Modelling MajorCAN with UPPAAL

Matias Bonet, Gabriel Donaire and Julián Proenza
Dept. de Matemàtiques i Informàtica, Universitat de les Illes Balears, Palma de Mallorca, SPAIN
matias.bonet@uib.es, gabriel.donaire@uib.es and julian.proenza@uib.es

Abstract

The Controller Area Network (CAN) protocol produces data inconsistencies in some scenarios. A previous work proposed a new protocol called MajorCAN which is a small modification to CAN. MajorCAN does not present the reported error scenarios thus ensuring data consistency. Although MajorCAN has been thoroughly simulated, no formal verification has been performed so far. In this paper we describe how we have modelled MajorCAN using a network of timed automata in UPPAAL. *This is the first step of its formal verification by means of model checking.*

1. Introduction

The Controller Area Network (CAN) protocol, a field-bus first developed for automotive applications, is widely used in the automation industry as well. The main reason for its success is its real-time and dependable behaviour. Among the dependability properties, the specification of this standard claims that CAN presents *data consistency*. This means that within a CAN network it is guaranteed that a frame is either simultaneously accepted by all nodes or by none. Besides the existence of the *error passive state* [3] in which this property does not hold, Rufino *et al.* identified [3] some specific scenarios in which some nodes receive a frame and some others do not. The same authors proposed a set of protocols to be executed on top of CAN to solve the problem [3]. In a later analysis new scenarios of inconsistent communication were identified in which both CAN and the proposed higher-layer protocols fail [2]. In order to cope with all these scenarios a modification to the CAN protocol called MajorCAN was proposed [2].

In order to illustrate the scenarios of Rufino *et al.* let us consider the case in Fig. 1. A disturbance corrupts the last but one bit of the *End Of Frame* field (EOF) of the set of nodes called X. In the next bit, these receivers start the transmission of an error flag. The *dominant* first bit of this error flag is seen by the nodes belonging to set Y and by the transmitter as an error in the last bit of their EOF. The nodes belonging to X will reject the frame, the nodes belonging to Y will accept the frame following the CAN's last bit rule, and the transmitter will schedule the frame retransmission. If the

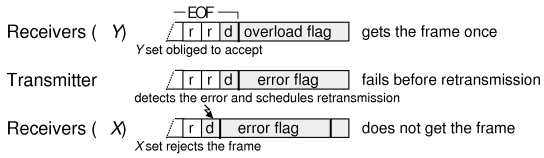

Figure 1. Inconsistency scenarios in CAN [3]

transmitter suffers a hardware failure that prevents it from completing the retransmission, the nodes belonging to Y receive the frame whereas those of X do not. Another scenario [2] which has a higher probability of occurrence happens if the transmitter can not see the error flag in the last bit of EOF due to an additional disturbance in that bit. In this case, the transmitter does not even try to retransmit the frame and the same inconsistent reception takes place.

The MajorCAN protocol [2] does not exhibit this kind of scenarios and it is designed to ensure that for each frame all nodes agree on whether to accept it or not. We aim at formally verifying that MajorCAN works properly by using the model checker UPPAAL [1]. The first step, which is to model the protocol, is described in this paper.

2. MajorCAN$_m$ protocol description

The MajorCAN protocol [2] is designed to ensure data consistency in the presence of up to m erroneous bits per frame. For this reason the notation MajorCAN$_m$ is used, where m has to be substituted by a specific value in each instantiation of the protocol. The following explanation uses m as a parameter to achieve the maximum generality.

Both in CAN and in MajorCAN, the EOF contains no relevant data. If a frame contains errors only in the bits of the EOF, it is a correct frame and could be accepted. Whether it is accepted or not is just a matter of agreement among the different nodes, this is what MajorCAN has to do. In contrast, if errors are in bits previous to the EOF the frame must be rejected. According to the CAN specification whenever a CRC error is detected, transmission of an error flag starts at the bit following the ACK delimiter, that is at the first bit of the EOF. Since a frame with a CRC error is clearly erroneous, its consistent rejection has to be ensured. Up to $m-1$ additional errors in the first bits of the EOF may delay the detection of said error flag by some nodes. Therefore a node

 doi:10.1109/EFTA.2007.4416948

first detecting an error in the $(m+1)$th bit of EOF or later can be sure that it was not caused by a CRC (or previous) error and thus it can accept the frame. In contrast, a node first detecting an error in the bits between the first one and the mth should make its decision based on what the others are doing. In order to determine what the others are doing, this node will sample the bit slots where only a node having detected the error in the $(m+1)$th bit of EOF or later would send its error flag. Aimed at doing this sampling able to tolerate the $m-1$ additional errors that may still occur, the node that accepts the frame signals this with an extended error flag, and the ones sampling the corresponding bits check $2m-1$ bits and perform a majority vote on these values.

More specifically, the MajorCAN EOF field will be divided in two subfields: a first one of m bits and a second one from the $(m+1)$th to the $(3m-3)$th bit (see [2] for a more detailed description of MajorCAN). A node detecting an error in the second subfield of EOF accepts the frame and notifies the acceptance with an error flag extended up to the $(3m+5)$th bit. In contrast, a node detecting an error in the first subfield must sample from the $(m+7)$th to the $(3m+5)$th bit, and perform a majority voting on these samples to see if the other nodes are accepting the frame. It is important to remark that if any node detects its second error during the bits corresponding to the EOF and the extended error flags, this is not signaled with any additional error flag. Otherwise error flags of second errors could spoil the agreement process.

Fig. 2 shows the behaviour of a MajorCAN$_5$ node when the first error is detected in different bits of EOF. Vertical arrows indicate the bits where the sampling is performed.

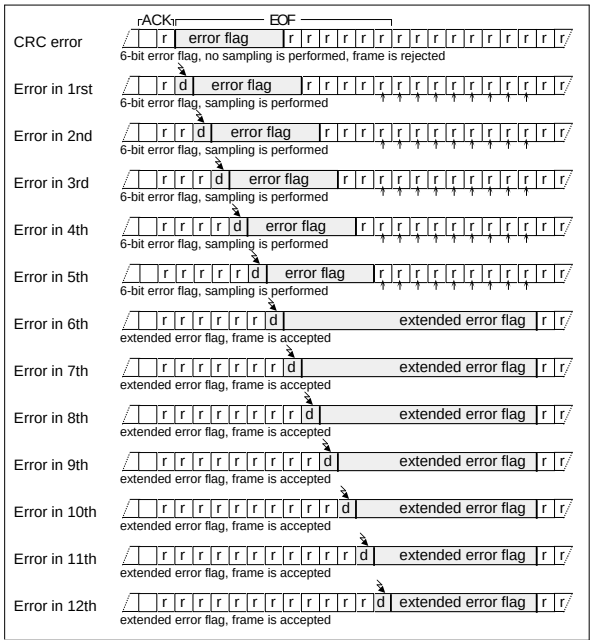

Figure 2. Behaviour of a MajorCAN$_5$ Node

3. UPPAAL modelling features

An UPPAAL [1] model is built as a set of concurrent processes. Each process is graphically designed as a timed-automaton. This automaton is represented as a graph which has *locations* as nodes and *transitions* as arcs between locations. The firing of each transition can depend on a *guard* and/or on a *synchronization*. A guard is an expression which uses the variables and clocks of the model in order to indicate when the transition is enabled, i.e. may be fired. The synchronization is the basic mechanism used in UPPAAL to coordinate the action of two or more different processes.

We have used two types of synchronizations. The first type is *binary synchronization* that uses normal channels declared as, e.g., `chan a`. When a process is in a location from which there is a transition labeled with `a!` the only way for the transition to be enabled is that another process is in a location from which there is a transition labeled with `a?` and viceversa. If at a specific instant there are several possible ways to have a pair `a!` and `a?`, one of them is non-deterministically chosen during model checking. And the second type is *broadcast synchronization* that uses broadcast channels declared as, for example, `broadcast chan b`. When one process is in a location from which there is a transition labeled with `b!` and one or more processes are in locations from which there is a transition labeled with `b?` all these transitions are enabled and if triggered, they are triggered all at the same time. However, if there are no processes in locations from which there is a transition labeled with `b?`, the transition labeled with `b!` is enabled anyway.

Locations in UPPAAL can be of three different types. Among them we are just going to describe two, which are the types used in our model; normal locations and committed locations. Normal locations do not require any specific explanation since they correspond to the behaviour described so far. In contrast committed locations are such that if a model has one or more active committed locations, no transitions other than those leaving said locations can be enabled and time may not pass until all active committed locations are abandoned. A committed location is differentiated from a normal location with a "C" in the center of the circle used for representing it. The last feature of UPPAAL is the concept of template. Any of the various processes (automata) constituting the complete model can be designed using a template. A template is an automaton definition which has a set of parameters that can be of any of the data types accepted by UPPAAL. This is particularly useful for modelling distributed systems like ours, since the same template can be used as a basis for defining each and everyone of the system's nodes.

4. A model for MajorCAN

Our model for MajorCAN is a network of timed automata that consists of three parts: first, an arbitrary number of instantiations of a template called `Node`, each one modelling the behaviour of a single MajorCAN node; second, an automaton called `bit_st` that forces all `Node` instantiations to exchange each bit in a synchronized fashion; and third, an automaton called `Error_Counter` that keeps the number of erroneous bits per frame under the maximum number m that MajorCAN$_m$ must be able to cope with.

4.1. Bit synchronization

As it happens in CAN, MajorCAN$_m$ nodes resynchronize each time a new bit is transmitted. The automaton called `bit_st`, which is depicted in Fig. 3, is intended to model this resynchronization. The aim is not to model the details of the CAN (or MajorCAN) resynchronization, but to provide a basic mechanism for the different nodes in our model to, first, evolve bit by bit in a synchronized manner as it would happen in a real network, second, decide the value of the bus during each bit period and third, halt the complete model evolution once the maximum number of bits has been transmitted.

In order to perform the last function, `bit_st` uses the clock `t` to limit the duration of each single bit and the variable `bit` to count the number of transmitted bits. From the initially active location, which is `time_bit`, `bit_st` evolves to location `end_time` to halt the model evolution as soon as `bit` reaches its maximum value. In contrast, while said value is not reached, `bit_st` evolves to the committed location `bit_transition` when the clock `t` indicates that the duration of a bit (`T_BIT`) has elapsed. In this transition, `t` is reset in order to start counting the duration of the next bit, `bit` is increased and the global variable `bus` that models the value of the bus in that moment is preset to a *recessive* value (logical '1'). From the committed location `bit_transition`, `bit_st` evolves immediately to the initial location generating the broadcast synchronization `next!`. This will be received by all instantiations of `Node` that will use this synchronization to evolve simultaneously and to decide the final value of `bus` for the current bit.

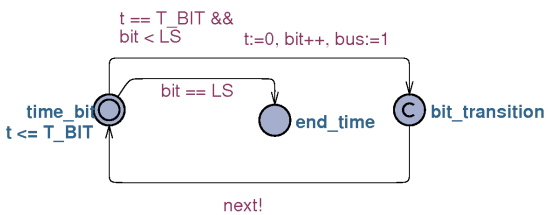

Figure 3. The `bit_st` automaton

4.2. The model for a node

A number of decisions have been made to simplify the modelling of the nodes. First, given that the operations that a node has to perform in MajorCAN in order to decide acceptation or rejection of each frame are the same for transmitter and receivers, we have decided to model both kinds of nodes with the same template (`Node`). Second, MajorCAN is supposed to ensure consistency for each transmitted frame, therefore the modelling of the frame retransmission is not required in order to prove data consistency. Note that for this reason the automaton `bit_st` stops generating `next!` when the bit counter reaches `LS`, which is the Last Sampling bit. And third, all the problems that MajorCAN solves are concentrated in the last bits of the frame. Thus, the rest of the frame's bits have not been modelled.

In Fig. 4, the resulting `Node` template can be seen. The first modelled bit of the frame is the ACK delimiter. In case an error is detected by any node in this bit or in a previous one by means of the CRC, said node has to start transmitting an error flag in the next bit, which is the first bit of EOF. The transition from the `initial` location to `ack_delimiter` models the transition to the ACK delimiter bit. Once in this location, `Node` may nondeterministically decide whether it has not detected an error in this bit or in the CRC (and thus it evolves to location `EOF`) or it has detected one (and thus it evolves to `error_flag_ACK`). In the second case, `Node` generates the binary synchronization `error!` for the error counter (see Section 4.3) to be able to count the number of erroneous bits. From location `error_flag_ACK`, `Node` sends a regular error flag updating `bus` to zero for `EFL` (Error Flag Length) consecutive bits and then irreversibly evolves to `reject_frame`.

In contrast, from location `EOF` each time a new bit has to be processed (`next?` is received), several possibilities have to be taken into account. On the one hand, `Node` may see the value '1' in the bus without suffering an error or, having `bus` the value '0', suffer an error that makes it see the value '1'. In both cases, `Node` goes back to location `EOF` and if it reaches the maximum number of bits of the EOF (`EOFL`) it irreversibly evolves to location `accept_frame`. And on the other hand, `Node` may also see the value '0' in the bus without suffering an error or, having `bus` the value '1', suffer an error that makes it see the value '0'. In both cases, `Node` evolves to location `erroneous_bit`.

From `erroneous_bit`, in case the error was detected after the `M` first bits of the EOF, `Node` transmits an extended error flag and later evolves to `accept_frame`. In contrast, if the error was detected in any one of the first `M` bits, `Node` transmits a regular error flag and then evolves to location `Sampling`. As soon as the bit counter reaches the first bit that has to be sampled (`FS`), `Node` increases the counter of dominant bits (`VAC`) each time it sees a '0' (either because there is a '0' in the bus or because there is a '1' but an error

locally changes this value) and increases the counter of recessive bits (VRC) each time it sees a '1' (again, either because there is a '1' or because there is a '0' but an error locally changes it). When the last bit (bit==LS) has been sampled, Node evolves to accept_frame if more dominant bits have been sampled or to reject_frame otherwise.

4.3. The error counter

As indicated at the beginning of Section 4, the goal of the Error_Counter automaton depicted in Fig. 5 is to limit the number of erroneous bits in a frame. From the description of the Node template provided in Section 4.2, it is clear that each time a node suffers an error that changes its view on the bus value, it notifies it by means of the error! binary synchronization. So each time a new bit is transmitted (next! is generated by bit_st), Error_Counter evolves to location error_bit. In case an error affects this bit in any node and the maximum number of errors (MAXE=M) has not been reached, the error count (EC) is increased but only when the next bit is transmitted (next?). Thereby if more than one node sees an error in this bit, the error count is increased only once. Note in Fig. 4 that when EC has reached its maximum value, the Node automaton is unable to generate more errors.

5. Conclusion and future work

With the final goal of demonstrating that MajorCAN$_m$ ensures data consistency at frame level in the presence of up to m erroneous bits, we have developed a set of automata to model the relevant features of the protocol behaviour. We have already performed a thorough simulation and debugging of the resultant model using the UPPAAL simulator, and we are planning to complete the model checking using the UPPAAL query language.

6. Acknowledgement

This work is partially supported by DPI 2005-09001-C03-02 and FEDER funding.

References

[1] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[2] J. Proenza and J. Miro-Julia. MajorCAN: A modification to the Controller Area Network protocol to achieve Atomic Broadcast. In *Proceedings of the IEEE Int. Workshop on Group Communications and Computations. IWGCC. Taipei, Taiwan*, April 2000.

[3] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcast in CAN. In *Proceedings of the IEEE 28th Int. Symp. Fault-Tolerant Computing. FTCS-28. Munich (Germany)*, June 1998.

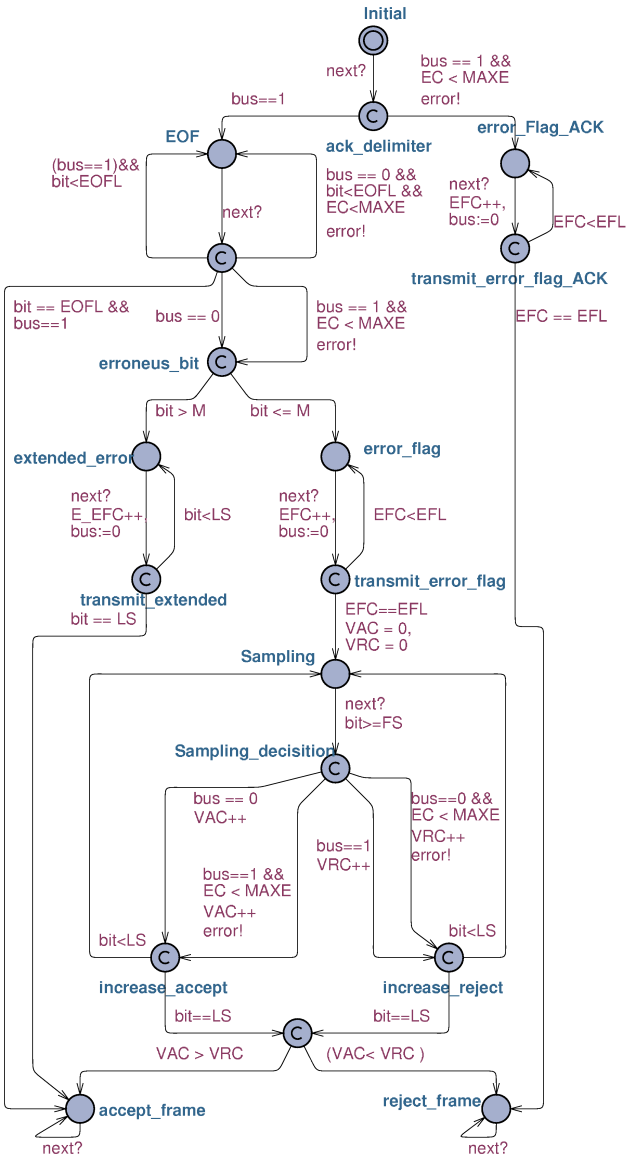

Figure 4. The Node template

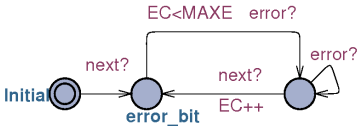

Figure 5. The Error_Counter automaton