A Finite State Analysis of Time-triggered CAN (TTCAN) Protocol using Spin

Indranil Saha and Suman Roy
Honeywell Technology Solutions Lab Pvt. Ltd. (HTSL),
151/1, Doraisanipalya, Bannerghatta Road, Bangalore 560 076, India
Email: {indranil.saha,suman.roy}@honeywell.com

Abstract

The paper presents a case study of the use of model checking for the analysis of an industrial protocol, a time triggered version of the CAN protocol (TTCAN). Our analysis of this protocol was carried out using the model checker Spin. The original CAN protocol can easily be modeled in Spin, but specifying TTCAN requires the provision of explicitly using time in the modeling language. With a view to express time triggered properties we use a discrete time version of Spin (DT-Spin). This extension allows one to quantify discrete time elapse between events by specifying the time slice in which they occur. Using DT-Spin we have been able to model TTCAN, and subsequently, verify a few of its time-triggered properties. This experience shows that it is possible to largely model TDMA-based protocols using discrete time.

1 Introduction

Model-checking techniques are increasingly being used for debugging and proving correctness of finite-state concurrent reactive systems. Many different types of properties of a system can be checked by these techniques: deadlocks, dead code, unspecified receptions, buffer overruns, etc. In this paper we present an application of model checking for the analysis of a time triggered version of CAN protocol. The Controller Area Network (CAN) [7, 10] 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. CAN has found extensive uses in automotive industries, industrial machineries, medical equipments, and even in some domestic appliances.

The arbitrating mechanism of CAN protocol (in [12] a comprehensive description of CAN protocol has been presented) ensures that all messages are transferred according to the priority of their identifiers and the message with the highest priority will not be disturbed. The next generation vehicles and some mission critical subnetworks will require additionally deterministic behavior in communication during service. Even at maximum bus load, the transmission of all safety related messages must be guaranteed. One way to solve this issue using CAN is the extension of the standard CAN protocol to a time triggered protocol TTCAN [4]. The communication is based on the periodic transmission of a reference message by a time master. This allows to introduce a system wide global network time with high precision. Based on this time different messages are assigned to different time windows within a basic cycle.

M. van Osch and S. C. Smolka [12] have formally specified the data link layer of the Controller Area Network (CAN). They checked their specification against twelve 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. Krakora and Hanzlek have done a series of work on the verification of CAN Protocol using the framework of timed automata [9]. They model the CAN protocol as a network of timed automata and prove a few properties in UPPAAL. In other research [8] they undertook a similar work where they model parts of the distributed system (application SW, operating system and communication bus) by automata which use synchronization primitive enabling their interconnection. Leen and Heffernen have carried out a formal specification and verification of TTCAN using the UPPAAL model checker [11]. A model of TTCAN was created using UPPAAL system editor. Models of three TTCAN network nodes and the physical medium were designed, while nine formal properties including deadlock were examined using the UPPAAL verification engine.

In this paper, we report a concise yet formal specification of the time triggered version of CAN Protocol [4] and the results of a number of verification experiments we have performed on the protocol. Our analysis of the protocol was performed using the model checker Spin [6]. Modeling of TTCAN requires one to express time explicitly and towards that we use a Discrete time version of Spin (DT-Spin) [1, 2]. This extension lets one quantify discrete time elapsed between events by specifying the time slice in which they occur. Using DT-Spin we specify TTCAN and

verify a few of its time-triggered properties. Our modeling approach demonstrates the potential of DT-Spin to model TDMA-based protocols like TTCAN.

2 Modeling TTCAN in DT-Spin

Time triggered communication in TTCAN is based on the reference message being transmitted regularly by the time master. Following the reference message there is a sequence of time windows of variable sizes that provide the time slots for individual message transmissions. Since the cyclic transmission schedule is synchronized by the repeated transmission of the reference message transmitted by the time master we assume that this message restarts the cycle time. In our model, the cycle time is captured by a timer *basiccycle* which is set to an arbitrary length of 16 time units in the *init* process.

In our model there is one Time Master *master* which has a single *writebuffer* and a single *readbuffer*. In TTCAN more than one Time Master are predefined to establish fault tolerance and functionality. A start-up algorithm picks up only a Time Master among all the potential time masters and it does not change unless the current Time Master develops a fault within itself. In this work we do not address the start-up issues of TTCAN protocol. We are only interested in the operation of TTCAN when the network is stable and the Time Master is working properly. Under these circumstances it is enough to consider one Time Master only. There are three other controllers modeled by an array, controller[0], controller[1] and controller[2]. In the basic model, all these controllers have a single *readbuffer* and a single *writebuffer*. The *readbuffer* and the *writebuffer* are of the datatype *Msg* which contains two data fields: *msgid* denoting the message identity, and *destination* denoting the controller which the message is meant for. The bus is modeled as a datatype containing three fields: *msgid*, *source* denoting the source of the message, and *destination* denoting the destination of the message.

The reference message is represented by the *msgid* 0. We assume that there are six messages with *msgid* ranging from 1 to 6. We arbitrarily assign these messages to the three controllers: *controller[0]* can send messages with *msgid* 1 and 6, *controller[1]* messages with *msgid* 3 and 5, and controller[2] messages with *msgid* 2 and 4. The *maxvalue* (=7) represents that no message exists in the *writebuffer* of a controller or in the bus. The number of messages are kept lower to get rid of state space explosion problem. We assume that the *master* is responsible only for sending the reference message, it does not send or receive any other messages. In the *init* process, the message with *msgid* 0 is written in the *msgid* data field in *writebuffer* of the *master*. For the other three controllers, appropriate messages are generated randomly. In the beginning of a new *basiccycle* a function *startbasiccycle* writes a message with *msgid* 0 in the *writebuffer* of the *master*. If only the *msgid* of the *writebuffer* of any controller is *maxvalue* (i.e., there is no message to write for that controller), then a new message is generated in the *writebuffer* of that controller.

Now we describe how we design different time windows within a *basiccycle*. In the design of the TTCAN protocol, the system designer enjoys enough freedom to decide on the number of time windows and their sizes. An off-line design tool is used to analyze the communication pattern, so that no conflict will happen. DT-Spin can be used to express temporal and causal relationships between two events in an interval of discrete time, i.e., we can say two events A and B occur in an interval $[i, i+1]$, $i \geq 0$, or event A triggers event B in the same interval. We use this fact to model TTCAN in DT-Spin [1, 2]. The main challenge in modeling protocols like TTCAN is to formally specify the time windows so that events in different windows should occur in a deterministic manner. We use the set-expire construct of DT-Spin to model a window. To model a *basiccycle* with n windows w1, w2, ... wn, we require n+1 timers - one to represent the *basiccycle* itself, and n timers to represent the duration of n windows. The basic window has a duration of d time-unit, and this *basiccycle* is divided among n windows with duration d1, d2,.... dn. We model the *basiccycle* as follows.We represent the timer corresponding to the *basiccycle* as bc, and that corresponding to the n windows as tmr1, tmr2,...tmrn.

```
do
::(bc>0)&&(bc<=d)->
  atomic{
  if
  ::((bc>=d-d1+1)&&(bc<=d))->
    set(tmr1, d1);
    // sequential statements for window w1
    expire(tmr1);
  ::((bc>=d-d1-d2+1)&&(bc<d-d1+1))->
    set(tmr2, d2);
    // sequential statements for window w2
    expire(tmr2);
  ..
  ..
  ::((bc>0)&&(bc<d-d1-d2- ... -d(n-1)+1))->
    set(tmrn, dn);
    // sequential statements for window wn
    expire(tmrn);
  :: else->
  fi;
}
od
```

Modeling of the consecutive windows in this way ensures that the events occurring in one window will not be interleaved with those belonging to another window, because

tmr1 can only expire if all the sequential statements of window w1 have been executed, and timer tmr2 is set once tmr1 expires.

We have designed the windows in a *basiccycle* in such a way that all the features of TTCAN can be properly specified. We have assumed delays for individual controllers which have been accommodated suitably by the length of each exclusive slot. The *basiccycle* has been divided into five time windows of different sizes mentioned below.

- The first window, where *basiccycle* decrements from 16 to 13, is made as an exclusive window for the *master*. In this slot it writes the reference message to the bus, other controllers read the reference message from the bus.

- The second window, ranging from 12 to 9 time units is designed as an exclusive slot for *controller[1]* where it writes the message on the bus that it wants to send. This is followed by the controller reading of the bus, for whom the *controller[1]* has written.

- The third window is kept free for future use. This free slot ranges from time units lower than 9 to 7.

- The fourth window is designed as an arbitrating window where *basiccycle* goes down from lower than 7 to 4. Here the controllers compete to write to the bus and the winner is determined by the arbitration procedure. During the arbitration the controller with minimum *msgid* is selected as the winner, and is allowed to write to the bus. The message is read from the bus by the destination controller in the same time slot.If there is no controller ready to write to the bus then the flag *nowinnerflag* is set, thus nothing happens in this slot.

- The fifth or last time slot with *basiccycle* time ranging from lower than 4 to 0 is designed as the exclusive slot for *controller*[2]. The operation in the slot is same as in the second slot.

Note that there is no exclusive slot for *controller[0]*. This is done intentionally, for it is required for the validation of some properties.

Remote Data Request. The model for Remote Data Request has been developed by enhancing the basic model. In the *Msg* datatype two new fields are introduced: a bit *request* which indicates whether a controller wants to send a message or wants to request for a message, and a *requestedmsgid* holding the *msgid* of the message requested. In the first *basiccycle* there is no requested message. In the beginning of next *basiccycle*s the *startbasiccycle* function initially checks for every controller whether it has some message to send or request for. Otherwise it determines whether a controller wants to send a message, or wants to request for a message, the data field *request* of the *writebuffer* of the controller is made 0 or 1 accordingly. A controller sends a request for a message only if it has already received its previously requested message. In the exclusive slot for a controller, if the controller has to send any request then it writes the *requestedmsgid* from its *writebuffer* to the bus. Also, it writes the requested message id in the *requestedmsgid* field of its *readbuffer*. The request field of its *readbuffer* is made 1. If the *msgid* of the writebuffer of the requested controller is different from *maxvalue* then the request is dropped. Otherwise the *msgid* of the bus (requested message id) is copied in the *msgid* of the *writebuffer* of the requested controller. The content of the source field of the bus is written in the destination field of the *writebuffer* of the requested controller. In the arbitration slot the winner performs the same operation as that of a controller in its exclusive slot. Only the arbitration is modified, requested messages get more priority than the general messages.

When a controller receives a message, it checks whether the received message is same as the message it requested previously. It does so by checking whether the received *msgid* is equal to *requestedmsgid* of its *readbuffer*. If it is so, then it makes the request field of the *readbuffer* 0.

Error Handling. Corrupt messages may occur due to signal dispersion during propagation along the bus line. If a controller detects an error in a message it is currently receiving, it will cease the reception of that message and transmits an error flag. The way we model the error handling is that in each slot after the transmission of a message, the bus is randomly given a status (a boolean variable), `ok` or `corrupt`. In the case of a normal message transmission, if the message is corrupt, then the *msgid* of *readbuffer* of the receiving controller is set to *maxvalue*, and the flag *errorflag* is set to 1. The *msgid* data field of the *writebuffer* of the sender of the corrupt message does not change, as it tries to resend the same message in an arbitration slot in the same *basiccycle* or in the next *basiccycle*. In case of Remote Data Request, the receiver of the corrupt message simply sets the *errorflag* to 1.

Fault Confinement. To model fault confinement, three new data fields have been introduced in the datatype Node: *tec* (denoting the transmit error counter), *rec* (denoting the receive error counter) and *status*. The value of the *status* field is 0, 1 or 2, depending on whether a controller is in *error-active*, *error-passive* or *bus-off* state. In the *init* process these three data fields are set to 0 for all the controllers, as all the controllers starts in *error-active* condition. If a data transmission is in error, the value of *tec* of the source controller, and *rec* of the controller receiving the transmitted data are increased by 1. The *status* field of the controllers is updated according to the value of their *tec* or *rec* field. After a successful message transmission the value of *tec* of the source controller and the value of rec of the

destination controller is decremented by 1, provided their value is greater than 0. If the *status* of any controller is 2 (in bus-off state), then in the *startbasiccycle* procedure the *msgid* and *requestedmsgid* fields of *writebuffer* are set to max-value so that they cannot participate in any data transmission. The arbitration procedure has been changed to model the fact that the controllers which are *error-passive* can only write to the bus if no *error-active* controller wants to write to the bus. In this procedure, first, *error-active* controllers are checked whether they want to transmit or request a message. If no such *error-active* controller is present then *error-passive* controllers take part in arbitration. Also, if there is an error then only *error-active* controller can transmit the *errorflag*.

Fault-Tolerance. Fault-tolerance of TTCAN has been modeled enhancing the model for Error Handling. In this model, *Master* and *Node* datatype contains a new field *buscorruptflag*. Two buses are named *bus1* an *bus2*. If the *buserrorflag* for any controller is 0, it transmits its message through *bus1*, if the flag is 1, then the controller sends its message through *bus2*. When a transmission is in error, then if the flag *buscorruptflag* of the source controller of the corrupted message is 0, then it in set to 1, and when the controller gets chance for the next time (either by winning in an *arbitration* slot or in its exclusive slot in the next *basiccycle*), it transmits the same message through *bus2*. If the flag *buscorruptflag* for a controller is 1 while its sent message is corrupted, then the flag is set to 0, and the message is discarded by writing *maxvalue* in the *msgid* or *requestedmsgid* field in the *writebuffer* of that controller.

3 Verification of Properties of TTCAN

We construct five models of TTCAN in DT-Spin - basic model, basic model with remote request, basic model with error handling, basic model with fault confinement and basic model with fault tolerance. We checked for the specification of properties that are gleaned from the properties of the original CAN protocol from ISO 11898.

Progression of Time (POT) Naturally we would like to avoid situations when the passage of time is blocked (referred to as *zero cycle* Bošnački and Dams [1]). This property should be verified to check the sanity of the model.

Starvation Freedom Properties We checked a few starvation freedom properties for the master and the dedicated node in a particular exclusive slot.

Starvation Freedom for the Time Master (SF1) In the first slot the time master eventually writes to the bus.

Starvation Freedom for the for the controller for which slots have been allocated in the basic cycle (SF2) We check two properties: in the second slot *controller[1]* eventually succeeds in writing to the bus; in the fifth slot *controller[2]* eventually succeeds in writing to the bus.

Starvation Freedom for the deprived node (SF3) Controller[0] eventually writes to the bus.

Data Consistency (DC) In the first slot all controllers eventually read whatever Time Master writes to the bus.

Bus Off (BO) In the third slot (Free Slot) no controller writes any message to the bus.

Bus Access Method (BAM) In the fourth slot (Arbitration Slot) either the bus is idle or the message with the highest priority gains access to the bus.

Automatic Retransmission (AR) A node that has lost arbitration will attempt to retransmit its message in the immediate next basic cycle.

Remote Data Request (RDR) If a node sends a remote data request, another node eventually provides an answer to the request.

Error Signaling (ES1) A corrupted frame is eventually flagged by the receiver of the frame.

Error Signaling (ES2) A corrupted frame is eventually flagged by the receiver controller if the controller is error active.

Fault Tolerance (FT1) In a basiccycle if bus1 becomes corrupt for transmission from a controller then the controller sends the same message on bus2 in the next basiccycle.

Fault Tolerance (FT2) In a basiccycle if bus2 becomes corrupt for transmission from a controller then the message is discarded, and a fresh message is transmitted through bus1 in the next basic cycle.

Each of these properties was encoded in SPIN using the standard LTL constructs. POT is a recurrence property while BO is an invariance property. SF1, SF2, DC, ESs and FT1 are response properties. BAM, AR, RDR and FT2 are precedence properties while SF3 is a guarantee property. For LTL formulas with the next **X** operator we used the converter "ltl2ba" (which converts standard LTL formulas into PROMELA never claims) [5]. For the limitation of space, we present only the LTL encodings of the property Bus Access Method (BAM) below.

Bus Access Method (BAM)

$\Box\,(p \rightarrow r\,U\,(t \vee \Diamond\,((s1 \wedge s2)\,U\,q)))$

$p \equiv basiccycle < 7,\ q \equiv basiccycle >= 4,\ r \equiv basiccycle! = 0,\ t \equiv$

properties	basic	basic + remote req.	basic+ remote req. + error handl.	basic+ remote req. + fault confin.	basic+ remote req. + fault toleran.
POT	Yes	Yes	Yes	Yes	Yes
SF1	Yes	Yes	Yes	Yes	Yes
SF2	Yes	Yes	Yes	Yes	Yes
SF3	No	No	No	No	No
DC	Yes	Yes	Yes	Yes	Yes
BO	Yes	Yes	Yes	Yes	Yes
BAM	Yes	Yes	Yes	Yes	Yes
RDR	NA	No	No	No	No
AR	Yes	Yes	Yes	Yes	Yes
ES1	NA	NA	Yes	Yes	Yes
ES2	NA	NA	NA	Yes	NA
FT1	NA	NA	NA	NA	Yes
FT2	NA	NA	NA	NA	Yes

Table 1. Verification Results for TTCAN

$nowinnerflag == 1$, $s1 \equiv bus1.msgid == controller[winner].writebuffer.msgid$,
$s2 \equiv bus1.destination == controller[winner].writebuffer.destination$.

In LTL $\Box$ stand for "always", $\diamondsuit$ stands for "eventually", and U stands for "strong until".

3.1 Verification results

The results of verification runs are presented in Table 1. An entry of "Yes" indicates that the given property holds of the given specification while an entry of "No" indicates otherwise. AN entry of "NA" indicates that the property was not relevant to the specification in question.

Our model does not suffer from any deadlock as it does not get into a zero cycle. The guaranteed timeliness property of TTCAN ensures the validity of starvation freedom properties like SF1 and SF2 and data consistency DC, while bus access method BAM remains true of all the specifications as expected. The remote data request property (RDR) can never be guaranteed to hold in our model as the destination node (for which the controller sends the message) may not always be able to write to the bus during a *basiccycle* as either it has to compete with other nodes or it may not enjoy the exclusive privilege to write to the bus in a slot. Error signalling and fault tolerance properties hold in the appropriate models. We also study the effect of guaranteed fair data transmission. It is captured by the following *fairness property:* every node attempting to write a message to the bus eventually succeeds in doing so, in other words, it is always the case that all the write buffers eventually become empty. These are modeled by Starvation Freedom properties. Our results show that this property is valid only for the controllers for which exclusive slots are provided.

4 Conclusion

Although we have considered a much simplified (level 1) design of TTCAN [4] in future we aim to add more complexities to the protocol design before formally analyzing it which can include an analysis of the startup algorithm of the protocol. Towards that we can use timeout based models [3] for synchronization of different events during start-up using discrete timers which can be easily captured by DT-Spin.

As observed earlier, our specifications are parameterized by the number of controllers in the network and the number of distinct message identifiers. In future we plan to verify these properties on parameterized versions of specifications using inductive techniques.

References

[1] D. Bošanački and D. Dams. Discrete-time promela and spin. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *LNCS*, pages 307–310. Springer-Verlag, 1998.

[2] D. Bošanački and D. Dams. Integrating real time into spin: A prototype implementation. In *Proceedings of the FORTE/PSTV XVIII conference*, pages 423–439, 1998.

[3] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Proceedings of FORMATS/FTRTFT*, 2004.

[4] T. Führer, B. Mü, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time triggered communication on can (time-triggered can-ttcan). *Robert Bosch, GmbH*.

[5] P. Gastin and D. Oddux. Fast ltl to büchi automata translation. In *Proceedings of 13th International Conference on Computer Aided Verification*, volume 2076 of *LNCS*, pages 694–707. Springer-Verlag.

[6] G. J. Holtzman. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003.

[7] ISO11898-Part1. Road vehicles - interchange of digital information - part 1: Controller area network (can) for high-speed communication. 1993.

[8] J. Krakora and Z. Hanzalek. Timed automata approach to can verification. In *Proceedings of 11th IFAC Symposium on Information Control Problems in Manufacturing*, 2004.

[9] J. Krakora and Z. Hanzalek. Verifying real-time properties of can bus by timed automata. In *Proceedings of FISITA*, 2004.

[10] W. Lawrenz. Can system engineering – from theory to practical applications. 1997.

[11] G. Leen and D. Heffernan. Modeling and verification of a time-triggered networking protocol. In *Proceedings of International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*, 2006.

[12] M. van Osch and S. A. Smolka. Finite state analysis of can protocol. In *Proceedings of Sixth IEEE International Symposium on High Assurance Systems Engineering Learning Technologies*, 2001.