Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 309 (2014) 31–49

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Modeling and Verification of CAN Bus with Application Layer using UPPAAL

Can Pan, Jian Guo,[1] Longfei Zhu

Software/Hardware Co-design Engineering Research Center
East China Normal University, Shanghai, China

Jianqi Shi

School of Computing, National University of Singapore, Singapore

Huibiao Zhu

Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai, China

Xinyun Zhou

State Key Laboratory of Information Security
Institute of Information Engineering, Chinese Academy of Sciences, China

Abstract

Controller Area Network (CAN) is a high-speed serial bus system with real-time capability. In this paper, we present a formal model of the CAN bus protocol, mainly focusing on the arbitration process, transmission process, and fault confinement mechanism. Moreover, 11 important properties are formalized in terms of the protocol. Based on the verification tool UPPAAL, we describe the system model and properties for performing verification work of the CAN bus protocol. The verification results indicate that some properties are not satisfied in CAN bus system, most of which are caused by the starvation and bus-off nodes. On this basis, the dynamic priority scheduling algorithm and bus-off recovery mechanism are applied, which indicates that some problems can be solved on the application layer.

Keywords: CAN bus protocol, Timed automata, Fault confinement, Application layer

1 Introduction

The Controller Area Network (CAN) is a high-speed serial bus system with real-time capabilities, widely used in embedded systems. CAN bus was first developed

[1] Corresponding author: jguo@sei.ecnu.edu.cn

http://dx.doi.org/10.1016/j.entcs.2014.12.004

by Bosch company [4] and then established as the international standard in ISO 11898 [5]. Multi-master broadcasting is the key feature of this serial bus system. All nodes can transmit data when the bus is free, and the collision of multiple simultaneous transmissions is solved by priority-based arbitration algorithm. The identifier of a message resolves the priority. A transmitter broadcasts a message to all nodes and each node decides whether the data is relevant according to the identifier received. One main feature is fault confinement mechanism. Each controller of CAN detects errors and takes appropriate measures to guarantee the data consistency and reliability. The interframe space is another characteristic. Data and remote frames are separated from the previous frame by an interframe space, during which no node has access to the bus. CAN bus is widely used in safety critical automotive electronics due to its real-time capability, low cost and reliable error confining mechanism.

To ensure the correctness of CAN protocol, formal methods based on the rigorous mathematical theory could be an effective and practical approach. The adoption of formal notations with a defined mathematical meaning enables the model to be expressed with precision and unambiguity. The properties that the protocol should exhibit can be represented in mathematical framework as well. Thus the correctness can be checked via exploring all states and transitions or mathematical proof. There are also some automatic or interactive tools to facilitate the process like SPIN [2], NuSMV [12], UPPAAL [14],Coq [3] among many other excellent tools.

CAN follows the abstract Open System Interconnection (OSI) reference model, and its protocol mainly defines the data link layer. In the first place, we present a formal model of the CAN protocol and the verification results of its properties based on timed automata [17]. We employ UPPAAL to model the CAN system and implement not only the arbitration process and the transmission process, but also the fault confinement mechanism and the interframe space characteristic. Besides, 11 important properties extracted from the CAN standard are also verified in UPPAAL.

In a CAN system, the application layer is open for users to define explicit algorithms according to their requirements. In fact, most of the errors in the CAN protocol can be avoided by the effective algorithms on the application layer. To show how this works and get more accurate verification results, we also integrate the algorithms on application layer into our models. Among various scheduling algorithms, we choose the representative dynamic priority scheduling algorithm. It means a message has both a static priority and a dynamic priority, and the dynamic priority can be promoted according to the times it fails in the arbitration. The dynamic priorities are compared prior to the static priorities, which greatly shortens the response time. Also, to keep the system from being deadlock, the nodes becoming bus-off state are usually reset. The bus-off recovery mechanism is applied after introducing the model of application layer. The influences on the system performance and properties are discussed and analyzed as well.

The paper is organized as follow. In section 2, a brief introduction to CAN data link layer is presented. Section 3 models the whole structure of CAN bus protocol, and then shows the four submodels. Section 4 presents the properties and the verification results, including the analysis. Additionally, in section 5, we illustrate the modeling process of application layer and show the changes to the verification results. Comparison with related work and conclusion are given in section 6 and section 7, respectively.

2 CAN Bus Protocol

The CAN network follows the abstract Open System Interconnection (OSI) reference model. This section gives a brief introduction to the data link layer of CAN, in which the main services are implemented. We introduce frame formats, arbitration mechanism, fault confining mechanism, and the interframe space. In a CAN network, N nodes are connected to a serial bus, as illustrated in Fig. 1. Any node can start to transmit a message when the bus is idle.

2.1 Frame Formats

The frame is the transmission unit of data link layer. In the CAN protocol, there are four types of frames. The data frame and remote frame transmit messages while error frame and overload frame broadcast signals.

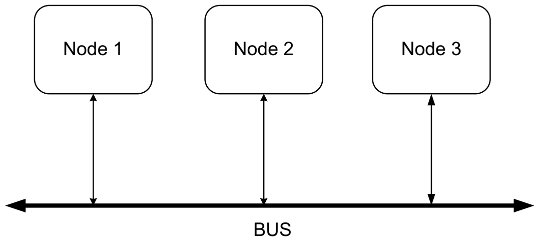

Fig. 1. CAN Bus Architecture

Data frame is the standard message format. As illustrated in Fig. 2, the SOF indicates the start of the frame. The arbitration field contains an identifier, and an RTR bit which indicates whether the message is a remote request. The control field specifies the data length, while the data field stores the actual content. A CRC sequence is contained in the CRC field to check the integrity of a received message. The ACK field acknowledges that the transmitted message is successfully received by at least one of the nodes. The 7-bit EOF marks the end of the frame. The remote frame is a message sent by any node to request another node to send a data frame with the identical identifier. It differs from data frame format in that the RTR bit marks as 0, and the data field is empty.

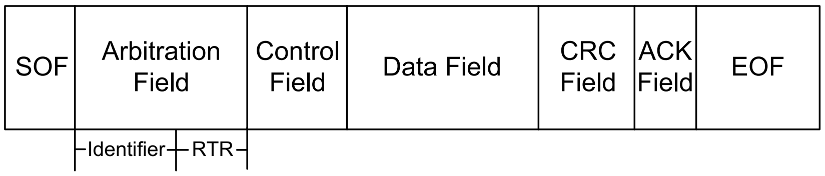

Fig. 2. Frame Format

The error frame is composed of two fields. The first is the error flag, which is 6 consecutive dominant (error-active node) or recessive (error-passive node) bits. The second is the error delimiter, with a fixed format of 8 consecutive recessive bits. But the recessive bits can be submerged by dominant bits sent to the bus at the same time. Any node can broadcast an overload message to delay the next transmission. Its format is the same as the active error frame, but the overload frame only occurs right after the last bit of EOF, while the error frame may occur whenever an error is detected.

2.2 Arbitration

The method used to solve the collision is the Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) mechanism. All nodes constantly monitor the bus in time. The nodes attempting to send messages start to write to the bus bit by bit when the bus is idle. The dominant bit (0) overwrites the recessive bit (1), which means zero has the higher priority. The data monitored on the bus is compared with the data sent by a transmitter. If one node sends a recessive bit but reads back a dominant bit, it means some other nodes are transmitting messages with higher priority identifiers. This node stops sending bits and immediately switches to listening-only mode.

2.3 Fault confinement

Messages with errors should be discarded and retransmitted when the bus turns to idle again. Fault confinement mechanism is proposed to stop a node which causes too many faults from developing into a permanent malfunction. A transmit error counter (TEC) and a receive error counter (REC) are assigned to every node. If the transmitter detects an error, the corresponding TEC increases, while the receiver detecting an error increases its REC. The RECs of all nodes receiving an error frame also increase. However, if a message is successfully transmitted and received, the TEC of the transmitter and the REC of receivers decrease. A node starts in the error-active state with its counters initialized to zero. If any TEC or REC reaches a certain value (128), the node enters the error-passive state, in which a node is unable to broadcast an error flag but writing to the bus is still possible if no error-active node wishes to write to the bus. If the TEC of a node reaches the maximum value (256), the node should be disconnected from the network. The node can return to the network only via a software reset.

2.4 Interframe space

Data frames and remote frames should be separated from preceding frames by a field called interframe space. But overload frames and error frames shall not be preceded by it. The interframe space contains the bit fields intermission (three recessive bits). But for error-passive nodes which have been the transmitter of the previous frames, eight recessive bits shall be sent following the intermission before trying to transmit its next message. However, the error-active node and the error-passive node that is not the transmitter of the previous message do not need to wait, which means these nodes may have already granted access during that period. The bus becomes idle after the interframe space, during which any node may access the bus.

3 Modeling

In our models, we use UPPAAL to model the specification of CAN bus protocol. In this section, we only model the CAN bus and its properties based on the ISO standard. However, to analyze its performance on the application layer, we further apply the dynamic priority algorithm and the bus-off recovery mechanism and verify the properties in the next section.

3.1 UPPAAL

UPPAAL is an integrated tool environment for modeling, simulation and verification of realtime systems [13]. It is not only suitable for automatic verification of safety and bounded liveness properties of networks of timed automata, but also appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. The simulator in UPPAAL is a validation tool which enables examination of possible dynamic executions of a system. The UPPAAL model checker can check invariant and reachability properties by exploring the state-space of a system.

Also, it is allowed to declare clock variables to record continuous time in UPPAAL. Four clock variables are introduced to respectively express the bits in the delay of overload and the interframe space after a successful transmission or the error frame.

3.2 Framework

Models can be divided into controller, arbitration, and transceiver, as shown in Fig. 3. There is only one controller in the whole system which is responsible for the synchronization of transactions between models, as well as the trigger of error frame and overload frame. The arbitration mainly deals with deciding the winner of the arbitration based on the message priority. And the transceiver is used to model the message sending and receiving, errors handling and the interframe space. There is one arbitration model and one transceiver model for every single node.

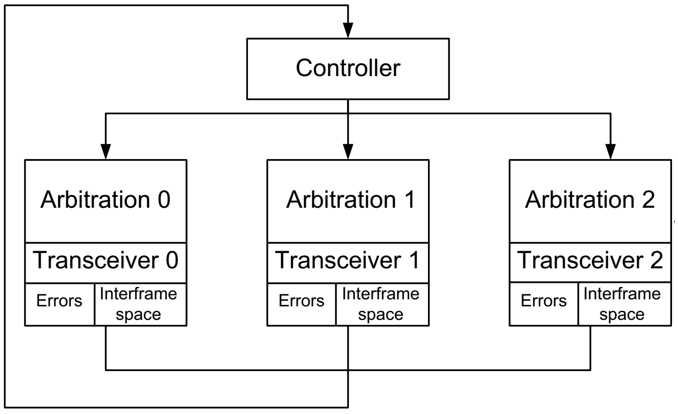

Fig. 3. The overall framework of all three models

In the controller model, the transaction processes are controlled and coordinated by variables and channels. During the phrase of reading and writing the message, an error frame may be generated if any error occurs. Moreover, the overload frame may be generated by nodes requesting a delay of the next transmission.

The arbitration solves the collision based on the message priority. Either data frame or remote frame can be randomly generated at first, then the identifiers of messages attempting to be transmitted will be compared. Once the winner is decided, the transmission process starts, which is synchronized by channels. Furthermore, the arbitration also deals with the problem of error-passive nodes trying to send the next message.

In the transceiver, the node winning the arbitration starts to send data to the bus, while others listen to the bus. When any error randomly occurs in any node, a signal will be sent to controller, which enforces an error frame to be broadcasted. All nodes receiving the error flag increase their error counters. Analogously, the overload frame will be broadcast before interframe space if any node requests it.

In this CAN model, there are three nodes named 0, 1, 2 which transmit messages with identifiers 1, 2, 3 respectively. In a practical CAN system, the data is sent bit by bit, and monitored at the same time. However, to abstract the process, the frame is simplified as an integer, representing the identifier of a frame. In addition, the bus is declared as an integer variable `bus`, and the writing and reading process of data is abstracted as the assignment to the `bus` variable and the reading from it.

3.3 Controller

The controller is shown in Fig. 4. This model synchronizes with the arbitration models and the transceiver models. The controller starts in the `idle` state, which means the bus is free. When a node is sending a message, the controller synchronizes with the arbitration by the channel `send[id]`. The controller waits until the comparison of message priorities starts and sends a synchronization signal (`start_arbitration!`).

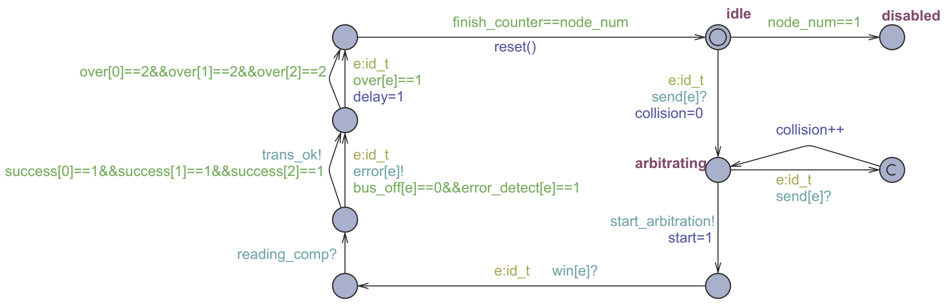

Fig. 4. Controller Model

During the reading process, one or more errors may occur. The controller triggers the channel `error[id]` as soon as any error is detected and the array `error_detect[id]` turns to 1. The occurrence of error, as well as the the broadcast of an error frame, is illustrated in the transceiver model. (See Section 3.5)

After the EOF of a frame, any node can send an overload frame to request a delay of next transmission. Once the array `over[id]` is set to 1, the global variable delay is set to 1 as well, indicating the broadcast to all nodes. (See section 3.5)

During a whole transmission process, an ideal situation is that a message is successfully sent and received by all nodes and all nodes return to the idle state. However, errors may occur during the transmitting process, after which an error signal is broadcast. In addition, an overload frame may be sent and broadcast so that all nodes wait for additional time before the start of next transmission process. After all these processes are completed, all nodes return to the idle state and prepare for the next transmission.

3.4 Arbitration Model

The arbitration process is implemented in the arbitration model of each node. The initial location is `idle`, which means that a node doesn't attempt to send any message. The variable `i` is in {1, 2} (e.g. `i:int[1, 2]`), donating which kind of frame will be sent. If i=1, this node sends a data frame, otherwise (i=2) it sends a remote frame. The synchronization with the controller is done by the channel `send[id]`, which means that the node `id` is sending a message with its corresponding identifier. As soon as the channel `start_arbitration` is triggered, all nodes pending for transmitting messages at the same time are competing, which is implemented in the function `compare()`. After the winner is decided, each node compares its own identifer with the winner's identifier. The node winning the arbitration jumps to `request_success` state, while others jump to `request_denied` state. All nodes return to the idle state once the transaction is finished, which is coordinated by the channel `trans_completed[id]`.

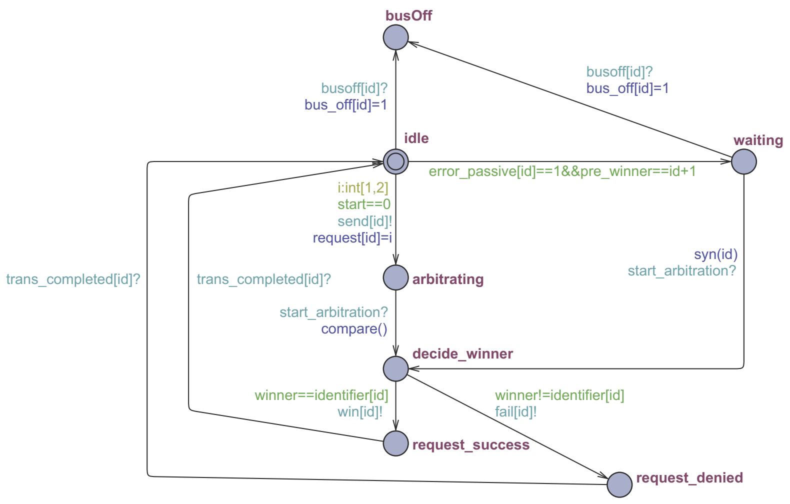

Fig. 5. Arbitration Model

In CAN system, it is specified that an error-passive node which is the transmitter of the previous message should wait for additional time during the interframe space before sending the next new message, during which other error-active nodes or error-passive node that is not the transmitter of previous message may gain access to the bus. In other words, this error-passive node can successfully transmit its next message only if no others are trying to transmit messages in this period. To illustrate this process, The array `error_passive[N]` is declared. The `error_passive[id]` is set to 1 when `node [id]` is in an error-passive state. In arbitration model, each node firstly checks whether it is in error-passive state and also happens to be the transmitter of the previous message. If so, the corresponding node jumps to the `waiting` state. If there are other nodes trying to transmit messages at the same time, the node in the `waiting` state fails to take part in the competition for bus after the completion of the interframe space. However, when this node is the only one trying to transmit, it grants the access to the bus. The procedure of arbitration is shown in Fig. 5.

Moreover, as mentioned in Section 2.3, every node has its own TEC (transmit error counter) and REC (receive error counter). To simplify these two error counters, in our models we declare only one variable error counter to represent them as a whole in order to reduce the system complexity and the number of reachable traces. It's worth mentioning that as the two error counters share the same characteristics, our abstraction is reasonable and has no side-effects on the verification results. The `error_counter` of a node is increased no matter the error is a transmitting error or a receiving error. When the error_counter of a node reaches the maximum limit, it turns to bus-off state.

In this arbitration model, the identifiers of nodes requesting to send messages are arbitrated to decide which node can transmit its data. Furthermore, if an error-passive node has sent a message, it should wait for additional time before continuously sending another message. Once the error counter of a node reaches the maximum limit, it skips to state `busOff` in this model as well.

3.5 Transceiver Model

The transceiver is shown in Fig. 6. After arbitration process, the sending and receiving process, as well as the error handling and interframe space are all implemented in the transceiver model of each node.

The node winning the arbitration starts to send data to the bus once the local channel `win[id]` is triggered, while others receive the message. In the local function `reading()`, all nodes, including the transmitter itself, respectively assign the value of `bus` to its own local variable `read`. If all nodes receive the message successfully, the channel `trans_ok` will be triggered, which means the end of frame. In addition, on receiving a remote frame, the node requested will prepare a data frame with the corresponding identifier and take part in the competition for bus access.

During the transmission, errors may occur. This situation is modeled by two alternative traces, through one of which data can be read successfully, while the other generates an error. Once `error_detect[id]` is set to 1, error signal is broadcast by the `error[id]` channel in `controller`. After an error flag is signaled, the value of bus is equal to the `error_frame`. All nodes detect the error signal on the bus and immediately discard the value of read variable, and at the same time the `error_counter`s of all nodes increase by 1. If the message is successfully sent and received, `error_counter`s of all nodes decrease on the other hand. It is worth mentioning that, if only the error-passive nodes detect the error while no error-active nodes do, the error cannot be known by others because an error-passive node is unable to broadcast an error flag. The fault confinement process can be illustrated in Fig. 7. As shown, all nodes are initialized from error-active state. As long as any node has detected an error, its error counter will be increased. And all nodes receiving the error frame increase their error counters too. On the other hand, if a frame is successfully transmitted, the error counter of the transmitter decreases (but not less than 0). Similarly, after a successful reception of a frame, the error counters of receivers decrease. The counters are usually increased or decreased by one, but sometimes by eight. The detailed principles are specified in the protocol. The error-active node will turn to error-passive state when its error counter reaches a certain value. However, one may return to error-active state when the error_counter drops within limit. If the error-counter finally reaches the maximum limit, the node will be bus-off and can neither send nor receive any message. In a real CAN system the bus-off node is able to return to the system only through user_request and 128 occurrences of 11 consecutive recessive bits.

In this model, the variable `error_counter` is declared to record the errors. The `error_counter`s of all nodes are initiated to 0. For every node, on receiving the error frame, the variable `error_counter` will increase by 1. However, if no error occurs, all nodes decrease their `error_counter`s in function `error_counter_decrease(id)`. If the `error_counter` reaches 4, the `error_passive[id]` is set to 1, turning to error-passive state. Of course, the `error_counter`s will decrease if a message is successfully sent and received. And the `error_passive[id]` can be set to 0 once the

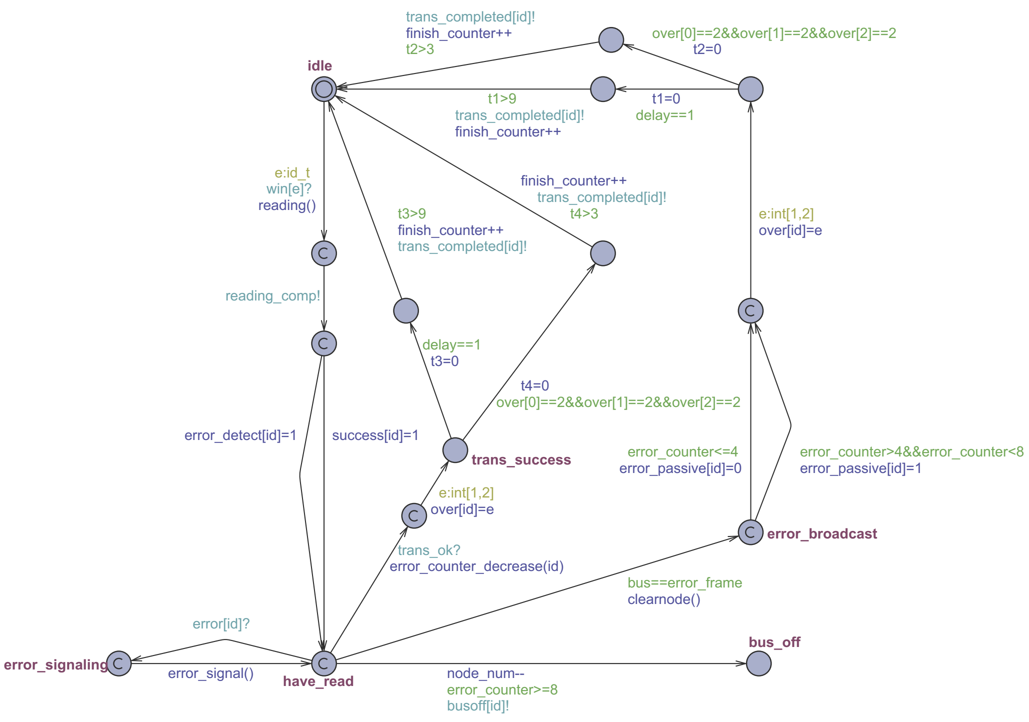

Fig. 6. Transceiver Model

error_counter is less than 4. But a node turns to bus-off state if its error_counter reaches 8. We assume the maximum limits for error-active nodes and error-passive nodes to be 4 and 8 respectively in order to simplify the models. For a bus-off node, the channel busoff[id] is triggered.

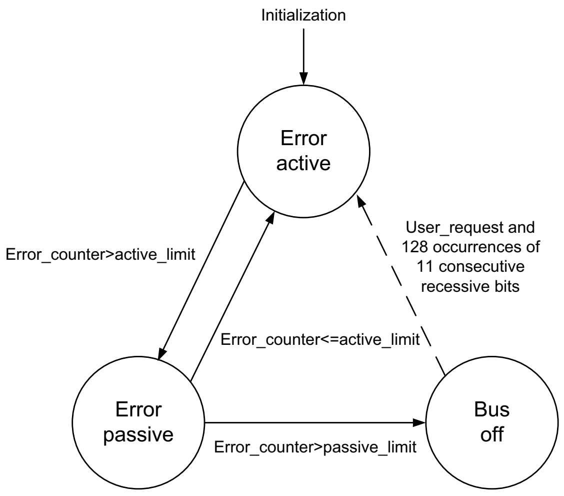
Initialization
Error active
Error_counter>active_limit
User_request and 128 occurrences of 11 consecutive recessive bits
Error_counter<=active_limit
Error passive
Bus off
Error_counter>passive_limit

Fig. 7. Fault Confinement

After the EOF, the node requesting a delay of the next frame should immediately send a overload frame next to it. On receiving the overload frame, all nodes should broadcast six dominant bits too, which means there can be 12 consecutive dominant bits. The overload delimiter follows the overload flags, consisting of eight recessive bits. After sending an overload flag, every node shall monitor the bus until it detects a recessive bit. At this point, all nodes realize that the overload flag is completed, and then start sending seven more recessive bits simultaneously, to complete the eight-bit-long overload delimiter. If no nodes need to send a overload frame, the EOF is followed by the interframe space. As shown in Fig. 8, the interframe space is a field used to separate data frame or remote frame from the preceding frames, consisting of intermission and bus idle. But the error-passive node, which is the transmitter of the previous transmission, should wait for additional suspension field (8 bits) before sending the next message. The process has been specified in arbitration.

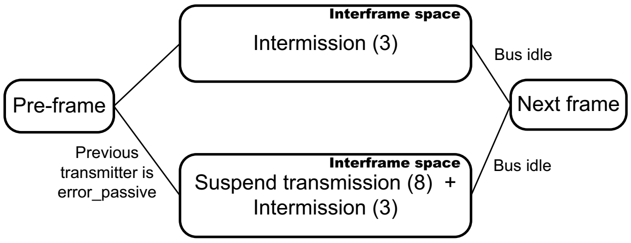

Fig. 8. Interframe Space

The model can randomly choose whether any node requests an overload frame before returning to `idle` state. Once the variable `delay` is set to 1, the overload frame (6 time units) will be produced and broadcast, followed by the interframe space (3 time units). When no overload frame is sent, the interframe space is transmitted right after the completion of EOF and before next data or remote frame.

4 Properties and Verification Results

For the purpose of verifying whether the models obey the specifications and requirements of CAN protocol, 11 properties based on the standard are proposed. All these properties are specified using UPPAAL property specification language. The verification results and analysis are provided as well.

4.1 Properties

Among all the eleven properties, eight of them are explicitly extracted from section 6 of ISO 11898-1 standard, while the remaining three properties are implicitly derived from this protocol. These properties can be divided into 3 categories, which are safety, liveness and invariant.

(i) **Safety**
Deadlock free (DF): The system will never be in a deadlock state. It is

essential to verify whether the system is deadlock free as the deadlock of any single node will lead to data loss or even system breakdown. It can be specified as

```
A[] not deadlock
```

For all paths the system will never be deadlock. `Deadlock` is a primitive in UPPAAL language used to check whether the system can be in deadlock state.

(ii) **Liveness**

Starvation Freedom (SF): Every node attempting to write a message to the bus eventually succeeds in doing so. This property is not explicitly mentioned in protocol, but it is important to ensure the starvation freedom property in a real-time network because the messages need to be sent eventually without loss. It can be specified as

```
arbitration(N). arbitrating --> arbitration(N). request_success
```

If the state `arbitrating` is reached, eventually the state `request_success` will be reached in the arbitration model of node N.

Remote Data Request (RDR): If a node sends a remote frame, eventually it will receive the message requested. This property is derived from section 6.7 of the CAN standard. It can be specified as

```
request[N]==2-->transceiver(N). read==N
```

If `remote[N]` is set to 2 (which means the message is a remote frame), then the variable `read` in `transceiver(N)` will eventually be N. In other words, when node N requests a message of the same identifier, it will finally receive one.

Error Signaling (ES): Corrupted frames can always be flagged by any node detecting the errors. This property is derived from section 6.9 of the CAN standard. It can be specified as

```
transceiver(N). error_signaling -->bus==error_frame
```

If the `error_signaling` state in `transceiver[N]` is reached, the `bus` value will be the value of `error_frame`.

Error_passive (EP): An error-passive node shall not send an active error flag. This property is derived from section 6.14 of the CAN standard. It can be specified as

```
A<>not(error_passive[N]==1 and transceiver(N). error_signaling
        and transceiver(N). error_broadcast)
```

For all possible transition sequences eventually there is not any time at which `error_passive[N]` is 1, and both `error_signaling` and `error_broadcast` states are reached in `transceiver(N)`.

Error_active (EA): An error-active node shall normally take part in bus communication and send an error flag when an error has been detected. This property is derived from section 6.13 of the CAN standard. It can be specified as

```
A<>not(error_passive[N]==0 and transceiver(N). error_signaling
          and not transceiver(N). error_broadcast)
```

For all possible transition sequences eventually it can not be at the same time when `error_passive[N]` is 0 and `error_signaling` state is reached while the `error_broadcast` cannot be reached in `transceiver(N)`.

Data Consistency (DC): A frame is simultaneously accepted either by all nodes or by no node at all. This property is derived from section 6.6 of the CAN standard. It can be specified as

```
A<>not(bus_off[N]==0 and bus==error_frame and
       transceiver(N).read!=error_frame)
```

For all possible transition sequences eventually there is not any time at which `bus_off[N]` is 0 and `bus` is equal to the value of `error_frame`, but the variable `read` in `transceiver (N)` is not equal to the value of `error_frame`.

Automatic Retransmission (AR): A node that has transmitted a corrupted message will attempt to retransmit the message when the bus becomes idle. This property is derived from section 6.11 of the CAN standard. It can be specified as

```
transceiver(N). error_signaling-->transceiver(N). idle
```

If `error_signaling` state in `transceiver (N)` is reached, the `idle` state in `transceiver (N)` will then be reached.

(iii) **Invariant**

Bus Access Method (BAM): The highest-priority message gains access to the bus. This property is derived from section 6.3 of the CAN standard, to verify that it is always the highest-priority message that gains access to the bus. It can be specified as

```
A[] (winner<=mid3 and winner<=mid2 and winner<=mid1)or
                          winner==idle
```

For all paths the value of `winner` is always smaller than or equal to `mid1`, `mid2` and `mid3`, otherwise `winner` is equal to `idle`.

Bus Off (BO): any node whose error counter reaches the max limit will eventually be bus-off. This property is derived from section 6.15 of the CAN standard, to verify whether any node will be bus-off. It can be specified as

```
E<> transceiver(N). bus_off and arbitration(N). busOff
```

There possibly exists a path through which `bus_off` state in `transceiver (N)` will be reached, as well as the `busOff` state in `arbitration (N)`.

Identifier Disjointness (ID): It is impossible that an arbitration takes place between data messages having identical identifiers. This property is mentioned in the standard that CAN system cannot solve the collision of two messages with the same identifier. So it is verified to ensure identifier exclusion. It can be specified as

```
A<> mid1!=mid2 and mid2!=mid3 and mid3!=mid1
```

For all possible transition sequences eventually `mid1`, `mid2` and `mid3` differ from each other.

There are the other four properties in section 6 of the standard that are not verified in our models, which are 'frames', 'information routing', 'system flexibility' and 'error detection'. 'Frames' indicates that the information on the bus shall be sent in fixed format and limited length. And 'error detection' indicates that five different kinds of errors can be detected. However since the frame format in our models are simplified and the occurrence of errors are randomly triggered, there is no need to verify these two properties. 'Information routing' means information is not transmitted by the configuration of nodes' addresses. Instead, receivers decide whether to accept messages based on acceptance filtering. The 'system flexibility' means nodes may be added to the CAN network without requiring any change in the software or hardware of any node. We do not verify these two properties in this model because they are satisfied in physical layer. Moreover, the properties EP, EA and DC are specified with `A<> not p` format because the nest statement structure, for example `A<> p--> q`, is not supported in UPPAAL specification language.

4.2 Verification results and analysis

The results of verification are shown in Table 1. An entry of YES indicates that the property has passed the verification and holds for CAN bus while NO indicates otherwise. Among all these 11 properties, 6 of them do not hold for CAN models.

Category	Property Name	Verification Result
Safety	DF	NO
Liveness	SF	NO
Liveness	RDR	NO
Liveness	ES	NO
Liveness	EP	YES
Liveness	EA	YES
Liveness	DC	YES
Liveness	AR	NO
Invariant	BAM	NO
Invariant	BO	YES
Invariant	ID	YES

Table 1
Verification Results

By observing and analysing the diagnostic traces generated in the simulator of UPPAAL, we give the reasons why some properties can not pass the verification.

First of all, we find out that models are deadlocked when all nodes reach the bus-off state. With the increase of the error counter, all nodes may eventually be in the bus-off state, unable to send or receive any message, which leads to the failure of DF.

As for the SF, a node may never win the arbitration and successfully send its message when there are always other nodes sending messages of higher identifiers at the same time. A node attempting to send a message may never succeed because of its low priority. That is why this property fails the verification. The RDR cannot pass the verification when the requested node happens to be in the bus-off state before it can reply to the remote request. Another reason is the starvation problem of low priority messages as we mentioned above. If the requested message is of low priority, it always waits for the bus access before it can be successfully transmitted. Similarly, another property AR cannot pass the verification for the same reasons as RDR as illustrated above.

Due to the fact that an error-passive node can not broadcast an error, the ES doesn't hold as well. According to the behavior rules of error-passive nodes, they can only send the passive error signal, which can be overlapped by other dominant bits. So the errors cannot always be signaled. Also, if an error-passive node happens to be the transmitter of the previous message and tries to send the next message, while other nodes are accessing bus at the same time, it may lose the arbitration even if it holds the highest priority identifier, thus the BAM fails to pass the verification.

From the analysis of the results, we conclude that there are three main factors that lead to the failure of some properties. One is due to the low priority of identifier that results in the starvation. Another is caused by nodes becoming in the error-passive state. The last one is that the nodes in bus-off state will not take part in further communication.

In summary, the event-triggered and priority-based mechanisms of CAN lead to the starvation problem. To solve the problem, Time Triggered Controller Area Network (TTCAN) [6] [9] [10] can be used as it is time-triggered. Another problem results from the fault confinement mechanism. This mechanism is designed to ensure performance reliability and stop ill-performed nodes from sending and receiving messages. However, the verification results indicate that the error-passive and bus-off nodes may cause the data loss or data inconsistency.

5 Application layer

The application layer is not explicitly defined for CAN bus. It is open for users to define their own algorithms according to the characteristics of the system when using CAN bus. To make it closer to the real systems and get more accurate results of the verification, in this section, we integrate algorithms in the application layer to the models mentioned above. The definition of the application layer is complicated, but considering the characteristics of our models as well as the properties we want to verify, here we mainly focus on the message scheduling algorithm and the nodes' recovery mechanism.

5.1 Modeling the application layer

There are some different scheduling algorithms such as dynamic scheduling [16] [1] and earliest deadline first [11]. Among all these frequently used algorithms, the dynamic priority scheduling is one of the most popular and representative algorithms.

Compared with the single identifier-based priority, the dynamic scheduling algorithm assigns two priority levels to a message. As we can see in Fig. 9, one priority is the static priority, which is the same as the message identifier. The other one is dynamic priority. The dynamic priority equals to the static priority initially. However, with the increase of failure times of arbitration, the dynamic priority can be promoted. When two or more nodes are accessing the bus at the same time, their dynamic priorities are firstly compared, and the message of the highest dynamic priority gains access to the bus regardless of the static priority. When the messages have the same dynamic priority, the arbitration is done based on the static priority.

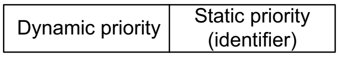

Fig. 9. Dynamic message ID

In our models, the algorithm is applied to the arbitration model by a function called `dynamic_compare()`. Two variables `failure_counter` and `dynamic_priority` are declared for each node. We assume that the dynamic priority is promoted by one when the `failure_counter` reaches three, which means the message has lost the arbitration for three times. When the message finally gains access to the bus, the `failure_counter` is reset to zero and the `dynamic_priority` turns back to the initial number.

The bus-off recovery mechanism requires the node becoming bus-off to recover as fast as it can to ensure the communication to be unblocked. However, in real system, it takes a short period of time for nodes to recover and causes a delay, so in our models, we assume that the bus-off node is reset in a certain period of time units. This period is declared by the variable `recovery_time`, but the specific value is determined by the physical medium.

5.2 Verification results

The verification results shown in section 4 indicates that six properties do not hold for the previous models. But when applied with the dynamic priority scheduling algorithm and the bus-off recovery mechanism, four of them are solved, which are DF, SF, RDR and AR. Table 2 shows the verification results of these six properties before and after adding the application layer. The failure of DF results from the nodes being bus-off, so with the recovery mechanism, the system will not deadlock. And as the SF, RDR and AR do not pass the verification due to the low message priority and bus-off problem, the dynamic priority scheduling algorithm can avoid these problems.

Category	**Property Name**	**Protocol Model**	**Application Model**
Safety	DF	NO	YES
Liveness	SF	NO	YES
Liveness	RDR	NO	YES
Liveness	ES	NO	NO
Liveness	AR	NO	YES
Invariant	BAM	NO	NO

Table 2
Comparison of verification results

As for the ES and BAM, they cannot pass the verification because of the error-passive nodes. The ES refers to the property that errors can always be flagged by nodes detecting them. But according to the protocol, if only error-passive nodes detect the errors, the error flag cannot be broadcast. Similarly, the BAM does not hold for the implementation model because of the same reason we mentioned in section 4.2, the error-passive nodes cannot continuously send two frames regardless of its message priority.

From the analysis above, we can conclude that the critical problems occurred in the data link layer can be partially solved by designing algorithms in the application layer. With regard to the error-passive nodes, the fault confinement mechanism is designed to stop the badly-behaved nodes from disturbing the communication. When a node makes too many mistakes, it is thought to be error-prone and measures are taken to make it transmit as less messages as possible. Our models are just strictly following the actual behavior rules of the error-passive nodes. The fault confinement mechanism tries to avoid the disturbance of badly-behaved nodes, but meanwhile some side effects may be brought out, such as the fact that the limitations to the behaviors of error-passive nodes may lead to data loss and data inconsistency.

6 Related work

As CAN bus is widely used in many automotive electronic systems, various approaches to verify its safety and reliability are proposed like [13], [8], [7], [15], etc. UPPAAL model is used in [13] to verify that their solution for clock synchronization over CAN achieves the desired precision even in the presence of various nodes and channels faults. The formal verification also shows that inconsistent channel faults are a severe threat to the clock precision, but their negative impacts can be reduced by choosing a suitable resynchronization period. And [8] focuses on the implementation of a functional coverage library in SystemC and only applies it on the verification of a CAN bus as a case study.

In [7], the authors use timed automata to verify CAN bus. Models mainly include arbitration process and transmission process without errors. Four properties are verified such as the deadlock freedom and starvation. In addition, they carry out experiments on the response time of sending messages. However, some circumstances are not covered in their models. Besides, as the models are not complete, the properties verified are limited. Our method involves some scenarios under which errors and overload requests may occur, which is close to real situations. Moreover, the fault confinement mechanism and the interframe space, as the main features of CAN, are also implemented in our approach. Some properties are verified during the fault confinement process, such as the data inconsistency resulted from the inability of error-passive nodes to signal errors. The finite-state verification project [15] uses finite-state analysis for verification. The work focuses on the three different kinds of controller chips widely used. All these three controller chips are modeled, verified and analyzed respectively. Although more properties are verified than before, the modeling process is not well explained and much attention is paid to the various kinds of controller chips. What's more, the overload frame and the interframe space are not included in their method.

7 Conclusion

In this paper, models of CAN bus protocol are presented, and 11 properties are verified based on the models. The results indicate that the main problems of CAN bus system are deadlock, starvation, and data inconsistency. The starvation is a consequence of event-triggered and priority-based mechanism, which leads to the starvation of messages with lower priority. Another problem results from the fault confinement mechanism. This mechanism is designed to ensure performance reliability and stop ill-performed nodes from sending and receiving messages, but in some special occasions it may cause the data loss or data inconsistency. Nevertheless, by applying the dynamic priority scheduling algorithm and bus-off recovery mechanism, we can see that the problems mentioned can be partly solved in application layer.

In the future, we will research on modeling and verification of the probability of CAN communication. As we mentioned before, some properties do not hold due to the occurrence of some incidents. The proposal of probability models can better model the real situation of the system. For example, the errors in our models are randomly triggered, however, the probability of the occurrence of errors can be calculated, and integrated in the models. Then properties could be verified on such probability models, which is more accurate and close to real situation.

Acknowledgment

This work was supported by Shanghai Knowledge Service Platform Project No. ZF1213), the Natural Science Foundation of China under Grant (No.91118008), 973 Project (No.2011CB302904), the project of science and technology commitment of shanghai (No. 12511504205), and Key Lab of Information Network Security, Ministry of Public Security (No. C12604).

References

[1] Anwar, K. and Z. Khan, *Dynamic priority based message scheduling on controller area network*, in: *Electrical Engineering, 2007. ICEE '07. International Conference on*, 2007, pp. 1–6.

[2] BellLabs, *Spin*, http://spinroot.com/spin/whatispin.html (2013).

[3] Coq, *Coq*, http://coq.inria.fr/ (2013).

[4] GmbH, R. B., *Can specification version 2.0*, http://esd.cs.ucr.edu/webres/can20.pdf (1991).

[5] ISO11898-1, *Controller area network (can) part 1: Data link layer and physical signalling*, http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=33422 (2003).

[6] ISO11898-4, *Controller area network (can) part 4: Time-triggered communication*, http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=36306/ (2004).

[7] Krakora, J. and Z. Hanzalek, *Timed automata approach to can verification*, 11th IFAC Symposium on Information Control Problems in Manufacturing, INCOM (2004).

[8] Kuznik, C., G. B. Defo and W. Müller, *Verification of a can bus model in systemc with functional coverage*, in: *SIES* (2010), pp. 28–35.

[9] Leen, G. and D. Heffernan, *Ttcan: a new time-triggered controller area network*, Microprocessors and Microsystems **26** (2002), pp. 77–94.

[10] Leen, G. and D. Heffernan, *Modeling and verification of a time-triggered networking protocol*, in: *ICN/ICONS/MCL* (2006), p. 178.

[11] Natale, M. D., *Scheduling the can bus with earliest deadline techniques*, in: *RTSS* (2000), pp. 259–268.

[12] NuSMV, *Nusmv - a new symbolic model checker*, http://nusmv.fbk.eu/ (2013).

[13] Rodriguez-Navas, G., J. Proenza and H. Hansson, *An uppaal model for formal verification of master/slave clock*, in: *6th IEEE Int'l Workshop on Factory Communication Systems (WFCS)* (2006). URL http://www.mrtc.mdh.se/index.php?choice=publications&id=1102

[14] UPPAAL, http://uppaal.org/ (2012).

[15] van Osch, M. J. P. and S. A. Smolka, *Finite-state analysis of the can bus protocol*, in: *HASE* (2001), pp. 42–52.

[16] Velasco, M., P. Martí, J. Yépez, R. Villà and J. M. Fuertes, *Schedulability analysis for can-based networked control systems with dynamic bandwidth management*, in: *ETFA* (2009), pp. 1–8.

[17] Waszniowski, L., J. Krakora and Z. Hanzálek, *Case study on distributed and fault tolerant system modeling based on timed automata*, Journal of Systems and Software **82** (2009), pp. 1678–1694.