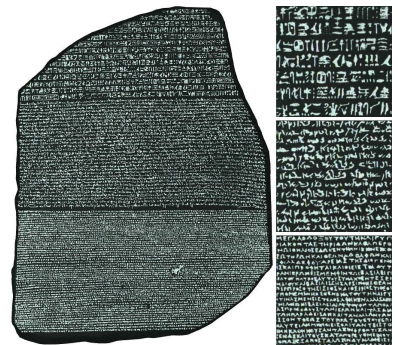
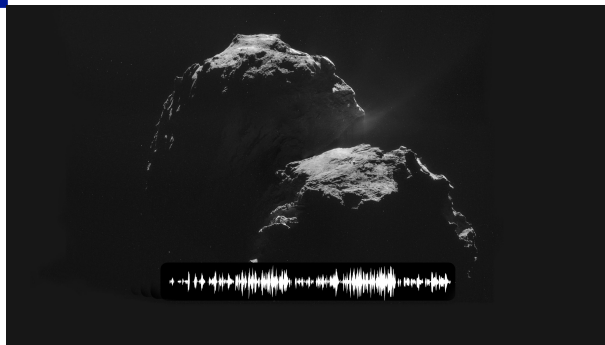




Altreonic

*From
Deep Space
To
Deep Sea*

Trustworthy Systems Engineering with GoedelWorks 3



First publication in
the Gödel Series

**SYSTEMS
ENGINEERING**



This publication is published under a

[Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License](https://creativecommons.org/licenses/by-nc-sa/3.0/).



***Altreonic NV
Gemeentestraat 61A B1
B-3210 Linden
Belgium***

***www.altreonic.com
info.request (@) altreonic.com***

***September 2011
Last update - July 2015***

***© Altreonic NV
Contact: goedelseries @ altreonic.com***

PREFACE

This booklet is the first of the Gödel* Series, with the subtitle "Systems Engineering for Smarties". The aim of this series is to explain in an accessible way some important aspects of trustworthy systems engineering with each booklet covering a specific domain.

The first publication was entitled "Trustworthy Systems Engineering with GoedelWorks" and explains the high level framework Altreonic applies to the domain of systems engineering. It discusses a generic model that applies to any process and project development. It explains the 16 necessary but sufficient concepts, further elaborated by defining a standard template structure for a Work Package. The version you are reading now is an update reflecting the metamodel used in version 3 of the GoedelWorks portal. This version is based on a more extended metamodel. In particular it provides a generic template metamodel for the internal structure of a Works Package.

Validation of the approach was achieved in several ways. The model was successfully applied to the import of the project flow of the ASIL (Automotive Safety Integrity Level) project whereby a common process was developed based on the IEC-61508, IEC-62061, ISO-DIS-26262, ISO-13849, ISO-DIS-25119 and ISO-15998 safety standards covering the automotive on-highway, off-highway and machinery domain. It was also used for in-house engineering projects such as the Qualification Package for OpenComRTOS Designer and for the development of a scalable e-vehicle.

Through these Projects, the GoedelWorks metamodel has undergone further refinements and improvements, bringing it closer to the daily practice of a rigorous Engineering Process. In parallel, Altreonic has developed a novel criterion called ARRL (Assured Reliability and Resilience Level) providing a first guideline on how components or subsystems can be reused, be it in a product family or across different application domains while taken failure mode into account. The ideas behind ARRL reflect very well the philosophy behind trustworthy systems engineering and therefore we dedicated a summarising chapter to it even though ARRL is the subject of another booklet.

The name of Gödel (as in GoedelWorks) was chosen because Kurt Gödel's theorem has fundamentally altered the way mathematics and logic were approached, now almost 80 years ago. What Kurt Gödel and other great thinkers such as Heisenberg, Einstein and Wittgenstein really did was to create clarity in something that looked very complex. And while it required a lot of hard thinking on their side, it resulted in very concise and elegant theorems or formulae. One can even say that any domain or subject that still looks complex today, is really a problem domain that is not yet fully understood. We hope to achieve something similar, be it less revolutionary, for the systems engineering domain and it's always good to have intellectual beacons to provide guidance.

The Gödel Series publications are downloadable from our website. Other titles in the series cover topics of Real-Time programming, steering control and on ARRL. Copying of content is permitted provided the source is referenced. As the booklets will be updated based on feedback from our readers, feel free to contact us at goedelseries@altreonic.com.

Eric Verhulst, CEO/CTO Altreonic NV

*: pronunciation [\[ˈkʊɪt ˈɡøːdəl\]](#) ([listen](#))

| | |
|---|-----------|
| PREFACE | 3 |
| 1. Trustworthy Systems Engineering | 6 |
| 1.1. What is a system? | 6 |
| 1.2. What is a Trustworthy system? | 8 |
| 1.3. What is Systems Engineering? | 9 |
| 1.4. The subdomains of Systems Engineering | 11 |
| 1.5. What is Resilience Engineering? | 13 |
| 2. Unified Systems Engineering | 15 |
| 2.1. A Process as a System | 15 |
| 2.2. Unified Semantics | 16 |
| 2.3. Interacting Entities Semantics | 18 |
| 2.4. A unifying model for Systems Engineering | 19 |
| 2.5. Systems Engineering: different views that need to be combined | 20 |
| 2.6. An informal view on Systems Engineering with GoedelWorks | 21 |
| 2.6.1. Real engineering is Process of iterations | 23 |
| 2.6.2. Traceability and configuration management | 24 |
| 3. Engineering real systems that can fail | 25 |
| 3.1. ARRL: the Assured Reliability and Resilience Level criterion | 25 |
| 3.2. Overview of existing criteria in the domain of trustworthiness | 25 |
| 3.2.1. Safety Integrity Level | 25 |
| 3.2.2. Quality of Service Levels | 26 |
| 3.3. The ARRL criterion | 27 |
| 3.4. Is this sufficient for antifragility? | 28 |
| 3.4.1. Antifragility assumptions | 28 |
| 3.4.2. Some industries are antifragile by design | 29 |
| 3.4.3. Do we need an ARRL-6 and ARRL-7 level? | 30 |
| 3.5. Automated traffic as an antifragile ARRL-7 system | 30 |
| 3.6. Is there an ARRL-8 level? | 31 |
| 3.7. Conclusions | 32 |
| 4. The systems Grammar of GoedelWorks | 33 |
| 4.1. Systems Grammar | 33 |
| 4.2. Terminology and its conventions in GoedelWorks | 36 |
| 4.3. Process Steps, instantiated as Work Packages in a concrete Project | 38 |
| 4.4. In the end, any project is iterative | 40 |
| 4.5. Systems Engineering as a collection of Views | 41 |
| 5. Description of the GoedelWorks Environment | 43 |

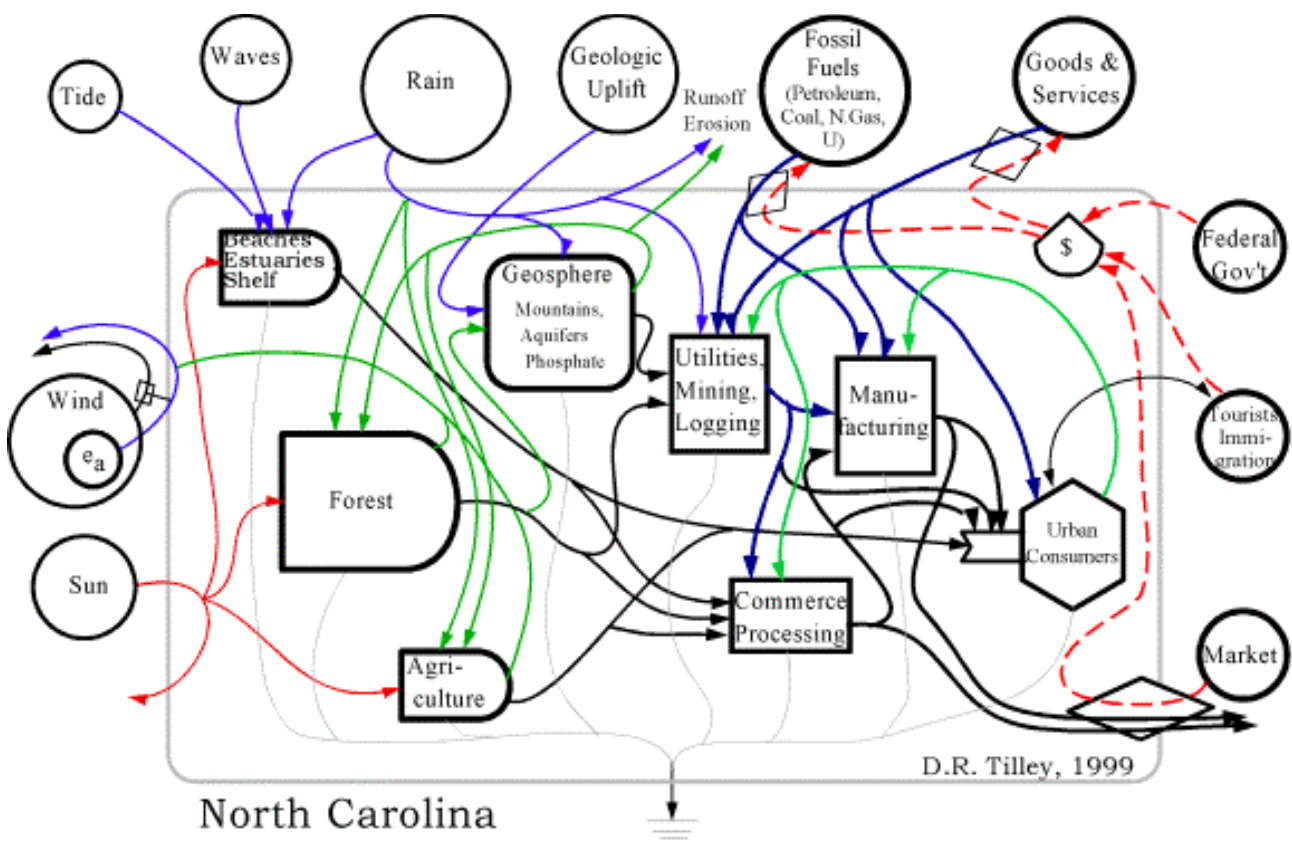
| | | |
|-----------|---|-----------|
| 5.1. | Principles of operation | 43 |
| 5.2. | Organisational functions of a GoedelWorks portal | 44 |
| 5.3. | GoedelWorks Systems Grammar | 45 |
| 5.3.1. | Top level view | 45 |
| 5.3.1. | View from inside a Work Package | 46 |
| 5.4. | Top level View in a GoedelWorks portal | 47 |
| 5.4.1. | Navigation tree view | 49 |
| 5.4.2. | The entity pane view | 52 |
| 5.4.3. | Query capability | 52 |
| 5.4.4. | GANTT chart | 54 |
| 5.4.5. | Change Log | 55 |
| 5.4.6. | Version and configuration management in GoedelWorks | 55 |
| 5.4.7. | Productivity supporting features | 57 |
| 5.4.8. | Administration | 57 |
| 5.4.9. | Glossary | 58 |
| 5.5. | A project example | 59 |
| 5.5.1. | The OpenComRTOS Qualification project | 59 |
| 5.5.2. | Planning the Qualification Project | 59 |
| 5.5.3. | The Planning Activities | 61 |
| 5.5.4. | Design Activities | 61 |
| 5.5.5. | Some statistics | 61 |
| 6. | Safety standards awareness in GoedelWorks | 62 |
| 6.1. | Safety standards for embedded reprogrammable electronics | 62 |
| 6.2. | ASIL: A safety engineering process focused around ISO-26262 | 62 |
| 6.3. | Certification, qualification after validation | 63 |
| 6.4. | Organisation specific instances of GoedelWorks | 65 |
| 7. | References | 65 |
| 8. | ANNEXES | 66 |
| 8.1. | Entities supported in GoedelWorks 3 | 66 |
| 8.2. | Entities defined in the ASIL Process | 69 |
| | Acknowledgements | 71 |

1. Trustworthy Systems Engineering

1.1. What is a system?

It might surprise you, but while the word "System" has its origins in the Greek philosophy, it is only in the last 50 years, with the rapid advance of electronics and computers that the word System became a standard technical term. The meaning is still the same, but what motivates the use of the word (vs. for example "machine") is the underlying complexity. Electronic Systems today can have thousands of components, each component being a complex System in itself. A modern state of the art Processor for example has a few billion of logical elements. Still, what we see on the outside is often only just a piece of plastic with some tiny metal bumps underneath. Such a System might be a part in a larger System such as your laptop or your car. Each of these larger Systems might again be just a component of a larger System. For example your laptop is connected to the internet System and your car is part of a transportation System. In these two examples, the smallest part might be just a few tens of nanometers large, the largest part is the world itself with its dimensions that is about 10.000 billion times larger. Therefore, let's consider the following definition:

A System is a layered and structured collection of subsystems, often called System components, that together act as a whole and provide a specific functionality.

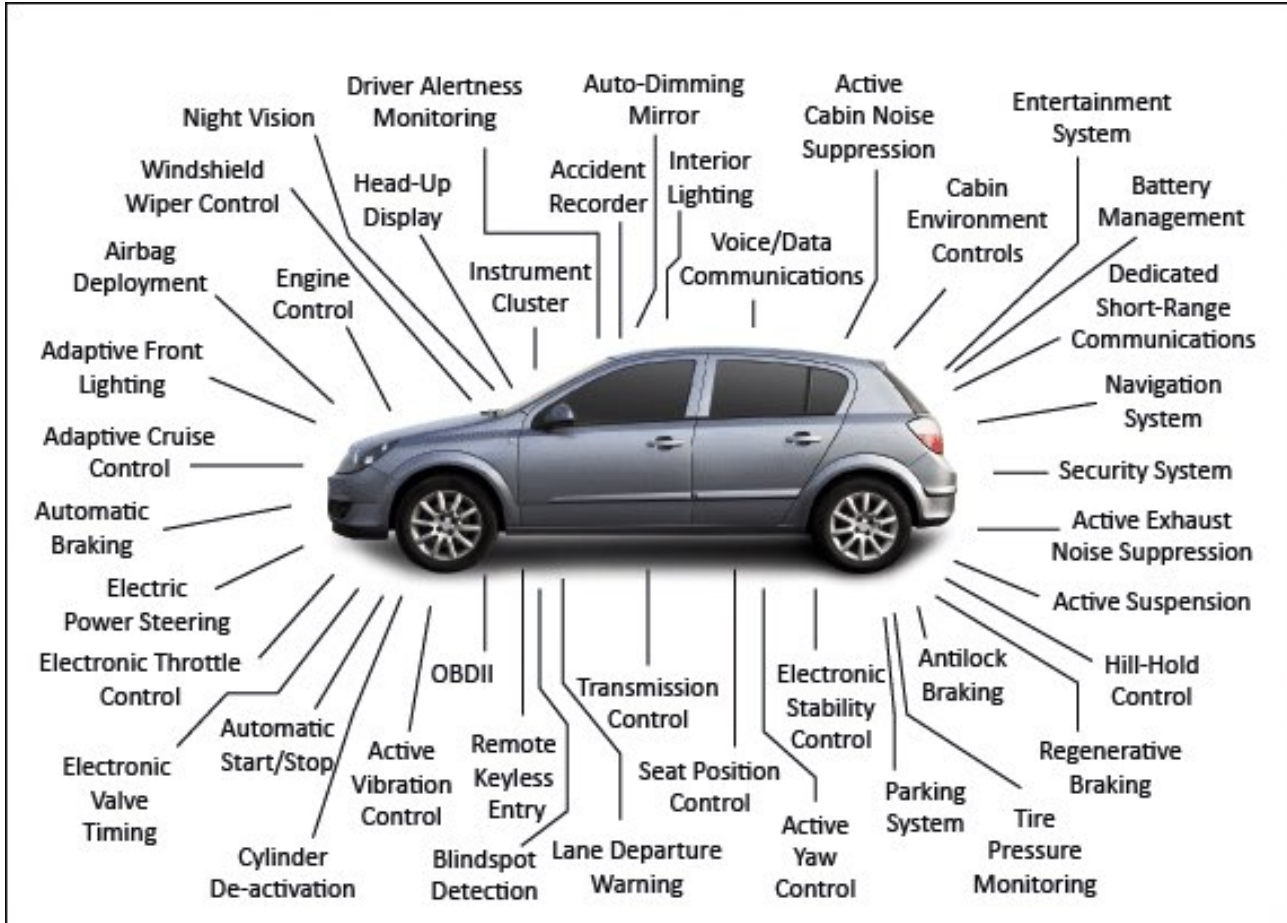


The System of North Carolina (Carolina (Src: <http://www.enst.umd.edu>))

This definition is still generic and fairly vague. In the following chapters we will make this definition more concrete (when we look more into the details), but more abstract as well (when we try to find the common properties for all Systems). Let's start by taking a look at some examples.

First of all, our natural environment is full of Systems. See for example the system of North Carolina in the picture. Every living creature, every plant is a so-called biological System. All of these, including our own human species, interact with each other somehow, in the System that we can call "life on earth". Also earth

tides). Humans are also social beings and our civilisation is full of social, economic and political Systems. Most of these have, more or less, spontaneously emerged and still evolve every day. While these natural Systems are very complex and interesting to study, they will not be our focus, even if we can certainly apply some of their mechanisms to the Systems that interest us. See for example ARRL-8 in the chapter at the end of this booklet. There is even a term called "social engineering" but this is not what we have in mind. On the contrary, this is maybe exactly what we want to avoid in real, trustworthy Systems Engineering Projects.



The many sub systems of the car as a System

The Systems that interest us are the human-made ones. Often such a System needs to be considered together with two other Systems. The first one is the environment in which the System will be used. The second is the operator interacting with the System. An example of such a System is a car. These Systems are often very complex and we could call them technological Systems. What distinguishes them from the biological Systems is that they are the fruit of our human intelligence and not of millions of years of evolution. What distinguishes them further is that these Systems often require the use of many other Systems (called tools) to make them and the use of many components and many Resources. As such, such a technological System has a long history if we trace the origins of every component. Each embodies decades, sometimes centuries of human knowledge. And while these Systems are not as complex as the natural ones, they are often the result of a concentrated effort to produce the System from just a few statements that describe its mission ("We will go to the moon and return") in a relatively short period of time.

1.2. What is a Trustworthy system?

The example given above is for the normal earthling perhaps too far away from his daily occupation, but illustrates very well the concept of Trustworthy Systems Engineering. The technology was still in its early days, there weren't that many computers yet, but a goal was defined to bring a man to the moon and back to earth safely. This set a process in motion that would produce the Saturn-V rocket, the Apollo cabin and Lunar Lander and it actually set 14 people on the moon and safely returned them. One mission even showed that the System was resilient enough to bring back astronauts when their Apollo life support System was seriously damaged by an explosion. Even if astronauts are clearly taking more risks than the average person, at the end of the day, what it came down to was that the risk was an accepted risk and no astronaut would have volunteered to step into the Apollo cabin on top of the Saturn-V rocket, if he wouldn't have had sufficient trust that he would



Apollo 13 safe return on earth

safely return. Therefore, space programs have been a great catalyser for developing the discipline of Systems Engineering. Hence, it is with pleasure that we recently witnessed the landing of Philae, part of the Rosetta mission, on a comet. As an engineering project it was simple yet very trustworthy in the context of many unknown parameters of the mission. On board was the Virtuoso RTOS, a precursor to our OpenComRTOS Designer. It made a small contribution to a large international team effort that after 10 years and 500 million kilometers away gave us the first close-up pictures and soil sample analysis of a comet.

There are of course other domains where Systems engineering developed more than in other domains. The aviation industry, the defence sector, the shipping industry, the medical sector, industrial manufacturing and railway have always had a concern for safety, especially as accidents had shown that things could go wrong and people could get killed. This has resulted in the emergence of safety standards. Table 1 lists the most prominent safety standards.

Examples of Safety and Systems engineering standards

| | |
|----------------------|---|
| IEC-61508 | Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems |
| ISO-26262 | Road vehicles -- Functional safety |
| DO-178B/C | Software Considerations in Airborne Systems and Equipment Certification |
| DO-254 | Design Assurance Guidance For Airborne Electronic Hardware |
| EN-50126 | Railway applications - The Specification and demonstration of reliability, availability, maintainability and safety |
| ECSS-E-ST-40C | ESA - Space - Software Engineering |

There are many more, dedicated to e.g. steel and concrete structures. Often these are more "normative" rather than describing a Process.

The major concern in these safety standards is to avoid that a System can harm or kill people. The point of view is one whereby the risk of this happening must be small enough and that the cause of this happening is a malfunctioning of the System, either because some parts break, or because external events (like the Titanic hitting an iceberg) cause this to happen. As explained in the previous chapter, any System is part of a larger System and in particular the environment in which it is used. A well executed Systems Engineering Process anticipates such failures and depending on their probability of occurrence, their severity and

potential consequences takes measures to keep the consequences within acceptable limits. In the less severe cases, a simple warning will be raised that repair is needed, but indicating that the System is still functional. In more severe cases, part of the functionality will be lost and the remaining parts will try to keep the System as functional as possible. In severe cases, a shutdown will be in order and all functionality will be lost. What makes safety related Systems Engineering challenging is that there is a long dependency chain, that the System might contain millions of parts, that many people will be involved and that many steps have to be taken. The challenge is to make this Process predictable and “managed”.

There are however other classes of failure causes that also must be considered. In safety most of the attention goes to physical causes. As our Systems, increasingly containing computing devices that are interconnected but also having connections to the outside world (a simple USB port is enough), are also vulnerable to deliberately injected faults. This is often done in a way to avoid detection.

We call this a **security** breach because the integrity of the System was jeopardised without any physical damage. Once the fault has been introduced and activated, often it will be indistinguishable from a physical fault and the safety related risks are the same.

Another source of risks is the human user itself and related to the way he can or is allowed to interact with the System. In this case the risks are related to many aspects. Many Systems are increasingly complex and no user is supposed to know all the details of the inner workings. Hence the interface with the System should be predictable and unambiguous. For new users, it should be intuitive. As the user, just like the environment, it is part of the larger System that controls the System, he should take the correct actions at all times. To minimise this risk, we say that the **usability** of the System must be adequate. The challenge here is that this involves familiarity, hence convention and even adequate training while the diversity in humans is large, and psychology is often not a discipline in which engineers excel.

Lately, as more and more Systems become interconnected and increasingly record our personal data, the need to protect this data is a factor as well. This in turn results in **privacy** issues but as more and more financial transactions are electronic, and our private data can be abused to cause financial or other harm. Until some years ago, this was a small risk, but as the border between embedded Systems and general purpose computing Systems (often called IT Systems) is becoming opaque, this aspect is gaining in importance.

With this view, we can understand that what really matters is that a user (whoever or whatever that is) can trust the System. We call this **Trustworthiness**. It is clear that for a System to be trustworthy, it is not sufficient to be safe. We define it as follows: A **trustworthy** System is a System that a user can trust to meet high standards of safety, security, usability and privacy.

| Trustworthy System | | | |
|----------------------------------|----------------------------------|-----------------------------------|---------------------------------|
| Safety | Security | Usability | Privacy |
| No physical fault can cause harm | No injected fault can cause harm | No interface fault can cause harm | No personal data can cause harm |

1.3. What is Systems Engineering?

While we have already used the term above, we have not elaborated on it. Systems Engineering can be defined as follows:

Systems Engineering is the collection of Activities in a Systems Engineering Project to define and develop a System that will meet all expectations.

This is a very generic definition. It applies to any human made System, whether it is a social or a technical one. We will focus our attention on those Systems that are complex enough to pose a challenge to develop them as trustworthy ones. A System can be classified as complex if it involves many activities, many

components and many decisions that need to be taken. It's not the sheer number of these that matter but most importantly how all the constituent elements interact (and that is sometimes in very indirect and subtle ways). In particular we are mostly concerned with embedded Systems, Systems whereby electronic hardware and software are used to make the System meet all its Requirements.

Ludwig Wittgenstein
“Philosophical Grammar”
65 years ago.

“A blueprint serves as a picture of the object which the workman is to make from it. ... for the builder or the engineer, the blueprint is used as an instruction or rule dictating how he should construct the building or machine. And if what he makes deviates from the blueprint, then he has erred, built incorrectly and must try again.”

“... What we may call ‘picture’ is the blueprint together with the method of its application”.

Wittgenstein defined Systems Engineering before the term even existed.

Systems engineering is however not an isolated set of activities. It is a System in itself that follows a Process with many dependencies, but in general we can distinguish 3 main activity domains:

Organisational Processes: Systems engineering can only work if it is itself embedded in an organisational environment that provides the pre-conditions for success. A simple example is recruitment and Human Resources management. A Systems engineering Project is a complex one and it requires people with the right skills. It also requires an organisational culture that favours a true engineering culture. The latter is not restricted to technical activities. An efficient organisation that is capable of good communication with potential users, planning Resources, procurement of parts and product manufacturing are equally well needed.

Supporting Processes: to execute a Systems Engineering Project, the organisation needs to have a number of technical Processes in place that are generic for all Systems engineering Projects. Their goal is to support the development and engineering to become less chaotic. A simple example is configuration management. This must be in place to avoid that engineers spend their time figuring out what changes their colleagues made to other parts of the System. It is also essential to make sure that the final System has a well known and coherent configuration.

Development Processes: these activities are the core of the work to be done. While the most gratifying ones are the development itself, true Engineering consists of a lot of other activities like defining and analysing the Requirements and Specifications, developing simulation Models, verifying

mathematically (using tools) the correctness of the Models, testing and verifying and finally integrating and validating the results.

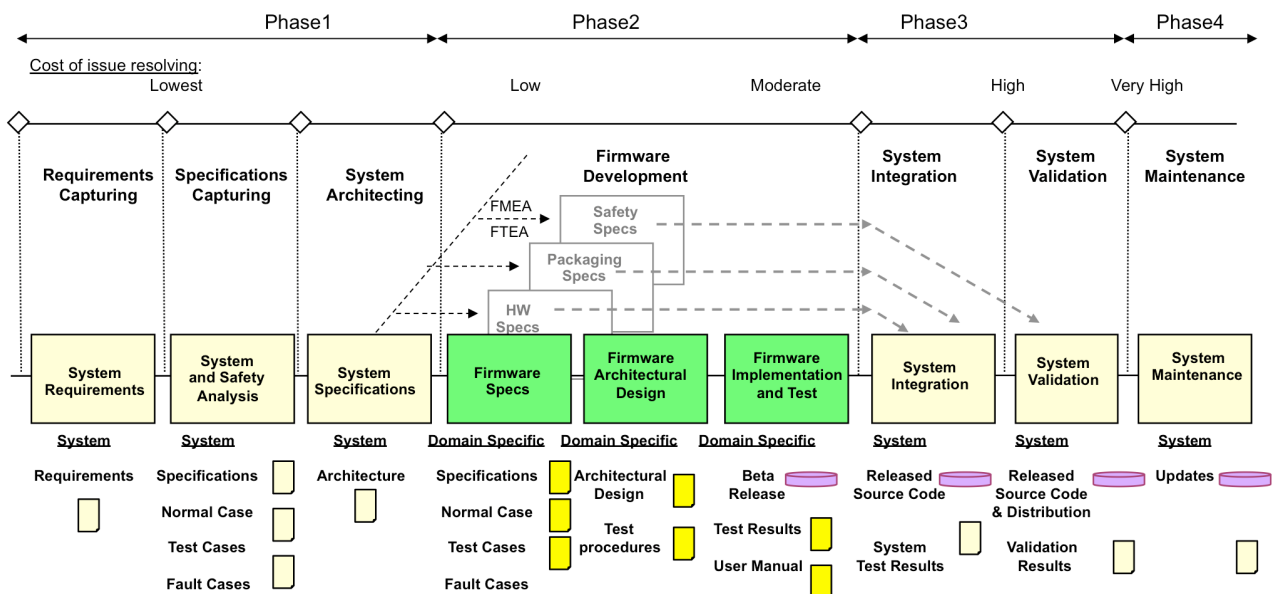
A Systems Engineering Project is seldom an isolated activity and requires a full **life-cycle view**. Activities that have an impact before the Systems Engineering Project starts are for example the formulation of goals and concepts, Requirements collection (from any stakeholder) including legal or societal Requirements. There might have been previous Projects in which sub-system parts were developed. Activities that must be taken into account after the System was developed are first of all the production of the System. Design for production is an important Requirement. Once the System enters its operational life, users must be trained and the System must be supported and maintained, maybe even upgraded or redesigned.

Finally, when the System is taken out of service, it must be disposed off. When the System is safety critical, two important Requirements will have to be met as well. The first one is **configuration management** because a System can only be safe if all its components work perfectly together. There have been airplane incidents that were caused by using a slightly different screw during maintenance. The other one is

traceability. When something fails, trust can only be restored and improved if the cause and chain of events resulting in the failure can be traced back.

1.4. The subdomains of Systems Engineering

Seen from the outside, it barely matters how a System is composed, what technologies were used and how it was put together. Different Systems can provide the same or similar functionality to its users, even if the underlying technology is drastically different. Originally radios had very few and fairly big components. By a way of speaking, one could almost see the electrons moving in the amplifier tubes. Today, one can listen to internet radio while there is no longer a traditional radio-wave receiver involved. The sound arrives in digital packets over a wire. Radios are now single chip devices using digital logic and software to provide the same function as the original coils and radio tubes.



An example of a systems engineering project split in subdomains during development

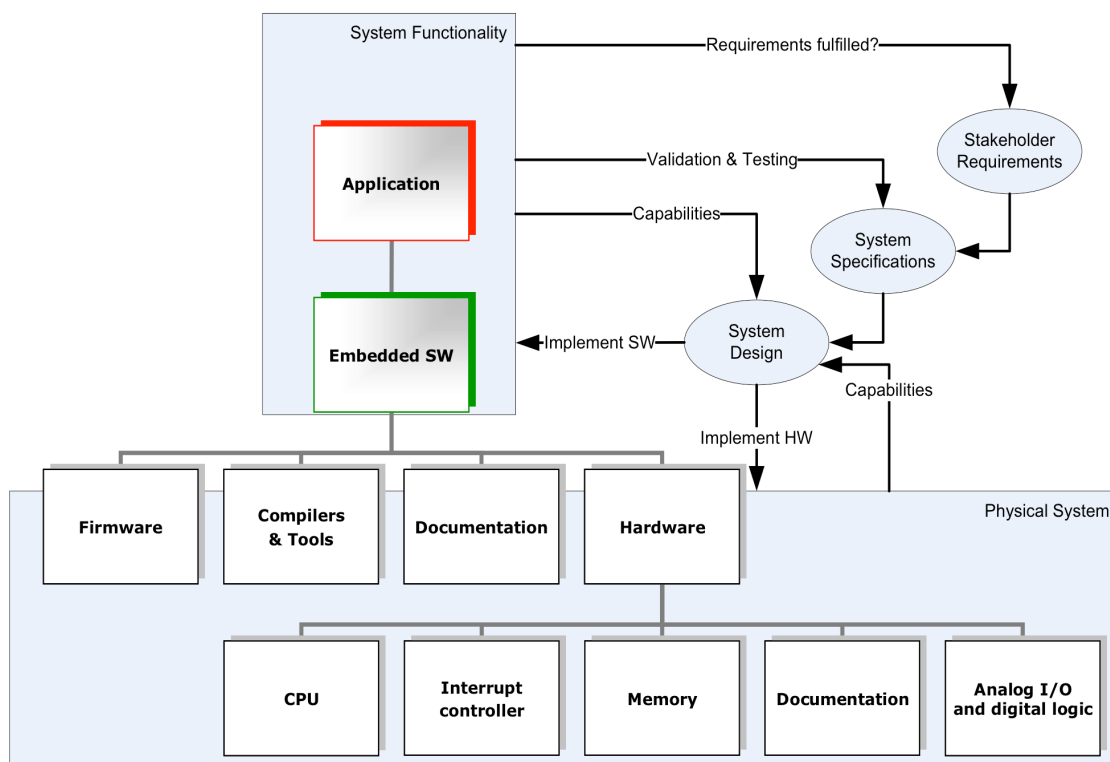
Nevertheless, once the System Requirements have been agreed upon and specified, engineers will select implementation technologies each requiring different knowledge and skills. Hence, the engineering Process will be split in **technical subdomains**, each developing their part, after which they come together again to deliver the System. The different subdomains are however not isolated. Decisions taken in one domain affect the capabilities in another domain. For example when a certain processor is selected, it will determine how much can be done in software. It will also affect the power consumption and maximum heat dissipation. A good System design is one whereby the right **trade-offs** are made to achieve the goals. The perfect solution doesn't exist because Requirements will conflict and one solution doesn't fit all.

Let's take a look at a typical embedded device or rather a System with embedded technology inside. Think about a printer, a pacemaker, a car, a house, a train, an airplane, a Mars rover, a submarine and so forth. Whether small or large, you are likely to need engineering activities in the following domains:

Mechanical and materials engineering: real-world devices are subjected to an often aggressive environment, putting stress on the embedded device. Vibrations, shocks, heat, cold, humidity, chemicals or even sunlight, the air or cosmic radiation will attack the packaging and the physical structure of the System. If this results in the integrity of the delicate electronics inside being damaged, or connections among them failing, the results can be fatal.

Power and energy engineering: all embedded systems need external energy to function. Increasingly, it is important that they use this energy efficiently and that they can cope with varying energy sources. A good power supply design will increase reliability and will decrease life-cycle costs.

Hardware engineering: this is today often the term used to designate all electronic engineering activities. Some Projects might decide to develop their own chips (ASICs), use reprogrammable ones (FPGAs) or just buy readily available processor parts (for example microcontrollers) and put them together to create a sub-system component. The hardware part of an embedded System might require specialists in many more specific domains: processor design, analog design, RF design, MEMS (tiny mechanical parts on a chip), etc. While the analog domain is often less complex, given the scale and speed of the circuits they remain a challenge. All hardware operating in the (synchronous) digital domain is clocked and the issue is that few billion elementary structures (often called gates) in the hardware share that clock. The challenge is that these billion gates operate as a huge state machine whereby all gates must remain synchronised. As the timing parameters of the gate circuits will vary with temperature and supply voltage, this is not a trivial matter. Today these issues are solved by applying large enough robustness margins during the design and by carefully controlling the production Process.



Dependencies for embedded software

Software engineering: on clocked reprogrammable processors a software program will often give the embedded device its application specific functionality. Whereas the hardware state machine is relatively static, a software state machine is dynamic and its behaviour can be data dependent. Contrary to hardware however, a software state machine has no properties that vary with external influences, hence it is (theoretically at least) possible to fully verify and predict the behaviour of software in detail. Software has no bugs, only errors.

There are undoubtedly more engineering disciplines that come into play. Human interface engineering is often a neglected one. Today, Production Engineering is more and more part of the Development Activities. Within the different engineering subdomains one can find more specialised subdomains. Algorithmic experts will develop the right algorithm to process the data, hydraulic engineers will add their bit, control

engineers will design the control loops to keep the System stable, etc. But one thing is certain, they all must work together to achieve trustworthy Systems Engineering.

In addition, there is a long **dependency chain** between the domains. For example, when developing embedded application software, the software engineer will need to do more than just write his small part. He will link third party libraries, he will use a compiler and linker in the assumption that they are error-free. Even when that is the case, the correct behaviour will depend on the correctness of the documentation (of the hardware as well as the software) and on the hardware being defect-free. Often, this will not be the case and then he has to find a work-around so that the application itself is still behaving as specified. The example, illustrated in Figure 6 illustrates how a good engineer must know about the other domains he is working in to achieve good results.

1.5. What is Resilience Engineering?

After we have reached beyond safety and went for trustworthiness, there is a further step worth pursuing, partly because it is becoming inevitable. As Systems become larger and more complex, it is no longer enough to design them for trustworthiness when they operate normally. We must think in terms of providing maximum required functionality, often called **Quality of Service**, taking into account that the System will not always have all its Resources available. This is more in line with the way biological Systems work: an ant colony remains an ant colony even if a fire wipes out part of the ant population.

This is not always the case in safety engineering practices: when a failure is detected - and this can be one of its million tiny components, the System will either shutdown or remain operational in an often severely degraded mode, or either coarse grain redundancy will be used to keep the functionality at its original level, but a next failure is then often catastrophic.

Resilience engineering works differently. It will look for architectures that can tolerate partial failures. It will not seek to maintain full functionality but will seek to maintain the best possible functionality with the Resources available. Resilient Systems have often architectures that are very modular by design and have a redundant but distributed capacity.



The South Pole Station Dome (being deconstructed)

A simple example is a meshed roof construction. The roof will not cave in because some rods are lost. Networked Systems (think about the internet or a mobile communication System) often also work this way. In the control engineering domain this was first introduced to keep Systems stable even if a major failure occurs. Often this requires running a simulation Model in the control loop. Examples are flight control Systems that control the plane using only the engines when for example parts of the wings are damaged. In general a resilient system will be designed to be **fault tolerant** in all cases. If a failure is detected it will not simply go into a so-called safe mode (like limiting the engine to 1000 rpm and flashing a warning), it will retain all functionality but with less redundancy should a subsequent failure occur. This approach is often avoided for reasons of cost. This is true if the measures are applied on an existing architecture. A well chosen architecture is however resilient by design and then the life cycle cost can even be lower.

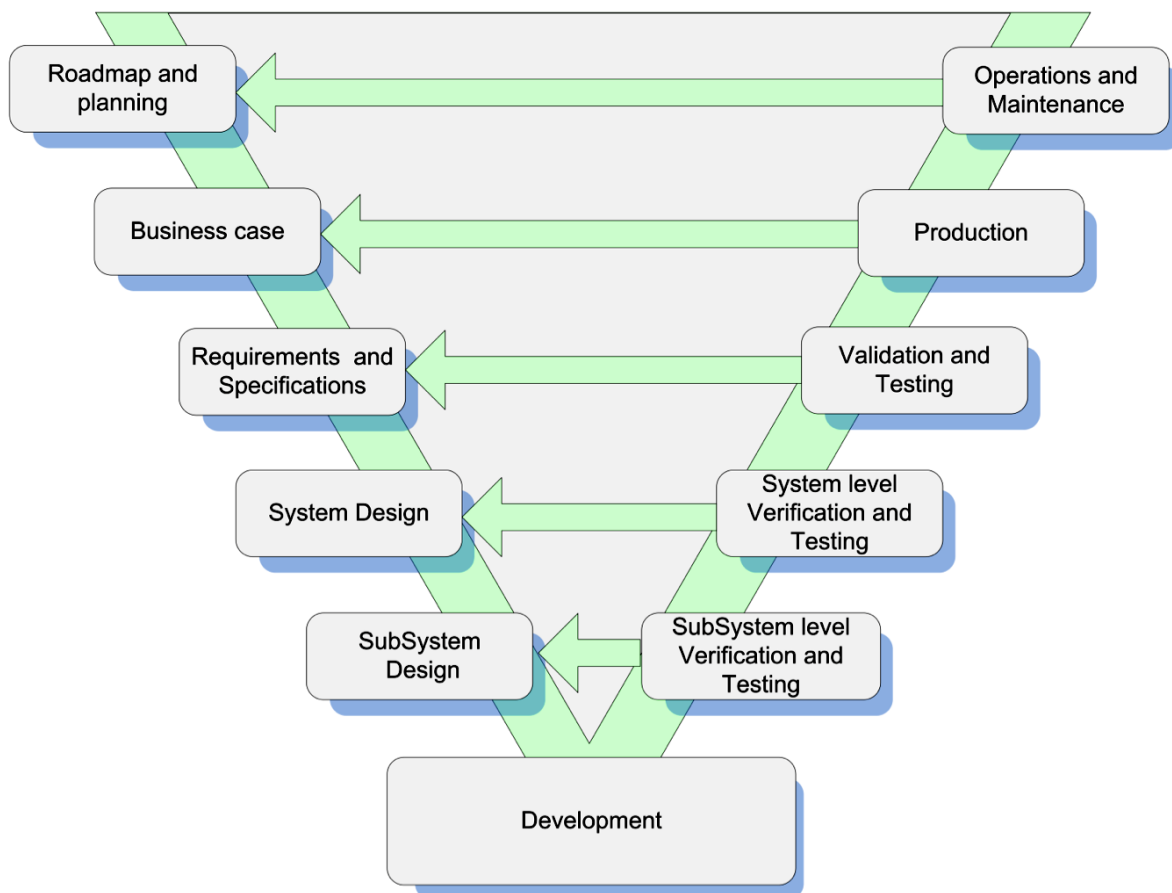
An important difference with more classical approaches is that a resilient System, besides having a more resilient architecture, gracefully adapts itself to a change in the available Resources. It will not fail immediately, but e.g. recover from the errors, maybe even repair itself, rearrange Resources so that it remains "alive". After all, this is the ultimate goal of any System. Recently, systems that adapt and even become better after they experience issues (a general term for anything that could or went wrong) have been called anti-fragile, to indicate that there is a step beyond resilience. We will explore this deeper in the chapter on the **ARRL (Assured Reliability and Resilience Level)** criterion.

2. Unified Systems Engineering

In this chapter we outline a generic, unifying approach to systems engineering. Based on the premise that engineering is essentially domain independent but that each domain applies the same type of Process Steps in a domain specific way.

2.1. A Process as a System

We have until now been speaking about Systems in terms of something physical that has to be developed. It answers the question of "**what**" is it that we need to develop. In Systems engineering, it is however equally important "**how**" it is developed. Therefore, a Systems engineering Project has two main types of components: the **Process** that is followed and the development **Project** of the System itself.



A simplified iterative V-process model

Let's take as an example the development of a piece of software. It includes activities of writing the source code statements, compiling them, running the resulting executable and testing it. This will iteratively result in a working program. But depending on the skills of the software programmer, this can take less or more time to get it right. A skilled programmer will follow a specific set of steps that will lead him faster and more predictably to a reliable piece of software. In essence, a skilled programmer has developed his own "Process" to deliver a better job. What Systems engineering does is to take the best of practices and to define them as a Process that should be followed by an organisation or Project team. Standards define these practices as recommendations, although **certification** might impose rather strict obligations. In any case, any organisation will have its own history and heuristic practices and they must be integrated with what standards might impose. For example, the software engineering Processes will define that the

software tools like the compiler must be qualified first, an adequate version management System needs to be used, coding rules need to be obeyed, peer review need to be in place, etc. The latter aspect also hints at a third component that defines the Project: the **Work Plan**. It defines **when** the different Activities should be executed using specified Resources.

Another example is testing of hardware. Testing will verify that the System and its component meet the Specifications. This is only possible if the testing happens in controlled circumstances. For example, the Specifications must be stable and approved, the test set-up must be well defined and repeatable and the test procedure must be defined. The testing will produce a deliverable as well, i.e. the test report. The way to see this is that the testing Process is like a small Project that requires specific Resources and produces a specific product, the test report. Hence, one can see that a Process, in casu a Systems Engineering Process is also a System that needs to be developed first. The sub-system components can include humans, equipment or simply all that is needed to produce a report, but it remains an Activity that must be trustworthy.

One of the first processes to be developed (and imposed) was the so-called **waterfall model**. It imposes a linear process flow whereby requirements are developed, then fed into development after which the system is tested and validated to see if it meets the original requirements. While correct in principle, such a rigid process rapidly breaks down because it assumes that the requirements are complete and perfect. Therefore the waterfall model was made more iterative by introducing feedback and verification steps early. This became to be known as the **V-model**. This is still one of the best models as it equally applies for very small systems and with very small steps. Such a process is then often called **agile or iterative**. In reality any process can be iterative if not only the order of the steps is considered but also the state of the different process entities. This will be explored further when discussing the GoedelWorks dependency relationships.

2.2. Unified Semantics

As we have seen in the previous sections, Systems Engineering touches many domains. We focused on the technical ones, but there are more. Systems Engineering starts with formulating a goal and that involves management, political, societal, financial and many more people. The first question to ask is whether they all speak about the same System. Even if they do, their perspective can be vastly different. The second question is whether they speak the same language. They might use different terms to talk about the same thing or they might use the same term to designate a different thing.

“... I don't want to get bogged down in semantics causing problems”.

Pervez Musharraf

Even in the technical domains this is very often the case. This is because **terms and words have a context** and a context has history. When a new domain emerges, people often borrow terms from another field and select it based on analogies. Or the word might have its origin in a different natural language and is then erroneously imported. In the technical domain, people often will use acronyms, typically not understood by newbies in the field but also it might happen that the original

meaning of the acronym was lost in time.

This is an essential observation for Systems Engineering. As we have seen, the scale and complexity of Systems Engineering is very wide and it is obvious that it involves a lot of communication between people coming from different domains. If they don't understand each other, how can they then develop the right System and do it right? Let's take a simple example, the word 'scheduling'. For the production engineer, this means the order in which a product is produced on a production line. For the electronic hardware engineer, this means mapping his signals correctly in the clock domain. For the software engineer, it means defining the order of execution of software processes. Moreover, people also develop a sense for orders of magnitude. For the production engineer, time might be measured in minutes. For the software engineer, it

is likely microseconds or milliseconds. For the hardware engineer, it is more likely nano- or picoseconds. When each of them says 'this is fast', they are most certainly thinking about very different time intervals.

Process Diagram Business Process
Management Business Process
Mapping Business Process
Modeling Communication
Process Diagram
Communication Process Model
Decision Making Process Design
Process Document Management Document
Management Open Source Elements Of
Communication Process Enterprise
Content Management Hiring Process
Interview Process Manufacturing
Process Marketing Strategy Process
Microsoft Document Management Pdf Document
Management Performance Appraisal
Process Performance
Management Process
Performance Review Process
Process Flow Chart Process
Flow Chart Examples Process Flow
Chart Excel Process Flow Chart
Template Process Flow
Diagram Process Mapping
Examples Procurement Process
Product Design Process
Product Development Process
Project Planning Process Sewage
Treatment Process Six Sigma

The same applies when we dig into the details of technical Specifications. Consider for example communication protocols. Some of them are described requiring 1000's of pages. How sure are we that two devices allegedly speaking the same protocol will never have differences in their protocol implementation? A single tiny difference and the communication can hang. Another example are processor instruction sets. Most processors, even within the same family, will have subtle differences and will use different terms. And when we look in the world of software, we see an explosion of terms, interface functions, all vaguely describing the same things but with obnoxious differences and side-effects when using them.

What above examples illustrate is that in Systems Engineering it is not sufficient to define terms and functionalities, one must also define their associated **behaviour**. This means that when terms and language are used one must also define their **semantics** in their context. Moreover, the semantics should be the same everywhere. We call this "**unified semantics**" and it results in two main benefits:

Firstly, unified semantics means that the same term is used in a unique way with a unique meaning. An important consequence is that overlapping semantics must be avoided, each term should describe a well defined and unique behaviour or property of the System. This is linked with another important property that is beneficial in any good system architecture: **orthogonality**. Choosing the right, orthogonal terms will often help in finding the right orthogonal architecture.

Secondly, specifying terms and concepts in a correct way is an important first step in Systems engineering. The reason being is that it is the first step in **formalization**. One can compare Systems Engineering with the activities of writing a novel composed of

sentences. If we don't want to have gibberish at the end, we must agree on the meaning of the terms and we must agree on grammatical rules on how to construct valid sentences.

The meaning of the terms and the Grammar rules do not define the sentences themselves. They define a framework that is more abstract than the sentences we will formulate. This is often called a meta-Model.

We often use the term **System Grammar** in the context of Systems Engineering. To reduce the confusion, in the remainder of this booklet we will often use an upper case letter to designate a term that has to be understood in the specific meaning of the system grammar detailed further on.

2.3. Interacting Entities Semantics

In the previous section we mainly talked about behaviour and properties of a System and why it is important that we describe these in a unique way. The behaviour and the properties of a System are however what we sometimes call '**emerging properties**'. While a System can be composed of tens or even billions of composing parts, none of them will result in the observed behaviour on its own. It is all the components working together 'in concert' that are responsible for the behaviour. At a more abstract level we use the terms '**Entities**' and '**Interactions**'. We can then describe the structural properties of a System as '**Interacting Entities**'. How they fit together we call that the '**architecture**'. Note that this is at an abstract level. In a Process for example Entities can be humans that take a set of written Requirements (another set of Entities) and transform them into written Specifications (another set of Entities). The Interactions are 'reading', 'writing' and likely also meetings during which the Requirements and Specifications are discussed.

*Wikipedia defines **system** as follows:
(from Latin *systema*) “whole compounded of several parts or members. It is a set of interacting or interdependent entities forming an integrated whole.”*

System characteristics include:

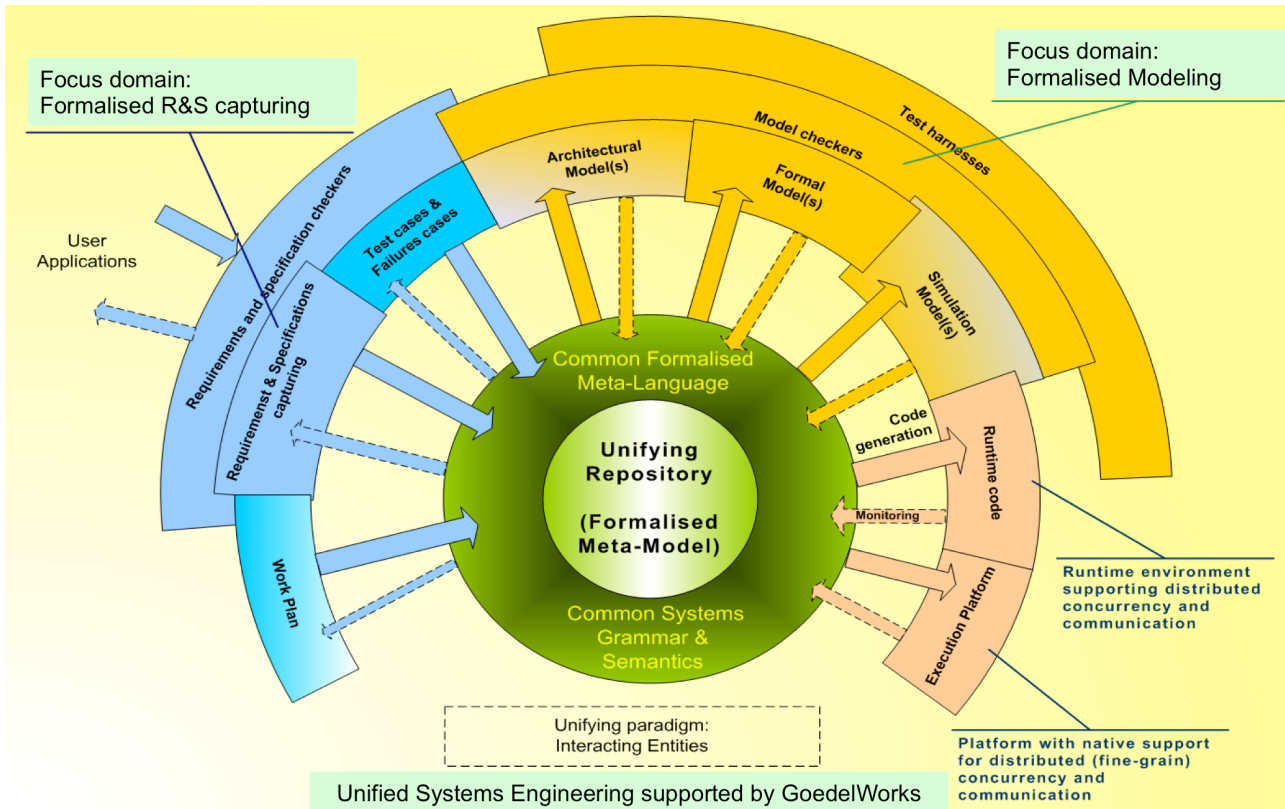
- 1. structure,*
- 2. a set of behavioural norms,*
- 3. interconnectivity and*
- 4. units that function independently within the system*

In the technical domain, Entities and Interactions can often be identified in a more concrete way. An Entity will be a physical component (a sub-system) and an Interaction will be a concrete exchange of information, obeying much stricter protocols than human language allows. Note that the Interaction can also involve transfer of materials or energy (e.g. a hydraulic pressure). This view has a number of benefits. First of all it neatly expresses how a System is composed of smaller Systems, etc. Interactions on the other hand allow us to use a component without needing to know all its internal details. It is sufficient to know how the Interaction is defined. Interactions and protocols allow us to hide the inner state of a composing Entity, even the way it is implemented, without jeopardising the way the System will behave.

Referring to the notion of unified semantics in the previous section, one can see that there is a benefit to keep this view of a System as a set of Entities and Interactions consistent in all phases of the Systems Engineering Process. Concretely, it pays off to map Requirements and Specifications fairly orthogonally to well defined Entities and Interactions. As we will see further, these are the components of various types of "**Models**", used e.g. for

simulation, formal verification, architecting, etc. Hence, it will be beneficial that all these Models have similar semantics. For the implementation this means that we should be targeting a concurrent, event-driven programming model and that even the hardware should facilitate this kind of programming model. Simulation models are for example often based on large system loops (convenient for simulation) but this often means that such simulation code is not easily translated into a concurrent, or even distributed program. This also makes such code less portable. The same applies for Formal Models that often rely on the notion of a global state machine. Again that might be convenient for verification, but not what the software needs.

2.4. A unifying model for Systems Engineering



A unifying view on systems engineering

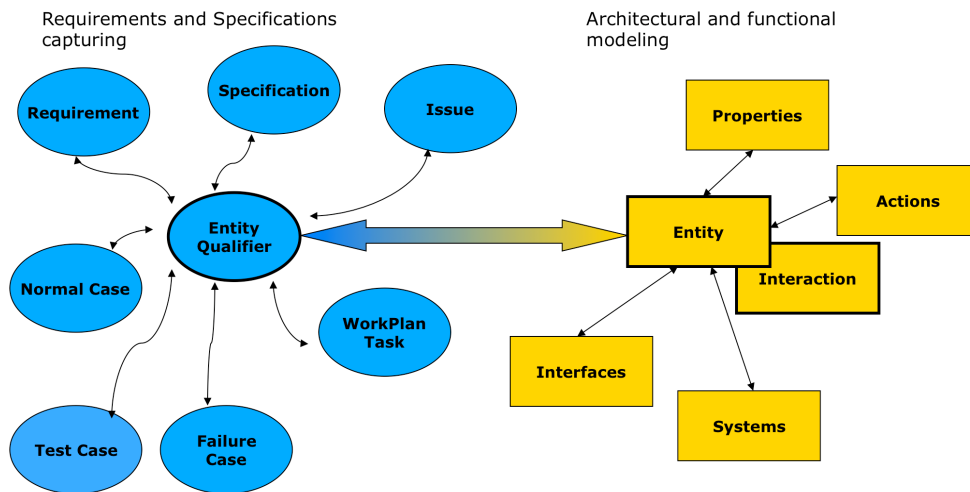
Reading safety engineering or Systems Engineering standards can be a daunting Task, certainly the first time. There are several reasons for this. First of all, these standards came gradually into being often driven by an industrial or societal need. Most of the time, it is the work of a committee, stretched over many years. In addition few scientific work has been done on the subject, although much work has been done on specialised subdomains. For this reason current standards are often heuristic in nature, albeit newer releases of the standards (e.g. ISO-26262) have clearly benefited from user feedback. It should be noted that there are two classes of standards. European standards are more prescriptive and normative, whereas US standards are more goal oriented, leaving it up to the user to prove that they have done everything necessary to reach the goals. Examples are e.g. DO-178B/C. These standards also follow more the philosophy of the CMMI maturity Model, whereby quality of the organisational Processes (of which Engineering is one) is seen as a result of the "maturity" and capability of the organisation and less the result of following a prescribed Process.

To get a clearer picture on Systems engineering, we have analysed different Systems Engineering Processes as described in the various standards and publications and tried to develop a generic meta-Model for it that can be applied to almost any engineering Project. We define the concepts at a generic level. Together they create the meta-Model of Systems engineering. Domain specific concepts can further be derived from these generic concepts.

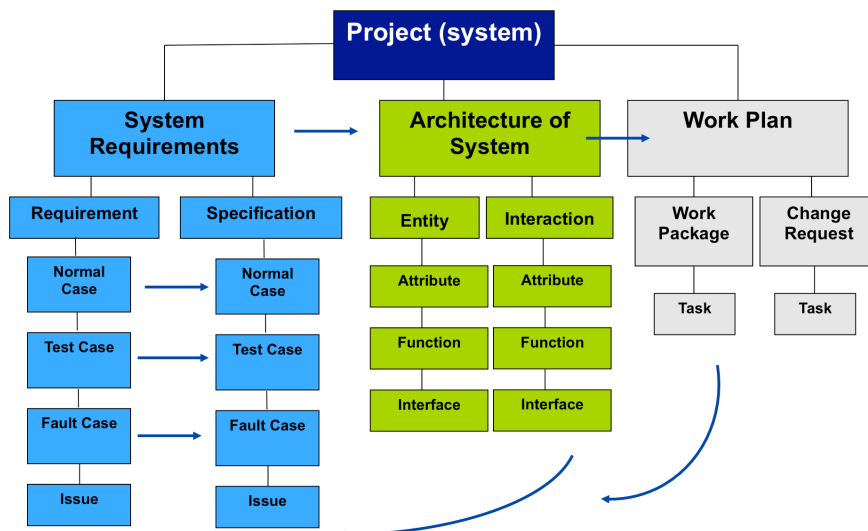
This list is certainly not complete but new terms and concepts should be refinements of these generic terms. Refinements can be driven by a specific domain or Process or by a further decomposition and definition of attributes. A typical example are Specifications that are refined into Functional Specifications such as Interface Specifications, Implementation Specifications and Test Specifications and non-functional Specifications that often are related to properties of the system like power consumption, maintainability, etc. By themselves these concepts do not define a Systems engineering Process or Project (as we will see further this distinction is for practical reasons). In the next sections we will link these concepts and define their possible state attributes.

2.5. Systems Engineering: different views that need to be combined

A major issue in systems engineering, and that applies also for many of its subdomains, is that engineering a system or product requires the convergence of different wide ranging views that each look from a different perspective. This partly explains why one finds that the terminology is not always consistent and why no standard or reference text provides coverage of all aspects. It also explains why one finds multiple complementary engineering approaches indicated with terms like “requirements driven engineering”, “test driven engineering”, “model driven engineering”, etc. In reality, one needs all of them.



The mental mapping from the intentional domain to the implementation domain



A simplified view of a systems engineering Project

In addition, a lot of the engineering activities really happen in people’s mind. Engineering is really the discipline that allows us to transform initially abstract ideas and concepts into real tangible objects. This transformation is really like a mathematical mapping from the abstract into the concrete domain. As such, this is reflected in Wittgenstein’s definition but also in the main activities one can distinguish in an engineering project. The previous two diagrams illustrate this. Requirements and Specifications capturing is about properties and statements about the system and its components. In the modelling side of

engineering we select entities and define how they interact so that they fulfil the required properties. This mapping phase can be seen as what really happens when we develop the system. Similarly, when executing the project, then the different Activities need to be planned in time. We call this the Work Planning, traditional the domain of "Project Planning".

2.6. An informal view on Systems Engineering with GoedelWorks

So, how do we go about "Engineering" a System? The first activity to perform is to figure out what the System should be able to do. This is often called **Requirements and Specifications capturing**. Requirements will be collected from many sources, often called the stakeholders. Given that there are many sources, a first challenge will be to make sure that the Requirements are well understood. Semantics come into play but also correctness. Words can be very vague and have multiple meanings. The first stakeholder is the potential user. What does he expect the System to deliver? Other stakeholders can be people with different interests: financial, political, technical, production related, etc. Often these Requirements will act like boundary conditions.

In the GoedelWorks metamodel we strictly distinguish between Requirements and Specifications. **Requirements** are often qualitative in nature and the collected Requirements will often not form a coherent set. There will be conflicting Requirements, nice-to-have Requirements, must-have Requirements as well as Requirements that are not even related to the System to be developed. Often, Requirements will not be a sufficient base for the engineering activities to start. Therefore, Requirements must be quantified and translated into concrete **Specifications**. The Requirement might say "the car will be a better one than the competitor's model". The Specification will give concrete numbers like speed, acceleration, fuel consumption, capability, etc. Specifications cover multiple domains. Most Specifications will only cover the functionality in normal operating conditions. Other Specifications will cover issues of testing. Important but not always trivial to obtain are the Specifications that cover the cases when things are not normal, like component failures or externally introduced failures or even damages to the System.

Once Specifications have been determined, engineers can start looking for possible implementations. We call this the **Modelling activities** because the Process is iterative and multiple Models will need to be developed. **Simulation Models**, including software based virtual prototypes, can be used to find the best possible implementation, but they play an important role in verifying the soundness of the Requirements and Specifications. It answers questions like "is this really what we want?" and "what-if- questions". **Formal Models** can be used to verify and prove that the implementation will be safe, or at least that some essential properties can be guaranteed. These two types of Models can often be developed in very different ways and using very different tools than the **implementation Models**. It can be beneficial that the implementation Model is obtained from a simulation Model (as it avoids a translation step) but not always (because it also limits the conceptual views on the System). Many people might not consider the final implementation as a Model, but the reality is that many implementations can meet the Specifications. Implementation Models are really **architectural Models** as they define how the System is structured. This is where the Entities and Interactions come into play. They are the execution seat of the Specifications and the way they are structured will result in the System's behaviour and properties as specified.

The above paragraphs were mainly concerned with defining and implementing the right System (the "what"). Engineering however is also about the "how" to get there and how to do that in the right way. This is related to the **Process** that needs to be followed with a Process being composed of a number of **Steps** that should be followed. In a concrete Project, these Steps become the **Works Packages**, each having their own specific **Work Plan**. A concrete Work Plan defines how the resources are to be used and when. In a trustworthy Systems Engineering Project, this entails more than developing the System and its components. As we have seen above, part of the work is to collect the Requirements and Specifications and to make sure they are complete and correct. Regrettably, this effort is not always explicitly planned for whereas research has shown that incorrect Requirements and Specifications are the most prominent cause for Projects failing

Packages. These are the core activities of each Project. They require **Resources**, Specifications as input and consist of sub-Work Packages and a number of distinctive Activities, often designated as **Tasks**.

The work starts in parallel to the Requirements and Specifications Work Package to collect background information and relevant References. A good Project is not developed in the void and should avoid reinventing the wheel. In addition, for certification purposes the relevant standards should be made available, databooks collected, etc. Organisation specific rules, procedures and Resources must also be available. We call such information **References**. These are strictly speaking not part of the Project, but can be very valuable sources of information.

A concrete Systems engineering Project will start with Work Package activities that verify that the **organisation** itself is capable of executing and supporting the Project. Is the right Systems Engineering culture in place? Is there a trustworthy, certified quality System? How is Human Resources planning organised? Not all these questions need to have a fully affirmative answer to allow Systems Engineering. Small and lean organisations can produce trustworthy products as well.

In parallel the **supportive Processes** need to be verified. Version management, configuration management, test capabilities, documentation, procurement including qualification procedures for acquiring external components, software tools, or subcontracting, etc. should be in place before the Project starts.

The major work is the development itself. The first work to be done is to carefully analyse the **Requirements** and the general context in which the System will be used. From these a coherent and complete set of **Specifications** must be derived from the Requirements. Specifications are related to the use of the System and the properties it must have, but already at this stage, three groups of Specifications must be completed. The most obvious ones are the "**normal case**" ones. These are related to the use of the System when everything is operational and the System is used as intended. A second class is related to testing, called the "**test cases**". These must be specified because testability will impact on the design. For example test points will draw extra current or state variables and parameters need to be logged, requiring extra memory.

The third group is the most difficult one. These are related to when faults or malfunctions occur. We call these the "**fault cases**". Finding these requires a careful analysis, often called de **HARA (Hazard And Risk Analysis)** in the context of safety engineering, but this equally applies to security related faults. This step will try to develop the general hazard and fault Models for the System (called Safety or Security Cases), resulting in Specifications for the System to mitigate or even annihilate the effects of the hazard and faults. In principle this step has to be done independently from the implementation while these Specifications must be available before any development or architecture is defined. This is because they can have an important impact on the implementation and its architecture, especially if the consequences of failures or security breaches are severe. Once an implementation has been decided upon, we also need to investigate the effects of faults in any of the components. This activity is often called a **FMEA (Failure Mode Effect Analysis)** and is complementary to the HARA described above. Such analyses lead to pre-conditions in terms of the quality and reliability of the components and the production process (to reduce the risks up front) but are needed to assure and assess the required behaviour when hazards and faults occur. The careful reader will have noticed that above analysis reflects the ARRL criterion we discussed in the previous chapter.

Once development can start, each Work Package will produce one or more so-called **Work Products**. They are well defined deliverables and implement the Specifications. We can divide these Works Products in two large groups. The first group contains the physical resulting **Items** of the development done and together they constitute the **System** being developed. The other set of outputs are related to the Process requirements. They carry the contract that comes with the WorkProducts in terms of documented evidence that the Process was followed. We call these **Artefacts**.

For this to happen the Work Package will need **Resources** (people, equipment, templates,...) and each Work Package will be composed of a generic set of Tasks. These can be grouped in the following seven Activities, each consisting of smaller phases.

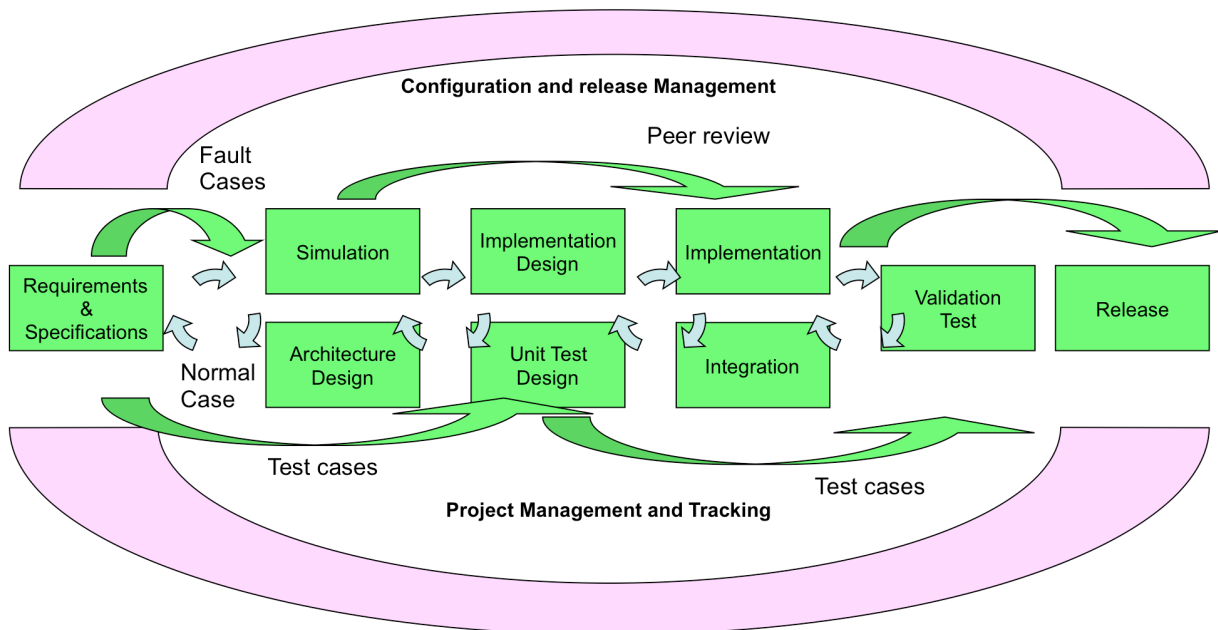
be executed in an **iterative and incremental** way (sometimes called an agile process although the term agile is not free from religious factionism). This provides early feedback allowing finding issues early on. The metamodel however expresses completeness and defines dependency relationships. The Engineering Process executed in a Project is very much one of **Refinement and Decomposition** whereby we gradually come closer to a final realisation.

At every moment decisions are taken and at every moment these decisions must be confirmed to be the right ones. Hence the metamodel does not express an absolute order in time for the Activities but a partial order in time of confirmed decisions. The Refinement and Decomposition Process however creates a **dependency graph** and hence imposes a strict order in which the different Project Entities can be approved. Similarly, a strict control of the configuration is needed as well. Whenever a constituent Entity in the dependency chain is changed, it might violate one of the dependency relationships or introduce unwanted side-effects. Hence, a Project configuration is not just the collected entities at a given point in time, but a complete graph. Before the last entity has been approved, this graph might not be coherent. For example, it might have missing entity nodes, missing dependency or decomposition links and entities that are in the process of being developed.

Hence strict configuration management is a must. Note that it applies as well to the composing Entities of a system as well as to how these Entities interact. It also applies to the composing Activities in a Work Package. If any of these is changed after being approved, then all depending Activities and Entities need to be reviewed before they can be approved again.

GoedelWorks allows to generate automatically the dependency and precedence graph for any Project Entity and hence also makes it easy to use these graphs for executing an impact analysis when for example a Requirement or Specification was changed or when a change is considered. The same feature makes it easy to e.g. verify the completeness and coherency of a Project. For example, if a Specification has no dependent Test Activity, then we know that the system cannot be approved as being in its final configuration.

2.6.1. Real engineering is Process of iterations



The Iterative nature of a project is more pronounced in the beginning

As one can see, the Systems Engineering Process has wide ramifications. It starts early and it ends long after the System was put to operational use. It is important also to see that the System is actually "defined" by

the Project that developed it as well as by the Process that was used. As we have seen, a Process is also a System and it requires the same steps for developing it in a Project. To illustrate this, consider the testing activities. They will follow a test plan, developed according to a template prescribed by the test Process. Developing a test plan, even as a template, requires the same steps as we described above (Requirements, Specifications, Development, Verification, Testing, Validation, ...). The Work Product is a test plan template, that is subsequently is used as a Resource for a Work Package whereby the test plan is a template and filled in specifically for the System or System component to be tested. This instance of the test plan then becomes a Resource for the real Test Tasks to be done on the Work Product in Development. Similarly, when a System uses components, procured from an external or internal party, this becomes a Resource for the development whereas the component was first developed in a previous Project. Therefore one can see that Systems engineering is not an isolated activity. In the bigger scheme of things, each Project will have its place in a culture and environment with a history and itself will give direction to future Projects.

Another aspect is that Systems engineering as described above seems to follow a sequence, a Flow from beginning to end. Viewed from a distance this is true. A good Engineering Project will know that for example Requirements are never really "final". Feedback from Development, Testing, etc. will detect weaknesses in the Requirements and Specifications and hence the latter might need to be adjusted. Once Development is done and Validation was approved, most likely the Systems will not exactly meet all Specifications due to small variations introduced during Development, uncertainties on design parameters and System components acquired from elsewhere. Determining the final values is called characterization (because it specifies a specific instance of the System that was developed).

2.6.2. Traceability and configuration management

If nothing is final and if the order of executing the different Steps and Activities is iterative, how can we then arrive at a System that can be validated? The way to reach this goal is **state configuration management**. As the attentive reader will have seen, the Flow imposes a dependency chain. Actually multiple ones, one for each Work Product back to the original Specifications and Requirements and then additional ones for Integration and Validation. The final System can only be approved (released for production or deployment) if all preceding Activities, intermediate Processes and Project Entities had been approved before. This is necessary for two main reasons. First of all, the configuration of the System must be consistent to allow Validation and Certification. Else, we would have many uncertainties. But, secondly it will save a lot of effort and Resources. For example, if Testing is done before Verification is done, it is very likely that Testing will not just find functional errors, but simple errors due to some of the Development not being done according to the specified Process (simple example: a wrong name was used for a variable).

The most important state transition is when a Project Entity goes from e.g. "in work" to "approved". At that moment, its configuration must be frozen. But because there is a precedence chain, it creates a partial order of the approval steps to be taken for developing the Work Product. This is the key to allow concurrent engineering on the different Work Products (Process artefacts, System components). One can work on anything in parallel, but approving entities can only be done in the order specified in the dependency graph. It also means that the architecture of System will allow more concurrent engineering and development when it is itself decomposed in concurrent Entities with well defined Interactions. This is a key observation for what is called today "**evolutionary**" validation and certification. Today many Projects are related to families of products and often changing one part will create a new member of the product family. If the architecture is not sufficiently modular and concurrent, then the whole System must be re-certified (re-verified, re-tested, re-integrated and re-validated). With a concurrent architecture, this work can be reduced, although never fully eliminated.

3. Engineering real systems that can fail

The attentive reader will have noticed that engineering a trustworthy system is much more than an error and trial process. While experience is a valuable asset, following a systematic approach can help in reducing the residual errors and hence will increase the trust one can have in the system. As such, this remains vague on what this means in the context of situations whereby faults or hazards occur and the trust in the system performing as intended can be lost. What we noticed is that while safety and engineering standards emphasise the process to be followed, they are often specific for a given system. Very little is said about how this applies to other systems, even if they reuse many of the same components. The result was the elaboration of the ARRL (Assured Reliability and Resilience Level) criterion.

The Assured Reliability and Resilience level criterion is introduced to be able to classify components and systems according to the way they can be trusted, especially in the context of fault behaviour. This chapter can be read independently from the other chapters but it gives a mental framework that helps to come to a systematic way of engineering trustworthy systems and products.

3.1. ARRL: the Assured Reliability and Resilience Level criterion

Systems engineering aims at developing systems that meet the requirements and constraints of its stakeholders. Increasingly systems must not only provide their intended functionality, but it must also be guaranteed in a **certifiable** way that such systems remain safe (and secure) when subjected to faults or hazardous situations.

From the safety point of view, the lower the required residual risks should be, the higher the safety related requirements, often expressed as **SIL (Safety Integrity Levels)**. The same applies for the subsystems whose faults can induce a safety risk.

We have argued before [1,2,3] that this view is rather narrow. In reality what matters is how much the stakeholders (including the users) consider the system as trustworthy whereby safety is one of the specified properties. Similarly, what a user expects is a **guaranteed QoS (Quality of Service)** level. Depending on the level, it guarantees that the system will be able to deliver its intended functionality even if faults occur. Hence, the ultimate case is one whereby the system survives faults. As this criterion is very wide, this led to the introduction of a novel more normative criterion, called **ARRL (Assured Reliability and Resilience Level)** that differentiates between the failure conditions and how the system copes with it.

Depending on the severity of the fault scenario and the desired continuity of the system's functions this requires increasingly higher levels of ARRL. In traditional systems engineering, the continuation of the services is achieved by reconfiguring the architecture and by redundancy. The question is whether this is sufficient or a necessary condition to reach the novel property of **antifragility** [10]. Before we answer the question, we recapitulate the existing notions of SIL, QoS and ARRL

3.2. Overview of existing criteria in the domain of trustworthiness

3.2.1. Safety Integrity Level

We consider first the **IEC 61508 standard** [4], as this standard is relatively generic. It considers mainly programmable electronic systems. The goal is to bring the risks to an acceptable level by applying safety functions. IEC 61508 starts from the principle that safety is never absolute; hence it considers the **likelihood of a hazard** (a situation posing a safety risk) and the **severity of the consequences**. A third element is the **controllability**. The combination of these three factors is used to determine a required **SIL (Safety Integrity Level)**, categorised in 4 levels, SIL-1 being the lowest and SIL-4 being the highest. These levels correspond with normative allowed Probabilities of Failure per Hour and require corresponding Risk Reduction Factors

that depend on the usage pattern (infrequent versus continuous). The risk reduction itself is achieved by a combination of reliability measures (higher quality), functional measures as well as assurance from following a more rigorous engineering process. The safety risks are generally classified in 4 classes, roughly each corresponding with a required SIL level whereby we added a SIL-0 for completeness. It must be said however that the standards allow quite some room for interpretation, in particular when it comes to the use of probabilities and assessment of the controllability factor.

| Safety Risk Classification | | |
|----------------------------|-------------|---|
| Category | Typical SIL | Consequence upon Failure |
| Catastrophic | 4 | Loss of multiple lives |
| Critical | 3 | Loss of a single life |
| Marginal | 2 | Major injuries to one or more persons |
| Negligible | 1 | Minor injuries at worst of material damage only |
| No Consequence | 0 | No damages, except user dissatisfaction |

The classification leaves room for residual risks but those are not considered design goals but rather as uncontrollable risks. Neither the user nor the system designer has much control over them. This is due to the existence of non-linear discrete subsystems (mainly digital electronics and software) which was elaborated further in [5]. This aspect will be important when we discuss the concept of **antifragility** further in this text.

The SIL level is used as a directive to guide selecting the required architectural support and development process requirements. For example SIL-4 imposes redundancy and positions the use of formal methods as highly recommended.

3.2.2. Quality of Service Levels

A system that is being developed is part of a larger system that includes the user (or operator) as well as the environment in which the system is used. Note as well that this is a hierarchical notion. A system can be a subsystem or a component in a large system and can also include services and processes that support the final mission of a system.

From the user’s point of view, the system must deliver an acceptable and predictable level of service, which we call the **Quality of Service (QoS)**. A failure in a system is not seen as an immediate risk but rather as a breach of contract on the QoS whereby the system’s malfunction can then result in a safety related hazard or loss of mission control, even when no safety risks are present. As such we can see that a given SIL is a subset of the QoS. The QoS can be seen as the availability of the system as a resource that allows the user’s expectations to be met.

| Quality of Service | |
|--------------------|--|
| QoS-1 | There is no guarantee that there will be resources to sustain the service. Hence the user should not rely on the system and should consider it as untrustworthy. When using the system, the user is taking a risk that is not predictable |
| QoS-2 | The system must assure the availability of the resources in a statistically acceptable way. Hence, the user can trust the system but knows that the QoS will be lower from time to time. The user’s risk is mostly one of annoyance and dissatisfaction or of reduced service. |
| QoS-3 | The system can always be trusted to have enough resources to deliver the highest QoS at all times. The user’s risk is considered to be negligible. |

The classification leaves room for residual risks but those are not considered design goals but rather as uncontrollable risks. Neither the user nor the system designer has much control over them. This is due to the existence of non-linear discrete subsystems (mainly digital electronics and software) which was elaborated further in [5]. This aspect will be important when we discuss **antifragility** further in this text.

3.3. The ARRL criterion

We introduce the ARRL or Assured Reliability and Resilience Level to guide us in composing safe and trustworthy systems. The different ARRL classes are defined in Table 3. They are mainly differentiated in terms of how much assurance they provide in meeting their contract in the presence of faults. The reader should keep in mind that the term component can also be a (sub)-system or system acting as components in a larger system.

We should mention that there is an implicit assumption about a system's architecture when defining ARRL. A system is composed by defining a set of interacting components. This has important consequences:

- The component must be designed to prevent the propagation of errors. Therefore the **interfaces** must be clearly identifiable and designed with a "guard". These interfaces must also be the only way a component can interact with other components. The internal state is not accessible from another component, but can only be made available through a well-defined protocol (e.g. whereby a copy of the state is communicated).
- The **interaction mechanism**, for example a network connection, must carry at least the same ARRL credentials as the components it interconnects. Actually, in many cases, the ARLL level must be higher if one needs to maintain a sufficiently high ARRL level at the level of the (sub)-system composed of the components.
- Hence, it is better to consider the interface as a component on itself, rather than for example assuming an implicit communication between the components.

| ARRL definition | |
|-----------------------------|--|
| Inheritance property | Each ARRL level inherits all properties of any lower ARRL level. |
| ARRL-0 | The component might work (“use as is”), but there is no assurance. Hence all risks are with the user. |
| ARRL-1 | The component works “as tested”, but no assurance is provided for the absence of any remaining issues. |
| ARRL-2 | The component meets all its specifications, if no fault occurs. This means that it is guaranteed that the component has no implementation errors, which requires formal evidence as testing can only uncover testable cases. The formal evidence does not necessarily provide complete coverage but should uncover all so-called systematic faults, e.g., a wrong parameter value. In addition, the component can still fail due to randomly induced faults, for example an externally induced bit-flip. |
| ARRL-3 | The component is guaranteed to reach a fail-safe or reduced operational mode upon a fault. This requires monitoring support and some form of architectural redundancy. Formally speaking this means that the fault behaviour is predictable as well as the subsequent state after a fault occurs. This implies that specifications include all fault cases as well as how the component should deal with them. |
| ARRL-4 | The component can tolerate one major fault. This corresponds to requiring a fault-tolerant design. This entails that the fault behaviour is predictable and transparent to the external world. Transient faults are masked out. |
| ARRL-5 | The component is using heterogeneous sub-components to handle residual common mode failures. |

3.4. Is this sufficient for antifragility?

The normative ARRL levels describe as the name says, levels of reliability and resilience. They approach the notion of graceful degradation by redundancy but assuming that in absence of faults the system components can be considered as error-free. The additional functionality and redundancy (that is also error-free) is to be seen as an architectural or process level improvement. But in all cases, contrary to the antifragility notion, the system will not gain in resilience or reliability. It can merely postpone catastrophic failures while maintaining temporarily the intended services. It does this by assuming that all types of faults can be anticipated, which would be the state of the art in engineering.

3.4.1. Antifragility assumptions

However, the proposed scheme introduces already two concepts that are essential to take it a step further. Firstly, there is redundancy in architecture and process and secondly, there is a monitoring function that acts by reconfiguring the system upon detecting a fault.

So, how can a system become “better” when subjected to faults? As we introduce a metric as a goal, we must somehow measure and introduce feedback loops. If we extrapolate and scale up, this assumes that the system has a type of self-model that it can use to compare its current status with a reference goal. Hence, either the designer must encapsulate this model within the system or the model is external and becomes part of the system. If we consider systems that include their self-model from the start, then clearly becoming a “better” system has its limits, the limit being the designers’ idea at the moment of conception. While there are systems that evolve to reach a better optimum (think about neural networks or genetic algorithms), these systems evolve towards a limit value. In other words they do not evolve, they converge.

If on the other hand we expand the system as in Figure 1, then the system can evolve. It can evolve and improve because we consider its environment and all its stakeholders of which the users as part of the system. They continuously provide information on the system’s performance and take measures to improve upon it. It also means that the engineering process doesn’t stop when the system has been put to use for the first time. It actually never ends because the experience is transferred to newer designs.

There are numerous examples of antifragile systems already at work, perhaps not perfect all the time though most of the time. A prime example is the aviation industry that demonstrates by its yearly decreasing number of fatalities and quality of service that it meets the criterion of antifragility. Moreover, it is a commercial success. So let’s examine some of its properties and extract the general principles, as reflected in the aviation standards and practice [6].

| Lessons to be learned from aviation safety | |
|---|---|
| Aviation specific | Generic property |
| The industry has a long track record | The domain has undergone many technological changes whereby an extensive knowledge was built up. |
| Development of systems follows a rigorous, quantifiable, certifiable process, that is widely published and adopted. | The process is open and reflects the past experience and is certified by an independent external authority. |
| Certification and Qualification requirements foster developing “minimal” implementations that still meet the operational requirements. | Systems are designed to be transparent and simple, focusing on the must-haves and not on the nice to haves. |
| Airplanes are designed to be 100% safe and to be operated in 100% safe conditions. The domain has a goal of perfection. | Any deviation is considered a failure that must be corrected. By design the system, its components and operating procedures aim at absence of service and safety degradation. |
| Any failure is reported in a public database and thoroughly analysed. | Any issue is seen as a valuable source of information to improve processes and systems. |
| Airplanes are operated as part of a larger worldwide system that involves legal authorities, the operators, the manufactures, the public and supervising independent authorities. | A (sub)system is not seen in isolation but in its complete socio-economic context. This larger system is self-regulating but supervised and controlled by an independent authority. |
| Airplanes have a long life time and undergo mid-life updates to maintain their serviceability | The focus is on the service delivered and not on the system as a final product. |
| Fault conditions are preventively monitored. The system is fault tolerant through redundancy, immediate repair and preventive maintenance. | A process is in place that maintains the state of the system at a high service level without disrupting the services provided. |

3.4.2. Some industries are antifragile by design

To remain synoptic, we will list a few key principles of the aviation industry and derive from them key generic principles which apply to other systems and provide them with antifragile properties.

Table 4 can also be related to many other domains that have a significant societal importance. Think about sectors like medical devices, railway, automotive, telecommunications, internet, nuclear, etc. They all have formalised safety standards which must be adhered to because when failing they have a high impact at socio-economic level.

At the same time, systems like railway that are confined by national regulations clearly have a higher challenge to continue delivering their services at a high level. As a counter example we can take a look at the automotive sector. Many more people are killed yearly in traffic than in airplanes, even if cars today are stuffed with safety functions. In the next section we will explore this more in detail.

Deducting some general properties out of the table 4, we can see that systems that could be termed antifragile are first of all not new. Many systems have antifragile properties. Often they can be considered as complex (as there are many components in the system) but they remain resilient and antifragile by adopting a few fundamental rules:

- **Openness:** all service critical information is shared and public.
- **Constant feedback loops** between all stakeholders at several different levels.
- Independent **supervising authorities**.
- The core components are designed at ARRL-4 and ARRL-5 levels, i.e. **fault tolerant**.

3.4.3. Do we need an ARRL-6 and ARRL-7 level?

An ARRL-5 system can be seen as a weak version of a resilient system. While it can survive a major fault, it does so by dropping into an ARRL-4 mode. The next failure is likely catastrophic. However airplanes are also designed as part of a larger system that helps to prevent reaching that state. Continuous build-in-test functions and diagnostics will detect failures before they become a serious issue. Ground crews will be alerted over radio and will be ready to replace the defective part upon arrival at the next airport. We could call this the ARRL-6 level whereby fault escalation is constrained by early diagnostics, monitoring and the presence of a repair process that maintains the operational status at an optimal level. Note that in large systems like server farms and telecommunication networks similar techniques are used. Using monitoring functions and hot-swap capability on each of the 1000's of processing nodes, such a system can reach almost an infinite lifetime (economically speaking). Even the technology can be upgraded without having to shut down the system.

The latter example points us in the direction of what a normative ARRL-7 level could be. It is a level whereby the system is seen as a component in a larger system that includes a continuous monitoring and improvement process. The later implies a learning process as well. The aviation industry seems to have reached this maturity level. The term maturity is no coincidence, it reminds of the maturity levels as defined by CMMI levels for an organisation. The table below summarises the new ARRL levels whereby we remind the reader that each ARRL level inherits the properties of the lower ARRL levels.

| ARRL-6 and 7 definitions | |
|--------------------------|---|
| ARRL-6 | The component (or subsystem) is monitored and designed for preventive maintenance whereby a supporting process repairs or replaces defective items while maintaining the functionality and system's services. |
| ARRL-7 | The component (or subsystem) is part of a larger "system of systems" that includes a continuous monitoring and improvement process supervised by an independent regulating body. |

3.5. Automated traffic as an antifragile ARRL-7 system

As we discussed earlier [1, 2, 3, 5], the automotive sector does not yet meet the highest ARRL levels as well as in the safety standards (like IEC-26262) [7] and in reality (1000 more people are killed in cars than in airplanes worldwide and even a larger number survive with disabilities)[8,9,11]. The main reason is not that cars are unsafe by design (although fault tolerance is not supported) but because the vehicles are part of a

much larger traffic system that is largely an ad-hoc system. Would it be feasible to reach a similar ARRL level as in the aviation industry? What needs to change? Can this be done by allowing autonomous driving?

A first observation is that the vehicle as a component now needs to reach ARRL-4, even ARRL-5 and ARRL-6 levels. If we automate traffic, we might be able to take human errors out of the equation, but then following design parameters become crucial:

- The margin for driving errors will greatly decrease. Vehicles already operate in very dynamic conditions whereby seconds and centimetres make the difference between an accident and not an accident. With automated driving, bumper to bumper driving at high speed will likely be the norm.
- The driver might be a back-up solution to take over when systems fail, but he is unlikely to be well enough trained and therefore to react in time (seconds).
- A failing vehicle can generate a serious avalanche effect whereby many vehicles become involved and the traffic system can be seriously disrupted.
- We can expect that the psychological acceptance for accidents will be much lower for when the accident was caused with the computer at the steering wheel than when a human driver is at the steering wheel.

Hence, **vehicles need to be fault tolerant**. First of all they constantly monitor and diagnose the vehicle components to prevent pro-actively the failing of subsystems and secondly when a failure occurs the function must be maintained allowing to apply repair in a short interval.

A second observation is that the automated vehicle will likely constantly communicate with other vehicles and with the traffic infrastructure. New vehicles start to have this capability today as well, but with automated vehicles this functionality must be guaranteed at all times as a disruption of the service can be catastrophic.

A third observation is that the current road infrastructure is likely too complex to allow automated driving in an economical way. While research vehicles have been demonstrated the capability to drive on unplanned complex roads, the question is whether this is the most economical and trustworthy solution.

Automated traffic can be analysed in a deeper way. Most likely, worldwide standardisation will be needed and more openness on when things fail. Most likely, fully automated driving providing very dense traffic at high speed will require dedicated highways, whereas on secondary roads the system will be more a planning and obstacle avoidance assistant to the driver.

One can even ask if we should still speak of vehicles. The final functionality is mobility and transport. For the next generation, cars and trucks as we know them today might not be the solution. A much more modular and scalable, yet automated, transport module that can operate off-road and on standardised auto-highways is more likely the outcome. Users will likely not own such a module but rent it when needed whereby operators will be responsible for keeping it functioning and improving it without disrupting the service. Independent authorities will supervise and provide an even playing field. Openness, communication and feedback loops at all levels will give it the antifragility property that we already enjoy in aviation. At that moment, Mobility will have become a Service.

3.6. Is there an ARRL-8 level?

One can ask the question whether we can define additional ARRL levels. ARRL levels 0 to 7 are clearly defined in the context of (traditional) systems engineering whereby humans are important agents in the required processes to reach these levels. One could say that such a system is self-adaptive. However the antifragile properties (even when only partially fulfilled) are designed in and require conscious and deliberate actions to maintain the ARRL level. If we look at biological systems we can see that such systems evolve without the intervention of external agents (except when they stress the biological system). Evolution as such has reached a level whereby the “architecture” is self-adaptive and redundant without the need for conscious and deliberate actions. We could call this the ARRL-8 level.

3.7. Conclusions

Taleb [10] defines antifragile mostly in the context of a subjective human social context. He quotes the term to indicate something beyond robustness and resilience that reacts to stressors (and alike) by actually improving its resistance to such stressors. Taking this view in the context of systems engineering we see that such systems already exist. They are distinguished by considering the system as a component in a greater system that includes the operating environment and its continuous processes and all its stakeholders. Further differences are a culture of openness, continuous striving for perfection and the existence of numerous multi-level feedback loops whereby independent authorities guide and steer the system as a whole. The result is a system that evolves towards higher degrees of antifragility. An essential difference with traditional engineering is that the system is continuously being redefined and adapted in an antifragile process.

We also defined two new levels for the normative ARRL criterion, ARRL-6 indicating a system that preventively seeks to avoid failures by repair and ARRL-7 whereby a larger process is capable of not only repairing but also updating the system in a controlled way without disrupting its intended services. Given the existence of systems with such (partial) properties, it is not clear whether the use of the neologism “antifragile” is justified to replace reliability and resilience, even if it indicates a clear qualitative and distinctive level. This will need further study.

4. The systems Grammar of GoedelWorks

Just by defining concepts, we don't have a System. This is true for the Project domain and this is true for the Process domain. In the Project domain, we create a System by defining how the different Entities interact. In the Process domain, we define how the different Entities are related. The concepts and their relationships define an emerging Process that has meaning just like the Grammar rules on the terms of a sentence give the sentence a specific meaning. Therefore, we call this the Systems Grammar.

4.1. Systems Grammar

To understand the emergence of meaning in a Project, one must look at how a System emerges in a Process. It is initially very much a mental, cognitive Process done by humans.

When a stakeholder speaks or thinks about the "**Mission**" of a System, he probably has a mental picture about the System. Likely this picture will resemble something he knows from seeing in the real world or a virtual world (e.g. movies and SF-novels). What follows is a mental Process of refinement. The mission, as a top level Requirement (e.g. "We need a deep ocean submarine"), is decomposed in more specific **Requirement statements** about the System until an atomic level has been reached, e.g. "the submarine should be yellow". The Process is one of **decomposition** and **refinement**, not one of derivation. We call this a **structural link** and it only applies to Entities of the same type (in this case Requirements). As is often the case, Requirements will be overlapping and not necessarily fully coherent. As they are defined without much analysis, there will be conflicting Requirements, certainly when we take into account the boundary conditions of a real implementation. Engineering is always a **trade-off** exercise.

Once everyone agrees on the Requirements (subject to further investigation) that are to be retained, we can consider these Requirements as "Approved". Often this is often called the **Kick-Off** point in a Project, at least if it is decided to go ahead with the next steps. Some Projects will start earlier, some Projects will then stop as this decision acts as a "gate" for the next steps.

The next step is to refine the Requirements into hard **Specifications**. What this means is that we derive concrete Specifications from the Requirements. Concrete means that they become measurable, verifiable and also implementable. They should also be unique. One can consider Specifications as quantified or qualified Requirements, which indicates that there is some continuity whereby Requirements evolve until they become concrete enough to be called Specifications. Practically speaking, we can select or develop designs that meet the Specifications and we can define tests that allow us to verify and validate it. The relationship between Requirements and Specifications is one of **dependency**. We call this an **association link**. Multiple Requirements will result in multiple Specifications and multiple Specifications will be derived from multiple Requirements whereby the presence of orthogonality ("separation of concerns") helps to see order in chaos. Similarly, **Verification Tasks** will have association links with the Specifications for the Process to be followed during the **Development Tasks** and **Test Tasks** will have an association link with the Specifications for the properties of the Work Products developed during the development. "Process" and "Project" can be seen as the container of the "how" and the "what" aspects of Systems engineering.

The actual development work to be done is the development of Entities that implement the Specifications. We call these the **Work Products**. For Process Entities, these will typically be templates or guidelines (when developing a Process). For Project Entities, these will be reports or documented evidence (when the Work Product is a Project Requirement) or the System or its System components (when the Work Product is a Project Requirement). The Work Products are the result of the work done in Work Packages, composed of Tasks (Planning, Design, Development, Verification, Test, Integration, Validation and Review Tasks). Hence Tasks are linked in a Work Package. **Validation Tasks** were not discussed yet and they verify that the Work Product meets their Requirements (answering the question "is this the right Entity?"). A Work Package is also linked with the inputs it needs: with **Specifications** (for the Work Products) and with its required **Resources**. One could see the Specifications also as Resources, but for clarity it is better to separate them.

In a System development Project the Work Package produces **Models** as their Work Products. Models can further be decomposed in sub-Models and finally into **Items** and **Interactions**. A Model has hence an association link with a Work Package and a structural link with it composing Items. One can also consider a Model as a subtype of a Work Product, but the separation was kept for a clearer distinction between Process and Projects.

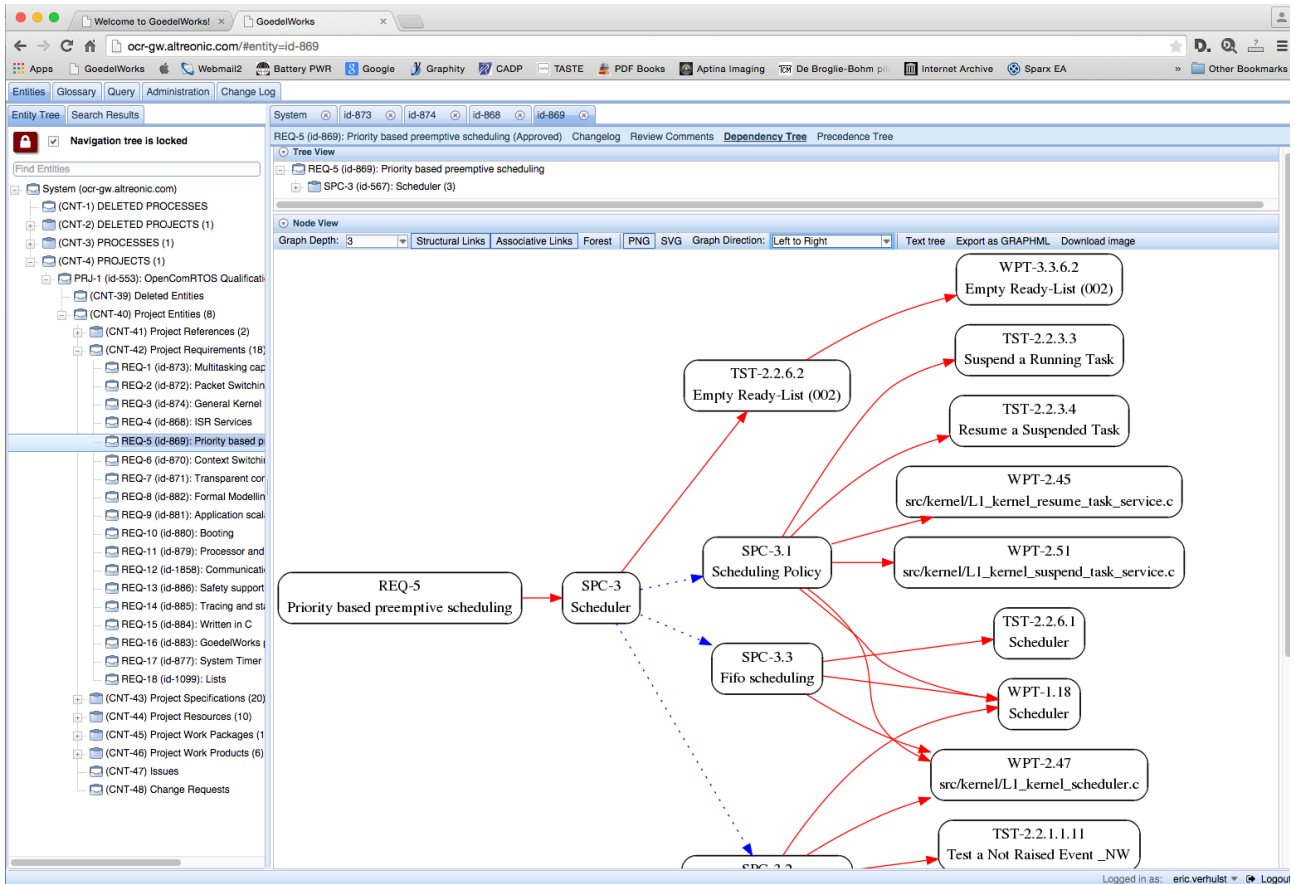
Note that almost any Entity can be decomposed into smaller sub-Entities. Additional Entities that are related to a Project are **References** that can be associated with Requirements and **Change Requests and Issues**. The latter two can be associated with any Entity.

The result of introducing these links can best be illustrated graphically. The dependency graph was generated for the Requirement 5 with id-869 in the OpenComRTOS Qualification Package. First we look at the textual representation below. We can follow the trace from the Requirement REQ-5 being refined into Specification SPC-3 with sub-Specifications SPC-3.1, 3.2 and 3.3 with defined Tests (TST) and dependent Test code (WPT) but also the implementation code (WPTs with a link to where it is stored in the version control repository). This representation is however not so easy to navigate, hence the next figure show the screenshot and an exported graphml. The latter can be manipulated in a graphml editor or viewer (example: Yed).

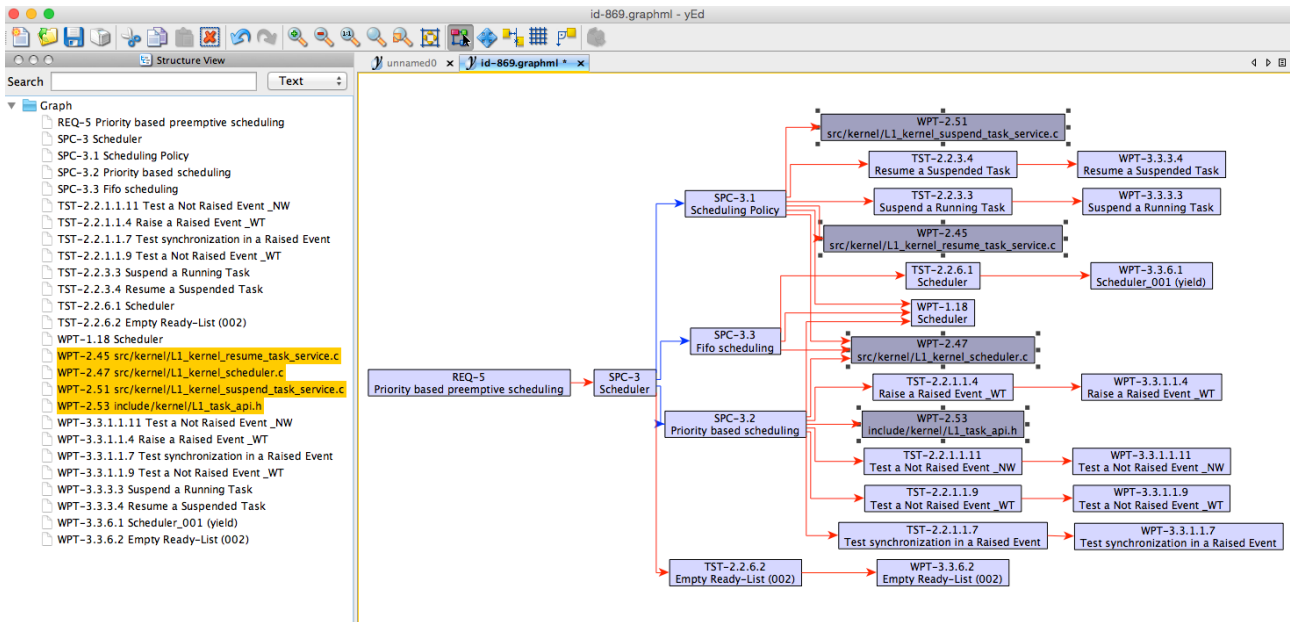
Dependency Tree for REQ-5:

```
REQ-5: Priority based preemptive scheduling
--SPC-3 - Scheduler
----TST-2.2.6.2 - Empty Ready-List (002)
-----WPT-3.3.6.2 - Empty Ready-List (002)
----SPC-3.2 - Priority based scheduling
-----WPT-1.18 - Scheduler
-----TST-2.2.1.1.4 - Raise a Raised Event _WT
-----WPT-3.3.1.1.4 - Raise a Raised Event _WT
-----TST-2.2.1.1.7 - Test synchronization in a Raised Event
-----WPT-3.3.1.1.7 - Test synchronization in a Raised Event
-----TST-2.2.1.1.9 - Test a Not Raised Event _WT
-----WPT-3.3.1.1.9 - Test a Not Raised Event _WT
-----TST-2.2.1.1.11 - Test a Not Raised Event _NW
-----WPT-3.3.1.1.11 - Test a Not Raised Event _NW
-----WPT-2.47 - src/kernel/L1_kernel_scheduler.c
-----WPT-2.53 - include/kernel/L1_task_api.h
----SPC-3.3 - Fifo scheduling
-----TST-2.2.6.1 - Scheduler
-----WPT-3.3.6.1 - Scheduler_001 (yield)
-----WPT-1.18 - Scheduler
-----WPT-2.47 - src/kernel/L1_kernel_scheduler.c
----SPC-3.1 - Scheduling Policy
-----TST-2.2.3.3 - Suspend a Running Task
-----WPT-3.3.3.3 - Suspend a Running Task
-----TST-2.2.3.4 - Resume a Suspended Task
-----WPT-3.3.3.4 - Resume a Suspended Task
-----WPT-2.45 - src/kernel/L1_kernel_resume_task_service.c
-----WPT-2.51 - src/kernel/L1_kernel_suspend_task_service.c
-----WPT-1.18 - Scheduler
-----WPT-2.47 - src/kernel/L1_kernel_scheduler.c
```

A unifying view on systems engineering



A .dot dependency graph in GoedelWorks



A dependency graph exported to a graphml viewer/editor

4.2. Terminology and its conventions in GoedelWorks

The attentive reader, especially when familiar with existing standards might notice some distinct differences between the GoedelWorks terminology and the one used in e.g. safety and systems engineering standards. For example most standards will not use the term specification and only talk about requirements. Some standards differentiate them by using qualifiers. E.g. DO-178, the avionic standard speaks of High Level Requirements (HLR) and Low Level Requirements (LLR), the latter often been associated with the details of the implementation. Some other standards like ECCS (the ESA standard) mix the two. Even if this can be explained (Specifications evolve by decomposition and refinement from Requirements), it leads to confusion because as we have seen, we need clear and unambiguous concepts to communicate. Another example is the use of the terms verification, testing and validation. These are also very often used differently depending on the standard and the context, often resulting in confusion.

Solving this issue is not trivial. One has essentially two options. The first option is to invent a new term. This has as drawback that it has to be created and introduced (often people use acronyms for achieving this). The second option is to use a term in natural language that exists with a close enough generally accepted meaning. This is the option adopted in GoedelWorks. In order to reduce the possible confusion, each of these GoedelWorks key terms is capitalised and in the portal a standardised acronym is used. See Annex for a detailed list.

In the table below one can find the specific terminology used in GoedelWorks. It covers the main Entities. Further on we will see that a GoedelWorks Project consists of a number of Works Packages, as instances of a prescribed Process Steps. A Work Package has itself a number of Artefacts that constitute the evidence for the performed Activities. These also serve as inputs to the next Tasks that need to be done. Following the dependency relationships, they also need to be approved before the following Tasks can be approved.

| GoedelWorks' specific Terminology | |
|--|--|
| Term | Definition |
| System | The System is the Entity under consideration as the object of the Systems Engineering activities. This is called the System under development. It interacts with two other Systems. The environment is the external System in which the System will be placed and the user or operator, with whom the System fulfills its mission. |
| Mission | The Mission is the top level Requirement for which the System is being developed. |
| Requirement | A Requirement is any statement by any stakeholder that is related to the mission of the System. |
| Specification | A Specification is a mapping of the Requirements into a verifiable Requirement, often by quantifying and qualifying the Requirement statements. A Specification must have a "Test Case" else it might be questionable if it can be verified that the Developed Entity meets its Specifications. |
| Reference | A Reference is any relevant input or information that is generic but is necessary or useful to carry out the engineering Project or Process. Often it will be external to the Project but related to it. |
| Process | A Process is a well defined number of Steps and Activities that define how a Project has to be executed. It also defines the Artefacts and Work Products that must be delivered allowing Verification and Validation of the Project execution. This can be used for Qualification of Certification of the System developed. |
| Project | A Project encompasses all activities, including those required by the Process agreed upon, that contribute to the realization of the System under development. A Project is composed of a number of Work Packages, instances of the Steps defined in the Process. |
| Work Package | A Work Package is a set of Activities that are related to the development of the Work Products. A Work Packages consists of Planning, Designing, Development, Verification, Test, Integration, Validation and Review Activities. Work Packages require Specifications and Resources to be able to execute the activities. |
| Work Product | A Work Product is the end-result of a Work Package and is one the concrete Entities that need to be developed to develop the Systems as a whole. |
| Artefact | An Artefact is an Entity that is produced along side the Work Product as a piece of evidence that the Work Product and hence the whole system meets its Requirements. |
| Resource | A Resource is an Entity that constitutes a Requirement for executing Work Package Activities. The main ones are human Resources qualified to take up a specific Role in the Project. Other Resources are most of the time material in nature. |
| Role | A Role is a human Resource meeting specific Requirements in terms of Capabilities as defined by the Process Specifications. |
| Planning | Planning defines how and when the available Resources should be used to execute a given Activity. |
| Designing | The Design Activity defines how the system will be architecturally structured and how the different Activities contribute to it. |

| | |
|---------------------|---|
| Development | The Development Activity is the activity that actually develops or implements the Work Products. |
| Verification | A Verification Activity is the activity that verifies that the development was done according to the agreed upon Process Specifications. |
| Testing | A Test Activity is the activity that verifies that the developed and verified Work Products meet their approved Specifications. |
| Integration | An Integration Activity combines the composing Items and Work Products into a larger System or subsystem component. |
| Validation | A Validation Activity is the activity that after integration of all System components and Work Products validates that the tested Work Products meet the original and approved Requirements. |
| Reviewing | A Review activity is a confirmation measure that confirms, best independently, that an Activity was done correctly and that its approval was justified. It also acts an extra feedback loop to detect oversights. |
| Model | A Model is a Work Product developed in a Systems engineering Project during the development activities. The finally approved System is considered as an implementation Model. |
| Item | An Item is a generic System component, making abstraction of the implementation. |
| Interaction | An Interaction is an Entity that creates a structural connection between one or more System Items. |
| Link | A Link is a relationship between one or more Entities. We distinguish between associative links (e.g. dependency relationships), decomposition links and process flow links (defining a partial order in time). |

4.3. Process Steps, instantiated as Work Packages in a concrete Project

As we will see further, in GoedelWorks we consider that a System is the result of a specific Project whereby a specific Process is followed. In a Process definition we will find a typical succession of Steps that generically define a partial order of activities. Examples of such Process are the ASIL flow (single V-model) or DO-178 (double V-model). In a concrete organisation these will have combined with organisation specific Steps, procedures and guidelines.

In a concrete Project, these Process Steps are instantiated as concrete Work Packages. In GoedelWorks the Work Package has been structured using a standardised template, a partial order of Work Package Activities and associated Artefacts. One could consider these as the micro steps of a small iterative V-model.

While the terminology was chosen to be generic (and applicable to almost any Process Step), it more or less reflects a typical Development Work Package as this is considered as the core of an engineering Project.

In the table below, we have listed the standardised Activities and their composing phases.

| Work package activities | |
|--------------------------------|--|
| <u>Kick-Off</u> | The starting moment for a Work Package. The Kick-Off meeting is a team event whereby the objectives of the Work Package are clearly identified, the Process to be followed defined, Resources assigned and approval is obtained from the stakeholders (external as well as internal to the Work Package). |
| Planning | This Activity defines how and when the Resources will be used to meet the Project's objectives. It assigns the Resources and defines a planning in time. |
| Designing | This activity defines how the Work Product will be structured. It defines for example the architecture, lay-out, algorithms, etc. One could say that it develops Models without executing them. All the Design Models should support the Development. |
| Developing | This Activity consists of the actual work to be done. It is the core activity supported by all other activities to reach the objectives in a trustworthy way. The end results are Models (simulation, formal, etc.) with the Implementation Model being the concrete goal of the Project. It focuses on developing "the right thing". In the software domain people might call this activity "Implementation" but generally speaking Development is the term used across multiple domains. |
| Verification | This Activity verifies that the development was done as prescribed according to the Process Requirements and Specifications. It focuses on developing "the right way". |
| Testing | This Activity consists of verifying that the developed Items meet their functional and non-functional Specifications. Testing is likely to be repeated after integration at the system level. The resulting measurements are then called "Characteristics". |
| Integration | This Activity combines the composing Items of the developed Work Products into a single entity (that can be a system, sub-system or component). It can be seen as part of Development but the focus is here on the interactions and connections between the Items. |
| Validation | This Activity will verify that the Work Product's Requirements are met. After Integration, the focus is on verifying that the "right system" was developed. The measured characteristics might differ from the Specifications and a decision is to be made to approve the Work Products on basis of the intended Requirements. |
| Review | This Activity will combine the results of all Activities and conduct a review seeking oversights and residual issues. In principle, Review activities are done during any activity and preferably involve independence and peer review as often prescribed in the Process Requirements and mandated by the standard to comply with. |
| <u>Release</u> | When all Activities and their corresponding evidence have been approved, the Work Products can be released, for example to transfer them to the production facilities. At this stage, the Work Products and their evidence will be "frozen" and enter strict configuration management. |

Each of these Activities can further be divided in four main phases:

| Activity stages | |
|---------------------|--|
| Planning | This phase extracts and refines the planning done at the Work Package level into a planning specific for the Activity. |
| Doing | The actual work to be done in the Activity. |
| Documenting | A documented record of what was done in the Activity. It must be complete in the sense that any element that has or had a possible impact on the resulting Work Product must be traceable and identifiable. |
| Confirmation | Each activity needs to be reviewed for residual issues and oversights. Often dictated by Process Requirements, reviews are best done by independent people. Their goal is to confirm in an independent way that the objectives were met. It also serves as additional feedback that seek to find oversights. |

The Activities hence produce two types of outputs. The actual Items developed are called **Work Products** whereas the supporting evidence are called **Artefacts**. In trustworthy (and hence certifiable) Systems Engineering any product comes with the supporting evidence (which can be seen as a contract) that it meets its Requirements and Specifications and that it was developed according to the Required and Specified Process (that is meant to assure the trustworthiness of the Work Products).

The final steps are to transfer the development to production. A good Systems engineering Project will have been designed with production and deployment issues in mind. This affects quality and it also affects the (financial) bottom-line. During the life time of a System (or product) good Engineering will have anticipated maintenance and even upgrading to assure that the System keeps performing at its specified level. Finally, when the System will be taken out of service, it will need to be disposed off. Modern, environmentally friendly Engineering will have thought about that as well.

4.4. In the end, any project is iterative

A superficial reading of the preceding chapters might raise the question if what we describe is a classical waterfall or V-Model for the engineering Process. This means that the different activities follow a strict linear order in time (Requirements-Specifications-Work Packages- etc. - Work Products). This is a wrong view and not desirable. First of all in a Systems engineering Project nothing is static unless frozen or approved. Most of the time Entities will be "in work". Secondly, most Systems engineering Projects will need to deliver hundreds or thousands of Work Products and concurrent engineering is almost a must. Thirdly, Systems engineering is an iterative Process. Requirements might have resulted in unrealistic Specifications or while working on the implementation, new issues will have been discovered and therefore Specifications adapted. Therefore, the classical V-Model is in reality an interconnected composition of multiple V-Models, essentially one for each Work Product. This applies as well for the Process Artefacts (e.g. documents and reports) as for the Project Work Products (e.g. the System Models and Entities developed).

To bring order in its execution Systems engineering works with "**states**" for all Project Entities. Each Entity (for example a Specification) starts his life when it is created. We call this state '**Defined**'. Once defined, it has to be completed and worked upon. We call this state "In Work". At some point in time, the Specification will be frozen, we call this "**Frozen for Approval**". If the Specification is then approved, we call this "**Approved**". At this moment anyone making any change to the Specification results in the Specification being put to "In Work" again.

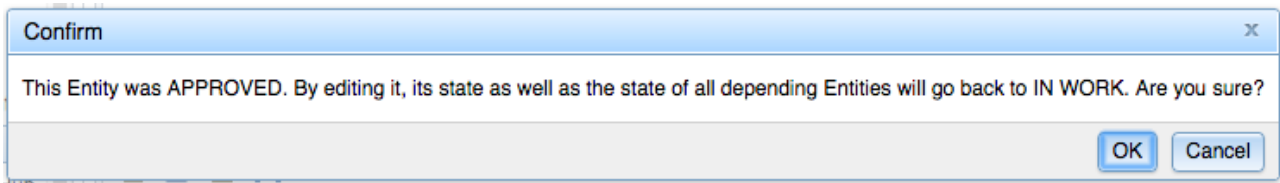
Why is this important? Configuration management is related to the association and structural links. For example Specifications are input to the Test Tasks. If the Specifications are not frozen and approved, testing will have to restart whenever the Specifications are changed. What results is that all Work Products can only

be approved if all preceding Entities have been approved. This defines the final, approved System configuration. The links automatically create a chain of approval events that synchronise the Project at the state transitions to "Approved", hence a partial order in time. As long as everything is not "approved" work can proceed in parallel, provided there is not a dependency. This also shows why a modular and concurrent architecture is beneficial. Not only does it promote concurrent and iterative engineering, it will also be more resilient as errors (and faults) will have less global impact.

GoedelWorks handles the state transitions using the following rules:

- No Entity can be approved unless all its preceding Entities have been approved.
- No Entity can be approved unless all its composing structural Entities have been approved.

This is illustrated with the following screenshot whereby invoking the edit mode on an Approved entity triggers a warning.



4.5. Systems Engineering as a collection of Views

The reader might be surprised that most of Systems Engineering can be described with only a handful of types of Entities and 2 types of relationships. This is the result of formalisation work whereby a hierarchy of meta-Models, Models and specific instances was further refined and grouped in an orthogonal way. The result was a concrete Systems Grammar.

What makes Systems Engineering confusing is that several views on the System are simultaneously present and this often results in different terminologies in the natural language domain.

In the Requirements and Specification phase, the subject of interest is a mostly related to properties of the System. As this should be done before an implementation (a System Model with its Entities) has been selected, the difficulty is that one has to think abstractly. Certainly for the HARA phase this is a challenge as one has to think about failure cases that should not happen at all (but eventually they will).

In the development phase, the Specifications will have been mapped onto Models and their Entities. This is an architectural view.

In the planning phase, the attention shifts to Resource planning with milestones and deliverables.

The most confusing is perhaps that before a trustworthy Systems engineering Process can proceed, the methodology must be defined and concretely this means that the Process is defined. Developing a Process is a Project in itself. It will follow a similar Process as the one it will develop, although more on an intuitive base because there is not standard Process defined. Process Requirements come from many stakeholders through external and internal standards and regulations. They have a societal context, for example because safety and security related risks have to be mitigated by law and certification will often be a legal Requirement before the System can be put in service or the product can be commercialised. While these standards speak of Work Products, these are actually Requirements and Specifications for the development Projects. If a specific organisation has adopted such a standard based Process, it will likely have developed template versions to reduce the administrative and management burden on the engineering itself. Hence, we see that a Work Product like e.g. a test plan will morph from a Reference into a Process Requirement, into a Process Specification, into a template and finally into a specific test plan resulting in a test report. Similarly, a component procured for the execution of a Project, might have followed a similar path. In each of these steps is likely to be given different names.

Trustworthy Systems Engineering with GoedelWorks 3

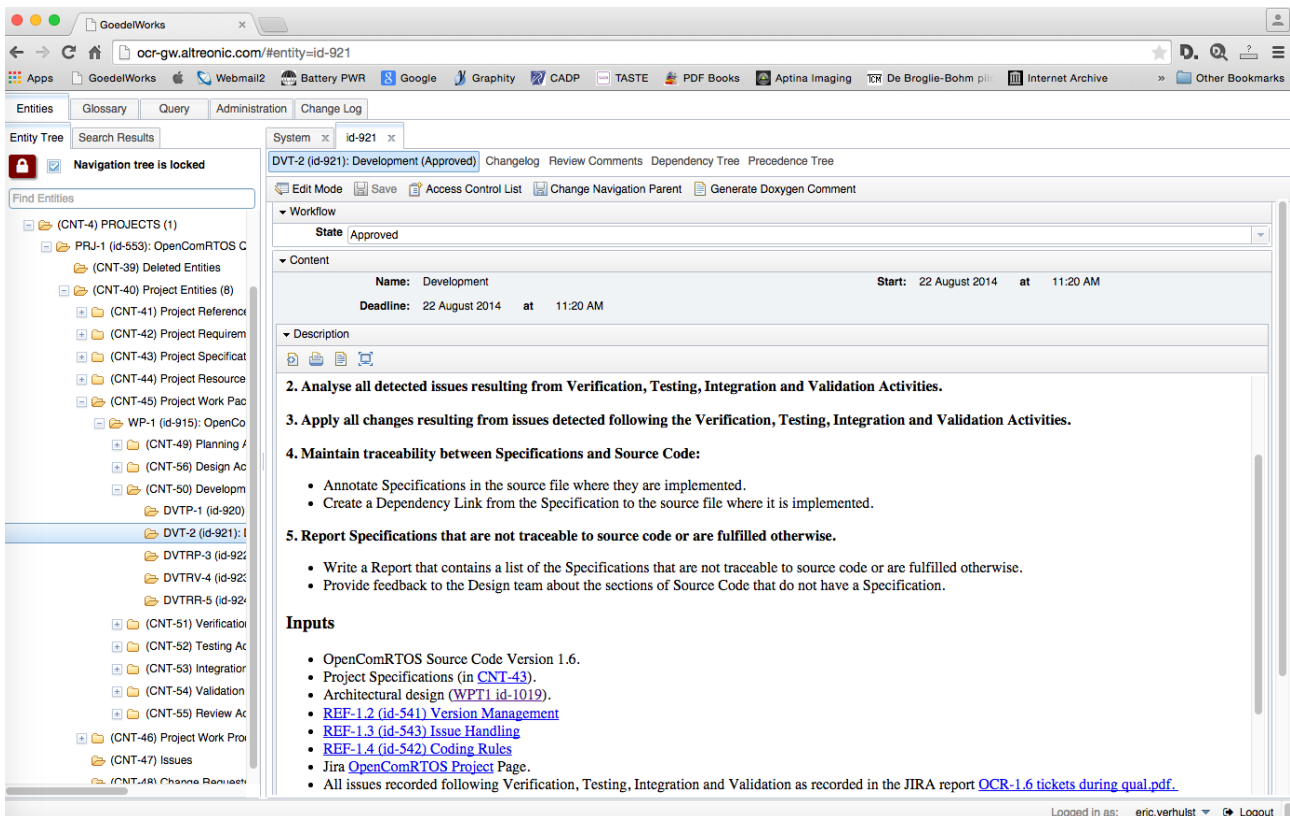
The result is that a System under development when approved and released for production is more than the System itself. It is the result of all the different views combined. As trustworthiness and quality are long term issues, it shows the importance of traceability. It also shows how this is embedded in the culture of the organisation, the region, the society and the education that people receive.

5. Description of the GoedelWorks Environment

In this chapter we describe the GoedelWorks environment. Based on a unifying systems grammar, it supports defining Processes, Projects and Work Plans in a traceable way so that the evidence (as Artefacts) remains linked with the Work Products, facilitating Qualification and Certification.

5.1. Principles of operation

The GoedelWorks environment is composed of a server program that manages all the data in a repository and manages the interaction with the users, who use a client software through a web browser. User can create, modify and delete entities, either belonging to a defined Process or a defined Project.



A GoedelWorks screenshot of a Development Task Entity

Some principles apply:

- As a convention GoedelWorks Entities start with an uppercase letter, else the acronym will be used.
- Each Entity receives a unique identifier upon its creation. This is for reasons of traceability.
- Entities are never physically deleted but moved to a Container with deleted entities.
- On the navigation tree, entities are grouped in Containers per type of Entity.
- The user can create dependency links as permitted by the System Grammar (see below).
- An Entity can be decomposed into sub-Entities.
- When being edited, a write-lock is automatically created and no other user is allowed to make modifications. Reading is always allowed.
- Different users have access rights to an entity depending on the permissions that were set for the his roles.
- Each change to an Entity is logged whereby the Entity receives a unique revision number.

- Entities of the same type are grouped in Containers.
- GoedelWorks is not document based but Entities can have documents (actually any file) as attachments and the user can export Entities (or complete Projects and Processes) to a timestamped hyperlinked html or pdf document.
- Some entities can be 'linked' with version control systems, such as Subversion and Git. In these cases the full content of the repository item (as well as the information about the revision number in the version control system, the timestamp of the latest modification and the user who did it) is copied to the GoedelWorks repository, so that it remains independent from external repository. The user is also allowed to modify the document and send the modifications to the version control system

5.2. Organisational functions of a GoedelWorks portal

Adopting a GoedelWorks portal brings many organisational functionalities together in a single environment. We list some of the most important ones:

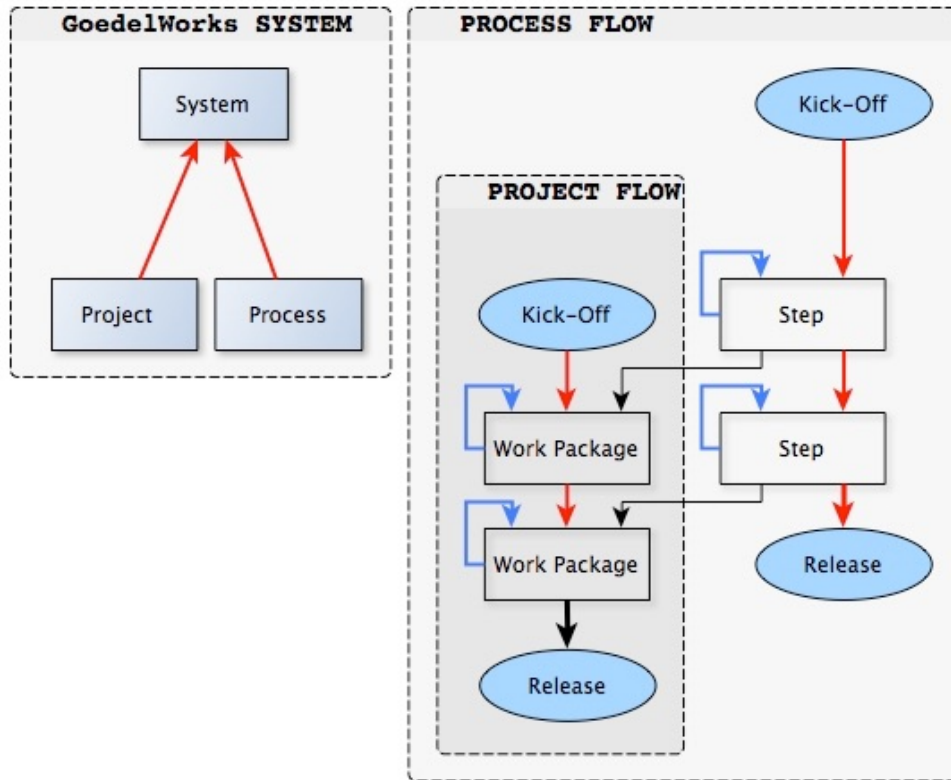
A GoedelWorks portal acts like a structured knowledge repository: while GoedelWorks offers the choice to organise for example a Process or a Project according to its Systems Grammar, this is not a must. The user can define small Containers (incomplete according to the System Grammar) and fill it with data about a specific topic. However, he can gradually structure the data (for example by linking the data with references) so that the data becomes a structured knowledge repository, easy to navigate and easy to share.

- A GoedelWorks portal acts like a safe communication and collaboration platform: accessible from any browser, teams can easily cooperate anywhere and anytime, while the data remains consistent with its version managed at the central repository.
- A GoedelWorks portal acts like an organisational set of guidelines: organisations can easily define their internal processes and integrate them with standards based Processes they have to follow for e.g. Qualification or Certification. When executing Projects, users can link their Activities to the Process Specifications.
- A GoedelWorks portal acts like a structured engineering environment, supporting a systematic execution of (engineering and other) Projects. The Systems Grammar is gently enforced so that the users automatically but incrementally define all the Project Entities and dependencies. When Requirements change, executing an impact analysis is a matter of seconds to generate the dependency tree.
- A GoedelWorks portal allows to generate the evidence needed for Qualification and Certification of engineered systems and products. While a Project is filled up with Entities, dependency trees ensure that the Project can only be approved when all Activities, Work Products and Artefacts are present and were approved in the right order. Moreover, as GoedelWorks' System Grammar acts like a domain independent meta-model, the Project can often be used for Qualification or Certification across multiple domains.

5.3. GoedelWorks Systems Grammar

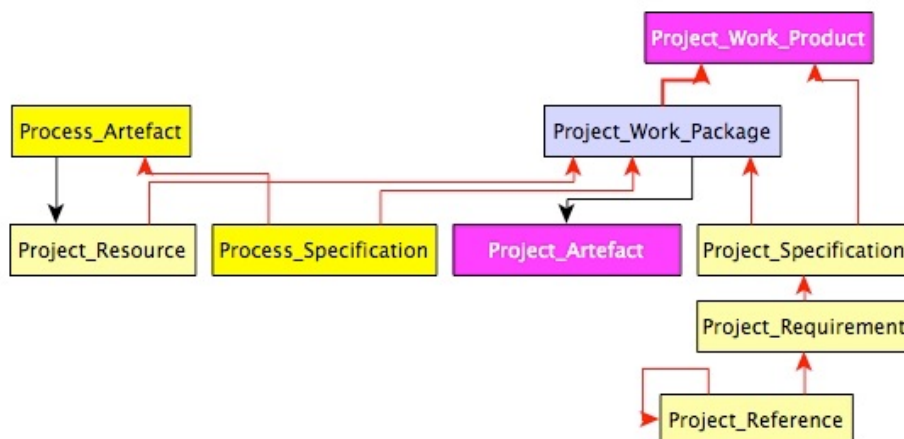
5.3.1. Top level view

The diagram below illustrates the view in GoedelWorks that a System or Product is the result of a Project that is executed by following a specified Process. The different Steps of a Process are often defined to be followed in a relative order (but iteration is allowed). As such a Process can be domain and/or organisation specific but generally independent of a specific Project. In a Project corresponding Works Packages are executed as an instance of the Steps defined in a Process flow.



Top level view of the Systems Grammar in GoedelWorks

The next diagram illustrates an abstraction of the Work Package internal Activities. For the sake of simplicity, we ignored the blue decomposition links as they apply to any Entity and mainly clutter the diagram. It illustrates how the Activities of a Work Package depend on Project Specifications derived by refinement and decomposition from Project Requirements, itself possible pointing to external References.



Top level view of the Work Package inputs and outputs

The WorksPackage also depends on Process Specifications and Project Resources. Part of these Resources are Project template Artefacts (e.g. for the different reports). A Work Package hence produces Project Work Products as well as the associated Project Artefacts, essentially the underlying evidence (often in the form of a document or report) that the Work Package delivered what was specified.

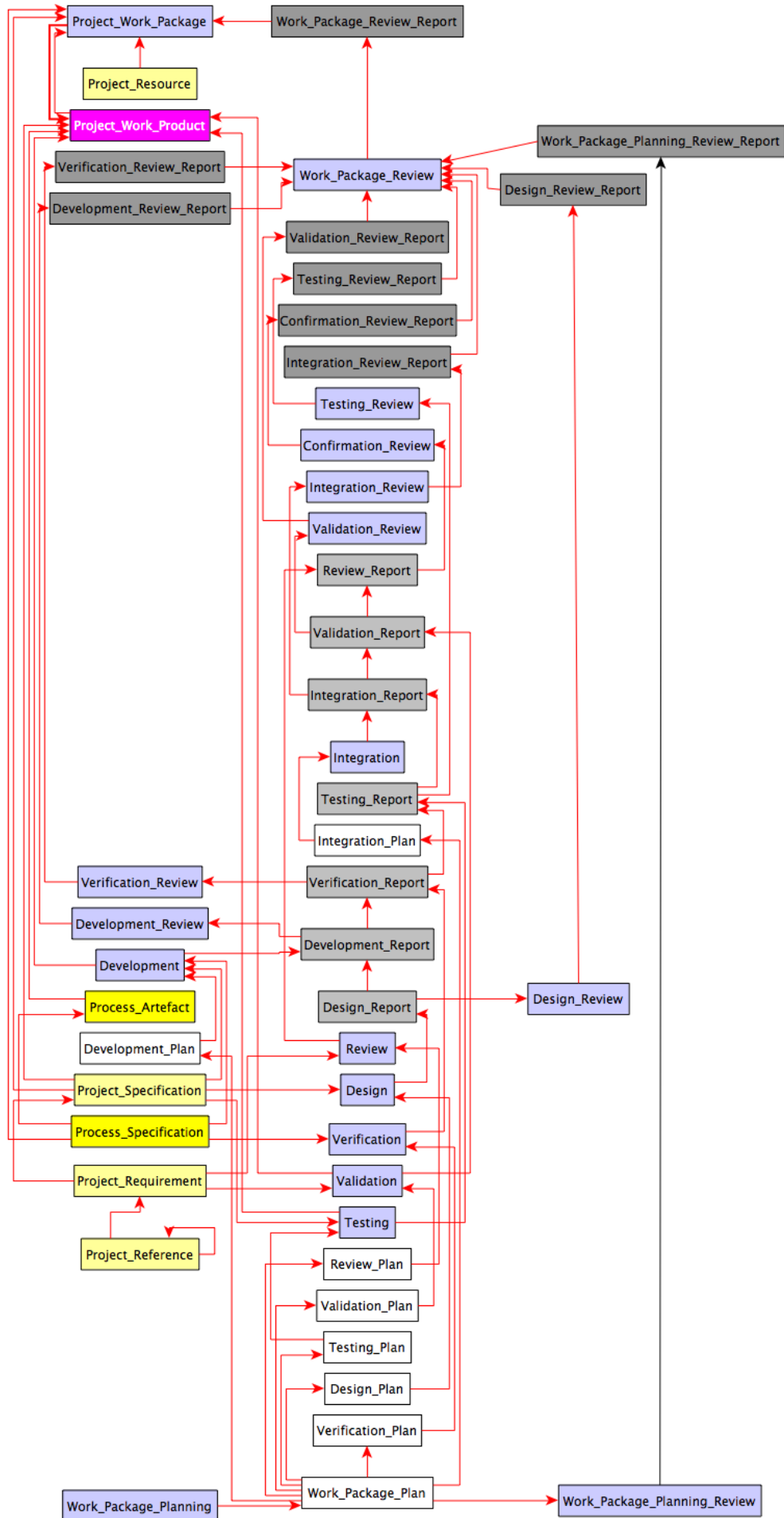
5.3.1. View from inside a Work Package

The next diagram on the following page shows the Activities, Work Products and Artefacts seen from the inside of a Work Package.

Everything starts with Planning, subsequently detailed for each type of Activity (Planning Design, Development, Verification, Testing, Integration, Validation and Review). Each Activity then results in a Report that is reviewed, first at the level of the Activity itself, and then at the level of the Work Package. The latter steps acts as confirmation measures.

The GoedelWorks user has an interest in keeping the general dependency chain in his mind when developing a Project. The Systems Grammar however does not impose a specific order of execution and often a top-down approach will be adopted to meet in the middle with a bottom-up approach. A Project can be finalised if two conditions are met:

- The dependency tree is complete and coherent.
- All Entities in the graph have been finalised and approved according to their dependencies.
- In practice, the latter statements mean that there must be at least a single path from any Requirement to any Work Product and Artefact and that no Entity should be left unconnected. The order of approval can only be top-down. For this reason, when starting a new Project, some planning is helpful. If the initial grouping is well done, less effort will be needed to re-organise the Project Entities to form a logical and coherent set.



5.4. Top level View in a GoedelWorks portal

GoedelWorks can be run from any web browser, although it is recommended to use the latest versions and to use a screen with a high resolution.

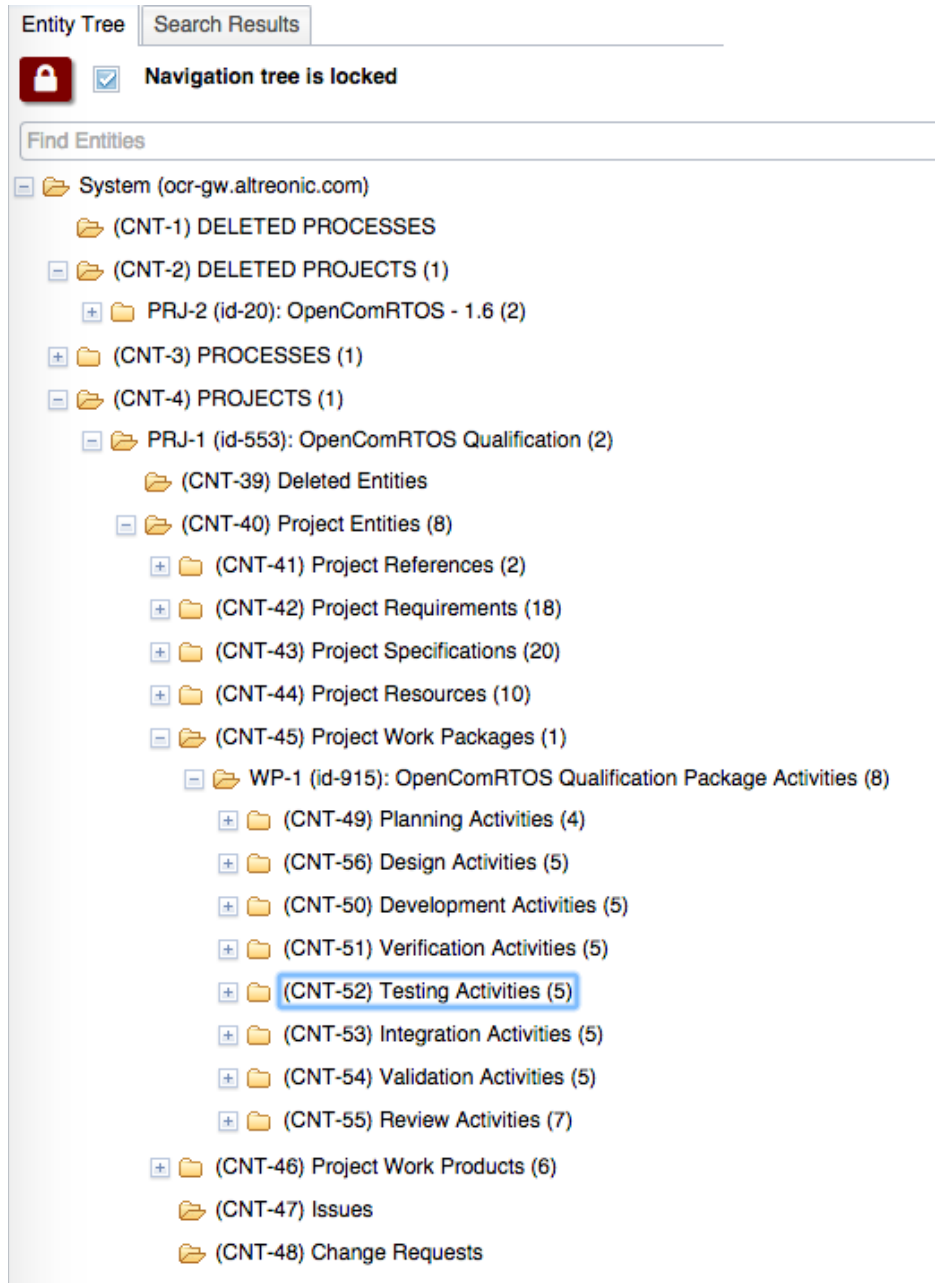
When opening the browser and entering the account name and password, the user will be presented with the main GoedelWorks view. It has the following panes:

- The Entities view on the right.
- The navigation view on the left (called the Entity Tree).
- At the top several tabs are visible, top to bottom:
 - Entities (normally open),
 - Glossary, Query,
 - Administration,
 - Change Log

We will look at these more in detail in the next sections.

5.4.1. Navigation tree view

The navigation tree panel allows the user to quickly navigate his portal. At the top level we see Containers (CNT) for the deleted as well as the active defined Processes and Projects. As mentioned before, in a GoedelWorks portal all Entities live forever, even when deleted and receive a unique identifier when being created. The reason for this decision is to support life-time traceability as well as providing the capability to review past decisions or to recover earlier versions of the entities.



A GoedelWorks screenshot of a navigation tree

Finally, a search function allows to filter a tree and display all the Entities containing the search string. In the example below, the search string was “Port”.

| Entities | Glossary | Query | Administration | Change Log |
|---|----------------|-------|----------------|------------|
| Entity Tree | Search Results | | | |
| WPT-19.4 (id-10): L1_GetPacketFromPort_A (In Work) | | | | |
| WPT-19.13 (id-148): L1_PutPacketToPort_W (In Work) | | | | |
| WPT-19.10 (id-151): L1_PutDataToPort_WT (In Work) | | | | |
| WPT-19.9 (id-152): L1_PutDataToPort_W (In Work) | | | | |
| WPT-19.12 (id-153): L1_PutPacketToPort_NW (In Work) | | | | |
| WPT-19.11 (id-154): L1_PutPacketToPort_A (In Work) | | | | |
| WPT-19.6 (id-155): L1_GetPacketFromPort_W (In Work) | | | | |
| WPT-19.5 (id-156): L1_GetPacketFromPort_NW (In Work) | | | | |
| WPT-19.8 (id-157): L1_PutDataToPort_NW (In Work) | | | | |
| WPT-19.7 (id-158): L1_GetPacketFromPort_WT (In Work) | | | | |
| WPT-19.14 (id-376): L1_PutPacketToPort_WT (In Work) | | | | |
| TST-2.2.1.10 (id-1645): Port (Approved) | | | | |
| TST-2.2.1.10.2 (id-1647): Port_API_Test_002 (Approved) | | | | |
| TST-2.2.1.10.3 (id-1648): Port_API_Test_003 (Approved) | | | | |
| ART-9 (id-1742): WP Planning Review Report (Approved) | | | | |
| SPC-6.2 (id-645): Dedicated input Port (Approved) | | | | |
| WPT-3.3.1.10 (id-1655): Port (Approved) | | | | |
| WPT-3.3.1.10.8 (id-1663): Port_API_Test_008 (Approved) | | | | |
| WPT-3.3.1.10.9 (id-1664): Port_API_Test_009 (Approved) | | | | |
| WPT-3.3.4.2 (id-1944): Waiting Packet in Task-Input-Port (002) (Approved) | | | | |
| WPT-19 (id-6): include/kernel/hubs/L1_hub_port_api.h (In Work) | | | | |
| WPT-19.1 (id-7): L1_GetDataFromPort_NW (In Work) | | | | |
| WPT-19.2 (id-8): L1_GetDataFromPort_W (In Work) | | | | |
| WPT-3.2.2.5 (id-1177): Port (Approved) | | | | |
| WPT-3.3.1.10.1 (id-1656): Port_API_Test_001 (Approved) | | | | |
| WPT-19.3 (id-9): L1_GetDataFromPort_WT (In Work) | | | | |
| TST-2.1.2.5 (id-1032): Port (Approved) | | | | |
| TST-2.1.2.5.1 (id-1136): Get-Interaction waits for complementary interaction in the Port Waiting List (Approved) | | | | |
| TST-2.1.2.5.2 (id-1137): Put-Interaction waits for complementary interaction in the Port Waiting List (Approved) | | | | |
| TST-2.1.2.5.3 (id-1138): Two Get-Interactions wait for complementary interactions in the Port Waiting List (Approved) | | | | |
| TST-2.1.2.5.6 (id-1141): Two Put-Interactions wait for complementary interactions in the Port Waiting List (Approved) | | | | |
| TST-2.2.1.10.1 (id-1646): Port_API_Test_001 (Approved) | | | | |

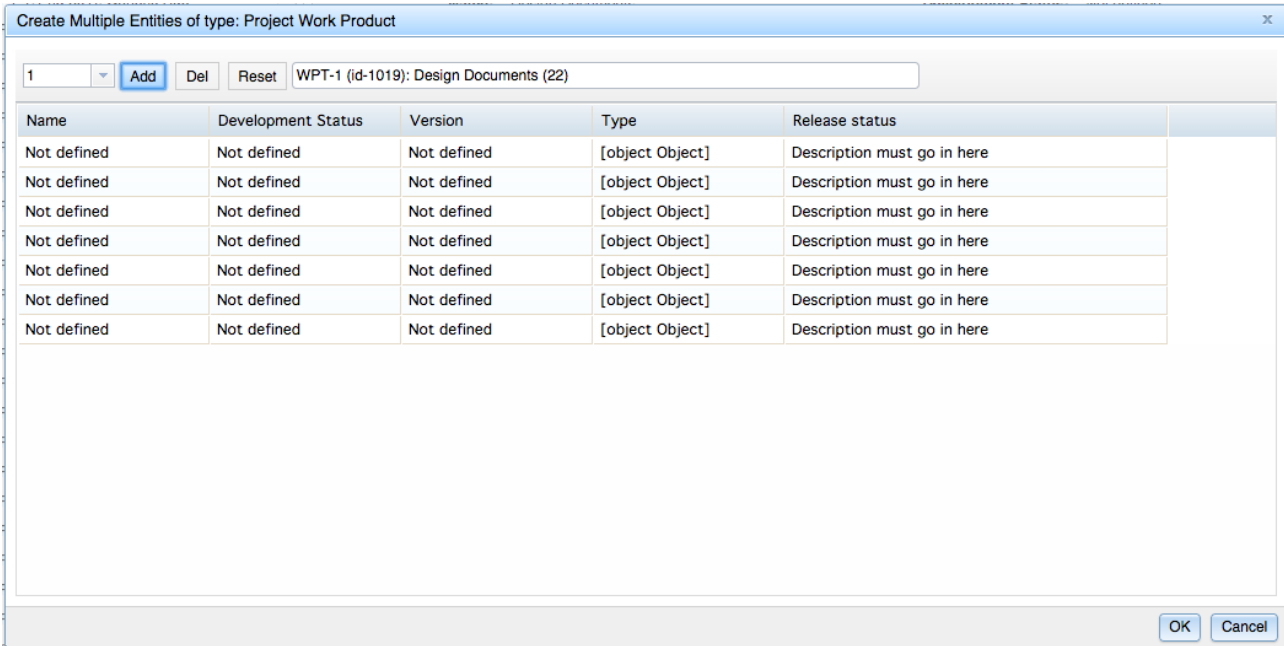
Applying a search filter on the navigation tree

The Entity Containers are sorted according to the Systems Grammar. Note also that Entities receive a navigation tree number. This number will change dynamically when the navigation tree is modified. Hence, the user should not use it to refer to an Entity but use instead the e-id unique identifier. The latter is also reflected in the url displayed in the address bar, for easy and quick reference as well as in the tab of the Entity view. The navigation tree nodes will also display the number of composing Entities.

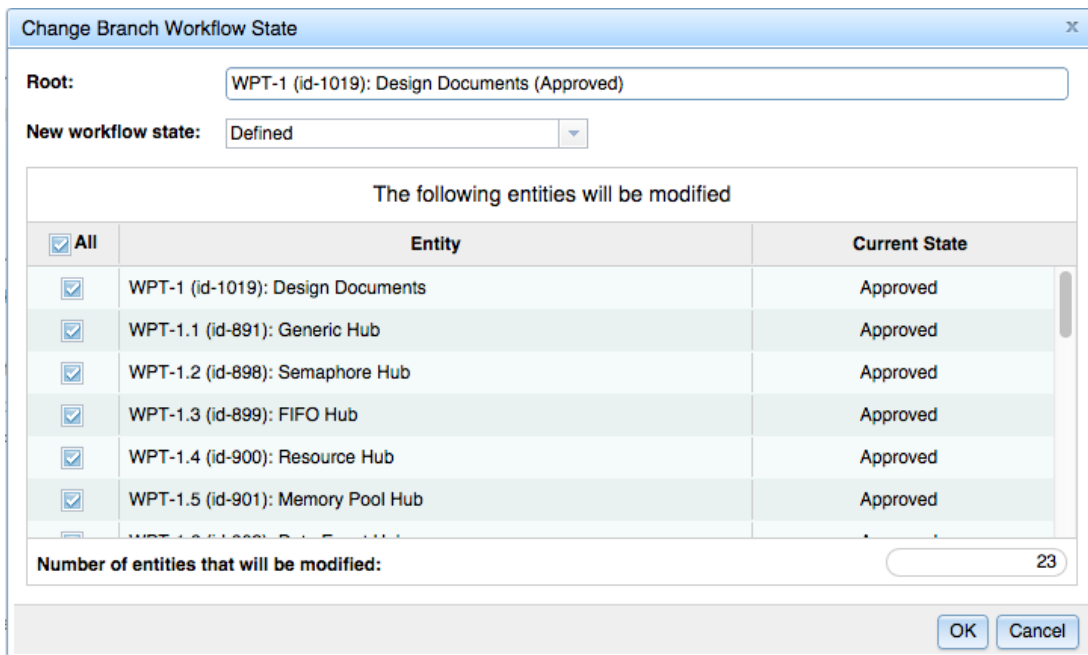
On the navigation tree, multiple operations are possible:

- Locking the tree: this is to avoid that the user accidentally changes the position of an entity in the navigation tree
- Create new Entities or compositional sub-Entities.
- Create a group of Entities (via a pop-up input table)
- Set the Work Flow status for a complete branch.
- Import data from a CSV file (table format).

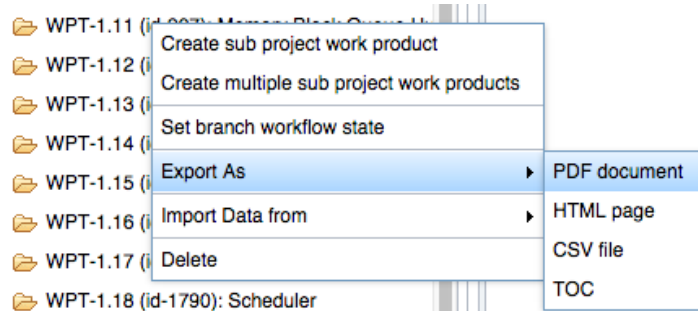
Internal model of a GoedelWorks Package



Tabular creation of Entities as a group



Changing the Work Flow status of a group of Entities on the navigation tree



Exporting navigation tree Entities

5.4.2. The entity pane view

Entity view showing the html editor pane

Most of the time, a GoedelWorks user will be working in the Entities view. In the main view, several operations can be performed, such as

- Editing a Summary text, the description (in html or text),
- Leaving and reviewing comments
- Changing the position in the navigation tree.
- Reviewing the change log.
- Generating a dependency or most often the precedence tree. Such a tree can be in a graphical or textual format or exported as a graphml file. The latter can be very handy to analyse a graph in depth using an external graph editor (like Yed).
- Defining the workflow state (any of: Defined, In Work, Frozen for Approval, Approved). Note that when an Approved Entity is edited again, its state as well as the state of all dependent Entities will be changed to “In Work” again.
- Defining access rights and permissions depending on the role of the user. This can be defined differently for any section of the items in a Navigation tree.

5.4.3. Query capability

Using the Query tab, the user can define and create his queries in his Project. A filter can also be applied to the navigation tree to reduce the number of irrelevant entities.

Trustworthy Systems Engineering with GoedelWorks 3

Of particular interest is the capability to generate a status overview of a Project. A table is then generated showing for example which Specifications are tested, implemented and what their workflow status is. The resulting report can then be exported.

The screenshot displays the GoedelWorks web interface in a Mozilla Firefox browser. The main content area shows a 'Reporting Results - CNT-1716' table. The table has columns for Entity, Tested, Implemented, Status, Children, Tested, and Implemented. The 'Tested' and 'Implemented' columns contain green 'true' or red 'false' values. The 'Status' column shows 'Approved' for all entries. The 'Children' column shows the number of child entities. The 'Tested' and 'Implemented' columns show the number of tested and implemented children, respectively.

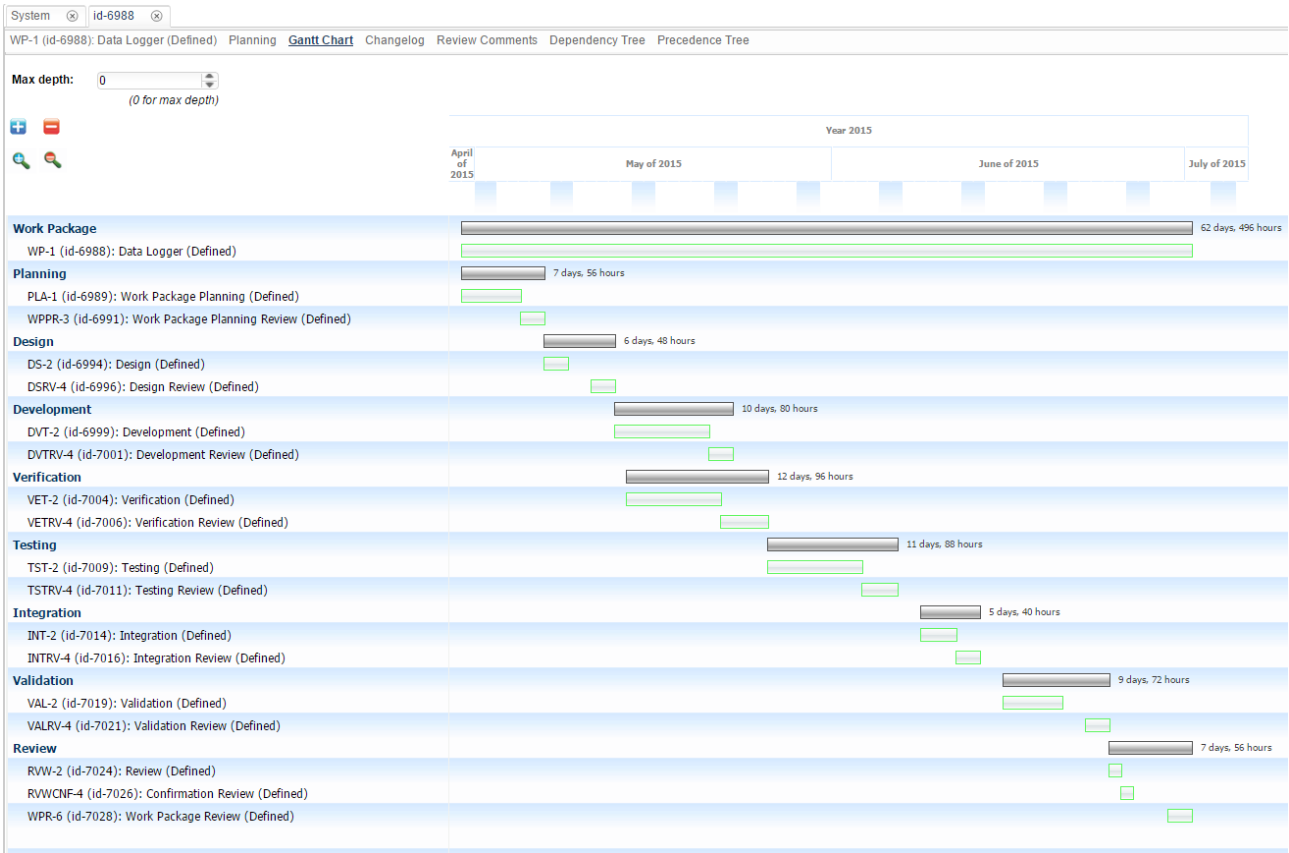
| Entity | Tested | Implemented | Status | Children | Tested | Implemented |
|--|--------|-------------|----------|----------|--------|-------------|
| SPC-1 (id-14833): Packet Switching Architecture (9) | false | false | Approved | 9 | 5 | 5 |
| SPC-2 (id-14839): Kernel Functionality (8) | true | true | Approved | 8 | 8 | 8 |
| SPC-3 (id-15215): Scheduler (3) | true | false | Approved | 3 | 3 | 2 |
| SPC-4 (id-15220): Context Switch (2) | true | true | Approved | 2 | 2 | 2 |
| SPC-5 (id-15210): Communication Layer (6) | false | false | Approved | 6 | 1 | 1 |
| SPC-6 (id-14985): Link Driver Functionality (7) | true | true | Approved | 7 | 7 | 7 |
| SPC-7 (id-14834): Scalability (5) | true | false | Approved | 5 | 0 | 0 |
| SPC-8 (id-14730): Booting | false | false | Approved | 0 | 0 | 0 |
| SPC-9 (id-15680): Performance (5) | false | false | Approved | 5 | 4 | 0 |
| SPC-10 (id-15778): Safety (7) | false | false | Approved | 7 | 3 | 2 |
| SPC-11 (id-15789): Tracer and Debugger (2) | true | false | Approved | 2 | 2 | 0 |
| SPC-12 (id-15770): Written in C | false | false | Approved | 0 | 0 | 0 |
| SPC-13 (id-15052): GoedelWorks | false | false | Approved | 0 | 0 | 0 |
| SPC-14 (id-15649): System Timer (3) | true | false | Approved | 3 | 1 | 1 |
| SPC-15 (id-15579): Memory Management (1) | true | true | Approved | 1 | 1 | 1 |
| SPC-16 (id-14820): Lists (3) | true | false | Approved | 3 | 3 | 0 |
| SPC-17 (id-15717): Hubs (11) | true | false | Approved | 11 | 12 | 0 |
| SPC-18 (id-15131): Task Specifications (18) | true | false | Approved | 18 | 16 | 14 |
| SPC-19 (id-15171): Formal Modelling (3) | false | false | Approved | 3 | 0 | 0 |
| SPC-20 (id-15272): Application Programmer Interface (14) | false | false | Approved | 14 | 13 | 13 |

The interface also includes a navigation tree on the left, a search bar, and a 'Reporting Results' tab. The browser address bar shows the URL '192.168.184.240/#entity=id-9831'. The user is logged in as 'root'.

Generated status report according to the links between the Entities

5.4.4. GANTT chart

GoedelWorks has the capability to generate GANTT charts showing the planned use of Resources and planned execution in time of Work Packages and their Activities. Note however that this function does not calculate any scheduling. It displays the planned execution in time based on the starting and termination times as entered by the portal users.



Gantt chart displaying Activity timelines

5.4.5. Change Log

The screenshot shows the GoedelWorks web interface. On the left is an 'Entity Tree' with a search bar and a list of entities. The selected entity is 'WPT-1.1 (id-891): Generic Hub'. The main content area is titled 'System x id-891 x' and shows a 'Change Log' for this entity. The log entries include:

- Group operation on 01-06-2015-15:28:45 by eric.verhulst (r12738)
- Group operation on 26-02-2015-15:41:36 by eric.verhulst (r12737)
- Group operation on 23-02-2015-10:32:35 by eric.verhulst (r12704)
- Group operation on 11-02-2015-17:50:48 by eric.verhulst (r12632)
- Entity id-891 modified on 10-09-2014-11:52:13 by antonio.ramos (r8453)
- Entity id-891 modified on 07-09-2014-20:46:57 by antonio.ramos (r7866)
- Link created on 07-09-2014-20:46:56 by antonio.ramos (r7865)
- Link created on 07-09-2014-20:36:58 by antonio.ramos (r7862)
- Entity id-891 modified on 07-09-2014-20:35:39 by antonio.ramos (r7861)

The expanded entries show a comparison between the 'After' and 'Before' states of the entity. The 'After' state shows a 'Summary' and 'Description' section for the 'Generic Hub', including a 'Logical View of the Generic Hub' with a detailed description of its interaction with other entities.

Change log entry of an Entity

Changes to entities are recorded at the Entity level (where revert operations are possible) as well as globally displayed at the portal's level. Each entry in the log will record the entities content before and after the change was applied. This can BE very helpful when analysing later on what changes were applied and why they were applied, as well as who was the user making the changes.

As a general principle, no information is ever physically deleted on a portal (the identifier is incremented whenever an Entity is created). This way, all design decisions are recorded for future reference and configuration management is unambiguous. Even deleted entities remain in the system in a dedicated Container.

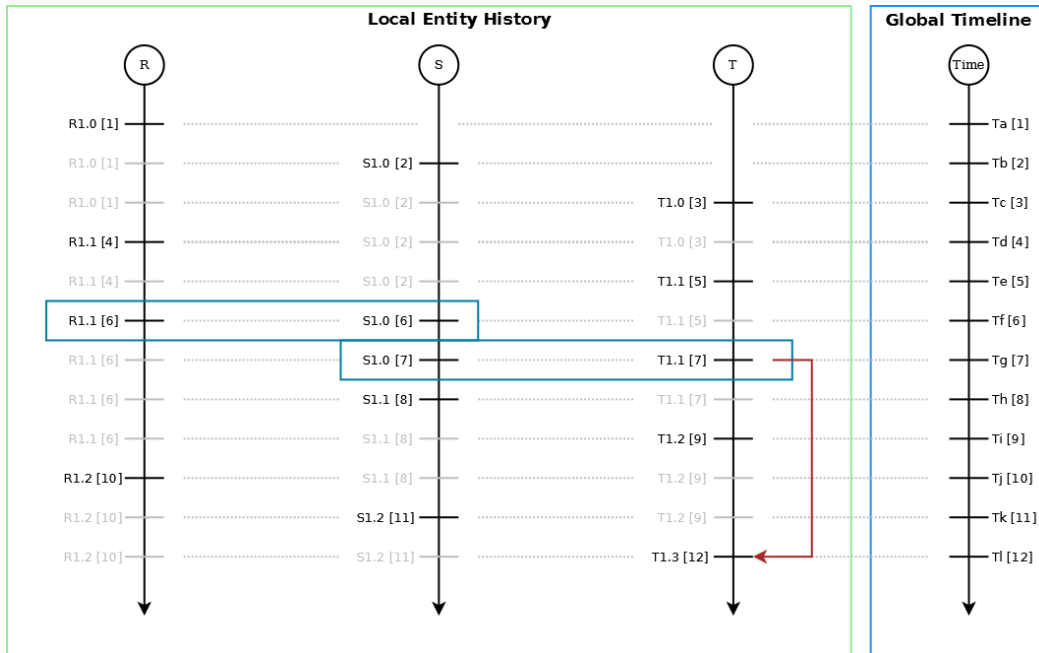
5.4.6. Version and configuration management in GoedelWorks

GoedelWorks provides life time support for version and configuration management for all Entities in a portal. This is achieved by following following main principles:

- No Entity is ever physically deleted. When deleted it is moved to a Deleted Entities Container so that it always can be recovered and reviewed.
- Every Entity receives a unique identifier upon creation. No identifiers are reused.
- Every change is recorded as a unique timestamped event recorded as a version number
- Every Entity has an associated change log

Using these principles it is possible to track the version and changes made and even to revert changes to any Entity, whether Process of Project related. This is illustrated in the following diagram.

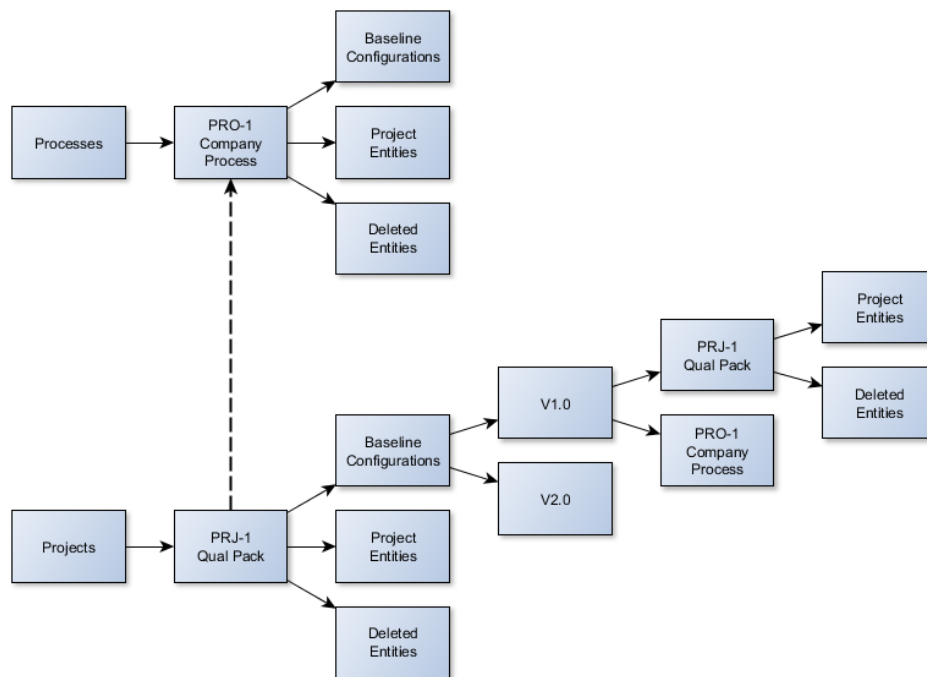
As such, one must remember that contrary to for example a software repository, what constitutes a coherent set of repository items is not just a collection of files in folders but a dependency graph.



Recording of Entity versions using a global unique identifier of changes

Entities can be decomposed and linked to multiple other Entities, Process as well as Project related, which also affects the capability to reverse changes, especially later on. This has an impact on how we can define baseline configurations. The latter must be a coherent set of Entities (in principle each the being the most up to date ones) as well as coherent set of links between the Entities. Therefore, establishing baseline configurations is only allowed at the level of a Process or a Project, as illustrated below:

Using this configuration capability, one can define for example product families.



Configuration management in GoedelWorks

5.4.7. Productivity supporting features

A GoedelWorks portal also has a growing of additional functions that increase the productivity of the user, reducing the number of clicks to achieve the same effect. We list a few:

- “Derive” operation: As Specifications are derived from Requirements, Users can, for instance, create new specifications based on a selected requirement. A new specification is shown in the editor. The initial values of the fields are copied from the selected requirement. When the entity is saved a dependency link is created. From a Specification a Work Product and a Work Package can be generated.
- “Purge”: If an authorised user decides that a deleted entity (or all the deleted entities in a container for deleted entities) should be completely removed from the system, then he is free to wipe them. This process will remove the data on the database as well as any attachments. A message is kept in the log though, that the entity was purged.
- Snapshot feature: the user can create a snapshot of a Project/Process. A snapshot is a self-contained zip file containing the PDF/HTML representation of the entity and copies of any attachments and version control system items (in case of folders, the complete content of the folder is copied, recursively)
- Fill reports: the user can ask the system to pre-fill the validation and verification reports with a template. He does this by selecting on the navigation tree a Requirement (of a Requirements Container. The Validation report will then be filled with the template (for example, the list of all Requirements. An example:

REQ-1 (id-1) Vehicle Speed (Approved)

- Derived specifications:

SPC-1 (id-2) Max speed (Approved)

SPC-2 (id-3) Torque (Approved)

- Justification: to be filled in.

- Status Report: User can check at any moment the implementation and testing status of any project.
- Edit entities in a table format
- Import of version control system items as GW Entities Import of version control system items as GW entities Integration with version control systems
- GoedelWorks provides the user with an interface to exchange data with different version control systems. The interface is achieved through Process Artefacts and/or Project Work Products. Content is copied from the repository to GoedelWorks and kept local. Data is only updated on user request. The user can also renew the contents of the file in the version control system by using GoedelWorks to submit a new version.
- Integration with software items: Project Work Products can be associated with software contents. A proper source code editor is available and it supports many programming languages. Contents can be taken from version control systems.

5.4.8. Administration

The administration panel is reserved for users with the necessary administration rights. Without going into details, it offers several options to manage the portal and its users. We list them briefly:

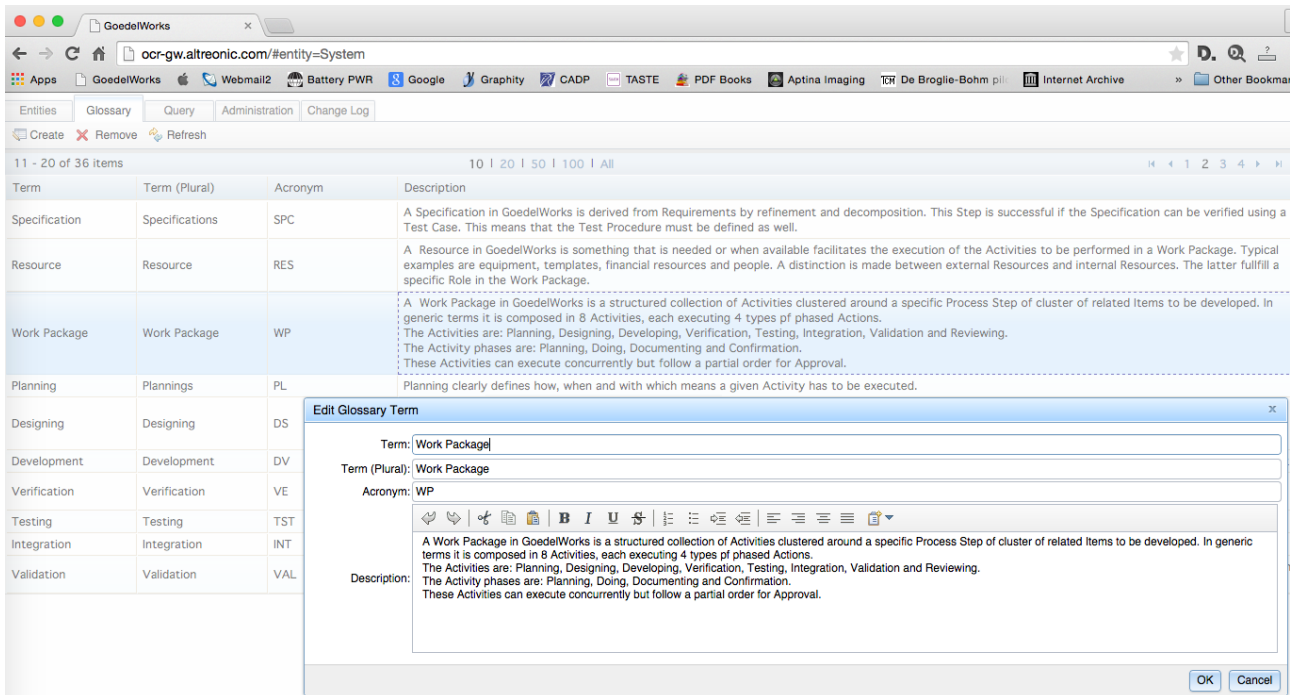
- Changing the welcome message and welcome image.
- Import and exporting Processes and Projects or any Entity.
- Manage the uploaded files (attachments and images used in descriptive texts).

Trustworthy Systems Engineering with GoedelWorks 3

- Setting default permissions for each user group (read, write and export rights).
- SMTP setting for notification mails;
- Viewing the error log;
- User management: personal contact details, group membership and session log.
- Managing and creating user groups;
- Cleaning up locked entities.

5.4.9. Glossary

The user can define his own glossary, currently regrouping the terms used at the level of the portal.



The screenshot shows a web browser window with the URL ocr-gw.altreonic.com/#entity=System. The page displays a 'Glossary' section with a table of terms. An 'Edit Glossary Term' dialog box is open, showing the details for the 'Work Package' term.

| Term | Term (Plural) | Acronym | Description |
|---------------|----------------|---------|--|
| Specification | Specifications | SPC | A Specification in GoedelWorks is derived from Requirements by refinement and decomposition. This Step is successful if the Specification can be verified using a Test Case. This means that the Test Procedure must be defined as well. |
| Resource | Resource | RES | A Resource in GoedelWorks is something that is needed or when available facilitates the execution of the Activities to be performed in a Work Package. Typical examples are equipment, templates, financial resources and people. A distinction is made between external Resources and internal Resources. The latter fulfill a specific Role in the Work Package. |
| Work Package | Work Package | WP | A Work Package in GoedelWorks is a structured collection of Activities clustered around a specific Process Step of cluster of related Items to be developed. In generic terms it is composed in 8 Activities, each executing 4 types of phased Actions. The Activities are: Planning, Designing, Developing, Verification, Testing, Integration, Validation and Reviewing. The Activity phases are: Planning, Doing, Documenting and Confirmation. These Activities can execute concurrently but follow a partial order for Approval. |
| Planning | Plannings | PL | Planning clearly defines how, when and with which means a given Activity has to be executed. |
| Designing | Designing | DS | |
| Development | Development | DV | |
| Verification | Verification | VE | |
| Testing | Testing | TST | |
| Integration | Integration | INT | |
| Validation | Validation | VAL | |

Edit Glossary Term

Term:

Term (Plural):

Acronym:

Description:

OK Cancel

Glossary screenshot

5.5. A project example

5.5.1. The OpenComRTOS Qualification project

As an example we concisely present the Qualification Project of the OpenComRTOS kernel. This concerns only the RTOS kernel of OpenComRTOS Designer, a modelling and programming environment for programming parallel and concurrent applications, running on a system with potentially multiple processing nodes. The design was originally done using formal methods.

At the time of starting the Project, the development was already completed (having reached v.1.6). The RTOS was already in use as well as ported to several targets. The Qualification Project's goal is to develop all the evidence to support the qualification of the RTOS kernel itself. This evidence can then be used to assess the certification for different standards (DO-178, IEC-26262, etc.) as the approach taken is generic and independent of a specific domain. This is a valuable position as OpenComRTOS is a tool that can be used across different domains without requiring changes.

5.5.2. Planning the Qualification Project

In this project the starting point was the existing source code v.1.6 of the OpenComRTOS kernel (with a Processor specific part being the Freescale powerPC specific code). This was initially imported and updated as the Project progressed. As the original kernel was formally developed, the available project data in the form a book is referenced too. Furthermore, we list all the Resources that are needed. These include tools such as the formal modelling tools, the development software, version management software, test chains, etc. See the following figure displaying the PowerPC tools.

The screenshot displays the GoedelWorks 3 interface. On the left, the 'Entity Tree' shows a hierarchical structure of project entities. The selected entity is 'WPPD-2 (id-917): Work Package Plan'. The right pane shows the 'Content' tab for this entity. The 'Summary' section includes the following information:

| | | | |
|----------|-------------------|---------------------|-------------|
| Name: | Work Package Plan | Development Status: | Implemented |
| Version: | 1.0 | Artefact type: | Document |

The 'Description' section contains the following text:

1. Scope

The qualification is restricted to the following parts of OpenComRTOS Designer:

1. The RTOS Kernel source code (v.1.6) and resulting binary libraries for a specific target processor.
2. The Kernel/hardware HAL for a specific target processor.
3. This release concerns the RTOS v. 1.6 for the Freescale 7448 PowerPC.

Rationale: following items from OpenComRTS Designer were excluded for following reasons:

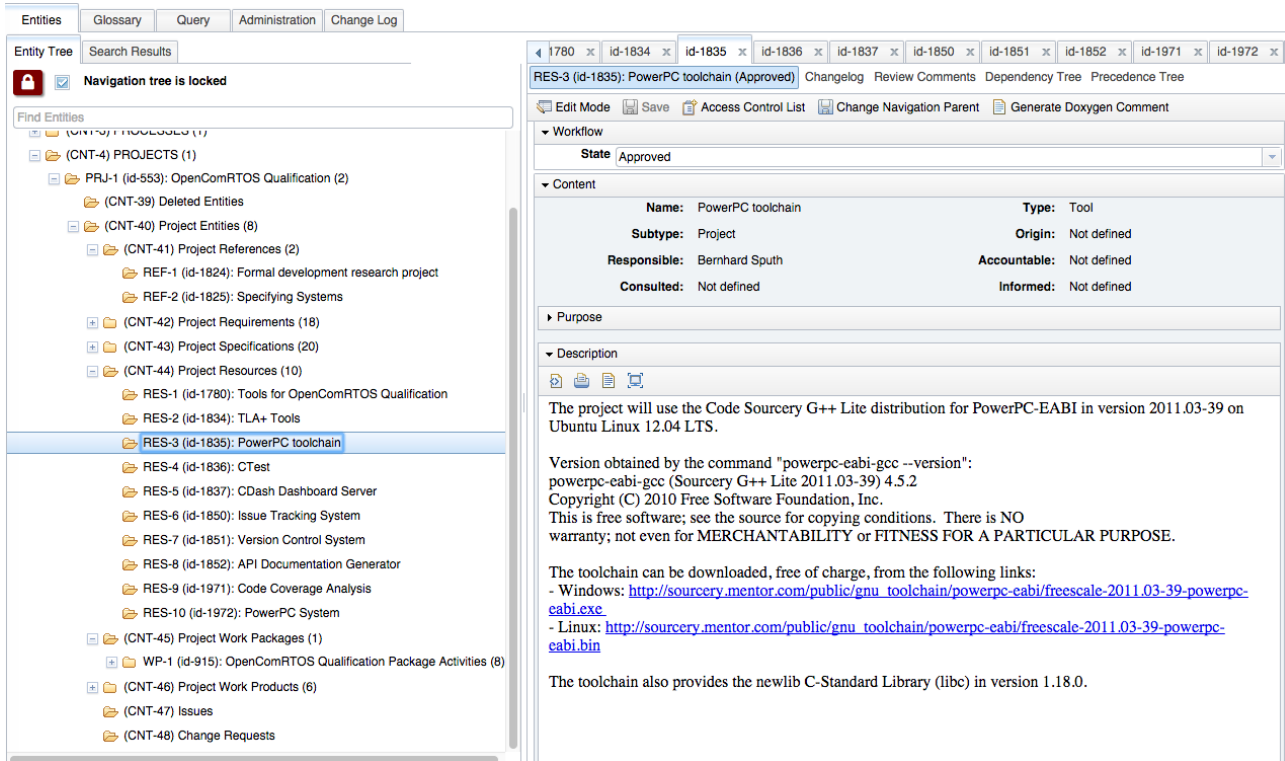
1. Visual Designer with source code generator: this is a visual modelling tool used as a productivity and documentation aid for the application developer. The software engineer can write the generated code without using the tools. Hence the tools are not needed for application development.
2. Event Tracer: this tool allows to visualise the execution trace of an application and is a supporting tool for finding errors and profile the application. Hence the tools are not needed for application development.
3. Task Level debugger: this tool allows to inspect an application at runtime and is a supporting tool for finding errors and profile the application. Hence the tools are not needed for application development.
4. I/O libraries: these tools are non-critical utility libraries and application specific.
5. Board Support Package: this is a target Board specific component that is to be redeveloped for every target system and hence is in the realm of custom engineering.

2. Objective

The objectives of this Qualification Package are:

Screenshot of the Work Package Plan Entity

Next a decision was to be made on how to structure the Work Plan. Should we use several Work Packages? Given that the input was existing and well tested source code, the development of the Qualification Package came down to filling in the gaps, starting with the existing Requirements and Specifications and then making sure that the traceability graph from Requirements to all Works Products is complete. This includes the default and standardised Work Package Activities. It starts by defining a Work Package Plan whereby the scope of the Project is clearly defined.



Screenshot of the Resources used in the Qualification Package Project

The execution of the Qualification essentially proceeded as follows:

All references, resources, source code were imported via the SVN repository

The general Requirements are refined into Specifications, grouped into several classes:

- Functional Specifications: these are related to the behaviour of the RTOS in terms of services offered to the application.
- Non-functional Specifications: these are related to derived properties (e.g. code-size).
- Interface Specifications: these are actually developed during the Design Activity.
- Implementation Specifications: these are actually developed during the Design Activity.

Note that an alternative Project organisation could have been to have a dedicated Work Package related to the Design only. This is certainly a must for larger projects but it also indicates that the engineering organisation has the freedom to decompose the Project in less or more Work Packages.

5.5.3. The Planning Activities

The Work Package Activities are planned at the level of the Work Package itself as well as at the level of the Activities. It is clear that at the WP level, the focus is on coordinating the work and the review activities.

5.5.4. Design Activities

The Design Activities produce a detailed Design Report that analyses the Requirements and Specifications and the defines an architecture and detailed implementation that the Development Activity can use.

The screenshot displays the GoedelWorks web application interface. The browser address bar shows the URL `ooc-gw.altreonic.com/#entity=id-891`. The main content area is titled "WPT-1.1 (id-891): Generic Hub (Approved)". It includes a "Workflow" section with "State: Approved" and a "Content" section with the following details:

- Name: Generic Hub
- Version: 1.0
- Type: Model
- Model type: Architectural
- Development Status: Ready for Design

Below the content is a "Description" section containing a diagram labeled "Figure 1. Generic Hub". The diagram shows a central circle labeled "Generic Hub (N-N)" connected to two vertical bars labeled "PWL" and "GWL". Above the central circle are five stacked boxes: "Hub State", "Control Operation", "Update Operation", "Synchronisation Operation", and "Synchronisation Predicate".

Below the diagram is the section "Generic Hub Behavior" with the text: "Interactions can be seen as a three step process: starting the interaction, processing the request, concluding the interaction." and a link for "Starting interactions: [SPC Service Request Types](#)".

Screenshot of the Generic Hub Work Product Entity

5.5.5. Some statistics

To illustrate how extensive the evidence can be, we provide following statistical data:

- Number of lines of code (C and some assembler): 6500
- Number of Entities: 1320
- Number of Links: 1697 dependency links and 1221 structural links.
- Project requirements: 18
- Project Specifications: 311
- Project Work Products: 541
- Project Tests: 372
- Number of pages in pdf snapshot: 1508
- Attachements: 177

6. Safety standards awareness in GoedelWorks

One of the issues in Systems engineering is that when Certification is a Requirement, many standards can be applicable. Moreover, legal Requirements will differ from country to country and depend on the application domain. In addition, standards are often either prescriptive but often outdated with respect to technology, either goal oriented but leaving it up to the engineering organisation to follow a certifiable Process. This is a bit strange as we have shown that Systems engineering is very universal.

The current situation is due to historical reasons and the state of the practice, including legal preoccupations in relationship to liability issues. For the purpose of this discussion we will limit ourselves to certifiable safety standards, applicable in the context of the automotive and machinery industry. These were introduced when the complexity increase resulting from the introduction of programmable electronics forced to think more systematically about how Systems with safety risks needed to be developed.

6.1. Safety standards for embedded reprogrammable electronics

The root of these standards is IEC-61508. It covers the complete **safety life cycle**, but needs interpretation for a sector specific application. It originated in the Process control industry sector.

The safety life cycle has 16 phases which can be roughly divided into three groups: analysis, realisation (development) and operation. All phases are concerned with the safe functioning of the System. Composed of 7 Parts, Parts 1-3 contain the Requirements of the standard (normative), while 4-7 are guidelines and examples for development and thus informative.

Central to the standard are the concepts of **risk** and **safety function**. The risk is seen as a statistical function of the hazardous event and the event consequence severity. The risk is reduced to a tolerable level by applying safety functions that may consist of electric, electronic or embedded software or other technologies. While other technologies may be used in reducing the risk, IEC 61508 only considers the electric, electronic and embedded software.

From this standard, extensions were developed for specific segments. For example:

- Automotive: coding standards like MISRA and later on ISO-26262
- Avionics: DO-178B/C and DO-254
- Railway: EN-50128
- Process Industry: IEC-61511
- Nuclear Power Plants: IEC 61513
- Machinery: IEC 62061

6.2. ASIL: A safety engineering process focused around ISO-26262

While in principle GoedelWorks can support any type of Project and Process, its meta-Model was tuned for Systems engineering Projects with a particular emphasis on safety critical Processes whereby the final system or product will have to pass qualification or certification. As such, it has been used in the context of the automotive domain, avionics, railway and space. Organisations can add and develop their own Processes as well as import them (when made available in a proper format).

The first Process that was imported is the ASIL Process. The ASIL Process is a Process based on several safety engineering standards, but with a focus on the automotive and machinery domain. It was developed by a consortium of Flanders Drive members and combines elements from IEC 61508, IEC 62061, ISO DIS 26262, ISO 13849, ISO DIS 25119 and ISO 15998. These elements were obtained by dissecting these standards in

semi-atomic requirement statements and combining them in a iterative V Process Model. It was enhanced with templates for the Work Products and domain specific guidelines.

In total the ASIL Process identified about 3800 semi-atomic requirement statements and about 100 Process Work Products, although this is a still an on-going effort. The ASIL Process also identifies 3 Process domains:

- Organisational Processes.
- Safety engineering and development Processes.
- Supportive Processes.

The ASIL Process Flow was imported by mapping all ASIL Entities onto GoedelWorks Entities and adding the missing Entities, association and structural links. Examples are:

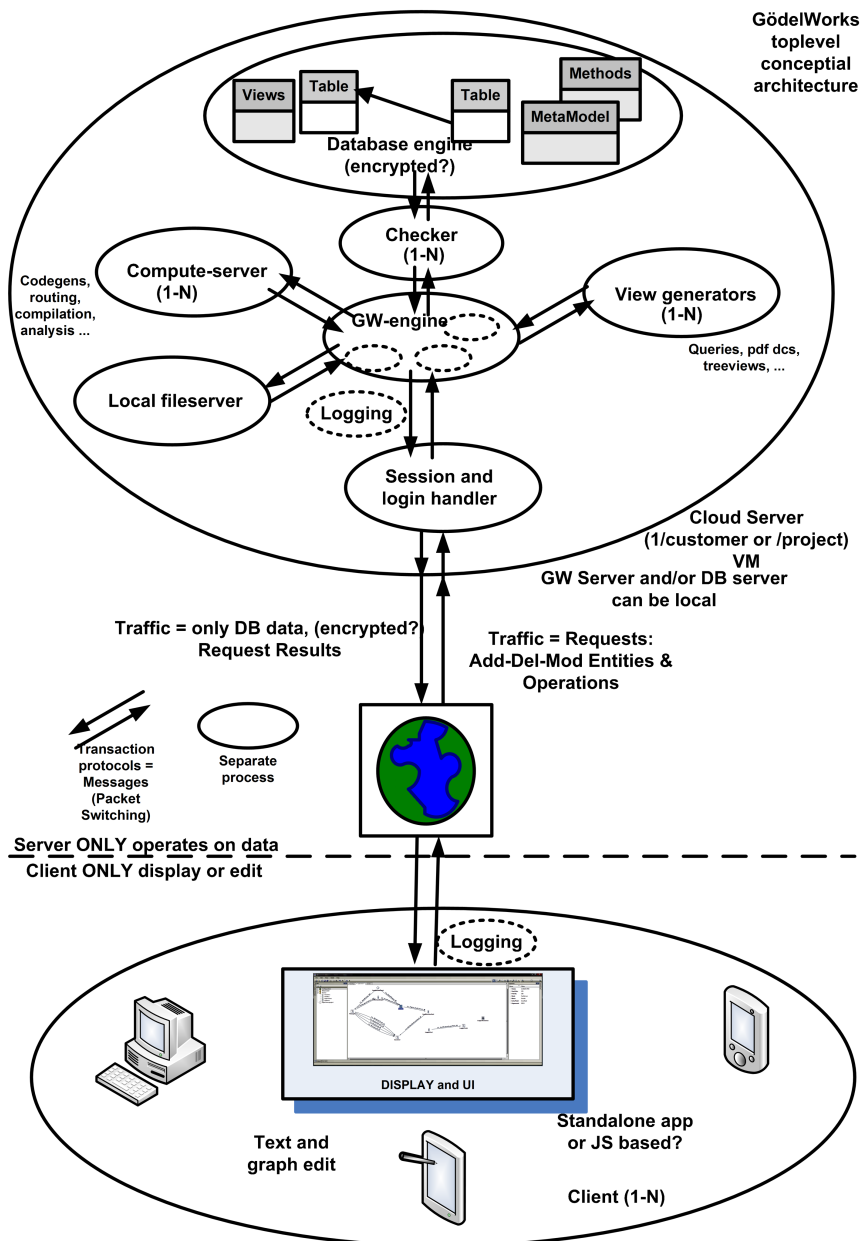
- Upon Work Package creation, a set of Tasks is added and structurally linked. The user can then add more Tasks as needed.
- Specification and Requirements Entities are added for the Work Products.

For this reason the imported ASIL still needs to be completed to create an organisation or Project specific Process. It is also likely that organisation specific Processes will need to be added. This is made possible since each Entity in GoedelWorks can be directly edited in the portal.

6.3. Certification, qualification after validation

If a Systems Engineering Project reaches the Validation stage and the System is approved, why is Certification still needed? Certification is first of all a legal Requirement. By definition, it is not good practice if certification would be done by the same organisation that executed the Project. Even the best organisation and best possible Process is still executed by humans and the goal of the Systems Engineering Process is to maximise success in a cost-efficient way. Therefore, Certification has to be seen as an additional re-validation step executed by an external auditing organisation. Certification does not attempt to discredit the Project's results, it seeks confirmation that the Requirements, at least those relevant for the Certification, were met and that there is evidence that everything was done that needed to be done. Therefore, Certification is often based on examining and reviewing the "Artefacts", essentially the trail of evidence generated during the Project, but it will also execute spot checks and anything else that might be needed.

Producing the evidence is something that must be done during the Project when the work is actually done. Examples are test reports, issue tracking records, meeting reports, etc. This work is what often scares companies as it doesn't come for free. Following a Process cost extra time and Resources, but has also benefits.



General set-up of the GoedelWorks environment.

The Project will become more predictable and traceable, errors are detected in an early stage (when they cost less to correct) and when considering life-cycle costs, it might turn out to be cost-efficient, especially if support and maintenance costs are included. In the worst case, a serious issue can be discovered when the System is in use and operational. Recalls to fix these issues can be very costly, not only financially but also in reputation damage, etc. Therefore, Certification is a must but there is every interest to reduce the cost.

Note also that the term Qualification is also often used. It is very much like Certification with the difference that there are often there no legal consequences and it is the term often used to indicate fitness for use of a component or tool. Hence, it is often the integrator that is responsible for assuring that a tool or component is of sufficient quality and can be trusted to be used in the context of his Project.

The GoedelWorks environment contributes to this on several levels by automating the engineering Process:

- The organisation uses a standards-aware Process.
- The approval Process reduces rework and double work.
- The certification artefacts are generated during development.
- Organisations can "pre-certify" by following the Process.

The cost of running a Systems engineering Project will also be reduced because the GoedelWorks server keeps track in a central repository of all changes and dependencies. In addition, people all over the world can collaborate because all data is centrally located and edited.

6.4. Organisation specific instances of GoedelWorks

The GoedelWorks environment is a very flexible tool for Systems engineering. It provides the following functionalities for a distributed team:

- Project support from Requirements unto release for production (no need to use the ASIL Project Flow).
- Process support from Requirements unto release for production (with ASIL Project Flow).
- Customisation and completion of the ASIL Flow, depending on the industry.
- Developing new, domain specific Process Flows.
- Adding organisation specific Processes.
- Creating a snapshot of the Project in html or pdf format, i.e. export the project or process contents in HTML or PDF formats, as well as other formats such as a self-contained archive (with attachments and source code repositories).
- Generating dependency and precedence trees.
- Import and exporting projects, processes or any other entity type. Entities from a given portal can be reused on another portal through this interface.
- User management.
- Knowledge management.

7. References

- [1] Eric Verhulst, Bernhard Sputh, Jose Luis de la Vara, Vincenzo de Florio, ARRL: a novel criterion for Composable Safety and Systems Engineering. SafeComp/SASSUR workshop. Toulouse, September 2013.
- [2] Eric Verhulst, Bernhard Sputh, Jose Luis de la Vara, Vincenzo de Florio. From Safety Integrity Level to Assured Reliability and Resilience Level for composable safety critical systems, ICSSEA, Paris, Nov. 2013.
- [3] Eric Verhulst, Bernhard Sputh. ARRL, a criterion for compositional safety and systems engineering. A normative approach to specifying components. IEEE ISRRE2013, Pasadena, November 2013.
- [4] <http://www.iec.ch/functionalsafety/>. Functional safety of electrical / electronic / programmable electronic safety-related systems (IEC 61508) (2005)
- [5] http://www.altreonic.com/sites/default/files/Altreonic_ARRL_DRAFT_WIP011113.pdf. From Safety Integrity Level to Assured Reliability and Resilience Level for Compositional Safety Critical Systems (internal white paper)
- [6] <http://www.rtca.org>
- [7] IEC: ISO: International Standard Road vehicles - Functional safety - ISO/DIS 26262 (2011)
- [8] BAA: Aircraft Crashes Record Office. <http://baaa-acro.com/index.html> (2013)
- [9] World Health Organisation: WHO global status report on road safety 2013: supporting a decade of action. Technical Report (2013)
- [10] Antifragile. Things that gain from disorder. Nassim Nicholas Taleb. Random House (Nov. 2012)
- [11] <http://ec.europa.eu/environment/noise/home.htm>

8. ANNEXES

8.1. Entities supported in GoedelWorks 3

Entities and their acronyms in GoedelWorks

| | |
|-------|-------------------------------------|
| SYS | System |
| PRO | Process |
| PRJ | Project |
| REF | Project Reference |
| REQ | Project Requirement |
| SPC | Project Specification |
| RES | Project Resource |
| WP | Project Work Package |
| WPT | Project Work Product |
| ISS | Issue |
| CHR | Change Request |
| PLA | Work Package Planning |
| WPPD | Work Package Plan |
| WPPR | Work Package Planning Review |
| WPPRR | Work Package Planning Review Report |
| WPR | Work Package Review |
| DSP | Design Plan |
| DS | Design |
| DSRP | Design Report |
| DSRV | Design Review |
| DSRR | Design Review Report |
| DVTP | Development Plan |
| DVT | Development |
| DVTRP | Development Report |
| DVTRV | Development Review |
| DVTRR | Development Review Report |
| VETP | Verification Plan |
| VET | Verification |
| VETRP | Verification Report |
| VETRV | Verification Review |
| VETRR | Verification Review Report |
| TSTP | Testing Plan |
| TST | Testing |
| TSTRP | Testing Report |
| TSTRV | Testing Review |
| TSTRR | Testing Review Report |

| | |
|---------------|-----------------------------------|
| INTP | Integration Plan |
| INT | Integration |
| INTRP | Integration Report |
| INTRV | Integration Review |
| INTRR | Integration Review Report |
| VALP | Validation Plan |
| VAL | Validation |
| VALRP | Validation Report |
| VALRV | Validation Review |
| VALRR | Validation Review Report |
| RVWP | Review Plan |
| RVW | Review |
| RVWRP | Review Report |
| RVWCNF | Confirmation Review |
| RVWCRR | Confirmation Review Report |
| WPRR | Work Package Review Report |
| REF | Process Reference |
| REQ | Process Requirement |
| SPC | Process Specification |
| RES | Process Resource |
| STP | Process Step |
| ART | Process Artefact |

8.2. Entities defined in the ASIL Process

This annex lists an extract of the key Entities identified by the ASIL project in the analysed safety standards. This is used as an example Process flow whereby a user can define his own Process or import another One. For more detailed information, please contact us at: [@altreonic.com](mailto:goedelseries)

1. ASIL Roles

- 1.1. Configuration manager
- 1.2. Hardware architect
- 1.3. Hardware developer
- 1.4. Hardware integrator
- 1.5. Project manager
- 1.6. Quality Assurance manager
- 1.7. Safety manager
- 1.8. Software architect
- 1.9. Software developer
- 1.10. Software integrator
- 1.11. Stakeholder, defined by impact analysis
- 1.12. Supply manager
- 1.13. System Architect
- 1.14. System Integrator
- 1.15. Test engineer
- 1.16. Validation engineer
- 1.17. Verification engineer

2. ASIL Work Products grouped into categories

- 2.1. Change management (3)
- 2.2. Configuration management (2)
- 2.3. Decommission and disposal (2)
- 2.4. Documentation (2)
- 2.5. Hardware related (10)
- 2.6. Installation and commissioning (8)
- 2.7. Integration and testing (6)
- 2.8. Project planning (2)
- 2.9. Production (4)
- 2.10. Qualification (4)
- 2.11. Safety related (18)
- 2.12. Software related (18)
- 2.13. Supplier related (12)
- 2.14. System related (4)
- 2.15. Validation (2)
- 2.16. Verification (3)

- 3. ASIL Work Packages grouped into related steps**
- 3.1. Organisational Processes (19)
- 3.2. Supporting Processes (75)
- 3.3. Document management (3)
- 3.4. Supply agreement management (24)
- 3.5. Configuration management (18)
- 3.6. Change management (9)
- 3.7. Verification (8)
- 3.8. Confirmation measures (6)
- 3.9. SIMPLAR Constraints (19)
- 3.10. Decomposition of safety integrity levels (1)
- 3.11. Criteria for coexistence (1)
- 3.12. Safety analyses (8)
- 3.13. Analysis of dependent failures (3)
- 3.14. Qualification of hardware components (3)
- 3.15. Qualification of software components (3)
- 3.16. Qualification of software tools (6)
- 3.17. Proven in use argument (5)
- 3.18. System Safety & Engineering Development Processes (261)
- 3.19. Definition scope of Project (15)
- 3.20. Definition methodology of HARA and safety goals (6)
- 3.21. Execution of HARA and safety goals (37)
- 3.22. Functional safety concept (4)
- 3.23. System development planning (4)
- 3.24. System design (20)
- 3.25. Hardware development planning (4)
- 3.26. Hardware design and development (18)
- 3.27. Software development planning (4)
- 3.28. Software design and development (11)
- 3.29. Software unit testing (4)
- 3.30. Software integration and testing (5)
- 3.31. Hardware integration and testing (5)
- 3.32. Hardware software integration and testing (4)
- 3.33. System and vehicle/machine integration and testing (4)
- 3.34. Safety validation (5)
- 3.35. Prototype installation (4)
- 3.36. Production (6)
- 3.37. Installation and commissioning (3)
- 3.38. Operation, maintenance and repair (10)
- 3.39. Decommissioning or disposal (7)

Acknowledgements

While GoedelWorks is a development of Altreonic, part of the theoretical work was done in the following projects:

- EVOLVE (Evolutionary Validation, Verification and Certification). This is an EU ITEA project executed from 2007 till 2011 with Open License Society (Altreonic's R&D partner).
- ASIL (Automotive Safety Integrity Level). This is a Flanders' Drive project executed from 2008 till 2011, with support from IWT, the Flemish Institute of Science and Technology.
- OPENCROSS (Open Platform for Evolutionary Certification Of Safety-critical Systems (automotive, railway, avionics). An FP7 IP EU project that researches way to reduce the certification costs across different domains (mainly automotive, aviation and railway)