

An Industrial Application of Model Checking to a Vessel Control System

Daniel Keating

Electrical and Computer Engineering
University of Canterbury
New Zealand

Email: daniel.keating@pg.canterbury.ac.nz

Allan McInnes

Electrical and Computer Engineering
University of Canterbury
New Zealand

Email: allan.mcinnnes@canterbury.ac.nz

Michael Hayes

Electrical and Computer Engineering
University of Canterbury
New Zealand

Email: michael.hayes@canterbury.ac.nz

Abstract—Model checking allows an abstracted finite state model of a system to be developed and a set of mathematically defined correctness properties, based on the design specifications, to be defined. The model checker performs an exhaustive state space search of the model, checking the correctness properties hold at each step. This paper describes how model checking has been applied to find and correct problems in the software design of a distributed vessel control system currently under development at a control systems specialist in New Zealand.

I. INTRODUCTION

A New Zealand based control systems specialist are currently developing a fully redundant distributed vessel control system. The Time-Triggered Controller-Area-Network (TTCAN) protocol is used for communication between modules on the system. In distributed systems such as this, the interaction between a number of concurrently executing modules often results in unexpected sequences of events occurring in the system. The complicated nature of these systems often leads to design errors which are difficult to detect using conventional techniques such as code walk throughs, peer reviews, unit testing, and functional testing [1]. The aim of this research was to apply model checking techniques to assist in verifying the correct operation of the implementation of the TTCAN protocol in the control system.

Using traditional testing techniques it is only feasible to focus on verifying a small subset of possible sequences of events that can occur in a system. As a result, catastrophic failures that have a low probability of occurring are often overlooked. This is especially relevant in concurrent systems, due to the complex sequences of events that can be generated. A model checker can be used to target complicated areas of a system, checking all possible sequences of events against specified correctness properties [1]. State space restrictions prevent large portions of the system from being verified by the model checker, but model checking is a useful tool which can be added to the development process, and used in addition to traditional testing techniques to help gain confidence in a design.

Section II gives an overview of model checking, the TTCAN protocol, and the implementation of the control system. Section III describes models of the procedures used to determine the active time-master and to select the currently active bus in

this implementation of TTCAN. Section IV describes the correctness properties developed to check against the model and presents the verification results. Finally, Section V concludes the paper.

II. BACKGROUND

This section briefly introduces model checking, gives an overview of the TTCAN protocol, and describes how it is implemented in the control system.

A. Model checking

Model checking is a technique used to automatically verify correctness properties against a finite state model of a system. A concurrent program can be visualised as a Finite State Machine (FSM) or automaton. The state of the system is the current expression being executed in each of the concurrent processes and the current set of values of variables at a specific time. The FSM contains nodes to represent every possible state the system enters and edges to represent possible transitions between states [4]. A model-checker traverses all possible paths through the concurrent system's FSM, checking specified safety and liveness properties at each step. If a property is disproved a counter-example is generated showing the sequence of events leading to the violation of the property. The counter-example is a valuable aid in debugging a system [3].

One model checker that has become increasingly popular in industry is the SPIN model checker. The input specification language to SPIN for describing a model is called Process Meta-Language (PROMELA). Correctness properties to be checked against the system are described using Linear Temporal Logic (LTL). A processes is created in PROMELA to represent the control flow of a program. The individual threads of a multi-threaded application or simultaneously executing nodes on a distributed network are modelled in PROMELA by creating multiple processes. Shared variables and message channels are used to model interprocess communication.

Temporal logic formulae are logical statements that allow sequences of a program's states to be described [2]. In SPIN, LTL formulae are used to specify safety and liveness properties to check against a model. Safety properties are checks that a correctness property holds over an exhaustive state space search of a model. A safety property specifies that an event or

combination of events never happens. This is represented in LTL by the always operator (\square), meaning the specified proposition is always invariantly satisfied throughout the search. A liveness property specifies that an event or combination of events eventually happens. Liveness is represented in LTL by the eventually operator (\diamond), meaning the specified property is eventually satisfied during the search. Checking liveness properties is important, as a safety property may hold in the trivial case where the process is doing nothing at all. Operators can be combined to specify more interesting properties, for example, the always and eventually operators are combined to check that an event occurs infinitely often in a system. The model must always eventually pass through a certain state to satisfy the property [4].

B. TTCAN overview

Modern vehicles contain complicated distributed control systems. Currently, the CAN protocol is one of the most popular communications protocols used in these types of networks [5]. CAN is an asynchronous event-triggered protocol, meaning that events are sent around the network as they occur. Due to the event-triggered nature of the protocol, conflicts can occur on the bus when multiple messages are sent simultaneously. A priority-based arbitration scheme is used to determine which message is transmitted on the bus. This introduces non-deterministic latency to message transmissions, complicating the design of systems with tight timing constraints, such as a closed-loop distributed control system like those used in vehicle's brake or steer by-wire systems [8].

The increasing size and complexity of these types of systems has introduced a need for a variant of the protocol that provides deterministic timing across the system. This makes system design simpler and the design of more complicated systems possible. Time-Triggered CAN (TTCAN) is a variant of the CAN protocol that offers more precise timing. The TTCAN protocol is the addition of a session layer (layer 5 of the OSI model), defined by ISO 11898-4 [7], to the existing data link (OSI layer 2) and physical (OSI layer 1) layers of the existing CAN protocol [6].

TTCAN is a synchronous (time-triggered) protocol, where the transmission of messages is based on the progression of a globally synchronised time base. Each node has a pre-defined schedule of messages to transmit in pre-allocated time-slots. This way conflicts on the bus are eliminated and message latencies can be guaranteed [8].

Time synchronisation is achieved by the periodic transmission of a specific message known as a 'reference message' from a designated time-master node. On receiving the Start of Frame (SOF) bit of the reference message, all node's pre-defined transmission cycles are restarted. Messages will be sent when their scheduled time-slot becomes active. The period elapsed between two consecutive reference messages is known as a 'basic-cycle', and a number of 'basic-cycles' with different message schedules may be repeated in a 'matrix-cycle' [5].

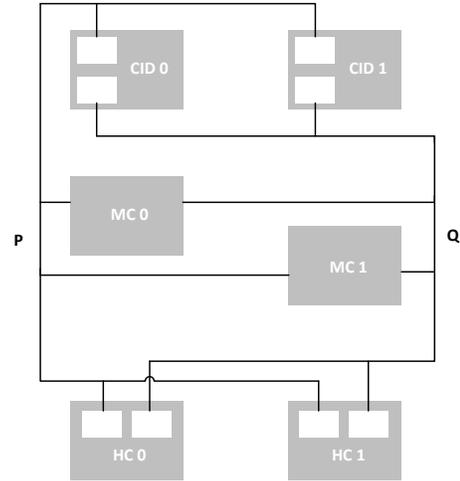


Fig. 1: Block diagram of the redundant control system.

Using a time-triggered protocol allows deterministic timing of transmissions and a higher portion of the overall bandwidth of the system to be utilised. However, the latency of transmissions may be greater than when using an event-triggered protocol.

C. Implementation of TTCAN in the control system

The system to be analysed is a dual redundant distributed vessel control system based on the TTCAN protocol. The redundant CAN buses are labelled P and Q in Figure 1. The vessel is usually wired with a bus running down each side of the vessel. The system consists of three main types of modules: Control Input Devices (CID), Master Controllers (MC), and Hydraulic Controllers (HC). Each module is connected to both of the redundant CAN buses.

CIDs take input actions from the operator such as adjusting the throttle or steering the vessel and translate these to input commands that are sent to the MC. The CIDs have redundant microcontrollers, each connected to one of the redundant CAN buses.

The MC processes control input commands from the CIDs and feedback messages from the HCs. It translates these commands and feedback into output demands that are sent to the HC. The microcontroller on the MC has two CAN controller modules, one connected to the P bus, and the other to the Q bus, as shown in Figure 1. When the currently active time-master MC transmits messages, they are sent on both CAN buses. The MC is the time-master of the system; it is responsible for synchronising the message schedules of the other network nodes. This is achieved by the active MC periodically sending a TTCAN sync reference message.

III. APPLYING MODEL CHECKING TO THE IMPLEMENTATION

This section describes the models developed for model checking the vessel control system. The model checking work here focuses on the implementation of the TTCAN protocol,

as this is a major component of the control system. The approach taken was to develop multiple models focused on specific aspects of the system, rather than one monolithic model. It was found that developing a number of separate smaller models was essential in checking this system due to its size and complexity. Using this technique, we were able to model the system efficiently without encountering the state space explosion problem. Two of the PROMELA models developed during the model checking work are described in this section. The models described are:

- A model of the active time-master election process, called the “voter” process in the implementation, analyses the election procedure that determines the active time-master on startup or reintegration.
- A model of the module in the MC that selects which of the redundant buses currently active buses the node is currently listening to, called the “signal picker” in the implementation.

After talking with the development engineers and analysing the code, it appeared that the correct operation of the redundant MC nodes is crucial to the safety of their system, and checking the correctness of the “voter” procedure is essential to the correct operation of the MC. The periodic sync reference message sent from the currently active time-master MC is the heart-beat of the system. The reference message triggers and synchronises the transmission schedules of all the other nodes in the system. Without the sync reference message, exclusive window messages, responsible for control of the vessel, are not transferred from the helm and throttle to the hydraulic controller. Also, crucial to the correct operation of the MC is the “signal picker” module. This is responsible for selecting which of the redundant buses the MC is currently listening to.

A. “Voter” model

Figure 2 shows the “voter” state machine from the implementation, that this model is based on. Scenarios where intermittent faults cause time-master nodes to periodically toggle on and off are modelled, as well as the addition of initial startup delays of the MC nodes.

The “voter” model consists of two MC nodes, a CID, and a HC. The MC nodes are represented in the model by `TimeMasterNode` processes and each is associated with a `PeriodicTimer` process, as shown in Figure 3. In this model, the MCs simulate the “voter” procedure. The `AdvTimeBusArb` process is responsible for handling the progression of time, message transmissions, and CAN bus arbitration in the model.

The “voter” state machine is implemented in the model as a non-deterministic `if ... fi` construct in the `TimeMasterNode` processes, as shown in Listing 2. Guard statements of the construct test the value of the `currentState` variable to determine the state in the “voter” procedure that the node is currently in. When the MC is first initialised, the state machine defaults to the `INVALID` state. Following this, a check is made to see if the node was previously the active time-master before the reset. If

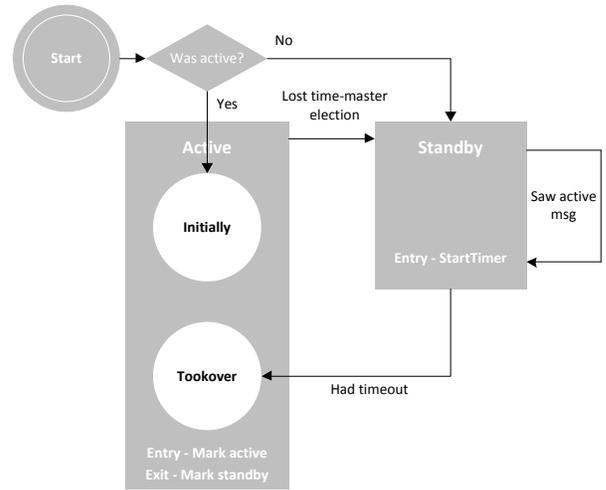


Fig. 2: State machine of time-master election process in MC nodes.

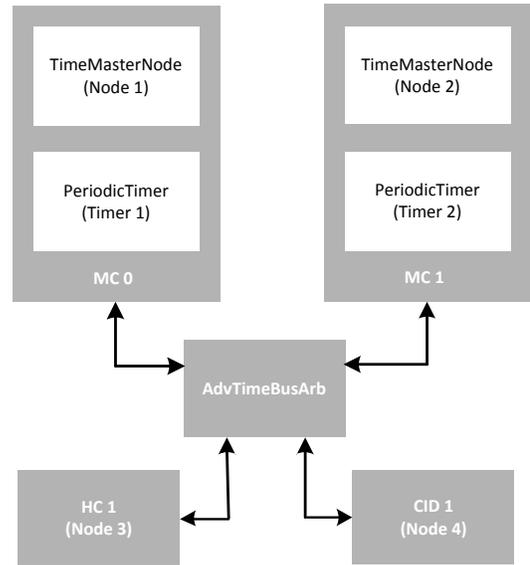


Fig. 3: Block diagram showing processes and connections between them in the “voter” model.

this is the case, the node will transition to the `INITIALLY` and then `TOOKOVER` states. If the MC was not previously the active time-master it will transition to the `STANDBY` state. On entering the `STANDBY` state, the MC will start a 3 s timeout by setting the `standbyTimer` variable to `STANDBY_TIMEOUT`. This timeout is reset when a “master status” message is received from an active time-master MC. If a “master status” message is not received within the 3 s timeout period, the backup time-master takes over as the active time-master, by transitioning from the `STANDBY` to `TOOKOVER` state. If both nodes are currently in the `TOOKOVER` state and configured as the active time-master the node with the highest priority, which is defined as the node with the lowest identifier, will become the sole active time-master.

A timer in each MC is used to periodically trigger transmission of a message representing the “master status” message used in the implementation. The “master status” message conveys the current state of each MC to the other; it allows a MC to determine whether the other MC is in active or backup mode. An abstraction is made to simplify modelling of the “master status” message transmission. In the model, when the “master status” message is correctly received at a node, the rxMsgBuff flag in the receiving node’s MsgStatus structure is set, as shown in Listing 2. When processing the “master status” message in the MC a global variable called ActiveTimeMaster is checked. This is used to determine whether the other MC node is in active or backup states. An assumption is made here that the “master status” message is always transmitted, received, and decoded correctly by the MC nodes.

Listing 1: PROMELA source from the “TimeMasterNode” process, used to check if a “master status” message has been received from the other MC node.

```

/* Received a master status message from other
MC node. */
IF MsgStatus[nodeNum].rxMsgBuff
  [MASTER_STATUS_OFFSET + nodeNum] != false ->
5   MsgStatus[nodeNum].rxMsgBuff
  [MASTER_STATUS_OFFSET + nodeNum] = false;
  recvMasterStatus = true;
FI;

```

Listing 2: PROMELA source from the “TimeMasterNode” process, used to model the “voter” state machine.

```

/* Update the active time-master state at the end
of each 50ms cycle. */
if
:: currentState == INVALID ->
5   if
  :: ActiveTimeMaster[nodeNum] != false ->
    currentState = INITIALLY;
  :: else ->
    currentState = STANDBY;
10  fi;
:: currentState == INITIALLY ->
  currentState = TOOKOVER;
:: currentState == TOOKOVER ->
  /* Had timeout from standby state, now enters
  the 'TOOKOVER' state. */
  ActiveTimeMaster[nodeNum] = true;
  /* Received master status message from a node
  which is currently an active master. */
  IF (recvMasterStatus != false)
20    && (ActiveTimeMaster[1 - nodeNum]
    != false) ->
    if
  :: nodeNum == NODE_1 ->
    /* Do nothing as this is the node with
    higher priority (lower identifier).
    Remains as the active time-master. */
  :: nodeNum == NODE_2 ->
    currentState = STANDBY;
    ActiveTimeMaster[nodeNum] = false;
30  :: else ->
    skip;
    fi;
  FI;
:: currentState == STANDBY ->
35  if
  :: standbyTimer != 0 ->

```

```

  standbyTimer = standbyTimer - 1;
  IF standbyTimer == 0 ->
    currentState = TOOKOVER;
40  FI;
  :: else ->
    standbyTimer = STANDBY_TIMEOUT;
  fi;
  /* If received a 'master status' message and
  the other node is the active master, then
  reset the timeout. */
45  IF ((recvMasterStatus != false)
  && (ActiveTimeMaster[1 - nodeNum]
  != false)) ->
50  standbyTimer = STANDBY_TIMEOUT;
  FI;
  :: else ->
    skip;
  fi;

```

The PeriodicTimer associated with each TimeMasterNode is responsible for externally triggering periodic events in the TimeMasterNode process. In this case, it is used to periodically toggle the time-master on and off at a certain rate, in order to model a periodic fault at the node. An external process must be used to handle this due to the technique used for modelling timing. A timeout period is passed as a parameter along with a count for the number of timeouts elapsed before the node toggles state. In this model, the timeout period is one basic-cycle, with a duration of 50 ms as is the case in the implementation.

A startup delay is modelled in the same way as in the other TTCAN protocol models. The initial startup delay is passed as a parameter when initialising the corresponding TimeMasterNode process. This is used to model the existing startup delay of the MC, due to initialisation of the boot-block and threading, and is able to be adjusted to simulate other delays.

B. “Signal picker” model

The “signal picker” is a module that exists in the MC and is responsible for selecting which of the redundant buses the MC is currently listening to. The “signal picker” model verifies the module’s interaction with the environment. The model simulates messages received on either bus, messages reporting errors from their source, dropped messages, and the periodic state updates of the module as occurs in the implementation. Correctness properties have been developed based on the developer’s design specifications. Assertions are used in the model to check the properties hold. The model checks each of the possible interleavings of events between the signal picker and the redundant buses. In this case, an untimed model has been used; it was not necessary to include the timing of events relative to each other to check the properties specified.

The model consists of a process that represents the “signal picker” module, and a process, called TriggerInputs, that models the interaction of the “signal picker” module with its environment. The “signal picker” reacts to events triggered by the TriggerInputs process.

The TriggerInputs process non-deterministically triggers events that occur in the environment and in the “sig-

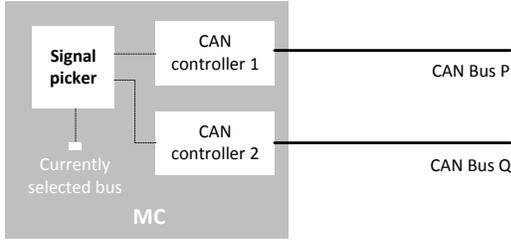


Fig. 4: Diagram showing “signal picker” module’s connections within the MC.

nal picker”, such as: receiving messages, simulating missed (dropped) messages, and initialising the periodic update of the state of the “signal picker”. The periodic update is triggered every 1.5s in the implementation, but is triggered non-deterministically in the model as the update could occur at any time relative to the messages being received on the bus.

The “signal picker” process model makes decisions about whether or not to swap buses whenever a new message is received. Received messages are non-deterministically determined to be either messages received without error or messages indicating sensor errors at the message source. The model also includes a check that the rules related to swapping to the redundant bus are not broken after a new message has been received or the state of the module has been updated.

IV. VERIFICATION OF MODELS

This section describes the correctness properties and failure scenarios developed to check against the “voter” and “signal picker” models, as well as the results of the verification.

A. “Voter” model

The LTL correctness properties checked against the model to gain confidence that the implementation of the voter procedure meets the designers original specifications are:

- 1) $\diamond \square (tm1_active \ \&\& \ !tm2_active)$ — The potential time-master node (Node 1) eventually always becomes the active time-master and Node 2 becomes the backup time-master.
- 2) $\diamond \square (tm2_active)$ — If Node 1 is disabled then Node 2 eventually always becomes the active time-master. In this test, after the initial startup sequence, Node 1 is disabled.
- 3) $\square \diamond !(tm1_active \ \&\& \ tm2_active)$ — Always eventually Node 1 and Node 2 are not both the active time-master.
- 4) $\diamond \square !(tm1_active \ \&\& \ tm2_active)$ — Eventually always Node 1 and Node 2 are not both the active time-master.
- 5) $\square !(tm1_active \ \&\& \ tm2_active)$ — Always potential time-master nodes Node 1 and Node 2 are not both the active time-master.

- 6) $\square \diamond (tm2_active)$ — Always eventually time-master node Node 2 is active when Node 1 is disabled following completion of the first basic-cycle.
- 7) Absence of violated assertions when the active time-master node (Node 1) is toggled on and off every 1.5 s.
- 8) Absence of violated assertions when the active time-master node (Node 1) is toggled on and off every 1.5 s, and an initial bootup delay of the faulty active MC is added.

The results of the verification of the correctness properties checked against the voter model are:

- Properties 1 – 4 pass verification without error.
- Property 5 fails verification. The error trails reveal two situations where both potential time-masters are simultaneously configured as the active time-master. On the initial startup of the system, both time-masters become the active time-master after an initial timeout; the voter state-machine then determines the winning time-master. Also, if an active time-master is powered down the state is stored in non-volatile memory and when the node is reactivated, if the backup time-master has taken over and become active, both nodes will be in this state.
- Property 6 passes verification with the backup time-master taking over as expected.
- Property 7 fails when the on/off period is 3 s (1.5 s on and 1.5 s off) and passes for all other intervals tested. This kind of fault could potentially be caused by a faulty alternator sending a voltage spike to the MC’s power supply, causing it to periodically switch on and off. The reason this period fails is because with the 3 s period the node is active just long enough so that the active time-master status message is able to be sent after startup, as illustrated in Figure 5. From tests of the MC hardware it was found the time-master status message is sent approximately 1.3 s after the node is powered up. This means, even with the 1.5 s periodic fault, the backup MC will think there is still an active MC on the network so will not timeout and takeover. In this case, 50% of the MC messages are effectively lost from the active MC due to the fault and the redundant MC will not takeover. An up down counter is used as a check to see if 1 or more messages have failed to be received at a node within a basic-cycle period. If this count becomes greater than 20 an error is flagged. This is the case in the model when the 3 s periodic fault is tested. The test shows that the redundant MC does not take over as expected in this situation.
- Property 8 passes when an extra delay is added to the time-master node.

B. “Signal picker” model

The following properties checked are based on the original developer’s design specification:

- 1) The module will only ever swap to the other bus if the current bus has received a message with an error value

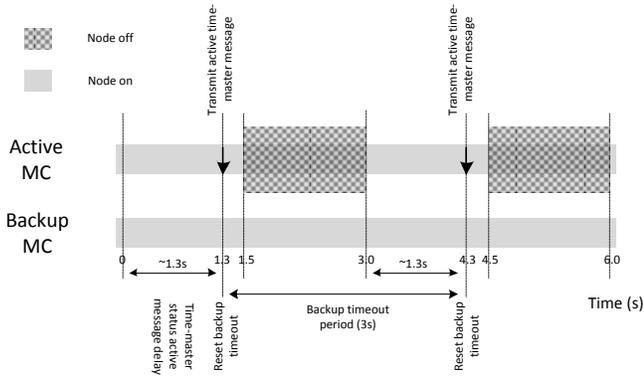


Fig. 5: Timing diagram showing the sequence of events that causes the backup time-master to fail to takeover as the active time-master as expected.

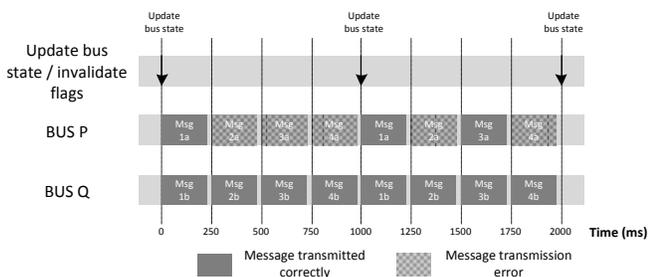


Fig. 6: Timing diagram for intermittent error on bus P.

or a message has not been received since the status flags have been last reset. There must be a message seen on the bus that will be swapped to and no error messages can be received on the bus. An assertion is used to check this property whenever the model swaps buses.

- 2) If a message is dropped since the last time the modules flags are reset a transition is made to the redundant bus. Again, an assertion is used to check this property.

Property 1 passes verification; however, property 2 fails. Messages can be dropped on the bus the module is currently listening to without swapping to the backup bus, as shown in Figure 6. The “signal picker” only swaps to the redundant bus if no message has been seen (received correctly) on the current bus or a message is seen with an error flag set. This means if there is an intermittent fault on the bus it is possible to lose a number of message since the last periodic reset of the modules flags. In this case, as long as one message is received the module will not switch to the redundant bus.

V. CONCLUSIONS

This paper shows an example of how model checking has been successfully applied to an industrial problem. The model of the “voter” procedure and correctness properties checked against it has uncovered an unexpected situation where an intermittent fault on the active MC prevents the backup MC taking over as expected. The “signal picker” model revealed a

situation where a fault at a certain rate on the active bus could prevent the MC from switching to the redundant bus. In the implementation, both these situations could potentially lead to a situation where the performance of the vessel is degraded or control of the vessel is lost. Due to the complicated sequences of events required to expose these problems, these issues may have been missed under the current test scheme. In uncovering these sorts of potentially hard to reproduce problems, the results become valuable during the test phase as it gives more insight into the design and the findings can be used to improve the test framework.

In applying model checking techniques to this implementation of TTCAN, we have found model checking to be a useful aid in the design phase of a large software project. The process of developing the models of the implementation allows a developer to gain a deeper understanding of the code. Simulations of the model often revealed complicated sequences of events that were not taken into account during the initial software design, and these can now be checked to ensure they do not cause the system to misbehave. Also, by creating a model of the software and automatically verifying it against a set of mathematically based design specifications, the developer can check the specifications against the implementation in a rigorous way. This gives the developer more confidence in the design, and that the design meets the specifications.

REFERENCES

- [1] Feather, M., Fickas, S. and Razermera-Marny, N.A., Model-checking for validation of a Fault Protection System, In Proc., 6th International Symposium on High Assurance Systems Engineering, 2001.
- [2] Holzmann, G.J., The SPIN Model checker: primer and reference manual, Addison-Wesley, 2004.
- [3] Stephan Merz, F. Cassez et al. (eds): Modeling and Verification of Parallel Processes. Springer-Verlag, LNCS 2067, pp. 3-38, 2001.
- [4] Ben-Ari, M. Principles of the Spin Model Checker, Springer London, 2008.
- [5] Leen, G. and Heffernan, D., TTCAN: a new time-triggered controller area network, Microprocessors and Microsystems Journal, vol. 26, no. 2, pp. 77-94, 2002.
- [6] ISO 11898-1, Road vehicles Controller area network (CAN) Part 1: Data link layer and physical signalling, International Standards Organisation, 2003.
- [7] ISO 11898-4, Road vehicles Controller area network (CAN) Part 4: Time triggered communication, International Standards Organisation, 2004.
- [8] Fuhrer, T., Muller, B., Dieterle, W., Hartwich, F., Hugel, R. and Walther, M., Time triggered communication on CAN (Time Triggered CAN - TTCAN), Seventh International CAN Conference (ICC), Amsterdam, Netherlands, 2000.