On Wheels, Nuts and Software

Michael Ellims

Pi Technology,
Ely Rd. Milton, Cambridge, CB4 6WZ England

`mike.ellims@pitechnology.com`

Abstract

In this paper I examine the issue of whether we can trust software systems and put forward an argument that in an absolute sense the answer must necessarily be no for a multitude of reasons. The paper then examines the question of whether this is an actual impediment to the successful application of software with particular reference to automotive applications. In particular I examine the question of whether our expectations of software are compatible with the realities of road vehicle manufacture and use. I conclude that with good methodology and integration within a whole vehicle development process, software based systems can, and will provide levels of safety above those which are experienced today so long as certain critical constraints are met.

Keywords: systems, requirements, safety, limits

1 Introduction

The question being addressed is "can we trust programmable technology?". In the context of safety related systems what does this question actually ask? Can we trust software? This question seems too limited, and perhaps should be, "can we develop software that is safe and reliable?". This is better, however this still excludes the environment in which the software operates and so the question can, and should, be rephrased as "can we develop software that is safe and reliable in all its interactions with the environment in which it operates?". This is the primary question which this paper attempts to address. The question is quite broad and to answer it we need to answer similar questions as to whether or not we can completely trust the components that make up software based systems.

There are a number of major elements that can be considered: system requirements, software requirements, the software itself and the hardware environment in which it operates i.e. processors, memory, wiring harness, sensors and so forth.

The paper is structured in two main parts. The first, section 2, broadly examines the question of whether we can trust the requirements, the software itself or the hardware on which it executes and attempts to establish what the limits are on our ability to construct software systems. The second, section 3, approaches the question by examining some of the evidence in the form of data on recorded failures in automotive software systems and goes on to put forward reasons why, when compared to other automotive systems, software based systems seem to be relatively successful. Section 4 examines the question directly by looking at some instances where software based systems add benefits, in particular for the case of electronic throttles. Section 5 then looks at what we can expect in the near term future and asks the question of whether the success achieved to date can continue. Section 6 examines some of the issues that the author has found problematic in the past and believes still need to be addressed. Section 7 draws some conclusions.

2 On the Limits of Correctness

2.1 Requirements

To be able to completely trust the requirements we need to be able to successfully capture all requirements. Is this possible? The answer to this question may be no, or at least not completely. Requirements capture involves explicitly stating what we want the system to do, what information is available to the system and stating what we don't want the system to do by means of a complete hazard analysis.

To perform this phase of the process successfully we have to know the answers to the questions above however the process is subject to very human limitations. For example in a study by Curtis, Krasner and Iscoe (1988) domain knowledge is identified as one of, if not the primary factor related to the success of a software system[1]. This is highlighted in [Redmill99] where the roles necessary to perform a HAZOP are enumerated. *Experts* feature prominently. Some of the areas listed as requiring input from experts include;

- Understanding hazards associated with the system
- Knowledge of *similar* systems
- Knowledge of the systems environment

What is being asked for here is that persons with relevant domain knowledge be involved in the hazard discovery

[1] This paper contains the truly wonderful quote, "Writing code isn't the problem, understanding the problem is the problem".

process. There are a number of issues associated with the process. For example in the second point noted above the word "similar" has been used. This is in recognition of the fact that no two systems are ever exactly the same. New systems usually introduce new functionality. Even if functionality were to remain unaltered something is *changing*; it may be the processor, the communications channels, the sensor set, it is however change.

The very fact that *change* is present means that there is no one who is "before the fact" an expert in all the aspects of the new system.

Another issue is that experts are usually expert in a single domain and for the development of systems requirements, this tends to imply they are not experts in programming. The converse is also usually true, programmers are not usually experts in the domain of the systems they are developing.

This in itself leads to some "interesting" problems. The first, and possibly most significant of which is communication. Experts in different fields do not use the same language, or rather do not always use the language in the same ways, this is true despite superficial similarities such as having "English" as a native tongue.

Possibly worse is that each profession tends to have its own formal notation (or notations) for encoding information. The point is raised by McDermid et. al. (1998) who point out that control engineers use differential calculus while it can be expected that "programmers" would use formal notations such as Z or B. Any translation for one system to another can introduce errors. If errors can be introduced then they probably will be. Of course translation from a system specified as a set of differential equations to English can, and usually will, also result in at least some translation problems.

Of course problems in communications are not limited to those associated with translation. Failure to pass information can also have catastrophic consequences. In their analysis of the Challenger disaster, Pinkus et al. (1997) argue that had the complete set of failure vs. temperature data been presented, including information for flights where no failures were detected, then the relationship would have been more apparent.

There is also the interesting question of how domain knowledge is gained in the first place. Domain knowledge derives from essentially two sources. Firstly there is the established body of knowledge that is available through formal education, books, research journals etc. Secondly, and more importantly for the current discussion is domain knowledge acquired though ones working life and professional activities. To some greater or lesser extent much of the knowledge acquired by this route will be empirical in nature and hence subject to greater or lesser degrees of validation and analytical analysis. Indeed it may not actually be possible to perform experimental validation.

For example consider the use of battery packs in hybrid electric vehicles. These packs are expected to last for the lifetime of the vehicle. However it is not possible to obtain precise data on how these packs will behave over the time periods for which they are required to operate before they are deployed. Various techniques can be used to obtain a good or rather reasonable estimate of their future performance using models of various types and applying empirical techniques, such as accelerated ageing. However precise data will not be available until the actual required life span (around 15 years) has passed. Of course by then technology will have moved on and the information may well be redundant.

2.2 Software

If we are given a "perfect" set of requirements then at least theoretically it would seem possible to produce "perfect" software (at least at the level of the source code) using techniques such as formal methods.

However, formal methods only usually take into consideration the specification for the software; they do not in general take into consideration any of the more "interesting" aspects of the hardware. Consider the following fragment of code:

```
if (((HW_can_var_name & 0xC0) &&
              (HW_can_var_name & 0x3F))…
```

This seems innocuous enough, until of course it's pointed out that the variable `HW_can_var_name` references a special purpose hardware register. Reading the register also clears it, so the second read is always zero. This of course should be hidden inside an access function and is of itself a rather trivial example but it demonstrates the principle.

The language that is used is also an issue. Most embedded systems are programmed in C. While this is not optimal from the point of view that C has many known flaws, e.g. Hatton (1995) and programming all systems in, for example, SPARK Ada (Barnes 1997) may be preferable, it is often necessary as no other compilers are available. Even SPARK Ada isn't perfect, it's just better. For example in chapter 2 of Barnes (1997) it is stated that analysis of floating point representations are performed using rational numbers.

Translators are available to convert Ada source to C source (e.g. Adatoccpptranslator 2004). However, even if perfect translation is achieved, the source itself is not being executed. It needs to be compiled. For some reason the compilation process still seems a bit of a black art, possibly because new and modified processor architectures are always becoming available. However, I will admit that it is not as bad as it once was, but that in itself has an interesting failure mode – on one project we lost 3 days (6 man days) of work because the two young engineers didn't even consider the possibility of their problem being a compiler error.

The following code compiled correctly for unsigned 32 bit integers,

```
    if (v1 < v2)…
```

But this did not,

```
    if (v2 > v1)…
```

Therefore if we accept that it is possible that errors can be injected into software by a compiler then we are left with no alternative but to attempt to reveal the errors by testing it.

There is also another oft-ignored issue. The processors used for embedded systems are usually what the desk top fraternity might "kindly" describe as grossly under powered. In addition some of the problems they are being asked to deal with are computationally quite complex; it is even possible that for some computations no analytical solution exists. The solution to this problem is to approximate solutions, usually with the introduction of large look-up tables that encapsulate the transfer function from inputs to outputs.

For example, on a modern PC a MatLab model of a combustion cycle in a 16 cylinder engine takes a factor of 10 longer than real time so it is hardly suitable for use in an embedded system. To compensate for this, the appropriate transfer function is normally "calibrated" as a look-up table using empirically derived data.

To give an idea of the potential size of the problem table 1 shows the size of the calibration data areas from three projects to develop engine control units developed over the period from 1990 to the present day. It should also be remembered that this is only for the main engine control unit.

Year	**Application**	**Data Size**
1990	Heavy duty "truck" engine	8K bytes
1998	Industrial Engine	32K bytes
2004	Passenger car	60K bytes

Table 1 : calibration sizes for three engine control systems.

Beizer (1990) noted this as an issue in 1990 and characterises the problem using the phrase "code migrates to data". The problem is in fact more general than the example of a transfer function given above, as the same techniques can be used for sequencing events or messages and controlling the behaviour of the system. In the limit this data can constitutes what Beizer terms "a hidden programming language" that possibly has as much affect on the system as the code itself.

The obvious problem with this migration, from the point of view of safety and reliability, is that we have moved the problem from one of ensuring that the program is correct to the associated problem of showing the data that controls the program is correct. A problem for which formal methods may be less applicable.

Of course when (not if, e.g. Yang 2002) neural networks are used in production systems, we have a whole new validation problem to consider.

2.3 Hardware

As we all know, software does not run in isolation. It runs on some sort of hardware platform. The problem with hardware, like software, is that it is also subject to errors in design, the example that springs to everyone's mind being the Pentium bug. However, this is not an isolated problem; many (possibly all) of the processors commonly used in embedded systems have some sort of hardware issues associated with them.

Hardware shares a number of characteristics very similar to software. For example showing that the hardware is correct seems to be as intractable a problem as showing software is correct. As with software, both formal methods and testing have been applied with varying degrees of success.

Interestingly there is a long history of attempting to adapt ideas from software testing to the hardware validation problem: for example Maurer (1990) approached the problem using context free grammars and recent work continues the trend. The most recent edition of IEEE Design and Test of Computers (March-April 2004) is devoted to this problem. A number of techniques for testing hardware are presented in that issue which in software terms range from functional testing at the system (i.e. chip) level to, most interestingly from the point of view of the current discussion, what Scafidi et. al. (2004) describe as "unit" testing of components in isolation.

Issues with hardware could be pushed down the chain to complete hardware modules, out into the connectors and the wiring harness and down on to the sensors and actuators. Unfortunately hardware devices often find unusual and interesting failure modes or behaviour which neither the manufacture nor the customer know about.

2.4 Retrospective

Above I have discussed a selection of problems that must be overcome if we are able to completely trust software.

Firstly the requirements have to be correct, and given the failings of human nature and the fact that we do not and in some situations cannot have access to perfect information, producing perfect requirements is probably impossible in the limit.

Secondly the actual software has to be correct. Again, if we ignore the problems associated with actually writing correct software, there are still serious issues associated with the use of compilers and the way in which solutions are approximated.

Just as bad, it appears that we can't even completely trust the hardware on which we are relying to correctly run the software!

3 Of Software in Practice

3.1 A reality check

Given all of the above the outlook looks bleak for being able to answer yes to the question "can we develop software that is safe and reliable in all it's interactions with the environment in which it operates?". However the argument given above seems more than slightly at odds with experience. Automotive software failures do not in general kill or maim large numbers of people every year! This is not to belittle the possibility that they may. It's a statement that currently they just don't.

Area Affected	Volume	Percentage	Examples
mechanical	638324	86.4	incorrect design, wrong parts
electric	71859	9.7	mainly problems with wiring harnesses being chaffed
assembly	10801	1.5	missing bolts, bolts not torqued correctly, bad welding
hydraulic	3553	0.5	contamination or leak of fluid
software	524	0.1 (1.9)	unknown, ECU replacements included

Table 2 : ascribed root causes of vehicle recalls for the year 2001.

However this view may not mesh with people's perception of reality. For example in the first three months of this year (2004) there have been two notable vehicle recall "campaigns".

- General Motors – vehicles recalled because of a delay activating antilock braking system at low speeds.
- Ford/Mazda – 470,000 Escape and Tribute SUVs because engine may stall at speeds below 40mph[2].

In these incidents, which are related to software, it is claimed that only eight minor injuries were caused.

In addition in the United States, the National Highway Traffic Safety Administration (NHTSA 2004) has opened a safety investigation into unintended vehicle acceleration in Toyota Camry and Lexus ES 300 cars. Again this may affect over 1 million vehicles. As the investigation is on-going it's too early to state if this is a software problem, but it also seems plausible that if a problem does exist it could well be a hardware issue. Throttles are known to stick: they are mechanical, they fail.

Based on only the above, the situation seems less than good. However consider that in AP2 (2004) it is also reported that GM may need to recall up to 1.8 million vehicles to replace faulty seatbelts. This is a simple mechanical system.

So how can we get some idea of how large the problem actually is? One thing we can do is look at accident statistics. For the year 2002 in the United Kingdom we find that there were a total of 3,431 road deaths (Dept. for Transport 2002). At the same time there were 302,605 casualties of which 35,976 where classified as serious i.e. requiring hospitalisation.

The above reference does not give the root cause of the accidents. However, the Parliamentary Advisory Council for Transport Safety (PACTS) web site does give some data in this area.

- young drivers (17-21) were involved in 15% injury crashes.
- 6% of all road casualties and 16% of road deaths occurred while a driver was under the influence of alcohol.
- 10% of collisions are related to driver fatigue.
- vehicle defects are a factor in an estimated 5% of collisions.

It would seem that there is a significant software problem, but from the above data it would seem to be that the neural network in the driver's head has been mis-trained! The software problem could be stated as being in "the nut on the nut behind the wheel"!

Returning to the original point, an examination of the 2001 recall data from the Department of Transport (see section 11 Appendix : UK Recall Data for details) reveals that a total of 739,107 vehicles were subject to recall.

Table 2 is my analysis of the root cause of each recall. Unfortunately the information above is not always precise enough to determine the exact cause of the fault, but most are obvious - e.g. a fault with airbags where one of four bolts was omitted can be ascribed to an assembly fault.

In the case of possible software errors 0.1% of the recalled vehicles (car, van, truck and motor cycles) I could classify the fault as a software error on the grounds that the unit needed to be reprogrammed. However if ECU replacement is also counted then this figure jumps to 1.9% of the recalls but again this could be because of hardware failures.

Looking at what the software errors actually are we find that;

- dashboard software warning light fails to come on – driver will not be aware that there may be uneven braking under extreme conditions (240)
- smart airbag software may not detect child seat (128)

Problems associated with electronic control units are as follows;

- unintended water ingress may cause airbag deployment (3055)
- active yaw control may leave differential clutch engaged resulting in damage to differential (156)

[2] From references AP1 and AP2 (2004).

- a fault with the unit may cause deployment of front airbag (14,045)

The last of the cases above is the most serious. The wording of the recall notice “exchange for a quality assured control unit” suggests a hardware fault rather than a software fault, but it is also possible the microprocessor has been mask programmed preventing its replacement in situ.

If (and it’s a big if) five percent of deaths and collisions and injuries are caused by vehicle faults then we could expect, in the UK, 171 deaths and 15,000 injuries to be attributable to the recalls above. Of these 2% or 4 deaths and 300 injuries would be caused by software systems. However we don’t see this many deaths attributed to software (or at least there is no evidence we do). In fact only a small number of deaths and serious injuries may be directly attributable to software, such cases arising mainly in America and which are still disputed.

Indeed a fair proportion of accidents that do involve vehicle faults can probably be ascribed to poor maintenance. Most of the author’s close calls are due to this, the list being impressive;

- loss of speed control, throttle pedal fell off, water ingress – rusted.
- loss of headlights at night, the Swiss AA man was very nice about it.
- partial loss of steering, main pivot came away from axle in Mont Blanc tunnel.
- loss of clutch, vehicle stuck in second, stalled.
- partial loss of steering, steering box failed (at 195,000 miles).

And that's only in old vehicles! In new vehicles, a clutch failure due to a hydraulic line failure, a starter motor relay failure, an alternator failure, and at a grand total of 23 miles - a rear bumper came lose on the M25[3]. The point of all of this is that all of these failures were mechanical in nature.

Casual inquiries within Pi Technology seem to indicate that many other engineers have experienced similar failures, the worst being a mechanical/hydraulic power steering system that wanted to turn hard right.

3.2 Why so good so far?

Given that we don’t seem to be killing and maiming people in large numbers because of defects in software systems one has to ask why? This is despite not adopting techniques such as formal methods and even often not completely understanding the technology and it’s interactions. Hoare (1996) has speculated as to generic reasons why this is the case. However I want to address what I believe are a number of reasons specific to the automotive environment.

[3] The London orbital motorway – often a four lane car park.

I believe that in some respects the automotive industry has a number of advantages over other industries. These can be briefly stated as follows;

- high-end first
- justified paranoia
- volume
- Californian Air Resources Board (CARB)

3.2.1 High-end first

New technologies are usually placed into high-end vehicles, i.e. expensive cars before they filter down into vehicles built for the masses. This has the advantage of low, initial volumes and relatively large development budgets.

The automotive industry uses these high-end applications to refine technologies and make them generic and thus cost effective before scaling up production by an order of magnitude. The scaling of course also increases risk by an order of magnitude.

3.2.2 Justified paranoia

One of the top priorities for any vehicle manufacture is to avoid recalls. This is understandable because, even if no mechanical parts are to be exchanged, the minimum cost of a recall will be around $50 per vehicle, which obviously for a million vehicles is a lot of money. This doesn't include the cost of lost sales or any costs resulting from litigation (fines, damages etc.). Audi sales dropped 60% over three years after an investigation into unintended acceleration, even though no fault was found and no recall was issued.

Vehicles are well tested before being released to the public. Typically a new vehicle will involve several years of testing before going into production, in a large variety of environments, winter testing being particularly problematic: hence the popularity of Waiorau in New Zealand for winter testing during the northern hemisphere summer.

It is instructive to look at one aspect of track work and the role it plays in safety. Specifically on a test track you can safely induce failures in the system to see what happens. In Pi Technology’s own test vehicles, track work has been used to gain assurance that the vehicle can be steered when the engine “fails”, that it can still be braked and that the vehicle is stable in corners.

Before production vehicles are released to the public individual vehicles in a test fleet will often be driven in excess of 100,000 km. This is in addition to performing purely software based testing e.g. unit testing as well as extensive bench testing.

3.2.3 Volume

The automotive industry produces large numbers of vehicles. In the UK alone each year in the order of one million vehicles are sold and a company like Ford produces on the order of three million vehicles per annum world wide. To gain some idea of what this means, if

70,000 vehicles of one model are on the road and each is driven for 5 hours per week this represents 18 million hours of use per year. This is probably an underestimate as many people spend more than an hour commuting one way!

Large manufactures also run large programmes to track faults, for the simple expedient that if you can fix a problem before you build more vehicles, then you can save on the recall costs.

The upshot of this is that there are statistically significant amounts of fault data available based on the *actual* usage profile of the vehicle fleet.

3.2.4 CARB

The automotive industry has one extremely bad side effect - pollution, and the other effects that ever increasing levels of traffic have caused. As far as I know, no one has ever had global warming in their hazard analysis! A side effect of pollution of course has been the legislative efforts to limit it. One of the more beneficial side effects being that engine control units now have to monitor their own "health".

For example one of the CARB regulations states "The monitoring method for the catalyst(s) shall be capable of detecting when a catalyst trouble code has been cleared (except diagnostic system self-clearing), but the catalyst has not been replaced".

The implications of this are that while it may be sufficient to monitor the temperature at which a catalyst operates to detect a problem, to meet the full requirement as given above probably means two heated oxygen sensors to compare oxygen level pre and post catalyst and in addition monitoring the performance of those sensors themselves for open or closed circuit conditions and how they are ageing via their switching frequency etc.

In addition, to ensure that the requirements will be met requires large amounts of engine testing to completely characterise the behaviour of the system. It has to be right: if the system is too sensitive then it will result in warranty claims, if it is not sensitive enough then the manufacture can be fined $25US per defect for each engine produced.

The effect of this is (or at least should be) that in general all electronic control systems will flag the fact that failures have been detected, whether they be within themselves, their associated sensors or in the actuators. Thus those items tend to be serviced before they fail catastrophically.

As an indication of the scale of the work involved, for one engine control unit I have been involved with, something like 24% of the total code is devoted exclusively to dealing with faults. But it must be noted that this does not include code embedded with control functions that actually detects the errors. It is probably safe to assume that around one third of the implemented functionality is directly associated with fault detection, mitigation and reporting.

3.3 Other mitigating factors

In section 2 I listed a number of factors that I believe limit the production of software based systems that could be considered absolutely safe. In summary these can be briefly stated as follows;

- requirements - do we know what we are doing?
- software - can we trust the code?
- hardware - will it always work?

In this section I want to examine, briefly, how the automotive development process deals with those specific issues and another important factor.

3.3.1 Requirements

On the subject of requirements there are several features of the process that lead us to believe that in general it is possible to produce adequate requirements for systems.

If we consider only new functionality, then obviously we start with the concept of what we want to achieve. This in turn will be followed by some design work and then, and in some respects most importantly, new functionality will be prototyped. This leads to an extensive body of empirical knowledge about the problem that will be encountered and more importantly how new functions will fail. In effect the process of prototyping increases the domain knowledge of the persons involved.

That prototyping is a (reasonably) effective tool for addressing the problems associated with requirements can be understood if one considers how the prototyping is performed. Firstly bench systems are usually made up for off vehicle development work. In cases where new or novel technology is being introduced full scale vehicles are also manufactured for use on test tracks, climate chambers, vibrations rigs, and later on the road, usually accumulating what for the average motorist would be significant mileage.

Once the development process has been completed and production ramp-up is being undertaken the testing continue, firstly with the manufacturers of the electronic sub-system (not usually the automotive manufacturers) who will replicate much of the above process (bench, track & test fleet) and finally pre-production with the vehicle manufacturers who again will run fleets to put significant mileage on vehicles.

Fundamentally there is nothing wrong with the process described above. The only significant issue arises from the fact that the work is being undertaken by (usually) two different groups of people which need to communicate. As noted above, communication is always an issue.

3.3.2 Software

In general the software written for vehicles cannot be considered rocket science. That is, it is not usually pushing the boundaries of what is technically possible (at least not yet) and as such is reasonably amenable to good software engineering practice.

At the design level, techniques such as dynamic memory allocation and recursion are usually banned and there is surprisingly little use made of pointers (for obvious reasons).

The coding process is in the main still largely performed by hand, although many manufactures, including ourselves (Wartnaby et. al. 2003) are attempting to at least semi-automate the process by auto-coding from Simulink "models" and similar tool sets.

Code is reviewed, and static analysis is usually performed, often only to meet the MISRA C guidelines (MISRA:1998), however tools such as PolySpace which perform non-standard execution of code are becoming commonplace.

At a minimum, systems will be black-box tested against requirements and for systems of a more risky nature it is common for unit testing at the functional level to be undertaken. For example the Ford standards require that statement coverage is achieved for all software and branch coverage is recommended for system with safety mitigating software. Daimler-Chrysler even has a software research team actively investigating automatic test data generation (e.g. Wegener 2001).

3.3.3 Hardware

Of all the three areas that I have considered hardware - at least as far as concerns microprocessor hardware is concerned - is the *least* amenable to being brought under control.

This is because it is, for the most part, out of our control!

As far as hardware *correctness* within an application is concerned the only technique open to the automotive sector is to perform tests that match the expected use of the any hardware features to gain confidence that the hardware will perform as expected.

On the subject of *reliability*, processor hardware does pretty well. For instance with one engine mounted ECU for which I have some information, of the warranty returns only three instances (0.001% of production) were recorded where failure of the ECU was attributed to the processor failing.

3.3.4 Loosely coupled systems

In the terms of Perrow (1999) current automotive systems can be characterised as loosely coupled, linear systems.

Automotive systems are loosely coupled in the sense that in general a failure in one sub-system does not normally severely impact the functionality of all other sub-systems. For example a failure in the breaking sub-system will not normally affect the steering and when it does it usually degrades the performance not remove it. Likewise a failure in the engine does not remove all braking ability, although there is a coupling in that there will be some loss of power assist, and loss of engine power can also affect power steering capability. However without the engine you should stop as the engine itself becomes a braking system[4].

There are other options available to the driver as well: if the engine runs away, you have the option of turning the ignition off, it should stop the fuel pump - but don't lock the steering wheel! The clutch can also be used in a manual to disconnect engine from the wheels. There is also the emergency (park) brake as a final backup system.

If all else fails there are crumple zones, seat belts and airbags all of which are independent of the driveline.

All of the above tend to limit the severity of systems failures, as noted in section 3.1 vehicles can suffer quite large systems failures and not cause accidents. This doesn't necessary mean they won't, but it does provide a safety margin.

4 Is it safe?

At this point we should be in a position to look at the main topic of this conference; that is, can we trust the software (systems)?

The arguments present in section 2 suggest that the answer should be no. However with the examination of data present in section 3 it was seen that in practice issues with software accounted for only 2% of safety related vehicle recalls (as recorded by the Department for Transport) vs. 84% for some potentially very serious mechanical failures.

If we examine just problems with throttles we find a total of 52,378 vehicles were affected by recall notices in the years 1998-2001. Analysis revels that 27,903 vehicles were recalled due to design flaws in the throttle body itself. A further 13,566 were cause by either mats being trapped under the throttle pedal or interference between the throttle cable and the front bulkhead. Some 3,243 were caused by interference between of the throttle cable or other linkages within the engine bay and 4,649 were indirectly caused by the introduction of electronic controls. But again, the direct cause is mechanical, part of the mass air flow (MAF) sensor could become detached and jam the throttle open. No instances were found where a throttle problem could be attributed to software.

The obvious question is why? There are a number of reasons. Firstly an electronic system removes the cable and other mechanical linkages from the system. At the same time it allows the introduction of redundancy[5], electronic throttle pedals have dual input sensors with different signals so they can be cross checked and failures detected. Hence if one fails a "trouble code" should be recorded and the warning lamp should come on.

Secondly the electronic throttles themselves are self-contained units, which reduces the possibilities of

[4] In effect what is known as "jake brake" though less efficient.

[5] Note the Volvo 740 has twin throttle cables so this is possible in a mechanical system, but not usual.

external mechanical interference, though internal failures can sill occur as noted above.

Thirdly, there are more possibilities to cross check the system, for example information from closed loop fuel control can be used to infer the approximate throttle position and hence be cross checked against the throttle position feedback from the throttle itself. Note that again the issue of domain knowledge is present – you have to know that the cross check described above can be performed.

Other inputs can also be used, for example it is possible to infer that if the brake has been *hard* on for some time and the throttle is wide open then something may be wrong. Note the use of the word hard above One advanced driving technique known as left foot braking involves the use of both the throttle and brake at the same time. Again you have to know about this to be able to take it into account.

5 Will it continue?

The features outlined above have, in general, served the industry reasonably well to date, but will it continue?

If one looks back at how electronic controls have been introduced we can note several features. First in the beginning there were single electronic systems, controlling simple functions such as spark timing. Over time these have evolved into more complex systems e.g. the modern engine control unit now has to control spark timing, perform knock detection and control, detect misfire, monitor the air/fuel ratio and health of the catalyst, run the throttle and fuel injectors etc.

In addition the number of computer controlled functions has increased, so we have ABS, traction control, instrument clusters, computer controlled automatic transmissions, electronic park brakes and so on. On it's own each of these systems is simple enough and in isolation their effects can be reasonably bounded and failure modes analysed and accounted for.

However, individual control systems no longer operate in isolation. They are all part of the same vehicle, and increasingly they need to communicate and co-operate with one another. In addition, many of the components that now interact were originally conceived with different design philosophies. For example instrument clusters were at one time standalone devices that displayed information such as engine speed, "mileage", engine revolutions etc. As such they have been traditionally treated as SIL 0 systems. However it was shown above that at least one safety recall has involved a cluster where a dashboard software warning light related to the ABS function failed to come on.

Given the above I perceive that many of the problems that will be seen in the future are going to be system problems where interactions between components that have “traditionally” supplied isolated functions are required to interact.

6 Continuing Issues

Flowers (1996) puts forward the thesis that all software failures can be attributed to a failure of management, and indirectly at least that may well be the case. Failures due to management can have any number of causes, such as failure to provide adequate tools, to allow enough time, to maintain proper oversight on contractors and suppliers and so on ad infinitum.

Part of this is due to the misconceived idea that “it’s just software” and that “software is easy, it’s just typing”. However, typing is to software, as drawing is to mechanical or civil engineering. It’s hard to do mechanical engineering if you can’t draw; but it’s not the only skill necessary.

The problem has several roots. Firstly software is intangible so all you see is the typing. Software tools like SimuLink may help here as they graphically and explicitly show the complex interactions in a familiar manner. If nothing else, they look like circuit diagrams and people “know” electronics is hard.

Secondly, for most people software *is* easy. For example, most graduates have experience of programming at some level, even if it’s only programming a spreadsheet. Unfortunately the problems faced as an undergraduate actually aren’t that complex. They can’t be. They have to fit within the structure of a university course. This then perhaps leads to an attitude of “If I can do it, why can’t you?”

Finally, as Kuhn (1962) would say, there has been a paradigm shift. The software in a vehicle will soon account for a significant proportion of the vehicle cost. Just over thirty years ago it accounted for nothing. The long history of isolated component systems hasn’t helped as we are only beginning to discover issues arising from the non-linear coupling effects associated with multiple, communicating components.

This then is the first challenge - management needs to understand that it’s not “just software”, it’s engineering. As such, more attention needs to be placed on the integration of the software development process in the context of the vehicle systems it interacts with both directly and indirectly. We need to be able to manage this additional complexity, to expect it and to build into the process activities and *time* to deal with it.

The above may all be true, however there are other systematic problems associated with the production of software based systems.

In the three issues I put forward as major impediments to writing absolutely correct software in section 2 requirements was listed first. This was not random, it is my experience that the quality of requirements is the biggest single barrier to being able to produce software based systems.

For example, it was stated in section 2.1 that *change* always introduces problems. On an engine we decided to implement knock detection by using an FFT algorithm running on a DSP coprocessor using one sensor for each of 16 cylinders. Normally this would be done using some sort of band-pass filter. While the requirements for sampling the data were correct, associated requirements for deciding if the sampled data indicate whether knock was present were not. In theory we knew what we were doing, but in practice we lacked the empirical domain knowledge, particularly during transitions between operating states of the engine. Happily a few *months* of experiments (i.e. we gave up and built prototypes) resulted in a working system that is also potentially capable of detecting failures in valves and the main bearings.

As a further example, a manufacturer has stated that they have problems getting their best engineers to work on requirements, mainly because it is perceived as low status work!

If we can't get the specifications right then the software has no hope of working first time, effectively you will be prototyping the system - even if that was not your intention. This is both a technical and a managerial issue. Technical in the sense that we need to be able to determine the requirements and document them. Managerial in the sense that the management of that process needs to be better defined and controlled. You can't build reliable, safe systems based only on prototypes.

Another issue we need to address is that we have to learn that system testing the software is, and probably always will be, a horrendous problem. It takes longer and costs more than anyone wants to pay, but worse, we leave it until last. If you have a requirements specification then you can decide how to test it. The very process of designing tests will change the specification. Again Beizer (1990) has noted that "more than the act of testing, the act of designing tests is one of the best bug preventers known". Which is precisely my experience.

Thus we need to design systems that can be tested, as components, as sub-systems and finally as complete systems. It is necessary therefore that we take the attitude that the testing of systems be considered as important as the design of systems and that it be done as an "up front" activity and not something that is done when we are finished. Thus being able to test a system is a *requirement* on any system; unfortunately it is a requirement that is often ignored until it's too late to directly incorporate it. Again this is both a technical and managerial issue. How do we design for test and how do we incorporate testing effectively into the development process.

We also have to learn how, at the vehicle level, to design systems that are, "simple". By this I don't mean trivial, I mean systems that have limited interactions and dependencies, systems that are amenable to analysis and systems that limit and isolate their own faults and that can communicate required fault information to other affected parts of the system. This should be possible; the author has performed such an analysis (Ellims 2001) on a simple distributed control system. Probably if the author can, anyone can.

As stated above this is an issue of managing complexity, it is also a technical issue in that we need to educate engineers that simplicity is a virtue.

All of the above are what can be termed "big issues". However until we can deal with these we can only make limited progress towards building systems that can truly be regarded as "safe". The big issues are what I have chosen to concentrate on in this paper. However, as the saying goes "for the want of a nail the shoe was lost". If we are honest we still have problems with software "nails". However, as stated previously this appears to be at least partly amenable to technical solutions.

7 Conclusions

It would appear that current programmable automotive systems appear to be safe, at least relative to some mechanical systems. This is possibly more due to good luck than good judgement. Luck that to date complexity has been limited, luck that we have not yet attempted to build tightly coupled systems, luck that the majority of failures do not have catastrophic consequences.

However as was noted in section 6 it's easy to identify issues associated with how automotive systems are developed. As the systems being deployed gain more interactions, become more tightly coupled and start to take over functions from the driver this approach will necessarily begin to fail.

Therefore the automotive industry is going to have to get good at systems engineering - fast. The first of the new generation of systems are just around the corner.

8 Afterword

It should also be noted that this paper is developed from the authors experience in the automotive and related industries. As such it should only be considered as being representative of the industry and practices within it, not as a definitive assessment. Neither can the author claim that the views expressed here are universally accepted within the industry and as such it should be considered to be a personal assessment of the state of play.

9 Acknowledgements

The following all contributed to the development of this paper either by discussing ideas with the author or reviewing final drafts. Graham Tebby, James Bridges and Keith Jackson of Pi Technology and John McDermid of the University of York.

10 References

Adatoccpptranslator: http://adatoccpptranslator.free.fr/ accessed May 2004.

AP1: Associated Press: 13 April 2004, in Detroit Free Press.

AP2: Associated Press: 15 April 2004, in Detroit Free Press.

DFP: Detroit Free Press, 9 March 2004.

Dept. for Transport (2002): Department for Transport, "Transport statistics bulletin: Road Casualties in Great Britain Main Results 2002". http://www.dft.gov.uk/ accessed May 2004.

Beizer, B. (1990): *Software Testing Techniques: Second Edition*. London, International Thomson Computer Press.

Curtis, B. Krasner, H. Iscoe, N. (1988): A field study of the software design process for large systems, *Comm. ACM* , **31**(11):1268-1287.

Barnes, J. (1997): *High integrity Ada: The SPARK approach*, Harlow, Addison Wesley Longman.

Ellims, M. Parker, P. Zurlo, J. (2002): Design and Analysis of a Robust Real-time Engine Control Network, *IEEE Micro (Special Edition: Critical Embedded Automotive Networks)*, **22**(4): 14-19.

Flowers, S. (1996): Software failure: Management Failure, Amazing Stories and Cautionary Tales, Chichester, John Wiley & Sons.

Hatton, L. (1995): Safer C: Developing Software for high-integrity and safety-critical systems. Maidenhead, McGraw-Hill.

Hoare, C.A.R. (1996) How did software get so reliable without proof? *Third Intl' Symp. on Formal Methods Europe (FME'96) Industrial Benefit and Advances in Formal Methods*, Oxford United Kingdom, LNCS 1051:1-17, Berlin, Springer Verlang.

Kuhn, T.S. (1962), *The Structure of Scientific Revolution*s, Chicago, University of Chicago Press.

McDermid, J. Galloway, A. Burton, S. Clark, J. Toyn, I. Tracey, N. Valentine, S. (1998): Towards industrially applicable formal methods: three small steps, and one giant leap. *Proc. Int'l Conf. on Formal Eng. Methods (ICFEM'98)*,Brisbane, Austrailia, 76-88, IEEE.

Maurer, P.M. (1990): Generating testing data with enhanced context-free grammars. *IEEE Software* **7**(4):50-55.

MISRA (1998): Guidelines for the use of The C Language in Vehicle based Software, Nuneaton, the Motor Industry Research Association.

NHTSA: http://www.nhtsa.dot.gov/ investigation PE04021 accessed May 2004.

Perrow, C. (1999): *Normal Accidents: Living with High-Risk Technologies*, Princeton, Princeton university press.

PACTS : Parliamentary Advisory Council for Transport Safety, http://www.pacts.org.uk/policy/briefings/statistics_uk.htm accessed May 2004.

Pinkus, R.L.B Shuman, L.J Hummon N.P. Wolf, H. (1997): *Engineering Ethics Balancing Cost, Schedule and Risk - Lessons Learned from the Space Shuttle*, Cambridge, Cambridge University Press.

Scafidi, C. Gibson, J.D, Bhatia, R. (2004): Validating the Itanium 2 exception control unit: a unit-level approach. *IEEE Design and Test of Computers*, **21**(2) 94-101.

C.E. Wartnaby, S.M. Bennett and M. Ellims, R.R. Raju, M.S Mohammed, B. Patel and S.C. Jones, (2003): Auto-generated production code development for Ford/Think Fuel Cell Vehicle Programme *SAE Word Congress,* 2003 *SAE Technical Paper Series 2003-01-0863*

Wegener, J. Baresel, A. Sthamer, H. (2001): Evolutionary test environment for automatic structural testing, *Information and Software Technology*, 43 841-854.

Yang, H. Yizhang, L, Wasacz, B. (2002): Neural Network Based Feedforward Control for Electronic Throttles, *SAE Technical Paper Series 2002-01-1149*

11 Appendix : UK Recall Data

All data on safety related vehicle recalls has been extracted from public sources. The data on vehicle recalls between the years 1998 and 2001 has been extracted from the following documents which were accessed from the Department for Transport (DfT) web site in April 2003 and stored locally on the authors computer.

RECALLS_CAMPAIGNS_BULLETTINS_1998_Jan-Jun.pdf

RECALLS_CAMPAIGNS_BULLETTINS_1998_Jul-Dec.pdf

to

RECALLS_CAMPAIGNS_BULLETTINS_2001_Jan-Jun.pdf

RECALLS_CAMPAIGNS_BULLETTINS_2001_Jul_Dec.pdf

These documents are no longer available from the web site and in their place access to a database has been provided where searches can be performed for specific vehicles. This is of course not ideal for the current paper so data after December 2001 is effectively not available via the web. The DfT have however agreed to supply the author with paper copies from July 2004.