Automotive Databus Simulation using VHDL

Karen Hale

Lucas Advanced Engineering Centre

Abstract

VHDL has been used to develop a simulator for automotive databus networks. This is a design tool for early assessment of system interactions. VHDL has proved ideal for this application due to the flexibility of modelling permitted by features such as generics and configurations. Analogue extensions to VHDL are eagerly awaited for extending the capabilities of the simulator.

Introduction

VHDL has been widely used as a hardware description language at both the register transfer level (RTL) and at the functional level. It has been used rather less at the system level.

This paper introduces the application area of automotive databusses and the subsequent need for a system simulation tool. A discussion of the benefits of using VHDL as a language to develop such a simulator is included and the features and structure of the resulting simulator are presented.

Automotive databusses

Over recent years there has been a rapid increase in the amount and complexity of electronics on vehicles. The days have gone when radios, electric windows, central locking and anti-lock brakes were considered a luxury, available only on the most expensive models of car. Today these items are no longer seen as out of the ordinary. A car is expected to at least have the option for all of these things and more. Items such as telephones and CD players are also required to be easily incorporated into a vehicle as and when required.

Today's vehicles still make use of the traditional wiring harness to provide the interconnection between these systems. This is now reaching its limit in terms of the demands which can be placed upon it. Its size is becoming unwieldy and it is not sufficiently adaptable to allow for every new item on an 'as you like' basis. It has also been recognised that the wire harness is costly. These factors have led the automotive industry to consider possible alternatives. These alternatives are databusses with automotive specific protocols. Several protocols have now been developed and standardised, for example CAN[1][2], J1850[3].

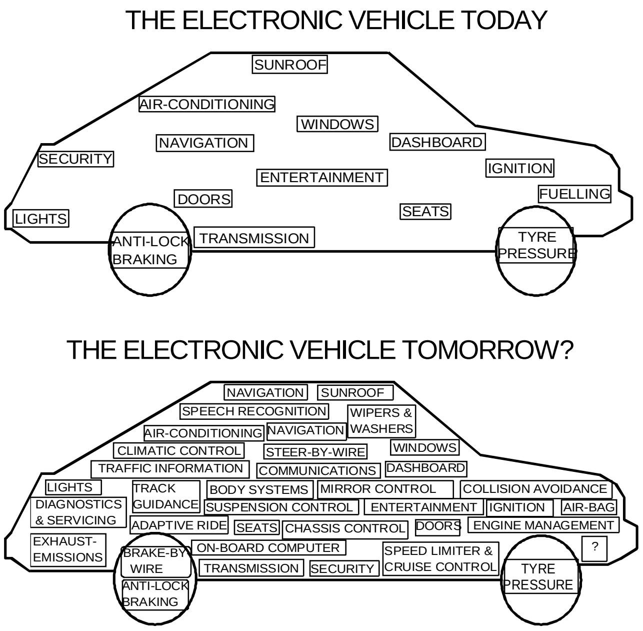

Figure 1. Vehicle electronics

Networks

Networks are a familiar concept in the modern office, and indeed in aircraft. Systems communicate with each other over a network of dedicated cabling equipped with connection points. Each system connected to the network can communicate with other systems by using an appropriate protocol. A databus provides the opportunity for less wiring and greater flexibility within a vehicle. Data from sensors can be sent to multiple destinations. The same physical network can be used as the communication medium between different units and different combinations of units. Individual units (e.g. sensors, controllers) can be replaced by alternatives without the need for new wiring. As new facilities become available for use within a vehicle, these can be introduced into the existing network without the need for costly and time-consuming wiring changes.

In addition to the 'comfort' benefits, there are also obvious safety advantages. The failure of particular sensors can be detected through the sophisticated diagnostics which become possible with a network. There is the possibility of temporary data sharing. This would allow information from an alternative source to be used until the vehicle was brought into a safe state, or serviced, depending on the failure. The remedial action required, or the severity of the failure, could be indicated to the driver. There are also possibilities of allowing the control of particular functions to be taken over by other control units if a serious failure was detected. For instance, if the engine controller were to fail, another controller may be able to provide reduced but safe control. This would reduce the consequences of the failure of particular controllers.

Message priorities

With a dedicated wiring system such as an automotive wiring harness, information is sent from its source directly to its destination. The time it takes for the information to reach its destination is known and is not subject to great variations. Replacing such a system with a network and communication protocol results in there no longer being any guarantees as to how long it will take for data to reach its intended recipient. The various units attempting to transmit and receive messages along the same physical medium must co-operate with each other as to who has the authority to transmit the next message. This arbitration is achieved by assigning priorities to messages. An identifier attached to a message indicates its source and destination. It also assigns it a priority. The communication protocols ensure that if more than one message requires to pass along the bus at the same time, the one with the highest priority will win control of the bus. The difficulty comes in selecting which messages should be assigned the highest priority.

It is quite clear that the command to turn on the sidelights does not need to be transmitted as urgently as a message to apply the brakes. There are many other situations which are far from straightforward. For example, is the information which is sent to or from the braking system always more important than that sent by the engine? How long is it reasonable to delay the message to turn on the lights? Or the radio? A simulation capability is clearly required in order to address such issues, and to determine which protocol is most suitable.

History

A few years ago, Lucas developed a databus simulator for use in the development of vehicle databus protocols[4]. However, the environment in which this simulator was written is no longer maintained and the simulator itself has ceased to fulfil our needs. Whilst the protocols themselves are now well-understood, the issues of message delays and system interactions still remain to be tackled. Understanding these issues is vital to the successful full-scale introduction of databus technology into vehicles. Proposed systems are becoming ever more complex, with more and more units being suggested for inclusion on a network.

The importance of understanding the effect of adding 'just one more' low priority controller cannot be overstated. The extra controller itself may be able to cope with the message latencies which will exist, but the extra traffic on the bus may cause another unit to exhibit unsatisfactory behaviour because it no longer receives information at an acceptable rate. Simulation is clearly the quickest and most cost-effective way of assessing the effects of such actions. In some cases simulation is the only way of investigating what the outcome of a particular scenario is likely to be; for example when assessing what will happen when a particular failure occurs. Thus there is a need for a simulator which is extendable and not tied to a single environment.

VHDL

A databus network is essentially discrete in nature, making VHDL an ideal candidate to use as the modelling language. The non-proprietary nature of VHDL, and its existence as an international standard also add to its suitability for developing a long-term tool. An additional attraction is the existence of an analogue extensions working group. Many of the systems which will interface to a vehicle network via digital controllers are continuous systems. Whilst digital approximations to these systems can be modelled, this is not ideal, and the inclusion of analogue modelling capabilities in the future will allow more detailed modelling of the effects of a network on the subsystems. The modular nature of VHDL also allows great flexibility of modelling, ensuring that any future extensions to the simulator can be made with a minimum of effort.

Simulator

An automotive databus simulator is a design tool to aid system understanding and development. It allows the implications on performance of particular architectures to be assessed. Additionally it allows the effects of using different protocols to be addressed.

One of the great strengths of VHDL is the ability to model components at different levels of abstraction, and to mix these within a simulation. Component interactions can be studied in detail, and once understood simplified models of this behaviour can be developed, allowing simulation effort to be concentrated in other areas. In the case of automotive databusses, the simulation of each bit in the protocol can be performed, or a simplified model of the protocol behaviour which refers to messages by their type can be used. For example, a behavioural protocol model would refer to a 'data message' whereas a bit-level model would use a bit-pattern.

The ability to mix the levels of simulation will become increasingly important as the proposed systems grow in both size and complexity. There will undoubtedly be times when there is apparently some 'odd' behaviour which will require close scrutiny. The reasons for such unexpected behaviour may only become apparent from the study of the low-level interactions which are taking place.

At the same time, the ability to incorporate more generalised descriptions into a simulation lead to the ultimate possibility of complete vehicle simulation. The arrival of analogue extensions to the language will bring this a step closer.

Modelling a databus network

A databus network is made up of a databus and various nodes. Each node has a protocol specific 'unit' which interfaces to controllers and sensors. These units are duplicated at each node in the network, whilst the controllers and sensors with which they interact are not necessarily the same, or even similar to each other.

Databus protocols are generally built in layers, as defined in ISO's Reference Model for Open Systems Interconnection (OSI)[5]. Such protocols can be modelled in the same hierarchical layers in VHDL, allowing future modifications to the specification of any layer to be incorporated quickly and easily.

In a practical system, an implementation of the protocol will supply additional features, e.g. sensor and actuator interfaces together with appropriate input and output buffers. Accordingly an 'ECU' (electronic control unit) model has been developed for network modelling. This provides a protocol model, simple buffering and an interface to sensors and actuators. The ECU model can be reproduced any number of times for a single simulation, each instance being customised for its particular situation by the use of generics.

Sensors provide the data for a bus, and actuators are the ultimate recipients of the data. Models for these components, as with the ECU model, can be replicated and customised as required by a simulation. The modularity of VHDL and the ease with which different entities and architectures can be included or removed from a simulation has allowed us to develop several generalised models for sensors and actuators. These can be customised with parameters such as identifier, name and repetition rate.

The protocol itself does not add time indicators to its messages. However, in order to collect useful data such time-stamps have been added to the messages. This does not affect the modelled transmission times since it is a parameter which is associated with a message, and not an integral part of the message itself. A non-intrusive monitor has also been developed to monitor the traffic on the bus.

Layered protocol model

Automotive databus protocols are specified in layers, as defined by the ISO 7-layer model. The data link layer and the physical layer are specified according to the ISO 8802.2/3 Local Area Network Standards. All other layers of the OSI model do not have counterparts within the definition of the CAN protocol but form part of the user's level.

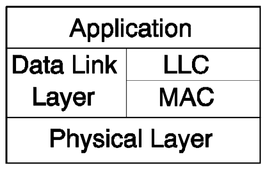

Figure 2. The OSI layers of the CAN protocol

The data link layer is formed of 2 parts- the logical link control (LLC) and the medium access control (MAC). The LLC layer is concerned with message acceptance filtering, overload notification and recovery management. Working under this, the MAC layer is concerned with message frame coding (bit-stuffing and de-stuffing), error detection, error signalling, acknowledgement, data encapsulation and decapsulation. Finally the physical layer is concerned with bit representations, timing and synchronisation.

The hierarchical nature of VHDL has allowed the protocol to be modelled in these layers. Thus a MAC layer entity instantiates a physical layer entity, and an LLC layer entity instantiates a MAC layer entity. A model of an ECU therefore requires an instantiation of the LLC layer- the top of the data link layer- along with suitable buffers for storing actuator and sensor data. The modelling of the physical layer makes use of enumerated types for data transfer to the MAC layer- 'recessive', 'dominant' and 'fault' bits- with appropriate resolution functions. Once the data has reached the MAC layer, record types are used for inter-layer communications. For example, communication between the LLC and MAC layers involves the transfer of signals defined as 'llc_mac_frame' and 'mac_llc_frame' types. The 'llc_mac_frame' type is for communication from the LLC layer to the MAC layer, and is a record with the fields 'command_field', 'identifier', 'data_length', 'data_field', and 'time_stamp'.

Access types

An apparently little used feature of VHDL has proved very useful in the modelling of particular entities. Access types provide powerful programming language type operations. They are similar to pointers in languages like C, and act as an address or handle to a specific object.

The functions NEW and DEALLOCATE are automatically available to any object which is declared to be of an access type. The first of these, NEW, allocates an area of memory of the size of the object in bytes, and returns the access value. DEALLOCATE is supplied with an access value and returns that area of memory back to the system. This allows objects of a dynamic nature to be modelled; for instance anything which requires some form of list to be created and maintained.

Access types have been employed in the modelling of a FIFO (first-in, first-out) buffer which is assigned to each of the network nodes. This is a system level model of an infinite-size buffer. This allows all information which is generated for the network to be monitored. The build-up of messages can be watched, and aid the choice of buffer size for an implementation. Additionally it is possible to see if there are some messages which never get launched onto the bus.

An example of the use of access types is given below, for entity FIFO, which uses linked lists.

```
ARCHITECTURE a1 OF fifo IS                      --
  TYPE fifo_store;                              --Declarations for type fifo_store
  TYPE fifo_ptr IS ACCESS fifo_store;           --as a record and fifo_ptr as an
  TYPE fifo_store IS                            --access type for fifo_store.
  RECORD                   --That is, fifo_ptr is an address, or handle, for a
   value :frame;           --fifo_store. Notice also that the incomplete type'
   prev_ptr : fifo_ptr;    --'feature of the language is employed. This allows
   next_ptr : fifo_ptr;    --the type fifo_store to be used in the definition of
  END RECORD;              --fifo_ptr before it has been defined. Incomplete
                           --types are needed whenever self-referencing
  BEGIN                    --structures, such as linked lists, are to be used.
   PROCESS
    VARIABLE fifo_new: fifo_ptr :=
          NEW fifo_store'(empty_frame, NULL, NULL)           --LINE 1
    VARIABLE fifo_old : fifo_ptr := fifo_new;
   BEGIN
      :
    IF data_in'EVENT THEN
     fifo_new.value := data_in;                              -- LINE 2
     fifo_new.next_ptr :=
          NEW fifo_store'(empty_frame, NULL, NULL);          -- LINE 3
     fifo_new.next_ptr.prev_ptr := fifo_new;                 -- LINE 4
     fifo_new := fifo_new.next_ptr                           -- LINE 5
    END IF;
      :
  END a1;
```

Figure 3. Example of the use of access types

In figure 3., line 1 declares the variable fifo_new to be of the type 'fifo_ptr', i.e. it is an access type. It is initialised to allocate a memory location for a fifo_store which is empty. The variable fifo_old is initialised to contain to the same value as fifo_new, that is, it points to the same fifo_store object.

On line 2 the value of the access types is updated. When an event occurs on data_in, the 'value' field of fifo_new is updated to contain the value of 'data_in'. Line 3 assigns the 'next_ptr' field to contain the access value for a new second fifo_store, which is empty. The second fifo_store created now requires it's 'prev_ptr' field to be updated to contain the access value for the fifo_store which was originally created. The 'prev_ptr' field of the second fifo_store can be accessed via 'fifo_new.next_ptr.prev_ptr'. Line 4 assigns this field the access value of the original fifo_store, which is held in the variable fifo_new. The objects addressed by each field on progressing through the code are indicated in figure 3.. Once this has been completed, the access value held in variable fifo_new can be updated to indicate the location of the next fifo_store (i.e. the second fifo_store to be created in this sequence). This is done on line 5. Thus each time the code is executed, a new empty fifo_store is created, and the access values held by the existing fifo_store objects are updated to indicate the previous and the next fifo_store objects in the chain. This creates a linked list. Manipulation of a second variable, fifo_old, allows the objects in the list to be accessed in the order in which they were created, and subsequently return the allotted memory to the system.

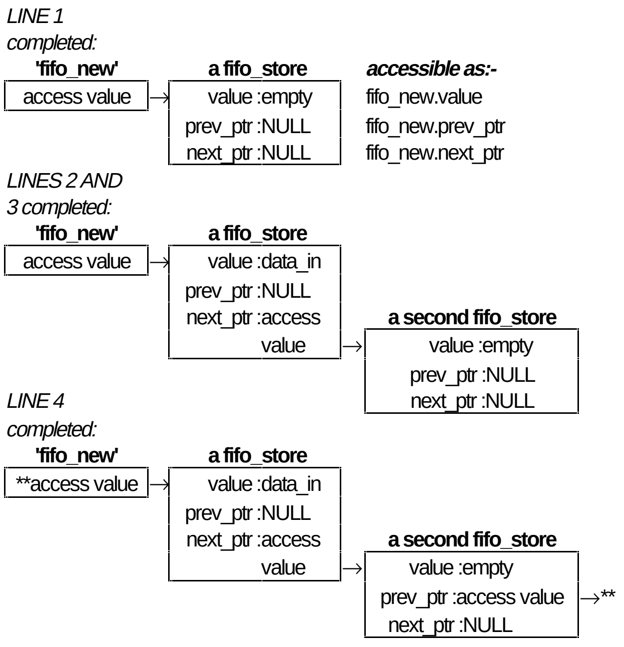

Figure 4. Access values

Simulation map

A graphical user interface is being developed for our simulator. This will allow the user to select components from a menu and place them on the display screen. Components will be 'wired' together graphically (see figure 5.), and any necessary values prompted for. Defaults will be available for use if the user fails to supply a value. The menus will allow the user to select the protocol to be used in the simulation. At the time of writing, (January 1994), only the CAN protocol has been modelled, but this will be extended to include CAN2, J1850, and any others which are required. Values which can be set by the user, for example the identifiers of sensors and actuators, will be available via a menu. Once the user has completed the simulation map and parameter setting, this will be indicated by selection of the appropriate menu items and the simulation can be prepared for execution.

Preparing for simulation

The preparation for a simulation involves the generation of a configuration file and a test harness. The configuration file contains the detail about which entities have been selected for use in the simulation, e.g. which protocol chips are to be simulated. The test harness contains the information about specific values which are to be applied in the simulation, e.g. the process time for a chip. Additionally the test-harness will contain the information detailing the interconnection of the components for the particular simulation. These files will be automatically generated from the graphical map which is built up by the user, and from the property selections made for the components.

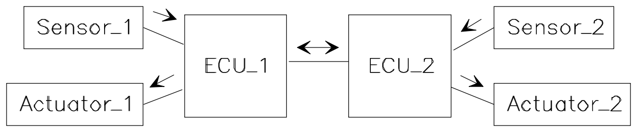

Figure 5. A simple 'map' of databus interconnections

Test harness

The overall simulation is described by a 'test harness', which typically looks as follows:

```
LIBRARY ..
     :
ENTITY simulation IS
END simulation;
---------------------------------------------------------------
ARCHITECTURE behav OF simulation IS
 CONSTANT bit_rate                 : TIME   := 1000 ns ;
 CONSTANT monitor_sample_period    : TIME   := 100 us ;
 CONSTANT ecu1_sensor_no           : INTEGER := 1 ; --no. of
                                      --sensors on ecu1
 CONSTANT ecu1_actuator_no         : INTEGER := 1 ; --no. of
         :                            --actuators on ecu1
---------------------------------------------------------------
 SIGNAL can_h : can_logic ;
 SIGNAL can_l : can_logic ;
 SIGNAL sensor_array1 : sensor_array_type(ecu1_sensor_no
                                         DOWNTO 1);
 SIGNAL actuator_array1 : actuator_array_type(ecu1_actuator_no
                                         DOWNTO 1);
 SIGNAL rx_latency1 : latency_array_type(ecu1_sensor_no
         :                               DOWNTO 1);
         :
 COMPONENT ecu
   GENERIC (bit_period        : TIME ;
            num_of_sensors    : Positive ;
            num_of_actuators  : Positive ) ;
   PORT (   sensor_array      :        sensor_array_type ;
                 :                                      ) ;
 END COMPONENT ;

 COMPONENT sensor
   GENERIC (  sensor_id       : Identifier_type ;
              data_length     : data_length_code ;
              sample_period   :TIME ;
              freq            : REAL );  -- in Hertz
   PORT (     sensor_out      : OUT    sensor_frame  ) ;
 END COMPONENT ;

 COMPONENT actuator
   GENERIC (  actuator_id       :Identifier_type ;
              repeat_period     :TIME ) ;
   PORT (     actuator_in       :         actuator_frame ;
         :                                               ) ;
 END COMPONENT ;

 COMPONENT logger
   GENERIC (  sensor_id         : identifier_type ) ;
   PORT (     latency_data      : latency_frame ) ;
 END COMPONENT ;

 COMPONENT bus_monitor
   GENERIC (  bit_period        : TIME ;
              sample_period     : TIME ) ;
   PORT (     can_h             : INOUT   can_logic ;
              can_l             : INOUT   can_logic ) ;
 END COMPONENT ;

 FOR ALL : logger     USE
  ENTITY work.logger(a1);
         :
 FOR ALL : actuator USE
  ENTITY work.actuator(a1);
---------------------------------------------------------------
BEGIN
 ecu1     : ecu
   GENERIC MAP (bit_rate , ecu1_sensor_no , ecu1_actuator_no )
   PORT    MAP ( sensor_array1,actuator_comm_array1,
                 can_h,can_l,actuator_array1,rx_latency1 ) ;
 sensor1_1: sensor
   GENERIC MAP ( 500 , 1 , 1100 us , 1.0 )
   PORT    MAP ( sensor_array1(1) ) ;
         :
 logger1_1: logger
   GENERIC MAP ( 500 )
   PORT    MAP ( rx_latency1(1) ) ;
 ecu2     : ecu
         :
 actuator2_1: actuator
   GENERIC MAP ( 500 , 100 sec )
   PORT    MAP ( actuator_array2(1),actuator_comm_array2(1) ) ;
---------------------------------------------------------------
 monitor1 : bus_monitor
   GENERIC MAP ( bit_rate,monitor_sample_period )
   PORT    MAP ( can_h,can_l ) ;
---------------------------------------------------------------
END behav ;
```

The library used is the one which is associated with the protocol selected by the user. Each library contains a 'definitions' package which incorporates definitions specific to that protocol. The name assigned to the simulation by the user is taken as the architecture name.

The connections between instances on the simulation map are represented by signals of the same name in the architecture. The connections, and hence the signals, will be assigned default names when they are created and will have a type appropriate to the port types which they connect. The user can specify more meaningful names if required. Any unconnected ports are defined as 'open'.

Logging simulation results

The primary reasons for simulating a databus are to discover the message latencies; i.e. how long it takes to get from source to destination and how this varies with bus loading. An entity known as a 'logger' has been developed solely to maintain an ASCII log file of latency data. There is a logger for each sensor requiring latency data to be recorded. The latency of a message is calculated by the ECU from the difference between the time stamps on the original data passing to the application layer, and the subsequently returned data confirmation. An array of latency data, with an entry for each sensor on the ECU, is maintained. Each logger monitors one array entry corresponding to a particular sensor identifier, and maintains a corresponding text file of latency related data. This information may be post-processed in whatever way the user chooses.

A 'bus-monitor'- entity has also been developed. This acts as a window onto the modelled data highway, sampling the activity on the bus at regular intervals. It maintains a count of the total number of samples which have been made during a simulation, and also a count of the number of times the bus was found to be busy. The bus is considered to be 'busy' if the MAC layer is receiving a frame or an interframe space. A calculation of bus loading averaged over time is made by dividing the number of busy samples by the total number of samples taken. Only one bus-monitor is required for a simulation.

Plotting the maximum latency of the lowest priority node in a typical network against the bus loading might yield a result similar to that shown in figure 6.

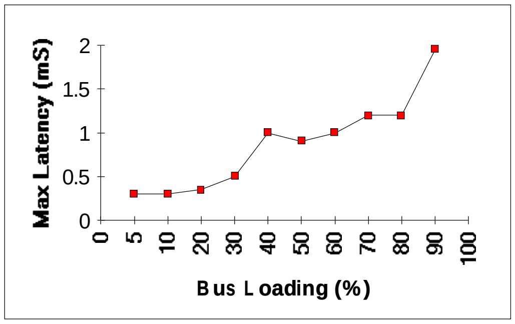

Figure 6. Typical results from a simulation

Once the simulation files have been prepared, the actual simulation can be performed on any vendors' simulator. VHDL simulators can be operated by the user entering instructions about the length of time a simulation should run, the model to be simulated, and the parameters to be logged, etc. An alternative to this is 'batch mode'. This allows the user to enter a single command line and then leave the simulator to follow the other instructions contained within the batch file. The databus simulator developed at Lucas will automatically generate the required batch file for the Model Technology simulator. Information about run length, parameters to be monitored etc. is again obtained from the user via menus. The necessary starting command line is also generated automatically from the menu system, so that the user of the simulator selects a menu item and the simulation begins automatically.

At the time of writing, the underlying simulator has been written. It has successfully reproduced the results of a large simulation performed on our previous databus simulator (which was verified against real data), with a simulation speed improvement in the order of 100 times. The graphical interface, necessary to allow non-VHDL trained personnel to use the tool, has not yet been completed.

Summary

VHDL is an ideal language for modelling databus networks. Whilst the language is primarily intended for hardware descriptions, the inclusion of 'software-like' features in the language has greatly aided the development of a databus simulator. The ability to model at different levels of abstraction, and to easily mix these levels through the use of configurations, has allowed the creation of a very useful and flexible tool. It allows performance prediction of different system architectures to be carried out quickly. Potential problem areas can be highlighted at an early stage in a project and detailed investigations into these areas can be carried out using the simulator.

Additionally, since VHDL is a hardware description language, descriptions for devices which connect to a databus can also be generated, and it is hoped that in the future manufacturers will supply models of their devices in VHDL. Meanwhile it is possible to create our own library of device descriptions.

The simulator is flexible, not tied to a specific vendor, and allows us scope with future developments. These will include the addition of more protocols, refinement of sensor and actuator models, and the enhancement of the graphical user interface. The addition of data analysis facilities is likely to be achieved through the development of interfaces to existing analysis tools.

References

[1] ISO 11519-2 Road Vehicles- Low Speed Serial Data communication- Controller Area Network (CAN)

[2] ISO 11898 Road Vehicles- Interchange of Digital Information- Controller Area Network for High Speed Communication

[3] ISO 11519-4 Road Vehicles- Low Speed Serial Data Communication :Part3 Class B Data Communications Network Interface (J1850)

[4] 'A general Purpose Simulator Applied to Automotive System Design', N.J. Carter, T. Lewis & S.Turner, Proceedings of the First European Conference on the Practical Applications of Lisp, Europal '90, March 1990.

[5] ISO 7498 Information Processing Systems- Open Systems Interconnection- Basic Reference Model