Design and Implementation of a Safety-Critical Application Targeting Modular Certification

Andreas Platschek and Nicholas Mc Guire
OpenTech EDV Research GmbH
A-2193 Bullendorf, Augasse21
<andi@opentech.at>, <der.herr@hofr.at>

Abstract

Modular Certification promises to allow the reuse of already certified applications as well as co-located non-safety related applications, by showing the independence of the application from other software employed on the system. While most publications focus on the operating system that guarantees the independence of applications, this paper takes a detailed look at the life-cycle of the application and how the development of such an application differs from developing a software application in a federated architecture.

In order to get an idea of how feasible an actual modular certification is, this paper starts off by analyzing the landscape of standards that support modular certification. This short summary of standards is followed by a generic analysis of the problems arising when going for a modular certification as well as the potential advantages gained. The rest of the paper presents a real-world example for designing and implementing a typical application of the automotive domain (an indicator control system) into an integrated architecture.

This real-world example does not only show the technical aspects, but one can easily deduce the economic implications of a federated approach - namely the simplification for small software companies to enter the market and write applications for the automotive industry. Furthermore this approach encourages the usage of pre-existing FLOSS software by ensuring the independence of applications and decoupling the system safety requirements from the application safety integrity by providing adequate isolation with respect to fault impact and propagation.

1 Introduction

As one of the first industries with stringent system safety requirements, the avionics industry introduced virtualization systems [1] for the deployment of safety critical applications. In the avionic industry these types of systems are called IMA (Integrated Modular Avionic) systems [3, 4], and their usage is known for a number of modern airplanes from fighter jets to civil transportation.

Some of the main advantages of the IMA approach are the high level of modularity - allowing to build systems that co-locate applications or run-time environments of different safety integrity levels. A further key issue is that the application development is de-coupled from physical run-time system constraints - basically a IMA application need not know on which node of the system it may be deployed. Further, and this is strongly echoed in all relevant safety standards, a IMA like concept built on strong temporal and spatial isolation allows for independence of monitoring and run-time/application, including system level response to faults in particular partitions and thus guarantees for the environment that an application/run-time will be exposed to. This later property is essential for composeability - which is maybe one of the strongest driving forces behind the IMA like system designs.

In other industries, the deployment of virtualized systems has not yet been that popular but many domains are moving slowly but surely towards this new design paradigm where virtualization plays a central role.

In example in machine tool industry a similar, though not quite as clearly visible, trend can be observed. Machine tools are traditionally built up of major components that are from different vendors, i.e. drives from vendor A, control system from vendor B, etc. This led to the same composeability problems for systems that need to be certified as in the avionics industry. The solution one can find in guiding standards like IEC 62061 is that a certified complex component can be treated as a low complexity component (Clause 6.7.4.2.3) in the system composition thus significantly reducing certification efforts (Clause 6.7.3.3) but without breaking the continuous cross-checking of requirements fulfillment and design match.

In order for deploying such an integrated/virtualized software system one needs of course a VMM (virtual machine monitor) of some sort that handles the scheduling of the various independent applications. As the central software module, which due to their central role must be implemented conforming to the highest safety integrity level, the design and implementation of these VMMs gets a lot of attention. The proper design of applications running on top of a VMM - especially to obtain a design that lends itself towards modular certification - is a topic which has, in my opinion been neglected. Only doing this right as well will result in a high probability that application software can be re-used without the need of re-certification for a very long time. In this paper, I discuss some of the problems I came across and present a design of a simple but not unusual for the MCUs in the automotive industry application to show some of the traps and pitfalls, and how to avoid them.

2 Standards

As already mentioned in the introduction, the avionic industry is the leading industry in the deployment of integrated solutions. Therefore the application specific standards naturally are avionic standards [3, 4, ?]. From the functional safety side of things the use of integrated approaches is backed up by the DO178C [5].

What about other industries? - As this paper focuses on the automotive sector, below attention is focused on the guidance given by ISO 26262 as the functional safety standard for automotive and IEC 61508 as the generic functional safety standard. Similar clauses can be found in standards in most industries.

2.1 ISO 26262

The latest standard on functional safety that has just been introduced into the automotive industry is the ISO 26262 [6]. This is the first domain specific standard on functional safety in this industry, and it also gives guidance on the usage of integrated software architectures. The most important parts here is *ISO26262-6, D2.2 - "Impact on shared resources"*. Although it allows some more dynamic mechanisms than the ARINC standards, it mandates the same properties (these properties will be discussed in section 3) of the system and prefers the mechanisms also used in ARINC653.

2.2 IEC 61508

IEC 61508 allows the use of pre-existing software side by side with software developed according to a safety standard. If the pre-existing software can not be guaranteed to have properties equivalent to a safety related development then the standard requires appropriate evidence of non-interference. Further IEC 61508 also mandates this in the case where software of different safety integrity levels are to be co-located on a single platform. Clauses 7.4.2.8 and 7.4.2.9 cover this aspect in IEC 61508-3 Ed2 2010.

A further aspect that is introduced in Clause 7.4.2.7 of the same standard is that where reasonable self-monitoring should be provided on data and control flow and the system shall be able to detect and react to failures.

It is quite natural to cover these requirements by appropriate system level isolation methods (MMU,IOMMU) as well as process level separation (address spaces) - but if one were to do this based on a full featured operating system then this OS would need to be at the highest safety integrity level claimed - which makes reuse of COTS components quite complex. This can be significantly simplified if the separation properties are provided by a relatively small and thus in principle certifiable hypervisor.

The second noted aspect - monitoring and fault-response - is covered in ARINC 653 by the health-monitoring facilities. This allows to have an independent instance that will respond to detected faults in the system and thus provides some level of guaranteed fault response independent of the faulting unit (OS or application).

3 ECU Design

As shown in the last section, there are a number of standards in a variety of domains that support integrated/partitioned architectures. To summarize the most important principles all of these standards that give guidance on how to reach independence of software applications on the same hardware node the three core principles are the same:

- separation in time (cyclic scheduling)
- separation in space (memory protection)
- independent communication (strict polling semantics)

Depending on the standard they can be softened a little bit (e.g. ISO 26262 also talks about other scheduling strategies than cyclic scheduling), but even if other strategies are followed, it is necessary to show the hard separation in time, which of course is harder to show with a more complex scheduling algorithm.

For this paper the only important thing is, that all of these high level principles are provided by XtratuM. They can be seen as SACs (more on safety application conditions in section 6) at the level of the hardware node, that are fulfilled by XtratuM. The more important issue for this paper are the issues that arise at partition (application) level.

4 Standardized Interfaces

Developing applications for the long-term usage relies on one crucial design element - the application must use a well defined API that will not change for a long time and is an accepted industrial standard. This obvious pre-requisite for long-term reusability of software is given in the automotive industry where such a stable API is specified in OSEK/VDX [7, 8, 9] and extended by AUTOSAR [10].

Therefore to create an integrated architecture for the automotive industry providing an OSEK/VDX compliant interface to the applications is mandatory. This interface provides all generic functions an application will need like task management (creation, scheduling, termination), communication (between tasks and with other applications), interrupt and even handling, resource and time management (alarms) as well as error handling. Having all those elements well specified assures that an application can be reused for a long time - essentially as long as a run-time environment that provides this interface is used.

5 Indicator Control Example

In order to test the approach I proposed in my thesis a relatively simple real-world problem had to be found. This example is an indicator control, as depicted in 1. This figure really just shows the inputs and outputs of the controller.

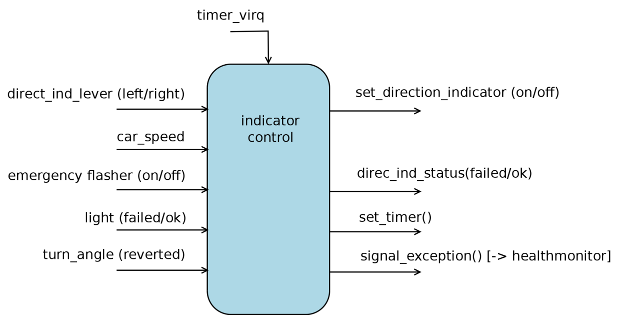
timer_virq
direct_ind_lever (left/right)
car_speed
emergency flasher (on/off)
light (failed/ok)
turn_angle (reverted)
indicator control
set_direction_indicator (on/off)
direc_ind_status(failed/ok)
set_timer()
signal_exception() [-> healthmonitor]

FIGURE 1: *Example: Indicator Control*

A general safety process roughly exhibits the following overall structure:

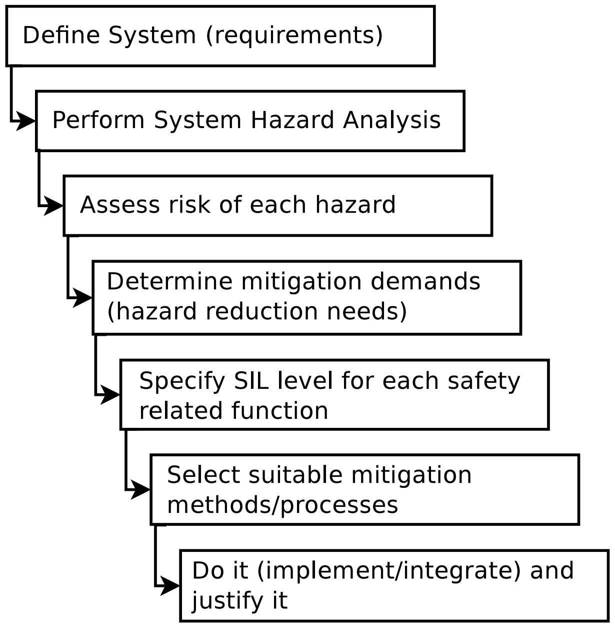
Define System (requirements)
Perform System Hazard Analysis
Assess risk of each hazard
Determine mitigation demands (hazard reduction needs)
Specify SIL level for each safety related function
Select suitable mitigation methods/processes
Do it (implement/integrate) and justify it

FIGURE 2: *Workflow of the safety process*

The ideal safety process is a waterfall model - in reality of course it is an iterative process at every step.

The approach taken in the design and implementation of this simple example represents a subset of the overall safety process and included the following steps:

- do a naive high-level design of the problem
- perform a hazard analysis on this naive high level design
- the output of the hazard analysis is used as input to refine the design
- do a detailed design of the components of the high level-design
- do a hazard analysis on the interfaces between the components

6 Hazard Analysis

Designing an application as simple as an indicator control seems trivial and one tends to think *"What can possibly go wrong?"*. The truth is, despite the example being that simple it was not that hard to find some weak spots. To get the hunt for problems into a structured form, a HAZOP (Hazard and Operability study) was performed on the naive design.

A HAZOP uses a set of generic keywords that cover all deviations possible these are interpreted in the specific context and allow a systematic scan for flaws [**?**].

This HAZOP lead to the discovery of a number of possible hazards that were not anticipated by the original design and were very useful in refining the naive design into the refined version of the high level design.

Besides the real design flaws that were found, a number of SACs (Safety Application Conditions) had to be defined and have been listed. These SACs specify the pre-conditions that have to be met, in order to allow a safe operation of the application.

SACs are an important tool to allow reuse of components (software and hardware) and to define the environment this component can be used in. They basically consist of restrictions due to

technical problems - e.g. situations were a clean solution would need an unproportionally high effort in time and money or were a clean solution would result in a system that is not usable due to size, power consumption, ...restrictions. An example of a technical problem that has to be handled outside of the software application in the indicator control example is *"If the battery is dying, it has to be assured that vital elements of the vehicle - e.g. the emergency flasher - function properly as long as possible, while others can be taken out of operation to save energy (e.g. the parking assistant). This kind of energy control has to happen on a vehicle wide level and can therefore not be handled by the indicator control system. "*

necessary assumptions - e.g. assumptions that had to be made at development time because the hard facts were not available. In these cases the terms for usage have to be communicated to the user of the application via the SACs. In the indicator control example, a SAC from this category is *"The indicator control application is occupies one OSEK partition. No other application is allowed to be located in this partition."*.

A safety application condition for a technical problem could be simply to restrict a system in its usage profile. If the overload stability of an OS is not known or can not be reasonably assured then a SAC may limit the OS load to a perceived "safe" value. This is akin to safety margins in other engineering disciplines (like buildings) - by restricting the system load to 30% CPU usage one gains a safety margin for the "unhandled cases".

The second type of SAC is more specific to generic software. In general safety related projects start out with a system and an appropriate analysis of the potential hazards related to this system. In generic software we can't always do that because we don't actually know the system. To mitigate this one needs to make "educated assumptions" and document them in the form of SACs. and example might be that one assumes that all memory of the safety applications are statically allocated at application launch and checked before the application goes "hot" - the SAC now simply would state that this assumption must fit the applications, if not then a re-assessment is needed.

At this stage we got a final version of the high level design that has been analyzed and is assumed to exhibit residual safety flaws with an acceptably low probability (assumed, because the HAZOP analysis is a team effort, but due to having no team for my thesis I had to do it on my own). Now a more detailed look at the problem can be done in order to create the detailed design.

7 Detailed Design

After the high level design has been completed a more detailed design can be done. For more complex applications the design could be broken down multiple times, for a simple application like the indicator control example the next level of design can already be as detailed as an interface design of the applications components.

Having the API and the communication between components designed, again a risk and hazard analysis has to be performed in order to find out if mitigations for all possible hazards are in place. For the indicator example a FMEA (failure mode and effects analysis) on the interface has been performed. Let's for example look at the following simple function:

```
StatusType read_input(struct in_data *input);
```

Performing a FMEA on this function results in the detection of the following possible hazards:

function: read_input

- pointer == NULL

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: check for null pointer, return ERROR_CODE

- pointer == 0xDEAD (also known as pointer to lala-land)

  what happens: access to invalid memory, application terminated through health monitor, health monitor is expected to signal error.

  effect: restart of partition

  mitigation: component is in undefined state, internal mitigation not possible ==¿ has to be handled by health monitor.

- pointer points to valid address space, but wrong data structure

  effect: no detection by system

  mitigation: extension of data structure by unique magic number.

The result of this FMEA could turn out in various ways. It could either show how all possible hazards are handled after applying the mitigations. If it hazards that are not handled by the mitigations, the detailed design has to be redone in order to assure safety.

8 Conclusion

While the overall process for a safety related application is relatively complex the main issue is the system level interaction and the dependencies. If these dependencies can be minimized and the individual functionalities isolated to a level where non-interference can be assured (with reasonable probability) then the overall safety process ends up being quite tractable - aside from it having significant positive effects on software quality and maintenance.

The method introduced here is efficient if a partitioned system forms the foundation - providing the essential system level properties of

- temporal isolation
- spatial isolation
- side effect free communication

are provided. In the case of the OVERSEE platform design, these system properties are conceptually provided by the XtratuM hypervisor. On top of these core safety properties, it is quite simple to define small, maintainable and safe applications for automotive systems.

By following standard APIs (OSEK/VDX, AUTOSAR) a highly composeable system can be achieved which exhibits strong safety properties.

9 Acknowledgments

This paper has been produced in the context of the OVERSEE project (FP7-ICT-2009-4, Project ID 248333).

References

[1] John Rushby: *Partitioning forequirements, Mechanisms, and Assurance*, 1999

[2] *OVERSEE*, 2010, HTTP://OVERSEE-PROJECT.COM

[3] *ARINC651 - Design Guidance for Integrated Modular Avionics*, Aeronatical Radio Inc., 1997

[4] *ARINC653 - Avionics Application Standard Software Interface*, Aeronatical Radio Inc., 2003

[5] *DO-178C - Software Considerations in Airborne Systems and Equipment Certification*, RTCA Inc. , 2011

[6] *ISO 26262 - Road vehicles – Functional Safety*, International Organization for Standardization, 2011

[7] OSEK/VDX consortium,OSEK/VDX Homepage, HTTP://OSEK-VDX.ORG/

[8] *OSEK Operating System Specification 2.2.3*, 2005, OSEK/VDX CONSORTIUM

[9] *OSEK/VDX Communication Version 3.0.3*, 2004, OSEK/VDX CONSORTIUM

[10] *AUTOSAR - Automotive Open System Architecture*, HTTP://AUTOSAR.ORG/

[11] *System Safety: HAZOP and Software HAZOP* , Felix Redmill, Morris Chudleigh and James Catmur, Wiley, 1999