An Unified Meta-Model for Trustworthy Systems Engineering

Eric Verhulst and Bernhard H. C. Sputh

Altreonic NV, Gemeentestraat 61A Bus 1, B3210 Linden, Belgium
{eric.verhulst, bernhard.sputh}@altreonic.com
http://www.altreonic.com

Abstract. This paper describes the theoretical principles and associated meta-model of a unified trustworthy systems engineering approach. Guiding principles are "unified semantics" and "interacting entities". Proof of concept projects have shown that the approach is valid for any type of process, 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 the ASIL process, an automotive safety engineering process that was developed to cover multiple safety standards.

Keywords: unified semantics, interacting entities, systems engineering, safety engineering, systems grammar

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, 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. 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. 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 level 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 (or 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 discuss about the formal properties and architecture of the system. Precise mathematical expressions are by convention bounded with imprecise natural language concepts. In this paper we present a middle ground. As a full mathematical approach is not yet within reach of the scale of systems engineering in general, we defined a meta-model that formalises the SE domain in a generic way. Only 17 orthogonal concepts were needed to define most SE domains. The resulting framework was proven to be capable importing a specific safety engineering process. Also many explicit guidelines or requirements of traditional safety engineering standards are found back.

1.1 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]. Without examining them in detail, these approaches suffer from a number of shortcomings:

- Often architectural models are 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 [3].
- Many modelling approaches focus only on a specific domain, requiring other tools to support the other SE subdomains. 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 semantically overlap. In other words, orthogonality and separation of concerns is lacking.

Despite these short comings, when properly used, architectural modelling contribute to a better development process. The approach we propose and implemented in GoedelWorks [4, 5] emphasizes the cognitive aspect of the SE process whereby the different activities are actually just different “views” on the system under development.

Note that the issues GoedelWorks aims to address are more and more being recognised as revelant. Safety is an increasingly necessary property of systems, but at the same time certification costs are escalating. A project that in particular looks at reducing certification costs by taken a cross-domain approach is OPENCOSS [6].

The remainder of the paper is organized as follows. The motivation behind the formalization of concepts and their relations are described in the next section, 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 in section 3. This formalization can also guide the definition and implementation of a concrete instantiation of a SE process. We conclude by a short description of the import of a large automotive focused safety engineering process flow, called ASIL [7]. Other pilot projects were executed as well, but not included by lack of space.

2 A Generic Framework for 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 detailed in section 3.

2.1 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. Part of the systems engineering work is to make the right trade-off decisions.

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). We call these elements “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, that any system component has often been developed in a prior project, hence the notion of “System of Systems” emerges naturally. 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. Unfortunately, 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. 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 fast car”. The derived specification could be “Topspeed 240 km/hr, 0 to 100 km/hr in 6 seconds”. 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. Even the term “requirement specifications”, a rather ambiguous one, is often used. 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”.

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. Attributes reflect intrinsic qualitative and quantitative properties of an entity (e.g. colour, speed, etc.) and have their own names, types and values. 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:

1. The traditional “use case” of entities (corresponding with the intentional view above);
2. 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.

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.

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. One should also note that the use of the terms "events", "messages" and "protocol" is more appropriate for embedded systems, but an interaction can also be an energy or force transfer between mechanical components. Or even two people discussing a topic.

Another important view in systems engineering is the project development view, which is 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 and the workplan tasks are important system development stages. Selecting such metrics and attaching them to work packages leads to the workplan specification.

2.2 Intentional Requirements, concrete Specifications

As mentioned previously, 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).

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. Fig. 1 illustrates this dependency using an extract of a 'Shift by Wire' project, done as part of the ASIL project [7].

Using a coherent and unified systems grammar provides us with the basis for building cognitive models from initially disjoint user requests. Requirements and

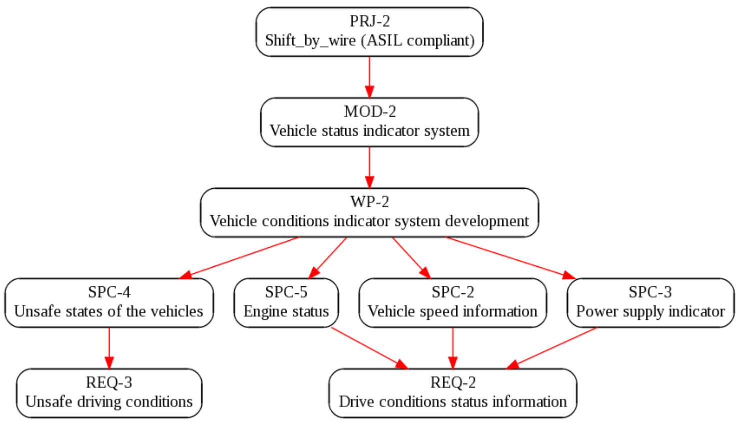
PRJ-2
Shift_by_wire (ASIL compliant)
MOD-2
Vehicle status indicator system
WP-2
Vehicle conditions indicator system development
SPC-4
Unsafe states of the vehicles
SPC-5
Engine status
SPC-2
Vehicle speed information
SPC-3
Power supply indicator
REQ-3
Unsafe driving conditions
REQ-2
Drive conditions status information

Fig. 1. Graphical Representation of Dependency Links in a GoedelWorks Project.

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.

Specifications are often formulated with the (hidden) assumption that the system operates without observable or latent problems. We call these the "normal cases". However, this is not enough. Specifications are met when they pass "test cases", which often describe the specific tests that must be executed 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.

3 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.

3.1 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 consists of 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.
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.
3. Process: A set of partially ordered activities or steps that is repeatable and produces the System.
4. Reference: Any relevant information that is not specific to the system under development but relevant to the domain in general.
5. Requirement: Any statement about the system by any stakeholder who is directly or indirectly involved.
6. Specification: Specifications are derived from Requirements by refinement. The criterion for the derivation is that the resulting Specification must be testable.
7. Work Product: The result of a Work Package.
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.
9. Entity: An Entity is a composing subset of a model. The interactions create the emerging system properties.
10. Work Package: A set of Tasks that, using Resources, produce a Work Product which meets its Requirements and Specifications. A Work Package shall at least have a Development-, a Verification-, a Test- and a Validation-Task.
11. Development-Task: A Task that takes as input the specifications and develops the Work Products.
12. Verification-Task: A task that verifies that the work done in the Development-Task meets the Process related Requirements and Specifications.
13. Test-Task: A Task that verifies that the result of a verified Work Product meets the System related Specifications.
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.
15. Resource: A Resource is anything that is needed for a Work Package to be executed.
16. Issue: An issue is anything that comes up during development that requires further investigation, mainly to determine if the issue is a real concern.
17. Change Request: A Change Request is an explicit request to modify an already approved Project Entity.

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 minimal and orthogonal set. Otherwise, the terminology confusion creeps in again.

The attentive reader will notice 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.

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.

3.2 Requirements vs. Specifications

It might come as a surprise, but many but bot all safety standards don't even use the term “specification”. Most standards use the term “requirement” often with a qualifying prefix. An example are the High Level Requirements (HLR) and Low Level Requirements (LLR) in DO-178C.[8] 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. The latter does not exclude early proof-of-concept prototypes.

3.3 Development, Verification, Testing and Validation

Another distinction is in the terms used to differentiate the Work Package Tasks. Verification is here linked with Process Specifications whereas Development and Testing are linked with System or Project Specifications. In the case of Development, Specification statements are necessary 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. Note, that Validation will include Testing activities, typically by operating the System in its intended environment.

3.4 The main complementary Views in SE.

The meta-model we introduced 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 that always interacts with two other Systems. The first one is the environment in which it will be used. This can literally be the rest of the real-world or a higher level system. The second one is the (human) operator actively interacting with the System. When developing a System, one must always take these two other Systems into account. Their interactions 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 one never has 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 Work Plan view. It consists of the activities that centered around the development that produces the system. It is related to the "what system?" question. The third one is the Process view. It answers the question: "How is the System to be developed?". 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 (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 provide the evidence that it meets the Specifications and Requirements. 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.

3.5 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. 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 (i.e. defined activities) 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. After the Work Package developing an Entity has been approved it becomes part of the evidence that the Entity meets the Requirements and Specifications.

This "morphing" of entities is another reason why 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.

3.6 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.

If we now make these Requirements concrete, we obtain Specifications that are derived from them by refinement. For example, we can first build a physical simulation Model that given parameters allow us to determine the Entity Specifications. The exercise of linking Specifications with Model Entities is one of mapping.

The different Process steps actually create dependency relationships. The Specifications depend on the Requirements. The Work Package related to developing 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 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.

Using a car as example, we illustrate another aspect that is tightly related with Requirements management. Assume that we have Requirements saying "The car shall drive like a sports car", "Fuel consumption shall be the lowest on the market" and "The car must be bullet proof". These Requirements are likely in conflict. While the examples are straightforward, in practice this conclusion is less trivial. This is why 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, the designer will have to 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.

3.7 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 that 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 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 loses 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 for each Work Product (that includes Models) a separate iterative flow, even if the overall Process flow is following a V-model, illustrated in Fig. 2. 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 entities concurrently. The only order that is imposed is the order in which entities can be approved.

4 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 to 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

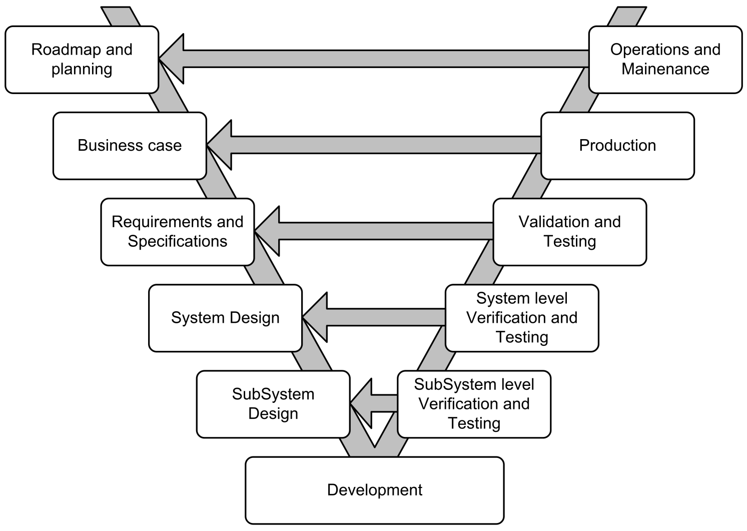
Roadmap and planning
Operations and Mainenance
Business case
Production
Requirements and Specifications
Validation and Testing
System Design
System level Verification and Testing
SubSystem Design
SubSystem level Verification and Testing
Development

Fig. 2. The overall V-Model Process Flow of GoedelWorks.

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. We subtype a Requirement into three classes, i.e. the “normal case”, the “test case” and the “fault case”, as refinements of the generic “use case”. These are defined as follows:

– Normal case: This related to a Requirement that covers the normally expected behaviour or properties.
– Test case: This relates to a Requirement that covers a mode in which the system is “tested”. 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). 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).

4.1 GoedelWorks as a supporting Environment

While the unifoied SE approach this paper presents, provides us with a coherent framework, it's 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 needed 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 in GoedelWorks was therefore entirely based on a client-server architecture using a browser as client and a database server. Additional requirements mostly relate to the useability:

- 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;
- Manage process and project entities following the system grammar;
- Change and entity state management;
- Queries and dependency analysis;
- Creating "snapshot" documents (in HTML or PDF format);
- Resource and Task planning.

Without going into detail, such an environment acts as a unique and central repository for Processes and Projects, facilitating concurrent team work and communication from early Requirements capturing till implementation.

4.2 Importing the ASIL Automotive centered Safety Integrity Level Process flow

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, hence the importance of traceability links. Organizations can add and develop their own Processes as well.

To validate the approach an existing safety engineering process was imported, called ASIL. It 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 the standards in semi-atomic Requirement statements and combining them in a iterative V-Model Process. 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. Also 3 Process domains were identified (Organizational Processes. Safety engineering and development Processes, Supportive Processes. More details can be found in [4].

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. Without going into details, the import of ASIL proved that the meta-model approach works and is consistent. For the interested reader, we refer to a generic description of the ASIL process flow in the reference document [4].

5 Conclusions

This paper presented a unified meta-model to develop and execute System Engineering Processes and Projects. SE was formalized through the use of a unifying paradigm based on the observation that systems, 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. GoedelWorks as a practical implementation of a supporting environment was developed. It was validated by importing a generic automotive focused process flow.

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. http://www.altreonic.com/sites/default/files/Systems%20Engineering%20with%20GoedelWorks.pdf, jan. 2012. Booklet published by Altreonic NV.
5. V. Mezhuyev, B. Sputh, and E. Verhulst. Interacting entities modelling 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. Software Considerations in Airborne Systems and Equipment Certification. . http://en.wikipedia.org/wiki/DO-178C, 2012.
9. Automotive functional safety. http://en.wikipedia.org/wiki/ISO_26262, 2012.