OpenCookbook: An integrated and formalised environment for systems engineering

Eric Verhulst, Vitaliy Mezhuyev, Oliver Faust
Open License Society, Altreonic
Eric.Verhulst, Oliver.Faust [at] Altreonic.com
Vitaliy.Mezhuyev [at] OpenLicenseSociety.org

Abstract

This paper describes the theoretical principles and the practical implementation of an open environment intended for requirements and specifications capturing in the systems engineering domain, but also covering modeling and workplan development till final release. It features a coherent and unified systems engineering methodology based on the Interacting Entities paradigm. It was thought out for the development of embedded systems, but it was proven to be an effective tool for a wide range of system domains. In order to support it, a generic web portal environment was developed, called OpenCookbook. It can be tailored to the needs of a specific organisation as well as accommodate engineering standards like IEC61508.

1. Introduction

Systems Engineering (SE) is considered to be the process that transforms a need into a working system. The need is often not expressed clearly enough and it is the result of the interaction of many stakeholders, each of them expressing their "requirements" in a specific domain language. 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 domains of expertise are always limited. In order to overcome these obstacles, formalization of expressions is required and this is what OpenCookbook attempts to support in the domain of SE. One type of formalization is formalization of natural language; the other type is the separation of concerns on the base of certain systems grammar.

An important aid in the formalization of such a SE process is that at the abstract and domain independent level, common concepts and a common structural architecture can be used. We call this the ***meta-ontological*** level (or conceptual level for easier understanding) vs. the often domain specific ontological level. Such a level is needed because the comprehension of natural language is context, and hence domain dependent, whereas at the level of reasoning about systems in the abstract, the domain specific differences can often be ignored. The meta-ontological level is described by a ***unified systems grammar***. It includes the concepts needed to define requirements, specifications, architectures and work plans when developing a system. The novelty of our approach is that the whole SE process is considered and approached in a formalized way. The approach taken was empirically proven by the development of a supporting tool and applying it to divergent domains.
The paper is organized as follows. The motivation behind the formalization of concepts and their relations are described in the next chapter. It 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 the subsequent chapters. OpenCookbook as a web portal supporting the proposed formalized SE process is presented next. 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.

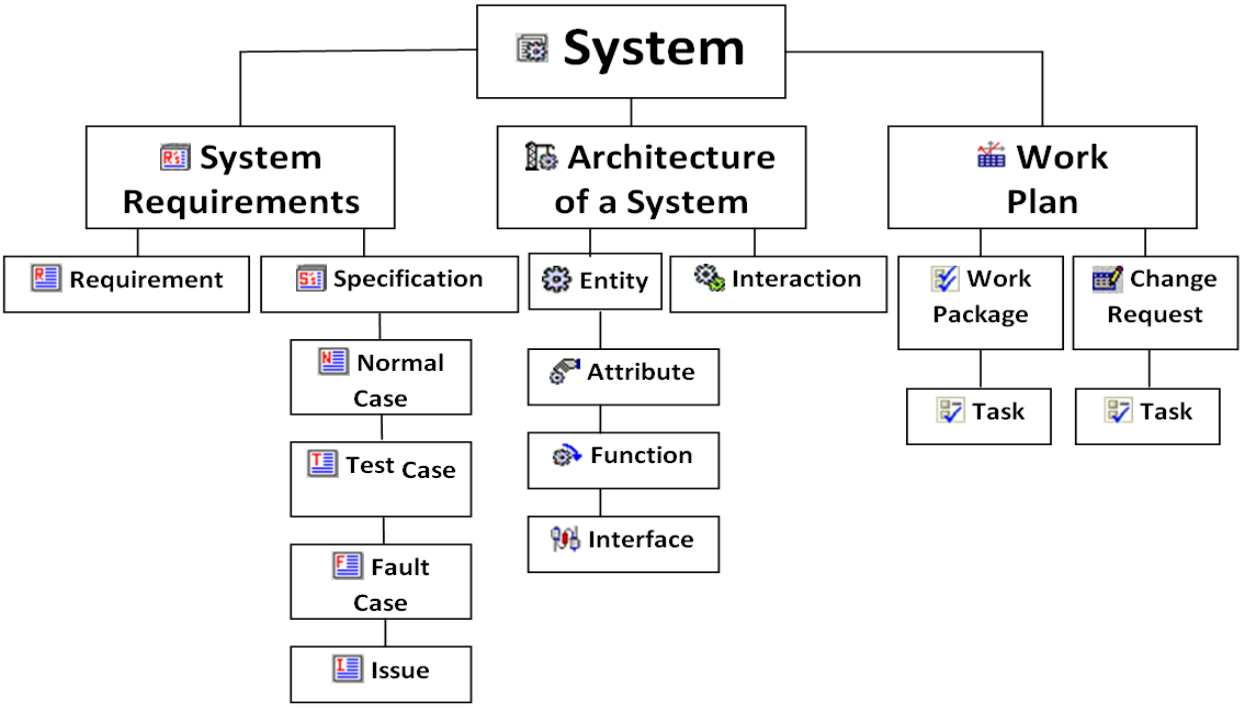
1.1 Intentional approach to systems engineering

Systems Engineering is the process that transforms a need into a working system. Initially we describe what a system is from the intentional perspective. 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 should be implemented. This is exemplified in the unified systems grammar as depicted in Figure 1.
At the highest requirement level a ***System*** is supposed to achieve its ***mission***. In order to achieve the mission, a System will *be* composed of sub-elements (often called 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 the interacting entities fulfill a functionality, which each individual entity does not fulfill. Note, that such a composing entity can be a system in its own right, hence the concept is hierarchical.
For example, a plane is a system of interacting entities (i.e. body, wings, chassis etc.) which separately are aspiring to fall, but which can fly as a whole.

As *entities and interactions form a system architecture*, all r*equirements achieve the mission of a system* as an aggregate. We make an explicit distinction between requirements and specifications. Specifications are linked with test cases and hence specifications are measurable instances of the initial (often imprecise) requirements. It is possible to have several systems with common requirements, but with different specifications (e.g. depending on boundary conditions like cost). Hence, the input for the architectural design is taken from the specifications and not directly from the requirements. Note that the use of the terms requirements and specifications in practice is not always consistent and the terms are often confused. Some people even use the term "requirement specifications", a rather ambigious one. Hence, we consistently use "requirements" as the required systems properties are not linked with a measurable test case. Once this is done, we can speak of "specifications".

From the structural or architectural perspective a system is defined by entities and interactions between the entities.

An Entity is defined by its own ***attributes*** and ***functions***. An attribute is an intrinsic characteristic of an entity. Attributes reflect qualitative and quantitative properties of an entity (e.g. color, 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 behavior of an entity. An entity can have more than one function. We use the term function in two meanings: 1) the traditional "use case" of entities; 2) the entities' internal behavior.



Functions define the internal behavior as opposed to external interactions. In a first approach, interactions are defined using a discrete time model, i.e. implemented as a sequence of messages. Interactions are caused by ***events*** and are implemented by ***messages.*** An interaction structure corresponds to some protocol and can be defined with inputs and outputs by a functional flow diagram. A 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) and a USB port (input-output).

Interfaces and interactions are related by the fact that an interface transforms an internal entity 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 appropriate functions internal to the entity. A data transfer is the simplest application of such functions. It should be noted that while an interaction happens between two entities, the medium that hosts the interaction can be a system in its own right. And we need take into account that its properties can also affect the system behavior. Examples are Internet backbones, long hydraulic channels, transmission lines, etc. One should also note that the use of the terms "events", "messages" and "protocols" is more appropiate in the domain of embedded systems but in a general an interaction can also be an energy or force transfer between mechanical components. For simulation purposes this will make no difference.

Another important view in systems engineering is the project development view based on the architectural decomposition of the system. In such an interpretation once entities are 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, etc. Change requests can be considered as well.
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 workplan specifications.

1.2 Relationships between meta-ontological and ontological levels

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 structured architectural descriptions (i.e. entities-interactions, attributes-values, event-function pairs), which in turn should result in measurable specifications.
Any entity has attributes with values of the appropriate type. For example if we consider the requirement '*the acceleration of the car is as least as high as 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 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 requirements refinement and its concrete definition.

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 "t*est 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 actions that can result in a system fault and for which the system design should cater.

2. Systems Grammar

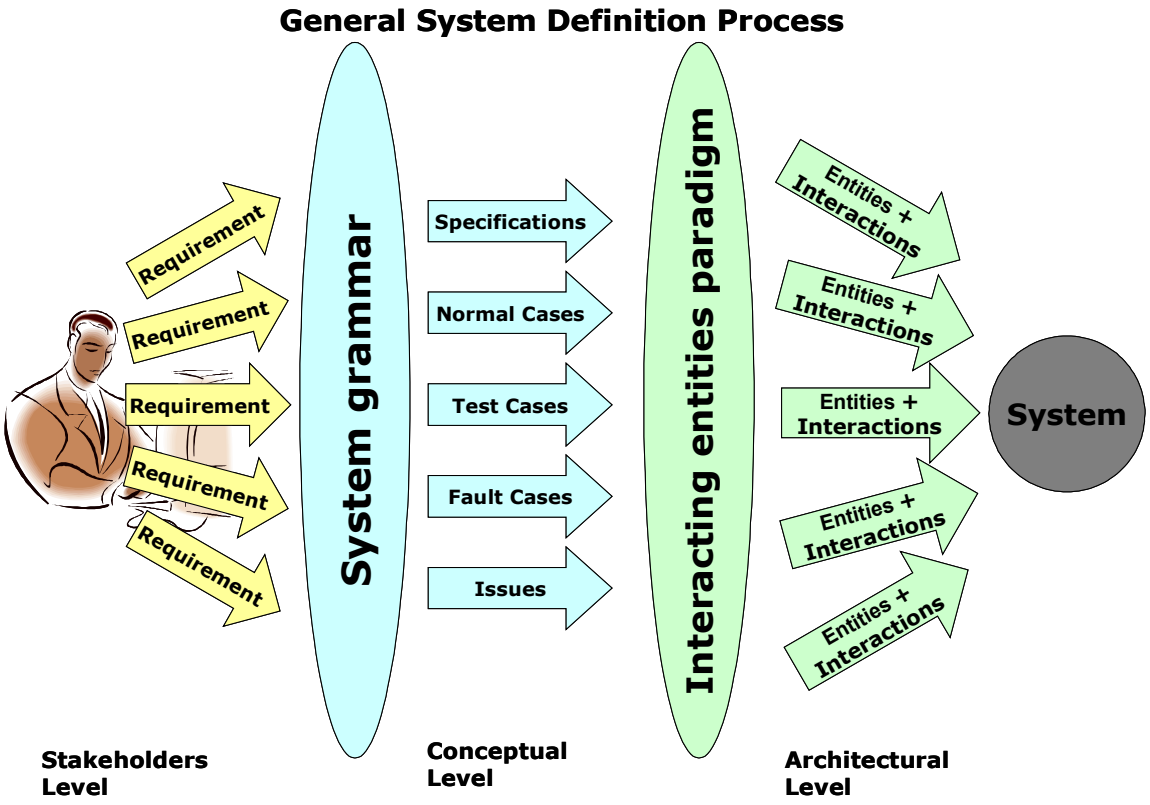


Figure 2: Using the Interacting Entities paradigm for System Definition

2.1 OpenCookbook as a supporting framework

OpenCookbook is a framework that supports the process of defining a system under development. It applies the unified systems grammar as follows. In OpenCookbook a system is defined and developed in an incremental and iterative way by numerous stakeholders. The systems grammar helps to structure, i.e. formalizing the thought process. During this process the OpenCookbook environment becomes the host for a living system specification.
Because of the need for distributed teamwork, we decided to implement OpenCookbook as a web-based environment supporting following activities in the course of a project developing a system or product:

- Requirements capturing
- Specifications capturing
- defining Normal and Test Cases
- defining Failure Cases and Issues
- defining Work Packages and Tasks (development, verification, test and validation)
- Architectural decomposition in Entities and Interactions (see *Fig. 1*).

All these activities are supported by a ***common Repository*** in order to facilitate a coherent systems model development. In a first step the model can be expressed in a natural language. Subsequent steps have to refine and formalize the model. The Repository is based on the unified ***systems grammar*** which acts as the meta-model and defines a semantically coherent framework. This frameworks help a designer in defining the system to be engineered. At each moment the up to date version of a the project documents can be generated. The implementation of OpenCookbook is based on a Content Management System (CMS) resulting in a web portal that is specific for each project. Initially we used an object-oriented database for the prototype implementation in the Plone CMS. The production verison was implementation in the Drupal CMS, benefiting from its powerful taxonomy support. The use of single repository with a coherent systems grammar enables the paramount requirement of traceability found in most engineering standards. Whenever an item is changed, it allows to trace it dependents and precedents by the use of a query.

OpenCookbook has been developed with the following requirements:

- Scalability – must support the development from small and simple to very large and complex systems.
- Generic – must be capable of modeling almost any type of system, independently of the domain.
- Extensibility – must offer the possibility of changing and modifying the meta-model (i.e. structure of repository based on the system grammar). This leads to the creation of domain-specific adaptations.
- Minimal semantics – the initial system must support the minimum semantics of the meta-model. However, OpenCookbook can have extensions later on.
- Isomorphism – must support structural conformity between the architectural model and a given domain.
- System analysis – must allow the analysis of the system under development using a formalized model checker, supporting:
    - The analysis of a requirements consistency, a completeness of the system description and a verification of the time and milestone dependencies in the work plan.
    - Signaling contradictory requirements and allowing choices on the basis of requirements priority (see *Fig. 3*).
    - Checking conformity between specifications and test cases.
    - Categorical analyses by different criteria (e.g. all requirements concerning specific entities, all safety requirements, all tasks need to solve to this date etc.).
    - Supporting "complexity measures":
        - Of entities (e.g. amount of attributes and functions, quantity of relations with other entities).
        - Of the system (e.g. general amount of entities, power of relations as amount of interactions / amount of entities, entities / category, coherence etc.).
        - Of tasks in the work packages (e.g. amount of task / time implementation, amount of task / developer etc.).

2.2 Principles of the Systems Grammar

A systems grammar is defined as a set of concepts which provide the base for a coherent and complete description of a system using natural language constructs. The systems grammar in OpenCookbook describes a project in three orthogonal views: requirements and specifications (conceptual view), architectural (structural or modeling view) and planning (development view) views. It is based on the following principles:

- A Systems Engineering approach.
- The Interacting Entities paradigm.
- A distinction between the ontological and meta-ontological levels in the systems definition.

The ontological level defines concepts related to real systems (physical, chemical, software, hardware etc.). E.g. all entities and interactions that are architecture related. The meta-ontological level defines generic concepts and is expressed by notions such as entity, interaction, requirements, specifications, test cases etc.

The cognitive model is the initial point for the architectural model definition. The transition from the cognitive into the

architectural level is achieved by methods similar to object-oriented design (decomposition, abstraction, encapsulation, typification, structuring, hierarchy defining etc.).

The systems grammar, depicted in *Fig.* 1, as well as its elements and implementation in OpenCookbook are described in detailed in the following subsections.

2.2.1 Requirements classifications

Requirements must provide a full description of what the system needs to offer, but in an abstract, preferably implementation independent way. System requirements are classified by different criteria:

- Target or intentional requirements that express the purpose of entities and their interactions.
- Structural requirements that reflect how the system aggregates entities and their interactions.
- Functional requirements that reflect the internal and external behavior of the entities.

Note, in general two types of knowledge exist: 1) conceptual and 2) procedural (or methodological). This means that knowledge relates to the "what" but also to the "how" concept. In other words, a cognitive model has methodological components. Practically speaking, this means that we not only use "is a", "has a", "consist of" etc. relations, but also "how to do" procedural models.

Thus, to develop a cognitive model the OpenCookbook repository has to reflect procedural and conceptual types of knowledge. We distinguish in the systems grammar two classes of terms: the first class is procedural (active, methodological, the 'how') part – e.g. interactions and functions, while the second class is conceptual (passive, descriptive, the 'what') part – e.g. attributes.

Requirements are typed as general (related to the system) or specific (related to an entity or other architectural part). Requirements can be categorized: performance, scalability, portability, extensibility, quality, usability, safety, reliability, maintainability, control, security, cost, convenience, robustness, recoverability etc. And each requirement is classified according to its relevance (mandatory, recommended, optional). This is needed for a more precise definition of the requirements categories. Finally, requirements are also classified according to a context pattern: enabling, dependency, maintainance, testing, avoidance, optimization, etc. Patterns reflect a context in which the goals of the requirements are to be achieved.

Using such classifictaion method allows us to formalize the model analysis and helps us to automate the decision making process in case of contradictory requirements (see figure 3

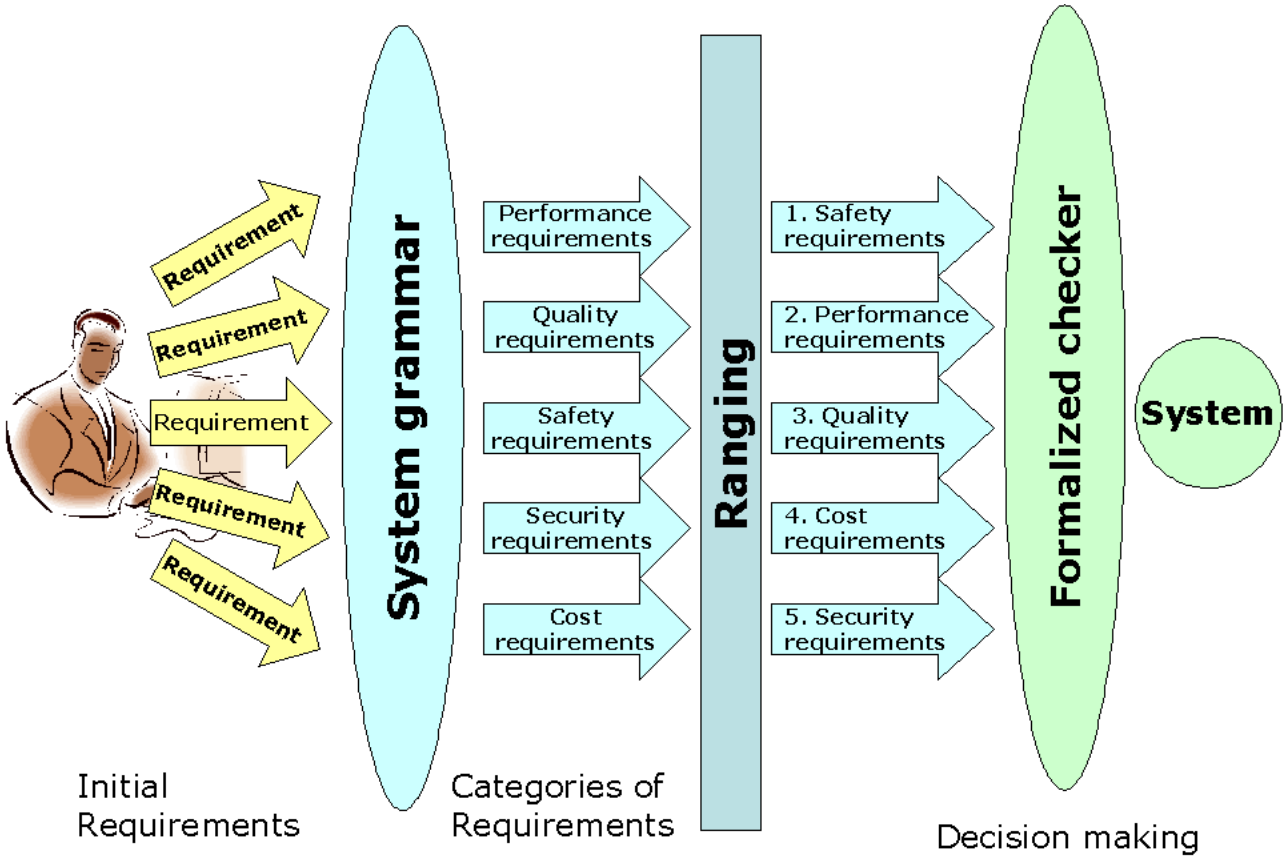


Figure 3: Categorical analysis of requirements

Categorical analysis of requirements allows us choose from all possible system states the one that meet the requirements.
This is part of the decision-making process. Note, different stakeholders can have different views on the importance of the requirements measures. E.g. often safety and cost requirements will be formulated. However, depending on boundary conditions, not all requirements will have the same importance. In such a case, we can use expert evaluation to apply an ordering on the system requirements. Experts then define numerical ranges for critical system requirements and hence define the development priorities.

2.2.2 Specifications, derived from requirements

In general, the requirements and specifications definition phases cannot be fully separated. While analyzing the requirements the developer has often an initial notion about decomposing the system into architectural parts and related specifications. Note, this is not always desirable, because it can prevent the engineering team from identifying and selecting better alternatives.
A specification is a quantified requirement which consists of Normal Cases, Test cases, Fault Cases and issues to be solved. The Normal Case is a description of a required system behavior or state. A Test Case is a specific type of requirement with pattern 'test', a category and a severity (critical, major and minor). A Test Case is derived from a Normal Case and can be related to an issue. The purpose of a Test Case is to verify a specification.
A Fault Case is derived from a normal case with pattern 'avoidance' and has the properties “category” and “severity”.

2.2.3 Architectural view

Following the specifications, the systems engineering process will define the system architecture. Each set of specifications is to be mapped onto selected entities that through their interactions define the system. Note, at this stage, each entity is only a functional ‘block’. The final implementation choice is done in work packages. Often this will result in having to make trade-off decisions between various alternatives. Such an architectural view emphasises the need for standards, but mainly at the interface level.

2.2.4 Modeling

While the architectural view is often seen as the activity of development, one must keep in mind that architectural entities and interfaces are actually there to let the system meet specifications. In practice this means that a selected architecture as an implementation is just a special case in the context of “modeling”. In practice one will need additional models that often represent at a more abstract level a partial set of the specifications to be met. E.g. formal models can be used to formally verify critical properties of the system and simulation models can be used to verify that the requirements of the system are coherent and complete without the need to simulate all the details of the implementation architecture. Hence, in practice it is better to speak of architectural, formal, simulation and implementation models both contributing to the development of the implementation architecture. Once the implementation has been finalised, a final validation can be done eventually resulting in measured deviations to the specifications. Often these are called the characteristics of the system as they can be different from one system to another even when build using the same architecture.

2.2.5 Workflow view

The third view in the systems grammar is the workflow view. It describes the development, i.e. the implementation activity of the systems engineering process.
We say that a work plan produces work packages and can include change requests. A work package consists of tasks (related to the development of a concrete entity or other architectural parts), description, start date, end date, dependencies, responsibilities. A task also has attributes: description, priority, deadline, deliverables, resources, manager.
Each entity has an attribute 'status' which reflects the development progress in time according to the project schedule. The entity status can be:

- **'Purpose identified'** - means that the entity has been identified and received the name and the purpose attributes.
- **'Attributes identified'** - means that the attribute set is complete (e.g. all attribute-value pairs are defined).
- **'Functions identified'** - means that the internal behavior is characterized (e.g. all event-function pairs are defined).
- **'Interfaces identified'** - means that all interactions between all entities are identified.
- **'Ready for implementation' -** means that all above status levels have been reached.
- **'Implemented'** - means that an entity is developed and complete.
- **'Approved'** - means that an entity has been validated to meet all test cases.
- **'Integrated into the system'** - means that the entity is fulfilling its requirements at the system level.

We also consider the project status of an entity in development.

- **'In work'** - means that the entity is now being defined.
- **'Frozen for reviews'** - means that the process of defining entities and their properties and behavior is halted to allow a coherent review.
- **'Frozen and approved'** - means that an entity is accepted as necessary element of the system.

3. Relation between the meta-ontological and ontological levels

Any domain has its own ontology and defines its specifics as a set of relations between entities. From the stakeholder's perspective each domain has its specific set of terms and associated rules of how things function. However, these terms and rules are often quite similar at a higher level of abstraction.
In OpenCookbook, ontologies are relations applied to real systems. Meta-ontological relations on the other hand are a reflection of ontologies in the more abstract domain of human cognition. This essentially means that a generic cognitive model is developed. Therefore, this domain is a meta-level versus the physical one.
The concepts of the systems grammar are linked by meta-ontological relations such as 'is described by', 'consists of', 'is descendant of', 'has attributes', 'achieves' etc.
In OpenCookbook these relations are implemented using references, e.g. between a requirement and an entity it refers to, similarly there is a reference between a specification and a requirement, etc. These relations can be both of the type 'one to one' and 'one to many'. Some relations are implicit (e.g. an aggregation of entities).

The systems grammar defines that a system is described by requirements and that requirements are the initial point in defining the system. In the requirements capturing phase the developer has to define names of entities and interactions to which a requirement is related. So, when defining requirements an initial architectural decomposition into entities and interactions takes place. This decomposition is also used in the work plan view, because a task or work package always concerns a concrete architectural part.

All entities and interactions have their own set of requirements and specifications as implicit purposes or target functions. As entities and interactions compose the system, the sum of purposes of all entities and interactions achieves the mission of the system. Thus, the main reasoning frame of OpenCookbook is an architectural modeling one. It provides a framework for all views that one can have in a SE project.

3.1 Applying the 'interacting entities' paradigm for the realization of requirements and specifications relations

Requirements and specifications reflect interactions between real entities at the ontological level, but at the same time requirements and specifications are entities with specific (logical) relations between them at the meta-ontological level. So, we can use the 'interacting entities' paradigm to reason about the relationships between requirements and specifications. It allows us to apply a unique approach for all phases of the systems engineering process.
The "interacting entities" paradigm is applicable for a wide range of system domains. In each domain interactions and entities can defined in a domain specific way. This is more a matter of denomination (using words that are commonly used in a specific domain) than one of substance.

OpenCookbook allows the decomposition of a system into entities and interactions. Its approach is anthropocentric and reflects the domain of human cognition. To understand a system, we need to decompose a united reality first into separate entities that implement specified properties, and by doing so the system re-emerges by defining the interactions between these entities.
A key question is: what kind of relations we have to implement in OpenCookbook to reflect the reality properly (or more correctly say, “enough for achieving the system as a goal”)? The systems model must be isomorphic to the real

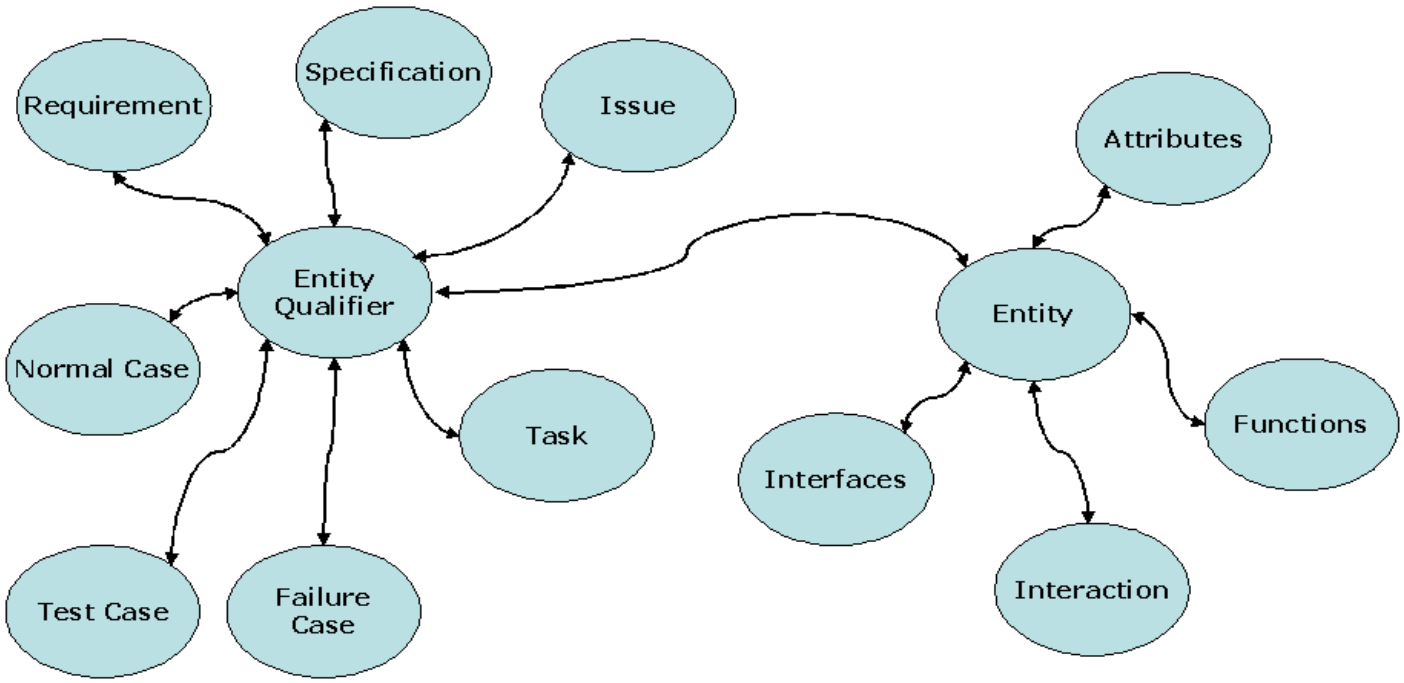


Figure 4: Relations between conceptual and architectural levels of system development

system. E.g. relational databases apply an "entity" – "relationship" paradigm and rely on the set theory. However, the relationships between database fields is not isomorphic to the cognitive model of a system under development.
The history of programmable systems is characterized by a transition from an imperative to a declarative approach. The application of the declarative paradigm in OpenCookbook implies defining predicates of a knowledge base and defining rules of logical deduction.
To develop a logical system which allows reflecting a wide range of domains we need general relationships such as: “is a”, “has a”, “consist of”, “is part of”, etc. These notions reflect structural relations which can be applied to any system, because any system can be defined as a set of having structure entities. Besides these structural relationships, we need also temporal (e.g. now, next, previous) and functional ones.
As a conclusion we can say that the "interacting entities" paradigm allows us to apply a declarative paradigm for cognitive model development. Defining taxonomy and logical relationships between its terms provides us with a model that is isomorphic with the domain of human cognition.

4. The transition from the meta-ontological to the ontological level

At the initial stages of the systems engineering process, a precise architectural decomposition into real entities and interactions does not yet exist. There is only an incomplete cognitive model expressed in the form of requirements and specifications. The task for the systems engineer is to transform it into the ontological domain, i.e. to develop an architectural model that will be isomorphic to the real system.
During the first stage of defining the system we allocate qualifiers (see *Fig. 4*) that will be used in the ontological domain. The architecture definition at this requirements and specifications capturing phase is nominal - we only have names, i.e. a vocabulary of objects and interactions, that is not yet the real, ontological or physical model. This is the first step of the transition from the meta-ontological to the ontological level.
Thus, the linking pin between ontological and meta-ontological levels is primarily the system itself i.e. the entities and interactions in the architectural view on the system (see *Fig. 5*).
The next step is the transformation of meta-ontological into ontological *relationships*, which are different by nature (e.g. subordination between requirements does not imply that such subordination exists between the architectural entities). In general, the development of methods for making the transition from the cognitive model to the architectural model is a challenging task. The essence of the method will consist in finding ontological relations and entities in a cognitive model.
A cognitive model must take into account the way humans think. In the case of requirements and specifications capturing we have a limited cognitive model. We suggested that the description reflects structural, functional and temporal relations. So, we have a final set of statements and the task is reduced to performing a linguistic analysis of the formalized requirements and specifications language. Introducing such a formalized language is now being performed.

5. Prototype development

To test the concepts and its applicability a prototype environment was developed using the Plone [3] and next the Drupal [4] Open Source Content Management System environments. This means that a new project or system-under-development is actually created like a web portal with specific modules that reflect the systems grammar. Utilities and scripts allow us i) to make the link between the different phases of the systems engineering process, ii) to run tests for checking consistency and completeness and iii) to generate a document. Using such an existing environment has many advantages. For example, support for multi-user administration and the accompanying review process is built in. Being a web based tool, it also caters for distributed team work. Other advantages are that existing plug-in modules can be used to e.g. create a Wiki, forum and repositories of the project background and foreground documentation.

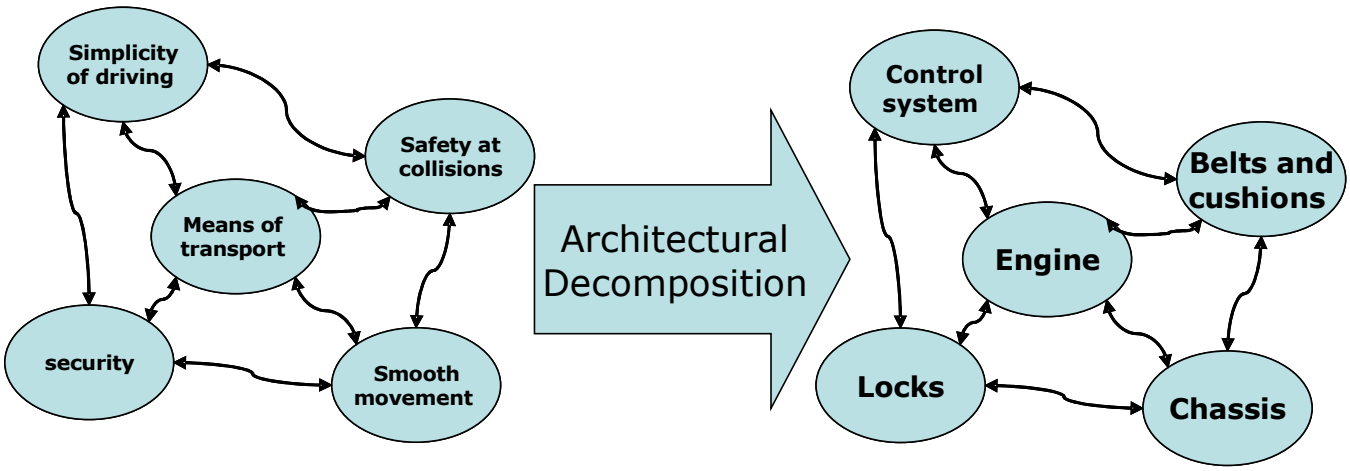


Fig. 5. Transition from conceptual into architectural level

Another aspect that is of importance in the global context of systems engineering is that we must avoid that such an environment is a stand-alone tool. As mentioned at the beginning, Systems Engineering (SE) is considered to be the process that transforms a need into a working system. Many domains are crossed and entities from one domain are reformulated or better said translated in another domain. The issue here is not so much syntax (syntax is often domain or tool specific) but semantics. Such translations, often involving human intervention, are not always univoque or straightforward because of the hidden or assumed context. A typical example are dataflow diagrams. A first sight the diagrams they look like connected processes that exchange data using the connections between the blocks. In reality, in a dataflow diagram the communication is implicit and actually often hides the hidden assumption of shared memory. Hence, translating dataflow diagrams to a process oriented programming system or to a distributed computing environment is not a straightforward task as data dependencies must be analysed, impact on performance must be analysed, etc. In the context of safety driven designs, this opens the door to human errors and to unintended side-effects, jeopardizing the correctness of the system under development. In general, one must be aware that different domains often have contradictory concepts, might have overlapping but still subtle differences in their semantics or worse might not have the equivalent concept at all.

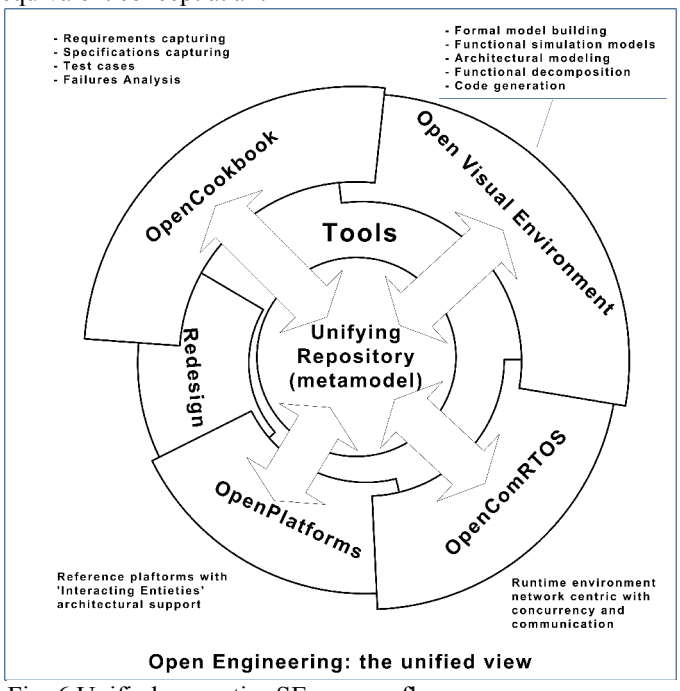


Fig. 6 Unified semantics SE process flow.

When standards are then used like e.g. UML, this fact often results in the emergency of a wide range of "dialects" to fill the gaps, but in the end undermining the usefulness of the original standard. For this reason, we adopted an approach based an "unified semantics" from the beginning and adopted a restricted architectural paradigm (interacting entities). In the end the goal is to define a single set of tools and components covering the whole processflow from requirements till the final realisation as an embedded system. Fig 6. illustrates this approach.

5.1 Requirements tracing

All activities in a systems engineering process can be seen as a coherent set of views on the same system under development. Therefore, any requirement, specification or task is linked with a set of entities. We also indicate in any architectural description references to corresponding requirements, specifications and tasks. Such links are needed for feedback and traceability between the different system views. It also allows conducting e.g. an impact analysis when changes are applied.

5.2 Experiments in different domains

In order to fine-tune the prototype and to verify the applicability to different domains, a number of limited experiments were conducted. Projects were defined to develop a Real Time Operating System (OpenComRTOS), a process flow supporting the IEC61508 safety standard, and a processor software environment. In the course of these experiments refinements were applied, but overall these experiments in diverse domains indicate the suitability of the approach. Most issues were related with the ergonomics of the environment and some deficiencies of the Plone CMS implementation. For this reason we later use Drupal CMS, with the additional benefit that it features a taxonomy system.

The Systems Engineering approach was also tested by mapping it onto a Business Process Engineering method. Here, it was found that the meta-ontological concepts fully apply although often a very different terminology is used or different tools. E.g. while a technical engineer might use virtual prototyping or CAD tools to simulate different user scenarios, a business manager will likely create a business plan, simulating the business process using a financial spreadsheet. In the context of a business environment this reflects that the "mission" of the system is to generate profit whereas in the engineering domain the mission is often to provide a certain functionality.

A final test was the use of the OpenCookbook modeling approach in the development of the OpenComRTOS mentioned above. Such a formal modeling approach raises even further the abstraction level, from the meta-ontologiocal domain in the fully abstract domain of mathematical logic. It was found that this was very helpful. A first point to support this statement is that the modeling technique works in an incremental way. From a small very abstract model the refinements and details are added until a model emerged that was very close to the implementation architecture. Each intermediate model was checked exposing logical errors in the design. As a consequence, the project progressed in small steps with each step being subjected to an intensive review

process via internet by all team members. Secondly, the abstraction level is completely removed from the implementation domain. This allowed us to detect the negative impact of being too familiar with the implementation domains and how this biases engineers and stakeholders as humans. The result was a much cleaner and more compact systems architecture. Furthermore, the team had a much greater confidence in the correctness of its architecture.

For the interested reader, we used the TLA/TLC modeling language and checker of Leslie Lamport [4]. This environment supports the notion of concurrent processes and communication between them. This corresponds with the Interacting Entities used for the OpenCookbook systems grammar.

6. Related work

The work done with OpenCookbook is closely related with work going on in other domains, such as architectural modeling. This has resulted in a number of graphical development tools and modeling languages such as UML and the recent SysML. Such approaches however suffer from a number of issues:

- 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 discovered in the tests, such an approach biases the stakeholders to think in terms of known design patterns and results in less optimal system solutions.
- Most of the modeling 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 lacking.
- Most of the tools on the market bring too many details to the top level, with little support for abstracting away the details. This undermines the overview and abstraction power.

Nevertheless, when properly used, such architectural modeling contributes to a better development process. Overall the OpenCookbook project 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.

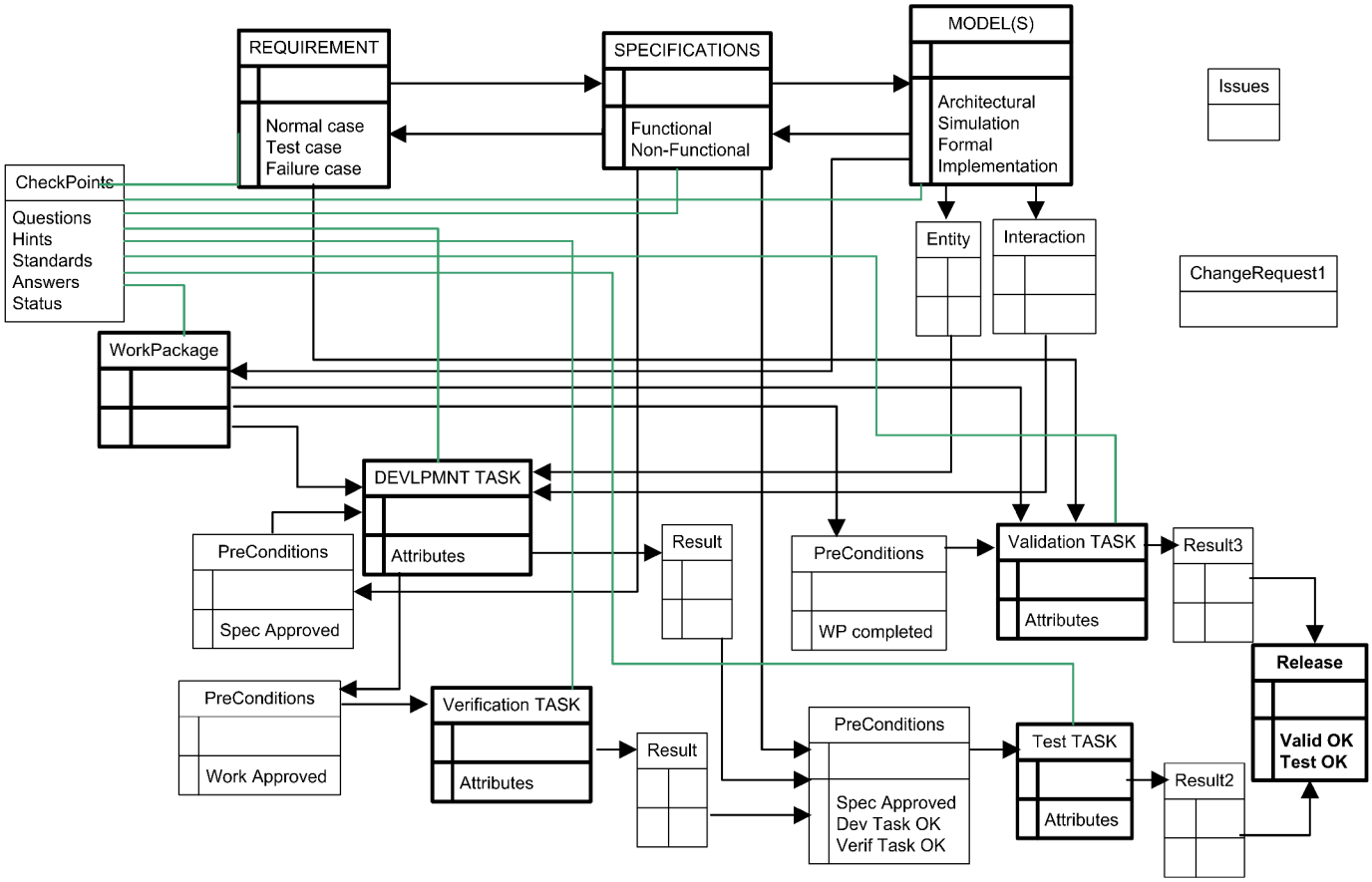


Figure 7: OpenCookbook systems grammar

7. Real world use

Although tests with the prototype version in different domains indicated the conceptual suitability of the approach, a number of issues were discovered during practical use. We list them below:

- Too abstract for the practical engineer: The prototype OpenCookbook implemented a formalised but rather abstract approach to systems engineering. Most engineers and stakeholders alike often rely mostly on heuristic knowledge and have a hard time formulating their thoughts in the systems grammar framework.
- OpenCookbook was defined with the implicit assumption that a project is started from scratch. However most projects will reuse parts of an existing architecture, hence the starting point should be a template project rather than an empty one.
- Many organisations use a lot of heuristic knowledge under the form of checklists. OpenCookbook has no concepts to include such knowledge and link it with the other elements of the OpenCookbook Systems grammar.
- Official standards like IEC61508 often define rules and conditions that must be satisfied in order to allow the project to be certified according to the standard. On the other hand many of the standards are often only defining conditions or guidelines for the development process, whereas often issues have their origin in the requirements and specifications phase.

This analysis has resulted in the definition of a number of extensions to the environment. We highlight the main differences with OpenCookbook. Essentially, the extensions put the work plan concepts at the same level of the requirements-specifications and modeling activities. The difficulty is that the first domain reflects the design view that is mainly time-independent whereas the work plan view requires taking into account a timely order of steps that have to be followed to arrive at a product release.
The complete OpenCookbook systems grammar is represented synoptically in Figure7.

7.1 Checkpoints

Given the similarity between heuristic knowledge and standard requirements, the notion of a "checkpoint" was introduced. A checkpoint can be related to heuristic knowledge or a specific standards rule but linked with any entity of the project (in any view). As such checkpoints are essentially entities that help in knowledge management, often generic and heuristic in nature. Checkpoints also work as on-line design wizards, shortening the project time as the learning curve is reduced. As such checkpoints are not part of a given project but are meta-rules applying to a specific, narrowly or broadly defined application domain. A typical use will be to support a product family whereby each product is different but e.g. reuses parts of a previous projects. Another class of checkpoints are related to a specific domain of properties, e.g. safety or security, where often deep domain knowledge must be combined with certification standards.

7.2 Issues and Change Requests.

Issues are essentially like checkpoints but they arise in the course of a project. When resolved, they can result in a checkpoint entering the pool of knowledge. In a project they act like a reminder.
Change requests on the other hand arise when during the course of a project approved specifications or decisions (like architectural choices) are to be modified or deemed to be modified. This can be due to changing stakeholder requirements or because during development (e.g. during testing) deviations from the specificatons or issues are discovered. As the acceptance of a Change Request can result in serious and costly rework, a Change Request must be carefully analysed for its impact. In the worst case, it can result in an early termination of the on-going project and the creation of a new one, although part of the work of the previous project can be reused.
Issues and Change Request can also be linked with any entity of the project.

7.3 Explicit difference between architectural, simulation and implementation models.

Although OpenCookbook is not a domain specific modeling environment (it does so only at the meta-level), the environment must keep track of all models developed and how the sub-entities relate to the other entities in the systems grammar. In most model driven architectural approaches developing a model is sometimes seen as defining the specifications or even developing the actual implementation whereby simulation and formal modelling are seen as supporting activities. Doing so carries the risk that properties of these different modelling domains are confused or taken for granted. Nevertheless, engineering is essentially modelling as far as all these modelling activities are done concurrently and at each stage the dependencies with the other elements of the systems grammar are visible. Therefore we opted to make this explicit. Note that in OpenCookbook the models themselves are external and often provided by third party tools. We also not the special case of the implementation model. When this is achieved, this basically means that of all the possible architectural models one was selected and approved as the one that is the system that meets the mission requirement.

7.4 Workplan and tasks.

In general specifications result in architectural entities (and interactions) that must fullfill these specifications. Often they will be grouped because functional clustering will occur. Such a functional cluster will then be assigned to a Work Package for implementation.
We opted for the explicit architectural paradigm of "interacting entities". This has the major benefit that it allows

a much easier separation of the architectural issues and helps in defining work packages are are small enough and well separated from each other. The granularity can be adapted to the nature or the working style of the organisation. The benefit of this architectural paradigm is also that in principle all sub-system entities only "interact" through well defined protocols and interfaces. As we strive to achieve unified semantics, this also results in a preference for concurrency at the implementation level. In the context of embedded systems resulting in a natural use of multi-tasking programming systems and multi-CPU execution platforms. This is one of the reasons why OpenComRTOS was formally developed as one of the first elements of Altreonic's systems engineering methodology. The runtime layer is essential for performance and safety properties of the system.

We distinguish the following classes of tasks. A development task is the actual development activity, but can include activities like simulation, prototyping or formal model verification. It can only start when the specifications are approved. The result however should be a selected implementation model.

A verification task is defined as the activity that will verify not an implementation but the development task itself. It can be seen as an audit of the development activity and need to verify checkpoints related to the development activity itself. It answer the question 'did we develop it right?' Typical examples are the adherence to coding rules, proper version management, design rules, review meetings, etc. Note that verification can only really start when the implementation has reached the status "work done", although this should not exclude spot check verification while the development is still going on. It is clear that verification should be done by different people that those carrying out the development.

A test task will test (according to the test cases of the specifications) the result developed. It can only start when the verification task was approved.

Finally we consider the validation task. A validation task will validate that the implementation result (after verification and testing was succesful) meets the original requirements. It answers the question 'did we develop the right system?'. Validation works in a top-down fashion. The final validation is the integration of all developed sub-systems. If this is succesful, the product can be 'released'. Note that at this stage some properties might be different from the specifications. We call these the characterisation of the system.

Another issue here is that often a workpackage is related to specific sub-system entity requiring activities like development, testing, verification and validation but in a given organisation people will have been assigned to a specific class of activities. E.g. testing will be done by the test department, and then often the test team is made responsible (or blamed when an issue is found). This is methodologically wrong. At all stages must the Work Package team leader remain responsible for all aspects of his assigned Work Package, because ultimately an issue found during testing will often be traced back to a design issue.

7.5 Other project items.

Other features were also introduced like the definition of roles, milestones, release point and version management. However these can be supplied by external environments (e.g. links to a software repository) or by using the build-in support of drupal.

Conclusions

OpenCookbook implements a formalized requirements and specifications capturing environment up to the level of identifying major architectural elements and workplan packages. The whole process is formalized through the use of a unifying paradigm based on the notion that every system in (most) domains can be described at an abstract level by a set of interactions and entities. 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 to define a system under development. Current work focuses on adding more formal verification processes. A link is being established as well with the OpenComRTOS development environment allowing to directly map specifications onto OpenComRTOS tasks and services.

References

1. (www.OpenLicenseSociety.org)
   i. Open License Society white paper.
   ii. OpenCookbook Systems Grammar
   iii. OpenComRTOS architectural design.
   iv. www.altreonic.com
2. www.plone
3. www.drupal.org
4. http://research.microsoft.com/users/lamport/tla/tla.html
5. http://www.omgsysml.org
6. http://www.uml-forum.com/

Project funding

Part of the work by Open License Spciety has been done under ITEA funding, project EVOLVE (Evolutionary Validation, Verification and Certification).

OpenCookbook is used in the Flanders Drive ASIL project by Altreonic to incorporate Safety related standards in a generic project methodology. OpenCookbook supports the development of the methodology using a customised version of the generic OpenCookbook.