Unified semantics for trustworthy systems engineering

This paper describes the theoretical principles and associated meta-model of a unified trustworthy systems engineering approach developed by Altreonic. Guiding principles are ``unified semantics'' and ``interacting entities''. Proof of concept projects have shown that the approach is valid for any type of processes, also non technical engineering ones. The meta-model was used as a guideline to develop the GoedelWorks internet based platform supporting the process view (focused on requirements engineering), the modelling process view as well as the workplan development view. Of particular interest is the integration of an automotive safety engineering process that was developed to cover multiple safety standards.

1 Introduction

Systems Engineering (SE) is considered to be the process that transforms a need into a working system. Discovering what the real need is often already a challenge as it is the result of the interaction of many stakeholders, each of them expressing their ``requirements'' in the language specific to their domain of expertise. None of the stakeholders will have a complete view outside his domain of interest and often will not be able to imagine what will be the final system. The problem is partly due to the fact that we use natural language and that our domain of expertise is always limited. In order to overcome these obstacles, formalization is required. The meta-model we developed is an attempt to achieve this in the domain of SE. In terms of the guiding principles, unified semantics comes down to defining univocal and orthogonal concepts. The interacting entities paradigm defines how these concepts are linked. The result is called a ``systems grammar'' in analogy with the rules of language that allow us to construct meaningful sentences (an entity), a chapter (a system) or a book (a system of systems). It defines the SE terms (standing for conceptual entities) and the rules on how to combine the conceptual entities in the right way to obtain a (trustworthy) system. What complicates the matter is that a system in the end is defined not only by its final purpose but also by its history (e.g. precursors), by the process that was followed to develop it and by the way its composing entities were selected and put together. Each corresponds to a different view and it is the combination of these views that result in a unique system, even if a different system could have been developed on the basis of the stated requirements.

Note that a process can also be considered as a system. The main difference with a system that is being developed is that the composing entities and they way they interact are different. For example humans will communicate and execute a process that delivers one or more results. The system being developed will be composed of several sub-systems that in combination execute a desired function, often transforming inputs into outputs. A process can therefore be seen as a meta level system model for a concrete project.

An important aid in the formalisation of SE is abstraction. An activity whereby lower levels concepts are grouped in separate, preferably orthogonal meta-level entities and whereby the specific differences are abstracted away. One could call this categorisation, but this ignores that the meta-level entities still have meta-level links (a form of interactions). This process can be repeated to define a next higher level, the meta-meta-level, until only one very generic concept is left. The exercise is complete if the reverse operation allows to derive the concrete system by using refinement. One could say that this is not different from what modelling defines. There is certainly an overlap, but what makes the approach different is that our approach does not just try to describe what exists, but tries to find the minimum set of conceptual entities and interactions that are sufficient to be used as meta-models across different domains. This is often counter intuitive because it doesn't align with our use of natural language. The latter is often very flexible, but therefore also very imprecise. Natural language is also associative whereby human communication is full of unspoken context information. Engineers have here a source of a fundamental conflict. To prove the ``correctness'' of a system, it must be described in unambiguous terms. At the same time even when using formal techniques, the use of natural language is unavoidable to allow discussing about the formal properties and e.g. architecture of the system. Precise mathematical expressions are by convention bounded with imprecise natural language concepts.

In this white paper we present a middle ground. As a full mathematical approach is not yet within reach of the scale of systems engineering in general, formalisation was achieved by defining a meta-model that formalises the domain. Only 17 orthogonal concepts were needed to define most SE domains. The resulting framework was proven to be capable of being mapped to a safety engineering process. Also many explicit guidelines or requirements of traditional safety engineering standards are found back. From the generic meta-level, we can deduct more specific meta-models by refinement.

2 Related Work

The work presented in this paper is closely related with work going on in other domains, such as architectural modelling. This has resulted in a number of graphical development tools and modelling languages such as UML [1] and SysML [2]. These approaches however suffer from a number of shortcomings:

- Most of the architectural models were developed bottom-up, e.g. as a means of representing graphically what was first defined in a textual format. Hence, such approaches are driven by the architecture of the system and its implementation. As we witnessed often, even when formal methods are used, such an approach biases the stakeholders to think in terms of known design patterns and results in less optimal system solutions. The OpenComRTOS project described in [3] has made this very clear but has also shown how formalisation can overcome this.
- Most of the modelling approaches limit themselves to a specific architectural domain only, requiring other tools to support the other SE domains. This poses the problem of keeping semantic consistency and hence introduces errors.

- Most of the tools have no formal basis and hence have too many terms and concepts that seem to overlap semantically. In other words, orthogonality and separation of concerns is weak or lacking.

Despite these short comings, when properly used, such architectural modelling contribute to a better development process. Overall the approach we propose and implemented in GoedelWorks emphasizes the cognitive aspect of the SE process whereas the different activities are actually just different ``views'' on the system under development. Most of the related approaches do not take these aspects into account.

The remainder of the white paper is organized as follows. The motivation behind the formalization of concepts and their relations are described in the next chapter. Which also presents the link between the abstract, domain independent meta-ontological level, and the domain specific ontological level. The concepts and the unified systems grammar itself are further described. This formalization can also guide the definition and implementation of a concrete instantiation of a SE process. Case studies, which demonstrate that this approach can be applied to different domains, conclude this paper.

3 Introduction to Systems Engineering

Here we give an introduction to our view on Systems Engineering, which provides the framework of understanding the 17 concepts of System Engineering on which more details are given later.

3.1 Intentional Approach to Systems Engineering

Systems Engineering is the process that transforms a need into a working system. Initially we describe the system from the ``intentional'' perspective. Example: ``We want to put a human on the moon''. From this perspective we can derive what the system is supposed to be (or to do). Another perspective is the architectural one. This perspective shows us how the system can be implemented. Often several implementations can meet the intentional goals and part of the systems engineering work is to make the right trade-off decisions.

At the highest intentional level, we can speak of the ``mission'' of the system. The mission is the top level requirement that the system must meet. In order to achieve the mission, a system will be composed of sub-elements (often called components, modules or subsystems). These elements are called "entities" and the way they relate to each other are called "interactions". The term system is used when multiple combined interacting entities fulfil a functionality, that they individually do not fulfil.

Note however that any system component has often been developed in a prior project, hence the notion of ``System of Systems'' emerges naturally. For example, a plane is a system of interacting entities (i.e. body, wings, engine, etc.) that separately all fall under the influence of gravity, but can fly as a whole. Similarly an embedded system is often assembled from standardised hardware and software components, but it's only when put together and an application specific layer is added that the embedded system can provides us with the required functionality.

As entities and interactions form a system architecture, all requirements achieve the mission of a system as an aggregate. However, requirement statements are often vague or imprecise because they assume an unspoken context. To be usable in the engineering domain we need to refine them into quantifiable statements. We say that we derive specifications. In doing so, we restrict the SE state space guided by the constraints that we must be able to meet by selecting from all the possible implementations the ones that meet all our requirements, or at least most of them. In the SE domain we link specifications with test cases allowing us to confirm that a given implementation meets the derived specifications in a quantifiable way. An example requirement statement could be ``a low noise receiver''. The derived specification could be ``85 dB S/N ratio in the frequency band F1-F2''. We can then define a test that will measure a given implementation. The specification also defines boundary conditions (e.g. cost, size) for the implementation choices and the context in which the system will meet the requirements. Hence, the input for the architectural design is taken from the specifications and not directly from the requirements.

In practice the use of the terms requirements and specifications is not always consistent and the terms are often confused. Some people even use the term ``requirement specification'', a rather ambiguous one. Hence, we consistently use the term ``requirement'' when the required systems properties are not linked with a measurable test case. Once this is done, we can speak of a ``specification''. Further on, we will introduce an upper case convention to make that even more explicit.

From the structural or architectural perspective a system is defined by entities and interactions between entities. An entity is defined by its attributes and functional properties. An attribute is an intrinsic characteristic of an entity. Attributes reflect qualitative and quantitative properties of an entity (e.g. colour, speed, size etc.) and have their own names, types and values. For example, the name and the purpose are descriptive attributes of an any entity. A function defines the intended behaviour of an entity. An entity can have more than one function. We use the term function in two ways:

- The traditional ``use case'' of entities (corresponding with the intentional view above);
- The entities' internal behaviour.

Functions define the internal behaviour as opposed to external interactions. In a first approach, interactions are defined using a partial order, i.e. implemented as a sequence of messages. Interactions are caused by events and are implemented by messages. An interaction structure corresponds to a protocol and can be defined with inputs and outputs in form of a functional flow diagram. State diagrams can be used to show event-function pairs on the transition lines between states.

An event is any transition that can take place in a system. An event can be the result of an entity attribute change (i.e. of changing the entity's state). A message can cause and can be caused by an event whereby the interaction between entities results in changes to their attributes and their state. E.g. in software systems an interaction implies some form of data transfer or messages between entities. Such messages can also invoke appropriate functions internal to the entity.

Interfaces belong to the structural part of an entity. An interface is the boundary domain of interaction between an entity and another entity. Interfaces can have input or output types, which define data, energy or information directions at interaction areas between the entities. Examples are an electric socket (input: electrical power or current), a fuel pipeline (output from the tank), or an USB port (input-output).

Interfaces and interactions are related by the fact that an interface transforms an entity internal event into an external message. A second entity will receive such a message through its interface, transforming the external message into an internal form. An interface can also filter received messages and invoke the appropriate entity internal functions. It should be noted that while an interaction happens between two entities, the medium, that enables the interaction, can be a system in its own right. We also need to take into account that its properties may affect the system behaviour. Examples are Internet backbones, long hydraulic channels, transmission lines, etc. One should also note, that the use of the terms ``events'', ``messages'' and ``protocol'' is more appropriate in the domain of embedded systems, but an interaction can also be an energy or force transfer between mechanical components. Or even two people discussing a topic or executing a trained dialogue. Examples are a rally pilot and his copilot navigating the streets.

Another important view in systems engineering is the project development view derived from the architectural decomposition of the system. In this view, once all entities have been identified, they are grouped into work packages for project planning. Each work package is divided into tasks with attributes, such as: duration, resources, milestones, deadlines, responsible person, etc. Defining the timeline of the workplan (i.e. deadlines, periods, limits etc.) and the workplan tasks are important system development stages. Selecting such measures and attaching them to work packages leads to the workplan specification.

3.2 Intentional Requirements, concrete Specification

As mentioned above, a system is described at the highest level by its requirements. Requirements are captured at the initial point of the system definition process and must be transformed into measurable specifications. These specifications are to be fulfilled by structured architectural elements (i.e. entities-interactions, attributes-values, event-function pairs).

Any entity has attributes with values of the appropriate type. For example if we consider the requirement ``the acceleration of the car is at least as high as the acceleration of the top 5 competitors'', we have an entity decomposition (`car'), which maps onto an attribute-value decomposition (with typification of attribute `acceleration' in the type `at least as high as' and value `top 5').

This means that at the cognitive level the qualitative requirements produce entities, interactions (i.e. architectural descriptions) and specifications (i.e. normal cases, test cases, failure cases), work plans, and also issues, to be resolved. The order of this sequence is essential and constitutes a process of refinement whereby we go from the more abstract to the more concrete.

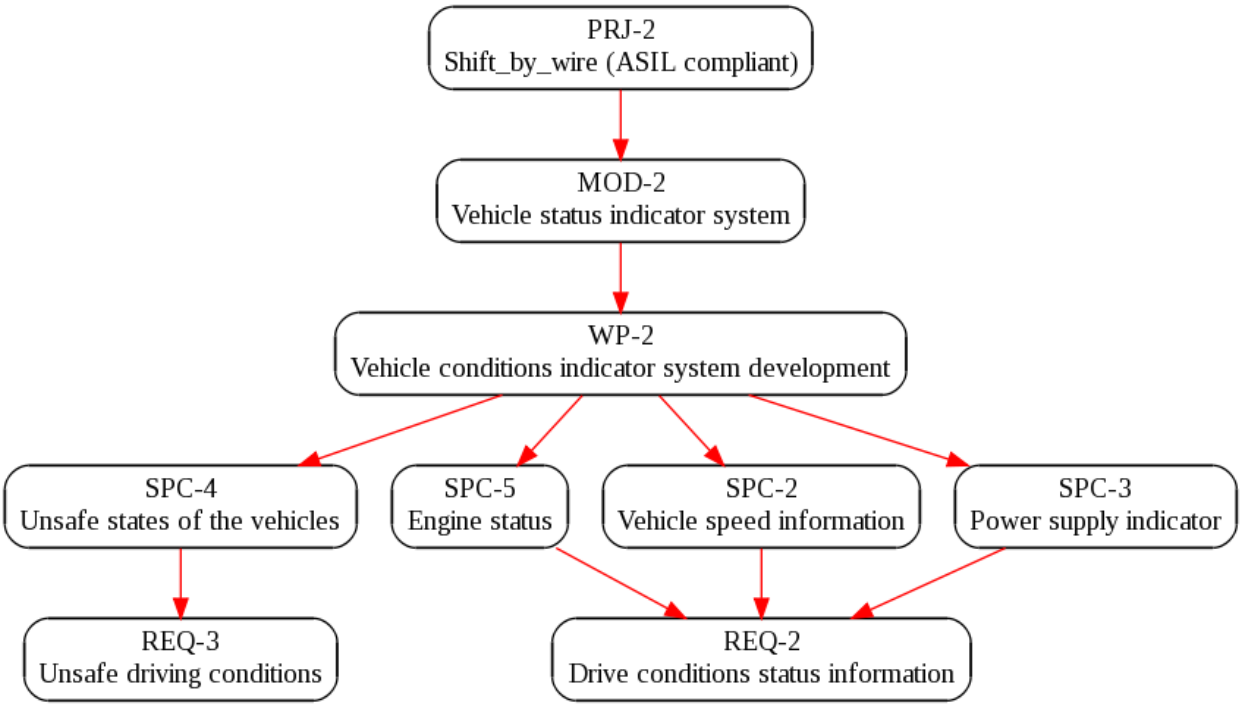

Figure 1 Graphical Representation of the Dependency Links within a GoedelWorks Project.

Using a coherent and unified systems grammar provides us with the basis for building cognitive models from initially disjoint user requests. Requirements and specifications are not just a collection of statements, but represent a cognitive model of the system with a structure corresponding to the system grammar's relations.

Capturing requirements and specifications is a process of system description. Specifications are derived from the more general requirements. This is necessary in order to make requirements verifiable by measurements. E.g. the initial requirement ``the car should be fast'' can be transformed into the specifications ``accelerating from 0 to 100 km/h in 6 seconds'' and ``having a top speed of at least 200 km/h''.

Specifications are often formulated with the (hidden) assumption that the system operates without observable or latent problems. We call this the ``normal cases''. However, this is not enough. Specifications are met when they pass ``test cases'', which often describe the specific tests that must executed in order to verify the specifications. In correspondence to test cases we define ``failure cases'', i.e. a sequence of events that can result in a system fault and for which the system design should cater. Note that security properties are considered as a sub-type of safety cases. We elaborate this further in the text.

4 The Notion of a Systems Grammar as a Meta-Model

In this section we outline the meta-model and its 17 concepts. We first list and define these concepts. To differentiate from the natural language terms, we use upper case for the first letter. Next we discuss the relationships between the concepts, the different views in SE and how this results in a process flow.

4.1 Overview of the Meta-Model

When we use the term System we assume it is being developed in the context of a Project. During the Project a defined Process is followed. The Meta-Model developed uses the following 17 concepts:

1. **System**: The System is considered to be the root of all concepts. It identifies a System as being defined by a (development) Project on the one hand and a (Systems Engineering) Process on the other hand. Acronym: SYS
2. **Project**: The set of activities that together result in the system becoming available and meeting all requirements. The Project is executed by following a defined Process. Acronym: PRJ
3. **Process**: A set of partially ordered activities or steps that is repeatable and produces the system. Examples are a safety process (e.g. meeting safety standards like ISO-26262, DO-178C), a development process, a testing process, a validation processes. Acronym: PRO
4. **Reference**: Any relevant information that is not specific to the system under development but relevant to the domain in general. Examples: datasheets, applicable standards, background articles, etc. Acronym: REF
5. **Requirement**: Any statement about the system by any stakeholder who is directly or indirectly involved. Examples: ``The system should be safe to use'', ``The system shall be environmentally friendly'', ``The system shall meet avionic requirements'', ``The box shall be a red one''. Acronym: REF
6. **Specification**: Specifications are derived from Requirements by refinement. The criterion for the derivation is that the resulting Specification must be testable. Example: ``The colour of the box shall be red with RGB values (250, 20, 10) with an allowed deviation of 5 units per colour''. Acronym: SPC
7. **Work Product**: The result of a Work Package. An example of a Process related Work Product is a Test Report. For example, in the Project a receiver is developed that when tested meets the Specifications that are documented in the Test Report. Acronym: WPT
8. **Model**: A model is a specific system-level implementation of a partial or full set of specifications. A model is composed of Entities and is a Project related Work Product. Examples are: a simulation model, a virtual prototype, a formal model, the implemented system (as the final Work Product). Acronym: MOD
9. **Entity**: An Entity is a composing subset of a model. The interactions create the emerging system properties. Examples are the power supply of a developed receiver, the enclosure and the antenna of a receiver being developed. Acronym: ENT
10. **Work Package**: A set of Tasks that using Resources produces a Work Product that meets Requirements and Specifications. A Work Package shall at least have a Development, a

Verification, a Test and a Validation-Task. Examples are: "The antenna related Work Package", the "Requirement review" Work Package, etc. Acronym: WP

11. **Development Task**: A Task that takes as input the Specifications and develops the Work Products. Examples are: writing the source code of a controller program, selecting and assembling the components of an enclosure, developing a PCB, etc. Acronym: DVT
12. **Verification Task**: A Task that verifies that the work done in the Development-Task meets the Process related Requirements and Specifications. An example is to verify that e.g. coding rules were followed during software development or simply that the right compiler switches were enabled. Acronym: VET
13. **Test Task**: A Task that verifies that the result of a verified Work Product meets the System related Specifications. An example is to have the colour of a box tested with a test set-up and measure the RGB values or measuring the frequency spectrum with a frequency analyser using predefined test signals using the equipment specified in the test plan. Acronym: TET
14. **Validation Task**: A Task that verifies that the tested Work Product meets the System related Requirements after integration with all Work Products constituting the System. An example is to validate that the assembled end-product is indeed a low-noise receiver or that when a system failure is injected, the system really comes to a fail-safe mode or switches to a hot backup controller. Acronym: VAT
15. **Resource**: A Resource is anything that is needed for a Work Package to be executed. Examples are people, equipment, financial resources, etc. Acronym: RES
16. **Issue**: An issue is anything that comes up during the Project that requires further investigation, mainly to determine if the issue is a real concern. For example, an engineer has a suspicion that an O-ring could pose problems in cold weather conditions and he is not sure that this was taken into account. Acronym: ISS
17. **Change Request**: A Change Request is an explicit request to modify an already approved Project Entity. An example is that the issue related to the O-ring was found to be a valid concern and as a result a different O-ring material need to be used, resulting in an important re-engineering work. Acronym: CHR

Figure 2 illustrates this graphically using the three letter acronyms mentioned above. It shows how a system is defined by a Project and by a Process followed during the development. The deliverables are always Work Products (Models as Project Work Products and Process Work Products). However the Process to be followed needs to have been developed as well.

We make abstraction here from often domain specific sub-typing (often introduced by qualifying attributes). One must be careful to keep the subtypes to a minimum and orthogonal set. Otherwise, the terminology confusion will creep in again.

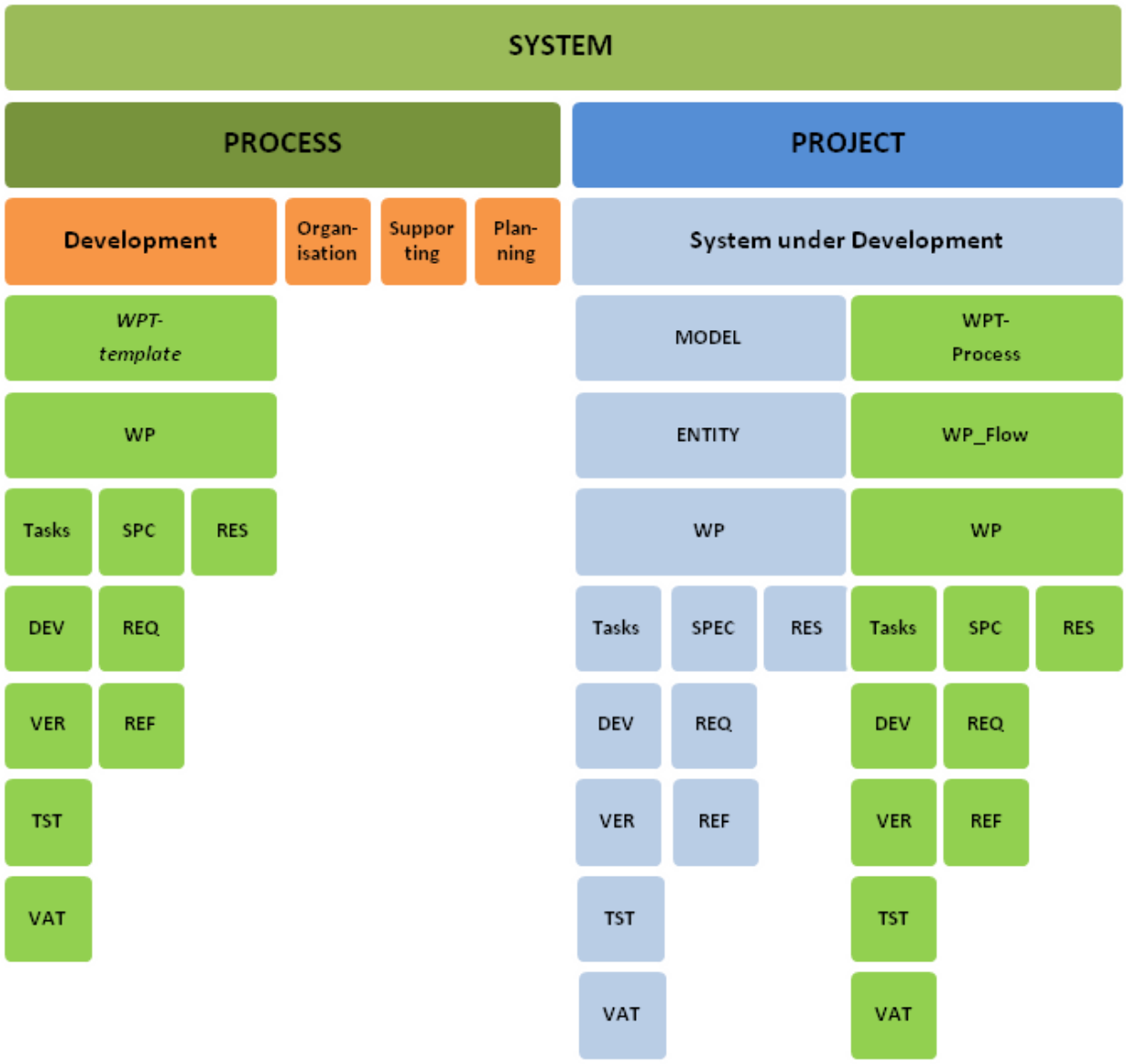

Figure 2 Graphical Representation of the GoedelWorks Meta-Model

The attentive reader will have noticed that the definitions above might not fully agree with his own notions and still leave some room for interpretation. This is largely due to the ambiguities of natural language and established but not necessarily coherent practices in how people use the natural language terms. In practice we also often observe that the same term can present in English a noun or a verb and only the context allows to distinguish between the two. The issue leads to confusion because people use shortcuts when communicating, typically omitting context related information.

This was for example confirmed by a survey done in the context of the FP7 OPENCOSS project whereby the respondents were asked to define certain terms and often came up with very different and even non-overlapping definitions.

Another aspect is that he might have noticed that the concepts are related. While we cannot really change language we stick to the terms as they are but clarify the definitions and why they were chosen. In addition, in the GoedelWorks environment the structure helps to enforce a specific meaning.

4.2 Requirements vs. Specifications

It might come as a surprise, but many safety and engineering standards don't even use the term ``specification''. Most standards will use the term ``requirement'' often with a qualifying prefix. An example is the concept of High Level Requirements (HLR) and Low Level Requirements (LLR) in D0-178C, even if this standard can be considered as relatively mature. Especially the LLR result in ambiguities like recently discussed on a DO-178C LinkedIn discussion forum: ``Are LLR the same as design?''. In ISO-26262 [9] a specification is defined as a set of requirements which, when taken together, constitute the definition of the functions and attributes of item or element.

To eliminate the ambiguity we clearly distinguish between Requirements and Specifications. A Requirement only becomes a Specification when it is sufficiently precise and constrained that we can define a way to test it. We can say that a Specification is a quantified Requirements statement. It comes into being by a refinement process that often will include trade-off decisions driven by the Project constraints. The point is that development engineering activities can really only start when the Specification stage has been reached, else we have too many degrees of freedom. Of course, this does not prohibit early proof-of-concept prototypes.

4.3 The Safety case and the Hazard and Risk Analysis

In the methodology we outline, we do not specifically speak about "safety". This might come as a surprise because many systems engineering standards focus on safety aspects. There is an historical reason for this. First of all, traditional engineering often addresses safety aspects in an implicit way, e.g. by applying well established safety margins and engineering practices. A typical example is construction engineering. This primarily changed when programmable electronics were introduced. First of all, electronics most of the time doesn't wear out perceptibly as the robustness margins at design and manufacturing time are sufficiently high. This is however changing as the continuous shrinking of the semiconductor features is reducing the margins. Another reason is the increased use of digital components. They can fail within a single clock cycle, for example due to a power supply glitch or another external disturbance. But most importantly, when programmed they become very complex and error prone at design time. For this reason, most safety engineering standards focus on how to improve the trust in the system from a safety point of view.

Traditionally, the engineer has here two main activities to do. The first is often called the HARA (Hazard and Risk Analysis) and is used to properly identify prior to development the potential "hazards", in essence the circumstances that can result in a safety risk. In terms of our methodology, the HARA results in Requirements of the subtype "safety case", in contrast to the "normal case" and the "test case". The normal case describes behavior and properties that we expect to have in the assumption that all works as specified. The test case adds requirements related to testing (as it has an impact on the design). The safety case adds the requirements whereby we explicitly take into account that things can go wrong. We

might e.g. take into account that the electronics can be influenced by an external electric field or that the memory becomes corrupted. The safety case Requirements will then mandate that we can coop with such circumstances. Form these Requirements we then deduce safety case Specifications giving us the boundaries (e.g. maximum disturbances we can coop with or acceptable safety risk levels). Once these have been determined, we can look at selecting an architecture or rather look for "safety measures" that must be part of the architecture to mitigate the effects of hazard related risks. For example, we can mandate that the electronics will be housed in a watertight enclosure and that we have a back-up battery.

The safety engineering standards will typically also provide Requirements and Specifications on the Process to be followed during the Project. For example, executing a Hazard and Risk Analysis is mandatory. Or the software shall be developed using approved coding rules, etc. Often, a HARA will also attempt to classify the hazard related risks in different severity classes allowing to relax the safety case related specifications. This is an issue of debate centered around the concept of a fail-safe state. If a fail-safe state results in the system loosing functional properties, can we then guarantee safety? This is in particular troublesome when the system is dynamically controlled. For example when a car looses drive-by-wire capability (literally, e.g. because a wire got broken) at high speed, can we then consider that limiting the engine speed is a sufficient safety measure?

A second major activity typically happens during and after the design and development. This works bottom-up and is often called an FMEA (Fault Mode Effect Analysis) and its variants like FTA (Fault Tree Analysis). It looks at the consequences of a failing component and deduces what the potential hazards could result from it. Typical example, if a power cable is broken, we lose the capability to control the system which can result in a potential hazard.

Using the traceability chain we can see that there is no real difference between the different subtypes (normal, test and safety case) of Requirements and Specifications in relationship to the System Entities that fulfill them. The HARA and FMEA must in the end result in overlapping dependency chains, whereby the safety analysis is only complete if all entities in the dependency tree can be approved. Hence the safety engineering process is iterative by nature. This is reflected in some domains mandating that the HARA and FMEA are repeated during and after architectural development.

4.4 Development, Verification, Testing and Validation

Another distinction is in the terms used to differentiate the Work Package Tasks. This was also motivated by the different meanings different safety and engineering standards give to terms like verification, testing and validation.

In the proposed methodology Verification is linked with Process Specifications whereas Development and Testing are linked with System (product) or Project Specifications. In the case of Development, Specification statements are necessary as input to guide the Development. Although, we say that Testing verifies that the system Specifications were met, we reserve the term Verification for verifying the way the Development was done. The logic behind this is that testing should not be used to find the errors and deviations of the development activities but to find the deviations from the System's

specified properties. Similarly, Validation comes after Testing and is meant to verify that the System as a whole (which implies that it includes Integration) meets the original Requirements statements. In essence, does it meet the Mission Requirements? Note, that this Validation will include Testing activities, typically by operating the System in its intended environment. It is not unlikely that this Testing then measures deviations from the Specifications because the Integration has introduced factors that were not accounted for. We call this the System Characteristics. If the development margins were high enough we might even find that the System Characteristics are better than the System Specifications (E.g. we measure 86 dB while 85 were specified). If we find lower values (e.g. 84 dB) than it is a management or business decision to still accept the system as being ``good enough'' or to reject it.

4.5 The main complementary views in SE.

The meta-model we introduced actually covers three main views that together define the system being developed. Before we elaborate on these, we should clarify what we mean with the term ``System''. In the SE context, the System is what is being developed in a SE Process. However, a System is never alone, it is an Entity on itself that always interacts with two other Systems. One is the environment in which it will be used. This can literally be the rest of the real-world (e.g. a car drives on roads) or a higher level system (e.g. an electronic injector on the power unit). Another one is the (human) operator actively interacting with the System (e.g. an airplane pilot). When developing a System, one must always take these two other Systems into account. Their interactions will influence the System under development (typically by changing the System's state, either by changing its energy level, either by changing the operating mode). The reader will notice that both Systems are characterised by the presence of elements that we never have fully under control. A human operator can be assumed to always give correct commands, but this cannot be guaranteed. The same for the environment. It can be anticipated but not predicted how these two systems will behave. This is the essence of safety engineering.

In the end SE can be seen as the converge of three views. The first one is the well known requirements view. It is concerned with the properties that Systems should and must have and relates to the well known question of ``What is the right System?''. The second is the development or architectural view. It consists of the activities that center around the development that produces the system. It is related to the ``what system?'' question. The third one is the Process view. It defines ``How is the System to be developed right?''. It defines on the one hand a partial order for the different Work Packages of the Work Plan, but it also defines the evidence that needs to be present at the end of a SE Project. What is less understood is that the deliverables of a SE engineering project are on the one hand the System itself (although it really is a collection of Entities that create the System after integration) and on the other hand the Process Work Products. In a systematic, controlled SE Project all these Work Products together define the System. The Work Products document it and together with the dependency chain as a whole provide the evidence that it meets the Specifications and Requirements. The Work Products also do not stand on their own, but define dependencies as well. The final decision of acceptance (e.g. for certification) is not an isolated one but the end-result of a whole tree of assumptions and dependencies.

The Process Work Products are sometimes called the artefacts as if they were by-products, which underestimates their value. They make the difference between development as an engineering activity and development as a crafting activity.

4.6 Morphing Work Products as Templates, Resources and Deliverables

Another important aspect to see is that a Process is also something that has to be developed like any other System. A Process will often be developed by people using as input their experience as well as real-world use cases. Developing a Process also requires a Work Plan and a set of Process Requirements resulting in Process Specifications. The deliverables of such a Process developing Project are on the one hand the Process itself and on the other hand the Specifications for the Work Products to be developed in a concrete Project. In essence, a Process will define templates that need to be filled in during a concrete Project. Hence, the Template becomes a Resource in a concrete Project whereby the Deliverable is again a Work Product.

A simple example is a test plan. A Process will define what we can expect from a test plan in generic terms (e.g. completeness, confidence, etc.). It acts as a Reference for further instantiation. Therefore, an organisation will have to derive an organisation, often domain specific test plan, but still a template enhanced with organisation specific procedures and guidelines. In a concrete project this enhanced template is a Resource that after the Work Package developing the Entity has been approved becomes part of the evidence that the Entity meets the Specifications.

This ``morphing'' of entities is another reason why often terminology can be confusing. It is related to implicit or explicit reuse of previously developed ``Entities'' and actually this is what engineering does all the time. All new developments somehow always include prior knowledge or reuse previously developed Entities that become components or Resources for new Projects. On the other hand it simplifies the understanding of SE by being aware that the finality of a SE Project is always a (coherent) set of Work Products. The Project and the Process are never the finality but the main means to reach the approved state of the Work Products.

4.7 Links and Entity Dependencies

In a real Project, the number of Entities grows quickly. This induces the need to group and structure them. Therefore we define ``structural'' links, i.e. an Entity can be composed of sub-Entities. This is not an operation of refinement but one of decomposition. An example is a Requirement that states ``The car should drive like a sports car''. This can be decomposed into requirements that relate to what people expect from a sports car, e.g. related to the engine power, acceleration, top speed, gear box, suspension, etc.

If we now make these Requirements concrete, we obtain Specifications that are derived from them. For example the Requirement ``Has a top speed in the top 10 car ranking'' results in a Specification ``Has a top speed of at least 250 km/h''. This Specification will need to be fulfilled by specific System Entities. For example, we can first build a physical simulation Model that given parameters like mass and acceleration allow us to determine the engine specifications. We can then select a number of engines

that meet the Specifications. The exercise of linking Specifications with Model Entities is one of mapping.

What the different steps did is creating dependency relationships. The engine characteristics depend on the Specifications and the Specifications depend on the Requirements. The Work Package related to developing the car power group will also depend on Resources. The composing Tasks also define dependency relationships. The Validation will depend on the Testing with the Testing depending on the Verification and the Verification depending on the Development.

These dependency relationships give us also the traceability requirements, allowing to trace back e.g. from the executing software source code back to the original Requirements. If the dependency chain is broken, we know that something was overlooked or not fully analysed. This property is further discussed in the next section.

The example also illustrates another aspect that is tightly related with Requirements management. Assume that there is a Requirement saying ``Fuel consumption shall be within the lowest 5% of the market'' or ``The car must be bullet proof''. These two Requirements are likely in conflict with the sports car driving one. While these examples are straightforward, in practice this might not be so trivial. This is why the different Models are needed. Simulation modelling or virtual prototyping allows us to verify the consistency of the Requirements in view of the available technology (found back as parameters of the model). For example, unless someone invents an ultra-efficient and ultra-powerful engine, the designer will have to choose and make trade-offs between either a fuel-efficient and light car, either a powerful and light car but with a higher fuel consumption or a very safe but heavy and fuel-inefficient car. Similarly, when using formal models we use them to verify critical properties. Often there is a relationship between being able to prove such properties and the complexity, read: architecture, of the System. For example if safety properties can't be proven, often the System will need to be restructured and simplified (which has also other benefits). The point here is that the dependency relationships introduce the necessity of iteration. Figure 3 illustrates some of the dependency links within the GoedelWorks meta-model and views. The dashed lines indicate implicit links, for instance in order to test the output of a Development Task in a Test Task it is necessary to know which Specifications were defined as input to the Development Task.

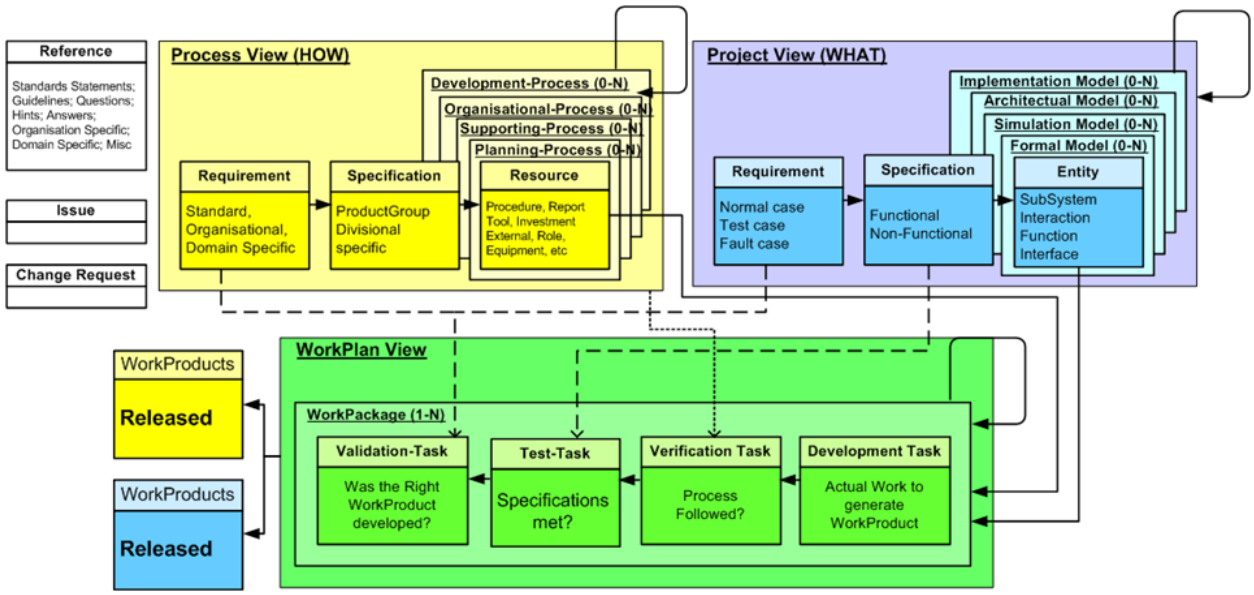

Figure 3 Main dependency links between GoedelWorks entities and views

4.8 State Transitions and Process Flow

The dependency chains identified earlier seem to indicate that a project always proceeds top-down, from Requirements till implementation. When taken literally (like in the waterfall process model), this cannot work because as we have seen Requirement statements do not necessarily form a coherent set and at least some modelling will be needed to weed out overlapping or to make the right trade-off decisions. In practice, some Entities will already exist or have been selected (e.g. when using COTS) and the dependency link is created later on. The way to introduce iterative processes is by assigning a ``state'' to the Project entities and combining them with the dependency relationships. Typically a Project entity will be created and becomes ``Defined''. At some point in time it will become ``In Work'' and when it has been properly worked on, it can become ``Frozen for Approval''. Following a subsequent review, it can then become ``Approved''. More subtler states can be defined but we illustrate the principle using the main ones.

The state ``Approved'' can only be reached if we follow the dependency chain in the reverse order. An entity can only be approved if the preceding entities in the chain have been approved. If any of them is not or looses that status, e.g. because of an approved Issue or Change Request, all depending entities loose that status as well.

The result is that we have now for each Work Product (that includes Models) a separate iterative flow, even if the overall Process flow is following a V-model, as illustrated in Figure 4. The order doesn't come from having predefined a temporal partial order between the Work Packages but from the precedence-dependency chains. Nothing prevents us from starting to work on all in parallel and in any order that is practical. The only order that is imposed is the order in which entities can be approved.

4.9 Pre-qualified components

An important conclusion of the proposed process flow is that we now formulate when a product or a system can be certified to meet e.g. a given safety engineering standard. This is best illustrated by considering for example a processor chip. Such a chip is often an key component in an electronic system and available through commercial channels from the semiconductor manufacturer. It will be programmed with software, typically itself consisting of "firmware" (a selected RTOS, board specific drivers, libraries) and the application software. But as we have seen, to approve the application as a whole all entities in the dependency chain must have been approved as well. In other words, each of these must have been validated, tested and verified to meet requirements and specifications. As such, this is not possible for commercial off the shelf (COTS) components or software tools and components. For example, the processor chip will have a datasheet and a hardware description in a documented form, but what assurance do we have that the description matches the component? As all developers know the data book describes but often does not explain all the details. There are likely documentation errors. Or the software libraries will not have been updated when the chip's design was upgraded or lurking silicon bugs are still hidden in the silicon.

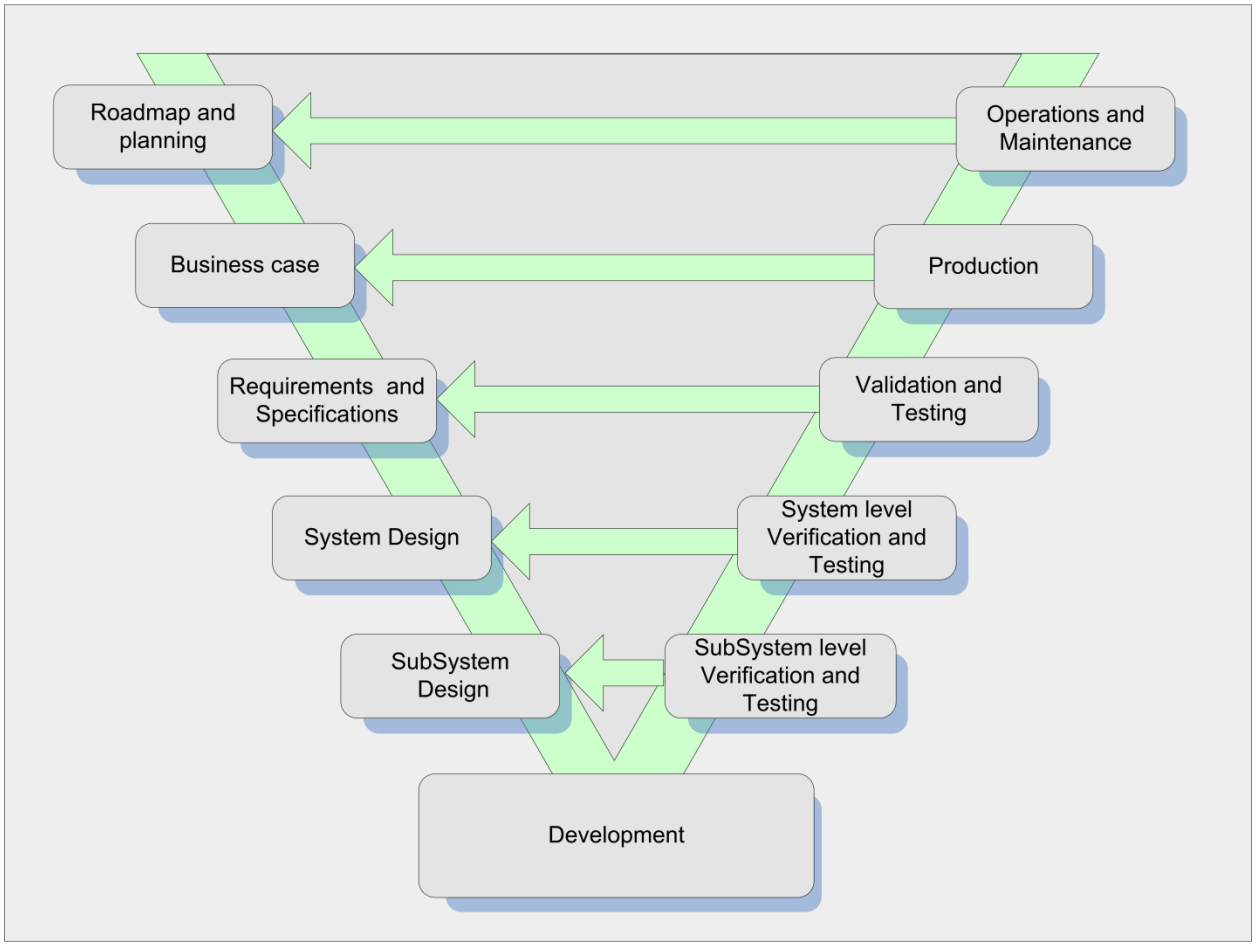

Figure 4 The overall V-Model Process Flow of GoedelWorks

To come to an "approved" component, all the evidence need to be present in a coherent way demonstrating the exact behavior of the component, including the unavoidable deviations. For example a silicon bug is not an issue if it is identified and if the software or hardware work-arounds are implemented and proven to be trustworthy. This way, the component can be integrated in a larger subsystem, etc.

It should be clear that if we want to achieve a higher level of trustworthiness in systems and products developed and aiming at a lower cost, that it is an advantage to use pre-qualified components and sub-system assemblies. The standard COTS approach has a hidden costs whereby every user or customer has over and over again to find the discrepancies between what the datasheet specifies and what the component does and even then he is never sure that there are no hidden errors left.

The solution therefore is to have components that come with their qualification evidence, in essence the collection of approved and validated Process and Project entities in a single package. This can justify a higher selling price. At the same time, we like to point out that the availability of source code (be it for software or any other domain) in itself is not a sufficient guarantee either. The source code needs to be verifiable, readable and commented, linked with the design documentation etc. Even when "proven in use", sometimes the only evidence left if no systematic development process was followed, trust has it limits when the software versions change too fast. In many cases, it will be safer to rely then on an older but known to be stable version.

5 Unified SE vs. Domain Specific Engineering

Another aspect that is worth highlighting is that the unified Process flow and meta-model we described is not specific for a particular domain. The reasoning applies to business processes, which can be classified as social engineering processes, as well as to technical engineering processes. In all cases, once we have agreed on what we need, we can define what will meet the needs and how we will reach that goal.

In the industry, much attention goes to supporting the development of safety critical systems and as such safety standards often define for each domain which process to follow. Each of them also has it own terminology. By introducing the generic meta-model (actually a meta-meta-model) we can cater for the different domains by defining subtypes.

We illustrate this by analysing Requirements. Requirements are often obtained by defining ``use cases'', often descriptions of scenarios that highlight some operational aspect of the system. When using UML, this will be done by a little graphic element representing a ``user'' (virtual or real). We prefer to subtype a Requirement into three classes, i.e. the ``normal case'', the ``test case'' and the ``fault case'' (see also section 4.3). These can be seen as refinement of the generic ``use case''. These cases are defined as follows:

- Normal case: This related to a Requirement that covers the normally expected behaviour or properties. An example is our sports car.
- Test case: This relates to a Requirement that covers a mode in which the system is ``tested''. An example is a processor chip on which test points need to be available to e.g. measure certain operational parameters (voltage, temperature). Test cases do not modify the ``normal case'' Requirements but have an impact on the architectural design.
- Fault case: This relates to a Requirement whereby faults in the system are considered. Faults are defined as occurrences whereby some components no longer meet their ``normal case'' Specifications (derived from ``normal case'' Requirements). An example is the loss of main power. Safety engineering then prescribes what we expect of the System when these occur. Hence we can consider a ``safety case'' as a subtype of a ``fault case''.

The approach whereby we start from a higher level more abstract meta-model allows us also to e.g. consider security aspects as a fault case. We can say that e.g. a security case is a fault case whereby the fault is maliciously injected versus a safety case whereby the fault is often physical in origin. This allows us to reuse a safety engineering approach (for which documented standards exist) to a security engineering approach (for which documented standards are often lacking).

6 GoedelWorks as a supporting Environment

While the unifying SE approach we outlined above provides us with a coherent framework, its applicability can only be validated by applying it to a real project whereby we have the issue that real projects very rapidly generate 1000's of entities. In addition we were of the opinion that such an environment requires the capability to support distributed multi-user project teams.

Therefore, first prototype environments were build, leading to early versions called ``OpenSpecs'' and ``OpenCookBook'' [5]. They allowed to refine the system grammar further, execute small test projects, but most importantly to find a suitable web based implementation. The latter was not so trivial as the complexity of a project database is rather high (largely due to the various links between the entities) and because of the ergonomic needs.

The final implementation was therefore entirely based on a client-server architecture using a browser as client and a database server. As such, GoedelWorks had additional requirements mostly related to the usability aspects:

- International multi-user support with entity specific access rights
- Security and privacy of the project data
- Capability to define, modify import and export processes and projects
- Create and manage process and project entities according to the system grammar
- Change and entity state management
- Queries and dependency analysis

- Creating ``snapshot'' documents (html or pdf).
- Resource and Task planning.

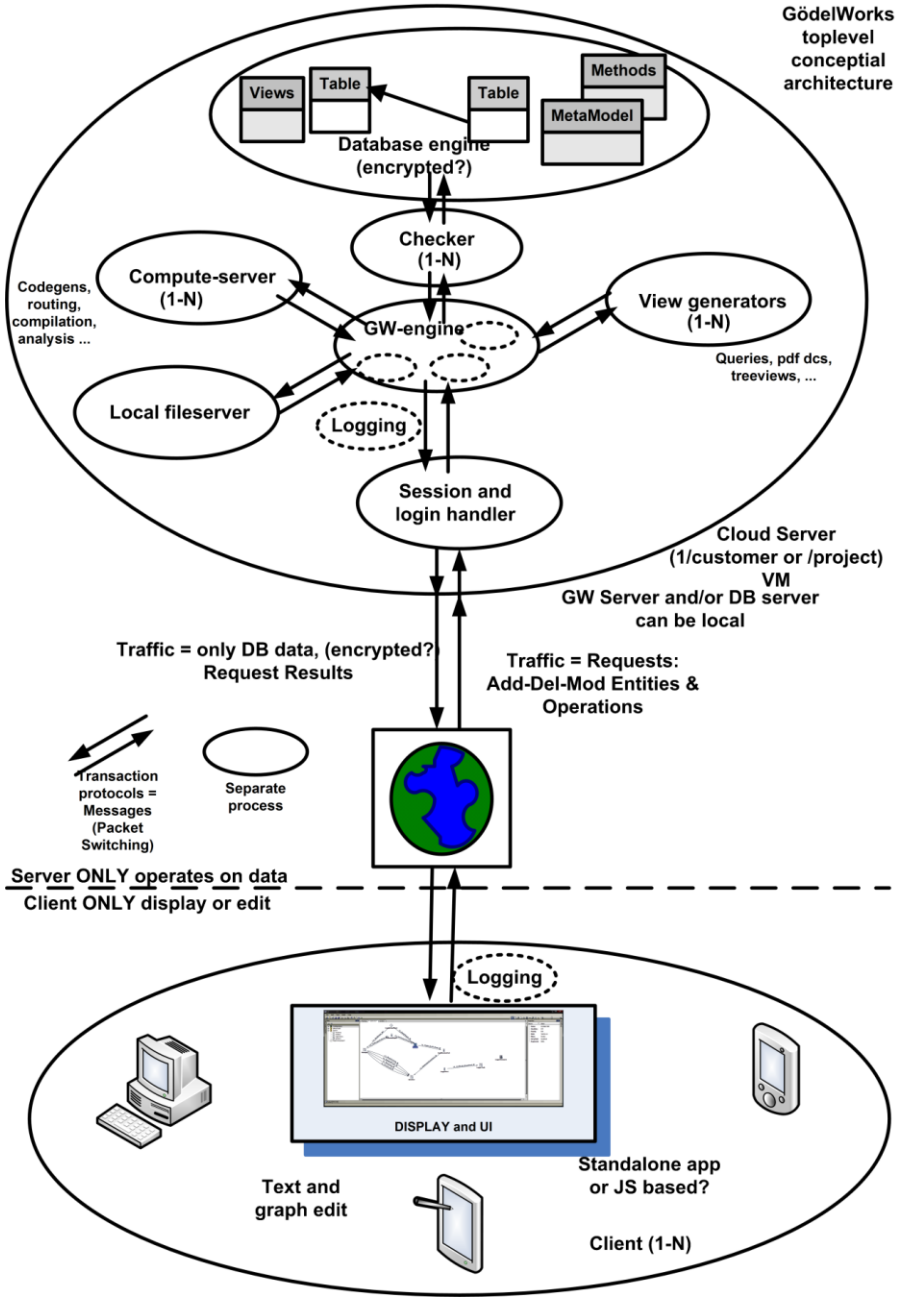

Figure 5 GoedelWorks architecture

Without going into detail, such an environment acts as a unique and central project repository for Processes and Projects, facilitating concurrent team work and communication from early Requirements capturing until final implementation. Interested readers are referred to the publication [4].

6.1 Importing the Automotive centered Safety Integrity Level (ASIL) Process

One of the issues in Systems engineering is that when certification is a Requirement, many standards can be applicable. 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 organization to follow a certifiable Process. This is a bit unfortunate 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 following 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.

6.1.1 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 roughly can be divided into three groups: analysis, realization (development) and operation. All phases are concerned with the safety function 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 which 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: MISRA and later ISO-26262
- Railway: EN-50128
- Process Industry: IEC-61511
- Nuclear Power Plants: IEC 61513
- Machinery: IEC 62061

6.1.2 ASIL: a common 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 and certification. Organizations can add and develop their own Processes as well as import them (when made available in a proper format).

A 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 [7] members and combines elements from IEC 61508, IEC

62061, ISO DIS 26262, ISO 13849, ISO DIS 25119, ISO 15998, CMMI and Automotive Spice. These 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 on-going effort. The ASIL Process also identifies 3 Process domains:

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

The ASIL Process Flow was imported by mapping all ASIL Entities on 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, whereby the extracted semi-atomic requirements became References.

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

6.2 Certification vs. 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 a good practice if certification would be done by the same organization that executed the Project. Even the best organization and best possible Process is still executed by humans and the whole goal of the Systems Engineering Process is to maximize success in a cost-efficient way. Therefore, certification has to be seen as an extra re-validation step executed by an external auditing organization. Certification does not try 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 everything 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, meeting reports, etc. This work is what often scares companies as it doesn't come for free [6]. Following a Process costs extra time and Resources, but has also benefits. 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 operational use and 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. The GoedelWorks environment contributes to this on several levels- by automating the engineering Process:

- The organization uses a standards-aware Process.
- The approval Process reduces rework and double work.
- The certification artefacts are generated during development.
- Organizations 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 allowing to find issues in an earlier stage. In addition, people all over the world can collaborate because all data is centrally located and edited.

7 Conclusions

This paper presents a unifying meta-model to develop and describe System Engineering processes and projects, independently of the domain. SE is formalized through the use of a unifying paradigm based on the notion that in most domains every system, including a process, can be described at an abstract level as a set of interactions and entities. A second observation is that a key problem in SE is the divergence in terminology, hence the use of unified semantics by defining a univoque and orthogonal set of concepts. We emphasize on interactions as a base concept of our approach more than on entities as e.g. in the object-oriented paradigm. This is supported by the use of a “systems grammar” that provides a standardized ontology and meta-model that defines the dependencies from the start. GoedelWorks as a practical implementation of a supporting environment was developed. It was validated by defining some real projects as well as by importing a generic automotive focused process flow.

Acknowledgments

Part of the work has been done under ITEA funding, project EVOLVE (Evolutionary Validation, Verification and Certification) 2007 - 2010.

References

1. Object Management Group: UML. http://www.uml.org/.

2. OMG Systems Modeling Language. http://www.omgsysml.org/.

3. E. Verhulst, R.T. Boute, J.M.S. Faria, B.H.C. Sputh, and V. Mezhuyev. Formal Development of a Network-Centric RTOS. Software Engineering for Reliable Embedded Systems. Springer, Amsterdam Netherlands, 2011.

4. Trustworthy Systems Engineering with GoedelWorks, Jan. 2012. Booklet in series Systems Engineering for Smarties". Published by Altreonic NV under Creative Commons license. Available from http://www.altreonic.com/sites/default/files/Systems%20Engineering%20with%20GoedelWorks_0.pdf

5. V. Mezhuyev, B. Sputh, and E. Verhulst. Interacting entities modeling methodology for robust systems design. In Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on, pages 75 - 80, Aug. 2010.

6. H. Espinoza, A. Ruiz, M. Sabetzadeh, and P. Panaroni. Challenges for an open and evolutionary approach to safety assurance and certification of safety-critical systems. In Software Certification (WoSoCER), 2011 First International Workshop on, pages 1-6, 29 2011-dec. 2 2011.

7. Automotive Safety Integrity Level Public Results. http://www.flandersdrive.be/_js/plugin/ckfinder/userfiles/files/ASIL%20public%20presentation.pdf, 2011.

8. DO-178C, Software Considerations in Airborne Systems and Equipment Certification. http://en.wikipedia.org/wiki/DO-178C, last visited 25.06.2012.

9. ISO_26262, Automotive functional safety. http://en.wikipedia.org/wiki/ISO_26262, last visited 25.06.2012.