TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy

SPES 2020 Deliverable D1.2.C-2

Domain Modelling

Overview

Software Plattform Embedded Systems 2020

Author: Daniel Ratiu
Thomas Kofler

Version: 1.0
Date: May 9, 2011
Status: Released

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

About the Document

In model based development, models should be used as primary development artefacts in all development phases. A model is an abstraction of a real thing for a given purpose. The real things to be modeled depend on the addressed domains – e. g., engine controller for cars, or a controller for automating an industrial process. In model-based development the purpose of models is to support different software engineering activities such as requirements engineering, specification, design or testing.

The different views of the system, even if address different (many times heterogeneous) aspects, they all refer to the business domain object to be built. *The common denominator of all models built in a project is the business domain knowledge.*

In this document we motivate the need for explicit use of domain knowledge in the model based development as first class entities. Instead of using codified domain knowledge as is used today only in the names of model elements, our aim is to go further and lift the domain knowledge direct in the modeling languages, up to the level of domain specific modeling languages.

Contents

1 Motivation

In the broadest sense, models are defined to be "abstraction of a real thing for a given purpose". The "real thing" that models usually abstract in the model based development is strongly related to concepts of the business domain of the system. The "purpose" of models is to fulfill different engineering needs during the development. Below we give different examples of models by emphasizing on the "purpose" and the "real thing":

- models for capturing the *requirements* of a *mill*
- models for capturing the *geometrical layout* of a *mill*
- models for capturing the *behaviour* of a *mill*
- models for capturing the *hazards* for a *mill*
- models for capturing the *costs* of an *edger*
- models for capturing the *requirements* of an *edger*

We remark that these two dimensions are orthogonal to each other. Software engineering is a quest for finding and defining models that are adequate for different engineering purposes. What about the subject of the modeling, namely the domain concepts that need to be represented with the models? The currently wide spreaded modeling languages (e. g., use-cases, data-flow specification, state machines, component diagrams, etc.) are well appropriate for representing the engineering purpose. Unfortunately, these modeling techniques are ignorant with respect to the reality that is to be modeled. The models are therefore agnostic and do not capture the domain semantics (intentional meaning) of the models. This meaning is only encoded informally in the names of identifiers of models elements – by anonymizing the names of model elements, we loose (almost) the whole domain meaning of most models. The same engineering tools are used across different domains. We say that the techniques are (mostly) ontologically free.

There is a broader spectrum under which the term "business domain" can be seen. Highly general business domains are for example "embedded systems" or "information systems". More restricted domains are "automotive", "automation", or "avionics". Inside of these specific domains are different sub-domains like: "military aircrafts", "civil aircrafts", "cargo aircrafts", "industrial automation", "electric cars", "classical cars". One can further restrict the business domain to the set of products realized by a company.

There are already pragmatic steps done in the direction of lifting the tools and languages to the domain. Notable progress is done by defining domain specific profiles for general purpose modeling languages like UML. For example, there is a UML profile for modeling and analysis of real-time embedded-systems. A UML profile is defined as:

> "A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain..."[1]

How can we get modeling constructs that are adequate for a domain? In this document we try to sketch a rough answer to this question.

[1] UML2.0 Infrastructure Specification

Software simulates the real world. Software simulates the real world: it acts and respond to the actions of the related systems as such it would know about the instance situation at hand. Parts of the software reference thereby to concepts from the application domain. Thereby is a bridge between the software parts and parameters of the system. Furthermore, software serves the role of an integrator between different sub-systems – e. g., it coordinates, orchestrates different parts of a technical system. Thereby, software parts might reference domain knowledge from different technical domains and views.

In the same manner, software is part of a larger system – it inherently interacts with the real world through actuators. Whenever the surrounding sub-systems change, the changes need to be reflected in the software as well. The software knows inherently about the domain it implements – by reading the software one can understand its domain. Part of this knowledge is captured in the requirements, other parts are not captured at all or are only implicit. For example: Part of the functional requirements naturally address some parts of the system, while other parts are orthogonal. The Orthogonality is always with respect to the chosen decomposition of the system. Therefore software system is part of a bigger system whose changes inherently are reflected in the software – otherwise the system as a whole would be inconsistent. The mediation between the software and the surrounding system is given by the domain knowledge.

In this document we built on the seamless abstraction layers and modelling concepts documents and our aim is to bring them closer to the domain (Figure 1).

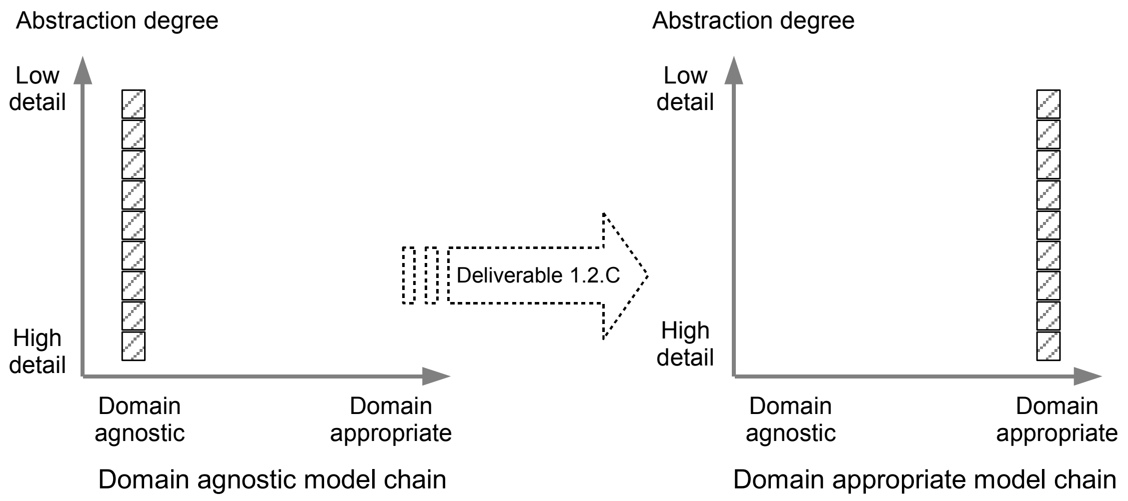

Figure 1: The scope of this deliverable

The need for domain engineering. Engineers/Specialists in a certain domain use a lot of implicit knowledge. Let us suppose that a freshman who just graduated from mechanics gets employed in a company. Supposing that the attended faculty has a reasonable good curricula, everything that our freshman learned in this faculty years is related to the mechanical domain. During the university the freshman has a lot of knowledge about for example the mechanical domain. Everything that this freshman needs to learn in order to be usable in this company belongs to the domain in the small. If engineers/specialists in a certain domain are able to bring their implicit knowledge in an explicit representation, knowledge about a domain is accessible to all employees of the company. Our freshman would easily be able to learn the company-specific part of the needed knowledge to be usable for the company. In the case that an engineer leaves the company, the implicit knowledge is not lost because it is explicitly modeled and therefore available for other engineers or beginners.

Requirements contain domain knowledge. When writing requirements, domain experts make use of domain knowledge. Requirements are nothing else than sentences about domain concepts. Understanding the business domain is a prerequisite of the requirements engineering since one can not write requirements before the application domain is deeply understood.

> "We consider that a careful and explicit description of the environment is an essential first stage in a serious development. This description should capture the behavior and properties of the environment that are independent of the presence and operation of the system to be constructed. In a library, for example, 'books', 'copies', and 'volumes' stand in certain relationships that must be understood by the developers of a system but can not be changed by them; another independent truth about the library is that a copy of a book, once borrowed, can not be borrowed again until it has been returned."[JZ93]

As an example, let's imagine that Jim, a domain expert in banking is reading the requirements for a pumping station. Besides the general English words, and his general background knowledge, Jim would not be able to understand much of these requirements. This is simply due to the fact that the requirements contain (explicitly through the denotation of domain specific words, and implicitly through the connotation of these words) a high amount of domain knowledge. Without this knowledge, a reader cannot understand much from a requirements document.

Domain knowledge has the highest stability. For a given domain, the domain knowledge is the most stable part. Based on the domain knowledge, the requirements are changing continuously. Furthermore, new requirements need to keep up with new technologies.

Requirements of a system are changing with a high speed to satisfy the new customers. Since each requirement represents a relation between several domain concepts, the number of requirements that can be written upon a system highly exceeds the number of domain concepts about that system. Furthermore, the the technologies used to implement the system are permanently developed. What changes at a slower pace is the knowledge about the business domain. In Figure 2 we schematically present this situation: the number of changes in requirements or in the technologies used to the implementation of the system highly exceed the change rate of domain knowledge.

Domain knowledge has an integrative role. During the development process are produced many heterogeneous artefacts that are at different abstraction levels, address different stakeholders and have different focus over the product. The single commonality between these artefacts is the fact that they all describe the same product (domain knowledge).

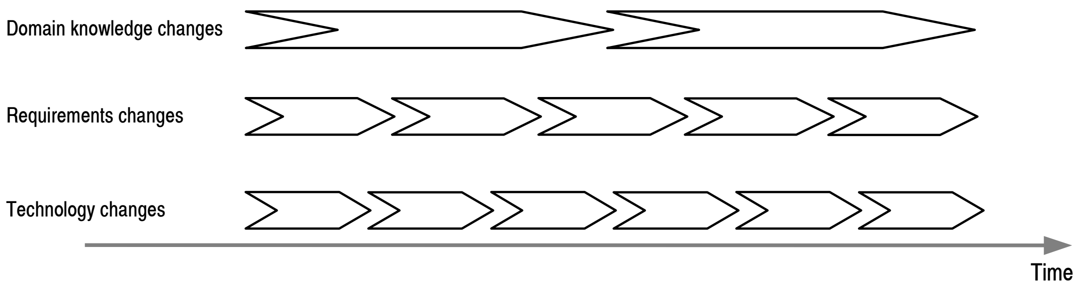

Figure 2: Stability of different development assets

2 A closer look at domain knowledge

Domain knowledge in software engineering is knowledge about the environemnt in which the target system operates. In systems engineering, domain knowledge is the reusable knowledge about the system and it's environment.

In our vision domain knowledge is the knowledge about the domain of interest which all engineering disciplines have in common. This knowledge is knowledge about the system under development including artefacts of different engineering disciplines. The product itself is free of information about artefacts, but the artefacts are linked to the concepts of the system under development.

2.1 Domain appropriateness spectrum

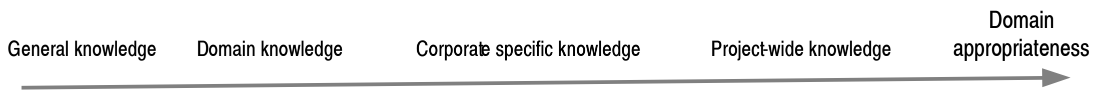

Figure 3: Specialization spectrum of knowledge

When considering a modelling language from the point of view of its adequacy to capture domain concepts, we speak about its domain appropriateness. Domain appropriateness of modelling languages is an important issue to their adoption by domain experts – even if the language appropriateness belongs mostly to pragmatics (and therefore it is less studied in the context of formal methods and MDB). We strongly believe it has a major impact in the adoption and spreading of MDB in the industry.

2.2 Specification spectrum

What should the domain models contain?

- Concepts,
- laws and

- constraints.

What are the kinds of knowledge that exist? What is the spectrum of knowledge?

- domain terminologies – contain concepts that have names (are lexicalized),
- taxonomies – arrange the concepts in an is-a hierarchy,
- weak ontologies – concepts and relations between them,
- constraints and physical laws – make explicit the constraints and physical laws that gouvern the conepts.

2.3 Domain engineering

The following definitions are from SEI-CMU[2]

"Domain engineering is a process for creating a competence in application engineering for a family of similar systems. Domain engineering covers all the activities for building software core assets. These activities include identifying one or more domains, capturing the variation within a domain (domain analysis), constructing an adaptable design (domain design), and defining the mechanisms for translating requirements into systems created from reusable components (domain implementation). The products (or software assets) of these activities are domain model(s), design model(s), domain-specific languages, code generators, and code components."[3]

Domain engineering:

- is the activity for building reusable components,
- is a systematic creation of domain models and architectures,
- for domain engineering to be useful, organizations need to understand the similarities and differences among applications from the same domain,
- supports application engineering which uses models and architectures to build systems.

Domain engineering process is divided into three phases:

- domain analysis: capturing the variation within the domain,
- domain design: constructing an adaptable design,
- domain implementation: mechanisms for translating the requirements into systems created from reusable components.

[2]http://www.sei.cmu.edu/domain-engineering/index.html
[3]http://www.sei.cmu.edu/domain-engineering/domain_eng.html

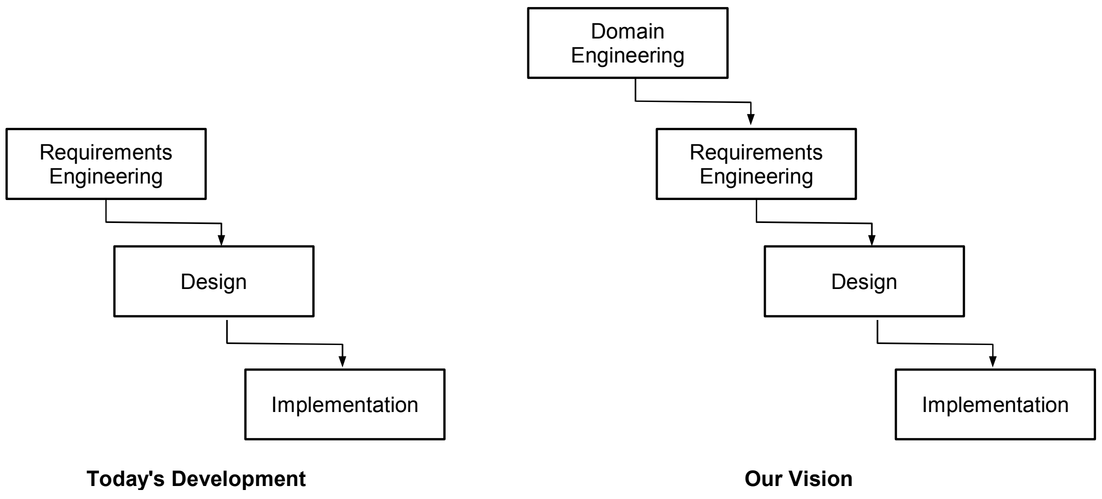

Figure 4: Place of domain engineering in the development process

2.4 Building domain models

How do we stablish the scope of a domain model? Modelling an entire domain is prohibitive, we strongly favour an incremental approach: The more knowledge the more things we can do, howevery many use-cases do not require a high completness degree. To establish the scrope we need to fix a set of competency questions [AGP04].

How do we get the domain model? Domain analysis through interviews with domain experts, reverse knowledge engineering from source code.

Building a domain model is a difficult endeavor, the best way is to build it in an incremental and target oriented manner. In [KR10b] we represented a general methodology for building a domain ontology (domain model). The general steps performed to build a domain ontology are described in [NM01].

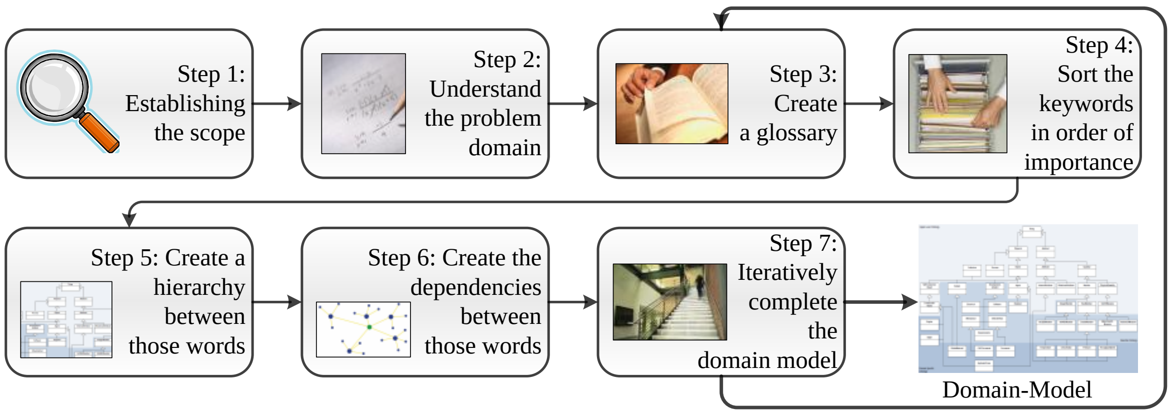
Step 1: Establishing the scope
Step 2: Understand the problem domain
Step 3: Create a glossary
Step 4: Sort the keywords in order of importance
Step 5: Create a hierarchy between those words
Step 6: Create the dependencies between those words
Step 7: Iteratively complete the domain model
Domain-Model

Figure 5: Steps creating a domain model at a glance

In Figure 5 we present these steps at a glance.

1. *Establish the scope of the ontology:*
   Before building a domain model, we have to establish its scope. This would enable us to work in a target oriented manner and would enable us to stop the building at a abstraction/detail level that produces models that are complete enough to perform the analyses and maximize results of the the building effort. In order to establish the scope of the model, we list a set of competency questions that the knowledge contained in the domain model should answer.

2. *Understand the problem domain:*
   The methodology of describing a problem domain is very similar in nearly every domain. A deep understanding of the problem domain is necessary to describe it. First step is to understand the problem domain. There are several ways to do that:

   - The domain engineer tries to understand the problem domain for his own. Since he is no expert regarding the problem domain, there is the possibility that he can't fully understand the problem domain.
   - The domain engineer consults a domain expert, who helps the domain engineer to understand the problem domain.

3. *Create a glossary:*
   Once the domain engineer has an understanding of the problem domain, he creates a list of keywords. These keywords represent the most important concepts for the domain and that are lexicalized (have a word as correspondence). Domain experts use these keywords in communicating information about the domain at hand. To establish a common meaning of the terms used in the domain, all stakeholders have to agree on the common definitions.

   Once a glossary is created, the keywords represent well-defined domain concepts. Thereby, by creating the glossary we lift the words to concepts. In the following, whenever we use "keyword", or "word" we mean the concept that it represents.

4. *Sort the keywords in order of importance:*
   Regarding to the importance of the keywords, we establish an order. This order helps us to identify the main concepts of the problem domain and that will be at first considered in the domain model.

5. *Create a hierarchy between concepts:*
   The concepts should be ordered into a taxonomy based on the is-a relation.

6. *Create the dependencies between concepts:*
   Some of the identified concepts have dependencies between each other. To gain an understanding of the complexity of the problem domain, it is important to discover those dependencies and make them explicit in the ontology.

7. *Iteratively complete the domain model:*
   We favor an incremental approach: start with a rather small set of concepts and incrementally expand this set with new concepts. After each iteration, we need to try to answer the competency questions and depending on the completeness of the answers supported by the current stage of the knowledge base, we might need to expand it with new concepts.

For more details see [KR10b, NM01].

Challenges with introducing the domain knowledge. Incrementality: The more domain knowledge the better the results, we should obtain clear benefits without necessary having comprehensive knowledge bases. There are many scenarios of use of domain knowledge that do not require a fully formalized and completely view over the domain; these scenarios can be arranged incrementally according to the measure in which they require domain knowledge. Automatability and tool support: Human investment in therms of domain experts – domain experts should see the return of investment as soon as possible. Long time for the return of investment – when performing domain engineering the investment is on moddle term. Obtaining commitment – when the domain is captured in detail then the different visions and understandings of stakeholders can be contradictory – the problem of commitment: "Given a domain model, does it correspond to the domain understanding by a certain category of users?". Much domain knowledge / common sense knowledge / knowledge of non-technical nature is ambiguous, reasoning is non-monotonic, probalistic, fuzzy, etc...

3 Use-cases for domain knowledge in systems engineering

In this section we show which benefits can be achieved using explicit domain knowledge in systems engineering. This list is not complete.

3.1 Requirements engineering

In requirements engineering domain knowledge can be used to validate requirements throw evaluation of the usage of domain concepts. Using domain knowledge we are able to report if concepts do not occur in the requirements. Wehn using more intelligent approaches we are able to report if synonyms occure in the requirement document and in the domain model.

In requirements engineering the recorded requirements are often not complete. If using domain knowledge as backbone, implicite knowledge about a concept used in the requirements document can be made explicite throw the domain knowledge.

If requirements violate domain laws, the domain knowledge can be used to complete missing requirements.

3.2 Definitions of domain architecture

" Domain architecture - A generic, organisational structure or design for software systems in a domain. The domain architecture contains the designs that are intended to satisfy requirements specified in the domain model. A domain architecture can be adapted to create designs for software systems within a domain and also provides a framework for configuring assets within individual software systems." [The Free Dictionary]

3.3 Definition of domain specific (modelling) languages

Domain specific languages are at an higher level of knowledge reuse – such languages would allow their users to express directly the domain concepts.

3.4 Backbone for the development

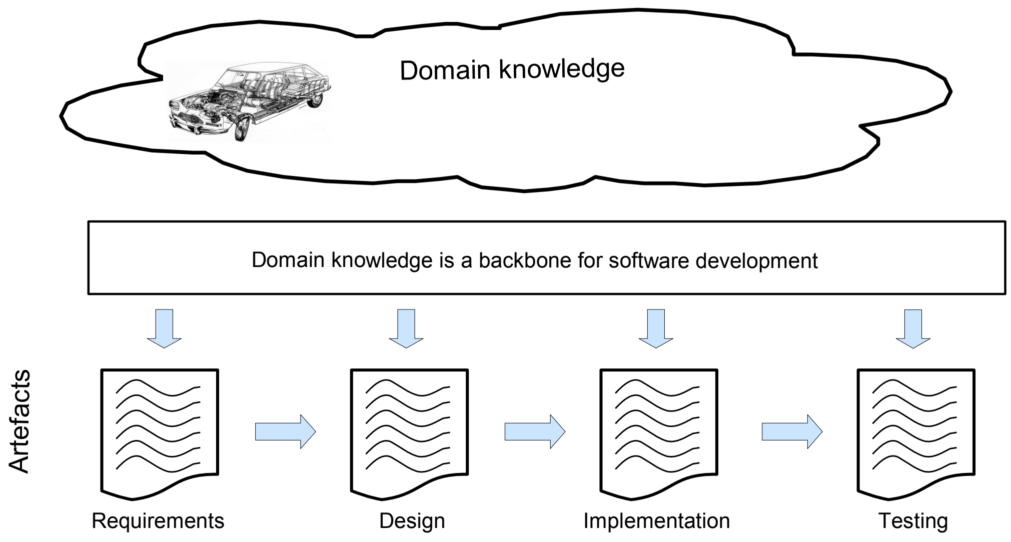

Figure 6: Domain knowledge is contained in all development artefacts

In Figure 6 we illustrate the use of domain knowledge as backbone in the development.

Domain knowledge is a backbone for the development process – from requirements engineering up to maintenance.

Requirements are written by domain experts using domain concepts.

Many thing are only implicit in requirements and the belong to the background knowledge (or general, reusable knowledge) of the domain. Making this knowledge explicit helps to an early validation of requirements, discovery of inconsistencies, improve their completeness and correctness.

Many design decisions are based on domain knowledge – choosing one or another decision is usually done based on the experience and backgroudn knowledge of developers.

Domain knowledge is used in tesing – many tests can test wether specific invariants of the domain are fulfilled by the software.

Software maintenance happens many times more years after the project was started. In the meanwhile, the original developers went away and they brought the knowledge with them. Having the domain knowledge codified helps the sharing and prevents the knowledge decay.

3.5 Traceability

Explicite domain knowledge allowes rich traceability. When domain knowledge is the common model of concepts for different engineering disciplines, it allows rich traceability. Software artefacts use the domain knowledge to link implementation to concepts of the domain model. CAD artefacts use the common domain knowledge to link details about the physical representation to the domain model, we are able to ask questions like: Where is the length of the conveyour reflected in the software model?

Traceability of the domain laws that are necessary for the software for orchestrating different components from different disciplines – e. g., Where is the relation between the time in which the material needs to be transported, the length of the band, the rotation of the motor, the electrical power required to spin the motor, the maximal acceleration that is allowed, etc.

Achieving this Objective. Strategy 1: Enriching the modelling constructs with information about the domain: The domain models speak then at the abstraction/conceptual level of the domain expert. In [KR10a] the authors of this paper describe a reusable unified framework for representing business domain concepts and development artefacts in systems engineering. The conceptual framework of this work can be used as starting point for describing a product in systems engineering which should further be used as the backbone for different disciplines. Classifying domain concepts according to this framework allows rich traceability on the whole model which includes the artefacts of different engineering disciplines.

Strategy 2: Define domain architectures (in terms of libraries of standard components): The domain experts do not start from skartch when describing a situation at hand.

4 Summary

In this document we motivate the need for explicit use of domain knowledge in the model based development as first class entities. Instead of using codified domain knowledge as is used today only in the names of model elements, our aim is to go further and lift the domain knowledge direct in the modeling languages, up to the level of domain specific modeling languages.

References

[AGP04] Oscar Corcho Asunción Gómez-Pérez, Mariano Fernández-López. *Ontological Engineering*. Springer, 2004.

[JZ93] M. Jackson and P. Zave. Domain description. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 56–64, 1993.

[KR10a] Thomas Kofler and Daniel Ratiu. Towards a reusable unified basis for representing business domain knowledge and development artifacts in systems engineering. In *Proceedings of the 2010 international conference on Advances in conceptual modeling: applications and challenges*, ER'10, pages 222–231, Berlin, Heidelberg, 2010. Springer-Verlag.

[KR10b] Thomas Kofler and Daniel Ratiu. Using domain models for dependency management in systems engineering – the rolling mill case study –. Technical report, Technische Universität München, 2010.

[NM01] Natalya F. Noy and Deborah L. McGuinness. *Ontology Development 101: A Guide to Creating Your First Ontology.* Stanford Universty, Stanford, CA, 94305, 2001.