Debugging Software Designs with ASD

Document ID : DocID
Author : Guy H. Broadfoot
Version : 2.0 (Final), 14/10/2004

Paradijslaan28-28a, 5611 KN Eindhoven, The Netherlands
Phone: +31 40 235 9090, Fax: +31 40 235 9099, E-mail: info@verum.com
Commercial Register Eindhoven no.: 17106874l

SUMMARY

This document describes an application of Analytical Software Design to an example Cruise Control design.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PURPOSE

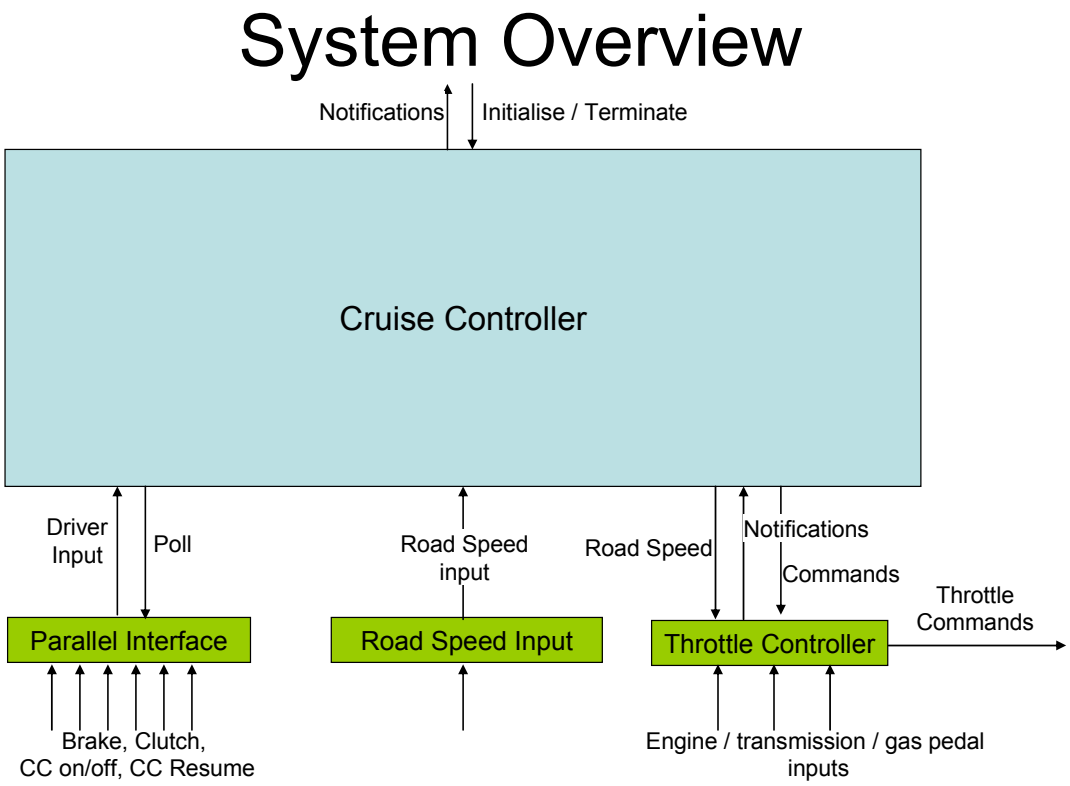
This document describes how Verum applies Analytical Software Design techniques to verify software designs before implementation.

2 CRUISE CONTROLLER DESIGN NOTES

2.1 SYSTEM CONTEXT

The Cruise Controller interfaces with a parallel interface from which driver input in the form of pedal switch and button switch signals are received. On the diagram, this is labelled as PI and all events from that interface are prefixed with ***pi***. This interface is polled by the software every 100 milliseconds. It provides data input and error status only in response to a poll command.

The Cruise Controller interfaces with a hardware interface from which information is received about the vehicle's road speed. This is a "push" interface and data arrives without being polled. We have not defined the form of this input or its interface; only that it is sufficient to enable us to compute the vehicle's road speed in cm/sec and provide this once every 100 milliseconds to the **TCI** interface.



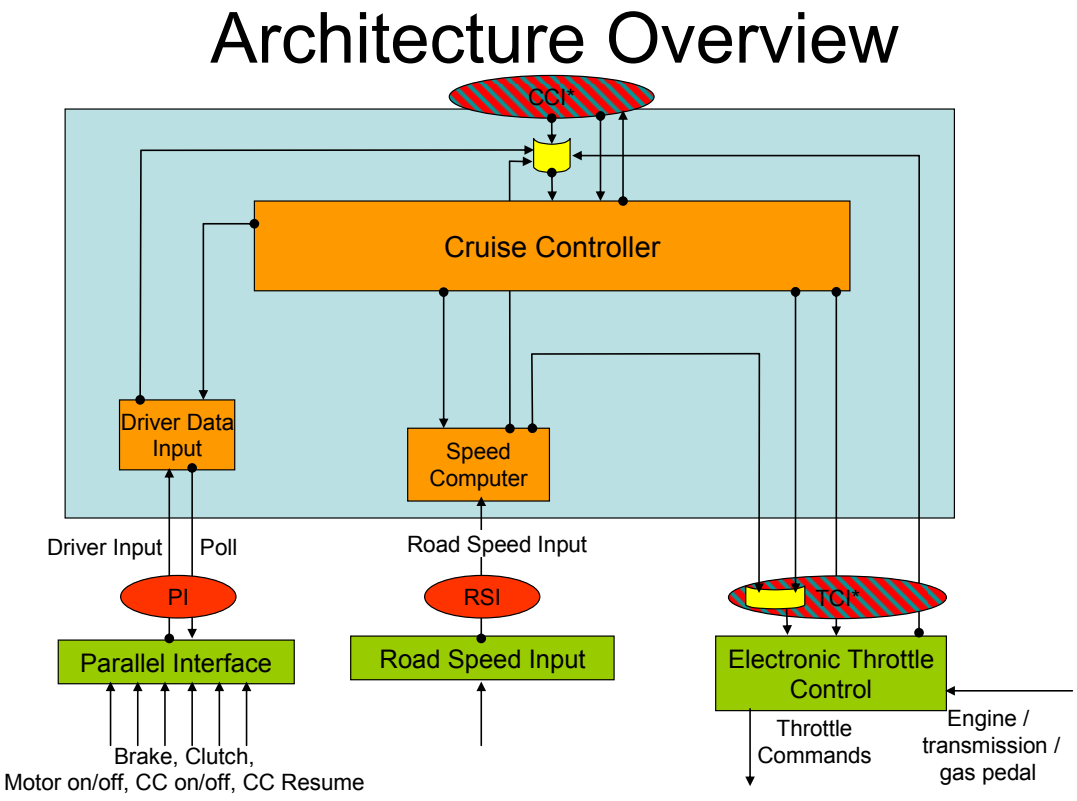
The Cruise Controller interfaces with an Electronic Throttle Control Unit via the **TCI** interface. This is a bought-in component whose behaviour is a given. If changes are required for some reason, this must be accomplished either by negotiating an engineering change with the supplier, making a hardware abstraction layer of software, or both. As seen from the specification document, many of the system's safety properties are actually implemented by this component.

The Cruise Controller interfaces with the rest of the Vehicle Management System via the **CCI** interface. It receives commands for initialisation via this interface and reports back error conditions.

2.2 LOGICAL ARCHITECTURE

The Cruise Controller Software has been divided into three processes (threads) or active components. These are: **CC**, the overall control process, **DDI**, the Driver Data Input Process and the **SCI**, the Speed Controller Process.

The **CC** process accepts commands from the Vehicle Management Interface (the Client) to initiate, terminate and activate. In addition, it receives commands and status notifications from the **DDI**, **SCI** and **TCI** components. The initiation and termination commands from the Client are executed via synchronous interfaces. All other commands and notifications are received via a FIFO input queue.



The separate activation command is provided to allow the Client flexibility in initialising all its various subsystems on the vehicle.

The **CC** accepts all driver input from the **DDI** process, determines which commands are allowed in which state and instructs other components to perform the necessary actions.

The **DDI** process polls the parallel interface and maps the input signals to logical driver input events. These are routed to the **CC** via its input queue for processing. The events corresponding to the driver pressing the brake and/or clutch pedal are combined into a single pair of “Disable” and “Enable” stimuli. It is not necessary to distinguish between the actual events because the **CC**’s response is the same in both cases.

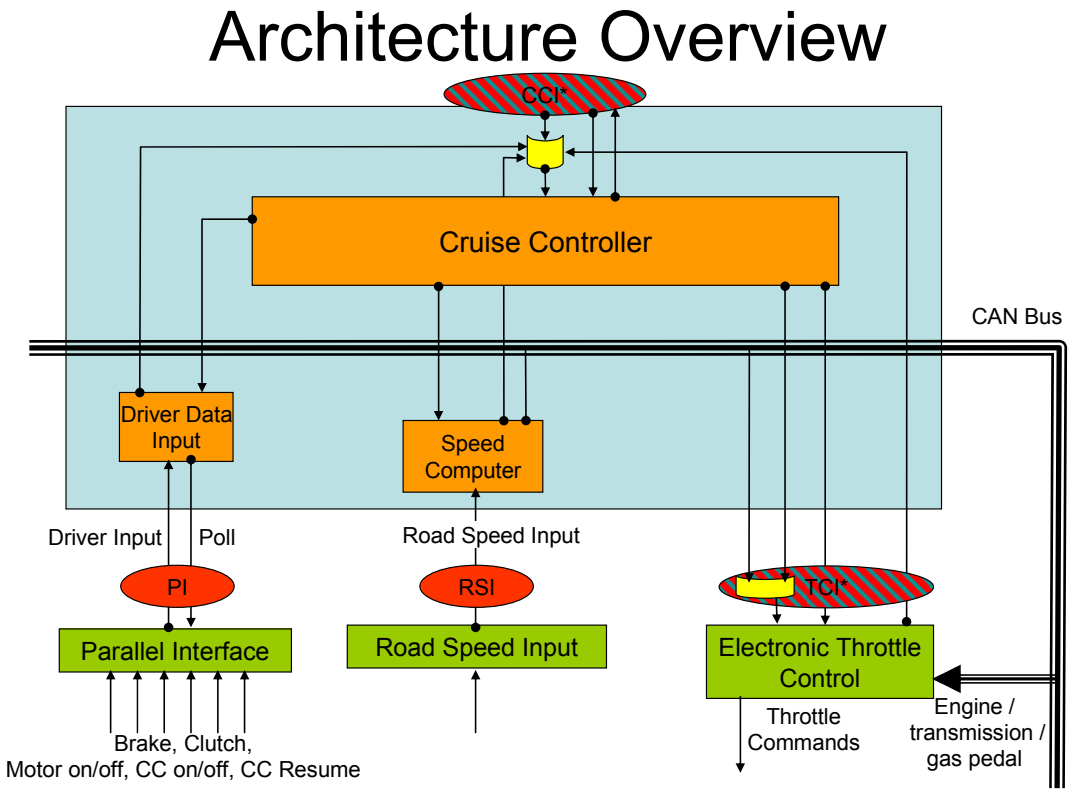
The **SCI** component receives the road speed input via interrupts, computes the actual road speed in cm/sec and passes this to the **TCI** at a frequency of once per 100 milliseconds.

The **TCI** component accepts commands to engage, resume and disengage from the **CC** component and speed inputs via its input queue from the **SCI** component. It also receives various inputs from the vehicle interface concerning transmission status, motor status and gas pedal

position. When engaged, the **TCI** compares the actual road speed with the current set point and issues the necessary throttle control commands.

2.3 PHYSICAL ARCHITECTURE

The components are connected to each other and to the Vehicle Management System via a 1.0 Mbit/s CAN Bus. The **CC**, **DDI**, **SCI** and **TCI** components are all packaged as individual ECU's and appear as nodes on the CAN bus network.



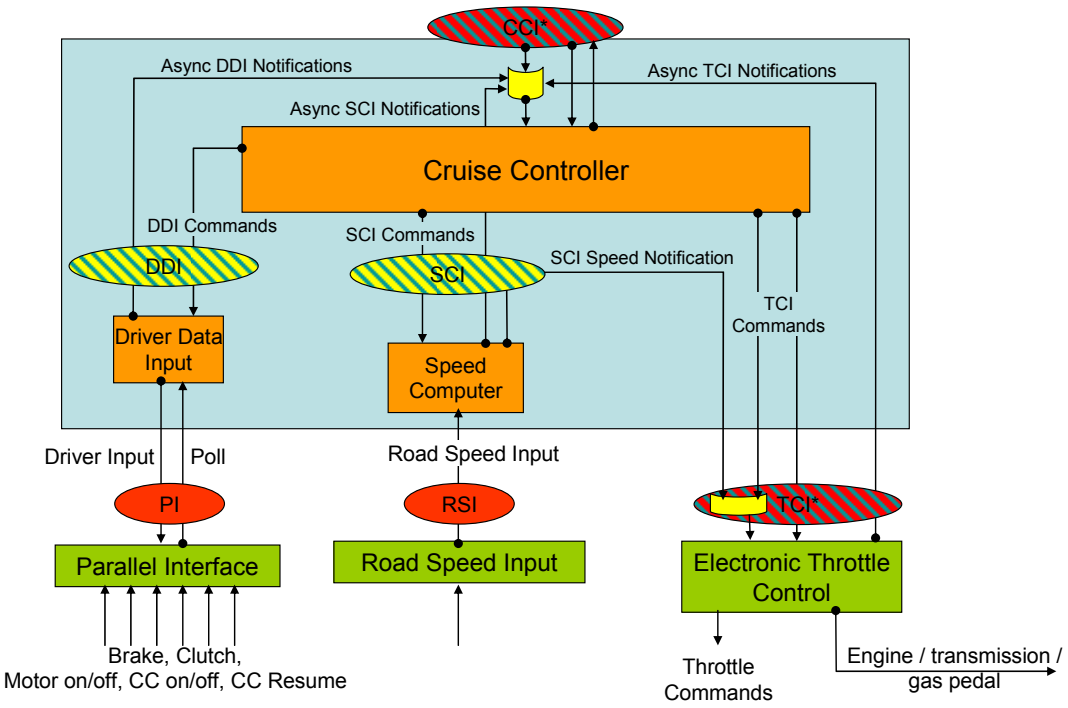
The **CC** component's only connections with the rest of the vehicle (apart from power supply) is via the CAN Bus.

The **DDI** has direct wired connections to its various switches and sensors via the parallel interface. The decoded inputs are sent to the **CC** component via the CAN bus. Similarly, the **SCI** component is directly wired to its source of input and it outputs its speed messages to the **TCI** via the CAN bus.

The **TCI** is connected directly to the throttle control hardware to control the vehicle speed. In addition, it receives message via the CAN bus from the engine management system (torque, RPM, etc.) and from the transmission (selected gear). With this information, plus the speed input, the **TCI** computes required throttle setting and controls the vehicle speed.

2.4 SPECIFYING DESIGNS AND INTERFACES

The behaviour of the **DDI**, **SCI** and **TCI** processes are specified as functional specifications describing their externally visible behaviour; we do not have the design of the **TCI** and we have not yet made the designs of the **DDI** and **SCI**. In order to correctly represent the **DDI** and **TCI** behaviour, abstract events are defined to represent the important input events from the vehicle interfaces.



In specifying **DDI**'s behaviour, for example, we do not need to distinguish between the various combinations of input signals in order to model its behaviour as seen by the **CC**. It is sufficient simply to recognise that input data arrives into the **DDI** and that it responds in a nondeterministic way by choosing one of the possible responses. Until we make the design of the **DDI**, we do not need to determine how this choice is made.

In a similar manner, the **TCI** component is modelled responding in a nondeterministic way to speed input, choosing whether or not to respond as though the speed is out of range and triggering an automatic disengagement action. Because this is a component purchased from an outside supplier, we do not have design information available to us. However, as long as we accurately model the possible range of its externally visible behaviours, we do not need to know how the design resolves choices that appear to us to be nondeterministic.

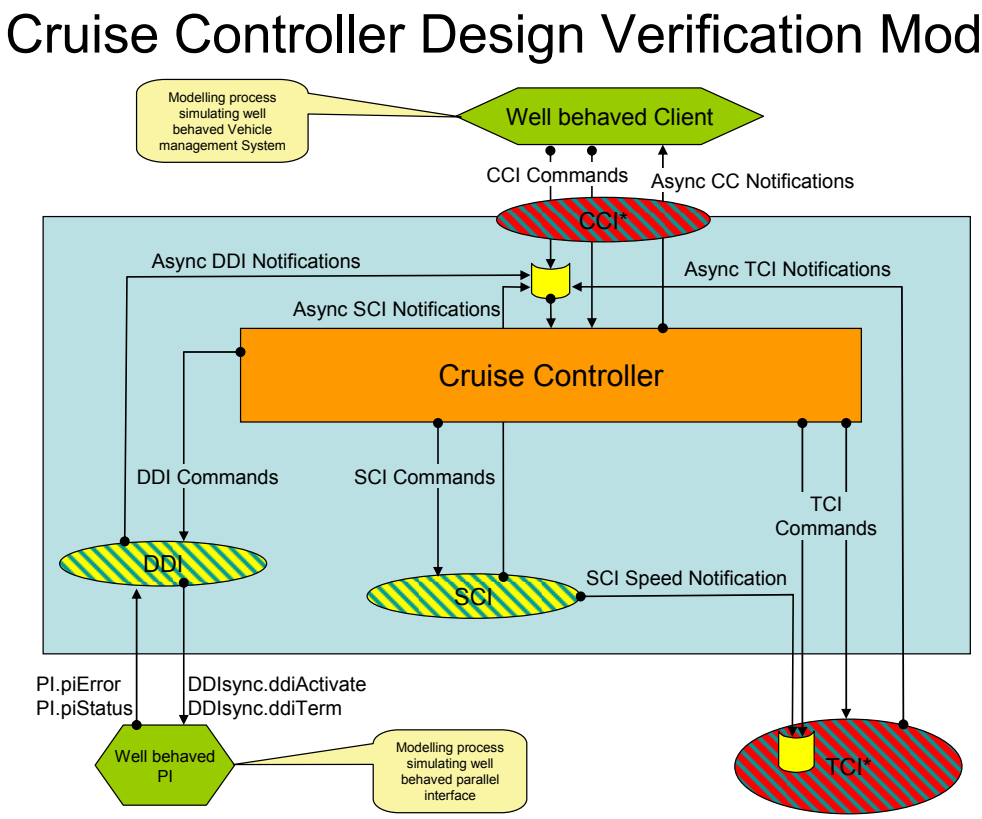
We specify both the design of the **CC** component and the interfaces (externally visible behaviour) as black boxes using the Sequence-based Specification method. The design is specified as a *fully specified*, deterministic Black Box Function; the interfaces of the other components specifying the externally visible behaviour are specified as *underspecified* black box functions to enable the nondeterministic behaviour to be captured. All the black box functions can be plotted in the form of state transition diagrams.

2.5 MODELLING FOR VERIFICATION

To verify the design of the **CC** component using model checking, we need CSP models of the **CC** component design and the interfaces of all the other components. These models are generated fully automatically from the black box specifications without needing any special CSP or modelling knowledge. But these alone are not quite enough.

It is part of the ASD design approach that all sequences of input stimuli are explicitly captured and specified, including those that represent illegal behaviour by one or more components. This illegal behaviour is preserved in the CSP models so that the correctness verification can check for it. However, the Model Checker assumes that all events offered by the models in each state are valid

and it will explore them all; this includes all illegal behaviour as well as all legal behaviour. In other words, the Model Checker behaves as a badly behaved Client, trying everything.



This means that in many cases, we cannot meaningfully check the design together with the interfaces it uses without providing some extra modelling to specify that our Clients will only behave correctly. In this case, we provide two small models that are not part of the design but exist only for the purposes of the verification; one of these behaves like a well-behaved Vehicle Management System and the other behaves like a well-behaved parallel interface to the **DDI** component. Between them, these two extra models constrain the inputs and outputs to the Cruise Control System to those that are legal and expected.

We could, of course, leave this illegal behaviour out of the black box interface specifications when it causes difficulties. We do not do this because when we design these other components we want to use the same interface specifications to check these other designs.

2.6 SPECIFYING TIMING ASSUMPTIONS

CSP is an untimed process algebra; designs are checked on the assumption that any event can occur infinitely quickly or infinitely slowly. The advantage of this is that correctness verification does not depend on the accidental ordering of events between parallel processes; all interleaving of events as allowed by the design are examined.

Many designs work in practice because there are known timing assumptions that are guaranteed by the environment. For example, the **DDI** component generates input events to the **CC** at a far lower rate than the **CC** can process them. This is because the **DDI** events represent interactions via switches with the human driver. The rates at which these can occur are limited by the physical characteristics of the switches and the physical ability of a human to operate them. Knowing this, the maxim arrival rate of such events at the **CC** can be decided and the maximum number of these events in the **CC** input can be computed. Similar analysis on the events between the **CC** and the other components leads to computing the queue length needed to ensure that all operations placing events in the **CC** queue are non-blocking.

In the Cruise Control design as it stands, a deadlock check will fail. This is because these timing assumptions are not built in to the model and the model checker assumes that the **DDI** can generate driver input at an infinitely fast rate until filling the queue and being blocked by it. To determine if the design is correct when subject to these timing assumptions, these assumptions have to be added to the model. This is done by a technique called “Yoking”.

A “Yoke” is a special constraining process in CSP that limits the behaviour of a producer process to the behaviour of the consumer process in such a way that the model reflects the timing assumptions but has no new behaviour added and no existing behaviour removed. The ASD modelling environment contains a generic template Yoke process that can be parameterised for a given design.

It is good practice to verify designs as far as possible without specifying timing assumptions. There are two principle reasons for this:

1. Designs with minimal timing assumptions are more robust and less easily broken by future modifications or hardware changes. By verifying designs without timing constraints and then adding only those essential to ensure correct behaviour, we usually end up with far fewer timing constraints than would otherwise be the case and these are clearly identified and documented for the future.

2. Adding timing constraints before verifying designs can lead to false assumptions being added to the models which in turn, prevent possible design errors from being detected.

When a design cannot be verified without specifying timing constraints, always attempt to modify the design where possible to avoid the need to add them. Timing assumptions should be added only in the last resort.

3 MODEL CHECKING

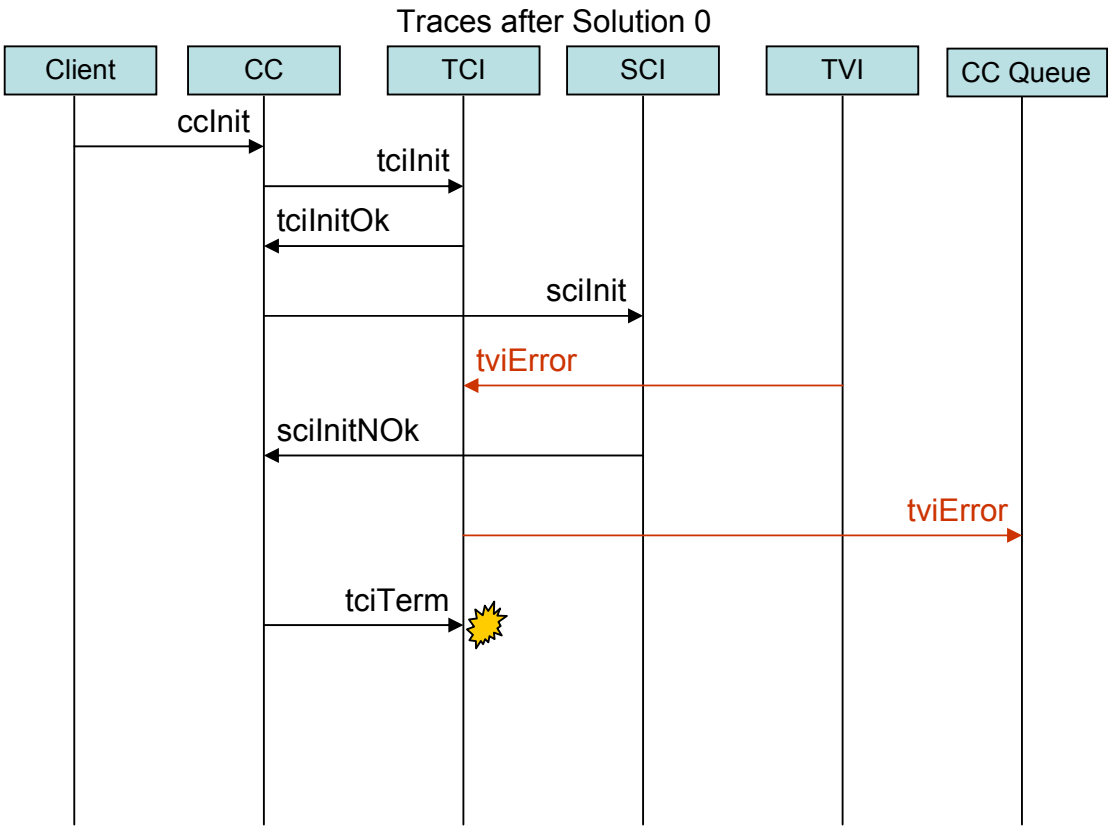
3.1 GENERAL

During the demonstration, we examined the mathematical models of the Cruise Controller design to ensure that no events are sent between components in such a way that this would violate the intended design behaviour. This is a general "sanity" check performed for every design; it is analogous to building architect instructing a structural engineer to perform a finite element analysis of a building's structural design in order to ensure that it will not collapse. In our case, we are examining the design together with the interface specifications of the other, used components in order to ensure that our design behaves according to the interface specifications of those other components.

3.2 SOLUTION 0

Check each component (or component interface) to see if the combined system forces any of them into an illegal response.

In this case, **TCI** generates an illegal response as shown by the following traces:



CC's client (the vehicle) commands it to initialise. The **CC** commands the **TCI** to initialise itself via the synchronous ***tciInit*** interface. The **TCI** initialises correctly and the **CC** commands the **SCI** to initialise via its synchronous ***sciInit*** interface. While **SCI** is initialising, the vehicle interface TVI detects and reports an error to the TCI. The TCI detects this, responds by posting a ***tciError*** event to the CC's input queue and terminates itself by entering the Un-initialised State. The SCI responds synchronously that it has failed to initialise.

The CC responds to the SCI initialisation failure (which it sees before it sees the asynchronous ***tciError*** notification because initialisation is performed synchronously) by issuing a ***tciTerm*** command to the TCI via its synchronous interface. This is illegal when TCI is in the Un-initialised State.

This cannot be solved with the interface behaviour of the TCI as currently specified. The problem is that the TCI component's Ready State is unstable and can transit to a state, namely the Un-initialised State, that defines as illegal some of the events allowed in Ready State.

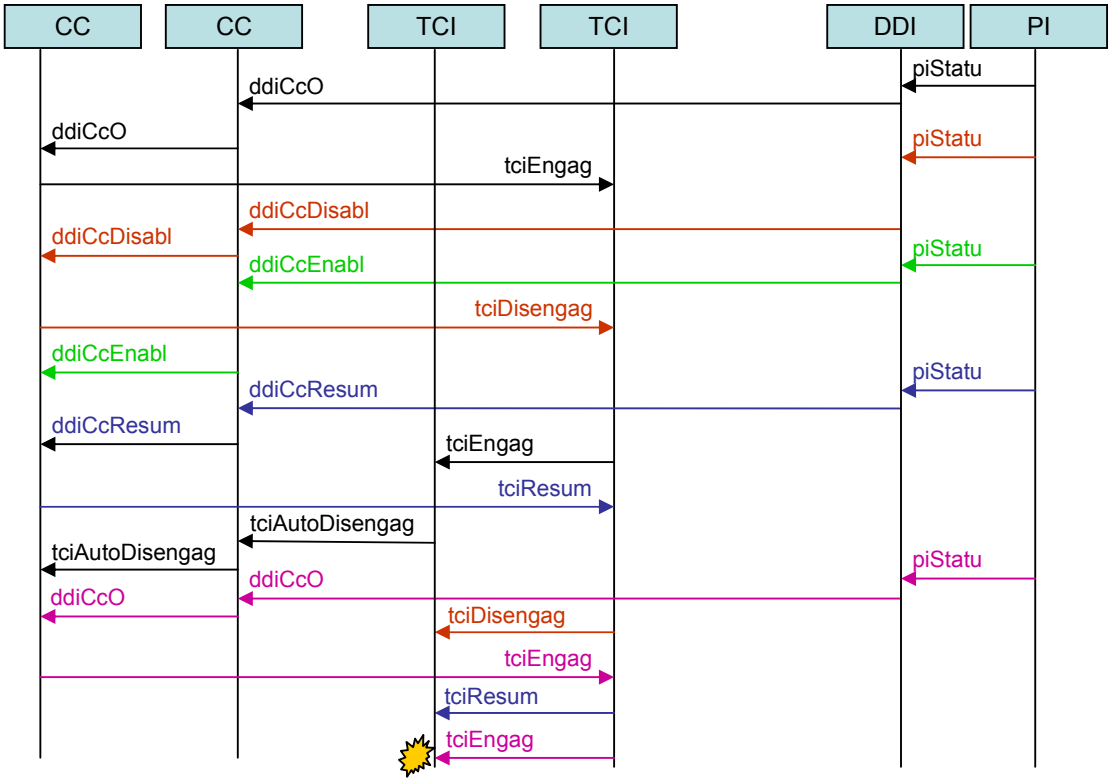
The proposed change is to introduce a stable Error State into the TCI interface behaviour. In this state, everything that is legal in Ready State is also legal in the new Error State, although it acts differently by ignoring everything except the ***tciTerm*** command.

All errors in TCI result in a transition to Error State followed by an event ***tciError*** being sent asynchronously to the CC. TCI remains in this state until it receives a ***tciTerm*** command from the CC, upon which it transits to Un-initialised State.

3.3 SOLUTION 3

Rerun the check on **TCI** to see if the proposed design changes solve our problem. Again, an error is encountered with the following trace (the trace shown starts after the system has completed initialisation):

Traces after Solution 3 – after initialisation and activation



This is a complicated case and at first sight it is not clear what is happening. Due to the length of the combined trace, it is helpful to "drill down" into the **TCI** component and look at its trace separately:

TCIsync.tciInit,
TCIsync.tciInitOK,
TCIinQueue.tciEngage,
TCIinQueue.tciDisengage,
TCIoutQueue.tciEngage,
TCIinQueue.tciResume,
TCI2CC.tciAutoDisengage,
TCIoutQueue.tciDisengage,
TCIinQueue.tciEngage,
TCIoutQueue.tciResume,
TCIoutQueue.tciEngage,
Illegal.TCI_Tag.

Now it is clear that **CC** instructed the **TCI** to engage itself via a ***TCIResume*** event and while the **TCI** was already in the Engaged State, the **CC** issued a ***TCIEngage*** event. This is a clear violation of the **TCI** interface specification by the **CC**.

The trace of **CC** events is as follows:

CCsync.CCInit,
TCIsync.TCIInit,

TCIsync.TCIInitOK,
SCIsync.sciInit,
SCIsync.sciInitOK,
DDIsync.ddiInit,
DDIsync.ddiInitOK,
CCsync.CCInitOK,
CCinQueue.CCActivate,
CCoutQueue.CCActivate,
SCIsync.sciActivate,
DDIsync.ddiActivate,
CCinQueue.ddiCCOn,
CCoutQueue.ddiCCOn,
CC2TCI.TCIEngage, (*)
CCinQueue.ddiCCDisable,
CCoutQueue.ddiCCDisable,
CCinQueue.ddiCCEnable,
CC2TCI.TCIDisengage,
CCoutQueue.ddiCCEnable,
CCinQueue.ddiCCResume,
CCoutQueue.ddiCCResume,
CC2TCI.TCIResume, (**)
CCinQueue.TCIAutoDisengage, (***)
CCoutQueue.TCIAutoDisengage,
CCinQueue.ddiCCOn,
CCoutQueue.ddiCCOn,
CC2TCI.TCIEngage.

Now we can see that this error is caused by the lack of synchronisation points in the protocol between the **CC** and the **TCI**. Because of the queue to **TCI**, the **TCI** is running behind the behaviour of the **CC**. As a result, when the **CC** encounters the ***TCIAutoDisengage*** event (***) it interprets it as being a response to the ***TCIResume*** command (**) it sent to the **TCI**; in fact, the ***TCIAutodisengage*** event (***) is a response by the **TCI** to the ***TCIEngage*** event (*).

In this situation, the **TCI** is effectively one set of state transitions behind the **CC**, leading to the sequence <**TCI**Resume, **TCI**Engage>. In other words, **CC** is trying to engage the **TCI** when it is already engaged.

There is no obvious solution to this problem without changing the **TCI** specification to give a notification when it changes state to Ready from Engaged in all cases, also when the **CC** instructs it to. This allows the **CC** to synchronise the unstable state transitions sufficiently to keep track of the **TCI** and avoid this problem.

This also requires us to make a change to the **CC** design. We have to include two new states, Suspending and Blocking, as “parking” states while the **CC** waits for confirmation that **TCI** has left Engaged State. The modified **TCI** behaviour will then be as follows: after **CC** sends a ***TCIDisengage*** command to the **TCI**, it will always receive a ***TCIError*** or ***TCIAutoDisengage*** event as a response. These may have been posted into the **CC** input before or after the ***TCIDisengage*** command was issued by the **CC**, but they appear to the **CC** as responses to that command.

3.4 SOLUTION 4

Now we rerun the check on **TCI** to see if the proposed design changes solve our problem. This time, the design generates no illegal behaviour from **TCI**.

Now we run a check for any illegal behaviour by any component to see if the system design functions correctly.

This check passes – there are no circumstances under which the **CC** component design emits an illegal response or any of the interfaces to **DDI**, **SCI** or **TCI** are misused by the design.

These are "timeless" verifications of functional behaviour. They do not take into account real delays in the systems that might have prevented some of these behaviours in practice. The checks assume anything can happen at any speed and checks all combinations of state transitions between all the components in the system.

This is an exhaustive examination of this design. Every possible scenario that the design can perform has been examined. It is equivalent to total test case coverage based on **execution paths** and not just executable statements.

The design is now ready to be analysed for ***safety*** and ***liveness*** properties to ensure that it meets its requirements.

4 CONCLUDING REMARKS

Verum Consultants are committed to the following software design principles:

1. Business Critical Software must be implemented from specifications and designs that are verifiably correct before implementation starts.

2. Software Architects and designers must restrict themselves to those architectures and designs that can be verified using currently available tools and techniques.

Verum invests heavily in research and development with leading universities to broaden the range of mathematical methods and tools we can apply to our client's needs.

During this exercise, we have seen how the dynamic behaviour of a software design can be examined by mathematical modelling and simulation. No programming was needed and indeed, none of the components involved in the design have been implemented before this verification took place.

This approach has two significant benefits:

1. It is much cheaper and quicker to verify designs this way. We have not invested in programming the design errors and then spent time debugging the code. Instead, we are able to verify designs and remove design errors before investing in programming. This results in implementations that are already at a higher quality level than is usually the case when testing starts, reducing project delays due to integration and testing problems. This means lower costs and shorter time to market.

2. It is much more certain to verify designs this way instead of trying to find such errors by testing the implementation. Many of the most difficult issues in software design are those related to timing problems, race conditions and concurrency. Such errors are very difficult to reproduce by testing, occur unpredictably and are very time consuming and expensive to find. More importantly, testing can never be complete; even the most extensive testing represents a very small sample of the total execution scenarios implemented in the software under test. Verifying designs mathematically before implementation is much more certain. These are mathematical proofs that hold in all possible cases and every execution scenario. They do not require Test Engineers to think of the "corner cases" and prepare test cases and test programs. The mathematical models of the designs embody every possible execution scenario and the model checker checks them all. This means higher product quality.