Runtime Observation of Functional Safety Properties in an Automotive Control Network

ABSTRACT
This paper exploits the observability of control messages in a control network to formally monitor safety properties to verify a control application’s correct behaviour. A monitor scheme is proposed based on a runtime verification method, which can verify selected properties of an application’s behaviour, including the verification of formally specified functional safety properties. A prototype hardware based circuit is developed to provide a monitor function. A case study example for an automotive gearbox control system is presented. The control application is evaluated in the target application environment, which is a controller area network (CAN) based network. The behaviour of the monitor is assessed and the results show that it is feasible to monitor and verify functional safety properties, as defined by the ISO 26262 standard for functional safety in road vehicles, using the proposed method.

Key words: Sensor networks; control networks; runtime verification; functional safety; embedded systems; automotive; ISO 26262

1. Introduction

Many control system designs are based on control network or sensor-level networks so that functionality can be distributed in space. Applications are found in industrial automation, transport vehicles, and many other fields. A large number of such small networks are employed in safety-critical applications and there are increasing demands from regulatory agencies and industries for product developers to ensure compliance with functional safety requirements.

Functional safety is an integrated part of the overall safety requirements for a product and is concerned with assuring that a system or equipment item operates correctly; taking into account the safe management of likely operator errors, system failures and environmentally induced problems.

This paper investigates the feasibility of monitoring safety property requirements for an application that runs in a control network based system, in real-time. The aim is to develop a specialised monitor device, which is a programmable hardware based circuit that observes the behaviour of an application in real time and reports on violations. The work suggests that conventional control network architectures lend themselves to the development of runtime monitoring schemes in a useful way where the various control messages are exposed on the network and the set of these messages can be interpreted in formal logic equations to inform an observer about the behaviour of some key properties. It is specifically proposed that such a monitor can be used to observe functional safety properties.

A case study example illustrates how such a monitor can be developed for an automotive application which is based on the controller area network (CAN) bus [1]. The case study example is based on a software controlled automotive gearbox, which can be seen as an important equipment item for safety evaluation.
Two distinct use cases are evaluated for the monitor employment. The first use case is where the development engineer carries out the verification process as part of the product development cycle, using the runtime monitor as a development tool. The second use case is where the runtime monitor is permanently installed in the system and is used to verify the correct behaviour of an application, through the lifetime of the product.

Based on the case study example it is demonstrated that it is feasible to verify functional safety properties for a product using such runtime verification. However, the value of using such a monitor for lifetime monitoring within a product has questionable benefits.

The remainder of this paper is organised as follows: section 2 describes some background for functional safety and runtime monitoring; section 3 summarises related research work; section 4 introduces the gear controller case study and its requirements; section 5 describes the case study implementation; section 6 discussed the evaluation and testing for the concept; and section 7 summarises the conclusions.

2. Background

Runtime monitoring for checking the performance behaviour of embedded systems’ applications is an established field. In this study the concept is extended to the monitoring of functional safety properties.

2.1 Functional safety

Functional Safety is achieved by product developers by designing and developing products to ensure all specified safety functions are implemented; and the level of performance required of each safety function is met. This is usually achieved by a development process that involves: identifying the formally required safety functions, based on risk assessment and management; assessment of the risk-reduction required by the safety function; ensuring the safety function performs to the design intent; and verifying that the system meets the assigned safety integrity levels. Then there is a requirement to conduct Functional Safety audits to assess evidence that the appropriate safety lifecycle management techniques are being applied consistently in the relevant lifecycle stages of a product.

2.2 The functional safety standards

Probably the best known functional safety standard in the electronics industry is the IEC 61508 functional safety standard and its automotive adaptation ISO 26262 [2]. The IEC 61508 document defines four general Safety Integrity Levels (SILs), where SIL 4 represents the most stringent safety level. The ISO 26262 document defines four Automotive Safety Integrity Levels (ASILs) where ASIL D is the most stringent safety level. Each level corresponds to the likelihood of failures for a safety function.

The ISO 26262 does not specify any development processes or technologies. Rather it assumes that development processes such as ISO 9001, CMM or similar process are already in place and imposes specific safety related requirements and outcomes on them.

2.3 Runtime monitoring

In this paper it is proposed that an independent monitor can be developed to monitor key safety properties during product runtime to confirm proper behaviour of an application during its execution phase. Such a runtime verification monitor is concerned with the monitoring of program execution behaviour so as to establish compliance with a requirements specification. The properties to be observed can be decided by the developers to gain confidence in the correct operation of the product or system.
The concept of the proposed runtime monitor makes the following key assumptions:

a) The runtime environment on its own cannot detect violations of an application’s safety properties. Traditional means of node diagnostics and network diagnostics cannot have awareness of an application’s specifications.

b) It is not sufficient to verify the behaviour of functional safety properties outside of the actual runtime environment. The verification of properties outside of the runtime context cannot make realistic assumptions on how the implementation environment can impact on the application’s behaviour in terms of functionality and timing.

c) The evaluation of the safety properties will not lead to erroneous conclusions resulting in false negatives. This is assuming the monitor itself does not fail and all runtime parameters that can impact on the verification are accounted for.

d) The target system will expose sufficient global variables to the network so that the monitor logic can be meaningfully expressed by observing the exported messages.

e) The assertion check process is sufficiently fast so that verification can be achieved in real-time while the system is executing.

In the proposed scheme the requirement specification of a system is stated as an executable specification that describes the behaviour of the system. That described system model is formally verified using a model-checker that can verify timed behaviour. Program code is generated from the specification model for product implementation. At runtime, selected properties that have already been formally verified in the model checker are evaluated. This evaluation during runtime is a form of verification that is based on assertion testing [3].

3. Related work

A runtime verification monitor is concerned with the monitoring of program execution behaviour to establish compliance with a requirements specification. The concept of monitoring system behaviour during runtime is well established as a means to employ a ‘lightweight’ formal verification method to assess runtime compliance behaviour in accordance with a product’s requirement specifications. Runtime monitoring methods have been proposed by Havelund and Roşu [4], Drusinsky [5], Havelund et al [6], and Sammapun [7], amongst others. The Monitoring and Checking (MaC) framework, by Lee et al [8] and Kim et al [9], proposes a scheme to automatically link low-level observations of program execution behaviour to the relevant monitored properties. PathExplorer (PAX) is a runtime verification tool by Havelund and Roşu [10] that uses linear temporal logic (LTL). Watterson et al. [11] provide an in-depth review in the context of the requirements for monitoring in embedded systems. Michael et al. [12] describe runtime execution monitoring schemes to assess whether formal assertions correctly capture the intent of some natural language requirements.

Many of the proposed methods to date are focused on solutions that perform on-line monitoring, and employ off-line processing of captured traces from the monitoring exercises. In the work described in this paper, a monitor device is proposed that can be embedded right into the product. The need for trace memory can be eliminated by using state by state evaluations during runtime.
Interest in evaluation for safety conformance in products spans many engineering disciplines as described by Saleh et al [13]. In recent times there is growing emphasis on the development of software related safety cases, as discussed by Hawkins et al. [14]. However, the concept of mapping specific safety requirements from a safety standard to runtime monitored properties is not well explored in the research literature. Clauses 7 and 8 of ISO 26262-6, which refer to software architectural design and software unit design respectively, each point to external monitoring techniques as a means of attaining ISO 26262 compliance, while the test case derivations covered by clauses 9 (Software Unit Testing) and 10 (Software integration and testing) lend themselves to a monitoring approach.

In automotive systems the need for behaviour monitoring is well understood, but solutions to the problems are incomplete. One significant outcome from research on these issues is the development of the AUTOSAR (AUTomotive Open System ARchitecture) [15] standard. AUTOSAR is an open, standardised software architecture for automotive E/E (Electrics/Electronics) systems; providing an infrastructure to assist with the development of in-vehicle software for the entire product lifecycle. Although the AUTOSAR recommendations are open to the inclusion of software monitoring, they do not suggest any formal approach. Lotoczky et al. [16] suggest a monitoring and debugging platform for AUTOSAR software components that reside on a target ECU, to assess the performance of a Software Component (SWC) and assist in debugging any anomalies discovered at this level.

4. Case study: requirements and the monitor logic

A case study is presented to illustrate the concept of monitoring defined system properties. A formal logic is selected to exactly specify the individual properties.

4.1 The case study

An automotive system, a gearbox controller, is considered for a case study example. This type of programmable E/E system falls within the scope of the ISO 26262 process for Functional Safety consideration. The software for the gearbox controller requires hardware on which to run, so the gearbox project software development is part of a gearbox system, including hardware and software components. The gearbox controller, therefore, requires an overall project plan, a safety plan, and a functional safety concept, including the ASIL level determination for the project.

As part of the product development process, the gearbox controller project will have a Functional Safety Assessment and a Functional Safety Audit leading to the definition of a set of Functional Safety Requirements for the development, as per Part 3 of the ISO 26262 standard. A technical safety concept and a set of technical safety requirements follow from these analyses. These analyses and assessments are included in the System Level Product Development and subsequently in the hardware and software product development.

The gearbox control module has the potential to cause significant harm, for example, by engaging an incorrect gear, or by failing to engage, but not immediate fatality, so for this example a notional ASIL level of C is suggested. In practice, determination of the appropriate ASIL level is a significant task for the project team, as part of the functional safety and technical safety activities mentioned above. These are the ultimate responsibility of the Safety Manager (as set out in Part 2 of the standard).

The use of an external monitoring facility to perform a ‘watchdog’ function is suggested here. In common with most embedded software development projects, ISO 26262 prefers testing of ‘production-intent hardware and software’ to ‘instrumented code’. The type of monitor
discussed in this paper supports the concept of observability of software performance using production code.

4.2 Choice of logic for the monitor
A key requirement of the monitor logic is to allow specific functional safety statements to be expressed as logic formulae that can be evaluated ‘on-the-fly’ so that the runtime behaviour of the system for such properties can be verified during a product’s normal operating phases. For the case study example, the past time linear temporal logic (ptLTL) was chosen. This is a simple modal temporal logic. Amir Pnueli [17] defined the linear temporal logic (LTL) which gives a powerful formalism for the concise specification of complex systems. A variant of LTL is ‘past time LTL’ (ptLTL) which refers to past states of an execution trace relative to the current state. An important feature of ptLTL for monitoring applications is that the satisfaction of a formula can be decided for the execution trace by evaluation of only the current state and its predecessor state \( s^{n-1} \). This negates the need for any special trace buffers, which can be costly and complex to implement. Figure 1 provides a brief summary of the ptLTL syntax and operators.

The ptLTL formula and syntax

The ptLTL formula \( \psi \) uses the following syntax:
\[
\psi ::= \text{true} | \text{false} | \text{AP} | \neg \psi | \psi \land \psi | \psi \lor \psi | \psi \rightarrow \psi | \psi \leftrightarrow \psi
\]

AP is the set of atomic propositions which represent the predicates. The standard propositional binary operators are used e.g. \( \land \) (conjunction), \( \lor \) (disjunction) etc.

Past time operators
The basic past time operators allow temporal reasoning and are listed in exhibit a). The monitoring operators are derived from the basic operators to provide a more intuitive and compact presentation, as shown in exhibit b).

a) Basic past time operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \square \psi )</td>
<td>‘always in the past ( \psi )’; i.e. ( \psi ) was always true in the past.</td>
</tr>
<tr>
<td>( \langle \rangle \psi )</td>
<td>‘eventually in the past ( \psi )’; i.e. ( \psi ) was true in a past state in time.</td>
</tr>
<tr>
<td>( \Diamond \psi )</td>
<td>‘previously ( \psi )’; i.e. ( \psi ) held in the immediately prior state. Past-time equivalent to the Next operator in LTL.</td>
</tr>
<tr>
<td>( \psi_1 S \psi_2 )</td>
<td>‘( \psi_1 ) strong since ( \psi_2 )’; i.e. ( \psi_2 ) held at some state in the past, since then ( \psi_1 ) has continued to hold true.</td>
</tr>
<tr>
<td>( \psi_1 S_w \psi_2 )</td>
<td>‘( \psi_1 ) weak since ( \psi_2 )’; i.e. ( \psi_1 ) was true all of the time, or else ( (\psi_1 S \psi_2) )</td>
</tr>
</tbody>
</table>

b) Monitoring operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \uparrow \psi )</td>
<td>‘start ( \psi )’; i.e. ( \psi ) was false in the previous state and is true in the current state. Equivalent to basic operators: ( \psi \land \neg \psi )</td>
</tr>
<tr>
<td>( \downarrow \psi )</td>
<td>‘end ( \psi )’; i.e. ( \psi ) was true in the previous state and is false in the current state. Equivalent to basic operators is: ( \neg \psi \land \psi )</td>
</tr>
<tr>
<td>( [\psi_1 , \psi_2 ]_s )</td>
<td>Strong interval operator. ( [\psi_1 , \psi_2 ]_s ) is true if ( \psi_2 ) has not been true since the last time ( \psi_1 ) was true, including the state where ( \psi_1 ) was true.</td>
</tr>
<tr>
<td>( [\psi_1 , \psi_2 ]_w )</td>
<td>Weak interval operator. ( [\psi_1 , \psi_2 ]_w ) is true if it satisfies ( [\psi_1 , \psi_2 ]_s ), or if ( \psi_2 ) was never true.</td>
</tr>
</tbody>
</table>
4.3 Monitoring timed behaviour
The ptLTL logic does not include features for the expression of physical time. In this case study example the monitor circuit includes physical clock/timer circuits in the hardware design, which are used for counting the progression of time to a resolution of 1 millisecond. Timer values are read at relevant times, in real time, and are used as variables for logic evaluations.

5. Case study: implementation
An automotive powertrain subsystem is developed to evaluate and demonstrate the feasibility of formally monitoring properties of a system based on sequences of network messages.

5.1 Gear controller example
The gearbox system design is based a prototype gear controller design, by Lindahl et al. [18]. The gearbox features five forward gears, a neutral gear and a reverse gear. The system uses a conventional controller area network (CAN) for communication between components. Figure 2a) illustrates a flow graph for the gear controller system which consists of five major components. The GearControl component represents the fundamental controller for the vehicle’s transmission.

The gear controller will change gear by requesting services from components in the system. A brief description of the components is presented:

Interface: The interface accepts requests from the driver's gear stick or an automatic gear-change algorithm. The Interface maintains status on the current status of the gear controller.

The Gear Controller: The gear controller receives message requests via the Interface component, over the control network. A gear change is achieved in the following steps:

1) accomplish a zero torque transmission, so that the current gear can be released
2) the current gear is actually released
3) the controller now adjusts for synchronous speed across the transmission
4) the move to the new gear is set
5) increase the engine torque to the same wheel tongue level as before the gear change

Under some driving conditions the engine might not be able to achieve zero torque or synchronous speed over the transmission, within defined time ranges. In this case the clutch is opened to facilitate the gear change. Once the clutch is open the path between engine and wheels is broken and the gearbox can then set the new gear as there now is no need for zero torque and synchronous speed. As the clutch physically closes it bridges the differences in speed and torque between the engine and the wheels.

The gearbox
The gearbox is expected to set a gear within the timeframe 100 to 300 ms and to release a gear in the timeframe 100 to 200 ms. If the maximum times are exceeded the gearbox will stop in an error state.

The clutch
This is an electrically controlled clutch which opens or closes within the timeframe 100 to 150 ms. If time bounds are not accomplished then an error state is reached.

**The engine**
The engine is used to establish zero torque and synchronous speed over the transmission. The engine has three modes of operation:

1) normal mode – engine provides the requested engine torque
2) zero torque mode – engines seeks to establish zero torque difference over the transmission
3) synchronous speed mode - engines seeks to establish zero speed difference between engine and the wheels

The maximum time bound to achieve zero torque is 400 ms. The maximum time bound to achieve synchronous speed is 500 ms.

![Flow graph diagram](image)

**Figure 2 Gear controller diagrams**

Table 1 shows a full list of the CAN messages that derive from the gearbox system design. It is this set of messages that form the inputs to the monitor circuit.
<table>
<thead>
<tr>
<th>Message</th>
<th>Semantics of the message</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReqNewGear (FromGear) (ToGear)?</td>
<td>Interface requests GearControl to change from current to a new gear</td>
</tr>
<tr>
<td>NewGear!</td>
<td>Gear change request is completed and now vehicle is in a new gear</td>
</tr>
<tr>
<td>ReqSet!</td>
<td>GearControl requests the GearBox to set a new gear</td>
</tr>
<tr>
<td>GearSet?</td>
<td>GearBox confirms that new gear is now set</td>
</tr>
<tr>
<td>ReqNeu!</td>
<td>GearControl requests the GearBox to set gear to neutral</td>
</tr>
<tr>
<td>GearNeu?</td>
<td>GearBox confirms that neutral gear is now set</td>
</tr>
<tr>
<td>OpenClutch!</td>
<td>GearControl requests the Clutch to open</td>
</tr>
<tr>
<td>ClutchIsOpen?</td>
<td>The Clutch confirms that the clutch is now open</td>
</tr>
<tr>
<td>CloseClutch!</td>
<td>GearControl requests the Clutch to close</td>
</tr>
<tr>
<td>ClutchIsClosed?</td>
<td>The Clutch confirms that the clutch is now closed</td>
</tr>
<tr>
<td>ReqSpeed!</td>
<td>GearControl requests the Engine to set speed</td>
</tr>
<tr>
<td>SpeedSet?</td>
<td>The Engine confirms that the speed is now set</td>
</tr>
<tr>
<td>ReqZeroTorque!</td>
<td>GearControl requests the Engine to achieve a zero torque state</td>
</tr>
<tr>
<td>TorqueZero?</td>
<td>The Engine confirms that the torque is now at zero</td>
</tr>
<tr>
<td>ReqTorque!</td>
<td>GearControl requests the Engine to set torque (i.e. it releases zero torque state, there is no message response expected in the design).</td>
</tr>
</tbody>
</table>

Table 1 List of CAN messages

Figure 2b) shows a block diagram for the gear controller implementation. A module called ‘Other CAN traffic’ is added to suggest there is other CAN message traffic on the same bus. The CAN bus data rate is set to 125 kbps in this case study example.

Figure 3 shows a timed automaton for GearControl component. The model is a simple finite state machine, which is extended with clock variables. The state machine was generated from a formal specification requirements document, using Uppaal [19]; a tool that is capable of modelling, simulating and verifying real-time systems. The resulting automaton for the GearControl represents a key part of the product’s requirements’ specification.
Figure 3 Timed automaton for GearControl component

Figure 4 shows an example of a natural language description for a required behavioural condition that represents a safety property that must be always respected when opening the clutch in a gear box system. The example is intended to illustrate the concept of precisely describing an English language statement in a formal logic.
Natural language description:
“The system must absolutely guarantee that the maximum allowed time for a complete gear change operation, to any forward gear, is 1000 msec, however, if the clutch is used during this gear change operation, then the maximum allowed time is not to be longer than 1250 msecs.”

The corresponding ptLTL formula:

\[ \Psi_1 := \uparrow ( \text{NewGear?} \land (\text{ToGear} > 0)) \rightarrow \]
\[ [ (\text{ReqNewGear}, \text{OpenClutch})]_s , \text{SysTimerExt} > 1000 \]_s \)
\[ \lor [ ! (\text{ReqNewGear}, \text{OpenClutch})]_s , \text{SysTimerExt} > 1250 \]_s \)

Figure 4  Example condition expressed in natural language and in ptLTL

5.2 The monitor hardware
A hardware-based runtime monitor circuit is developed to non-intrusively monitor the message traffic on the CAN bus so as to establish representations for runtime events, which model atomic propositions (AP). This logically represents a trace of runtime events as a finite word over the AP, and a monitor can check for inclusion in the language generated by the relevant ptLTL formulae.

Figure 5  The monitor connected to the CAN bus

Figure 5 illustrates a block diagram for the hardware monitor architecture. The monitor circuit, excluding the bus transceiver, is instantiated in the fabric of a single integrated circuit. The CAN bus is monitored by a CAN bus ‘sniffer’ component and messages of interest are captured, based on their known message identifiers. Note, this arrangement is a passive monitor to the CAN bus. For example, in the case of a corrupted received message, this node would not attempt to react by sending an error frame as dictated by the normal CAN protocol. The following additional components make up the monitor scheme:

Filter – an AP evaluator
The filter component is a logic circuit that performs on-the-fly evaluation of each atomic proposition. The filter is connects directly to the CAN bus sniffer and includes a logic unit to represent each AP element, which can include a CAN message, a message parameter, or a timer value. The filter outputs the AP set, for example the AP includes (SysTimerExt > 1250), (OpenClutch!), (ToGear > 0) etc.

**Event recogniser – state change update event detector**
The combination of the AP updates from the filter unit represents a state change, based on a reconstructed state diagram. The output from the event recogniser includes the atomic propositions.

**Runtime checker - formula verdict evaluation**
The runtime formula checker establishes that for each event the truth of a formula \( \psi \) can be decided in order to establish if a trace \( t \) satisfies \( t \vDash \psi \), up to the current state. Under a failure situation, the ‘Result’ output advises on which formula failed and this can be flagged to the user via a defined interface.

**Independent clock/timers**
Figure 6 shows the inclusion of some independent hardware clock/timers to the monitor circuit.

![Block diagram for the hardware monitor architecture](image-url)

**Figure 6** Block diagram for the hardware monitor architecture

5.3 Verifying properties
The specification of the gear controller design was reviewed, and some selected example safety statements, or requirements, were translated into properties which can be expressed as ptLTL formulae. The scope of the ISO 26262 Standard encourages this approach.

In the experimental system, twelve safety properties were defined. Two of the safety properties will be presented here as examples to illustrate the scheme. A formal equation is written to represent an intended safety property. Consider the following two stated safety requirements.

- **Gear change safety watch dog timer**
An example requirement states that the system must absolutely guarantee that the maximum allowed time for a complete gear change operation, to any forward gear, is 1000 msec, however, if the clutch is used during this gear change operation, then the maximum allowed time is not to be longer than 1250 msecs. The equivalent ptLTL formula is:

\[
\Psi_1 := \uparrow (\text{NewGear}\, \land \, (\text{ToGear} \, > \, 0)) \rightarrow \\
[ \, (\, \text{[ReqNewGear!} \, , \, \text{OpenClutch!]}_{\text{s}} \, , \, \text{SysTimerExt} \, > \, 1000)_{\text{s}} \, ) \\
\lor \, [ \, ! (\, \text{[ReqNewGear!} \, , \, \text{OpenClutch!]}_{\text{s}} \, , \, \text{SysTimerExt} \, > \, 1250)_{\text{s}} \, )]
\]

Using data generated during the configuration of the CAN bus messages, each message has a unique identifier, so message names can be used as symbols in the formula presentation, and can be easily interpreted by the monitor.

Note, time variables from the observed application are not exported as message variables on the CAN bus, so the monitor will need to have its own sense of time, based on independent clocks, as noted earlier. In the equation above the SysTimerExt represents one of the monitor’s own timers.

- **Safe behaviour of clutch operation**
Another example safety requirement states that the system must absolutely guarantee that it will not normally open the clutch, but only in the following cases: 1) when changing from one forward gear to another forward gear, and the zero torque is not achieved, or 2) when changing from one forward gear to another and zero torque is achieved, but then the synchronous speed is not achieved. The equivalent ptLTL formula is:

\[
\Psi_2 := \uparrow (\text{OpenClutch!}) \rightarrow \\
(\text{[FromGear} \, > \, 0) \, \land \, (\text{ToGear} \, > \, 0) ) \\
\land \, ( \, [ \, \text{RegNewGear?} \, , \, \text{TorqueZero?} \, ]_{\text{s}} \, \lor \, ( \, [ \, \text{TorqueZero?} \, , \, \text{SpeedSet?}]_{\text{s}} \, )
\]

5.4 **The monitor’s resources and scaling up**
In the prototype implementation for the monitor the number of logic cells required is proportional to the number of atomic propositions that are to be observed by the filter component, and on the number of ptLTL formulae that are to be evaluated by the runtime checker, i.e. the number of safety properties that are defined. However the inclusion of further safety properties does impact on the processing requirements overhead for the monitor as the evaluation time is prolonged. It is possible to design a dedicated processing model for the runtime checker that does not require a circuit implementation for each safety property and thus implies less hardware overhead on scaling up on the number of observed properties. Reinbacher et al.[20] propose such an architecture.

5.5 **Programming the safety properties**
The addition of a safety property involves taking a ptLTL formula as input and producing synthesizable VHDL code for a circuit which ‘checks’ that input formula. The code is then
processed in a synthesis tool for the target FPGA platform. The coding of the formulae is directly based on the algorithms presented by Havelund and Rosu [4].

5.6 Monitor evaluation time considerations
In a worst-case situation the monitor circuit will need to perform an evaluation during the time between the receipt events of two consecutive messages on the bus, where there is no time gap between the two messages. Considering the CAN bus example as used in the prototype example, the minimum size of the CAN frames used is 70 bits (excluding interframe spacing). At a bus bit rate of 125 kbps the time to send one message is 560 µsecs. For a 1Mbps CAN bus this message transfer time is 70 µsecs. The hardware monitor is capable of processing its evaluation within the 70 µsecs time slot. For faster networks however, for example fieldbus networks, this processing time would impose the need for some additional timing considerations.

5.7 Applicability for other automotive networks
The CAN network was chosen for the case study. Although CAN has been a dominant automotive control network for many years now, higher performance control networks such as FlexRay are used in applications that require higher-level features such as greater bandwidth, redundancy and more stringent real-time deterministic behaviour. The proposed runtime monitor concept can be applied in other such control network architectures where the control messages can be observed ‘on the fly’ by a network based node. The network topology, i.e. linear bus, star, hybrid etc., will not be an issue as long as the message set can be observed by a monitor node. However, restrictions can apply for some of the more advanced control network protocols which are now emerging. For example, the Time Sensitive Network (TSN) architecture is currently being proposed as a scheme that has the potential to become a future dominant standard for automotive networks. The TSN is a set of standards developed by the IEEE Time Sensitive Networking Task Group [21]. The standards define schemes and mechanisms for the time-sensitive transmission of data over Ethernet networks. The standard will provide explicit path control, bandwidth and stream reservation, redundancy for data flows and distribution of control parameters for time synchronization and scheduling. The authors are contributors to this standard [22] and claim that the message monitor scheme can be realised within individual streams which effectively represent a virtual network where messages are observable. However, if the control application spans multiple streams then a more complex solution to the observation of control messages will be required. For faster networks the processing time required for the monitor, i.e. within the latency time of a single message, will be more challenging.

5.8 The prototype evaluation environment
A prototype evaluation system was developed to support the experimental work. The system is illustrated in figure 7. Five experimental nodes form the gear controller; as shown in the diagram below the CAN bus network. This gear controller represents an automotive powertrain, which combines the following: GearControl, GearBox, Clutch, Engine and the Interface. Each one of these nodes represents a separate ECU that emulates the behaviour of a real component, e.g. the GearBox emulates the functionality of a real automotive gearbox. Each node is implemented based on the following technology:

- a single Spartan-II FPGA integrated circuit
- on-chip IP for a CAN controller core
- on-chip ECU application as modelled and formally verified on Uppaal tools
- an off-chip CAN transceiver

The safety property monitor node is implemented as a separate CAN node as seen on the upper part of figure 7 and includes the following features:

- a single Spartan-II FPGA integrated circuit
- on-chip IP for a CAN controller core
- on-chip implementation of the monitor as represented in figure 6.
- runtime checker is implemented as a program on an on-chip PicoBlaze core
- an off-chip CAN transceiver
- a serial interface to an external computer

The external PC computer, connected to the monitor node, is used to program monitor features and to store experimental results. An oscilloscope/analyser is attached to the CAN bus for independent monitoring of CAN message frames and evaluation of their timing behaviour. This is an Agilent 6032A (2 GSa/s) oscilloscope which has a built-in CAN 2.0A triggering facility.

![Diagram of the prototype evaluation environment](image)

**Figure 7** The prototype evaluation environment

### 5.9 The emulation models

There exist on the market sophisticated models for emulation of the engine and gearbox components that can be targeted to FPGA implementations. For example engine emulation models are available from National Instruments, dSPACE and others for use in the development of automotive ECUs (electronic control unit) to support the HIL (hardware in the loop) development process. However, in the prototype evaluation environment presented here much simpler models were developed to evaluate the concept of runtime evaluation. The node components are designed to provide programmable response times that allow the monitor scheme to be evaluated. For example the engine node reacts to the following signals, where the response time is the delay from receiving an input signal to providing an output signal:
The response time is pre-programmed for an experimental test run and a programmable variance feature is also implemented.

Note, the implementation of the CAN network in the prototype evaluation environment is a real CAN compliant implementation [1] and it is not an emulation. Thus the system behaviour of that network in relation to its timing, jitter, message latency, arbitration etc is realistic for the evaluations and testing. The CAN transmission bit rate is programmable up to 1 Mbit/s.

6. Evaluation and testing

For the evaluation of the proposed runtime verification scheme it is important to consider the fault models that apply to the various aspects of the system development and then to devise some suitable tests to assess the validity of the scheme.

6.1 Fault model for the gear controller application

The gear controller application is first considered in isolation where the implementation in the CAN embedded system is abstracted away so that the focus can be on the design and conformance of the core gear controller application. For the application, a design error is where an executable specification of a design differs from the designer’s intent. In this project the designer’s intent for the gear controller product is expressed as a requirement specification [18] which is an executable specification that describes the behaviour of the system. Formal model-checking is carried out for the design based on the Uppaal tool suite. A well-defined fault model is used to inform the testing, which includes the following features:

- State coverage for all state reachability
- Transition coverage to traverse all transitions
- Reachability properties to assess if a given state formula can be satisfied by any reachable state
- Safety properties to guarantee (invariantly) that some defined condition will never happen
- Liveness properties to guarantee that something will eventually happen, e.g. any message that has been sent should eventually be received

As stated in section 5 the overall gear controller design is implemented using five automata: GearControl, GearBox, Clutch, Engine, and Interface. The GearControl is centric to the design. The Uppaal tool, as a real-time model checker, was used to evaluate the full model, to include queries that cover all of the fault model cases. This provides confidence that the design was properly implemented in compliance with the specification, including all the timing requirements. The test results are provided in [18].

6.2 Fault model for the CAN implementation environment

A fault model for the implementation environment is assessed so that the impact of CAN based faults on the verification of the gear controller can be understood. The gear controller is

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Programmable response time</th>
<th>Resolution</th>
<th>Programmable variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReqSpeed</td>
<td>SpeedSet</td>
<td>50 to 350 msecs</td>
<td>+/- 50 nsecs</td>
<td>Yes</td>
</tr>
<tr>
<td>ReqTorque</td>
<td>None</td>
<td>500 msecs timeout</td>
<td>n/a</td>
<td>No</td>
</tr>
<tr>
<td>ReqZeroTorque</td>
<td>TorqueZero</td>
<td>50 to 200 msecs</td>
<td>+/- 50 nsecs</td>
<td>Yes</td>
</tr>
</tbody>
</table>
implemented in the test-bed environment as described in section 5. Consideration of a fault model for CAN might include the various processors, memories, CAN controller, CAN transceiver, watchdogs, and other hardware and software components. Faults such as: stuck-at-dominant, stuck-at-recessive, bit-flipping, babbling-idiot etc. - all form part of the fault model. Resulting error information will typically impact on the network bus where communications can be corrupted and destroyed; even though the CAN protocol provides elaborate error detection schemes, and enables a node to distinguish short disturbances from permanent failures, based on fault confinement mechanisms.

In this paper it will be assumed that the CAN network and associated node processors are functioning correctly so the focus is on assessing the behaviour of the gear controller application in the CAN implementation environment, so that a safety manager can verify correct behaviour of an application based on the assumption that the underlying network behaves correctly. In other words the concern on CAN related faults is separated from the concern on the application’s behaviour for the purposes of verifying the behaviour of functional safety properties. Of course, CAN related failures will need to be understood and addressed in the overall safety certification of a full system.

However, even with the assumption that a CAN system is working properly there still remain some important issues that will affect the functional performance of the hosted application. Some of the key issues are as follows:

- In spite of the inherent error detection schemes in the CAN protocol, there still remains a total residual error probability for undetected corrupted messages, which is calculated [1] to be less than: (message error rate * 4.7 * 10^{-11})

- CAN is not a time-triggered network so contention for bus access gives rise to significant jitter times which can impact on the determinism of any hard real-time applications.

6.3 Fault model for the actual monitor node

Vehicle design experience shows that adding fault protection to systems often increases complexity and thus the likelihood of a failure, where the fault protection logic itself can contribute to the failure. The monitor node and its potential unintended effects must be considered. Some of the following faults are possible:

- CAN related faults at the monitor node, including: transceiver faults, possible erroneous transmission by CAN controller etc.
- A false detection of a failure leads to an undesired system reaction with negative safety consequences.
- Potential inconsistency can arise as CAN message errors in the last bits of the frame can force some nodes to accept a message while others reject it.
- The real-time clock may drift or fail in other ways.

6.4 Functional testing

Functional testing was used to validate the correct functional behaviour of the gear controller application in the test-bed system and included the following tests:

- All combinations of forward gear changes for conditions of no clutch activation
- All combinations of gear changes for conditions without clutch activation
- All combinations of reverse gear select with and without clutch activation

The emulated engine and clutch nodes contain timers that create simulated delays for parameters such as engine speed change, wheel torque change and clutch engage/disengage times. In normal behaviour there is some randomisation of these delays used to emulate real behaviour, within a predefined variance. There is the facility to program these delays to specific time values.

6.5 Faults induced by the injection of deliberate programmed violations
In the design of the test strategy a number of features of the observed program were modified so that the system would behave in a manner that would violate some specific requirements. In this way the behaviour of the monitor in its ability to detect such violations was assessed. Ten such tests were devised, where table 2 illustrates four example deliberate violations which are detectable by the monitor.
<table>
<thead>
<tr>
<th>Test No.</th>
<th>Nature of the test</th>
<th>The program violation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>T001</td>
<td>Test for an abnormally slow clutch operation</td>
<td>An error was injected by programming an additional 500 msecs delay into the clutch function.</td>
<td>Formula evaluation successfully detected the violation as the total gear change operation exceeded 1250 msecs</td>
</tr>
<tr>
<td>T002</td>
<td>Test for correct conditions for operating the clutch</td>
<td>An error was injected in the program by calling a clutch open operation even though the system had achieved zero torque.</td>
<td>Formula evaluation successfully detected the violation</td>
</tr>
<tr>
<td>T003</td>
<td>A programmed error requests to open the clutch too early, before torque is checked.</td>
<td>In S1, OpenClutch! is signalled.</td>
<td>The monitor’s ptLTL formula detects the violation and the error result is flagged.</td>
</tr>
<tr>
<td>T004</td>
<td>Although a reverse gear is requested, a programmed error during runtime requests a sync speed to the engine.</td>
<td>In S5, ReqSpeed! is signalled.</td>
<td>The monitor’s ptLTL formula detects the violation and the error result is flagged.</td>
</tr>
</tbody>
</table>

Table 2 Example fault injections by program violation

6.6 Timing Tests
A number of timing tests were performed so as to functionally assess the real-time behaviour compliance of the gear controller system. One exemplary test is reported here; which is referred to as the GearChangeFirstSecond test. The test was configured as follows:

1) The set up was as shown in figure 7.
2) The Interface module initiates a ReqNewGear message to change from first to second gear. The gear change operation concludes with the Interface receiving the NewGear message.
3) The tests were run, under programmed control, for thousands of iterations for various combinations of programmed response times.

Three selected test cases, which have interesting programmed response times, are listed in Table 3. The Test Case 1, in column 2 of the table shows the upper limits for response times, as specified in the requirements specification [18]. The Test Case 2 in column 3 shows responses times which are set at 97.5% of the upper limit values and Test Case 3 in column 3 shows responses times which are set at 95% of upper limit values. The final row of the table shows the result status for each of the three test cases. The results show that there are some failures. The three test cases can be summarised as follows:

**TEST CASE 1 (Maximum limit response times)**
All response times are preset to the maximum limit response times of the requirements specification.

Result: On repeated runs of the test, the gear change failed on each run.

Observations: On issuing ReqZeroTorque, the GTimer value exceeds 250 ms causing a transition to CheckClutch2 state and then the GTimer value exceeds 150 ms causing a transition to the COopenError state.

**TEST CASE 2 (Maximum response times - less 2.5%)**

18
All response times are preset to 97.5% of the maximum limit response times of the requirements specification.

**Result:** On repeated runs of the test, the gear change failed on each run.

**Observations:** On issuing `ReqZeroTorque`, the `GCTimer` value exceeds 250 ms causing a transition to `CheckClutch2` state and then the `GCTimer` value exceeds 150 ms causing a transition to the `COopenError` state.

**TEST CASE 3 (Maximum response times - less 5%)**

All response times are preset to 95% of the maximum limit response times of the requirements specification.

**Result:** On repeated runs, the gear change is correctly performed each time. The clutch is opened during the gear change but this is expected for the defined set of response times.

<table>
<thead>
<tr>
<th>Signal/Response</th>
<th>Test case 1</th>
<th>Test case 2</th>
<th>Test case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReqSpeed/ SpeedSet</td>
<td>Top limit for all response times msecs.</td>
<td>500</td>
<td>487.5</td>
</tr>
<tr>
<td>ReqTorque/</td>
<td>Undefined</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ReqZeroTorque/ TorqueZero</td>
<td>400</td>
<td>390</td>
<td>380</td>
</tr>
<tr>
<td>ReqSet/ GearSet</td>
<td>300</td>
<td>292.5</td>
<td>285</td>
</tr>
<tr>
<td>ReqNeu/GearNeu</td>
<td>200</td>
<td>195</td>
<td>190</td>
</tr>
<tr>
<td>ReqNewGear/ NewGear</td>
<td>1205</td>
<td>1175</td>
<td>1145</td>
</tr>
<tr>
<td>OpenClutch/ ClutchIsOpen</td>
<td>150</td>
<td>146.25</td>
<td>142.5</td>
</tr>
<tr>
<td>CloseClutch/ ClutchIsClosed</td>
<td>150</td>
<td>146.25</td>
<td>142.5</td>
</tr>
</tbody>
</table>

**RESULT**

<table>
<thead>
<tr>
<th>Test case 1</th>
<th>Test case 2</th>
<th>Test case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fails always</td>
<td>Fails always</td>
<td>Never fails</td>
</tr>
</tbody>
</table>

**Table 3 Test cases for timing evaluation**

The Test Case 1 failure is predictable as the test is conducted on the limit of the timing specification. However, it is important to understand why failure occurs when the system is operated within the timing specifications, as in Test Case 2, which allows a 2.5% margin on the maximum timing limits. The application program has already been verified in the Uppaal formal model to show it to be compliant with respect to the timing requirement specifications. The reason for the non-compliance relates to the CAN’s message latency and jitter behaviour. Table 4 shows some actual readings where the influences of message latency time overhead and jitter are measured. The time measurement shown is the interval between two CAN bus messages: the response time from the receipt of a `CloseClutch` message to the receipt of the corresponding `ClutchIsClosed` message.

<table>
<thead>
<tr>
<th>Actual response time µs</th>
<th>Specified response time µs</th>
<th>Time difference µs</th>
<th>Suggest explanation of difference in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>152,153</td>
<td>150,000</td>
<td>2,153</td>
<td>Overhead of two required CAN messages</td>
</tr>
<tr>
<td>153,291</td>
<td>150,000</td>
<td>3,291</td>
<td>Overhead of two required CAN messages  + one higher priority messages</td>
</tr>
<tr>
<td>154,407</td>
<td>150,000</td>
<td>4,407</td>
<td>Overhead of two required CAN messages  + two higher priority messages</td>
</tr>
<tr>
<td>155,519</td>
<td>150,000</td>
<td>5,519</td>
<td>Overhead of two required CAN messages  + three higher priority messages</td>
</tr>
</tbody>
</table>

**Table 4 Measured response times**

As stated in section 5 a CAN message will have a transmission latency time of some 1.04 ms. Thus the two CAN messages add an overhead message delay time of 2.08 ms. Furthermore,
jitter in the CAN network occurs as a CAN controller may need to wait for higher priority messages to be sent before a lower priority message can be sent.

A note on CAN timing calculation
In the prototype implementation CAN messages are 8 bytes long. The length of the formatted CAN frame can be calculated. The maximum CAN frame size is not the simple arithmetic sum of all the bits in the frame. The CAN protocol imposes some additional bits to support a bit stuffing technique where a hamming distance of six is imposed on most of the fields in the CAN frame to ensure proper clock synchronisation. The upper bound on frame size may be calculated using the equation below:

\[ \left\lfloor \frac{34 + 8S_m}{5} \right\rfloor + 47 + 8S_m = \text{Maximum Frame Size (bits)} \]

In the equation, \( S_m \) is the number of bytes in the CAN data field for some message \( m \). The value of \( m \) is assumed to be 8 in this application. A ‘Standard CAN’ frame format is assumed. The integer floor function in the equation is used to calculate the number of stuff bits. There are 47 other bits in the CAN frame. From this equation, the maximum frame size is thus 130 bits.

At a CAN bit rate of 125 kbps (8 μs bit time), such an eight-byte message will take (130 * 8) μsecs, or 1.04 ms to transmit on the bus. Note, the 1.04 ms is the theoretical maximum transmission time. The values in Table 5 above suggest some variance on the actual values. The actual CAN message transmission time will be influenced by the number of actual stuff bits and by jitter associated with the CAN controller’s internal operation.

Furthermore, Davis et al [23] show response time analysis for CAN for calculation of the worst-case response time for each message, within the full priority based message set on the network. These values can then be compared to the message deadlines to determine if the system is schedulable.

6.7 Conclusions on test and evaluation results
The testing to date shows that it is feasible to formally verify properties of an application at runtime using the monitor scheme. The actual functional safety properties can be formally verified, first by a model checker and then in the real application environment.

The test and evaluation conclusions need to be assessed separately for the two distinct proposed use cases for the runtime monitor. These use cases are as follows:

Use Case #1
The runtime monitor is used to verify the correct behavior of the gear controller application, which is operating in the CAN system. This task is carried out by the verification engineer as part of the product development cycle. Any detected failure conditions are observed by the engineer. Live test and measurement instruments are connected to the system under test as illustrated in figure 7.

Use Case #2
The runtime monitor is permanently installed in the system and is used to verify the correct behavior of the gear controller application, through the lifetime of the product. The system must react to any detected failure condition to bring the system to some defined safe state. There is no live test equipment connected to the system.

Table 5 lists six major concerns that have been highlighted during the evaluation of the runtime monitor concept in the context of the case study. For the Use Case #1 the monitor can
successfully observe the functional behavior of the application and can verify the safety properties. The underlying behavior of the CAN network can be observed using test equipment and the concern issues with the network can be abstracted away from the runtime verification testing for the application. In other words, the impact of message latencies can be known for any test run and thus the time impact can be factored into the timing calculations. In the case of message errors during testing, a test can be re-run. This will allow the engineer to verify the behavior of the gear controller application in spite of violations that are introduced by the operation of the underlying network; thus separating the concerns for the application and for the host environment.

However, the situation for Use Case #2 is very different. There are some significant concerns relating to the use of such a runtime monitor for product lifetime monitoring as the influence of the CAN network behavior cannot be separated from the measurements made by the monitor node.
<table>
<thead>
<tr>
<th>Concern</th>
<th>Nature of concern</th>
<th>Suggested actions USE CASE #1</th>
<th>Suggested actions USE CASE #2</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual error in CAN</td>
<td>A ‘healthy’ network has a residual message error rate (4.7 \times 10^{-11})</td>
<td>Observe errors with test equipment and rerun verification.</td>
<td>Safety Manager must decide if CAN reliability model is fit for purpose.</td>
<td>Use Case #1 has merit, in spite of residual error.</td>
</tr>
<tr>
<td>Message latency and jitter</td>
<td>Network latency jitter is significant and impacts on accuracy of verification.</td>
<td>Measure latency with test equipment and factor out of measurements.</td>
<td>Measurement margins need to be expanded. This may not be acceptable by Safety Manager.</td>
<td>Use Case #1 has merit. Use Case #2 has questionable merit.</td>
</tr>
<tr>
<td>Intrusiveness of monitor faults</td>
<td>The monitor itself can potentially introduce faults in the system.</td>
<td>Observe monitor faults with test equipment and rerun verification.</td>
<td>This is a serious reliability concern.</td>
<td>Use Case #1 has merit. Use Case #2 has reliability concern.</td>
</tr>
<tr>
<td>Message observation point consistency</td>
<td>There are known issues with CAN that lead to inconsistency of message observation.</td>
<td>Observe inconsistencies with test equipment and rerun verification.</td>
<td>This is a serious reliability concern as multiple monitoring is too complex.</td>
<td>Use Case #1 has merit.</td>
</tr>
<tr>
<td>Underlying CAN errors and failures</td>
<td>There are known fault models for CAN which give reliability concerns.</td>
<td>Safety Manager must decide if CAN reliability model is fit for purpose.</td>
<td>Safety manager must decide if CAN reliability model is fit for purpose.</td>
<td>This is a separate issue from the use of a monitor. The monitor cannot improve the reliability model.</td>
</tr>
<tr>
<td>Reaction to detecting an error.</td>
<td>A reaction response to a monitor detected fault can itself be a serious safety concern.</td>
<td>Not a concern as a reaction feature is proposed.</td>
<td>Requires the reaction feature to consider in initial requirement specifications...</td>
<td>Use Case #1 has merit. Major concerns are introduced with Use Case #2.</td>
</tr>
</tbody>
</table>

Table 5 Concerns resulting from the test evaluation work

7. Conclusions

A new concept has been proposed whereby defined functional safety properties for an application can be formally monitored during the operating phase of a system, so that the application’s behaviour can be formally verified in its runtime environment. The ISO 26262 standard for functional safety in road vehicles is used to guide the definition of the functional safety requirements. A monitor solution was developed that can observe safety properties in real time. The monitor can be realised on a single integrated circuit with relative modest gate count requirements. A scheme is proposed where a control network’s messages are observed and the sequencing and timing of such messages will accurately infer the proper system behaviour for selected properties. By this means no internal access to the system’s variables is required and the system is not aware of the monitor’s presence.

A case study example for an automotive gearbox control system has been presented, which demonstrates the feasibility of the scheme. Example safety properties were defined and they were evaluated during the operating phase of the system. The timing behaviour of the monitor...
was evaluated and the proposed method is shown to be feasible for real-time operation for a typical control network. The monitor can process its evaluations within a single message transfer time on the bus. Considerations so far have been focussed on relatively low-speed control networks. However, the concept will be extended to faster fieldbus and automotive networks.

The results from the evaluation testing highlights the need for designers to take underlying network timing behaviour into consideration in the system specification phase; as a system’s real-time behaviour can be significantly influenced by the network operation.

Two distinct use cases have been evaluated for the monitor deployment. The first use case is where the development engineer carries out the verification process as part of the product development cycle, using the runtime monitor as a development tool, and this is shown to provide a useful verification assessment for functional safety properties. However, the second use case, where the runtime monitor is permanently installed in the system and is used to continuously verify the correct behaviour of an application during product lifetime, is a much bigger challenge as the underlying network latencies and jitter characteristics make it difficult to measure real-time behaviour. Further, for the second use case, a decision on how to react to a safety violation raises complex safety questions which still need more investigation.

The key aim of the research work has been met where the results have demonstrated that it is feasible to develop a monitor to verify functional safety properties in real time. However, there is still a lot of work to do as various questions have been raised during the course of the work. Further research will investigate the use of better methods to implement a formal logic and a set of tools that will allow a seamless development cycle from the requirements specification to the automatic generation of a monitor circuit.

References


