Additions to State Builder for Network Modelling

Jon Bell[1]

Doc. ref. SD/REQ/01
Version 1.0
July 9, 2002

[1]email jpb@aber.ac.uk

1 Introduction

Investigations carried out in the course of the SoftFMEA project have led to the idea that the existing AutoSteve *State Builder* tool can be used as the basis for modelling electrical systems that incorporate a network in such a way that the effects of electrical faults can be found. Therefore this approach can be considered as fulfilling the requirements for SoftFMEA Workpackage 3. This document sets out requirements for the *State Builder* based extensions to AutoSteve necessary for simple modelling and simulation of systems that incorporate network components.

The requirements specified here are limited to those deemed necessary to allow modelling of electrical failures, they will not be sufficient to allow any more than simple modelling of network failures. Modelling of these failures requires changes elsewhere, such as to the functional language, to allow late fulfilment of a function to be modelled. There are also other problems arising from modelling of closed loop systems that are not addressed here. These will have to be confronted in the later implementation. Examples of open and closed loop circuits are discussed in [4] and some of the problems with simulating closed loop circuits are introduced in [1].

There are two aspects to the extensions necessary to AutoSteve. These are extensions to the existing state chart language itself and any additional GUI components that might be wanted to allow details of the network to be specified. The document has been divided into two parts, mirroring these two aspects of the implementation.

A simple example has been prepared to illustrate some of these requirements. It has not been included in this report, but a copy can be made available.

1.1 Objectives and scope

This document is intended to act as the requirements specification for additions to the existing AutoSteve tool that are necessary to allow simple modelling of systems that make use of a network. It includes both additions to the state chart language and to the tool's user interface to enable the network part of a system to be specified.

1.2 Limitations

This document does not include any discussion of the need for the specified changes nor of the rationale behind them. These are discussed in earlier reports, see [3, 5, 2]. It will not specify how these changes are to be incorporated into AutoSteve. However notes appended to the document discuss possible approaches to the incorporation of the changes. These should not be taken as constituting a design specification.

2 Overview of the required changes

As network components can readily be modelled at a behavioural rather than a structural level, the aim of the proposed changes is to allow individual components' behavioural models to send and receive messages. These messages will naturally model network messages. It is assumed that components' behaviour will be modelled using the AutoSteve state machine language, *State Builder*. Clearly it is necessary that a physical medium for the transmission of these messages be specified. Investigations suggest that all protocols used in the automotive sector use broadcast messages identified by source (or at least a message identifier associated with the source), not by intended recipient, so this should be correctly modelled. The requirements this gives rise to in the state machine language are specified in the next section, Section 3.

Although the most important aspect of the changes to be made to AutoSteve are to the state machine language itself, there might be some value in adding GUI components to enable the user to specify all aspects of the network at once. It should be possible to specify all additional information, such as the identifiers of messages sent by the component, in the component model. This is illustrated in the worked example that accompanies this report. Therefore a minimal implementation should need no additional user interface components. However, it might well be valuable to allow the user to specify all details of the network at once, having a (possibly temporary) data store to hold all message identifiers and possibly correct message values used by a system. This would allow consistency checking across the network. These requirements are discussed in Section 4.

3 Additions to the state chart language

In this section the required additions to the state chart language (*State Builder*) are specified. In many cases there will be a reference to a note, which will suggest a possible approach to fulfilling the requirement. These suggestions are out of place in a requirements specification, but have been included in an appendix to encourage discussion about possible approaches.

3.1 transmitting messages

As several components' state charts will communicate with each other, we need a method allowing a component to transmit a message. See note A.1. This method should be called on a node (such as a pin), as a component such as a CAN terminal will be connected to more than one data transmission route — the CAN network itself, and the connection to its associated ECU. The transmit method will need to transmit a "message" data type, as specified in requirement 3.2.

3.2 Message data type

A message needs to be identifiable, so needs two attributes - identifier and content. Both of these fields can be integers. This is necessary as messages are typically received from several sources, so the meaning of the content depends on the message identifier. See note A.2.

3.3 Receiving messages

On receiving a message, its data (identifier and content) must be made available to the receiving component. We therefore need a method that returns a "message" object. See note A.3.

3.4 Signal routes

The routes over which messages should be passed over are to be specified in the schematic. As broadcast messages are to be modelled, any node on the specified network should receive all messages transmitted by any node on the network. See note A.4.

3.5 Transmitters

Any node on a network should be able to transmit messages. All nodes on that network should be able to receive any transmitted message, so as to model broadcast messages. See note A.5.

3.6 Multiple receivers

As messages are broadcast, several receivers might react to a message concurrently. See note A.6.

3.7 Received called on node

As a CAN terminal might receive messages both from the network and from its associated ECU or a bridge from either of the two networks it joins, and these cases might fire distinct transitions, some way of distinguishing between the source of the messages is needed. Using the node (e.g. pin) that receives the message is sufficient.

3.8 Saving data

The additional commands and data (such as lists of interesting message identifiers) must be capable of being saved along with the other details of the component. If a central data source is used to allow consistency checking of the network components, this also needs saving. This is not needed in a minimal implementation.

3.9 Consistency checking

On saving the existing consistency checks will need extending to warn the user if the following errors are found...

- A receiver expects messages with an identifier that no component sends.
- A receiver expects messages with content values that are never sent in messages with the specified identifier.
- A transmitter sends messages which no component acts on.

These should be no more than warnings as the apparent errors could possibly be deliberate. See note A.7.

4 Additional user interface components

This section describes the additional user interface facilities and components that might be wanted to allow the user to specify the necessary details of the network and the components that use it. Examples of such additional data are the identifiers of network messages a component might send and ids of messages a component will react to. In “full CAN” a terminal will only forward to its associated ECU those message that the ECU is interested in, so their identifiers need to be known.

As suggested in Section 2, a minimal implementation should have no need of any additional user interface components. However users might welcome a facility to allow the network to be specified. As specifying the network as one item requires some sort of central data store, this enables consistency checking across the network.

4.1 Network specification frame

If users are to be allowed to specify all details of the network side of a system at once, then some sort of frame allowing the components connected to the network to be listed, along with the message identifiers of any messages they send and the messages to which the react, will be required.

4.2 Data store

If a facility to allow the network to be configured (requirement 4.1) is to be provided, then a central data store to hold the configuration will be required. It will be necessary for this information to be saved, in case work is interrupted during modelling of the network. This central data store will also support consistency checking (requirement 3.9).

A Notes on additions to state chart language

This appendix adds notes on the required additions to the state chart language, suggesting how they might be implemented. It is appreciated that these notes do not really belong in a requirements specification so they have been separated into this appendix. It should be stressed that they are intended as suggestions and so do not constitute requirements in themselves.

The existing `transmit()` and `received()` methods appear to be a useful starting point for these changes, and relationships between these and the proposed additions are discussed here.

These additions to the language will, of course, mean working on the C++ code in AutoSteve.

A.1 Note on message transmission

The existing transmit facility looks a suitable starting point. The only obvious change is the need for it to send a Message data object (see 3.2) rather than an integer.

A.2 Note on message modelling

The existing transmit/received facility does not have a data type for messages. Transmit requires and integer, and received returns one. We cannot simply use an integer as this can represent the message identifier or the contents, but not both. Clearly the meaning of a message's contents might differ according to identifier, so both need modelling. In the worked example, I have added a Message data type, as specified in this requirement (3.2), and have an overloaded transmit method that takes a Message and a gotMessage method that returns an object of the same type.

If we wish to model full CAN with the CAN terminals as separate components we might well need two distinct message types, with and without the identifier field. This is to allow the ECU to send a simple message to its terminal, and the terminal will add the identifier. The existing transmit method will not work here, as the integer is not wanted by the CAN terminal component, which simply adds the identifier and forwards the contents as a CAN message.

A.3 Note on receiving messages

This is a close counterpart to the existing received method. The only necessary change is that it must return an object of type Message, see 3.2. A possible alternative is to have a method that returns true when a message is received and maybe fires a transition into a "processing" state. This would simplify modelling of CAN terminals that are not interested in the contents of messages but leaves the problem of making the message available to the receiving component.

A.4 Notes on routes for messages

This is clearly closely similar to the current transmit/received facility. Dave Ellis suggested using a net to distinguish between the electrical and network sides of the system.

A.5 Note on transmitters

The existing transmit/received facility appears to specify that any one node can transmit to the "communications network". Together with the need to give messages identifiers (see 3.2) this is the most important shortcoming with the existing facility.

A.6 Note on multiple receivers

I believe this is already supported by transmit/receive, so should be no problem. It is certainly possible to model broadcast messages using the event passing facility in State Builder.

A.7 Note on consistency checking

This requires checking for consistency between state machines, of course, which is not easy. Clearly there will need to be some sort of association between the components at system level to support this. This is perhaps more a want than a real need - the user can do any such checking manually. Some sort of GUI with a central data store for the network components (requirements 4.1 and 4.2) would allow this checking, of course. Indeed it might be necessary.

B Notes on additional interface components

No design for any such window has been prepared, I imagine simply listing all transmitters together with their associated message identifiers, so two editable text areas should suffice. It will, of course, be necessary to parse the entered string to get a list of all the items. This needs more work, but as it is not necessary in a minimal implementation, this can wait.

References

[1] Jon Bell. The heater circuit - matters arising. SoftFMEA internal report, 2002.

[2] Jon Bell. Modelling behaviour. SoftFMEA document ref. SD/TR/MM/01, 2002.

[3] Jon Bell. Proposed approaches to network simulation. SoftFMEA document ref. SD/TR/03, 2002.

[4] Jon Bell. Systems with telematic components. SoftFMEA document ref. SD/TR/02, 2002.

[5] Jon Bell. Using a state machine language for behavioural modelling. SoftFMEA Document ref. SD/TR/FSM/01, 2002.