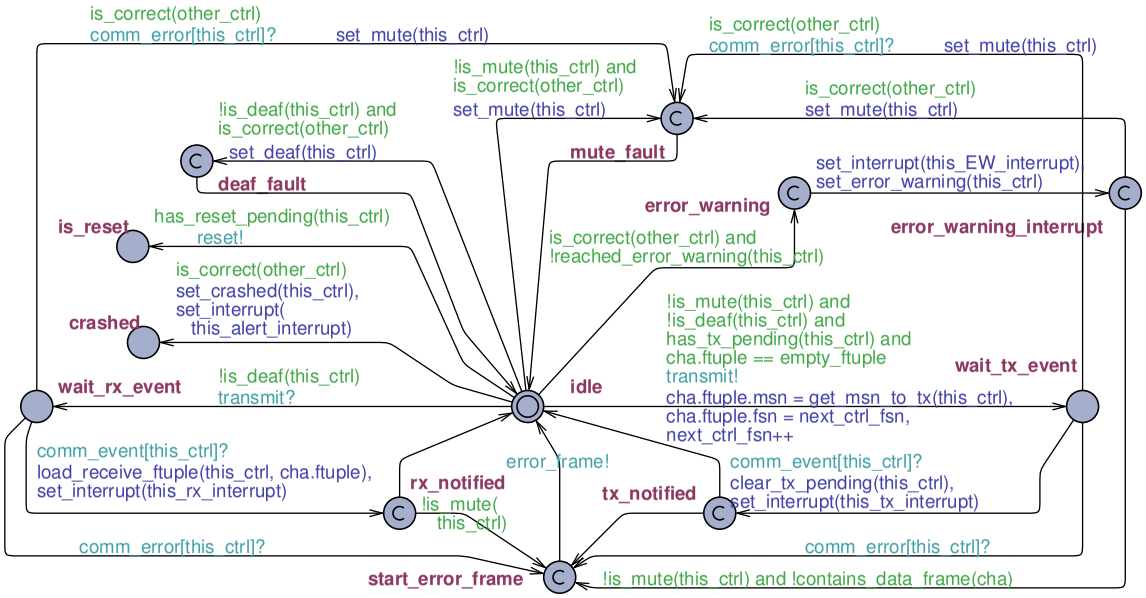


Figure 8: ctrlP automata template.

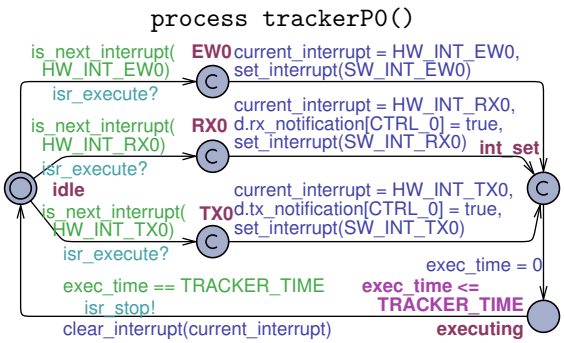


Figure 9: `trackerP0` automata template.

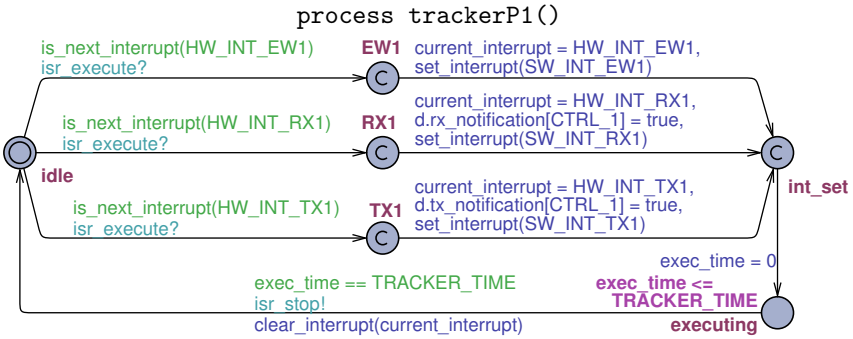


Figure 10: `trackerP1` automata template.

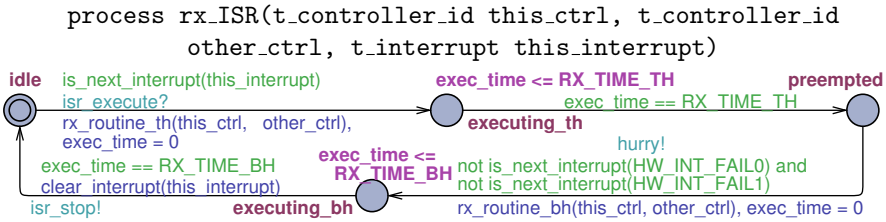


Figure 11: `rx_ISR` automata template.

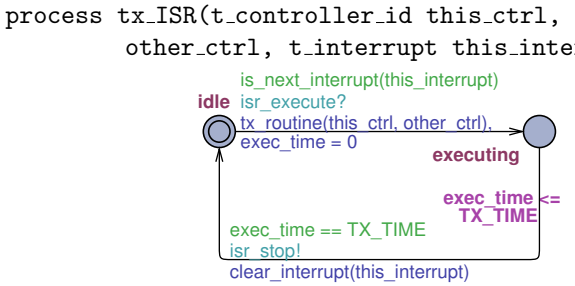


Figure 12: `tx_ISR` automata template.

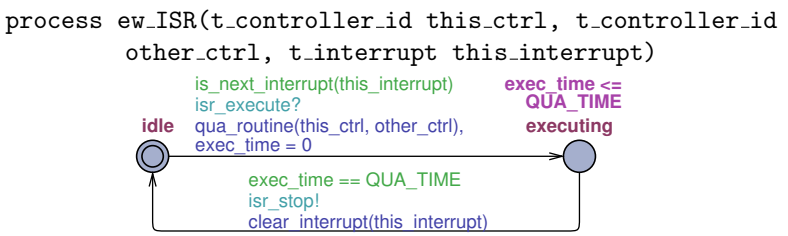


Figure 13: `ew_ISR` automata template.

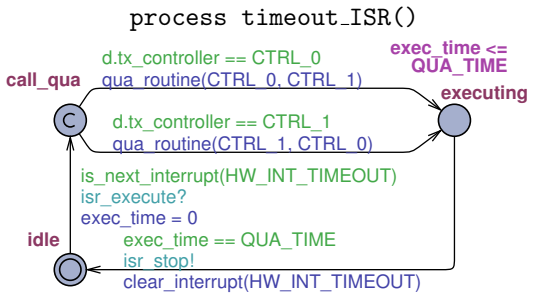


Figure 14: `timeout_ISR` automata template.

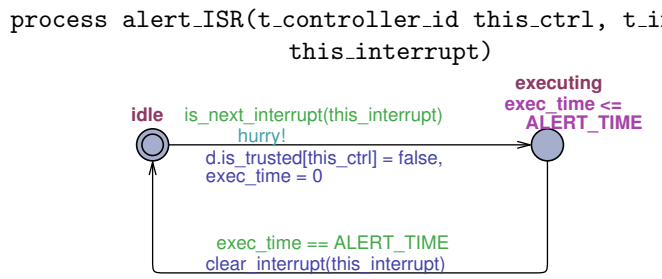


Figure 15: `alert_ISR` automata template.

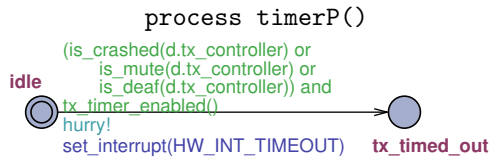


Figure 16: `timerP` automata template.

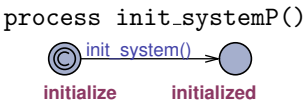


Figure 17: `init_systemP` automata template.

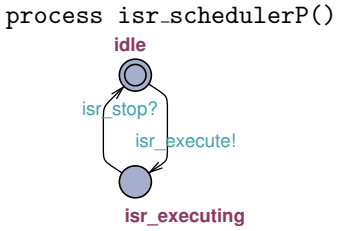


Figure 18: `isr_schedulerP` automata template.

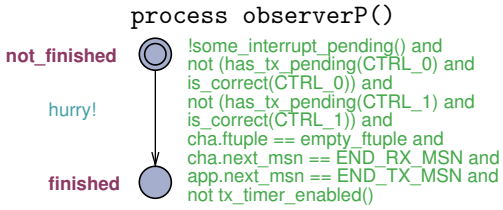


Figure 19: `observerP` automata template.

```
application ← applicationP();
tx0 ← tx_ISR(CTRL_0, CTRL_1, SW_INT_TX0);
tx1 ← tx_ISR(CTRL_1, CTRL_0, SW_INT_TX1);
rx0 ← rx_ISR(CTRL_0, CTRL_1, SW_INT_RX0);
rx1 ← rx_ISR(CTRL_1, CTRL_0, SW_INT_RX1);
ew0 ← ew_ISR(CTRL_0, CTRL_1, SW_INT_EW0);
ew1 ← ew_ISR(CTRL_1, CTRL_0, SW_INT_EW1);
alert0 ← alert_ISR(CTRL_0, HW_INT_FAIL0);
alert1 ← alert_ISR(CTRL_1, HW_INT_FAIL1);
timeout ← timeout_ISR();
tracker0 ← trackerP0();
tracker1 ← trackerP1();
ctrl_0 ← ctrlP(CTRL_0, CTRL_1, HW_INT_EW0,
HW_INT_RX0, HW_INT_TX0, HW_INT_FAIL0);
ctrl_1 ← ctrlP(CTRL_1, CTRL_0, HW_INT_EW1,
HW_INT_RX1, HW_INT_TX1, HW_INT_FAIL1);
timer ← timerP();
isr_scheduler ← isr_schedulerP();
channel ← channelP();
init_sys ← init_systemP();
obs ← observerP();

system channel, application, tracker0,
tracker1, ctrl_0, ctrl_1, tx0, tx1,
rx0, rx1, ew0, ew1, timer,
isr_scheduler, init_sys, timeout,
obs, alert0, alert1;
```

Listing 5: Automata instantiations.

last function is used by the app automaton to ensure that it does not perform an mtx-request while an ISR is executing, thereby modeling the fact that the application is preempted by the ISRs.

In our model interrupts can be generated by one of the two ctrlP automata, by the timer automaton, or by the two trackerP automata. The interrupts generated by the ctrlP automata correspond to the hardware interrupts of a CAN controller: the successful transmission of a frame (HW_INT_TX0 and HW_INT_TX1), the successful reception of a frame (HW_INT_RX0 and HW_INT_RX1), error warning (HW_INT_EW0 and HW_INT_EW1), or a controller failure (HW_INT_FAIL0 and HW_INT_FAIL1). The timer automaton generates the transmission timeout interrupt (HW_INT_TIMEOUT). Finally, the two trackerP automata each generates a different software interrupt to invoke a transition in each of the media management ISR automata (SW_INT_TX0 or SW_INT_TX1 to invoke a transition at the corresponding tx automata, SW_INT_RX0 or SW_INT_RX1 to invoke a transition at the corresponding rx automata, and SW_INT_EW0 and SW_INT_EW1 to invoke a transition at the corresponding ew automata).

The order in which the priorities have been assigned to the interrupts is the one from Table 1.

To model the occurrence of an interrupt, an edge of an interrupt generating automaton (ctrl0, ctrl1, timer, tracker0, or tracker1) invokes the set_interrupt function, passing to it as a parameter the interrupt that we want to be modeled by the edge. This marks the given interrupt as pending in the IVT. Pending interrupts are then handled by the ISR automata, i.e., the trackerP automata, the alert_ISR automata, and the media management ISR automata.

The invocation of an ISR is modeled as follows. Each ISR automaton has an idle and an executing location. The edges that lead from the idle to the executing locations are all protected by guards that are simple calls to the is_next_interrupt function with a specific interrupt as a parameter. Note that there can be more than one edge from the idle location if the given automaton models an ISR that handles more than one interrupt; specifically, the two trackerP automata templates have three edges outgoing from their idle location because each modeled tracker handles three different CAN interrupts from the corresponding controller.

There are no two is_next_interrupt guards that have the same interrupt as a parameter. This means that, at any given instant, only one single edge exiting any of the idle locations of all the ISR automata is enabled—namely, the one corresponding to the highest priority pending interrupt. However, just having the edge enabled does not cause the ISR automaton to enter the corresponding executing location. This is so because the model checker could decide to keep the automaton in the idle location indefinitely. So, to force an ISR automaton to enter the executing location as soon as the guard becomes true, we need to use an urgent synchronization (isr_execute). This urgent synchronization is signaled by the isr_scheduler automaton (Figure 18). Once it has been signaled, the isr_scheduler automaton remains in the isr_executing location until the ISR automaton that received the urgent synchronization leaves its executing location. When this occurs, the ISR automaton synchronizes with the isr_scheduler automaton using the isr_stop synchronization. This returns the isr_scheduler automaton to its idle location, from which another isr_execute synchronization can be signaled. In this manner, whenever an interrupt is pending and no ISR is executing, the ISR automaton that is modeled to handle the currently highest priority pending interrupt enters its executing location.

The time spent at each executing location corresponds to the worst-case execution time of the modeled ISR. To ensure that the appropriate time passes at each executing

location a combination of a local clock variable (exec_time), an invariant, and a guard that involves the clock variable and a constant is used.

Regarding the function that the ISRs implement, each ISR automata models that function when it takes its corresponding edge to the executing location. Specifically, the trackerP automaton models the tracker's function by setting the appropriate software interrupts and, if necessary, by setting the appropriate tracking variables; the alert0 and alert1 ISRs simply set the booleans d.is_trusted[CTRL_0] and d.is_trusted[CTRL_1] to false to mark the corresponding controller as no longer trustworthy; and the automata modeling the media management ISRs call the appropriate routine when transitioning to the executing location: the tx_ISR automata call the tx routine, the rx_ISR automata call the rx routine, and the timeout and ew_ISR automata call the qua routine. Finally, all ISR automata clear their respective interrupt before returning to their idle location.

Note that, as can be seen in Figure 11, there are two executing locations (executing_th and executing_bh), two constants (RX_TIME_TH and TX_TIME_BH), and two function calls (rx_routine_th and rx_routine_bh) for the rx ISR automata template. This is so because in the model the rx routine has been split into two parts: a top half and a bottom half. The top half corresponds to the rx routine's instructions that appear before the deliver() function call. The bottom half corresponds to that function call and the instructions that follow it in the rx routine. The reason for splitting the rx ISR in two is to model the preemption of an rx ISR by an alert ISR.

7.1.2. Transmission timeouts

To model the expiration of the tx timer we assume that the timer's timeout has been set appropriately, i.e., that it only expires if the tx controller crashes or suffers a permanent fault—which can be a *mute fault*, i.e., a fault that prevents the controller from transmitting, or a *deaf fault*, i.e., a fault that prevents the controller from receiving. This is then the condition we use to protect the transition to the tx_timed_out location in the timerP automata template (see Figure 16). In that transition the automaton models the generation of a timeout interrupt. This interrupt is then handled by the timeout ISR automaton. Note that after a timeout has once occurred, no further timeouts occur. This is so because of the modeling hypothesis that at most one controller of the node will fail. Thus, an edge returning from the tx_timed_out location to the idle location is unnecessary.

7.1.3. Transmission and reception of frames through the channel

The transmission of a frame by a transmitter, which can be either the channel or one of the two controllers, is modeled as follows. First, a new f-tuple is assigned to the cha.f_tuple buffer. This happens in the channel automaton in the bottom edge from the idle to the reset_time location (see Figure 6) and in the ctrlP automata in the edge from the idle to the wait_tx_event location (see Figure 8). At the same time, the automaton modeling the transmitter signals a synchronization using the transmit synchronization. This synchronization forces any ctrlP automaton in the idle location to transition from that location to the wait_rx_event location—unless that automaton had previously transitioned to the deaf_fault location, which models a controller becoming deaf, i.e., unable to receive. Moreover, the transmit synchronization causes the channel automaton, if it was in the idle location, to take the edge to the reset_time location, from which it proceeds to the signaling location, thereby resetting to zero the cha_time clock variable. The channel automaton then remains in

the signaling location for 44 units of time (BROADCAST_TIME = 44) due to the combination of the clock invariant at that location and the guard protecting the edge exiting that location. Meanwhile, the ctrlP automata remain in the wait_rx_event location (if the automaton models a currently receiving controller) or the wait_tx_event location (if the automaton models a currently transmitting controller) until the 44 units of time have passed and they exit their current location due to a synchronization from the channel automaton. The automata waiting in their respective locations models the controllers, and the channel being busy with the signaling for 44 units of time.

These 44 units of time are to be interpreted as 44 bit times, which is the time to signal the shortest possible frame (a CAN 2.0A remote frame or data frame with zero bytes of data, both without any bit stuffing [9]). We chose this value because of the driver's real-time constraints: as described in Section 3.1, the deadline for handling a frame transmission or reception is the instant at which the next frame has been signaled. The worst-case occurs when the time to the next finished signaling is the shortest and the ISRs that must cooperate take the longest amount of time to finish. To ensure that our driver works correctly under these conditions, we model the time to signal any frame to always be the shortest possible (44 bits). Specific values can then be assigned to the constants used in the invariants of the executing locations to check if these values satisfy the real-time constraints. If they do not, at least some of the queries (see Section 8.3) will not be satisfied.

After this amount of time has passed, the signaling of the message is finished and the channel automaton takes the transition to the signaling_finished location.

From the signaling_finished location either an omission discrepancy or a consistent communication is modeled. As described earlier, an omission discrepancy occurs when one of the modeled node's controllers accepts a frame while the other rejects it. On the other hand, we say that a consistent communication occurs when both controllers accept the signaling of the frame. Both omission discrepancies and consistent communications are modeled using two types of synchronization generated by the channel automaton and received by the ctrlP automata. These two types are communication event (comm_event) synchronizations and communication error (comm_error) synchronizations.

A comm_event synchronization causes the corresponding ctrlP automaton to take an edge that models the invocation of a reception or transmission interrupt (see the incoming edges to the rx_notified and tx_notified locations in Figure 8). Moreover, the transition models a receiving controller loading its reception buffer or a transmitting controller clearing its transmission pending flag. After the communication event, the modeled controller may detect an error (transition to start_error_frame) or not (transition to idle).

A comm_error synchronization, on the other hand, causes the corresponding ctrlP automaton to take a transition that models the controller having detected an error in the frame, which, however, did not lead the frame to be aborted. There are two ways this can occur: the controller was not able to abort the frame because it became mute (location mute_fault) or the controller signaled an error that was mistaken for an overload signaling because of additional errors (the transition from wait_rx_event to start_error_frame).

To avoid a sequence of infinite frames with communication errors we limit the number of consecutive communication errors to less than OMISSION_DEGREE. This way the number of consecutive omission discrepancies is lower than the number of consecutive retransmissions the driver executes due to receive notification omissions (which is less than or equals OMISSION_DEGREE, see Listing 4). This is justified if we assume that

the driver parameter OMISSION_DEGREE has been set appropriately, so that it is always greater than the number of consecutive IMO scenarios that can occur. If it was not set appropriately, the number of consecutive omission discrepancies due to IMOs could be greater than the retransmissions and the driver's retransmissions may not avoid IMOs between the nodes.

An overload/error signaling is initiated by means of the error_frame synchronization. This synchronization forces the channel automaton to transition to its error_frame_tx location. The channel automaton then remains at that location for at least 14 bit times (ERROR_FRAME_TIME = 14), which is the minimum length of an error frame (6 error flag bits + 8 error delimiter bits) [9]. The channel automaton may remain longer at that location if an additional error_frame synchronization is signaled by one of the ctrlP automata. After the error frame, the channel automaton returns to its intermission location for the 3 intermission bits that follow any error frame [9]. If no additional error occurs, the channel automaton models the channel becoming free again after the 3 intermission bits by returning to its idle location. If an additional error frame is initiated during the intermission, the channel automaton takes another pass through the error_frame_tx location.

The modeling of permanent faults is accomplished by the ctrlP automata. The crash of a controller is modeled by a transition to the crashed location, which does not have any outgoing edges and thus models the crash as being permanent. Regarding link faults, we distinguish two different types, each modeled differently. A fault in a link may manifest itself in such a way that it does not allow the controller to transmit anything, but it still allows it to receive frames from others. Such a fault is modeled by a transition to the mute_fault location. A fault may also manifest itself in a way that it prevents the controller from receiving frames. If such a fault occurs during an idle channel, intermission, end of frame, or ACK delimiter [9], the controller may not detect any fault as long as it does not attempt to transmit a frame itself, but simply see the communication channel as idle even when some other controller is transmitting. This is modeled by a transition to the deaf_fault location. Once this transition is taken, and the ctrlP automaton is back at the idle location, that automaton can no longer take transitions that model the reception and transmission of frames. However, the automaton can still take a transition modeling a crash, a reset, or an error warning. Specifically, the transition to the error_warning location after a transition through the deaf_fault location models, among other things, the attempt of the controller to transmit a frame after it has become deaf. Note that a deaf controller can still transmit error flags.

All edges to a location modeling a permanent fault are protected by a guard that checks that the other ctrlP automaton has not already modeled a permanent fault at the other controller. This is so because of the modeling hypothesis that at all times at least one of the two controllers will be correct and has a correct link.

When a given ctrlP automaton transitions to the error_warning location, that automaton generates an error warning interrupt [set_interrupt(this_EW_interrupt)] and marks an error warning as having occurred at the corresponding controller [set_error_warning(this_ctrl)]. This marking, once set, is never cleared and in combination with an appropriate guard [!reached_error_warning(this_ctrl)] ensures that a given ctrlP automaton can take the transition to the error_warning location only once. Note that we cannot use the error warning interrupt for this because that interrupt is cleared by the tracker automaton. After the error warning interrupt has been generated, the controller can initiate the transmission of an error frame (transition from the

`error_warning_interrupt` location to the `start_error_frame` location) as long as the controller had not suffered a mute fault and, since we do not model aborted frames, as long as no one is currently transmitting a data (or remote) frame. It is also possible for the controller to suffer a mute fault immediately after the error warning interrupt. In that case, also no error frame is initiated.

8. Model verification

The purpose of the model is to verify that a ReCANcentrate node using the reCANdrv driver satisfies the properties of Section 4. Since the model was implemented using UPPAAL, we used the UPPAAL query language, described next, to perform this verification.

8.1. The UPPAAL query language

Using UPPAAL's query language [8], a user can define a series of properties, called queries, to be tested by the UPPAAL model checker. During a verification, the model checker automatically generates all the execution paths of the model that are required in order to verify each property. In case the specific property has to hold in all the execution paths, the model checker generates all of them.

The following classes of properties that can be expressed in the UPPAAL query language are relevant to the queries that verify the ReCANcentrate node properties.

Reachability properties: these test if a specific condition (a boolean expression over locations, variables, and clocks [8] of the model's timed automata) holds in some state of the potential behaviors of the model. These properties are expressed as $E \diamond\ p$, which tests if there exists an execution path in which the condition p eventually (in some state of the path) holds.

Safety properties: these test if a specific condition holds in all the states of an execution path. They can be expressed in two forms. First, $E\Box\ p$, which tests if there exists an execution path in which p holds for all the states in the path. Second, $A\Box\ p$, which tests if for every execution path, p holds for all the states in the path.

Liveness properties: these test if a specific condition is guaranteed to hold eventually. They can also be expressed in two forms. First, $A \diamond\ p$, which tests if for every execution path, p holds for at least one of the states in the path. Second, $q \rightsquigarrow p$, which tests if every execution path that starts from a state satisfying q reaches later on a state in which p holds.

The UPPAAL query language also provides a *forall* quantifier, expressed as

$$\forall(v : t[f, l])\ b$$

The quantifier returns *true* if for all values v of type t in the range $[f, l]$, both inclusive, the boolean expression b is true.

Finally, another feature of the UPPAAL query language used in this paper is the notation $a.l$, where a indicates one of the automata of the UPPAAL model used during the verification and l indicates a location of that automaton. This notation indicates an expression that is *true* when the automaton a is in the location l.

8.2. Query helper functions and helper automata

The queries used to verify the properties use the following helper functions:

- *fsns_in_range(b, f, l)*, returns *true* if all f-tuples of buffer b have an fsn in the integer range $[f, l]$, both inclusive.
- *msns_in_range(b, f, l)*, returns *true* if all f-tuples of buffer b have an msn in the integer range $[f, l]$, both inclusive.
- *fsns_in_range2(b, f1, l1, f2, l2)*, returns *true* if each f-tuple of buffer b has an fsn in the integer range $[f1, l1]$, both inclusive, or in the integer range $[f2, l2]$, both inclusive.
- *msns_in_range2(b, f1, l1, f2, l2)*, returns *true* if each f-tuple of buffer b has an msn in the integer range $[f1, l1]$, both inclusive, or in the integer range $[f2, l2]$.
- *has_fsn(b, f)*, returns *true* if there is an f-tuple with fsn f in the f-tuple buffer b.
- *has_msn_fbuf(b, m)*, returns *true* if there is an f-tuple with msn m in the f-tuple buffer b.
- *count_fsn(b, f)*, returns the number of times that an f-tuple with fsn f is stored in the f-tuple buffer b.
- *is_sorted_by_fsn_fbuf(b)*, returns *true* if the f-tuples of f-tuple buffer b are sorted by increasing fsns.
- *is_sorted_by_msn_fbuf(b)*, returns *true* if the f-tuples of f-tuple buffer b are sorted by increasing msns.

The queries also make use of an instantiation `obs` of the observer automaton `observerP` and they use the boolean array `d.is_trusted`, which indicates for each controller whether it has alerted of its failure or not.

8.3. Queries

To verify the ReCANcentrate node properties we used a series of preliminary queries to check that the elements of the different buffers of the model can only have values within a specific range. As an example, for the `cha.rxbuf` buffer and the `app.rxbuf` buffer the queries are the following:

```
A□ fsns_in_range(cha.rx_fbuf,
FIRST_TX_FSN, LAST_TX_FSN)
```

```
A□ msns_in_range(cha.rx_fbuf,
FIRST_TX_MSN, LAST_TX_MSN)
```

```
A□ fsns_in_range2(ctrls[CTRL_1].rx_fbuf,
FIRST_RX_FSN, LAST_RX_FSN,
FIRST_TX_FSN, LAST_TX_FSN)
```

```
A□ msns_in_range2(ctrls[CTRL_1].rx_fbuf,
FIRST_RX_MSN, LAST_RX_MSN,
FIRST_TX_MSN, LAST_TX_MSN)
```

Where FIRST_TX_FSN and LAST_TX_FSN are the fsn of the first and last f-tuple generated by the controllers, respectively; FIRST_TX_MSN and LAST_TX_MSN are the first and last msn generated by app, respectively; FIRST_RX_FSN and LAST_RX_FSN are the fsn of the first and last f-tuple generated by the channel, respectively; and FIRST_RX_MSN and LAST_RX_MSN are the msn of the first and last f-tuple generated by the channel, respectively. Regarding the specific values of these constants, they are adjusted such that, for each pair, the range of integers in-between each pair does not overlap with any other pair of these constants. Moreover, they are adjusted such that the number of messages (msns) to be mtx-requested by the app and the number of frames (f-tuples) to be transmitted by the channel are both three. The chosen value is three since with a value of one the model checker will let the model evolve to the set I_1 of all possible states reachable after an mtx-request and a reception. This set I_1 is the set of all possible states from which a new mtx-request or reception could occur. Thus, with a value of three the model checker will verify the queries for two mtx-requests and two receptions from all states in I_1, i.e., from all possible initial states. Note that two additional mtx-requests and receptions are required to verify the properties related to the relative order between msns and fsns.

The range-checking queries for the remaining buffers are analogous to the above. Since all range-checking queries are satisfied, the remaining queries only need to consider the verified range of values for the different buffers. These queries are, grouped by property, the following:

P1 - Pass on integrity

The following query checks that for all reachable states, having an fsn generated by the channel in the app's reception f-tuple buffer implies that the f-tuple is in either one of the controllers' reception buffers. Since in the model an f-tuple is only inserted in a controller's reception buffer when a reception is modeled, the query proves property P1.

```
A□ ∀ (F: int[FIRST_RX_FSN, LAST_RX_FSN])
has_fsn(app.rx_fbuf, F) imply
(has_fsn(ctrls[CTRL_0].rx_fbuf, F) or
has_fsn(ctrls[CTRL_1].rx_fbuf, F))
```

P2 - Double reception implies single pass on

As described above, an f-tuple is only inserted in a controller's reception buffer when a reception is modeled. Moreover, once the automaton `obs` reaches the `finished` location, no more pass-ons occur. Thus, the following query proves property P2.

```
A□ ∀ (F: int[FIRST_RX_FSN, LAST_RX_FSN])
obs.finished and
has_fsn(ctrls[CTRL_0].rx_fbuf, F) and
has_fsn(ctrls[CTRL_1].rx_fbuf, F) imply
count_fsn(app.rx_fbuf, F) == 1
```

P3 - Pass on validity

Property P3 has the form *P unless Q*. This is equivalent to $\neg Q \rightarrow P$, where '$\neg$' indicates negation and '$\rightarrow$' material implication. This form is the one used by the query that proves the property:

```
A□ ∀ (F: int[FIRST_RX_FSN, LAST_RX_FSN])
obs.finished and
not (
(has_fsn(ctrls[CTRL_0].rx_fbuf, F) and
 not d.is_trusted[CTRL_0] and
 not has_fsn(ctrls[CTRL_1].rx_fbuf, F)) or
(has_fsn(ctrls[CTRL_1].rx_fbuf, F) and
 not d.is_trusted[CTRL_1] and
 not has_fsn(ctrls[CTRL_0].rx_fbuf, F))
) imply
has_fsn(app.rx_fbuf, F)
```

P4 - No duplicate pass on

Since by hypothesis one controller is always correct and has a correct link, all f-tuples generated by the channel are received by at least one controller. This makes the verification of this property straightforward:

```
A□ ∀ (F: int[FIRST_RX_FSN, LAST_RX_FSN])
count_fsn(app.rx_fbuf, F) ≤ 1
```

P5 - No pass on of self-received messages

This is already proved by one of the in-range queries. Specifically, the following query proves that all f-tuples stored in `app.rx_fbuf` originated from the channel, and were therefore not self-received.

```
A□ fsns_in_range(app.rx_fbuf,
FIRST_RX_FSN, LAST_RX_FSN)
```

P6 - *Ordered pass on*

Since the f-tuples generated by the channel have monotonically increasing integers as fsns, and f-tuples are appended to `app.rx_fbuf` one after the other, the following query proves property P6.

```
A□ is_sorted_by_fsn_fbuf(app.rx_fbuf)
```

P7 - Guaranteed transmission

By construction the modeled application mtx-requests an msn for each value in the range `[FIRST_TX_MSN, LAST_TX_MSN]`, both inclusive. Moreover, to model a transmission, one of the modeled controllers encapsulates an msn in an f-tuple and then adds that f-tuple to the `cha.rx_fbuf` buffer. Thus, the following query proves property P7:

```
A□ ∀ (M: int[FIRST_TX_MSN, LAST_TX_MSN])
obs.finished imply
has_msn_fbuf(cha.rx_fbuf, M)
```

P8 - Bounded retransmissions

This property can be verified, for instance, with the following query:

```
A◇ obs.finished
```

P9 - FIFO transmission

The query for this property follows a similar logic to the one of property P6:

```
A□ is_sorted_by_msn_fbuf(cha.rx_fbuf)
```

P10 - Bounded time to satisfy an mtx-request

The automaton modeling the application has only two locations, called `idle` and `wait_tx_success`. The automaton can only transition from the `idle` location to the `wait_tx_success` location if an mtx-request is modeled. Similarly, it can only transition from `wait_tx_success` back to `idle` if an mtx-request is satisfied, i.e. when a boolean variable indicates a tx-success. Moreover, from the query of property P7, it is known that the automaton modeling the application cycles to the `wait_tx_success` for every modeled mtx-request. Thus, it must only be shown that the automaton does not remain indefinitely in the `wait_tx_success` location. This is accomplished with the following query:

```
application.wait_tx_success ⇝ application.idle
```

9. Conclusion

The paper presents the design and verification of reCANdrv, a media redundancy management driver for the nodes of a ReCANcentrate network. ReCANcentrate is a CAN-compliant replicated star topology with enhanced error-containment and fault-tolerance mechanisms. The goal of reCANdrv is to allow a CAN application executing on a ReCANcentrate node to correctly exchange information through the ReCANcentrate network as long as a correct link with a correct controller remains available to the application's node. Since ReCANcentrate has been designed for safety-critical applications, it must be verified that this goal is achieved. To this end, the goal is formalized as a series of properties based on a few realistic assumptions. The properties are then verified by means of model checking. For this, a model of a ReCANcentrate node is created as a network of timed automata using UPPAAL. Afterwards, a series of queries, written in the UPPAAL query language, are presented. These show that the model indeed satisfies the properties.

The driver provides a virtual CAN controller interface to the CAN application executing on the nodes. Thus, it makes the existence of the underlying replicated communication medium transparent to the application. It is possible, therefore, to execute standard CAN applications and higher-layer protocols based on CAN on ReCANcentrate nodes.

Moreover, since the only requirement put on the channel is that it provides a single logical broadcast domain, reCANdrv works on any node connected by means of two CAN controllers, each with its own link, to a CAN channel that provides a single logical broadcast domain. Thus, the ReCANcentrate node architecture and reCANdrv do not only work with a replicated star topology (Figure 20d). Instead, reCANdrv can also be used with a simplex bus topology where each node is connected to that bus through two links (Figure 20a), a replicated bus topology where the two buses are coupled (Figure 20b), or a simplex star topology where each node is connected to the single star through two links (Figure 20c). Note that to best benefit from the driver, it is recommended that the channel provides error containment between the two links of each node.

Also note that if controller failures are considered negligible with respect to channel and link failures, then reCANdrv can be used on nodes implemented with low-cost off-the-shelf microcontrollers that provide dual on-chip CAN controllers, instead of nodes

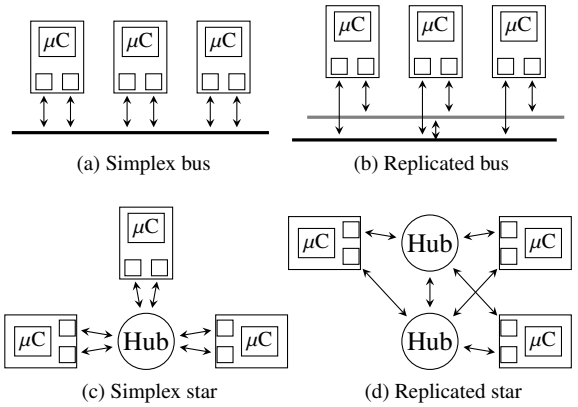


Figure 20: Example topologies for reCANdrv.

with two self-diagnosing controllers. In that case the reCANdrv design could remain the same, with the only change that it would not handle alert interrupts. However, the implication of not using self-diagnosing controllers would be that, if a controller starts to fail, the driver would no longer be able to provide to a ReCANcentrate node all the properties described in Section 4. For instance, property P1 would not be satisfied because a bogus frame could be read from a controller's reception buffer and be passed on to the application. As another example, property P4 may not be satisfied because a controller may generate a spurious reception interrupt.

References

[1] Manuel Barranco and Julián Proenza. Towards Understanding the Sensitivity of the Reliability Achievable by Simplex and Replicated Star Topologies in CAN. In *IEEE International Conference on Emerging Technologies and Factory Automation*, 2011. in press.

[2] Manuel Barranco, Luís Almeida, and Julián Proenza. ReCANcentrate: a replicated star topology for CAN networks. In *10th IEEE International Conference on Emerging Technologies and Factory Automation, 2005. ETFA 2005.*, volume 2, Catania, Italy, Sept. 2005. doi: 10.1109/ETFA.2005.1612714.

[3] Manuel Barranco, Julián Proenza, Guillermo Rodríguez-Navas, and Luís Almeida. An active star topology for improving fault confinement in CAN networks. *IEEE Transactions on Industrial Informatics*, 2(2):78–85, May 2006.

[4] Manuel Barranco, Julián Proenza, and Luís Almeida. Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate. *Computer*, 42(5): 66–73, May 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.145.

[5] Manuel Barranco, David Gessner, Julián Proenza, and Luiís Almeida. First prototype and experimental assessment of media management in ReCANcentrate. In *ETFA 2010. 15th IEEE International Conference on Emerging Technologies and Factory Automation, Bilbao, Spain*, 2010.

[6] Manuel Barranco, Julián Proenza, and Luís Almeida. Reliability improvement achievable in CAN-based systems by means of the ReCANcentrate replicated star topology. In *8th IEEE International Workshop on Factory Communication Systems*, pages 99–108, May 2010.

[7] Manuel Barranco, Julián Proenza, and Luís Almeida. Quantitative Comparison of the Error-Containment Capabilities of a Bus and a Star Topology in CAN Networks. *IEEE Transactions on Industrial Electronics*, 58(3):802–813, March 2011. ISSN 0278-0046. doi: 10.1109/TIE.2009.2036642.

[8] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. November 2004. URL `http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf`.

[9] Robert Bosch GmbH. CAN specification version 2.0. Technical report, Robert Bosch GmbH, 1991.

[10] Giuseppe Buja, Juan R Pimentel, and Alberto Zuccollo. Overcoming Babbling-Idiot Failures in CAN Networks : A Simple and Effective Bus Guardian Solution for the FlexCAN Architecture. *IEEE Transactions on Industrial Informatics*, 3 (3):225–233, 2007.

[11] Daniela Cancila, Roberto Passerone, Tullio Vardanega, and Marco Panunzio. Toward Correctness in the Specification and Handling of Non-Functional Attributes of High-Integrity Real-Time Embedded Systems. *IEEE Transactions on Industrial Informatics*, 6(2):181–194, May 2010. ISSN 1551-3203. doi: 10.1109/TII.2010.2043741.

[12] Goran Cengic and Knut Akesson. On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics. *IEEE Transactions on Industrial Informatics*, 6(2):145–154, May 2010. ISSN 1551-3203. doi: 10.1109/TII.2010.2040393.

[13] Goran Cengic and Knut Akesson. On Formal Analysis of IEC 61499 Applications, Part A: Modeling. *IEEE Transactions on Industrial Informatics*, 6(2): 136–144, May 2010. ISSN 1551-3203. doi: 10.1109/TII.2010.2040392.

[14] Robert Davis, Alan Burns, Reinder Bril, and Johan Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35:239–272, 2007.

[15] L.-B. Fredriksson. CAN for critical embedded automotive networks. *IEEE Micro*, 22(4):28–35, July 2002. ISSN 0272-1732. doi: 10.1109/MM.2002.1028473.

[16] David Gomez-Gutierrez, Guillermo Ramirez-Prado, Antonio Ramirez-Trevio, and Javier Ruiz-Leon. Observability of Switched Linear Systems. *IEEE Transactions on Industrial Informatics*, 6(2):127–135, May 2010. ISSN 1551-3203. doi: 10.1109/TII.2009.2034737.

[17] D Herrero-Perez and H Martinez-Barbera. Modeling Distributed Transportation Systems Composed of Flexible Automated Guided Vehicles in Flexible Manufacturing Systems. *IEEE Transactions on Industrial Informatics*, 6(2):166–180, May 2010. ISSN 1551-3203. doi: 10.1109/TII.2009.2038691.

[18] M Kloetzer, C Mahulea, C Belta, and M Silva. An Automated Framework for Formal Verification of Timed Continuous Petri Nets. *IEEE Transactions on Industrial Informatics*, 6(3):460–471, August 2010. ISSN 1551-3203. doi: 10.1109/TII.2010.2050001.

[19] Wolfhard Lawrenz. *CAN System Engineering. From Theory to Practical Applications*. Springer, 1997.

[20] Microchip. *dsPIC30F Family Reference Manual*, 2006.

[21] Julián Proenza and José Miro-Julia. MajorCAN: A modification to the Controller Area Network protocol to achieve atomic broadcast. *IEEE International Workshop on Group Communication and Computations, Taipei, Taiwan*, 2000.

[22] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 150, Washington, DC, USA, 1998. IEEE Computer Society.

[23] J. Rufino, P. Veríssimo, and G. Arroz. A Columbus' egg idea for CAN media redundancy. In *Digest of Papers, The 29th International Symposium on Fault-Tolerant Computing Systems*. IEEE, June 1999.

[24] Jose Rufino, Carlos Almeida, Paulo Verissimo, and Guilherme Arroz. Enforcing Dependability and Timeliness in Controller Area Networks. In *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, pages 3755–3760. IEEE, November 2006. ISBN 1-4244-0390-1. doi: 10.1109/IECON.2006.348102.

[25] Michael Short and Michael J. Pont. Fault-Tolerant Time-Triggered Communication Using CAN. *IEEE Transactions on Industrial Informatics*, 3(2):131–142, May 2007. ISSN 1551-3203. doi: 10.1109/TII.2007.898477.