Interacting Entities Modelling Methodology for Robust Systems Design

Vitaliy Mezhuyev, Bernhard Sputh
Open License Society
Berdyansk, Ukraine
e-mail: Vitaliy.Mezhuyev,
Bernhard.Sputh@OpenLicenseSociety.org

Eric Verhulst
Altreonic NV
Linden, Belgium
e-mail: Eric.Verhulst@Altreonic.com

Abstract - **This paper describes the theoretical principles and the practical implementation of OpenCookbook, an environment for systems engineering. The environment guides and supports developers during requirements and specification capturing over architectural modelling and workplan development till validation and final release. It features a coherent and unified system engineering methodology based on the interacting entities paradigm. In order to implement it, a generic web portal was developed. Targeting embedded systems, it nevertheless 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 engineering standards like IEC61508.**

Keywords - systems grammar; ontology; unified semantics; interacting entities.

I. 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 domain specific language. None of the stakeholders has a complete view outside their domain of expertise and is often not able to imagine, what the final system will be. The problem is partly caused by 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 related to natural language. The other 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, a common system grammar can be used. We call this the *meta-ontological* level vs. the domain specific ontological level. Such a level is necessary 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, test, validation, verification and development tasks, architectures and work plans for a system development. The novelty of our approach is that the whole SE process is considered in a unified way. The other novelty is that we introduce a so called *process view* on a system under development. The purpose of the process view is to obtain a correct system design flow at the organisational level. The approach taken is empirically proven by the development of a supporting tool and applying it to divergent domains.

II. Structure of the Paper

The paper is organized as follows. An analysis of related work is presented in the next section. Section 4 describes the base principles of the methodology proposed in the paper. 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. OpenCookbook, as a web portal supporting the proposed formalized SE process, is presented in the section 5. OpenCookbook can guide both the definition and implementation of concrete instantiations of the SE processes. Case studies, which demonstrate that this approach can be applied to different domains, finish this paper together with conclusions and a list of future work.

III. Related Work

The work done with OpenCookbook is closely related to ongoing work in other domains; see, e.g., [1-4]. There are a number of graphical development tools and modelling languages, such as UML [5][6] and SysML [5][7]. Unfortunately, these approaches have a number of shortcomings:

- Most architectural models are developed bottom-up, e.g., as a means of representing graphically what first was defined in a textual format. Hence, such approaches are driven by the system architecture and its implementation. As we discovered in test scenarios, such an approach biases the stakeholders to think in terms of known design patterns, often resulting in suboptimal system solutions.
- Most of the modelling approaches limit themselves to a specific architectural domain, requiring other tools to support the other SE domains. This poses the problem of keeping semantic consistency and hence introduces errors.
- It results in the emergency of a wide range of dialects to fill the "semantic gaps", but in the end these dialects undermine the usefulness of the original standard (this is the reason why from the beginning we based our approach on unified semantics and adopted a restricted architectural paradigm of interacting entities).
- 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 are lacking.

- Most of the tools available on the market bring too many details to the top level, with little support for abstracting away the details. This undermines the power of overview and abstraction.

Overall our methodology 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.

IV. THE PRINCIPLES OF THE METHODOLOGY

A. Systems grammar

A Systems Grammar (SG) is defined as a set of concepts, which provide the base for a coherent and complete description of a system using natural language. The SG in OpenCookbook describes a project in three orthogonal views: requirements and specifications (intentional view), architectural (extensional or design view) and planning view. 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 views on a system beings defined.

Every domain has its own ontology as a set of concepts and relations between these concepts. The ontological level defines concepts which are related to real systems (physical, chemical, software, hardware, etc.). The meta-ontological level defines generic concepts and is expressed in the SE domain by notions such as entity, interaction, requirement, specification, test case, 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.

Let us emphasise here the link between our approach and the ontologies, used in the Semantic Web [8]. There is also a division on domain specific ontologies and abstract meta-ontologies (or *top-level ontologies*), e.g.:

- The Standard Upper Ontology [9];
- Sowa’s top-level ontology [10];
- Cyc’supper ontology [11].

The ontology O of domain d is defined as the set of concepts and relations between concepts $\mathrm{O}^d \triangleq \langle \mathrm{X}, \Re \rangle$.

$X = \{x_l \mid l = 1,...,L\}$ - the finite set of concepts of a domain.

$\Re = \{r_k \mid k = 1,...,K\}$ - the finite set of relations between concepts (e.g., Is-A, Part-Of etc).

An ontology can also include other elements like axioms, constraints, deduction rules, functions of interpretation, etc.

In an ontology the concepts of a domain are divided in classes and individuals (see, e.g., the OWL - Web Ontology Language [12]). The concepts of the system grammar in our approach are similar with the classes of ontology, which are instantiated by individual definitions of a system. The concepts and relations of the system grammar are common for the SE domain, but specific for the level of a system definition. For an intentional view, e.g, we use the concept of *requirement* which is linked with *specification* by the relation “produce”; for design we use *entities* linked by *interactions*; for planning view we use *tasks* linked by implementation relations.

On these relations we put logical conditions, defining the state transitions between different steps of a system definition (see as examples the Figures 1 and 2). Exactly these state transitions are the definition of the *process view*. So the sense of proposed approach is to consider the SE process as the *labelled state transition system*, where labels represent the steps of a system definition.

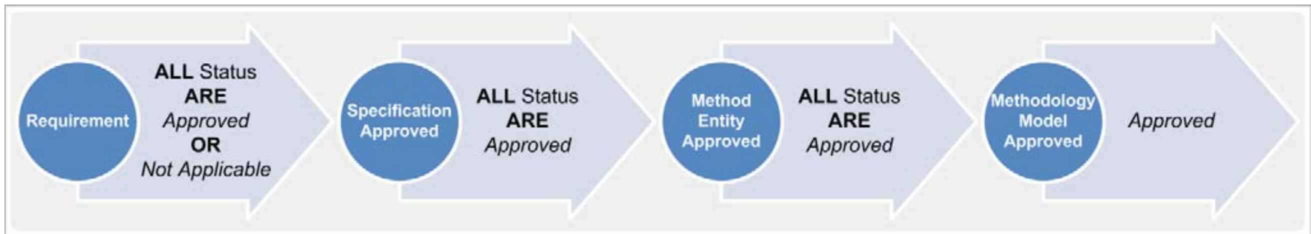

Figure 1. The state transitions at the intentional level of a system definition

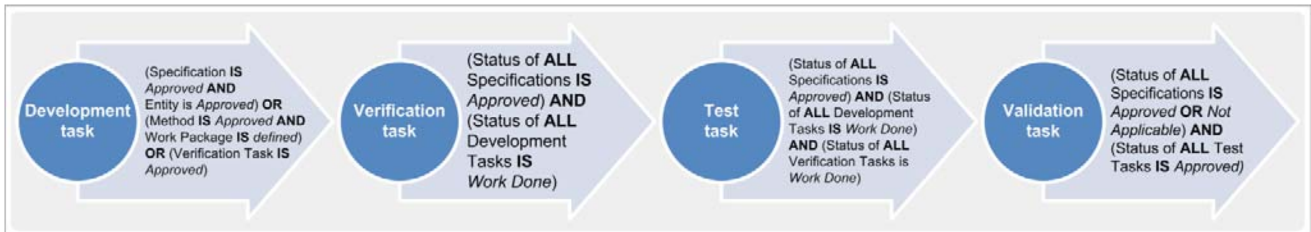

Figure 2. The state transitions at the implementation level of a system definition

The other novelty of our approach is the expansion of the ontological model (system grammar) by adding a set of methods to use it.

$M = \{m_n \mid n = 1, ..., N\}$ - the finite set of methods used in the ontology.

So we define an ontology as: $O^d = \langle X, \Re, M \rangle$

As examples of methods can be considered:

- system description (corresponding to intentional or requirements views);
- system design methods (corresponding to extensional or architectural views);
- validation and verification.

Let us note here, that the general application of ontology in Semantic Web is knowledge reuse.

Our approach can be also used for other aims, e.g., for checking the correspondence of a system definition to standards (this topic will be considered in future papers).

B. Base concepts and methods of the intentional view

Let us pay attention to the proposed methods of system description and design. As was said, Systems Engineering is the process that transforms a need into a working system. Initially, we describe what a system is from an *intentional* (requirements) view. From this perspective we can derive what the system is supposed to be (or to do). Another view is the *extensional* (architectural) one. This perspective shows us how the system should be implemented. The process view defines how to develop a system in the right way, including validation and verification stages (see Figure 3).

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). In the approach, presented in the paper, we call these elements ***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 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 and fault 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, that in the industrial practice the terms requirements and specifications are often not used consistently, which leading to confusion. Some people even use the term "requirement specifications", a rather ambiguous one. Hence, we consistently use "requirements" as the required system properties. Specifications are seen as quantified requirements with their associated test methods.

Capturing requirements and specifications is the most important part of the system description process. Specifications are derived from the more general requirements. This is necessary in order to make requirements verifiable by measurements. For example, 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 events that can result in system faults the system design should cater for. The idea is that by formulating the failure cases, we begin to understand, what can go wrong *before* the real system design starts, which contributes to prevents possible disasters.

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

C. Base concepts and methods of the extensional view

From the extensional or architectural view 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 of entities. 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 represented by ***messages*.** An interaction structure corresponds to a protocol which can be defined by a functional flow diagram or a 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. It can, for instance be the result of an entity attribute change (i.e., a change of the entity's state). A message can cause and can be caused by an event. An interaction changes

the state of all entities involved in the interaction. 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 be of the *input or output type*, which defines the direction of data, energy or information transfer at the interaction between the entities. Examples of interfaces are an electric socket (input: electrical power or current), a fuel pipeline (output from the tank), a USB port (input-output), etc.

Interfaces and interactions are related by the fact, that interfaces transform events, which are internal to an entity, into external messages. A second entity will receive such a message through its interface, transforming the external message into its internal representation (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. 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 implies an energy, matter or information transfer between entities.

D. Linking 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.

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”*).

The transition from the intentional requirements to the extensional architectural level is achieved by abstraction, decomposition, typification, structuring, hierarchy definition and other methods. 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), work plans and tasks, as well as issues to be resolved. The order of this sequence is essential and constitutes the *process view* on a system definition.

Note, at the initial stages of the systems engineering

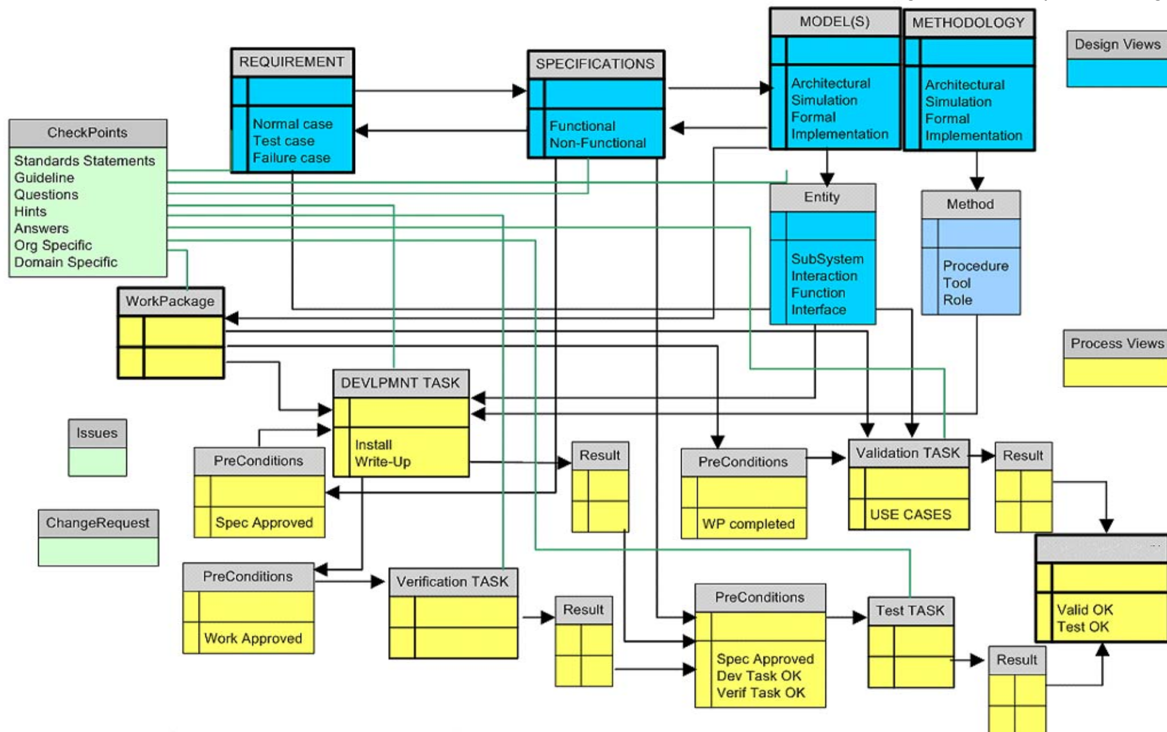

Figure 3. The Design and process views on a system under development

process, a precise architectural decomposition into real entities and interactions does not yet exist. There is only an incomplete cognitive 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.

We propose the next method as a novelty of our SE approach. During the first stage of system definition, we allocate nominative *qualifiers* to be used in the extensional (architectural and work planning) domain. At the intentional phase the architecture definition is nominal - i.e., we only have names in the vocabulary {X} of entities and interactions, which is not yet the real ontological model. This is the first step towards the transition from the intentional to the 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 the analysis 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 to make the transition from the intentional requirements model to the extensional architectural model is a challenging task. The hardest problem is finding architectural relations in an intentional cognitive model, reflecting structural, functional and temporal relations of a real system.

E. Planning, validation and verification views

Another important point of view in SE is the project development view, which in our approach is based on *qualifiers* as result of architectural system decomposition. Once identified, the entities will lead to fulfilment of the specifications are grouped into *work packages*, which in turn are used 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. Some of the tasks are "horizontal" as they are not directly related to specific entities but to methodology requirements. For example, version management is a typical methodology requirement for any safety driven engineering project.

Defining the timeline of the workplan (i.e., deadlines, periods, milestones, etc.) and the tasks are important system development stages. Selecting such measures and attaching them to work packages leads to specifications of a workplan.

We distinguish the following classes of tasks. A development task is the actual development activity, but can include other activities like simulation, prototyping or formal model verification. Following the *process view,* it can only start once the specifications are approved (see Figure 3).

A verification task is defined as the activity that will not verify an implementation but the development task itself. It can be seen as an audit of the development activity and has to verify checkpoints related to the development itself. It answers 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 groups of engineers than those carrying out the development.

A test task will test (according to the test cases of the specifications) the results developed. It can only start once the verification task has been approved.

Finally we consider the validation task. A validation task will validate that the implementation result (after verification and testing was successful) 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 successful, 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.

V. PROTOTYPE DEVELOPMENT

To validate the concepts and its applicability for the SE domain prototype environments were developed using first Plone [13] and next Drupal [14] Open Source Content Management Systems (CMS). The release version is 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.

Being a web based tool, it naturally caters for distributed team work. Other advantages are that existing plug-in modules can be used. At each moment the up to date version of the project documents can be generated.

OpenCookbook supports following activities in the system definition process:

- 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.

VI. CONDUCTED EVALUATION OF THE PROPOSED APPROACH

In order to fine-tune the prototype and to verify its applicability to different domains, a number of partial evaluations were conducted. Projects were defined to develop a Real Time Operating System (OpenComRTOS) and supporting tools [15], 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 tests, in diverse domains, indicate the suitability of the approach. Most issues were related to the ergonomics of the environment and some deficiencies of CMS used.

Our systems engineering approach was also mapping it onto a Business Process Engineering method. Here we found that the unified systems grammar is fully applicable, although often a very different terminology is used or different tools. While a technical engineer might use virtual prototyping or CAD tools to simulate different use 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 the using the OpenCookbook for the formal development of the OpenComRTOS operating system [15]. The formal models used during the development of OpenComRTOS, have a very high abstraction level, but this level of domain abstraction fully corresponds to the meta-ontology used by OpenCookbook.

OpenCookbook supports an incremental way of a modelling process. Starting from a small and very abstract model, refinements and details are added until a model emerges that is very close to the implementation architecture. Each intermediate model can be 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 by all team members (aided by the fact that OpenCookbook is a web-based tool). As a result, the abstraction level gradually 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. Therefore, the result was much cleaner and had more compact systems architecture.

Application of the process view in parallel with the design and the planning views allows us guarantee the correctness of order of steps of the system development process.

CONCLUSIONS

The *meta-ontology*, which provides a unified grammar for a system definition, is proposed. The concept of meta-ontology expands the notion of the top level ontology, as it used in the Semantic Web approach.

With the *design* view, allowing to check if we develop the right system, the *process* view, allowing to check if we develop the system in a right way, is introduced.

The OpenCookbook prototype environment was developed to evaluate the proposed methodology in different SE domains.

FUTURE WORK

Future developments will be devoted to a formalisation of the meta-ontology for the process view. This will give to the user a possibility of developing domain specific methodologies.

Properties of the model of SE processes as the labelled state transition system will be further explored. This will strengthen the theoretical underpinnings of the proposed approach and its further practical implementations.

Mapping between different levels of a system definition will be further developed. Further research is also needed to better understand the interplay between the ontology domain and process domain.

REFERENCES

[1] Alexander Kossiakoff and William N. Systems Engineering Principles and Practice. - Wiley-Interscience. – 2002. - 488 p.

[2] Benjamin S. Blanchard and Wolter J. Fabrycky. Systems Engineering and Analysis (5th Edition). - Prentice Hall. - 2010. - 800 p.

[3] Joseph E. Kasser. A Framework for Understanding Systems Engineering. - BookSurge Publishing. - 2007. - 378 p.

[4] Mark Austin, Vimal Mayank, and Natalia Shmunis. PaladinRM: graph-based visualization of requirement organized for team-based design // System engineering. - Volume 9. - Number 2. - 2006. – page 129.

[5] Tim Weilkiens. Systems Engineering with SysML/UML. Modeling, Analysis, Design. - Morgan Kaufmann Publishers Inc. - 1st edition, 2008. - 320 p.

[6] http://www.uml-forum.com/

[7] http://www.omgsysml.org

[8] Berners-Lee T., Hendler J., and Lassila O. The Semantic Web // Scientific American. - May. – 2001.

[9] http://suo.ieee.org/

[10] http://www.jfsowa.com/ontology/toplevel.htm

[11] http://www.cyc.com/cyc-2-1/cover.html

[12] OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/

[13] www.plone.org

[14] www.drupal.org

[15] Bernhard H.C. Sputh, Oliver Faust, Eric Verhulst, and Vitaliy Mezhuyev. OpenComRTOS: A Runtime Environment for Interacting Entities // Communicating Process Architectures 2009. – IOS Press. – 2009. – pp. 173 – 184.