Finite-State Analysis of the CAN Bus Protocol

Michiel van Osch
Department of Math and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
mvosch@win.tue.nl

Scott A. Smolka
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
sas@cs.sunysb.edu

Abstract

We formally specify the data link layer of the Controller Area Network *(CAN), a high-speed serial bus system with real-time capabilities, widely used in embedded systems. CAN's primary application domain is automotive, and the physical and data link layers of the CAN architecture were the subject of the ISO 11898 international standard. We checked our specification against 12 important properties of CAN, eight of which are gleaned from the ISO standard; the other four are desirable properties not directly mentioned in the standard. Our results indicate that not all properties can be expected to hold of a CAN implementation and we discuss the implications of these findings. Moreover, we have conducted a number of experiments aimed at determining how the size of the protocol's state space is affected by the introduction of various features of the data link layer, the number of nodes in the network, the number of distinct message types, and other parameters.*

1. Introduction

The *Controller Area Network* (CAN) [ISO93, Law97] is a high-speed serial bus system with real-time capabilities, widely used in embedded systems. It was developed by Robert Bosch GmbH, a leading manufacturer of automotive equipment, and controller chips have been available since 1989. In 1993, the physical and data link layers of the CAN architecture were the subject of the ISO 11898 international standard.

Currently, there are more than 15 chip manufacturers who collectively make in excess of 50 different types of CAN controller chips. CAN's main application domain is automotive: it is used in most European passenger cars, and has been targeted for use by truck and off-road vehicle manufacturers. However, CAN has also found extensive use in industrial machinery, medical equipment, and even in some domestic appliances. One concrete measure of CAN's popularity is that, as of June 2000, Philips Semiconductors has sold 100 million CAN transceivers [Phi00], the device that connects a CAN controller to a CAN network.

CAN is a serial bus system with multi-master capabilities; that is, all nodes are able to transmit data and multiple nodes can request access to the bus simultaneously. In CAN networks there is no addressing of subscribers or stations in the conventional sense, but, instead, prioritized messages are transmitted. A transmitter broadcasts a message to all CAN nodes whereupon each node decides, on the basis of the identifier received, whether it should process the message or not. The identifier also determines the priority that the message enjoys in competition for bus access. This form of bus arbitration is called Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority (CSMA/CD + AMP).

A key feature of CAN is what the ISO standard refers to as *data consistency*: a transmitted message is simultaneously received by all nodes or by no nodes. This atomicity property is characteristic of many embedded systems where universal agreement among nodes is essential to the correct operation of the system [Cri90, HT93]. Another important feature of CAN is its *high transmission reliability*. The CAN controller registers a station's error and evaluates it statistically in order to take appropriate measures, which may extend to disconnecting the CAN node producing the errors.

In this paper, we present a concise yet formal specification of the CAN data link layer protocol and the results of a number of verification experiments we have performed on the protocol. To the best of our knowledge, this is the first attempt to formalize and verify this all-important protocol. More specifically, our contributions can be seen as the following:

- Using the Murφ verification system [Dil96], we have produced a total of nine CAN specifications. This total is arrived at as follows. We consider three successively more elaborate implementations of the CAN controller that have been proposed in the literature. For each controller implementation, we consider three successively feature-enriched versions of the protocol, starting with a specification of the basic bus-arbitration protocol, and then adding on such key features as the ability of a node to request data from the source, error handling, and fault confinement. This layered presentation facilitates the understanding of these features and their specification, and their interactions. All specifications are fully parameterized by the number of nodes in the network and by the number of message types.

- For each specification, we used Murφ to check 12 important properties of CAN. Eight of these are gleaned from the ISO standard, and the other four are desirable properties not directly mentioned in the standard. Moreover, we have conducted a number of experiments aimed at determining how the size of the protocol's state space is affected by the introduction of new features, the number of nodes in the network, the number of distinct message types, and other parameters. This yielded a total of 195 verification runs, with state-space sizes ranging from 13 states to 4.2 million states.
- No specification satisfies all 12 properties and the results we present (Table 1) illustrate this point. We discuss the implications of these findings and hope that our results, along with the specifications themselves, will serve as a useful guide to future CAN implementors.

The rest of the paper develops along the following lines. Section 2 briefly describes the CAN data link layer protocol. Section 3 provides an overview of the Murφ verification system. Section 4 explains how we encoded the protocol in the input language of the Murφ tool. Our verification results are presented in Section 5 while Section 6 considers related work. Section 7 contains our concluding remarks. Although space limitations allow us to only highlight our Murφ specifications and accompanying performance data, an Appendix presents the rule definitions for CAN's basic arbitration procedure. Complete versions can be found at [vO01].

2 The CAN Data Link Layer Protocol

The CAN data link layer is standardized in ISO 11898, and its services are implemented in the Logical Link Control (LLC) and Medium Access Control (MAC) sub-layers of a CAN controller. A CAN network consists of N controllers plus a serial bus. Each controller is directly connected to the bus. When the bus is idle, a controller may start to write a message to the bus which will be broadcast to every controller in the network (including itself). When the message has been received successfully by all controllers, each node in the network decides on its own to process the message or discard it, a process known as *acceptance filtering*. Figure 1 illustrates the CAN architecture for the case $N = 4$. Controller 3 is broadcasting a message to the other controllers. Only the second controller discards the message as it has no interest in it.

A standard CAN message, the format of which is illustrated in Figure 2, consists of the following fields. The SOF bit delineates the start of the message (also called a "frame"). The arbitration field contains an identifier, encoding the priority of the message, and an RTR bit, indicating whether or not the message is a remote request. In the case of a request, the RTR bit is set and the message does not contain any data. The response to the request is a message with the same identifier but with the data. The control field mainly specifies the number of bytes of data contained in the message, while the data field stores the actual content of the message (a maximum of eight bytes). For example, in a CAN network in a car, this may contain the current speed of the vehicle. The CRC field contains a 15-bit CRC (cyclic redundancy code) sequence which is used to check the integrity of a received message. The ACK field allows a node to check that the message it transmitted was successfully received by at least one of the controllers. The 7-bit EOF bit sequence marks the end of the message.

Over the years, different types of CAN controllers have been developed, all of which comply to the ISO 11898 standard but differ in implementation. In particular, a separation can be made between:

Basic CAN controllers with one or more receive buffers (to store messages received from the bus but not yet processed by the node's CPU) and only one write buffer (to store a message to be written to the bus). Such controllers suffer from the problem known as "inner priority inversion", where the transmission of a high-priority message is prevented by the presence of a low-priority message in the single write buffer.

Intermediate CAN controllers with one or more receive buffers (like a basic CAN controller) but with more than one write buffer. Intermediate controllers are aimed at overcoming the problem of inner priority inversion.

Full CAN controllers (also known as "CAN controllers with object storage") having dual-ported RAM to store messages and enough memory to store one message of each possible identifier value in a designated memory location. Such controllers eliminate inner priority inversion (since there is a buffer slot for each message priority level) but are subject to "data overwrite": a newer message with a certain identifier value overwrites an older message of the same identifier value, before the older message has had an opportunity to be written to the bus or processed by the node.

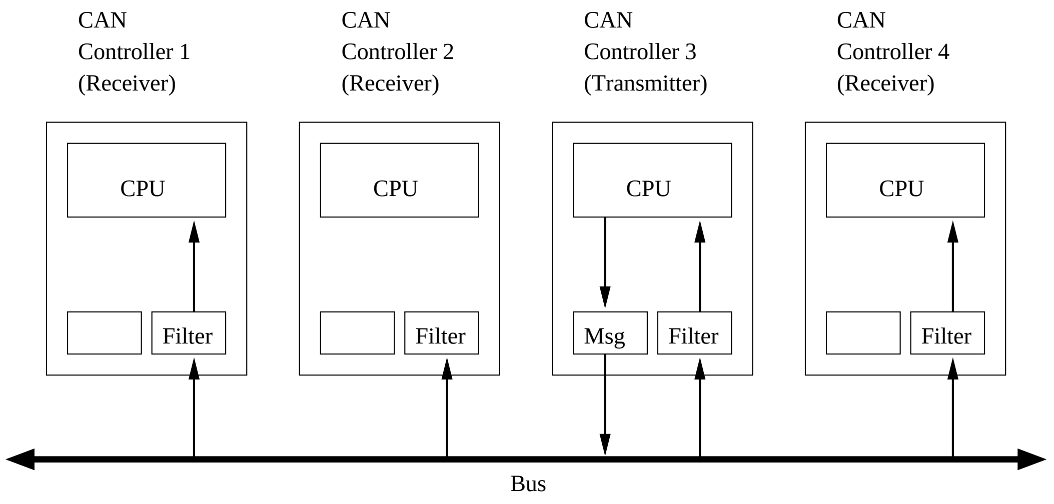

Figure 1. CAN architecture.

The functionality of the CAN data link layer can be separated into a number of services, including bus arbitration, remote data requests, error handling, and fault confinement. We begin with bus arbitration.

Bus Arbitration. Arbitration is required if more than one node is accessing the bus. The bus access method used in CAN is called Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority (CSMA/CD + AMP), and works as follows. While the bus is idle, a "hard synchronization" on the SOF bits transmitted by those nodes attempting to write to the bus occurs, after which each of these nodes sends the bits of its message identifier and monitors the bus level. Bits with a dominant value (zero) overwrite those with a recessive value (one) due to the wired-and nature of the CAN physical layer. If, at some point, node i sends a recessive bit but reads back a dominant bit, then the message identifier of some other transmitting node has a lower binary value and therefore a higher priority than node i's message. Node i therefore loses the arbitration and switches to listening-only mode. At the conclusion of this process, the node with the highest-priority message wins the arbitration and this message has been broadcast to the other nodes.

Remote Message Requests. Controllers can also request data. A controller does this by broadcasting the identifier of the message it wants to receive to the other controllers. The controller that can write the corresponding message will do so as soon as the bus becomes idle by taking part in a normal arbitration procedure. If there is a collision between a request for a certain message and the response to that request, the response will win the arbitration.

Error Handling. If a CAN controller detects an error in a message it is currently receiving (due e.g. to signal dispersion during propagation along the bus lines), it will cease reception of that message and start the transmission of an *error flag*: six bits of the same polarity. (If the controller in question is the one transmitting the corrupted message, it will also cease transmission.) The error flag is a (intentional) violation of the bit-stuffing rule, which states that each message may only contain up to five consecutive bits with the same polarity. Upon receiving the error flag, the other controllers will detect the bit-stuffing error and transmit error flags of their own. When all controllers have detected and reported the error, the bus becomes idle and a new arbitration procedure may commence (in which the transmitter of the previously corrupted message can again try to access the bus). This approach to error handling, in which a local error is transformed into a global error, is called "globalization of local errors" and is the key idea behind the network-wide data consistency provided by CAN.

Fault Confinement. Should a controller develop a more permanent malfunction, steps should be taken to prevent it from continuing to transmit/detect corrupted messages. CAN provides such a mechanism. Every controller is equipped with two counters: REC (Receive Error Counter) and TEC (Transmit Error Counter). Whenever a node detects an error, the corresponding counter is incremented, and when it successfully transmits or receives a message, the corresponding counter is decremented. A node starts in the *error-active* state with its counters initialized to zero. If either counter reaches a certain value (256), the node enters the *bus-off* state, is disconnected from the network, and can rejoin the network only via a software reset. There is also an intermediate state, *error passive* (when one of the counters reaches the value 128), in which a node is unable to broadcast an error flag and writing to the bus is possible only if no error-active controller wishes to write to the bus.

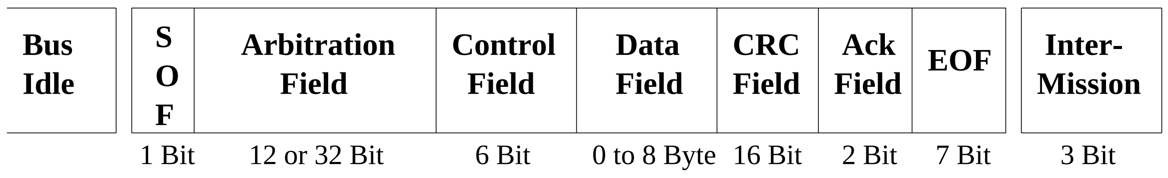

Figure 2. CAN message format.

3 The Murφ Verification System

The Murφ verification system [Dil96] consists of the Murφ compiler and the Murφ description language. The Murφ compiler generates a special-purpose verifier (C++ program) from a Murφ description. The Murφ description language is based on a collection of guarded commands (condition/action rules), which are executed repeatedly in an infinite loop. During loop execution, a rule is chosen nondeterministically for execution from among the set of rules whose conditions are satisfied. Execution of a rule is atomic.

Verification in Murφ is based on explicit-state enumeration. A Murφ state is an assignment of values to all of the (global) variables of the description. A Murφ verifier performs depth- or breadth-first search in the state graph determined by a Murφ description, storing all the states it encounters in a large hash table. When a state is generated that is already in the hash table, the search algorithm does not expand its successor states (as these were expanded when the state was originally inserted in the table).

A Murφ specification consists of constant and type declarations, variable declarations, procedure declarations, rule definitions, and a description of the start state. Data types of the language include subranges, enumerated types, arrays, and records. Each rule is a guarded command consisting of a condition and an action. The condition is a boolean expression and the action is a sequence of statements.

Verification is accomplished by augmenting a Murφ description with invariants and liveness formulas, the latter of which are specified using a subset of linear-time temporal logic (LTL). If during state-space generation a state is encountered in which an invariant is violated, Murφ reports an error and halts. Liveness formulas are checked after the state space has been generated. The latest version of Murφ is 3.1; however, LTL model checking is not supported in versions beyond 2.70L (due to an incompatibility between liveness checking and symmetry reduction, as reported on the Murφ web site). Hence we used 2.70L for the results reported in this paper.

4 Specifying the CAN Protocol in Murφ

In this section, we describe how we specified the CAN bus protocol in the input language of the Murφ verification system. We chose Murφ mainly because of the simplicity of its guarded-command specification language and its support for global variables; the latter makes it particularly easy to model a shared-bus architecture in a distributed system.

We begin with the specification of the bus arbitration procedure, assuming basic CAN controllers (one transmit buffer per controller). The rule definitions for this specification are given in Appendix A. We subsequently enrich this specification to cover the various extensions of the CAN protocol. In the specification, the nodes of a CAN network are represented as an `ARRAY[0..N-1]` of `Nodetype`, where `Nodetype` consists of a `Readbuffer` field, in which a message read from the bus is stored, and a `Writebuffer` field, in which a message to be written to the bus is stored. Both `Readbuffer` and `Writebuffer` are of type `Messagetype`, a record type consisting of two fields: `MessageID`, ranging from 0 to `Max_Value+1`, and `NodeID`, ranging from 0 to `N`.

The `MessageID` field corresponds to the identifier field of the standard CAN message format, which is used to determine message priority. We have paired this field with the `NodeID` field to ensure that sets of messages transmitted by any pair of nodes in the network are mutually disjoint, a prerequisite for modeling a CAN network. Note that the data content of a message is abstracted away in our specification.

Like a message, the bus is modeled by a record consisting of a `MessageID` field and a `NodeID` field, the combination of which represents the message currently being transmitted on the bus. The initial values of `MessageID` and `NodeID` are `Max_Value+1` and `N`, respectively, representing the empty bus. The `Readbuffer` and `Writebuffer` fields of a node are initialized similarly.

For the purpose of modeling the arbitration procedure, we divide an arbitration cycle into three phases: In the *processing* phase, nodes may perform internal actions; for example, a node may check whether or not it has won the (previous) arbitration. In the *writing* phase, some subset of the nodes attempt to write a message to the bus. In the *reading* phase, all nodes read the contents of the bus.

We use a collection of five condition/action rules to specify the arbitration procedure. For the sake of discussion, we number the rules 1 through 5, although this has no bearing on the order in which the rules are executed. Three of the rules apply to the processing phase, two of which we now describe. Rule 1 places a valid message in the write buffer of a node in the network, the choice of node being nondeterministic. Murφ's RULESET construct, which can be thought of as syntactic sugar for creating a copy of its component rules for every value of its quantifier, is used for this purpose. Repeated execution of Rule 1 will nondeterministically determine a subset of the nodes that wish to engage in the arbitration procedure. Rule 2 switches the arbitration phase from PROCESSING to WRITING when at least one node has indicated that it is attempting to write to the bus (by the presence of a non-empty write buffer).

Rule 3 is applicable to the writing phase. It determines, from among those nodes attempting to write to the bus, the node with the highest-priority message, sets the contents of the bus equal to this message, and changes the arbitration phase to READING. This rule captures the essence of CAN's AMP (Arbitration on Message Priority) arbitration procedure. Rule 4 applies to the reading phase. It atomically sets the read buffer of each node to the contents of the bus and switches the phase back to PROCESSING. Once back in the processing phase, with all read buffers nonempty, Rule 5 is applicable. In this rule, every node checks whether its NodeID matches the NodeID of the message on the bus. If so, it has won the arbitration and may clear its write buffer; only the winner is allowed to do this. All read buffers are cleared and the phase remains PROCESSING, thereby re-enabling the first two rules.

There is also a rule for the start state of the system, which sets the phase to PROCESSING and clears each node's read and write buffers. We have also extended the specification of the arbitration procedure along several lines, which we now discuss.

Remote Requests. To model remote requests for a message, we extend Messagetype with the field Request, a boolean variable indicating whether or not a message is a request. Rule 1, which nondeterministically chooses a node to place a message in its write buffer, is modified as follows. Two node Ids, i and j, are nondeterministically chosen, along with a message Id, and the NodeID of node i's write buffer is set to j. If $i \neq j$, the Request field of node i's write buffer is set to true, indicating that node i is requesting a certain type of message from node j; it is otherwise set to false. In the original version of Rule 1, only one node Id, i, was required as the rule always set the NodeID of node i's write buffer to i. Rule 5 must also be modified: the node that is the target of the request will place the requested message in its write buffer if the buffer is empty; otherwise the response to the request is lost.

Error Handling. For this extension, we add a Status field to Messagetype and to the record type modeling the bus. Status is an enumerated type that can assume the values OK and CORRUPT. We also add a Participant field to Nodetype, a boolean variable indicating if the node is an active participant in the current arbitration cycle: a node will become inactive for the remainder of the current arbitration cycle after detecting a corrupt message. Moreover, rules are added to introduce corrupt messages into the system, during the reading phase (zero or more read buffers may become corrupted) or during the writing phase (the bus may become corrupted causing every node to read a corrupted message during the subsequent reading phase).

We also add a rule, applicable during the processing phase, that checks each node to see if it has received a corrupt message, and, if so, clears the node's Readbuffer and sets its Participant flag to false. A related rule detects if some node has set its Participant flag to false, and, if so, sets the Status of the bus to CORRUPT prior to setting the phase to READING. In the subsequent reading phase, all nodes detect that the bus has been corrupted and become inactive, leading to a new arbitration cycle.

Fault Confinement. Fault confinement is specified by augmenting Nodetype with the fields REC (Receive Error Counter), ranging from 0 to Max_BUSOFF; TEC (Transmit Error Counter), also ranging from 0 to Max_BUSOFF; and Error_State, an enumerated type ranging over the values ERROR_ACTIVE, ERROR_PASSIVE, and BUS_OFF. Existing rules modeling the occurrence of correct or corrupt transmissions/receptions are augmented to update the appropriate counters. Moreover, a "bus becomes idle" rule is added to transition nodes between the three error states, and to appropriately restrict their behavior according to the state they are in. See the discussion on fault confinement in Section 2 for further details.

Nodes with More Than One Write Variables. The changes to the specification needed to model CAN controllers providing additional buffering for write variables mainly concern the definitions of the data types. For intermediate CAN controllers, we simply extend the type of a node's Writebuffer field from Messagetype to ARRAY[0..B-1] OF Messagetype. The message written to the bus is the highest-priority one in the array. We also sort the array every time an identifier is added or removed, which has several benefits: the highest-priority message in the array will always occupy position zero; and the size of the protocol's state space is significantly reduced since non-sorted array configurations are eliminated from consideration.

For full CAN, the type of Writebuffer becomes ARRAY[0..N-1] OF ValueArray, where N is the number of nodes in the network and `ValueArray` is the type `ARRAY[0..Max_Value] OF Boolean`. A one in location (i, j) of this array indicates the presence of a pending message of `MessageID` i and `NodeID` j. Encoding full CAN in this style is superior, in terms of state-space overhead, to the more direct approach of extending the length of the `Writebuffer` array in intermediate CAN to `Max_Value`.

5 Verification Results

We checked our specifications for the following 12 important properties. The first eight were derived from the ISO standard; the remaining four represent desirable properties of the protocol that did not specifically appear in the ISO standard:

Bus Access Method (BAM) : The highest-priority message gains access to the bus (ISO 11898, Section 4.2).

Data Consistency (DC) : A frame is simultaneously accepted either by all nodes or by no node at all (ISO 11898, Section 4.5).

Remote Data Request (RDR) : If a node sends a remote frame, another node answers with the corresponding data frame (ISO 11898, Section 4.6).

Error Signaling part 1 (ES1) : Corrupted frames are flagged by any transmitting node (ISO 11898, Section 4.8).

Error Signaling part 2 (ES2) : Corrupted frames are flagged by any active nodes (ISO 11898, Section 4.8).

Automatic Retransmission part 1 (AR1) : A node that has lost arbitration will attempt to retransmit its message when the bus becomes idle (ISO 11898, Section 4.10).

Automatic Retransmission part 2 (AR2) : A node that has transmitted a corrupted message will attempt to retransmit the message when the bus becomes idle (ISO 11898, Section 4.10).

Bus Off (BO) : A node in the bus-off state can neither send nor receive any messages (ISO 11898, Section 4.14).

Starvation Freedom (SF) : Every node attempting to write a message to the bus eventually succeeds in doing so.

Synchronous Broadcast (SB) : All reading of the bus is synchronous; i.e. the read variables of all participating nodes have the same value at all times.

Identifier Consistency (IC) : A valid identifier value always consists of a `MessageID` less than `Max_Value+1` and a `NodeID` less than `N`.

Identifier Disjointness (ID) : It is not possible that an arbitration takes place between data messages having identical identifiers.

Each of these properties was encoded in Murφ using the LTL constructs `INVARIANT` (**BAM**, **BO**, **SB**, **IC**, **ID**) and `LIVENESS` (**DC**, **RDR**, **ES1**, **ES2**, **AR1**, **AR2**, **SF**). Liveness properties were of the form `ALWAYS b1 -> EVENTUALLY b2`, for boolean propositions `b1` and `b2`. The LTL encodings of **BAM** and **DC** for basic CAN appear in in Appendix A for illustration purposes; for the encodings of the other properties, please see [vO01].

All properties were checked against nine versions of the specification obtained by successively increasing the number of transmit buffers available to the controllers (yielding basic, intermediate, and full CAN controllers) and by successively adding new features to the specification as follows: start with arbitration only, to which we add remote requests and error handling, to which we add fault confinement.

The results of the verification runs are presented in Table 1. An entry of “Yes” indicates that the given property did indeed hold of the given specification while an entry of “No” indicates otherwise. An entry of “N/A” indicates that the property was not relevant to the specification in question.

Given that our specifications are parameterized by the number of nodes, message identifiers, write buffers and other parameters, the results reported in Table 1 were obtained on representative instances of each specification. Such instances ranged from 6 nodes and 9 message identifiers for basic CAN (approx. 4 million states) to 2 nodes and 4 message identifiers for full CAN with error handling and fault confinement (approx. 2.03 million states). Smaller instances were also checked and our results were consistent across all instances we considered of a given specification.

For those entries where a property was indeed relevant to a specification, a “Yes” result was expected (or at least hoped for). But, as can be seen from the table, this was not always so. The reasons for these “No” cases are as follows.

DC does not hold for the fault-confinement versions because if only passive receivers detect the error, none of them will report it to the bus, thereby permitting other nodes to accept the message as successfully transmitted.

The invariant property **RDR** (when applicable) can never be guaranteed to hold because either the message containing the requested data is lost due to the lack of available write buffers (in basic and intermediate CAN), or this message fails to win arbitration due to other nodes continually

	basic arbitr.	basic + request + error	basic + confine	interm. arbitr.	interm. + request + error	interm. + confine	full arbitr.	full + request + error	full + confine
BAM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DC	N/A	Yes	No	N/A	Yes	No	N/A	Yes	No
RDR	N/A	No	No	N/A	No	No	N/A	No	No
ES1	N/A	Yes	Yes	N/A	Yes	Yes	N/A	Yes	Yes
ES2	N/A	N/A	Yes	N/A	N/A	Yes	N/A	N/A	Yes
AR1	Yes	Yes	No	No	No	No	No	No	No
AR2	N/A	Yes	No	N/A	No	No	N/A	No	No
BO	N/A	N/A	Yes	N/A	N/A	Yes	N/A	N/A	Yes
SF	No	No	No	No	No	No	No	No	No
SB	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
IC	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ID	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 1. Verification results.

sending higher-priority messages (see also the discussion of **SF** below).

AR1 and **AR2** do not hold for intermediate and full CAN because of internal arbitration: if a controller is buffering more than one type of message, the message with the highest priority among them is written to the bus. As long as there is a higher-priority message in the transmit buffers, a lower-priority message need never get a turn. Moreover, **AR1** and **AR2** do not hold for the confinement versions as a node that has lost arbitration or detects that it has transmitted a corrupted message could enter the bus-off state at the end of the current arbitration cycle. In this case, it will be prohibited from re-transmitting any messages to the bus.

SF never holds because a message of a certain priority can continually lose out to higher-priority messages in the arbitration procedure. Therefore the CAN bus is inherently unfair. As noted above, the absence of fairness also underlies CAN's failure to satisfy several other properties, including **RDR**, **AR1**, and **AR2**.

Murφ also checks for deadlocks during state-space generation. Deadlock was found in the three versions of the specification incorporating fault confinement, and occurs when all nodes enter the bus-off error state.

Time and Space Requirements: Murφ checks invariant properties during state-space generation and therefore verifying an invariant requires little additional computational overhead. The overhead for the liveness properties, on the other hand, is significant. Liveness properties that do not hold are checked very fast (Murφ generates a counter example in a matter of seconds). For properties that do hold, however, the complete state space must be explored, which takes as much time as generating the state space itself. Memory-wise, verifying a liveness property that does hold takes roughly twice the amount of memory required by Murφ to generate the state space.

Table 2 illustrates the computational resources required for state-space generation for an increasing number of nodes and message identifiers. The specification in question models basic CAN controllers, bus arbitration, remote requests, and error handling. A much more extensive collection of performance data can be found in [vO01].

5.1 Discussion of Results

From our analysis of the results of Table 1, it is possible to identify three main factors that contribute to a property failing to hold for a given specification of the CAN bus protocol:

- Starvation of low-priority data transmissions (properties effected: **RDR**, **AR1**, **AR2**, **SF**).
- Nodes becoming bus off or entering the error-passive state as part of the protocol's fault-confinement procedure (**DC**, **AR1**, **AR2**).
- Limited number of write buffers in basic and intermediate CAN (**RDR**).

Of these, the issue of message starvation has received the most attention within the CAN community and has resulted in a number of higher-level, application-layer protocols aimed at circumventing the problem. For example, the CANopen networking system [FB01], developed in the context of an EEC ESPRIT III project, supports a variety of communication services that provide guaranteed and/or periodic data transmission within a CAN network. One such service is the Process Data Object (PDO) which implements

nodes	values	states	rules fired	time (sec)	memory
2	1	249	372	0.10	3.32 Kb
	2	745	1120	0.15	12,97 Kb
	3	1489	2244	0.31	25.92 Kb
	4	2481	3744	0.52	43.19 Kb
	5	3721	5620	0.83	54.77 Kb
	6	5209	7872	1.27	90.67 Kb
	7	6945	10500	1.87	120.9 Kb
	8	8929	13504	2.56	155.5 Kb
	9	11161	16884	3.53	194.3 Kb
	10	13541	20640	4.46	235.7 Kb
3	1	4336	7440	0.84	75.48 Kb
	2	23557	40512	5.74	410.1 Kb
	3	68842	118494	21.29	1.17 Mb
	4	151369	260664	51.67	2.45 Mb
	5	282316	486300	112.01	4.58 Mb
	6	472861	814680	205.90	7.67 Mb
	7	734182	1265082	354.24	11.91 Mb
	8	1077457	186784	578.80	17.47 Mb
	9	1513864	2609064	893.13	24.55 Mb
	10	2054581	3541200	1329.47	33.31 Mb

Table 2. State-space statistics for basic CAN with error handling.

a producer-consumer style of data transmission. By associating an "inhibit time" with a PDO, which defines the minimum time that has to elapse between consecutive invocations of a PDO service, starvation for low-priority communication objects can be eliminated.

A similar approach has been taken by the ISO Task Force TC22/SC3/WG1/TF6, which is developing a Time-Triggered version of CAN (TTCAN) to be implemented at the session layer of the ISO OSI protocol stack [FMD$^+$00]. TTCAN assigns a time window to each controller during which the controller is allowed to access the bus, thereby ensuring guaranteed periodic data transmission within a CAN network.

Given its importance to the CAN community of developers, it would be interesting to determine the precise effect of guaranteed fair data transmission on the results of Table 1. To this end, we encoded such a fairness property in the Murφ verifier. The property is of the form: *It is always the case that all write buffers eventually become empty*. Under this assumption, it is possible to show that the starvation-freedom property **SF** always holds and consequently so do **AR1** and **AR2** for intermediate and full CAN (except for the fault-confinement versions). **RDR** continues to be plagued by the lack of write buffers in basic and intermediate CAN.

6 Related Work

In [R$^+$98], Rufino et al. describe several scenarios under which the CAN data link layer protocol does not properly implement atomic broadcast. These scenarios arise when transmission faults hit the last two bits of the 7-bit end-of-frame delimiter (see Figure 2) and result in duplicate message reception and inconsistent message omissions: some nodes accept a frame and others reject it, a clear violation of property **DC** (data consistency). The authors analyze the occurrence probabilities of these scenarios (which are non-negligible) and propose a protocol to run on top of CAN that ensures fault-tolerant broadcast.

In contrast, our results involve scenarios under which **DC** fails to hold due to the inability of error-passive nodes to report erroneous frames to the bus. Moreover, our CAN specifications do not explicitly model the EOF bit sequence and, consequently, the error scenarios reported in [R$^+$98] are not uncovered by our analysis.

In this paper, we have used formal methods to analyze the CAN protocol proper. A somewhat different approach is taken by Kendall in [Ken01] where he describes the *CANDLE* tool and high-level language for specifying CAN-based control systems. In a *CANDLE* specification of a control system, the underlying network model is an abstraction of a CAN network that correctly implements atomic broadcast. The specification may then be compiled into an intermediate format accepted by the verification tools in the

OPEN/CÆSAR tool environment [Gar98].

7 Conclusions

We have presented a family of specifications of the CAN protocol and checked these specifications for 12 important properties. Our results of Table 1 indicate for each specification which properties do and do not hold. As discussed in Section 5, our specifications are parameterized by the number of nodes in the network and number of distinct message identifiers (among other parameters), and a wide range of instances of the specifications (in terms of state-space sizes) were checked. A useful direction for future work would be to verify these properties on parameterized versions of the specifications using inductive techniques, i.e., without resorting to instantiation.

We also plan to extend our verification efforts to TTCAN, the time-triggered version of CAN discussed in Section 5.1.

Acknowledgements

The authors are grateful to the anonymous referees for their valuable comments. Research supported in part by NSF grant CCR-9988155 and ARO grants DAAD190110003 and DAAD190110019.

References

[Cri90] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. Technical report, IBM Almaden Research Center, San Jose, CA, 1990.

[Dil96] D. L. Dill. The Murφ verification system. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[FB01] M. Farsi and M. Barbosa. *CANopen Implementation*. Research Studies Press Ltd., 2001.

[FMD$^+$00] T. Fuhrer, B. Muller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time triggered communications on CAN. In *Proceedings of 7th International CAN Conference*, 2000.

[Gar98] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98), Lecture Notes in Computer Science*, pages 68–84, 1998.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*. Addison-Wesley, Chapter 5, 1993.

[ISO93] ISO 11898: Road vehicles – Interchange of digital information – Controller area network (CAN) for high-speed communication, 1993.

[Ken01] D. Kendall. CANDLE: A tool for efficient analysis of CAN control systems. In *Proceedings of RT-TOOLS (Workshop of Real-Time Tools)*, Aalborg, Denmark, August 2001.

[Law97] W. Lawrenz. *CAN System Engineering – From Theory to Practical Applications*. Springer-Verlag, New York, 1997.

[Phi00] Philips sells 100 million CAN transceivers. `www.semiconductors.philips.com`, 2000.

[R$^+$98] J. Rufino et al. Fault-tolerant broadcasts in CAN. In *Proceedings of 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, pages 150–159, Munich, Germany, 1998. IEEE Computer Society.

[vO01] M. van Osch. `http://www.win.tue.nl/~canbus/`, 2001.

A Murφ Specification of Basic CAN

```
-- Nondeterministically choose subset of nodes to participate in arbitration.

RULESET i: 0..N-1; msg: 0..Max_Value
DO
  RULE "1:  Initialization of nodes"
    ! Wants_to_Write(Node[i]) & Bus_Is_Idle() &
    Timephase = PROCESSING
==>
  BEGIN
    Node[i].Write.Identifier.MessageID := msg;
    Node[i].Write.Identifier.NodeID := i;
  END; -- of rule
END; -- of ruleset

RULE "2:  Timeswitch"
   Timephase = PROCESSING &
   EXISTS i: 0..N-1 DO Want_to_Write(Node[i]) END & Bus_Is_Idle()
==>
  BEGIN
    Timephase := WRITING;
  END; -- of rule

-- The highest priority message is written to the bus.

RULE "3:  Arbitration procedure"
  Bus_Is_Idle() & Timephase = WRITING
==>
  BEGIN
    FOR i : 0..N-1 DO
     IF Wants-to_Write(Node[i]) &
        Lessthan(Node[i].Write.Identifier, Bus.Identifier) THEN
       Bus.Identifier := Node[i].Write.Identifier;
     END; -- of if
   END; -- of for
   Timephase := READING;
  END; -- of rule

-- Message broadcast to all nodes

RULE "4:  Broadcast"
  Timephase = READING
==>
  BEGIN
    FOR i:0..N-1 DO
      Node[i].Read.Identifier:=Bus.Identifier;
    END; -- of for
    Timephase := PROCESSING;
  END; -- of rule
```

```
-- Winner is determined and has successfully broadcast a message.

RULE "5:  Determine winner"
  FORALL i:0..N-1 DO Has_Read(Node[i]) END & Timephase = PROCESSING
==>
  BEGIN
     FOR i:0..N-1 DO
      IF Equal(Node[i].Read.Identifier,Node[i].Write.Identifier) THEN
           Clear_Node(Node[i].Write);
      END; -- of if
      Clear_Node(Node[i].Read);
    END; -- of for
    Clear_Bus();
  END; -- of rule

----------------------
-- Sample LTL formulas
----------------------

INVARIANT ``Bus Access Method''
  Bus_is_Idle() |
  FORALL i:0..N-1 DO Want_to_Write(Node[i]) ->
       (Bus.Identifier.MessageID < Node[i].Write.Identifier.MessageID) |
       ( (Bus.Identifier.NodeID <= Node[i].Write.Identifier.NodeID) &
         (Bus.Identifier.MessageID < Node[i].Write.Identifier.MessageID) )
END;

LIVENESS ``Data Consistency''
  ALWAYS EXISTS i:0..N-1 DO Node[i].Read.Status = CORRUPT END ->
  EVENTUALLY EXISTS i:0..N-1 DO Has_Read(Node[i]) END &
             FORALL i:0..N-1 DO Has_Read(Node[i]) ->
                                Node[i].Read.Status = CORRUPT END;

--------------
-- Start state
--------------

STARTSTATE

BEGIN
  Timephase:=PROCESSING;
  FOR i:0..N-1 DO
    Clear_Node(Node[i].Write);
    Clear_Node(Node[i].Read);
  END;
  Clear_Bus();
END; -- of startstate
```