Split, Send, Reassemble: A Formal Specification of a CAN Bus Protocol Stack*

Rob van Glabbeek Peter Höfner
Data61, CSIRO, Sydney, Australia
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
rvg@cs.stanford.edu Peter.Hoefner@data61.csiro.au

We present a formal model for a fragmentation and a reassembly protocol running on top of the standardised CAN bus, which is widely used in automotive and aerospace applications. Although the CAN bus comes with an in-built mechanism for prioritisation, we argue that this is not sufficient and provide another protocol to overcome this shortcoming.

1 The CAN Bus Protocol

"A *Controller Area Network (CAN bus)* is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other in applications without a host computer." [1] Robert Bosch GmbH developed it in the 80s and published the latest release in 1991 [9].

The protocol is message-based and was designed specifically for automotive applications but is now also used in other areas such as aerospace, maritime and medical equipment. The CAN bus was designed to broadcast many short messages to the entire network. The broadcast mechanism provides data consistency in every node of the system. Typical information sent are sensor data, such as speed or temperature. Due to its simplicity, it is easy to implement; however, its capabilities are rather limited, in particular w.r.t. payload and security.

CAN Bus Limitations. In the CAN specification, version 2.0, there are two different message formats to send data in a (typical) CAN network [9]. The only difference between the two formats is that the standard frame format supports a length of 11 bits for the identifier, and the extended frame supports a length of 29 bits. The payload of both messages is *8 bytes only*.[2]

CAN is a low-level protocol and offers no (standard) support for any security feature. Applications are expected to deploy their own security mechanisms. Failure to do so can result in various sorts of attacks. A lot of media attention was generated when cars were hacked and remotely controlled. The best security mechanism is to ensure that only trustworthy applications have access to the CAN bus. An alternative is the use of authentication and encryption, for instance through HMAC [7, 14] or GMAC [3].

The Need for Fragmentation and Reassembly. As soon as encryption and authentication is implemented, the messages used will be longer than 8 bytes; and even without encryption messages are often that long. Therefore there is the need for a fragmentation/reassembly protocol. In this paper we will present such a protocol on top of the CAN bus, which remains unchanged. One reason why we designed our own protocols is that they carry less overhead than off-the-shelf solutions.

*Supported by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179.

[1] http://en.wikipedia.org/wiki/CAN_bus (accessed February 23, 2017)

[2] CAN-FD introduces frames with more than 8 bytes; but so far this extension of CAN has not been adopted by the industry.

Preliminary Report. Final version to appear in:
MARS 2017

© R.J. van Glabbeek & P. Höfner
This work is licensed under the
Creative Commons Attribution License.

The Need for Prioritisation. The CAN protocol comes with an in-built priority mechanism. It uses a bit-wise comparison method of contention resolution, which requires all nodes on the CAN bus to be synchronised at the point when transmission begins.

"The CAN specifications use the terms 'dominant' bits and 'recessive' bits where dominant is a logical 0 [...] and recessive is a logical 1 [...]. If one node transmits a dominant bit and another node transmits a recessive bit then there is a collision and the dominant bit 'wins'. This means there is no delay to the higher-priority message, and the node transmitting the lower priority message automatically attempts to re-transmit six bit clocks after the end of the dominant message. This makes CAN very suitable as a real time prioritized communications system."[3] A node that sent a recessive bit and detected a collision ceases transmission and will attempt a retransmission of its own message later on. Since CAN identifiers are unique for each message type and sender, and constitute the first part of any message, all but one nodes will stop while transmitting the CAN identifier.

While this in-built priority mechanism works if CAN drivers and CAN controllers[4] are considered only, it is not always sufficient due to the problem of *priority inversion*. We illustrate this by an example (cf. Fig. 1). Assume three nodes are attached to the CAN network: a mission board, a microcontroller and a camera—this is part of the architecture of our research vehicle (cf. Sect. 2).

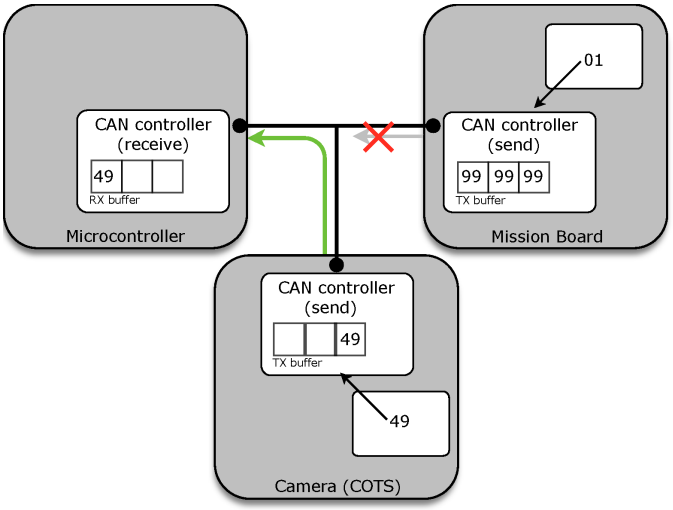

Figure 1: Blocking Behaviour of High-Priority Messages

Assume that the camera sends a constant stream of messages of medium priority—say with CAN ID 49. This message stream could be interrupted by a message of high priority, let say 01, which should be sent from the mission board to the microcontroller. Unfortunately, before this really important message is generated, another application on the mission board generates 3 absolutely unimportant message of low priority (here CAN ID 99). They are transferred to the CAN controller and stored in the transmission (TX) buffer; they are scheduled to be sent via the CAN bus as soon as possible. Since there is a constant stream of more important messages (the ones stemming from the camera), these messages are never sent. As a consequence the high-priority message cannot be passed on to the TX buffer, and hence is stuck at the mission board.[5]

This example shows that there is need for another priority mechanism running on each node separately. Such a mechanism should retract one of the low-priority messages from the TX buffer of the mission board and replace this message with the high-priority one, which will be sent immediately; after

[3] `https://en.wikipedia.org/wiki/CAN_bus` (accessed February 23, 2017)

[4] A node on a CAN bus is equipped with a *CAN controller* and a *CAN transceiver*. The CAN controller has a small number (typically 3) of *TX buffers*, where outgoing messages are stored before transmission, and a small number of *RX buffers*, which store incoming messages. The CAN transceiver manages the actual transmission of messages via the CAN bus—it normally sends the message of highest priority stored in a TX buffer first. The software that sends messages to the controller (which are then stored in the TX buffers), initiates the cancellation of messages, and requests messages received, is called a *CAN driver*.

[5] Obviously this example could be avoided by having multiple CAN controllers on the mission board; but this cannot be guaranteed and often far more applications run on a single node than the number of CAN controllers available.

that the low-priority message will be stored back to the buffer. There are two possible solutions for such a priority mechanism: (a) it could be integrated in the CAN driver, or (b) it is implemented as yet another protocol that works between the CAN driver and the fragmentation/application layer. We decided to implement the latter option to have a clear separation of concerns.

The protocol, called *multiplexer* and formally specified in Sect. 7, will be an interface between the CAN driver and several instances of the fragmentation protocol.

Our Contribution. We present a protocol stack to be used on top of the CAN bus, consisting of fragmentation and reassembly protocols, as well as a multiplexer. It has been formally modelled and partly analysed, implemented, and is successfully used in a research vehicle. The hardware as well as (the implementation of) the CAN bus protocol remains untouched; hence our protocol stack is ready to be deployed on any system featuring a CAN bus.

2 Our Research Vehicle

We use our protocol stack within a research vehicle, namely a small quadcopter. This off-the-shelf quadcopter with customised hard- and software was developed as part of the SMACCM (Secure Mathematically-Assured Composition of Control Models) project, a 4.5 year 18 million USD project funded to build highly hack-resilient unmanned aerial vehicles under DARPA's HACMS (High-Assurance Cyber Military Systems) program. The team consists of formal verification and synthesis groups in Rockwell Collins, Data61 (formerly NICTA), Galois Inc, Boeing and the University of Minnesota.

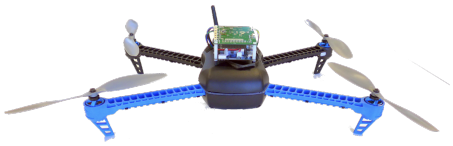

Figure 2: Our Research Vehicle

Our quadcopter is equipped with two boards: a mission board and a control board. This architecture is artificially made more complex by adding a trusted gateway and an untrusted COTS (commercial off-the-shelf) component on the bus to introduce some of the complexities of larger air vehicles. One of the goals of the project is to show that even if an intruder gets hold of the COTS component, or of one of the untrusted applications running on the mission board (e.g. Linux), this will not invalidate essential security properties of the vehicle.

The use of the CAN bus for communication between the two boards and the COTS component (via the gateway) is a design decision based on the popularity of CAN in aviation and automotive applications. It shows that the use of CAN does not stand in the way of hack-resilience.

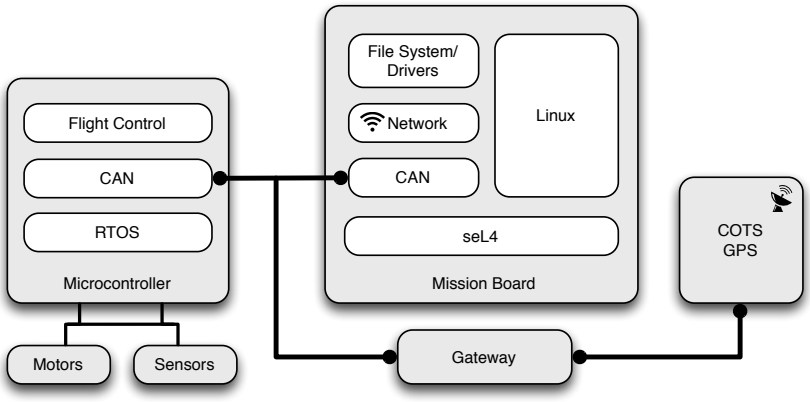

Figure 3: Architecture of the air vehicle with use of CAN bus.

Fig. 3 (and in an abstract version also Fig. 1) shows a sketch of the vehicle's architecture. The left hand side shows the flight controller, which is a pixhawk board with an ARM Cortex M4 CPU; it has direct connections to sensors and actuators. The mission board in the central part of

Fig. 3 is more powerful: a TK1-SOM board with an ARM Cortex A15 CPU with virtualisation extensions running the seL4 microkernel for providing isolation in a mix of trusted and untrusted applications on top. The bottom and right-hand-side boxes in Fig. 3 show a gateway between the trusted part of the internal network on the left and the untrusted part of the internal network that connects to an unverified component on the right. The purpose of the gateway is to validate network packets from the right and only let through well-formed packets to allowed destinations.

The gateway is essential to achieve hack-resilience in the presence of untrusted components. Giving such components direct access to the CAN bus might give rise to denial-of-service (DoS) attacks: a hostile component might supply a continuous stream of high-priority packets, thereby inhibiting any other traffic. The gateway at least ensures that any message coming from a COTS component has a CAN identifier that labels it as such. The identifier gives the message a lower priority than safety critical messages from trusted components. Our multiplexer is designed to ensure that in these situations safety critical messages cannot be blocked by untrusted components. It is possible to entrust all security issues to the gateway. However, a simpler gateway need not investigate and restrict information flowing towards the COTS component; in this case safety critical messages that need to be kept private could be encrypted.

For the protocols presented in this paper and used in our research vehicle, we assume a functionally correct CAN bus. The proposed architecture and kernel-provided isolation on the mission board minimises the potential attack surface compared to standard systems. Messages that need splitting arrive from a ground station—which plans and manages missions—to the mission board and are forwarded to the microcontroller; the messages itself are often longer than 8 bytes, and when encrypted exceed definitely the payload of a standard CAN message.

3 Assumptions and Requirements

During the design of the fragmentation protocol we had to make some assumptions. All assumptions are realistic and can be assumed for our case study without loss of generality.

Assumptions on the CAN Bus.

1. For the verification of two central correctness properties of the protocol—any message received has been sent, i.e. split messages are not reassembled in the wrong way, and any message sent is received—we assume a perfect channel and assume that *every message sent via the CAN bus will be received* by all nodes that are connected to the CAN bus. However, for our more basic correctness properties, such as the absence of deadlocks in protocol components or in the entire protocol stack, and the unreachability of error states, we do not make such an assumption.
   (a) The CAN protocol specifies an automatic retransmission of faulty messages after transmitting an error frame. Error frames may be sent by transmitting or receiving nodes. This happens on a lower protocol layer than the one modelled here.
   (b) In case resending of an entire message is needed (e.g. if fragments are lost), we leave this task to the application layer/user. The reason for this decision is that most information sent via the CAN bus is time-sensitive; so if the information is not sent immediately, it will be outdated. Examples are GPS-coordinates or telemetry data.
2. We allow for the possibility that a CAN message is sent and received twice—possibly because one of the receivers used the error frame to ask for a resend. Our protocol is required to deal with such a repeated fragment.
3. Messages sent over the CAN bus are *not reordered*. This is a realistic assumption. The CAN protocol sends one packet after the other; fragmented messages stemming from different sources

may be interleaved, but reordering of messages sent by a single sender does not occur. Even if a packet is lost, the resending happens before the next frame is processed and sent.
A CAN controller allows overtaking of low priority messages by high priority messages by first offering the highest priority message stored in its TX buffers for transmission. The above assumption therefore only rules out reordering after messages have been transmitted on the CAN bus.

Requirements on the Input Data.

4. Every message type, as determined by a CAN identifier (ID), has a *unique sender*.[6] This requirement is reasonable and reflects the CAN protocol in the way it is used in the automotive industry. As a consequence it is impossible to have two senders sending messages with the same ID at the same time, and hence collisions are avoided.[7] By using wrong identifiers this requirement can be violated, yielding collisions and message loss. Thus it should be ensured that this requirement is satisfied.
5. Every message we are going to split has a *fixed length*, determined by its message type. As a consequence we know in advance into how many fragments a message needs to be split. In case the message lengths can vary we assume that there is an upper bound and that shorter messages are extended by 'dummy bits'.

Requirements for our Fragmentation Protocol.

6. The *distribution of the message IDs* is fixed at compile time. We do not restrict the choice, as long as the assumption 'unique sender for every message ID' is maintained, and adjacent IDs are allocated to adjacent fragments of messages of the same type.
7. We assume that the application layer, after submission of a message to an instance of the splitting protocol for transition over the CAN bus, will wait for an acknowledgement (positive or negative) before submitting a new message to that instance of the splitting protocol. The application layer may at any time submit a cancellation request for the last message submitted; this will speed up the (now likely negative) acknowledgement.
8. Our protocol has to support *legacy nodes*. Independent of the software components we are adding (fragmentation protocol, authentication, encryption), the original CAN protocol should still be available and it must be possible to send and receive ordinary CAN messages.

Our fragmentation protocol uses a new CAN identifier for each fragment of each message type. One might wonder if this does not lead to a shortage of CAN IDs. Within our research vehicle this problem did not occur, and experts from the automotive side have ensured us that in typical applications there is an abundance of unused CAN IDs, in particular when using the extended frame format with 29-bit identifiers that allow over half a billion identifiers.

4 Related Work

Several fragmentation protocols to be deployed on top of the CAN bus have been developed in the past.

The *ISO-TP* [13] or *ISO 15765-2* protocol is an international standard for sending data packets over a CAN bus that exceed the 8 byte maximum payload. ISO-TP splits longer messages into multiple frames, adding metadata that allows the interpretation of individual frames and reassembly into a complete message packet by the recipient. The protocol can handle message of up to 4095 bytes. The first fragment can only carry up to 7 bytes when using *normal addressing*, instead of the standard 8 bytes. Hence every message longer than 7 bytes should be split; as a consequence special care has to be taken so that this protocol can handle legacy messages.

[6] Even if two applications would send messages of the same message type, one could use two different IDs.

[7] The in-build priority messages will take care of messages with different IDs sent at the same time (see Sect. 1).

Shin [10, 11] follows the spirit of ISO-TP and describes a protocol similar to ours. Due to his design the payload of each CAN message is split up into an 8-bit message identifier, a 7 bit sequence number, which points to the next fragment, and the actual payload, which has a maximum capacity of 6 bytes—2 bytes less than in original CAN frames and in our approach. As a consequence we have far less overhead than Shin's approach. Moreover, his reassembling routine does not take packet loss into account.

The *TP 2.0* protocol, sometimes also called *VW TP 2.0*, (see e.g. [12]) introduces a connection-oriented approach. It first sends a couple of messages to establish a "channel" between the sender and the recipients, then exchanges and sets up channel parameters such as the number of frames (fragments) to be sent, and finally transmits the message (by encoding the channel number into the CAN ID). Although the used CAN frames can use the full pay load of 8 bytes, the overhead lies in the setup-phase.

The *Service Data Object* (SDO) protocol,[8] which is part of CANopen[9] (see e.g. [4])—a communication protocol for embedded systems used in automation—also implements segmentation and desegmentation of longer messages. SDO is used for setting and for reading values from the object dictionary of a remote device. Because the values can be larger than the 8 bytes limit of a CAN frame, the SDO protocol implements also a fragmentation protocol. However, the fragmentation itself is a subprotocol and using SDO or even the entire CANopen infrastructure is again too much of an overhead.

All of these protocols bear a more or less heavy overhead w.r.t. the number of messages. Moreover, none of these solutions appear compatible with the priority mechanism offered by our multiplexer. In fact, adopting any of them stands in the way of inserting a multiplexer between the fragmentation protocol and the CAN driver, thus taking care of priority inversion, as we do here. Therefore, we designed our own simple protocol that exactly meets our needs; by using the vast amount of CAN identifiers available, our protocol does not have any overhead w.r.t. the number of messages sent.

5 Protocol Stack: Overall Structure and Informal Description

Before presenting our formal specifications, including formally defined data structures and unambiguous process-algebraic specifications, we describe the overall structure of our protocol stack, including all messages sent between components. By abstracting from formal details we discuss the 'big picture'.

Figure 4 gives a schematic representation of our protocol stack. Every hardware component (node) contains exactly one instance of each of the two protocol chains. We distinguish 4 different layers.

- The application layer: applications are components that send messages to and receive messages from the CAN bus (via the other 3 layers).
- The CAN layer: this layer combines the CAN controller and the CAN driver. We will model a simple instance of this layer.

The remaining two layers connect the application with the CAN layer—they split and reassemble messages if necessary, and handle messages with high priority first.

- The fragmentation/reassembly layer: the fragmentation protocol receives a message from an application, and, depending on the type of the message, simply forwards the message if it is a short or legacy, or otherwise splits the message into fragments of 8 bytes each. These fragments have the form of standard CAN messages and can (now) be sent via the CAN bus. The reassembly protocol receives such fragments that were sent via the CAN bus and forwarded by the CAN driver. It stores the fragments until a full message can be reassembled and transmitted to the application.

[8] `http://www.can-cia.org/can-knowledge/canopen/sdo-protocol/`
[9] `http://www.can-cia.org/can-knowledge/canopen/canopen/`

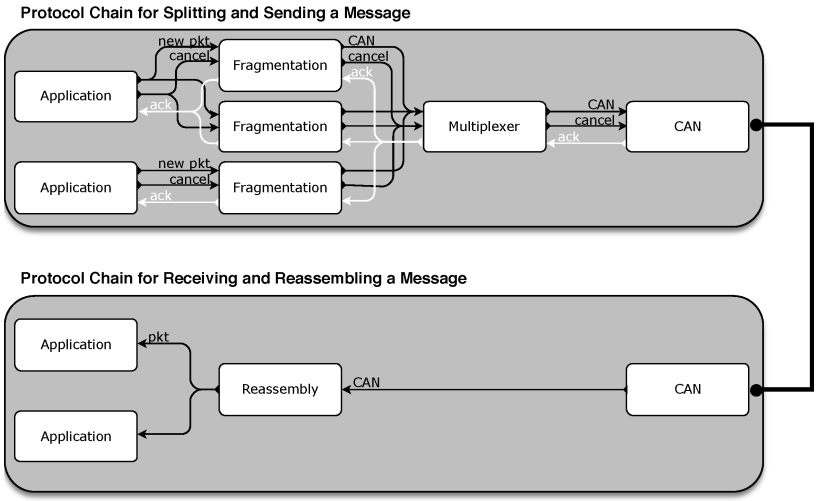

Figure 4: Message passing between the different components

- The multiplexer accepts messages from different instances of the fragmentation protocol (all running on the same hardware) and stores them in a priority queue. It always sends the message that needs to be transmitted next to the CAN driver, if necessary after cancelling a lower-priority message sitting in the TX buffer of the CAN controller. This prevents the blocking example of Sect. 1.

All these layers exchange information through message passing. In the remainder of the section we will discuss the different types of messages that occur in our protocol chain.

Protocol Chain for Splitting and Sending a Message. Any application is allowed to submit *new messages* `newpkt`. Every such message contains a message type and a payload. Depending on the type the message is sent to a particular instance of the fragmentation protocol. For this we assume that *for each message type there is exactly one instance of the fragmentation protocol*; an instance, however, is allowed to handle multiple message types, provided that they all stem from the same application. An application usually sends one new message at a time; in case a second packet is injected before the previous one has been fully handled, the protocol yields an error and deadlocks. To avoid this scenario the fragmentation protocol returns an acknowledgement message `ack`, informing the application that the handling of the message is finished—for each `newpkt` received exactly one `ack` is sent. The acknowledgement can be positive or negative, depending on whether the message was successfully sent via the CAN bus or not. The application has also the possibility to inject a `cancel`-message to the fragmentation protocol, which will then stop handling the last message injected by the application, unless it is already finished with it.

The fragmentation protocol receives new messages from the application and splits them; the resulting CAN messages are passed on to the multiplexer. As in case of an application, the fragmentation protocol awaits an acknowledgement `ack` (positive or negative) before sending the next CAN message. In case the fragmentation protocol receives a cancellation request, it stops splitting a message, and in case the multiplexer has not yet acknowledged the last fragment submitted to it also informs the multiplexer about the cancellation request by sending it a `cancel`-message. The multiplexer returns exactly one acknowledgement for each fragment received; it does not accept a second fragment from the same fragmentation instance before this `ack` has been sent. It accepts a cancellation request at any time.

The multiplexer schedules all messages received in an appropriate order (high priority messages first); the messages involved in this are again `can`, `cancel`, and `ack`. The CAN-messages that were sent by the multiplexer are handled by the CAN layer and end up in the TX buffer of the CAN controller, from where they are transmitted over the CAN bus.

Protocol Chain for Receiving and Reassembling a Message. After the messages were transmitted via the CAN bus, they are stored in the RX buffer of the CAN controller. The CAN layer stores and handles all incoming messages of type `can` and forwards them to the reassembly protocol, which stores these messages—there is no pendant to the multiplexer in the receiving chain. As soon as a packet is fully received and reassembled, it is delivered to the appropriate application, using a message of type `pkt`.

A formal specification of the multiplexer is presented in Sect. 7. Formal specifications of all the above-mentioned protocols are given in App. B. This includes the processes themselves, specified in the process algebra AWN (see below), as well as a detailed definition of the data structure involved.

6 AWN: A Specification Language for Protocols[10]

Ideally, any specification is free of ambiguities and contradictions. Using English prose only—as is still state of the art in protocol specification—this is nearly impossible to achieve. The use of *any* formal specification language helps to avoid ambiguities and to precisely describe the intended behaviour. The choice of a specification language is often secondary, although it has high impact on the analysis.

For this paper we choose the modelling language AWN [5], which provides the right level of abstraction to model key protocol features, while abstracting from implementation-related details. As its semantics is completely unambiguous, specifying a protocol in such a framework enforces total precision and the removal of any ambiguity. AWN is tailored for modelling and verifying routing and communication protocols and therefore offers primitives such as **unicast** and **multicast/groupcast**; it defines the protocol in a pseudo-code that is easily readable—the language itself is implementation independent; and it offers some degree of proof automation and proof verification [2, 1], using Isabelle/HOL [8].

AWN is a variant of standard process algebras, extended with a local broadcast mechanism and a novel *conditional unicast* operator—allowing error handling in response to failed communications while abstracting from link layer implementations of the communication handling—and incorporating data structures with assignments; its operational semantics is defined in [5].

We use an underlying data structure (described in detail in App. B) with several types, variables ranging over these types, operators and predicates. First order predicate logic yields terms (or *data expressions*) and formulas to denote data values and statements about them. Our data structure always contains the types `DATA`, `MSG`, `ID` and $\mathscr{P}(\texttt{ID})$ of *application layer data*, *messages*, *identifiers* and *sets of identifiers*. The messages comprise *data packets*, containing application layer data, and *control messages*.

In AWN a network is modelled as a parallel composition of components. Here any party in the network that can be addressed as the recipient of a message is a component. On each component several processes may be running in parallel. Components communicate with their direct neighbours, which in our current application are all other components in the network.

The *Process expressions* are given in Table 1. They should be understandable without further explanation; we add a short description in App. A.

7 A Formal Specification of the Multiplexer

In this section we present two out of four AWN-processes that entirely specify our multiplexer.

The Main Loop. The basic process $\texttt{MULTIPLEXER}_H$ (Process 9) receives messages from the fragmentation protocol or the CAN driver. Since the multiplexer is not always ready to receive messages, we equip

[10] Parts of this section are published in [6].

$X(exp_1,\ldots,exp_n)$	process name with arguments
$P+Q$	choice between processes P and Q
$[\varphi]P$	conditional process: proceed as P, but only if φ evaluates to `true`
$[\![\texttt{var} := exp]\!]P$	assignment followed by process P
broadcast$(ms).P$	broadcast message ms followed by P
groupcast$(dests, ms).P$	multicast ms to all destinations $dests$
unicast$(dest, ms).P \blacktriangleright Q$	unicast ms to $dest$; if successful proceed with P; otherwise with Q
deliver$(data).P$	deliver data to application layer
receive$(\texttt{msg}).P$	receive a message and store its contents in the variable `msg`
(ξ, P)	process P with initial valuation of its variables
$V \langle\langle W$	parallel valuated processes on the same component
$id:V$	addressed component
$C \| D$	parallel composition of addressed components

Table 1: Process Expressions

the process with an in-queue (see App. B); so technically the multiplexer receives a message from this queue. $\texttt{MULTIPLEXER}_H$ maintains two data variables `prio` and `txs`. The former implements a priority queue which contains all CAN messages to be sent via the CAN bus; the later is a local storage which keeps track of the CAN IDs currently sent by or stored in the TX buffers of the CAN controller.

Process 9 Multiplexer—Main Loop[11]

$\texttt{MULTIPLEXER}_H(\texttt{prio},\texttt{txs}) \stackrel{def}{=}$

1. **receive**(`msg`) .
2. (
3. [`msg` = `can`(`cid`,`data`)] /* new fragment */
4. $\texttt{NEW\_CAN}_H$(`msg`,`prio`,`txs`)
5. + [`msg` = `cancel`(`cid`)] /* cancellation message received */
6. $\texttt{CANCEL\_C}_H$(`cid`,`prio`,`txs`)
7. + [`msg` = `msgd`(`tid`,`ack`(`suc`))] /* message from CAN controller */
8. $\texttt{ACK\_C}_H$(`suc`,`tid`,`prio`,`txs`)
9.)

First, a message has to be received (Line 1). After that, the process $\texttt{MULTIPLEXER}_H$ checks the type of the message and calls a process that can handle this message: in case a CAN message is received from the fragmentation protocol, the process $\texttt{NEW\_CAN}_H$ is called (Line 4); in case of an incoming cancellation request the process $\texttt{CANCEL\_C}_H$ is executed (Line 6); and in case a message from the CAN driver is read, the process $\texttt{ACK\_C}_H$ is called (Line 8). In case a message of any other type is received, the process $\texttt{MULTIPLEXER}_H$ deadlocks; it is a proof obligation to check that this will not occur.

New CAN Message. In case a new CAN message is sent from an instance of the fragmentation protocol, the process $\texttt{NEW\_CAN}_H$ stores the CAN message and determines whether the newly received message is important enough to be forwarded directly to the CAN driver. The formal specification is shown in Process 10.

The received CAN message is first stored in the queue `prio` (Line 2), which contains all messages to be sent via the CAN bus. The protocol just stores the newly received message, it does not check for emptiness of $\texttt{prio}(\texttt{cid})$. Therefore, to guarantee that no message is lost the property $\texttt{prio}(\texttt{cid}) = \bot_{\texttt{msg}}$ needs to hold before Line 2 is executed; it needs to be proven. The protocol then determines whether the message should directly be forwarded to the CAN driver—this is the case if the CAN ID is among the

[11] The numbering of the processes is according to App. B.

Process 10 New CAN Message Received[11]

```
NEW_CAN_H(msg,prio,txs) =def
 1. [ msg = can(cid,data) ]        /* distill cid out of msg */
 2.   ⟦prio(cid) := msg⟧          /* store message in priority queue */
 3.   (
 4.     [ cid ∈ n_best(prio) ]     /* message should be scheduled */
 5.       (
 6.         [ tx_cid(tid,txs) = ⊥_cid ]     /* TX buffer tid is free */
 7.           ⟦txs(tid) := (cid,false)⟧
 8.           unicast(C_H, msgd(tid,msg)) .     /* pass message to CAN driver, to put in free slot */
 9.           MULTIPLEXER_H(prio,txs)
10.         + [ ∀tid ∈ TX : tx_cid(tid,txs) ≠ ⊥_cid ]     /* cancel message with lowest priority */
11.           (
12.             ⟦wid := getWorstTX(txs)⟧     /* identify TX buffer containing lowest CAN ID */
13.             (
14.               [ tx_abort(wid,txs) = false ]     /* TX buffer wid is still active */
15.                 ⟦txs(wid) := (tx_cid(wid,txs),true)⟧     /* set the abort-flag of buffer wid */
16.                 unicast(C_H, msgd(wid,cancel())) .     /* cancel contents of buffer wid */
17.                 MULTIPLEXER_H(prio,txs)
18.               + [ tx_abort(wid,txs) = true ]     /* TX was already asked to clean up */
19.                 MULTIPLEXER_H(prio,txs)
20.             )
21.           )
22.       )
23.     + [ cid ∉ n_best(prio) ]     /* message not important enough to be scheduled right now */
24.       MULTIPLEXER_H(prio,txs)
25.   )
```

n messages with lowest CAN IDs currently stored in `prio` (Line 4). Here n equals the number `#TX` of TX buffers available in the CAN controller. Lines 5–22 present all actions to be performed in case the message is forwarded to the CAN driver.

In case there exists an empty TX buffer `tid`, which is currently not used, the message should be sent to this TX buffer, and there is no need to erase a used TX buffer. The empty buffer `tid` is chosen in Line 6.[12] The CAN message is then forwarded to the connected CAN driver $\mathtt{C}_H$ in Line 8. Since the CAN driver needs also the name of the TX buffer to be used, the value `tid` is sent next to the CAN message `msg`. The multiplexer also updates the local variable `txs` (Line 7), which keeps track of those CAN identifiers that are currently sent by or stored in the TX buffers. By this, the newly received message has been handled and the process can return to the main routine (Line 9).

In case all available TX buffers are used (Line 10), the least important message—the CAN message with the largest CAN ID—needs to be removed from the TX buffer and rescheduled later. This avoids the blocking example presented earlier. In Line 12 the process $\mathtt{NEW\_CAN}_H$ determines the name of the TX buffer that contains the 'worst' message currently handled for sending. The CAN message that should be stored in this particular TX buffer cannot be put there immediately; a cancellation request needs to be sent first, and an acknowledgement needs to be received that informs the multiplexer about a free TX buffer. The routine checks whether a cancellation request was sent earlier, using the function $\mathtt{tx}_{\mathtt{abort}}$. If this is the case, it returns straight to the process $\mathtt{MULTIPLEXER}_H$; otherwise a cancellation message is sent to the CAN driver $\mathtt{C}_H$, identifying the TX buffer that needs cancellation (Line 16).

[12] Since `tid` is a free variable, it will be instantiated with a value that validates $\mathtt{tx}_{\mathtt{cid}}(\mathtt{tid},\mathtt{txs}) = \bot_{\mathtt{cid}}$; so the condition in the guard is satisfied iff $\exists \mathtt{tid} \in \mathtt{TX} : \mathtt{tx}_{\mathtt{cid}}(\mathtt{tid},\mathtt{txs}) = \bot_{\mathtt{cid}}$.

If the newly received CAN message is not important enough to be forwarded to the CAN driver immediately (Line 23), the process $\texttt{NEW\_CAN}_H$ just returns to the main process (Line 24), where it awaits a new message. The stored message will be handled later when a TX buffer becomes available.

8 Properties and Formal Analysis

After defining the splitting and reassembly protocol in a formal and unambiguous manner (including the multiplexer), we can now focus on verification tasks. A detailed analysis and verification is out of the scope of this paper—which concentrates on the necessity of the protocols and their formal specification. In this section we list a series of desired properties—a first analysis using model checking techniques indicates that these properties are satisfied.[13]

Unreachability of `ERROR` State. In our specification we use a special state `ERROR` that is reached by a component process whenever it receives an input that is unexpected, and for which no proper response in envisioned. As `ERROR` is not an AWN primitive, we proposed to simply implement it in terms of AWN primitives as a deadlock: $\texttt{ERROR}() \stackrel{def}{=} \texttt{[false]ERROR}()$.

Our first requirement on the correctness of the overall protocol is that none of its components will ever reach the `ERROR` state. Since the application layer is not part of our specification, we cannot show that it behaves properly. Instead, we have to formulate a requirement on the communications between the application layer and our protocol; we only require the unreachability of `ERROR` under that condition.

The Protocol is Deadlock Free. Another requirement is that our protocol is deadlock free, in the sense that each reachable state has an outgoing transition. As termination of our protocol is not envisioned, a deadlock is a clear case of undesirable behaviour. This requirement does not rule out any state where no further activity occurs due to lack of input from the application layer, for the possibility of such input is modelled as an outgoing transition.

Each Component of the Protocol is Deadlock Free. A related requirement is that each component in our specification is deadlock free. The components are all instances of the fragmentation protocol, the multiplexer, the CAN receiver, the CAN sender, and the reassembly protocol. Optionally, the input queue of the multiplexer can be regarded a separate component, too.

This requirement is neither weaker nor stronger than the above requirement that the entire protocol is deadlock free, for it could be that a deadlock of the entire protocol occurs when two components fail to properly communicate, even though each of them could make progress if only the other one cooperated.

Any Message Received Has Been Sent. Here a message counts as 'sent' when it is submitted by the application to the fragmentation protocol; it counts as 'received' when it is passed on by the reassembling protocol at each node listed as a destination of the message to the corresponding application layer.

This requirement is definitely violated in case our reassembly protocol reassembles messages the wrong way. Such a situation can occur in case of message loss on the CAN bus.

For this reason, we can only hope to establish the property under the condition that no message loss occurs. In our model, this is quite simple, as the possibility of message loss is not modelled. This follows the advice of experts on the CAN bus, saying that any message that is actually sent from a TX buffer at the transmitting end is always picked up by a RX buffer at the receiving end.

Any Message Sent Is Received. This may be regarded as the central requirement of the protocol. In fact, some of the requirements above are in some sense entailed by this requirement, as the presence of deadlocks will surely manifest itself as failure to handle and receive further messages.

[13] In fact we did find an error in the multiplexer that has been eliminated in the current version.

Obviously, this requirement cannot be guaranteed if there is message loss on the CAN bus. Thus, as before, we assume that no loss on the CAN bus occurs. Since the CAN bus handles higher priority messages in preference to lower priority messages, a given message that is submitted to the protocol by the application layer at a node may never be selected by the CAN bus if a steady stream of higher-priority messages is send by another node; so this property does not hold. Consequently, further the assumption that there is no steady stream of higher priority messages is required.

The property is also violated if the input queue of the multiplexer is grossly unfair, in the sense that it has no time to accept a message from one of the fragmentation processes because it is continuously busy accepting incoming messages from other fragmentation processes on the same node. Since the input queue of the multiplexer is an order of magnitude faster than the CAN bus, there should be no situation where there is contention in getting into that queue.

One of the main dangers faced by the CAN bus and its surrounding protocols is a Denial of Service (DoS) attack. In our application (see Sect. 2), we protect the bus against DoS attacks by giving only trusted software access to a (secure) CAN bus. Unsecure components may also send messages to the bus, but those message first pass through a trusted gateway, which performs rate limiting.

The proof of this requirement will undoubtedly be the hardest part of the verification effort. It probably requires various intermediate results, such as '*every fragment sent by a fragmentation process is received by the reassembly process on each of its destination nodes*'.

Buffers Have a Maximal Length. Our formal specification employs message buffers in two places. One is (in) the CAN receiver; the other is the input queue of the multiplexer. Both buffers are modelled as FIFO queues. Following the specification both have unbounded capacity. In reality, buffers have a bounded capacity, and overflows will occur when trying to exceed it.

For the CAN receiver an upper bound cannot be given without a timing analysis comparing the capacity of the CAN bus and the 'working speed' of the reassembly protocol.

The input queue of the multiplexer, on the other hand, has a maximal length. To calculate this length, we add the maximal number of messages it could receive from any fragmentation process, times the number of fragmentation processes running on the node, plus the maximum number of message received from the associated CAN controller. For each type of message we calculate an upper bound.

The Application Layer Can Always Succeed in Submitting a New Message. A requirement mentioned above guarantees that, under certain conditions, messages submitted will eventually reach their destinations. As a liveness property for the application layer, this is only convincing if in addition the application layer can always succeed in submitting to the fragmentation protocol any message its want to transmit.

9 Conclusion and Future Work

In this paper we have presented a formal and unambiguous specification of a fragmentation and reassembly protocol, as well as a multiplexer. These protocols are running on top of a standard CAN bus and hence can directly be applied in many areas such as the automotive space. We have argued why both protocols are needed in real applications; and have shown applicability by running our protocols on a research quadcopter.

Last but not least we have listed a couple of important properties our protocol stack should satisfy. It is our belief that the presented properties do hold for our formally specified protocols, at least under some assumptions (as we have pointed out). Future work could provide formal proofs for these properties.

References

[1] T. Bourke, R.J. van Glabbeek & P. Höfner (2014): *A mechanized proof of loop freedom of the (untimed) AODV routing protocol*. In F. Cassez & J.-F. Raskin, editors: *Automated Technology for Verification and Analysis* (ATVA '14), LNCS 8837, Springer, pp. 47–63, doi:10.1007/978-3-319-11936-6_5.

[2] T. Bourke, R.J. van Glabbeek & P. Höfner (2016): *Mechanizing a Process Algebra for Network Protocols*. *Journal of Automated Reasoning* 56(3), pp. 309–341, doi:10.1007/s10817-015-9358-9.

[3] M.J. Dworkin (2007): *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D, National Institute of Standards and Technology (NIST), U.S. Department of Commerce, doi:10.6028/NIST.SP.800-38D.

[4] K. Etschberger (2001): *Controller Area Network: Basics, Protocols, Chips and Applications*. IXXAT Automation GmbH.

[5] A. Fehnker, R. J. van Glabbeek, P. Höfner, A. K. McIver, M. Portmann & W. L. Tan (2012): *A Process Algebra for Wireless Mesh Networks*. In H. Seidl, editor: *European Symposium on Programming* (ESOP '12), LNCS 7211, Springer, pp. 295–315, doi:10.1007/978-3-642-28869-2_15.

[6] R.J. van Glabbeek, P. Höfner, M. Portmann & W.L. Tan (2016): *Modelling and Verifying the AODV Routing Protocol*. *Distributed Computing* 29(4), pp. 279–315, doi:10.1007/s00446-015-0262-7.

[7] H. Krawczyk, M. Bellare & R. Canetti (1997): *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104 (Informational, Errata Exist). Available at `http://tools.ietf.org/html/rfc2104`.

[8] T. Nipkow, L. C. Paulson & M. Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer, doi:10.1007/3-540-45949-9.

[9] Robert Bosch GmbH, Stuttgart, Germany (1991): *CAN Specification*, Version 2.0. Available at `http://www.bosch-semiconductors.de/media/ubk_semiconductors/pdf_1/canliteratur/can2spec.pdf`.

[10] C. Shin (2014): *A framework for fragmenting/reconstituting data frame in Controller Area Network (CAN)*. In: *International Conference on Advanced Communication Technology* (ICACT '14), IEEE, pp. 1261–1264, doi:10.1109/ICACT.2014.6779161.

[11] C.M. Shin, T.M. Han, H.S. Ham & W.J. Lee (2010): *Method for Transmitting/Receiving Data Frame in CAN Protocol*. Available at `http://www.google.com/patents/US20100158045`. US Patent App. 12/543,876.

[12] C. Sommer & F. Dressler (2015): *Vehicular Networking*. Cambridge University Press, doi:10.1017/CBO9781107110649.

[13] International Organization for Standardization (2011): *Road Vehicles – Diagnostic Communication Over Controller Area Network (DoCAN) – Part 2: Transport protocol and network layer services*. ISO 15765-2:2011(en), ISO. Available at `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=54499`.

[14] S. Turner & L. Chen (2011): *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*. RFC 6151 (Informational). Available at `http://tools.ietf.org/html/rfc6151`.

A Informal Description of the Process Expressions of AWN

In this appendix we describe the *Process expressions*, given in Table 1.

A process name X comes with a *defining equation*

$$X(\mathtt{var}_1,\ldots,\mathtt{var}_n) \stackrel{def}{=} P\ ,$$

where P is a process expression, and the $\mathtt{var}_i$ are data variables maintained by process X. A named process is like a *procedure*; when it is called, data expressions exp_i of the appropriate type are filled in for the variables $\mathtt{var}_i$. Furthermore, φ is a condition, $\mathtt{var}:=exp$ an assignment of a data expression exp to a variable $\mathtt{var}$ of the same type, *dest*, *dests*, *data* and *ms* data expressions of types $\mathtt{ID}$, $\mathscr{P}(\mathtt{ID})$, $\mathtt{DATA}$ and $\mathtt{MSG}$, respectively, and $\mathtt{msg}$ a data variable of type $\mathtt{MSG}$.

Given a valuation of the data variables by concrete data values, the process $[\varphi]P$ acts as P if φ evaluates to $\mathtt{true}$, and deadlocks if φ evaluates to $\mathtt{false}$.[14] In case φ contains free variables that are not yet interpreted as data values, values are assigned to these variables in any way that satisfies φ, if possible. The process $[\![\mathtt{var}:=exp]\!]P$ acts as P, but under an updated valuation of the data variables. The process $P+Q$ may act either as P or as Q, depending on which of the two is able to act at all. In a context where both are able to act, it is not specified how the choice is made. The process $\mathbf{broadcast}(ms).P$ broadcasts (the data value bound to the expression) *ms* to all connected components, and subsequently acts as P, whereas the process $\mathbf{unicast}(dest,ms).P \blacktriangleright Q$ tries to unicast the message *ms* to the destination *dest*; if successful it continues to act as P and otherwise as Q. We abbreviate $\mathbf{unicast}(dest,ms).P \blacktriangleright P$ by $\mathbf{unicast}(dest,ms).P$; this covers the case where the subsequent behaviour after the unicast is independent of its success. The process $\mathbf{groupcast}(dests,ms).P$ tries to transmit *ms* to all destinations *dests*, and proceeds as P regardless of whether any of the transmissions is successful. The process $\mathbf{receive}(\mathtt{msg}).P$ receives any message m (a data value of type $\mathtt{MSG}$) either from another component, from another process running on the same component or from an application layer process connected to that component. It then proceeds as P, but with the data variable $\mathtt{msg}$ bound to the value m. In particular, $\mathbf{receive}(\mathtt{newpkt}(id, data))$ models the injection of data from the application layer, where the function $\mathtt{newpkt}$ generates a message containing the application layer *data* and the identifier *id*, here indicating the message type. Data is delivered to the application layer by $\mathbf{deliver}(data)$.

The internal state of a sequential process described by an expression P in this language is determined by P, together with a *valuation* ξ associating data values $\xi(\mathtt{var})$ to the data variables $\mathtt{var}$ maintained by this process. In case a process maintains no data values, we use the empty valuation ξ_0. A *valuated process* is a pair (ξ,P) of a sequential process P and an initial valuation ξ.

Finally, $V \langle\langle W$ denotes a parallel composition of valuated processes V and W, with information piped from right to left; in typical applications [6] W is a message queue. This yields a *component expression*.

In the full process algebra [5], *node expressions* $id:V:R$ are given by component expressions V, annotated with a component identifier *id* and a set of nodes R that are connected to *id*. In the current application we do not encounter cases were components send messages to other components that are not connected. As a consequence, the annotation $:R$ occurring in node expressions is of no significance, and omitted. We speak instead of *addressed* component expressions $id:V$. In our application V is for example a specification of a generic CAN driver, whereas $id:V$ denotes a specific CAN driver occurring in the system, such as the one on the mission board. The identifier *id* of this driver is used as an address by components that send messages to this driver.

[14] As operators we also allow *partial* functions with the convention that any atomic formula containing an undefined subterm evaluates to $\mathtt{false}$.

A network is modelled as a parallel composition of addressed component expressions, using the operator $\|$.

In our specification, we will use a process ERROR, representing a state that needs to be avoided at all costs. Showing that this state is in fact unreachable ought to be part of a verification effort. When specifying a part of a system (as we do in this report) we make assumptions on the behaviour of components outside our specification that communicate with our part. The ERROR state may be reachable if those external components violate the assumptions. Hence, showing that ERROR is unreachable involves verifying aspects of the correctness of these external components. Formally, we define ERROR as a process name with defining equation $\texttt{ERROR}() \stackrel{def}{=} \texttt{[false]ERROR}()$, representing a *deadlock*.

B Formal Specification of all Protocols

The following 6 sections will present formal specifications of all the above-mentioned protocols. This includes the specifications themselves, given in AWN, as well as a detailed definition of the data structure involved. As the semantics of AWN is completely unambiguous, specifying a protocol in such a framework enforces total precision and the removal of any ambiguity.

B.1 Data Structure: Mandatory Types and Messages

In this section we set out the basic data structure needed for the detailed formal specification of our fragmentation protocol. As well as describing *types* for the information handled at the nodes/components during the execution of the protocol we also define functions which will be used to describe the precise intention—and overall effect—of the various update mechanisms in our protocol.

B.1.1 Components

In our formalisation of the CAN bus we consider a finite set $\mathscr{H}$ of *hardware components*; in Fig. 1 these are the microcontroller, the mission board, and the camera. The defining characteristic of this set is that there is exactly one CAN driver $\mathtt{C}_H$ for each hardware component $H \in \mathscr{H}$. The COTS component does not count, as it only partakes to the trusted CAN bus via the gateway (cf. Fig. 3).

Furthermore, we consider a finite set $\mathscr{A}$ of *applications*. Each application is a party that sends messages via the CAN bus. An application is located on a hardware component, and on each hardware component can be multiple applications. Figure 4 shows two hardware components, each with two applications.

Finally, there is a finite set $\mathscr{S}$ of CAN *software components*—the white rectangles of Fig. 4. Each of them will be specified by an addressed component expression as defined in App. A. In our model a CAN driver is modelled as the parallel composition of two software components: one dealing with transmission, and one with receipt of CAN messages.

B.1.2 Mandatory Types

The process algebra AWN always requires the following data structure: application layer data, messages, component identifiers and sets of component identifiers.

1. The ultimate purpose any communication protocol is to deliver *application layer data*. The type `DATA` describes a set of application layer data items. An item of (encrypted) data is thus a particular element of that set, denoted by the variables $\mathtt{data}, \mathtt{ndata} \in \mathtt{DATA}$. Since we also inform the application layer about progress, we add special strings such as "message successfully sent" to the set `DATA`. Moreover, the empty data string is denoted by $\varepsilon \in \mathtt{DATA}$.
2. *Messages* are used to send information via the network. In our specification we use the variable `msg` of the type `MSG`. All message types will be described below.
3. The type `ID` describes a set of identifiers. In our application it is the disjoint union of a set `AID` of *application identifiers* (exactly one for each application from $\mathscr{A}$), a set `SID` of *component identifiers* (exactly one for each software component from $\mathscr{S}$) and a set `CID` of CAN IDs. Moreover, we use a set `MT` of *message types*, which we assume to be a subset of `CID`. For each hardware component $H \in \mathscr{H}$, the constants $\mathtt{M}_H$, $\mathtt{R}_H$ and $\mathtt{C}_H$ of type `SID` denote the identifiers of the unique multiplexer, reassembly protocol, and transmitting CAN driver within H. The variable `aid` ranges over `AID` and indicates the (ultimate) sender of a message—an application. The variables `cid`, `bid` range over

CID; and mt and nmt over MT. Finally we make use of a special element $\perp_{\mathtt{mt}} \in \mathtt{MT}$, denoting the absence of a message.

A message sent over the CAN bus is normally only a fragment of a larger message (stemming from the application layer), although we allow for CAN messages that are not fragmented. Henceforth, we use the word *fragment* for such a message. Message types are allocated to entire messages, whereas CAN IDs are allocated to fragments. A CAN ID determines the message type of the whole message, as well as the fragment counter indicating which fragment of it is currently been transmitted. A message type uniquely determines the sender of the message, the set of recipients, and the number of fragments into which the message is split. Here we take as CID an initial segment of the natural numbers. In our implementation, 11 bits are reserved for CAN IDs. Given a message type that calls for fragmentation into 3 parts, the CAN IDs form an interval such as 52–54. The message type is then simply denoted by the first element of this interval; in the example 52. It is in this sense that $\mathtt{MT} \subseteq \mathtt{CID}$. We use an injective partial function

$$\mathtt{canid} : \mathtt{MT} \times \mathbb{N} \rightharpoonup \mathtt{CID}$$

that, given a message type *mt* and a fragment counter k, which is no larger than the number of fragments for messages of type *mt*, returns a CAN ID. In our example, $\mathtt{canid}(52,2) = 53$. Due to injectivity, if $\mathtt{canid}(\mathtt{mt},\mathtt{no})$ is defined, the values mt and no can be retrieved. For the implementation of the protocol, this function is implemented as a static table, stored at each component. Sometimes it is possible to reduce the size of this table since only those messages types have to be specified that are actually sent or received by the fragmentation and reassembly protocols.

B.1.3 Messages

Messages are the main ingredient of all our protocols and are used to distribute information. The message types used range from new messages to be split and injected by the application layer, via messages to acknowledge successful sending, to actual CAN messages that are sent via the CAN bus. To generate theses messages, we use functions

$$\begin{aligned}
\mathtt{newpkt} &: \mathtt{MT} \times \mathtt{DATA} \to \mathtt{MSG}\,,\\
\mathtt{pkt} &: \mathtt{MT} \times \mathtt{DATA} \to \mathtt{DATA}\,,\\
\mathtt{can} &: \mathtt{CID} \times \mathtt{DATA} \to \mathtt{MSG}\,,\\
\mathtt{cancel} &: \mathtt{MSG}\,,\\
\mathtt{cancel} &: \mathtt{CID} \to \mathtt{MSG}\,,\\
\mathtt{ack} &: \mathbb{B} \to \mathtt{MSG}\,, \text{and}\\
\mathtt{msgd} &: \mathtt{TX} \times \mathtt{MSG} \to \mathtt{MSG}\,.
\end{aligned}$$

A message $\mathtt{newpkt}(mt\,,d)$ is of type *mt* and has the payload d. It is injected by the application and received by the fragmentation protocol, which, if necessary, then splits the message into smaller fragments.

A reassembled message which is returned to an application by the reassembly protocol is given by $\mathtt{pkt}(mt\,,d)$, where *mt* and d are again the message type and the actual data, respectively.[15]

The function $\mathtt{can}(cid\,,d)$ generates a CAN message with CAN ID *cid*, containing the data d. By $\mathtt{can}(\mathtt{canid}(mt,k)\,,d)$ a fragmented CAN message is obtained, containing the data d and the CAN ID

[15] pkt does not generate a message, but is of type DATA. The reason is that, in AWN, messages are used to send information to software components. Since we do not model the application, pkt is a message delivered to the environment, which has to be of type DATA.

$\texttt{canid}(mt,k)$. This message is the k^{th} fragment of a message of type *mt*; we abstract from all other details of such a CAN message.

The functions $\texttt{cancel}()$ and $\texttt{cancel}(cid)$ are used to request the cancellation of a message sent before. The only difference between these two are that the former tries to cancel the last message sent, whereas the latter requests the cancellation of a CAN message with CAN ID *cid*.

The acknowledgement message generated by $\texttt{ack}(b)$ is used to communicate the status of a message. $b \in \mathbb{B} = \{\texttt{true},\texttt{false}\}$ is a Boolean value: $\texttt{ack}(\texttt{true})$ indicates that the message or fragment was sent successfully, whereas $\texttt{ack}(\texttt{false})$ indicates failure, which either stems from a sending problem (e.g. a hardware failure) or from the message being successfully cancelled.

The last function we use to generate messages is `msgd`. It is essentially a wrapper function that takes an arbitrary message as input and returns the same message with an additional variable attached— the identifier of a transmit (TX) buffer (an element of the set `TX` of TX buffer identifiers).[16]

We assume that all functions building messages are injective, so that for example the values *mt* and *data* can be retrieved from $\texttt{newpkt}(mt,data)$. Likewise, we can distil the message type *mt*, the fragment number *no* and the payload *d* from $\texttt{can}(\texttt{canid}(mt,no),d)$.

B.1.4 Message Type Table

Information about the sender, the potential destinations, the length, and the identifier of the corresponding instance of the fragmentation protocol for each and every CAN message are stored in a static table. All components have access to a copy of this table.[17] Such an information table is defined as a set of entries, exactly one for each message type.

Formally, we define the identifier table as a (total) function

$$\texttt{id\_tab} : \texttt{CID} \to \texttt{AID} \times \mathscr{P}(\texttt{SID}) \times \mathbb{N} \times \texttt{SID}\ .$$

Here $cid \mapsto (aid, rids, parts, fid)$ identifies for every CAN ID *cid* (and hence for every message type) its (unique) sender $aid \in \texttt{AID}$, a list of receivers $rids \subseteq \texttt{SID}$, the number *parts* of CAN messages into which the (entire) message needs to be split, and the unique identifier *fid* of the fragmentation protocol handling the CAN message. Since we assume that the length of any message of a given type is fixed, this number can easily be determined. Note that `id_tab` is total; hence $\texttt{id\_tab}(cid)$ always returns a value. We use projections $\pi_1, \pi_2, \pi_3, \pi_4$ to select the corresponding component from the 4-tuple.

In the formal model (and indeed in any implementation) we need to extract information form `id_tab`. To this end, we define the following functions.

1. The sender of a message with CAN ID *cid*:

$$\begin{aligned} \texttt{sender} &: \texttt{CID} \to \texttt{AID} \\ \texttt{sender}(cid) &:= \pi_1(\texttt{id\_tab}(cid)) \end{aligned}$$

2. The set of potential (allowed) receivers of a message with CAN ID *cid*:

$$\begin{aligned} \texttt{rec} &: \texttt{CID} \to \mathscr{P}(\texttt{SID}) \\ \texttt{rec}(cid) &:= \pi_2(\texttt{id\_tab}(cid)) \end{aligned}$$

[16] A more detailed description of the TX buffers is given in Sect. B.5.1.

[17] Of course, when implementing the protocol, each node would only store the bits of the table that are actually needed by the node.

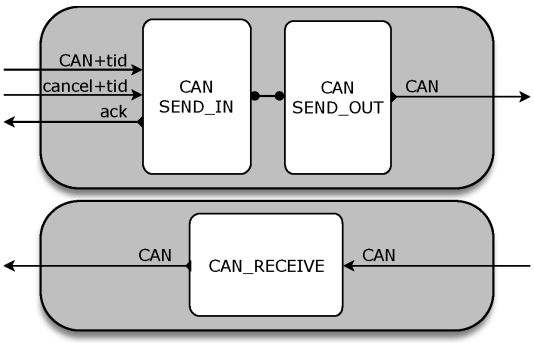

Figure 5: Structure of the CAN driver

3. The number of CAN messages into which a message with CAN ID *cid* is split:

$$\begin{aligned} \texttt{fragments} &: \texttt{CID} \to \mathbb{N} \\ \texttt{fragments}(cid) &:= \pi_3(\texttt{id\_tab}(cid)) \end{aligned}$$

   In case no splitting is needed, e.g., if the message is a legacy message, and the message should be handled as standard CAN message, `fragments` should be set to 1. This guarantees backwards compatibility of our protocol.
4. The name of the instance of the fragmentation protocol responsible for handling the message:

$$\begin{aligned} \texttt{frag} &: \texttt{CID} \to \texttt{SID} \\ \texttt{frag}(cid) &:= \pi_4(\texttt{id\_tab}(cid)) \end{aligned}$$

The value $\texttt{id\_tab}(cid)$ only depends on the message type of a message with CAN ID *cid*; hence $\texttt{id\_tab}(mt) = \texttt{id\_tab}(\texttt{canid}(mt,k))$ for every $k \leq \texttt{fragments}(mt)$.

B.2 The CAN Driver

In this section, we present a formal specification of the CAN driver, using the process algebra AWN. The presented driver is pretty simple and puts messages received straight into the corresponding TX buffer. In particular it does not provide a message queue to store messages until they are handled. Such a queue is not needed since it is part of the multiplexer, which we will present later.

The overall structure of the CAN driver—sketched in Fig. 5—consists of two independent components that do not interact. The first handles message sending, the second message receipt.

- The protocol that handles message sending is split into two processes: the first process, called $\texttt{CAN\_SEND\_IN}_H$ is able to receive messages from the multiplexer. These messages are either CAN messages or cancellation messages—both contain a TX-buffer identifier *tid*. If a CAN message is received, it is stored in the corresponding TX buffer, and the process $\texttt{CAN\_SEND\_OUT}_H$ is called, which subsumes the behaviour of $\texttt{CAN\_SEND\_IN}_H$ and also handles the message sending. If a `cancel`-message is received, the process erases the corresponding TX buffer.
- The protocol that handles message receipt is a single process $\texttt{CAN\_RECEIVE}_H$. It models a simple queue, which stores all incoming messages and forwards them to the reassembly protocol as soon as that protocol is ready to handle the next message.

All these processes are parametrised with the name $H \in \mathscr{H}$ of the hardware component on which the CAN driver is located.

B.2.1 Data Structure

Each CAN controller provides a number of transmit (TX) buffers. Each buffer is able to store a complete CAN message for transmission over the CAN bus. Our architecture offers 3 TX buffers on the microcontroller, and one on the mission board; both boards offer 2 receive (RX) buffers.[18] We assume that every TX buffer on a board has a unique identifier—the set of all identifier is $\mathtt{TX}$.

We abstract from the concrete number of buffers, and model the buffers as a (total) function $\mathtt{buffer}$ of type $\mathtt{TX} \to \mathtt{MSG}$. If the TX buffer identified by *tid* stores a message *msg*, then $\mathtt{buffer}(tid) = msg$; to indicate that the buffer is empty, we use the special element $\bot_{\mathtt{msg}} \in \mathtt{MSG}$. We define the function space of all these functions as

$$\mathtt{BUFFER} \triangleq \mathtt{TX} \ \to \ \mathtt{MSG}\ .$$

In our formal specifications we often use an assignment to change a particular value of a function, e.g. $[\![\mathtt{buffer(tid)} := \mathtt{can(cid,data)}]\!]$ (Line 4 of Process 1). Implicitly this states that all other values stay the same.

The CAN driver is supposed to send the message of highest priority (with lowest CAN ID) next. To this end we define a partial function

$$\mathtt{best} : \mathtt{BUFFER} \ \rightharpoonup \ \mathtt{TX}$$

that determines the TX buffer that contains this CAN message that is most urgent; in case there are different messages with the same priority it chooses non-deterministically. We claim that the function $\mathtt{best}$ is actually deterministic in our setting. Formally the function is required to satisfy

$$\begin{array}{ll} & \mathtt{best}(\mathit{buffer}) = \mathit{tid} \\ \Leftrightarrow & \exists \mathit{cid}, d : \big(\mathit{buffer}(\mathit{tid}) = \mathtt{can}(\mathit{cid}, d)\big) \wedge \\ & \qquad \big(\exists \mathit{tid}', \mathit{cid}', d' : \mathit{buffer}(\mathit{tid}') = \mathtt{can}(\mathit{cid}', d') \Rightarrow \mathit{cid} \leq \mathit{cid}'\big)\big)\ . \end{array}$$

Note that $\mathtt{best}(\mathit{buffer})$ is undefined if and only if all TX buffers are empty, i.e., if $\mathit{buffer}(\mathit{tid}) = \bot_{\mathtt{msg}}$ for all $\mathit{tid} \in \mathtt{TX}$.

We use a queue-style data structure for modelling an inbox of the CAN receiver. In general, we denote queues of messages by $[\mathtt{MSG}]$, denote the empty queue by $[\,]$, and make use of the standard (partial) functions $\mathtt{head} : [\mathtt{MSG}] \rightharpoonup \mathtt{MSG}$, $\mathtt{tail} : [\mathtt{MSG}] \rightharpoonup [\mathtt{MSG}]$ and $\mathtt{append} : \mathtt{MSG} \times [\mathtt{MSG}] \to [\mathtt{MSG}]$ that return the "oldest" element in the queue, remove the "oldest" element, and add a packet to the queue, respectively.

In our protocol specification below, all messages received via the CAN bus are stored until the reassembly protocol is ready to handle them. This may not be needed in the implementation of this protocol, as a timing analysis may show that when a message arrives, the protocol is always ready to handle it. However, in the forthcoming verification of the correctness of our protocol we do not want to depend on this timing analysis, and hence incorporate incoming message queues. Thus, a separate verification—involving the timing analysis—will be needed to show that these queues can be omitted.

At the moment we assume an infinite queue, which is unrealistic. It is an easy task to model a finite queue, where messages are lost in case the buffer is full. Of course then properties such as "every message sent will be received" may not hold; they require carefully designed preconditions.

This section concludes with a table summarising the entire data structure we use for the CAN driver. It summarises not only the data structure presented in this section, but also recapitulates the necessary part of the structure presented in Sect. B.1. Similar tables will be given at the end of every section that discusses data structure (cf. Sections B.3.1, B.4.1 and B.5.1).

[18] The second RX buffer, however, should not be used due to a hardware bug.

Basic Type	**Variables**	**Description**
MSG	msg	messages
DATA	data	data/payload of messages
TX	tid	identifiers for TX buffers
CID	cid	CAN IDs
SID		CAN software component identifiers
Complex Type	**Variables**	**Description**
$\texttt{BUFFER} \triangleq \texttt{TX} \to \texttt{MSG}$	buffer	array of (CAN) messages, modelling the TX buffers
[MSG]	msgs	message queues
Constant		**Description**
$\bot_{\texttt{msg}} : \texttt{MSG}$		special message symbol (indicating absence of a message)
$[] : [\texttt{MSG}]$		empty queue
$\texttt{M}_H : \texttt{SID}$		multiplexer identifier for hardware component H
$\texttt{R}_H : \texttt{SID}$		reassembling protocol identifier for hardware component H
$\texttt{C}_H : \texttt{SID}$		transmitting CAN driver identifier for hardware component H
$\texttt{D}_H : \texttt{SID}$		receiving CAN driver identifier for hardware component H
Function		**Description**
$\texttt{cancel} : \texttt{MSG}$		cancellation message
$\texttt{ack} : \mathbb{B} \to \texttt{MSG}$		acknowledgement to multiplexer
$\texttt{can} : \texttt{CID} \times \texttt{DATA} \to \texttt{MSG}$		create CAN messages out of identifier and payload
$\texttt{msgd} : \texttt{TX} \times \texttt{MSG} \to \texttt{MSG}$		wrapper function to add a TX-identifier to a message
$\texttt{rec} : \texttt{CID} \to \mathscr{P}(\texttt{SID})$		set of potential (allowed) receivers of a message
$\texttt{best} : \texttt{BUFFER} \rightharpoonup \texttt{TX}$		returns the name of the TX that contains the 'best' CAN message
$\texttt{head} : [\texttt{MSG}] \rightharpoonup \texttt{MSG}$		returns the 'oldest' element in the queue
$\texttt{tail} : [\texttt{MSG}] \rightharpoonup [\texttt{MSG}]$		removes the 'oldest' element in the queue
$\texttt{append} : \texttt{MSG} \times [\texttt{MSG}] \to [\texttt{MSG}]$		inserts a new element into the queue

Table 2: Data structure for the CAN Controller/Driver

B.2.2 Formal Specification

Sending Messages. The sending procedure consists of two different processes. The first solely deals with receiving messages (from the multiplexer); and the second also with sending messages on the CAN bus. The second process subsumes the behaviour of the first process by offering it as a alternative to sending. The first process is only called in the initial state of the protocol, and as a potential behaviour of the second process. Both processes maintain a single variable `buffer`, which models the corresponding TX buffer (see above).

The first process, named $\texttt{CAN\_SEND\_IN}_H$ and depicted in Process 1, starts with receiving a message `msg` from the connected multiplexer $\texttt{M}_H$ (Line 1). After that the process checks the type of the message received:

In case the message is a CAN message with an additional value `tid`, which indicates the destined TX buffer, the process stores the message into the TX buffer `tid` (Line 4) and calls the process $\texttt{CAN\_SEND\_OUT}_H$, which handles the sending of CAN messages, as well as the processing of further messages from the multiplexer (by calling $\texttt{CAN\_SEND\_IN}_H$). Note that the new message is copied into the TX buffer regardless whether the buffer already contains a message. If it does, that previous message is lost. It is up to the multiplexer (Sect. B.5) to avoid such a scenario.

In case the process receives a cancellation request for TX buffer `tid` (the incoming message has the form $\texttt{msgd}(\texttt{tid}, \texttt{cancel}())$, Line 6), depending on the status of the corresponding TX buffer, differ-

Process 1 CAN driver—Sending Routine I

```
CAN_SEND_IN_H(buffer) =def
 1. receive(msg) .
 2. (
 3.    [ msg = msgd(tid,can(cid,data)) ]      /* new CAN message for TX buffer tid */
 4.       ⟦buffer(tid) := can(cid,data)⟧      /* override TX buffer */
 5.       CAN_SEND_OUT_H(buffer)
 6.    + [ msg = msgd(tid,cancel()) ]         /* cancellation message for TX buffer tid */
 7.       (
 8.          [ buffer(tid) = ⊥_msg ]         /* TX buffer tid is already cleared */
 9.             CAN_SEND_OUT_H(buffer)
10.          + [ buffer(tid) ≠ ⊥_msg ]       /* TX buffer tid contains a message */
11.             ⟦buffer(tid) := ⊥_msg⟧       /* erase TX buffer */
12.             unicast(M_H ,msgd(tid,ack(false))) .
13.             CAN_SEND_OUT_H(buffer)
14.       )
15. )
```

ent actions are performed. If buffer `tid` is empty (Line 8) there is no message to be cancelled and the protocol performs no action; it calls the process $\texttt{CAN\_SEND\_OUT}_H$ to proceed. If the buffer is not empty (Line 10) the process erases the buffer. After that the multiplexer is informed about the successful cancellation, which is done by unicasting the message $\texttt{msgd}(\texttt{tid},\texttt{ack}(\texttt{false}))$ to the connected multiplexer $\texttt{M}_H$. Since our unicast is blocking (the protocol is stuck if the recipient of the message is not ready to receive the message), there could be a possibility that the protocol deadlocks while trying to send the acknowledgement. To prevent this, we make sure that the multiplexer is *input enabled*, meaning that the multiplexer is always ready to receive messages (cf. Sect. B.5).

In case another type of message would be received the protocol deadlocks after message receipt. However, we can show that no other message type can be received and hence the protocol is free of deadlocks.

The main purpose of the second process, named $\texttt{CAN\_SEND\_OUT}_H$ and depicted in Process 2, is to send the CAN messages stored in the TX buffers. It also offers the behaviour of $\texttt{CAN\_SEND\_IN}_H$, to process another incoming message as an alternative to any activity that could block further progress.

Process 2 CAN driver—Sending Routine II

```
CAN_SEND_OUT_H(buffer) =def
 1.    CAN_SEND_IN_H(buffer)      /* another message incoming */
 2. + [ tid = best(buffer) ∧ buffer(tid) = can(cid,data) ]   /* messages in TX buffers to be sent */
 3.    (
 4.       CAN_SEND_IN_H(buffer)
 5.       +
 6.       groupcast(rec(cid),can(cid,data)) .
 7.       ⟦buffer(tid) := ⊥_msg⟧
 8.       unicast(M_H ,msgd(tid,ack(true))) .
 9.       CAN_SEND_OUT_H(buffer)
10.    )
```

If there is another incoming message, the process may run $\texttt{CAN\_SEND\_IN}_H$ (Line 1); otherwise $\texttt{CAN\_SEND\_OUT}_H$ determines the CAN message that should be send next (if any). This is done by use of the function `best` in Line 2. Since `best` returns the name of a TX buffer that contains a CAN mes-

sage[19] the second conjoint $\texttt{buffer}(\texttt{tid}) = \texttt{can}(\texttt{cid},\texttt{data})$ is not a restriction—it is used to distil the CAN ID cid of that message.

In Line 6 the process sends the CAN message $\texttt{can}(\texttt{cid},\texttt{data})$ with highest priority to the intended recipients of this message, which are determined by the function rec. This transmission uses the CAN bus and is carried out by the CAN controller. It will succeed only when there are no CAN messages with higher priority sent by other CAN controllers on the bus. As long as the transmission is pending, the process has the option to process another incoming message by executing $\texttt{CAN\_SEND\_IN}_H$ (Line 4). After the message has been successfully sent via the CAN bus, the TX buffer is erased (Line 7) and the connected multiplexer is informed about success by an ack-message (Line 8). As remarked before, the multiplexer is always ready to accept this acknowledgement. Finally, the process returns to $\texttt{CAN\_SEND\_OUT}_H$ to either accept another incoming message from the multiplexer or to transmit another buffered message on the CAN bus.

Line 4 guarantees that the process can receive another message (via Line 1 of Process 1). Without this line Process 2 can give rise to stagnation (deadlock), since the **groupcast**-action of Line 6 can only succeed if the CAN bus is not busy sending higher-priority messages from other nodes.

Receiving Messages. We assume that any message sent via the CAN bus is actually received by all CAN controllers/drivers that are supposed to receive the message. For this reason, a CAN driver should always be able to perform a receive action, regardless of which state it is in. We introduce a process $\texttt{CAN\_RECEIVE}_H$ (see Process 3), modelling a message queue, that runs in parallel with $\texttt{CAN\_SEND\_IN}_H$/ $\texttt{CAN\_SEND\_OUT}_H$. Every incoming message is stored in this queue, and piped from there to the reassembly protocol $\texttt{RASS}_H$, which we will discuss later in Sect. B.4, whenever $\texttt{RASS}_H$ is ready to handle a new message. The process $\texttt{CAN\_RECEIVE}_H$ is always ready to receive a new message.

Similar to the process call in Process 2 the **receive**-action in Line 8 is needed to guarantee that messages can be received at any time.

Process 3 CAN driver—Receiving Routine

$\texttt{CAN\_RECEIVE}_H(\texttt{msgs}) \stackrel{def}{=}$

1. /* store incoming message at the end of msgs */
2. **receive**(msg) . $\texttt{CAN\_RECEIVE}_H(\texttt{append}(\texttt{msg},\texttt{msgs}))$
3. $+$ **[** $\texttt{msgs} \neq [\,]$ **]** /* the queue is not empty */
4. (
5. /* pop top message and send it to the reassembly protocol */
6. **unicast**($\texttt{R}_H$,$\texttt{head}(\texttt{msgs})$) . $\texttt{CAN\_RECEIVE}_H(\texttt{tail}(\texttt{msgs}))$
7. /* or receive and store an incoming message */
8. $+$ **receive**(msg) . $\texttt{CAN\_RECEIVE}_H(\texttt{append}(\texttt{msg},\texttt{msgs}))$
9.)

B.2.3 Initialisation

To finish our specification, we have to define an initial state for the CAN driver. The initial state requires the assignment of any variable occurring in a process. Such an assignment is provided by *valuation functions*, offered by AWN (cf. App. A).

The initial process C_H of the CAN driver on hardware component $H \in \mathscr{H}$ is given by the expression

$$\texttt{C}_H{:}(\xi, \texttt{CAN\_SEND\_IN}_H(\texttt{buffer})) \,\|\, \texttt{D}_H{:}(\zeta, \texttt{CAN\_RECEIVE}_H(\texttt{msgs}))\ ,$$

[19] In fact an invariant we might need to show is that all TX buffers only contain CAN messages.

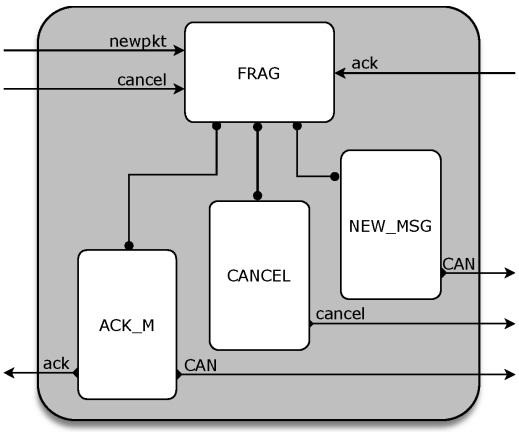

Figure 6: Structure of the Fragmentation Protocol

with $\mathtt{C}_H$ and $\mathtt{D}_H$ the component identifiers of the sending and receiving CAN components, and with

$$\xi(\mathtt{buffer}(\mathtt{tid})) = \bot_{\mathtt{msg}} \quad (\forall \mathtt{tid} \in \mathtt{TX}) \ \wedge \ \zeta(\mathtt{msgs}) = [\,] \,.$$

This says that initially all TX buffers controlled by a CAN driver are empty, and the message queue of CAN messages received by a CAN receiver is empty.

B.3 Fragmentation Protocol

In this section, we present a formal specification of the fragmentation protocol. Our model, which is sketched in Fig. 6, consists of 4 processes, named $\mathtt{FRAG}_H$, $\mathtt{NEW\_MSG}_H$, $\mathtt{CANCEL}_H$ and $\mathtt{ACK\_M}_H$:

- The basic process $\mathtt{FRAG}_H$ receives a message from the application layer or from the multiplexer and, depending on the type of the message, calls other processes. When there is no message handling going on, it idles until a new message arrives.
- The process $\mathtt{NEW\_MSG}_H$ describes all actions performed by the fragmentation protocol when a new message is received from the application. This includes the possibility of reaching an unrecoverable error state in case the application injects a message before a previous message was successfully handled or cancelled. The application layer should be programmed in such a way that this error state is always avoided.
- The process $\mathtt{CANCEL}_H$ handles all actions to be performed when an application request the cancellation of a message previously sent.
- The process $\mathtt{ACK\_M}_H$ describes the protocol behaviour in case an acknowledgement message (positive or negative) is received from the multiplexer. Depending on the situation, this process reports the status to the application, sends the next fragment, or reaches an unrecoverable error state.

B.3.1 Data Structure

The main intention of the protocol is to split data given from the application layer. For this purpose we define functions to manipulate data. Since all these functions can be seen as bit-level operations, we are not giving the exact definitions.

The function $\mathtt{head}_8 : \mathtt{DATA} \to \mathtt{DATA}$ extracts the first 8 bytes from a given `data`; or returns the entire data in case it is shorter.

The function $\texttt{tail}_8 : \texttt{DATA} \to \texttt{DATA}$ is the complement of $\texttt{head}_8$ and returns the remaining data (if any) after 8 bytes have been chopped off.

The following table summarises the entire data structure we use for the fragmentation protocol.

Basic Type	**Variables**	**Description**
MSG	msg	messages
DATA	data, ndata	stored data
$\mathbb{N}$	no	fragment counter
$\mathbb{B}$	abort, suc	Boolean flags
MT	mt, nmt	message types
CID		CAN IDs
SID		CAN software component identifiers
Constant		**Description**
$\varepsilon : \texttt{DATA}$		the empty data string
$\bot_{\texttt{mt}} : \texttt{MT}$		special message type symbol, denoting undefined message type
$\texttt{M}_H : \texttt{SID}$		multiplexer identifier for hardware component H
Function		**Description**
$\texttt{newpkt} : \texttt{MT} \times \texttt{DATA} \to \texttt{MSG}$		creates application-layer message out of message type and data
$\texttt{cancel} : \texttt{MSG}$		cancellation message from application layer
$\texttt{cancel} : \texttt{CID} \to \texttt{MSG}$		cancellation message to multiplexer
$\texttt{ack} : \mathbb{B} \to \texttt{MSG}$		acknowledgement from multiplexer
$\texttt{can} : \texttt{CID} \times \texttt{DATA} \to \texttt{MSG}$		create CAN messages out of identifier and payload
$\texttt{canid} : \texttt{MT} \times \mathbb{N} \rightharpoonup \texttt{CID}$		returns a CAN ID for a given message type and a fragment counter
$\texttt{head}_8 : \texttt{DATA} \to \texttt{DATA}$		takes the first 8 bytes from a given data streams
$\texttt{tail}_8 : \texttt{DATA} \to \texttt{DATA}$		removes the first 8 bytes from a given data streams
$\texttt{fragments} : \texttt{CID} \to \mathbb{N}$		number of CAN messages into which a message is split

Table 3: Data structure for the Fragmentation Protocol

B.3.2 Formal Specification

The Main Loop. The basic process $\texttt{FRAG}_H$ receives messages from the application layer or the multiplexer; one at a time. This process maintains four data variables mt, data, no, and abort, in which it stores the message type last handled, the data received from the application and not yet fragmented, the number of fragments already sent, and a flag to signal that the application requested cancellation. The process is considered to be currently handling an application-layer message iff the value of no is non-zero.

Process 4 Fragmentation—Main Loop

$\texttt{FRAG}_H(\texttt{mt},\texttt{data},\texttt{no},\texttt{abort}) \stackrel{def}{=}$

1. **receive**(msg) .
2. (
3. [msg = newpkt(nmt,ndata)] /* new message to be sent; distill nmt and ndata */
4. $\texttt{NEW\_MSG}_H(\texttt{nmt},\texttt{ndata},\texttt{no},\texttt{abort})$
5. + [msg = cancel()] /* cancellation message received */
6. $\texttt{CANCEL}_H(\texttt{mt},\texttt{data},\texttt{no},\texttt{abort})$
7. + [msg = ack(suc)] /* message from multiplexer */
8. $\texttt{ACK\_M}_H(\texttt{suc},\texttt{mt},\texttt{data},\texttt{no},\texttt{abort})$
9.)

The routine is formalised in Process 4. First, a message has to be received using the command **receive**(msg) (Line 1). The message stems either from the application or from the multiplexer (cf. Sect. 5). After that, the process $\texttt{FRAG}_H$ checks the type of the message and calls a process that can handle this message: in case of a 'fresh' message from the application layer, the process $\texttt{NEW\_MSG}_H$ is called (Line 4)—note that during the process call the values of mt and data are updated with the new values nmt and ndata extracted from the message; in case of an incoming cancellation request the process $\texttt{CANCEL}_H$ is executed (Line 6); and in case a message from the multiplexer (ack) is read, the process $\texttt{ACK\_M}_H$ is called, carrying the Boolean flag suc received from the multiplexer. In case a message of any other type is received, the process $\texttt{FRAG}_H$ deadlocks; it is a proof obligation to check that this will not occur.

New Message. The process $\texttt{NEW\_MSG}_H$ describes all actions performed by the fragmentation protocol when a new message is injected by an application hooked up to the instance of the fragmentation protocol.

Process 5 New Message Received

$\texttt{NEW\_MSG}_H(\texttt{mt},\texttt{data},\texttt{no},\texttt{abort}) \stackrel{def}{=}$

1. [$\texttt{no} \neq 0$] /* protocol is sending another message */
2. ERROR /* Enter error state and die */
3. + [$\texttt{no} = 0$] /* protocol ready to start handling new message */
4. **unicast**($\texttt{M}_H$, $\texttt{can}(\texttt{canid}(\texttt{mt},1),\texttt{head}_8(\texttt{data}))$) . $\texttt{FRAG}_H(\texttt{mt},\texttt{tail}_8(\texttt{data}),1,\texttt{abort})$

In case the fragmentation protocol is in the process of sending (fragmenting) a message, which was previously received, the value of no is different to 0, since this integer counts the number of fragments already sent. If this is the case (Line 1), the process deadlocks with an error; the entire fragmentation stops and cannot be restored. It is up to the application hooked up to the fragmentation protocol to avoid this scenario.

In case the fragmentation protocol is not handling another message ($\texttt{no} = 0$), the process $\texttt{NEW\_MSG}_H$ starts splitting the data: it chops off the first 8 bytes of data, using the function $\texttt{head}_8$, and then creates a CAN message, which is sent to the connected multiplexer $\texttt{M}_H$. The corresponding CAN ID is determined by $\texttt{canid}(\texttt{mt},1)$—the first fragment of message type mt is being handled. Before the second fragment can be sent to the multiplexer, the fragmentation protocol has to await an acknowledgement of the multiplexer (which actually stems from the CAN driver and is only forwarded by the multiplexer). Hence the process returns to $\texttt{FRAG}_H$, where it can receive a new message, while updating the fragment counter no and the remaining data to be fragmented.

Cancellation Message. An application always has the possibility to cancel the message previously sent. Since one of our assumptions is that the application layer only submits one message a time (otherwise the fragmentation protocol yields a deadlock—see Line 2 of Process 5), only the last message can be cancelled.

Process 6 Cancellation Request

$\texttt{CANCEL}_H(\texttt{mt},\texttt{data},\texttt{no},\texttt{abort}) \stackrel{def}{=}$

1. [$\texttt{no} \neq 0$] /* protocol is sending a message */
2. **unicast**($\texttt{M}_H$, $\texttt{cancel}(\texttt{canid}(\texttt{mt},\texttt{no}))$) . $\texttt{FRAG}_H(\texttt{mt},\texttt{data},\texttt{no},\texttt{true})$
3. + [$\texttt{no} = 0$] /* protocol is not sending a message */
4. $\texttt{FRAG}_H(\texttt{mt},\texttt{data},\texttt{no},\texttt{abort})$ /* nothing to cancel */

Similar to Process 5 the cancellation process checks whether the fragmentation protocol is currently sending/splitting a message; as before this check is done by evaluating the statement $\texttt{no} = 0$. In case

the protocol is currently working on splitting a message, the cancellation request is forwarded to the multiplexer $\mathtt{M}_H$ (Line 2); it is the multiplexer who can stop the current fragment to be sent. Since the multiplexer is connected to multiple instances of the fragmentation protocol, the cancellation message must now carry the unique CAN ID of the message to be cancelled. The fragmentation protocol itself just stops splitting the `data` further; it signals the cancellation process by setting the flag `abort` to `true`. The protocol does not yet set the value of `no` to 0—indicating that it is idle again. This is only done after an acknowledgement from the multiplexer has been received. In case the fragmentation protocol is idle and receives a cancellation message—this can happen in case that the cancellation message was sent by the application just before the acknowledgement of successful transmission was sent to the application—the cancellation request is ignored and the protocol returns to the Process $\mathtt{FRAG}_H$ (Line 4).

Notification from the Multiplexer. Independent of whether a transmission was successful or not, the CAN controller will send an acknowledgement via the multiplexer: in case of a successful transmission the contents of this message is '`true`', in case of a failure, or a cancellation, the value '`false`' is transmitted—the value is stored temporarily in the variable `suc`.

Process 7 Acknowledgement from the Multiplexer

$\mathtt{ACK\_M}_H(\mathtt{suc},\mathtt{mt},\mathtt{data},\mathtt{no},\mathtt{abort}) \stackrel{def}{=}$

1. $[\,\mathtt{no} \neq 0\,]$ /* process sending */
2. (
3. $[\,\mathtt{suc} = \mathtt{true}\,]$ /* positive acknowledgment received */
4. (
5. $[\,\mathtt{no} = \mathtt{fragments}(\mathtt{mt})\,]$ /* last fragment sent successfully */
6. **deliver**("message successfully sent") . $\mathtt{FRAG}_H(\mathtt{mt},\varepsilon,0,\mathtt{false})$
7. $+\,[\,\mathtt{no} \neq \mathtt{fragments}(\mathtt{mt}) \wedge \mathtt{abort} = \mathtt{false}\,]$ /* send next fragment */
8. **unicast**$(\mathtt{M}_H, \mathtt{can}(\mathtt{canid}(\mathtt{mt},\mathtt{no}+1), \mathtt{head}_8(\mathtt{data})))$. $\mathtt{FRAG}_H(\mathtt{mt},\mathtt{tail}_8(\mathtt{data}),\mathtt{no}+1,\mathtt{false})$
9. $+\,[\,\mathtt{no} \neq \mathtt{fragments}(\mathtt{mt}) \wedge \mathtt{abort} = \mathtt{true}\,]$ /* message cancelled, no fragment to be sent */
10. **deliver**("message sending failed or aborted") . $\mathtt{FRAG}_H(\mathtt{mt},\varepsilon,0,\mathtt{false})$
11.)
12. $+\,[\,\mathtt{suc} = \mathtt{false}\,]$ /* sending failed or aborted */
13. **deliver**("message sending failed or aborted") . $\mathtt{FRAG}_H(\mathtt{mt},\varepsilon,0,\mathtt{false})$
14.)
15. $+\,[\,\mathtt{no} = 0\,]$ /* process is not sending */
16. (
17. $[\,\mathtt{suc} = \mathtt{true}\,]$ /* positive acknowledgment received */
18. ERROR
19. $+\,[\,\mathtt{suc} = \mathtt{false}\,]$ /* sending failed or aborted */
20. $\mathtt{FRAG}_H(\mathtt{mt},\mathtt{data},\mathtt{no},\mathtt{abort})$
21.)

As in the previous process, Process 7 only takes actions when $\mathtt{no} \neq 0$, meaning that the process is in a state where it handles a message. In case the process is not handling a message (Line 15) it either deadlocks with an error (in case of a positive acknowledgement—Lines 17–18), or ignores the message (Line 20). In the future we hope to show that both situations cannot occur. The asymmetric treatment is immaterial, and follows the implementation.

Lines 1–14 handle the case when the process receives an acknowledgement and is handling a message right now—the standard case. Lines 4–11 handle the case of a positive acknowledgement, indicating the transmission of the last fragment to the multiplexer—and thereby to the CAN controller—was successful. If this is the case, the fragmentation protocol uses the function `fragments` to determine whether the last fragment of the split message was sent. If this is the case (Line 5), the protocol informs the

application about the success (**deliver**("message successfully sent")), sets the values of `no` and `abort` to 0 and `false`, respectively (to indicate that the process is ready to receive a new message from the application), and returns to the main process $\texttt{FRAG}_H$. In case the message that is currently handled has not been sent entirely and was not aborted (no cancellation request received), the protocol chops the next 8 bytes of `data` to be sent and creates a new CAN message, which is passed on to the multiplexer $\texttt{M}_H$ (Line 8); the local data (`no` and `data`) are adapted accordingly. In case a positive acknowledgement is received, but the message was cancelled meanwhile (Line 9), the protocol informs the application layer and returns to an idle state, with `no` set to 0, the `data` being 'removed' and the Boolean flag `abort` set to `false` (Line 9). In case the fragmentation protocol receives a negative acknowledgement, the protocol behaves the same: it informs the application layer, and returns to an idle state.

B.3.3 Initialisation

The fragmentation protocol F_{id} is initialised by $id{:}(\xi, \texttt{FRAG}_H(\texttt{mt},\texttt{data},\texttt{no},\texttt{abort}))$, with *id* the identifier of any particular instance of the fragmentation protocol, and

$$\xi(\texttt{mt}) = \bot_{\texttt{mt}} \wedge \xi(\texttt{data}) = \varepsilon \wedge \xi(\texttt{no}) = 0 \wedge \xi(\texttt{abort}) = \texttt{false}\ .$$

This says that no message has been received, and no data is stored; moreover the protocol is neither sending nor aborting a message.

B.4 Reassembly Protocol

The reassembly protocol collects fragments of messages sent via the CAN bus. It stores and reassembles the messages; as soon as a message is fully received, the message is sent to the application layer. The protocol also checks whether some fragments are lost and drops partly assembled messages that cannot be completed. It can be implemented in AWN as a single process: $\texttt{RASS}_H$.

B.4.1 Data Structure

If a fragment of a message is received by a component, it may need to be stored until the full message can be recreated by reassembling the fragments. Since reordering does not happen on the CAN bus, and using assumption (7) in Sect. 3, it suffices to have a single data storage for every application as sender of the fragment.[20]

Hence, the data type of the store is defined as the function space

$$\texttt{STORE} \triangleq \texttt{AID} \ \rightarrow\ \texttt{MT} \times \mathbb{N} \times \texttt{DATA}\ .$$

For every sender (an application, with unique ID $aip \in \texttt{AID}$) a message type $mt \in \texttt{MT}$ and a number $k \in \mathbb{N}$ is stored, indicating the message type of the message received last and number of fragments received so far. Additionally, the function stores the concatenated data $d \in \texttt{DATA}$ from these fragments. In the formal model (and indeed in any implementation) we need to extract information form `store`. To this end, we define the functions[21] $\texttt{mtype} : \texttt{STORE} \times \texttt{AID} \rightarrow \texttt{MT}$, $\texttt{lastfrag} : \texttt{STORE} \times \texttt{AID} \rightarrow \mathbb{N}$, and

[20] This differs from an earlier specification and implementation. There we assumed a data storage for every message type; the new version improves memory storage dramatically.

[21] In contrast to `id_tab`, the function `store` will change during protocol execution. That is why we add it as argument to the extractions.

$\texttt{contents} : \texttt{STORE} \times \texttt{AID} \to \texttt{DATA}$ by

$$\begin{aligned} \texttt{mtype}(\mathit{store}, \mathit{aid}) &:= \pi_1(\mathit{store}(\mathit{aid})) \,, \\ \texttt{lastfrag}(\mathit{store}, \mathit{aid}) &:= \pi_2(\mathit{store}(\mathit{aid})) \,, \\ \texttt{contents}(\mathit{store}, \mathit{aid}) &:= \pi_3(\mathit{store}(\mathit{aid})) \,. \end{aligned}$$

The function `mtype` returns the message type handled at the moment for sender *aid*; `lastfrag` extracts the number of fragments a node has received so far; and the data received so far is accessed by `contents`.

The main intention of the reassembly protocol is to reassemble data received via the CAN bus. For this purpose we define the function

$$\texttt{append}_8 : \texttt{DATA} \times \texttt{DATA} \to \texttt{DATA} \,,$$

which concatenate two strings of data.[22]

The following table summarises the entire data structure we use for the reassembly protocol.

Basic Type	**Variables**	**Description**
MSG	msg	messages
DATA	data	stored data
$\mathbb{N}$	no	fragment counter
MT	mt	message types
CID		CAN IDs
AID	aid	unique sender (application) identifier
Complex Type	**Variables**	**Description**
$\texttt{STORE} \triangleq \texttt{AID} \to \texttt{MT} \times \mathbb{N} \times \texttt{DATA}$	store	storage of received fragments
Constant/Predicate		**Description**
$\varepsilon : \texttt{DATA}$		the empty data string
$\bot_{\texttt{mt}} : \texttt{MT}$		special message type symbol, denoting undefined message type
$\texttt{R}_H : \texttt{SID}$		reassembling protocol identifier for hardware component H
Function		**Description**
$\texttt{can} : \texttt{CID} \times \texttt{DATA} \to \texttt{MSG}$		create CAN messages out of identifier and payload
$\texttt{canid} : \texttt{MT} \times \mathbb{N} \rightharpoonup \texttt{CID}$		returns a CAN ID for a given message type and a counter
$\texttt{sender} : \texttt{CID} \to \texttt{AID}$		determines unique sender ID for a particular message type
$\texttt{mtype} : \texttt{STORE} \times \texttt{AID} \to \texttt{MT}$		the message type of the message currently reassembled
$\texttt{lastfrag} : \texttt{STORE} \times \texttt{AID} \to \mathbb{N}$		indicates the number of fragments received so far
$\texttt{contents} : \texttt{STORE} \times \texttt{AID} \to \texttt{DATA}$		returns the (reassembled) data received so far
$\texttt{fragments} : \texttt{CID} \to \mathbb{N}$		number of CAN messages into which a message is split
$\texttt{pkt} : \texttt{MT} \times \texttt{DATA} \to \texttt{DATA}$		create data packet to be delivered to application
$\texttt{append}_8 : \texttt{DATA} \times \texttt{DATA} \to \texttt{DATA}$		concatenates data strings

Table 4: Data structure for the Reassembly Protocol

B.4.2 Formal Specification

The process $\texttt{RASS}_H$ models all events that occur after a CAN message is received by a node. This includes the reassembly of messages as well as their delivery. This process maintains a data variable `store`, in which it stores received data fragments that await further action.

[22] The index 8 is only a reminder that we append 8 bytes to a data string, and to indicate that this function follows the same

Process 8 Reassembly

```
RASS_H(store) ≝
 1. receive(msg) .
 2. [ msg = can(canid(mt,no),data) ]      /* distill mt, no and data from CAN fragment */
 3.    ⟦aid := sender(mt)⟧
 4.    (
 5.       [ no = 1 ]      /* new message */
 6.          (
 7.             [ fragments(mt) = 1 ]      /* full message received (consists of one fragment only) */
 8.                deliver(pkt(mt,data)) .
 9.                ⟦store(aid) := (⊥_mt, 0, ε)⟧      /* erase data from the data storage */
10.                RASS_H(store)
11.             + [ fragments(mt) > 1 ]      /* fragment to be stored */
12.                ⟦store(aid) := (mt, 1, data)⟧ RASS_H(store)
13.          )
14.       + [ no ≠ 1 ∧ mt = mtype(store,aid) ∧ no = lastfrag(store,aid)+1 ]
15.          /* message needs reassembly */
16.          ⟦store(aid) := (mt, no, append_8(contents(store,aid), data))⟧
17.          (
18.             [ lastfrag(store,aid) = fragments(mt) ]      /* full message received */
19.                deliver(pkt(mt, contents(store,aid))) .
20.                ⟦store(aid) := (⊥_mt, 0, ε)⟧      /* erase data from the data storage */
21.                RASS_H(store)
22.             + [ lastfrag(store,aid) ≠ fragments(mt) ]      /* message not yet complete */
23.                RASS_H(store)
24.          )
25.       + [ no ≠ 1 ∧ (mt = mtype(store,aid) ∧ no = lastfrag(store,aid)) ]      /* repeated fragment */
26.          RASS_H(store)      /* ignore repeated fragment */
27.       + [ no≠1 ∧ (mt≠mtype(store,aid) ∨ (no≠lastfrag(store,aid)+1 ∧ no≠lastfrag(store,aid))) ]
28.          /* non-initial fragment out of order */
29.          ⟦store(aid) := (⊥_mt, 0, ε)⟧
30.          RASS_H(store)
31.    )
```

First, the message has to receive a CAN messages from the CAN driver (**receive**(msg)). After that, the process $\mathtt{RASS}_H$ extracts from it the data received; it also distils the message type mt and fragment counter no from the CAN ID (Line 2). From the message type the process also determines the sender aid of the message (Line 3). As long as a node did not receive all fragments of a message, it cannot deliver the (reassembled) message. Hence upon arrival fragments are sorted by their sender and the carried data is concatenated to any existing data belonging to the same message received before. There need to be as many data queues as there are senders, or as many as there are senders that can potentially send messages to the current node. As soon as all fragments of a fragmented message are received, the message can be passed on to the application layer.

Lines 5–13 cover the case when a first fragment of a split message arrives; here it is sufficient to check the fragment number no. The process then checks whether the CAN message under consideration is just an ordinary CAN message or a fragment (which needs reassembling). In the former case (Lines 7–10)[23] the message is delivered to the application layer (Line 8), the local storage is erased (Line 9) and

lines as the functions $\mathtt{head}_8$ and $\mathtt{tail}_8$.

[23] As stated earlier a 1 in the fragments field of the identifier table indicates a standard CAN message (for example a legacy message).

the process returns to its main routine (Line 10). If the message received is the first fragment of a longer (split) message, the process just stores the fragment (Line 12).

In case the fragment received is not the first fragment, the process checks whether the received fragment fits in the sequence of received messages. This is achieved by the fragment counter no, using $\texttt{no} = \texttt{lastfrag}(\texttt{store},\texttt{aid})+1$. Since we assume that messages are not reordered (see Sect. 3), this counter must have a value that is exactly one higher than the one of the previous message. Line 14 also checks the message type. If the message type does not fit the stored one, the fragment, although it might have the correct fragment number, does not fit. If these checks evaluate to true (Line 14) the received data is concatenated to the data received before from the same sender; and the fragment counter is incremented, which is done by storing no as second component (Line 16). After this update, the process checks whether all fragments of a split message have been received. If this is the case ($\texttt{lastfrag}(\texttt{store},\texttt{aid}) = \texttt{fragments}(\texttt{mt})$) the message is delivered and the data is erased from the data storage. In case there are still fragments missing, the process returns to the main routine, awaiting new messages.

The remaining lines (Lines 25–30) handle the case where an out-of-order fragment is received. If a fragment is received with the same message type and the same fragment counter ($\neq 1$) as the last fragment, the protocol assumes that it is a repeated fragment and ignores the entire message. If fragments of a message were lost or reordered, the reassembly protocol will fail—a check for this is implemented in Line 27. The data received so far is erased and the entire message will be ignored.

Note that when an application cancels a message before the fragmentation protocol has generated all fragments, only a prefix of the full sequence of fragments will ever reach the reassembly protocol. In that case the incomplete message will sit in the slot allocated to its sender until a first fragment of a new message from the same sender arrives. At that time Line 5 of Process 8 will be invoked, and the incomplete message is discarded (Line 9 or 12).

B.4.3 Initialisation

The reassembly protocol R_H is initialised by $\texttt{R}_H : (\xi, \texttt{RASS}_H(\texttt{store}))$, where $\texttt{R}_H$ is the identifier of the local instance of the reassembly protocol, and $\xi(\texttt{store}(\texttt{aid})) = (\bot_{\texttt{mt}}, 0, \varepsilon)$ for all applications $\texttt{aid} \in \texttt{AID}$.

B.5 The Multiplexer

The multiplexer combines several instances of the fragmentation protocol with the CAN driver and the CAN controller. In particular it can receive messages from multiple fragmentation protocols and uses an internal prioritisation mechanism to forward the most important messages. It also checks whether some CAN messages should be erased from the TX buffers (out-queue of the CAN controller) as soon as a new CAN message arrives. By this mechanism we avoid the example where high-priority messages are blocked by low-priority messages (cf. Sect. 1).

The protocol consists of 5 processes: $\texttt{MULTIPLEXER}_H$, $\texttt{NEW\_CAN}_H$, $\texttt{CANCEL\_C}_H$, $\texttt{ACK\_C}_H$ and QMSG.

- The process $\texttt{MULTIPLEXER}_H$ is similar to $\texttt{FRAG}_H$. It is the basic process that receives messages, and, depending on the type of the message, calls other processes. When there is no message handling going on, it idles until a new message arrives.
- The process $\texttt{NEW\_CAN}_H$ handles a new CAN message received from a fragmentation protocol. In particular it stores the CAN message and determines whether the message is important enough to be forwarded straight to the CAN driver.

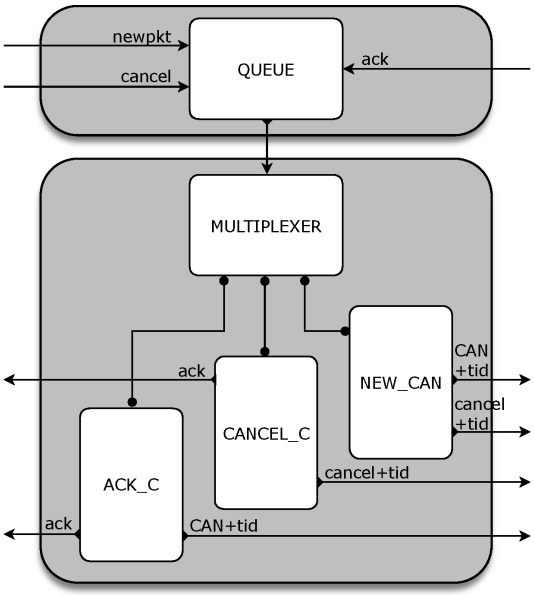

Figure 7: Structure of the Multiplexer

- The process $\texttt{CANCEL\_C}_H$ handles all actions to be performed when a cancellation message arrives from the fragmentation protocol.
- The process $\texttt{ACK\_C}_H$ reacts on incoming acknowledgements, stemming from the CAN driver. It forwards the acknowledgement to the fragmentation protocol to inform it that a new fragment could be sent; at the same time it sends another CAN message to the CAN driver, in case there is one in the local storage.
- The last process QMSG concerns message handling. Whenever a message is received, it is first stored in a message queue. If the multiplexer is able to handle a message it pops the oldest message from the queue and handles it. This process ensures that the multiplexer is input enabled, as required in Sect. B.2.2.

B.5.1 Data Structure

The multiplexer has to store all incoming CAN messages, which were generated by instances of the fragmentation protocol. Since we assume that each CAN ID determines a unique sender, and that each application is sending only one message a time, it suffices to implement an array that maps CAN IDs to messages.

$$\texttt{PRIO} \triangleq \texttt{CID} \to \texttt{MSG}$$

Our formal specification leaves the implementation details open and defines $\texttt{prio} \in \texttt{PRIO}$ as a function. An implementation can be a function, an array, or a priority queue—it is the latter that has been implemented in our research vehicle.

The multiplexer also keeps track of the messages currently stored in the TX buffers. We assume a set TX of TX-buffer identifiers; the variables tid and wid range over TX. It is sufficient to store the (unique) CAN ID together with a flag that indicates whether the CAN controller is currently cancelling the message of the indicated TX buffer. To this end we define a function of type TX_CID, where

$$\texttt{TX\_CID} \triangleq \texttt{TX} \to \texttt{CID} \times \mathbb{B} \ .$$

For a function $txs \in \texttt{TX\_CID}$ we define projections to access the first and second components.

$$\begin{aligned} \texttt{tx}_{\texttt{cid}} : \texttt{TX} \times \texttt{TX\_CID} &\to \texttt{CID} \\ \texttt{tx}_{\texttt{cid}}(tid, txs) &:= \pi_1(txs(tid)) \\ \texttt{tx}_{\texttt{abort}} : \texttt{TX} \times \texttt{TX\_CID} &\to \mathbb{B} \\ \texttt{tx}_{\texttt{abort}}(tid, txs) &:= \pi_2(txs(tid)) \end{aligned}$$

The first function distils the CAN ID of a message that is currently stored in the TX buffer with identifier *tid*; the second one signals whether a cancellation message has been sent to the corresponding TX buffer. Moreover, we define a function that collects all CAN IDs that are currently stored in the TX buffers.

$$\begin{aligned} \texttt{cids} : \texttt{TX\_CID} &\to \mathscr{P}(\texttt{CID}) \\ \texttt{cids}(txs) &:= \{cid \mid \exists tid : \texttt{tx}_{\texttt{cid}}(tid, txs) = cid \neq \bot_{\texttt{cid}}\} \end{aligned}$$

Finally we will use a partial function $\texttt{getWorstTX} : \texttt{TX\_CID} \rightharpoonup \texttt{TX}$ that determines the name of the TX buffer with the least urgent message (the one with the largest CAN ID):

$$\texttt{getWorstTX}(txs) = tid \;\Leftrightarrow\; \texttt{tx}_{\texttt{cid}}(tid, txs) = \max(\texttt{cids}(txs)) \; .$$

When a TX buffer becomes available, we want to select a new message from the priority queue to send next—the one with the smallest CAN ID that is not already forwarded to the TX buffers:

$$\begin{aligned} \texttt{newtask} : \texttt{PRIO} \times \texttt{TX\_CID} &\to \texttt{CID} \\ \texttt{newtask}(prio, txs) &:= \min\{cid \mid prio(cid) \neq \bot_{\texttt{cid}} \wedge cid \notin \texttt{cids}(txs)\} \end{aligned}$$

where $\min\{\}$ is defined to be the special element $\bot_{\texttt{cid}}$. By this latter definition `newtask` becomes a total function; $\texttt{newtask}(prio, txs) = \bot_{\texttt{cid}}$ indicates that no message needs to be scheduled next.

To implement a proper prioritisation mechanism and to avoid the blocking example of Sect. 1 a multiplexer has to determine the n CAN messages in the priority queue with highest priority (lowest CAN identifier)—these messages should be sent as soon as possible. Here n corresponds to the number `#TX` of available TX buffers.

In case there is only one TX buffer, as it is the case for the mission board, we can define a partial function similar to `best` (Sect. B.2.1) that determines the identifier of the CAN message with highest priority currently stored in $prio \in \texttt{PRIO}$:

$$\begin{aligned} & \texttt{1\_best}(prio) = cid \\ \Leftrightarrow\; & \exists d : \big(prio(cid) = \texttt{can}(cid, d) \wedge \\ & \qquad (\exists cid', d' : prio(cid') = \texttt{can}(cid', d') \Rightarrow cid \leq cid')\big) \; . \end{aligned}$$

In case of n TX buffers we require a function that determines n messages. The easiest way is to define a recursive function. The base case ($n = 1$) is a total function corresponding to the function `1_best`:

$$\begin{aligned} \texttt{n\_best} : \mathbb{N} \times \texttt{PRIO} &\to \mathscr{P}(\texttt{CID}) \\ \texttt{n\_best}(1, prio) &:= \begin{cases} \{\texttt{1\_best}(prio)\} & \text{if } \texttt{1\_best}(prio) \text{ is defined} \\ \{\} & \text{otherwise .} \end{cases} \end{aligned}$$

Here, the second case describes the situation of an empty queue `prio`.
The recursive case ($n > 1$) is then defined as

$$\texttt{n\_best}(n, prio) = \texttt{n\_best}(1, prio) \cup \texttt{n\_best}(n-1, prio') \; ,$$

where $prio'(cid) = prio(cid)$ for all $cid \neq \texttt{1\_best}(prio)$ and $prio'(\texttt{1\_best}(prio)) = \perp_{\texttt{msg}}$ (in the case that $\texttt{1\_best}(prio)$ is defined).

This recursive definition seems to be overly complicated. In fact when implementing a priority queue a simple position argument is enough to check whether a given CAN message *msg* is among the *n* messages with highest priority: just insert *msg* into the queue and check if its position is smaller than *n*. Since we want to abstract from implementation details, we chose, however, to present a recursive definition which allows different implementations.

In the specification presented below we have to determine whether a new message should be moved to a TX buffer immediately. Since the number #TX of TX buffers is a constant, we define a function

$$\begin{aligned} \texttt{n\_best} &: \texttt{PRIO} \rightarrow \mathscr{P}(\texttt{CID}) \\ \texttt{n\_best}(prio) &:= \texttt{n\_best}(\#\texttt{TX}, prio)\ . \end{aligned}$$

As for the CAN receiver, we use a queue-style data structure for modelling an inbox queue. In particular we make again use of the standard functions $\texttt{head} : [\texttt{MSG}] \rightharpoonup \texttt{MSG}$, $\texttt{tail} : [\texttt{MSG}] \rightharpoonup [\texttt{MSG}]$ and $\texttt{append} : \texttt{MSG} \times [\texttt{MSG}] \rightarrow [\texttt{MSG}]$.

The following table summarises the entire data structure we use for the multiplexer.

Basic Type	**Variables**	**Description**
MSG	msg	messages
DATA	data	stored data
TX	tid, wid	identifiers of TX buffers
$\mathbb{B}$	abort, suc	Boolean flags
CID	cid, bid	CAN IDs
SID		CAN software component identifiers
Complex Type	**Variables**	**Description**
[MSG]	msgs	message queues
$\texttt{PRIO} \triangleq \texttt{CID} \rightarrow \texttt{MSG}$	prio	priority queue for CAN messages
$\texttt{TX\_CID} \triangleq \texttt{TX} \rightarrow \texttt{CID} \times \mathbb{B}$	txs	array of CAN IDs currently in TX buffer
Constant/Predicate		**Description**
$\perp_{\texttt{msg}} : \texttt{MSG}$		special message symbol (indicating absence of a message)
$\perp_{\texttt{cid}} : \texttt{CID}$		special CAN ID symbol, denoting undefined CAN ID
$[] : [\texttt{MSG}]$		empty queue
$\texttt{C}_H : \texttt{SID}$		sending CAN driver identifier for hardware component *H*
Function		**Description**
$\texttt{cancel} : \texttt{CID} \rightarrow \texttt{MSG}$		cancellation message from fragmentation protocol
$\texttt{cancel} : \texttt{MSG}$		cancellation message to CAN driver
$\texttt{ack} : \mathbb{B} \rightarrow \texttt{MSG}$		acknowledgement (from CAN driver and to fragm. prot.)
$\texttt{can} : \texttt{CID} \times \texttt{DATA} \rightarrow \texttt{MSG}$		create CAN messages out of identifier and payload
$\texttt{msgd} : \texttt{TX} \times \texttt{MSG} \rightarrow \texttt{MSG}$		wrapper function to add a TX-identifier to a message
$\texttt{newtask} : \texttt{PRIO} \rightarrow \texttt{CID}$		selecting the most urgent CAN ID from priority queue
$\texttt{n\_best} : \texttt{PRIO} \rightarrow \mathscr{P}(\texttt{CID})$		returns CAN IDs that should be scheduled
$\texttt{tx}_{\texttt{cid}} : \texttt{TX} \times \texttt{TX\_CID} \rightharpoonup \texttt{CID}$		distils CAN ID of message in TX buffer
$\texttt{tx}_{\texttt{abort}} : \texttt{TX} \times \texttt{TX\_CID} \rightharpoonup \mathbb{B}$		signals whether cancellation signal was sent to TX buffer
$\texttt{getWorstTX} : \texttt{TX\_CID} \rightharpoonup \texttt{TX}$		returns TX buffer with least urgent message
$\texttt{head} : [\texttt{MSG}] \rightharpoonup \texttt{MSG}$		returns the 'oldest' element in the queue
$\texttt{tail} : [\texttt{MSG}] \rightharpoonup [\texttt{MSG}]$		removes the 'oldest' element in the queue
$\texttt{append} : \texttt{MSG} \times [\texttt{MSG}] \rightarrow [\texttt{MSG}]$		inserts a new element into the queue
$\texttt{frag} : \texttt{CID} \rightarrow \texttt{SID}$		gives name of fragmentation protocol handling a message

Table 5: Data structure for the Multiplexer

B.5.2 Formal Specification

The Main Loop. The basic process $\texttt{MULTIPLEXER}_H$ (Process 9) receives messages from the fragmentation protocol or the CAN driver. Since the multiplexer is not always ready to receive messages, we equip the process with an in-queue (see below); so technically the multiplexer receives a message from this queue. $\texttt{MULTIPLEXER}_H$ maintains two data variables `prio` and `txs`. The former implements a priority queue which contains all CAN messages to be sent via the CAN bus; the later is a local storage which keeps track of the CAN IDs currently sent by or stored in the TX buffers of the CAN controller.

Process 9 Multiplexer—Main Loop

$\texttt{MULTIPLEXER}_H(\texttt{prio},\texttt{txs}) \stackrel{def}{=}$

1. **receive**(`msg`) .
2. (
3. [`msg` = `can`(`cid`,`data`)] /* new fragment */
4. $\texttt{NEW\_CAN}_H$(`msg`,`prio`,`txs`)
5. + [`msg` = `cancel`(`cid`)] /* cancellation message received */
6. $\texttt{CANCEL\_C}_H$(`cid`,`prio`,`txs`)
7. + [`msg` = `msgd`(`tid`,`ack`(`suc`))] /* message from CAN controller */
8. $\texttt{ACK\_C}_H$(`suc`,`tid`,`prio`,`txs`)
9.)

First, as usual, a message has to be received (Line 1). After that, the process $\texttt{MULTIPLEXER}_H$ checks the type of the message and calls a process that can handle this message: in case a CAN message is received from the fragmentation protocol, the process $\texttt{NEW\_CAN}_H$ is called (Line 4); in case of an incoming cancellation request the process $\texttt{CANCEL\_C}_H$ is executed (Line 6); and in case a message from the CAN driver is read, the process $\texttt{ACK\_C}_H$ is called (Line 8). In case a message of any other type is received, the process $\texttt{MULTIPLEXER}_H$ deadlocks; it is a proof obligation to check that this will not occur.

New CAN Message. In case a new CAN message is sent from an instance of the fragmentation protocol, the process $\texttt{NEW\_CAN}_H$ stores the CAN message and determines whether the newly received message is important enough to be forwarded directly to the CAN driver. The formal specification is shown in Process 10.

The received CAN message is first stored in the queue `prio` (Line 2), which contains all messages to be sent via the CAN bus. The protocol just stores the newly received message, it does not check for emptiness of $\texttt{prio}(\texttt{cid})$. Therefore, to guarantee that no message is lost the property $\texttt{prio}(\texttt{cid}) = \bot_{\texttt{msg}}$ needs to hold before Line 2 is executed; it needs to be proven. The protocol then determines whether the message should directly be forwarded to the CAN driver—this is the case if the CAN ID is among the n messages with lowest CAN IDs currently stored in `prio` (Line 4). Here n equals the number $\#\texttt{TX}$ of TX buffers available in the CAN controller. Lines 5–22 present all actions to be performed in case the message is forwarded to the CAN driver.

In case there exists an empty TX buffer `tid`, which is currently not used, the message should be sent to this TX buffer, and there is no need to erase a used TX buffer. The empty buffer `tid` is chosen in Line 6.[24] The CAN message is then forwarded to the connected CAN driver $\texttt{C}_H$ in Line 8. Since the CAN driver needs also the name of the TX buffer to be used, the value `tid` is sent next to the CAN message `msg`. The multiplexer also updates the local variable `txs` (Line 7), which keeps track of those CAN

[24] Since `tid` is a free variable, it will be instantiated with a value that validates $\texttt{tx}_{\texttt{cid}}(\texttt{tid},\texttt{txs}) = \bot_{\texttt{cid}}$; so the condition in the guard is satisfied iff $\exists \texttt{tid} \in \texttt{TX} : \texttt{tx}_{\texttt{cid}}(\texttt{tid},\texttt{txs}) = \bot_{\texttt{cid}}$.

Process 10 New CAN Message Received

```
NEW_CAN_H(msg,prio,txs) ≝
 1. [ msg = can(cid,data) ]        /* distill cid out of msg */
 2.   ⟦prio(cid) := msg⟧          /* store message in priority queue */
 3.   (
 4.     [ cid ∈ n_best(prio) ]     /* message should be scheduled */
 5.       (
 6.         [ tx_cid(tid,txs) = ⊥_cid ]     /* TX buffer tid is free */
 7.           ⟦txs(tid) := (cid,false)⟧
 8.           unicast(C_H, msgd(tid,msg)) .     /* pass message to CAN driver, to put in free slot */
 9.           MULTIPLEXER_H(prio,txs)
10.         + [ ∀tid ∈ TX : tx_cid(tid,txs) ≠ ⊥_cid ]     /* cancel message with lowest priority */
11.           (
12.             ⟦wid := getWorstTX(txs)⟧     /* identify TX buffer containing lowest CAN ID */
13.             (
14.               [ tx_abort(wid,txs) = false ]     /* TX buffer wid is still active */
15.                 ⟦txs(wid) := (tx_cid(wid,txs),true)⟧     /* set the abort-flag of buffer wid */
16.                 unicast(C_H, msgd(wid,cancel())) .     /* cancel contents of buffer wid */
17.                 MULTIPLEXER_H(prio,txs)
18.               + [ tx_abort(wid,txs) = true ]     /* TX was already asked to clean up */
19.                 MULTIPLEXER_H(prio,txs)
20.             )
21.           )
22.       )
23.     + [ cid ∉ n_best(prio) ]     /* message not important enough to be scheduled right now */
24.       MULTIPLEXER_H(prio,txs)
25.   )
```

identifiers that are currently sent by or stored in the TX buffers. By this, the newly received message has been handled and the process can return to the main routine (Line 9).

In case all available TX buffers are used (Line 10), the least important message—the CAN message with the largest CAN ID—needs to be removed from the TX buffer and rescheduled later. This avoids the blocking example presented earlier. In Line 12 the process $\texttt{NEW\_CAN}_H$ determines the name of the TX buffer that contains the 'worst' message currently handled for sending. The CAN message that should be stored in this particular TX buffer cannot be put there immediately; a cancellation request needs to be sent first, and an acknowledgement needs to be received that informs the multiplexer about a free TX buffer. The routine checks whether a cancellation request was sent earlier, using the function $\texttt{tx}_{\texttt{abort}}$. If this is the case, it returns straight to the process $\texttt{MULTIPLEXER}_H$; otherwise a cancellation message is sent to the CAN driver $\texttt{C}_H$, identifying the TX buffer that needs cancellation (Line 16).

If the newly received CAN message is not important enough to be forwarded to the CAN driver immediately (Line 23), the process $\texttt{NEW\_CAN}_H$ just returns to the main process (Line 24), where it awaits a new message. The stored message will be handled later when a TX buffer becomes available.

Cancellation Message. All actions to be taken if a cancellation request is received are formalised in Process 11. Any cancellation message received carries the CAN ID `cid` of the message to be cancelled.

In case the multiplexer has previously handled the message already, the value of $\texttt{prio}(\texttt{cid})$ is $\bot_{\texttt{msg}}$ (Line 1). This situation can happen if for example the message was sent successfully, but the acknowledgement was not received by the time the fragmentation protocol requested the cancellation. In this case, the process has nothing to do, ignores the message and returns to the main process $\texttt{MULTIPLEXER}_H$ (Line 2).

Process 11 Cancellation Message Received

```
CANCEL_C_H(cid,prio,txs) =def
 1. [ prio(cid) = ⊥msg ]        /* nothing to cancel */
 2.    MULTIPLEXER_H(prio,txs)
 3. + [ prio(cid) ≠ ⊥msg ]      /* send cancellation message to CAN driver */
 4.    ⟦prio(cid) := ⊥msg⟧
 5.    (
 6.       [ tx_cid(tid,txs) = cid ]      /* determine TX buffer */
 7.          (
 8.             [ tx_abort(tid,txs) = true ]      /* cancellation already sent */
 9.                MULTIPLEXER_H(prio,txs)
10.             + [ tx_abort(tid,txs) = false ]
11.                ⟦txs(tid) := (tx_cid(tid,txs),true)⟧
12.                unicast(C_H,msgd(tid,cancel())) .
13.                MULTIPLEXER_H(prio,txs)
14.          )
15.       + [ ∀tid ∈ TX : tx_cid(tid,txs) ≠ cid ]      /* message not in TX */
16.          unicast(frag(cid),ack(false)) .
17.          MULTIPLEXER_H(prio,txs)
18.    )
```

In case the message that needs cancellation is still stored in the buffer `prio` the process first erases it from the buffer in Line 4. Afterwards it checks whether the CAN driver needs to be informed as well. This check is performed by analysing `txs`, which keeps the status of the TX buffers. In case there is a CAN message with identifier `cid` in TX buffer `tid`, the actions of Lines 8–13 are taken.

If the process $\texttt{CANCEL\_C}_H$ has sent a cancellation request before, i.e., the flag $\texttt{tx}_{\texttt{abort}}(\texttt{tid},\texttt{txs})$ is true, the process does not take any action, and returns to the main process (Line 9). Otherwise the flag is set to true (Line 11) and a cancellation message is sent to the CAN driver (Line 12).

If the message with CAN identifier `cid` can be found in `prio` but not in any of the TX buffers (in `txs`), no cancellation request needs to be forwarded and the cancellation process is finished (the message was not yet forwarded to the CAN driver). As a consequence the process informs the instance of the fragmentation protocol responsible for this message about the successful cancellation, by sending an `ack`-message in Line 16.

Notification from the CAN Driver. When an `ack`-message is received from the CAN driver by the multiplexer, the process $\texttt{ACK\_C}_H$ is executed.

If the multiplexer received a message that carries an identifier of the TX buffer (`tid`), but has no local knowledge about it ($\texttt{tx}_{\texttt{cid}}(\texttt{tid},\texttt{txs})$ is undefined), something went wrong and the protocol deadlocks with an error. In the future we hope to show that this situation cannot occur.

Lines 3–22 present the standard case—an acknowledgement from `tid` is received, informing about the status of a message with CAN ID `cid`. Independent of the contents of the acknowledgement, the TX buffer is erased (Line 4). By this the array `txs` is adapted to the status of the actual TX buffers of the CAN driver—the message with ID `cid` was either sent successfully or erased from the buffer.

If the sending of the CAN message was successful ($\texttt{suc} = \texttt{true}$, Line 6), the message is erased from the priority queue (Line 7), and the corresponding instance of the fragmentation protocol (the one which is responsible for CAN ID `cid`, and determined by $\texttt{frag}(\texttt{cid})$) is informed about the success (Line 8). Since the TX buffer `tid` is now empty, the process checks whether there are pending messages that needs to be scheduled. If $\texttt{newtask}(\texttt{prio},\texttt{txs}) = \bot_{\texttt{cid}}$ (Line 14) no message needs to be scheduled and the process returns to the main process $\texttt{MULTIPLEXER}_H$, where it can receive new, incoming messages. In

Process 12 Acknowledgement from the CAN driver

```
ACK_C_H(suc,tid,prio,txs) =def
 1. [ tx_cid(tid,txs) = ⊥_cid ]        /* mailbox idle */
 2.    ERROR
 3. + [ tx_cid(tid,txs) = cid ∧ cid ≠ ⊥_cid ]        /* mailbox non-idle, distill cid */
 4.    ⟦txs(tid) := (⊥_cid,false)⟧
 5.    (
 6.       [ suc = true ∨ prio(cid) = ⊥_msg ]        /* positive acknowledgment received */
 7.          ⟦prio(cid) := ⊥_msg⟧        /* erase message */
 8.          unicast(frag(cid),ack(suc)) .
 9.          (
10.             [ newtask(prio,txs) = bid ∧ bid ≠ ⊥_cid ]        /* more messages to be handled */
11.                ⟦txs(tid) := (bid,false)⟧
12.                unicast(C_H,msgd(tid,prio(bid))) .
13.                MULTIPLEXER_H(prio,txs)
14.             + [ newtask(prio,txs) = ⊥_cid ]        /* nothing to handle at the moment */
15.                MULTIPLEXER_H(prio,txs)
16.          )
17.       + [ suc = false ∧ prio(cid) ≠ ⊥_msg ]        /* task needs rescheduled */
18.          ⟦bid := newtask(prio,txs)⟧        /* determine next task */
19.          ⟦txs(tid) := (bid,false)⟧
20.          unicast(C_H,msgd(tid,prio(bid))) .
21.          MULTIPLEXER_H(prio,txs)
22.    )
```

case there is at least one message stored in `prio` that is not yet forwarded to the CAN controller, the process chooses the message with best (least) CAN ID, using the function `newtask(prio,txs)`; the least CAN ID is stored in the variable `bid`. This CAN message is then unicast to the controller, together with the name `tid` of the TX buffer to be used (Line 12); the local knowledge about `tid` is adapted in Line 11: the CAN ID `bid` is stored together with the flag true, indicating that no cancel request are sent for this message.

In case of a negative acknowledgement (`suc`$=$`false`), it matters whether the unsent CAN message (with identifier `cid`) still occurs in the priority queue (`prio(cid)` $\neq \bot_{msg}$). If this is the case (Line 17), the message must have been removed from the TX buffer in order to make place for a message with a higher priority. Since a TX buffer is now empty, a new CAN message `bid` can be passed to the CAN driver (if there is any). This scheduling happens in Lines 18–21, and is similar to Lines 10–13.

If the message `cid` that was dropped from the TX buffer does not occur in the priority queue (Line 6), the message must have been removed from the TX buffer in response to a cancellation request from the fragmentation protocol. The latter is informed about the successful cancellation in Line 8. Subsequently, the multiplexer schedules another message for transmission over the CAN bus if appropriate (Lines 9–16).

Message Queue. To guarantee that our system is non-blocking, the multiplexer is equipped with an in-queue. This input enabled queue (Process 13) runs in parallel with $\texttt{MULTIPLEXER}_H$. Every incoming message from the fragmentation protocol or the CAN driver is first stored in this queue, and piped from there to the multiplexer $\texttt{MULTIPLEXER}_H$, whenever $\texttt{MULTIPLEXER}_H$ is ready to handle a new message.

Here we simply assume that the length of the queue is unbounded. However, in the future we hope to show that under some mild conditions a queue with a limited capacity is equally effective.

The process is identical to the CAN receiver (Process 3 of Sect. B.2.2), except that the unicast-procedure is replaced by a **send**-action, which triggers the forwarding to the process $\texttt{MULTIPLEXER}_H$.

Process 13 Message Queue

```
QMSG(msgs) =def
1.    /* store incoming message at the end of msgs */
2.    receive(msg) . QMSG(append(msg,msgs))
3.  + [ msgs ≠ [] ]      /* the queue is not empty */
4.    (
5.       /* pop top message and send it to the multiplexer */
6.       send(head(msgs)) . QMSG(tail(msgs))
7.       /* or receive and store an incoming message */
8.       + receive(msg) . QMSG(append(msg,msgs))
9.    )
```

The **receive**-action in Line 8 is needed to guarantee input-enabledness, meaning that the process QMSG is always ready to receive a new, incoming message, even when the process is about to send a message itself.

B.5.3 Initialisation

The multiplexer M_H is initialised by $\mathtt{M}_H : \big((\xi, \mathtt{MULTIPLEXER}_H(\mathtt{prio}, \mathtt{txs})) \between (\zeta, \mathtt{QMSG}(\mathtt{msgs}))\big)$, where $\xi(\mathtt{prio}(\mathtt{cid})) = \bot_{\mathtt{msg}}$ for all cid, $\xi(\mathtt{txs}(\mathtt{tid})) = (\bot_{\mathtt{cid}}, \mathtt{false})$ for all tid, and $\zeta(\mathtt{msgs}) = []$.