OPENCOOKBOOK: UNIFIED AND FORMALISED ENVIRONMENT FOR SYSTEMS ENGINEERING

This paper describes theoretical principles and practical implementation of an environment for systems engineering. The environment guides and supports developers during requirements and specification capturing over architectural modelling and workplan development till final release. It features the coherent and unified system engineering methodology based on the Interacting Entities paradigm. In order to realize it, a generic web portal was developed, called OpenCookbook. It was thought out for embedded systems development, but it was proven to be an effective tool for a wide range of other system domains. OpenCookbook can be tailored to the needs of a specific organisation as well as accommodate to the engineering standards like IEC61508.

Introduction

Systems Engineering (SE) is considered to be a process that transforms a need into a working system. The need is often not expressed clearly enough, because 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 knowledge is required and this is what OpenCookbook attempts to support in the domain of SE. One type of formalization is the formalization of natural language; the other type is the separation and structuring of concerns on the base of certain system grammars.

An important aid in the formalization of such an SE process is that at an abstract and domain independent level, common concepts structured in a common system grammar can be used. We call this the meta-ontological level 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 abstract reasoning about systems 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 for system development. The novelty of our approach is that the whole SE process is considered in a unified and formalized way. The approach taken is 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 is described in the next section. This section introduces 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 sections. OpenCookbook, as a web portal supporting the proposed formalized SE process, is presented next. The OpenCookbook can guide both definition and implementation of concrete instantiations of SE processes. Case studies, which demonstrate that this approach can be applied to different domains, con-
1.1 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 modelling view) and planning (development view) views. It is based on the following principles:

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

Every domain has its own ontology as a set of concepts and relations between concepts. From the stakeholder’s perspective, each domain has its specific set of terms and rules of its manipulation. However, these terms and rules are often quite similar at a higher level of abstraction.

The ontological level defines concepts which are related to real systems (physical, chemical, software, hardware etc.). The meta-ontological level defines generic concepts and it is expressed by notions such as entity, interaction, requirements, specifications, test cases etc. The meta-ontological concepts of the systems grammar are linked by 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. Relations can be of type 'one-to-one' and 'one-to-many'. Some relations are implicit (e.g. an aggregation of entities).

As it was mentioned, the Systems Engineering is the process that transforms a need into a working system. Initially, we describe what a system is from an intentional (requirements) perspective. From this perspective we can derive what the system is supposed to be (or to do). Another perspective is the extensional (architectural) one. This perspective shows us how the system should be implemented. This is exemplified in the unified systems grammar as depicted in Fig.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 elements (often called modules or subsystems). These elements we call Entities and the way, they relate to each other, we call Interactions. Note, that such a composing entity can be a system in its own right; hence the entity concept is hierarchical. The term system is used when interacting entities exhibit a functionality, which each individual entity does not exhibit.

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 requirements 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 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 specifications and not directly from requirements.

Note, 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 ambiguous one. Hence, we consistently use “requirements” as the required system properties. Only after requirements have been defined, we can derive from it specifications.

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

Specifications are often formulated with the (hidden) assumption that the system operates without observable or latent problems. We call this the “normal cases”. However, this is not enough. Specifications are met when they pass “test cases”, which often describe specific tests that must be 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. Such failure cases the system design should cater. The idea is: by formulating the failure cases, we starting understand, what can go wrong before the real system design, what allow prevents possible disasters.

Thus, using a coherent and unified systems grammar provides us with the basis for building cognitive model from initially disjoint user requests. Requirements, specifications, normal, tests, fault cases are not just a collection of statements, they represent a cognitive model of the system with a structure which corresponds to the system grammar relations.

1.2 Interacting Entities Paradigm

From the extensional 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, name and purpose are descriptive attributes of all entities.

Functions define internal behaviour in contrast 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 they are implemented by messages. An interaction structure corresponds to some protocol and can be defined by a functional flow diagram or message sequence chart. 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. An interaction changes state of all entities involved in the interaction. E.g. in software systems an interaction implies some form of messages transfer between entities. Such messages can transfer data or 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 two or more entities. Interfaces can have input or output types, which define directions of data, energy or information transfer at interaction 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 interfaces transform events, which are internal to entities, into external messages. A second entity will receive such a message through its interface, transforming the external message into an internal form (event). An interface can also filter received messages and invoke appropriate functions internal to the entity. Data transfer is the simplest application of such interactions. 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 behaviour. Examples are Internet backbones, long hydraulic channels, transmission lines, etc. One should also note, that using the terms “events”, “messages” and “protocols” is more appropriate in the domain of embedded systems, but in general an interaction imply an energy, matter or information transfer between entities.

Another important point of view in SE is the project development view, which in our approach is based on architectural system decomposition. In such an interpretation entities, once they are identified, 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 tasks are important system development stages. Selecting such measures and attaching them to work packages leads to workplan specifications.

### 1.3 Intentional and extensional levels of a system definition

As mentioned above, at the highest level a system is described by its requirements, which we consider as intentional level of a system definition. Requirements must be transformed into extensional architectural descriptions (i.e. entities-interactions, attributes-values, event-function pairs), which in turn should result in measurable specifications.

E.g. every entity has attributes with values of the appropriate type. For example, if we consider the requirement 'the acceleration of the car is at least as high as the 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').

Thus the transition from the intentional requirements to the extensional architectural level is achieved by the decomposition, abstraction, encapsulation, typification, structuring, hierarchy defining methods (see fig. 2). This means, the intentional qualitative requirements produce extensional entities, interactions, interfaces, attributes, functions (i.e. architectural elements descriptions) and specifications (i.e. normal cases, test cases, failure cases), next work plans and tasks, as also issues to be resolved. The order of this sequence is essential and constitutes the process of system definition.

Note, 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

---

Fig. 2. Transition from intentional into extensional level
model which is expressed in the form of requirements and specifications. The task for the systems engineer is to transform it into the extensional domain, i.e. to develop an architectural model that will be isomorphic to the real system.

As a novelty of our SE approach we propose the next method. During the first stage of system definition, we allocate qualifiers (see Fig. 3) that will be used in the extensional (architectural and work planning) domain. The architecture definition, at this intentional phase, is nominal - we only have names, i.e. a vocabulary of entities and interactions, which is not yet the real, ontological or physical model. This is the first step towards the transition from the intentional to extensional level.

The linking pin between intentional and extensional levels is primarily the system itself, i.e. the entities and interactions in the architectural view on the system.

The interesting problem is analyses of intentional and extensional relationships. These relationships are different by nature (e.g. subordination between requirements does not imply that such subordination exists between architectural entities). In general, the development of methods for making the transition from the intentional requirements model to the extensional architectural model is a challenging task. The hardest problem will be finding architectural relations in a intentional cognitive model.

Note, in the case of requirements and specifications we have a limited cognitive model. We suggested that the description reflects structural, functional and temporal relations of a real system. 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 and methods of its analyses will be the topic of our next publications.

2. SYSTEM DEFINITION PROCESS

2.1 OpenCookbook as a supporting framework

OpenCookbook is a framework that supports the process of defining the system which will be developed. It applies the unified systems grammar, as was described in the previous text. In OpenCookbook, a system is defined and developed in an incremental and iterative way by numerous stakeholders. The systems grammar helps us to structure the thought process. During this process the OpenCookbook environment becomes the host for a living system specification.

Because there is a need for distributed teamwork, we decided to implement OpenCookbook as a web-based portal which supports following activities in the system definition process (see Fig. 4):

- Requirements capturing;
- Transforming Requirements into Specifications (with definition of Normal, Test, Failure Cases and Issues);
- Architectural decomposition in Entities and Interactions.
• Defining Work Packages and Tasks (Development, Verification, Test and Validation).

All these activities are supported by a common Repository in order to facilitate a coherent model development of a system. In a first step, the model is expressed as natural language requirements. Subsequent steps have to refine and formalize this conceptual model. The Repository is based on a unified systems grammar which acts as the meta-model and allows separate and refine expressed requirements.

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 modelling almost any type of system, domain independent.
• Extensibility – must offer the possibility to change and modify the meta-model (i.e. change the structure of based on the system grammar repository). 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:
  o The analysis of requirements to specifications correspondence, a completeness of the system description and a verification of time and milestone dependencies in the work plan.
  o Signalling contradictory requirements and allowing choices on the basis of requirements priority.
  o Checking conformity between specifications and test cases.
  o Categorical analyses by different criteria (e.g. all requirements concerning specific entities, all safety requirements, all tasks need to solve to this date etc.).
  o 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.).

3. Prototype development

To test the concepts and its applicability a prototype environment was developed using first Plone [3] and next Drupal [4] Open Source Content Management Systems (CMS).

To be specific, the prototype was done using the Plone CMS. The production version was implemented with the Drupal CMS, benefiting from its powerful taxonomy support.

In both tools a new project or system-under-development is created like a web portal with specific modules that reflect the systems grammar. Utilities and scripts allow us i) to make a link between different phases of the systems engineering process, ii) to run tests for checking consistency and completeness and iii) to generate documents. Using such existing environments has many advantages. For example, support for multi-user administration and the accompanying review process is built in. Being a web-based tool, it naturally 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 which aid project background and foreground documentation. At each moment the up to date version of the project documents can be generated.

Another important aspect, with respect to the global context of systems engineering, is that we must avoid that such an environment is a standalone tool. As mentioned at the beginning, Systems Engineering 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 unique or straightforward, because of the hidden or assumed context. A typical example is dataflow diagrams. At first sight these diagrams look like connected processes that exchange data using the connections between blocks. In reality, in a dataflow diagram the communication is implicit and actually often hides the assumption that shared memory is used. Hence, translating dataflow diagrams to a process oriented programming system or to a distributed computing environment is not a straightforward task, 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 or overlapping concepts, there might be as subtle differences in their semantics or they might not have the equivalent concept at all.

When used the weak standards like UML, it results in the emergency of a wide range of “dialects” to fill the gaps, but in the end these dialects undermine the usefulness of the original standard. This is the reason why from the beginning we base approach on unified semantics and adopted a restricted architectural paradigm (interacting entities). The goal is to define a single set of tools and components covering the whole process flow from requirements till the final implementation.

3.1 Experiments in different domains

In order to fine-tune the prototype and to verify the applicability to different domains, a number of 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 to the ergonomics of the environment and some deficiencies of the Plone CMS implementation were also discovered. For this reason we switched to the Drupal CMS, which has the additional benefit as a taxonomy system.

The Systems Engineering approach was also tested by mapping it onto a Business Process Engineering method. Here, we found that the metalevel 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. This reflects that in a business environment the “mission” of a system is to generate profit, whereas in the engineering domain the mission is often to provide certain functionality.
A final test was using the OpenCookbook modelling approach in the development of the Open-ComRTOS operating system. Such a formal modelling approach raises the abstraction level even further, from the meta-ontological domain to the fully abstract domain of mathematical logic. For us this higher level of abstraction was very helpful. A first point to support this statement is that the modelling technique works in an incremental way. Starting from a small, very abstract model, refinements and details are added until a model emerges that is very close to the implementation architecture. Each intermediate model is checked which exposes logical errors in the design. As a consequence, the example projects 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. The result was 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, for the formal modelling we used the TLA/TLC modelling language and checker of Leslie Lamport [4]. This environment supports the notion of concurrent processes and communication between them. This corresponds to the Interacting Entities used for the OpenCookbook systems grammar.

5. 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 rely mostly on their heuristic knowledge and have a hard time formulating their thoughts in a 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 in the form of checklists. OpenCookbook has the concept 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 standards define only conditions or guidelines for the development process, whereas many 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...
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 entities and interactions. 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 meta-model (high level ontology) 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 mapping specifications onto OpenComRTOS tasks and services.

REFERENCES

1. Режим доступу: [www.OpenLicenseSociety.org](http://www.OpenLicenseSociety.org)  
   i. Open License Society white paper.  
   ii. [OpenCookbook Systems Grammar](http://research.microsoft.com/users/lamport/tla/tla.html)  
   iii. [OpenComRTOS architectural design](http://www.uml-forum.com)  
2. Режим доступу: [www.plone](http://www.plone)  
3. Режим доступу: [www.drupal.org](http://www.drupal.org)  
4. Режим доступу: [http://research.microsoft.com/users/lamport/tla/tla.html](http://research.microsoft.com)  
5. Режим доступу: [http://www.omg SYSML.org](http://www.omg SYSML.org)  
10. Mark Austin, Vimal Mayank and Natalia Shmunis. PaladinRM: graph-based visualization of require- 
    ment organized for team-based design. // System engineering. – 2006. – V.9, N.2. – P.129.  

Межуєв В., Верхалст Е. Open-Cookbook: уніфіковане й формалізоване середовище для проектування систем

Межуєв В., Верхалст Э. OpenCookbook: унифицированная и формализованная среда для проектирования систем

В статье раскрываются теоретические принципы и особенности практической реализации программной среды для проектирования систем. Среда поддерживает разработку систем, начиная с этапа выявления требований и спецификаций, через этап архитектурного моделирования и формирования рабочего плана к выпуску конечного продукта. Среда использует методологию системного проектирования, основанную на парадигме взаимодействующих сущностей. Для реализации среды был создан web-портал, который получил название OpenCookbook. Сначала OpenCookbook был задуман для разработки встроенных систем, однако была доказана его эффективность в применении к широкому диапазону других предметных областей. OpenCookbook может быть адаптирован к потребностям определенной организации, так же, как и к требованиям технических стандартов, к примеру IEC61508.