Design and verification of a media redundancy management driver for a CAN star topology

David Gessner, Manuel Barranco, and Julián Proenza, *Member, IEEE*

***Abstract*—Some of the severe dependability limitations of Controller Area Network (CAN) can be overcome by replacing its bus topology with a star topology. Thus, a replicated star topology with advanced error-containment and fault-tolerance mechanisms for CAN, called ReCANcentrate, has been proposed. Its two hubs are coupled with each other and create a single logical broadcast domain. This allows each node to easily manage the replicated star by means of a software driver, called reCANdrv, that abstracts away the details of this replication. The goal of reCANdrv is to manage the star's media redundancy transparently for a CAN application, allowing it to exchange information through the star while tolerating faults. This paper describes the design of reCANdrv, the specification as properties of reCANdrv's correct redundancy management, and the verification of these properties by means of model checking.**

***Index Terms*—fault tolerance, media redundancy management, replicated star topology, formal verification, model checking, UPPAAL, field buses, Controller Area Network**

I. Introduction

THE use of Controller Area Network (CAN) [1] has steadily increased since its release. This trend is expected to continue at least for the next 15 years [2] and, as was predicted by some authors, e.g. [3], CAN is still the most widely-used in-vehicle network. Further, due to the cost-sensitivity of the automotive industry, the low-cost CAN is an appealing candidate even when advanced real-time and dependability features are required [4]. Several solutions have been developed to improve these features in CAN [4], and some of them are being combined in the CANbids project. Moreover, integrated field bus architectures are expected to provide interoperability among CAN and other protocols [5].

The interest in CAN is clearly seen not only in the amount of academic work being done to leverage the above-mentioned features, e.g. [6]–[19]; but also in the industry, where CAN plays an important role in dependability-related systems.

For highly-reliable applications, two particular approaches that use star topologies [20] were proposed for CAN to overcome the dependability limitations imposed by its bus topology [21]. The first, CANcentrate [21], attacks CAN's limited error containment with a simplex star topology by allowing the star's central element—the hub—to isolate faulty hub ports. The second, ReCANcentrate [21], provides fault tolerance through media redundancy with a replicated star with two CAN hubs. In this way, it eliminates the several single points of failure of CAN and the sole single point of failure the hub represents in CANcentrate. The actual effectiveness of these stars to increase the reliability when compared with CAN has recently been quantitatively corroborated [22]–[24].

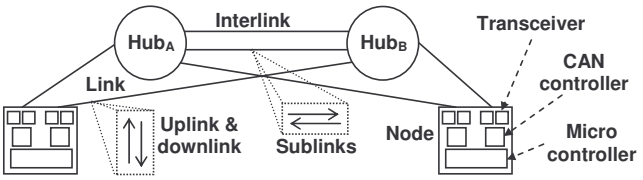

Fig. 1. ReCANcentrate architecture.

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

The hubs couple the signals from their uplinks in an internal AND-gate, whose result is exchanged through the interlinks. Each hub then couples the incoming interlink traffic with the aforementioned AND-gate in a second AND-gate. The result is signaled by each hub on its downlinks. All this is performed within a fraction of the bit time, thereby preserving CAN's *in-bit response* and implementing CAN's *wired-AND* while providing a network-wide single broadcast domain. In other words, in the absence of faults all CAN controllers sample the same bit-value for each bit received from the hubs. This allows a media redundancy management approach for the nodes that is, as opposed to other approaches, e.g. [9], compatible with traditional event-triggered CAN applications.

The nodes' media redundancy management is implemented by the *reCANdrv* software driver. Its goal is to provide a *correct redundancy management*, i.e., to transparently and properly manage the available media redundancy under faults. This means that each node's application must be able to access the channel if it has a correct connection to a correct hub. In particular, reCANdrv has to provide mechanisms to tolerate permanent faults affecting one of the hubs, or the link, transceivers, or CAN controller that connects the node to one of the hubs. Transient channel faults are not handled by the

D. Gessner, M. Barranco and J. Proenza are with the Systems, Robotics and Vision group (SRV), Departament de Matematiques i Informatica, Universitat de les Illes Balears (UIB), 07122 Palma de Mallorca, Spain (e-mail: david.gessner@uib.es; manuel.barranco@uib.es; julian.proenza@uib.es)

driver, but through the CAN error handling mechanisms of the CAN controllers [25]. Additionally, reCANdrv tolerates, to some extent, inconsistent message omission (IMO) scenarios [26], [27] caused by transient or permanent faults.

The viability of a preliminary version of reCANdrv, which does not include all the features of the current design, has already been experimentally assessed by means of a prototype [28]. Nevertheless, since ReCANcentrate is intended for applications that require high reliability, it is of utmost importance to also ensure that the driver has been designed correctly. For this, it is appropriate to use formal methods, which in the context of industrial systems have recently gathered increased interest [29]–[34].

This paper introduces the design of reCANdrv; formalizes as properties the correct redundancy management it provides; describes a model of reCANdrv that was implemented as a series of timed automata; and formally verifies, by means of model checking, that the model satisfies the properties.

II. Fault model

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

Everything beyond the nodes' CAN controllers (on the transceivers' side) is part of the channel, which may fail in arbitrary ways as long as no faults occur that partition the nodes into subsets that cannot communicate with each other. That is, a *single broadcast domain* is assumed at all times. This is realistic because ReCANcentrate achieves its single broadcast domain through its redundant interlinks.

It is also assumed that the nodes' microcontrollers and the implemented software routines are not affected by faults, e.g., that they are not affected by memory corruptions.

III. Media management

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

When a frame is exchanged through the network, i.e., when a *communication event* occurs, each node expects its two controllers to quasi-simultaneously notify of that event with an interrupt. Thus, when no faults occur, the node manages transmissions and receptions as follows. If the node successfully transmits a frame, the tx and the non-tx controller generate an interrupt to notify of the transmission and reception of this frame respectively; thus, the node only needs to accept the transmission and release the reception buffer of the non-tx controller. If the node receives a frame sent from another node, it is notified of this by its two CAN controllers and simply consumes the frame received at one of the controllers and releases the reception buffers of both controllers.

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

If the controller that omits is the non-tx controller, the driver does not discard it because the node can still correctly receive and transmit through the tx controller.

If the tx controller omits, the driver diagnoses it as faulty by initiating a *transmission (tx) timer*, which is enabled when the application requests the transmission of a message from the driver. It is disabled once the driver tells the application that the requested message has been transmitted. If the timer expires before the tx controller notifies of a successful transmission, the driver discards it, uses the other controller as the tx controller, resets the tx timer, and instructs a retransmission through the new tx controller. This ensures that if the old tx controller was not able to transmit, a transmission is performed through the other controller. Note that the tx timer must be set appropriately: it must only expire after a permanent fault prevents the tx controller from communicating.

Specifically, the timer's value must be greater than the *driver's worst-case transmission response time* d with an error model that includes transient faults only. Note that d excludes any queuing delay at the application, as d is the time between when a transmission is requested from the driver by the application until the application is notified of the successful transmission by the driver. Thus, a way to ensure that the timer's value is correct for a given message is to make it greater than the message worst-case response time, R, which does include the queuing delay at the application. The analysis proposed in [35] can be used to calculate R if the application queues the messages in FIFO order. This is so because reCANdrv uses the controllers' hardware buffers following a FIFO policy. R would be the worst-case response time of the FIFO-queued message according to [35] plus some overhead. This overhead would be comprised of the time the driver takes to notify the application of a successful transmission and the overhead of recovering from transient

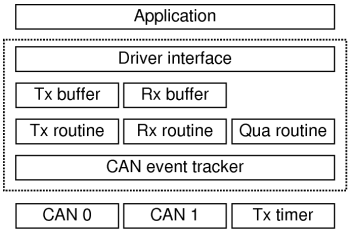

Fig. 2. Basic reCANdrv architecture.

errors. The former can be measured and for the latter an error model similar to the one in [36] can be used.

A controller is also discarded when it generates an *error warning* interrupt, which indicates that the controller's error counters [25] reached a threshold. This prevents the controllers from entering the *error-passive state* [25], in which they could inconsistently exchange frames, leading to IMOs.

Finally, if the non-tx controller omitted a notification while the tx controller notified a transmission, and the number of consecutive omission discrepancies has not reached a threshold, the driver instructs a retransmission through the tx controller as a best-effort attempt to prevent IMOs. However, if the threshold is reached, no further retransmissions are performed since in that case a permanent fault is more likely.

IV. The architecture of reCANdrv

Fig. 2 shows reCANdrv's architecture. At the top is the application, at the bottom the two controllers and the tx timer. The driver is a software layer in-between that allows the application to use the two controllers as if there was only one. Below are the driver's *transmission (tx)* and *reception (rx) buffers*. When the application requests the transmission of a message, a copy is stored in the driver's tx buffer and in the transmission buffer of the tx controller. The driver uses its copy for management operations, e.g., if the driver diagnoses the tx controller as faulty before that controller successfully transmits, it uses the copy to transfer the message to the other controller. Regarding the rx buffer, it allocates the last received message. By comparing the received message with the message, if any, in the tx buffer, the driver can check if it received a message from another node or one that it itself transmitted. When the driver is notified of a reception, it immediately copies the received message from one of the controllers to the driver's rx buffer. This ensures that a copy of the received message is saved as soon as possible, which can prevent some message losses if a controller fails.

The driver has several cooperating *media management routines*. These are shown in Fig. 2 and are the *transmission (tx) routine*, the *reception (rx) routine*, and the *quarantine (qua) routine*. They are invoked by the *media management ISRs* (not shown), which are ISRs that call with the appropriate parameters the corresponding media management routine. The media management ISRs are the *ew0* and *ew1* ISR, which are invoked after an error warning and call the qua routine; the *rx0* and *rx1* ISR, which are invoked after a reception and call the rx routine; and the *tx0* and *tx1* ISR, which are invoked after a transmission and call the tx routine.

TABLE I
Interrupts, ISRs, and functions of reCANdrv.

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

The media management routines are invoked after error warnings, transmissions, and receptions by means of software interrupts generated by the *CAN event tracker* function (Fig. 2). This function is called by the *tracker ISRs*: *tracker0* and *tracker1*. The tracker0 ISR is invoked when the first controller generates a transmission, reception, or error warning interrupt; whereas the tracker1 ISR is invoked when the second controller generates the equivalent interrupts.

The CAN event tracker also sets a *tracking variable* to keep track of what interrupts occurred. These variables are used by the media management routines to cooperate with each other.

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

Tab. I lists the interrupts handled by reCANdrv, the invoked ISRs, and the functions called by the ISRs. For the tracker ISRs, the generated software interrupt is also indicated. The handled interrupts are ordered by descending priority.

V. Properties of correct redundancy management

A *correct controller* is one that is not affected by faults and is in the *error active state* [25], i.e., in a state in which it can fully participate in the communication. A *correct link* is one that is connected to a non-faulty hub, is not affected by permanent faults, and whose transient faults have a frequency low enough to not trigger error warning at its controller.

The goal of reCANdrv is to provide correct redundancy management, i.e., to correctly manage the redundancy of ReCANcentrate. As indicated previously, for this reCANdrv must allow the application to transmit and receive through the channel as long as there is one correct controller with a correct link. This goal can be specified as a series of properties, which are listed at the end of this section. However, first a series of terms, notations, and assumptions need to be introduced.

A *message* is a unit of information that a node's application deals with. A *frame* is a unit of information exchanged on the channel. It is assumed that each message can be encapsulated in a single frame. Nevertheless, in the case of retransmissions multiple frames may contain the same message. Moreover, frames can also not encapsulate messages because a frame may also serve control functions on the communication channel. Examples are *error*, *overload*, and *remote frames* [25].

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

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

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

The properties of correct redundancy management are based on the following assumptions: (i) the channel provides a single broadcast domain; (ii) there is no unicast or multicast addressing of messages, but only broadcast addressing to the applications; (iii) the application does not mtx-request a new message until it was notified by the driver of a tx-success of the previous mtx-request; (iv) as soon as a controller is no longer correct, it generates an alert interrupt and immediately stops generating interrupts and traffic on the channel; (v) at least one CAN controller remains correct and has a correct link at all times; (vi) the CAN controllers are BasicCAN [37], i.e., they only have a single transmission buffer; (vii) the driver's implementation handles a communication event before the subsequent communication event occurs (this is realistic as measured in [28] with the reCANdrv prototype).

Having introduced these terms, notations, and assumptions, reCANdrv's correct redundancy management can be formalized as the following properties. **(P1) Pass on integrity:** any message M_j passed on to the application was received in a frame $F_i(M_j)$ by at least one controller; **(P2) Double reception implies single pass on:** each message M_j originated in a remote node and received in a frame $F_i(M_j)$ by both controllers is passed on to the application exactly once; **(P3) Pass on validity:** each message M_j originated in a remote node and received in a frame $F_i(M_j)$ by at least one correct controller is passed on to the application, unless a single controller received $F_i(M_j)$ and that controller alerted of its failure; **(P4) No duplicate pass on:** each message M_j originated in a remote node and received in a frame $F_i(M_j)$ by at least one controller is passed on to the application at most once; **(P5) No pass on of self-received messages:** no message M_j is passed on to the application when it was self-received in a frame $F_i(M_j)$; **(P6) Ordered pass on:** if messages M_j and M_k are passed on to the application, then M_j is passed on before M_k only if M_j was received by any one correct controller before M_k was received by any one correct controller; **(P7) Guaranteed transmission:** if the application mtx-requests a message M_j, one of the controllers transmits a frame $F_i(M_j)$; **(P8) Bounded retransmissions:** the controllers perform a bounded number of retransmissions of frames $F_i(M_j)$ encapsulating a message M_j; **(P9) FIFO transmission:** if the application mtx-requests M_j and M_k, then M_j is transmitted before M_k only if the application mtx-requested M_j before M_k; **(P10) Bounded time to satisfy an mtx-request:** if the application mtx-requests a message M_j, the driver notifies the application within a finite amount of time of a tx-success.

VI. A MODEL OF RECANDRV

This section introduces the model used to verify the above properties. The model was implemented using the UPPAAL model checker [38], which has been used previously to verify CAN applications and CAN-based systems, e.g., [39]–[43]. Unfortunately, these previous works are not a good starting point for the modelling presented here. This is so because each one of them focuses on different features of CAN and their levels of abstraction are inadequate to model reCANdrv.

Using UPPAAL a model is implemented using a network of timed automata. However, to meet this journal's space limitations, the model is described in a more abstract way. The technical details of the implemented UPPAAL model are available in [44] and the model itself at [45].

A. Model components

The model is basically comprised of the communication channel and a single node. The single node has two CAN controllers, an application executing on it, a tx timer, and the reCANdrv driver. Only a single node is considered because the properties that formalize the correct redundancy management of reCANdrv are local properties of a node. All the other nodes are represented abstractly by the channel. Fig. 3 shows the model's components, which will be described shortly.

Messages are modeled as non-zero positive integers, which correspond to the msns of the messages. Frames are modeled as tuples, called *f-tuples*, of two non-zero positive integers.

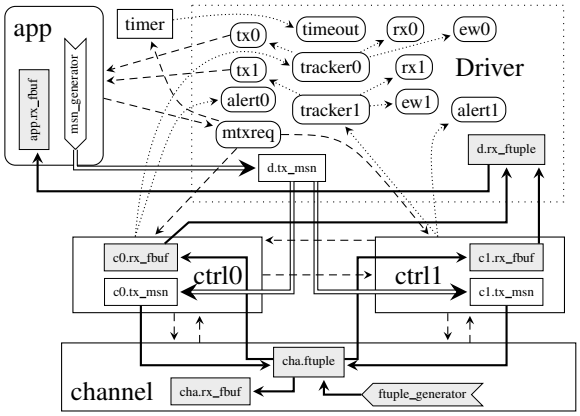

Fig. 3. Model overview.

One integer represents the encapsulated message and its value is the msn of that message. The other represents the frame and its value corresponds to the frame's fsn. To more easily verify the model, the integers used for the msns take values from a subset S_m of consecutive non-zero positive integers. Similarly, the fsns take values from another non-overlapping subset S_f of consecutive non-zero positive integers, i.e., $S_m \cap S_f = \varnothing$.

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

For f-tuples generated by one of the controllers, the msn is the one of the message encapsulated in a frame that is modeled as being transmitted. For example, if a controller transmitting the frame $F_i(M_j)$ is modeled, the f-tuple is (i, j). On the other hand, the msns used in the f-tuples created by the channel can have arbitrary values since the specific messages that are passed on to the application are irrelevant to verify the properties. However, for convenience, the msns used in the f-tuples generated by the channel do not overlap with the ones used in the f-tuples generated by the node's controllers.

As indicated previously, Fig. 3 shows the main components of the model. The bottom shows the modeled communication channel. The channel can generate new f-tuples and contains a single-slot f-tuple buffer (`cha.ftuple`) to store the f-tuple that models the currently being signaled frame and a multi-slot f-tuple buffer (`cha.rx_fbuf`) that stores all the f-tuples that have been transmitted. Everything above is part of the node.

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

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

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

The channel's model is implemented by a single timed automaton; whereas the node's model is implemented by several, one for each component of the modeled node. Note that the automata corresponding to the software components are implemented such that deriving the actual code implementation for a prototype of reCANdrv is fairly simple. Each such automata a basically invokes a driver function f_a in pseudo C that does not directly interact with the controllers and the timer, but by means of additional functions $f_{a_1}, f_{a_2}, \ldots, f_{a_n}$ [44]. These abstract away the details of the models of these hardware components. Thus, a code implementation of the driver basically requires to rewrite these additional functions to adapt them to the details of the actual hardware controllers and timers.

B. Model behavior

Initially the channel is free. This means that the channel's `cha.f_tuple` buffer initially contains an empty f-tuple representing an idle channel. The signaling of a frame on the channel is modeled as the corresponding f-tuple being stored in the `cha.f_tuple` buffer. Once the signaling is finished, the channel overrides the `cha.f_tuple` buffer with an f-tuple modeling the intermission between frames [25]. Shortly after that the channel becomes free again, that is, the channel overwrites the intermission f-tuple with the empty f-tuple.

An mtx-request by the app is modeled by the app generating a new msn and invoking the mtxreq routine with that msn as a parameter. The mtxreq routine copies the msn to the `d.tx_msn` buffer. It then models the issuing of an ftx-request to the tx controller as follows. It copies the msn from the `d.tx_msn` buffer to the tx controller's `tx_msn` buffer and sets a boolean

variable to indicate to the tx controller that a transmission is pending. Meanwhile, the app waits until it receives a signal that the mtx-request was completed successfully.

The subsequent transmission is modeled when the channel is free, which it indicates by overwriting the `cha.f_tuple` buffer with an empty f-tuple, as described above. When this occurs, and assuming that the channel does not initiate the signaling first by copying its own f-tuple to `cha.f_tuple`, the transmission is modeled as follows. The tx controller creates an f-tuple from the msn located in its `tx_msn` buffer and from the next fsn of the integer subset S_{f1} shared with the other controller model. It then writes into the `cha.f_tuple` buffer the created f-tuple and signals to the channel and the other controller that it is transmitting a message. This causes the channel to store a copy of the f-tuple in the `cha.rx_fbuf` buffer and both controllers to wait until the channel notifies that the signaling of the message is finished. This waiting models the controllers being busy receiving or transmitting the frame. This means that the frames are modeled as never being aborted by errors. Aborted frames are not modeled because the fact that a frame is aborted does not trigger the driver's execution.

An f-tuple reception begins when the channel generates an f-tuple, copies it to the `cha.f_tuple` buffer, and then indicates that it is signaling a frame. Upon this the modeled controllers wait until the channel notifies that the signaling finished.

The channel's notification that the signaling of a frame finished can be of two types: a *communication event*, in which case the controller accepts the frame, or a *communication error*, which leads the controller to reject the frame. Given a frame, the model signals a communication event to both controllers or, alternatively, a communication event to one of them and a communication error to the other one. This latter case is used to model omission discrepancies between the controllers of the modeled node, which is important to test the driver's best-effort attempt to reduce the number of IMOs. The model does not signal a communication error to both controllers since that represents the case where both controllers reject a frame, which does not invoke the driver.

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

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

Finally, the rest of circumstances that can invoke the driver are the occurrence of an alert, a tx time out, and an error warning. They are modelled to occur when they provoke the maximum interference with the ISRs that manage a communication event, i.e. immediately after the signaling of a frame finishes. In particular, since an error warning does happen due to a controller detecting a channel error, the signaling of the corresponding error frame is also modelled, unless the controller is no longer able to transmit.

VII. Model verification

The properties of Section V need to be expressed using UPPAAL's query language to be verified with the model.

A. The UPPAAL query language

UPPAAL's query language [38] allows to define properties to be tested by the UPPAAL model checker. During a verification, UPPAAL automatically generates all the execution paths of the model that are required in order to verify each property.

The following types of properties are relevant for this paper. (i) $E\diamond\ p$: tests if there exists an execution path in which the condition p (a boolean expression over locations, variables, and clocks [38] of the model's timed automata) eventually (in some state of the path) holds. (ii) $E\Box\ p$: tests if there exists an execution path in which p holds for all the states in the path; (iii) $A\Box\ p$: tests if for every execution path, p holds for all the states in the path; (iv) $A\diamond\ p$: tests if for every execution path, p holds for at least one of the states in the path; (v) $q \rightsquigarrow p$: tests if every execution path that starts from a state satisfying q reaches later on a state in which p holds.

UPPAAL also provides a *universal* quantifier, expressed as $\forall(v : t[f, l])\ b$. It returns *true* if for all values v of type t in the range $[f, l]$, both inclusive, the boolean expression b is true.

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

B. Query helper functions and helper automata

The queries use the following functions. (i) *fsns_in_range(b, f, l)*: returns *true* if all f-tuples of buffer b have an fsn in the integer range $[f, l]$, both inclusive; (ii) *msns_in_range(b, f, l)*: returns *true* if all f-tuples of buffer b have an msn in the integer range $[f, l]$, both inclusive; (iii) *fsns_in_range2(b, f1, l1, f2, l2)*: returns *true* if each f-tuple of buffer b has an fsn in the integer range $[f1, l1]$, both inclusive, or in the integer range $[f2, l2]$, both inclusive; (iv) *msns_in_range2(b, f1, l1, f2, l2)*: returns *true* if each f-tuple of buffer b has an msn in the

integer range $[f1, l1]$, both inclusive, or in the integer range $[f2, l2]$; (v) *has_fsn(b, f)*: returns *true* if there is an f-tuple with fsn f in the f-tuple buffer b; (vi) *has_msn_fbuf(b, m)*: returns *true* if there is an f-tuple with msn m in the f-tuple buffer b; (vii) *count_fsn(b, f)*: returns the number of f-tuples with fsn f stored in the f-tuple buffer b; (viii) *is_sorted_by_fsn_fbuf(b)*: returns *true* if the f-tuples of f-tuple buffer b are sorted by increasing fsns; (ix) *is_sorted_by_msn_fbuf(b)*: returns *true* if the f-tuples of f-tuple buffer b are sorted by increasing msns.

The queries also use an observer automaton `obs` that has a single edge from its initial location to a location `finished`. That automaton transitions to `finished` once the model has finished modeling the transmission and reception of all frames, and the execution of the driver. Each modeled controller has an abstract data type (ADT) that encapsulates the buffers of the given controller. There are two such ADTs, which are stored in the `ctrls` array and are accessed by the constants `CTRL_0` and `CTRL_1`. Finally, the boolean array `d.is_trusted` indicates for each controller whether it has alerted of its failure or not.

C. Queries

To verify the properties of Section V some preliminary queries were used to check that the elements of the different buffers of the model can only have values within a specific range. As an example, for the `cha.rxbuf` buffer and the `ctrls[CTRL_1].rx_fbuf` buffer the queries are the following:

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

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

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

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

Respectively, `FIRST_TX_FSN` and `LAST_TX_FSN` are the fsn of the first and last f-tuple generated by the controllers; `FIRST_TX_MSN` and `LAST_TX_MSN` are the first and last msn generated by app; `FIRST_RX_FSN` and `LAST_RX_FSN` are the fsn of the first and last f-tuple generated by the channel; and `FIRST_RX_MSN` and `LAST_RX_MSN` are the msn of the first and last f-tuple generated by the channel. The values of these constants are such that, for each pair, the range of integers in-between does not overlap with any other pair of these constants. Moreover, they are such that the number of messages (msns) to be mtx-requested by the app and the number of frames (f-tuples) to be transmitted by the channel are both 3. This is so since with a value of 1 the model checker will evolve to the set I_1 of all possible states reachable after an mtx-request and a reception. Thus, I_1 is the set of all possible states from which a new mtx-request or reception could occur. With a value of 3 the model checker will verify the queries for two mtx-requests and two receptions from all states in I_1, i.e., from all possible initial states. Note that two additional mtx-requests and receptions are required to verify the properties related to the relative order between msns and fsns.

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

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

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

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

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

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

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

P4: Since by hypothesis one controller is always correct and has a correct link, all f-tuples generated by the channel are received by at least one controller. This makes the verification of this property straightforward:

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

P5: This is already proved by one of the in-range queries. Specifically, the following query proves that all f-tuples stored in `app.rx_fbuf` originated from the channel, and were therefore not self-received.

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

P6: Since the f-tuples generated by the channel have monotonically increasing integers as fsns, and f-tuples are appended to `app.rx_fbuf` in sequence, the following query proves P6.

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

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

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

P8: This property can be verified with the following query:

```
A◇ obs.finished
```

P9: This query follows a similar logic to the one of P6:

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

P10: The automaton modeling the application has two locations: `idle` and `wait_tx_success`. It can only transition from the `idle` location to the `wait_tx_success` location if an mtx-request occurs. Similarly, it can only transition from `wait_tx_success` back to `idle` if an mtx-request is satisfied, i.e. when a boolean variable indicates a tx-success. Moreover, from the query of P7, it is known that the automaton modeling the application cycles to the `wait_tx_success` for every modeled mtx-request. Thus, it must only be shown that the automaton does not remain indefinitely in `wait_tx_success`. This is accomplished with the following query:

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

VIII. Conclusion

The paper presents the design and verification of reCANdrv, a media redundancy management driver for the nodes of a ReCANcentrate network. ReCANcentrate is a CAN-compliant replicated star topology with enhanced error-containment and fault-tolerance mechanisms. The goal of reCANdrv is to provide a correct redundancy management, i.e., to transparently and properly manage for the application the available media redundancy under faults. Since ReCANcentrate has been designed for systems that require high reliability, it must be verified that this goal is achieved. To this end, the goal is formalized as a series of properties based on a few realistic assumptions. The properties are then verified by means of model checking. For this, a model of a ReCANcentrate node is created using UPPAAL. The model is presented in an abstract way ([44] contains a description of the implementation details). Afterwards a series of UPPAAL queries are presented. These show that the model indeed satisfies the properties.

The driver provides a virtual CAN controller interface to the application executing on the nodes. Thus, it hides the underlying replicated communication medium and the treatment of faults from the application. It is thus possible to execute with increased reliability standard CAN applications and higher-layer protocols based on CAN on ReCANcentrate nodes.

Furthermore, since the only requirement put on the channel is that it provides a single logical broadcast domain, the ReCANcentrate node architecture and reCANdrv can also be used in networks relying on other topologies as long as they provide a single logical broadcast CAN domain, e.g., a replicated bus topology where the two buses are coupled.

Moreover, the results of this paper allow to use model checking to verify the properties of this kind of reCANdrv-based networks. It will be easy to incorporate reCANdrv-based nodes into any other model without having to specify their low-level mechanisms, e.g. the driver, CAN controllers, timer, etc., but just their properties. In this sense, the work presented here is coherent with the common practice of abstracting away the lower level mechanisms that substantiate certain properties and just model the properties themselves.

Another contribution of the model presented here is that some parts can easily be used to verify other systems, e.g., the approach used to model interrupts and ISRs, described in detail in [44], can be used for the verification of other systems that use these mechanisms. Also, the approach of verifying the properties provided by reCANdrv by considering how modeled messages are passed between different buffers can be used to verify the properties of other distributed systems.

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

Acknowledgment

This work was supported by the Spanish Science and Innovation Ministry with grant DPI2008-02195, by the Spanish Economy and Competitivity Ministry with grant DPI2011-22992, and by FEDER funding.

References

[1] *ISO 11898 - Road Vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication*, ISO Std.

[2] H. Zeltwanger, "Controller Area Network — introduced 25 years ago," *CAN Newsletter*, pp. 18–20, March 2011.

[3] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in Automotive Communication Systems," *Proc. of the IEEE*, vol. 93, no. 6, 2005.

[4] J. Pimentel, J. Proenza, L. Almeida, G. Rodríguez-Navas, M. Barranco, and J. Ferreira, *Dependable Automotive CAN Networks*. Handbook on Automotive Embedded Systems. CRC Press. Edited by Nicolas Navet and Françoise Simonot-Lion, 2009.

[5] R. A. Gupta and M.-Y. Chow, "Networked Control System: Overview and Research Trends," *IEEE Trans. on Industrial Electronics*, vol. 57, no. 7, pp. 2527–2535, Jul. 2010.

[6] L.-B. Fredriksson, "CAN for critical embedded automotive networks," *IEEE Micro*, vol. 22, no. 4, pp. 28–35, Jul. 2002.

[7] J. Ferreira, L. Almeida, J. Fonseca, P. Pedreiras, E. Martins, G. Rodriguez-Navas, J. Rigo, and J. Proenza, "Combining Operational Flexibility and Dependability in FTT-CAN," *IEEE Trans. on Industrial Informatics*, vol. 2, no. 2, pp. 95–102, May 2006.

[8] G. Buja, J. R. Pimentel, and A. Zuccollo, "Overcoming Babbling-Idiot Failures in CAN Networks: A Simple and Effective Bus Guardian Solution for the FlexCAN Architecture," *IEEE Trans. on Industrial Informatics*, vol. 3, no. 3, pp. 225–233, 2007.

[9] M. Short and M. J. Pont, "Fault-Tolerant Time-Triggered Communication Using CAN," *IEEE Trans. on Industrial Informatics*, vol. 3, no. 2, pp. 131–142, May 2007.

[10] B. Hall, M. Paulitsch, K. Driscoll, and H. Sivencrona, "ESCAPE CAN limitations," *SAE Trans. J. Passenger Cars - Electron. Elect. Syst*, vol. 116, pp. 422–429, 2008.

[11] G. Rodriguez-Navas, "Orthogonal, Fault-Tolerant, and High-Precision Clock Synchronization for the Controller Area Network," *IEEE Trans. on Industrial Informatics*, vol. 4, no. 2, pp. 92–101, 2008.

[12] F. Gil-Castineira, F. Gonzalez-Castano, and L. Franck, "Extending vehicular CAN fieldbuses with delay-tolerant networks," *IEEE Trans. on Industrial Electronics*, vol. 55, no. 9, pp. 3307–3314, 2008.

[13] H. Zeng, M. D. Natale, P. Giusto, and A. Sangiovanni-Vincentelli, "Stochastic Analysis of CAN-Based Real-Time Automotive Systems," *IEEE Trans. on Industrial Informatics*, vol. 5, no. 4, pp. 388–401, 2009.

[14] T. Herpel, K.-S. Hielscher, U. Klehmet, and R. German, "Stochastic and deterministic performance evaluation of automotive CAN communication," *Computer Networks*, vol. 53, no. 8, pp. 1171–1185, Jun. 2009.

[15] M. Nahas, M. J. Pont, and M. Short, "Reducing message-length variations in resource-constrained embedded systems implemented using the Controller Area Network (CAN) protocol," *Journal of Systems Architecture*, vol. 55, no. 5-6, pp. 344–354, May 2009.

[16] P. Martí, A. Camacho, M. Velasco, and M. El Mongi Ben Gaid, "Runtime Allocation of Optional Control Jobs to a Set of CAN-Based Networked Control Systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 4, pp. 503–520, 2010.

[17] H. Zeng, M. D. Natale, P. Giusto, and A. Sangiovanni-Vincentelli, "Using Statistical Methods to Compute the Probability Distribution of Message Response Time in Controller Area Network," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 4, pp. 678–691, 2010.

[18] T. Hoppe, S. Kiltz, and J. Dittmann, "Security threats to automotive CAN networks—Practical examples and selected short-term countermeasures," *Reliability Engineering & System Safety*, vol. 96, no. 1, pp. 11–25, Jan. 2011.

[19] A. Monot, N. Navet, and B. Bavoux, "Impact of clock drifts on CAN frame response time distributions," in *16th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, 2011.

[20] M. Barranco, J. Proenza, and L. Almeida, "Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate," *Computer*, vol. 42, no. 5, pp. 66–73, May 2009.

[21] M. Barranco, J. Proenza, G. Rodríguez-Navas, and L. Almeida, "An active star topology for improving fault confinement in CAN networks," *IEEE Trans. on Industrial Informatics*, vol. 2, no. 2, pp. 78–85, May 2006.

[22] M. Barranco, J. Proenza, and L. Almeida, "Quantitative Comparison of the Error-Containment Capabilities of a Bus and a Star Topology in CAN Networks," *IEEE Trans. on Industrial Electronics*, vol. 58, no. 3, pp. 802–813, Mar. 2011.

[23] M. Barranco, J. Proenza, and L. Almeida, "Reliability improvement achievable in CAN-based systems by means of the ReCANcentrate replicated star topology," in 8^{th} *IEEE Int. Workshop on Factory Communication Systems*, May 2010, pp. 99–108.

[24] M. Barranco and J. Proenza, "Towards Understanding the Sensitivity of the Reliability Achievable by Simplex and Replicated Star Topologies in CAN," in *16th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, 2011.

[25] Robert Bosch GmbH, "CAN specification version 2.0," 1991.

[26] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues, "Fault-tolerant broadcasts in CAN," in *FTCS '98: Proc. of the The 28th Annual Int. Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1998, p. 150.

[27] J. Proenza and J. Miro-Julia, "MajorCAN: A modification to the Controller Area Network protocol to achieve atomic broadcast," *IEEE Int. Workshop on Group Communication and Computations, Taipei*, 2000.

[28] M. Barranco, D. Gessner, J. Proenza, and L. Almeida, "First prototype and experimental assessment of media management in ReCANcentrate," in *ETFA 2010. 15th IEEE Int. Conf. on Emerging Technologies and Factory Automation, Bilbao, Spain*, 2010.

[29] G. Cengic and K. Akesson, "On Formal Analysis of IEC 61499 Applications, Part A: Modeling," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 2, pp. 136–144, May 2010.

[30] G. Cengic and K. Akesson, "On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 2, pp. 145–154, May 2010.

[31] D. Herrero-Perez and H. Martinez-Barbera, "Modeling Distributed Transportation Systems Composed of Flexible Automated Guided Vehicles in Flexible Manufacturing Systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 2, pp. 166–180, May 2010.

[32] D. Cancila, R. Passerone, T. Vardanega, and M. Panunzio, "Toward Correctness in the Specification and Handling of Non-Functional Attributes of High-Integrity Real-Time Embedded Systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 2, pp. 181–194, May 2010.

[33] D. Gomez-Gutierrez, G. Ramirez-Prado, A. Ramirez-Trevio, and J. Ruiz-Leon, "Observability of Switched Linear Systems," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 2, pp. 127–135, May 2010.

[34] M. Kloetzer, C. Mahulea, C. Belta, and M. Silva, "An Automated Framework for Formal Verification of Timed Continuous Petri Nets," *IEEE Trans. on Industrial Informatics*, vol. 6, no. 3, pp. 460–471, Aug. 2010.

[35] R. Davis, S. Kollmann, and V. Pollex, "Controller Area Network (CAN) schedulability analysis with FIFO queues," *Proc. of the 2011 Euromicro Conf. on Real-Time Systems (ECRTS 2011)*, 2011.

[36] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, pp. 239–272, 2007.

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

[38] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on Uppaal," November 2004.

[39] J. Krakora, L. Waszniowski, P. Pisa, and Z. Hanzalek, "Timed automata approach to real time distributed system verification," *5th IEEE Int. Workshop on Factory Communication Systems.*, pp. 407–410, 2004.

[40] G. Rodriguez-Navas, J. Proenza, and H. Hansson, "Using UPPAAL to Model and Verify a Clock Synchronization Protocol for the Controller Area Network," *10th IEEE Conf. on Emerging Technologies and Factory Automation*, vol. 2, pp. 495–502, 2005.

[41] G. Rodriguez-Navas, J. Proenza, and H. Hansson, "An UPPAAL Model for Formal Verification of Master/Slave Clock Synchronization over the Controller Area Network," *6th IEEE Int. Workshop on Factory Communication Systems*, pp. 3–12, 2006.

[42] M. Bonet, G. Donaire, and J. Proenza, "Modelling MajorCAN with UPPAAL," in *12th IEEE Conf. on Emerging Technologies and Factory Automation*. IEEE, 2007, pp. 1404–1407.

[43] G. Leen and D. Heffernan, "Modeling and Verification of a Time-triggered Networking Protocol," *Int. Conf. on Networking, Int. Conf. on Systems and Int. Conf. on Mobile Communications and Learning Technologies (ICNICONSMCL'06)*, vol. 2, no. 1, pp. 178–178, 2006.

[44] D. Gessner, "Design and verification by means of model checking of reCANdrv: a media redundancy management driver for the nodes of a ReCANcentrate network," Master's thesis, Universitat de les Illes Balears, 2011. [Online]. Available: http://srv.uib.es/ref/1259.html

[45] "ReCANdrv UPPAAL model file," 2011. [Online]. Available: http://dmi.uib.es/jproenza/archive/UppaalModelReCANdrv/reCANdrv.zip

David Gessner received the first degree in informatics engineering and a master in communication and information technologies from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2010 and 2011, respectively.

He is currently pursuing the Ph.D. degree in computer science at the UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, dependable communication topologies, and field-bus networks such as CAN.

Manuel Barranco received the first degree in informatics engineering and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 2003 and 2010, respectively.

He is currently a lecturer in the Department of Mathematics and Informatics at UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, clock synchronization, dependable communication topologies, and field-bus networks such as CAN.

Julián Proenza received the first degree in physics and the doctorate in informatics from the University of the Balearic Islands (UIB), Palma de Mallorca, Spain, in 1989 and 2007, respectively.

He is currently holding a permanent position as a lecturer in the Department of Mathematics and Informatics at UIB. His research interests include dependable and real-time systems, fault-tolerant distributed systems, clock synchronization, dependable communication topologies, and field-bus networks such as CAN.

Dr. Proenza is a Member of the IEEE Industrial Electronics Society