6.1: Methodologies for Ontology Development

Several specific methodologies for ontology development exist following the general idea depicted in Figure 5.1.1, notably the older Methontology and On-ToKnowledge, the more recent NeON and Melting Point methodologies, and the authoring-focused recent ones OntoSpec, DiDOn, and TDDonto, and the older "Ontology Development 101" (OD101)[1]. They are not simply interchangeable in that one could pick any one of them and it will work out well. Besides that some are older or outdated by now, they can be distinguished in core approach, being between:

- micro-level ontology authoring vs. a macro-level systems-view of ontology development;
- isolated, single, stand-alone, ontology development vs. collaborative development of ontologies and ontology networks.

Micro-level methodologies focus on the viewpoint of the details emphasizing formalization aspects, which goes into *ontology authoring*, for it is about writing down the actual axioms and design choices that may even be driven by the language. Macro-level methodologies, on the other hand, emphasize the processes from an information systems and IT viewpoint, such as depicted in Figure 5.1.1. They may merge into comprehensive methodologies in the near future.

Regarding the second difference, this reflects a division between 'old' and 'new' methodologies in the sense that the older ones assume a setting that was typical of 20 years ago: the development of a single monolithic ontology by one or a few people residing in one location, who were typically the knowledge engineers doing the actual authoring after having extracted the domain knowledge from the domain expert. The more recent ones take into account the changing landscape in ontology development over the years, being towards collaboratively building ontology networks that cater for characteristics such as *dynamics, context, collaborative, and distributed development*. For instance, domain experts and knowledge engineers may author an ontology simultaneously, in collaboration, and residing in two different locations, or the ontology may have been split up into inter-related modules so that each sub-group of the development team can work on their section, and the automated reasoning may well be distributed over other locations or remotely with more powerful machines.

The remainder of this section provides an overview of these two types of guidelines.

Macro-level development methodologies

Waterfalls

The macro-level methodologies all will get you started with domain ontology development in a structured fashion, albeit not all in the exact same way, and sometimes that is even intended like that. For instance, one may commence with a feasibility study and assessment of potential economic benefits of the ontology-driven approach to solving the problem(s) at hand, or assume that is sorted out already or not necessary and commence with the actual development methodology by conducting a requirements analysis of the ontology itself and/or find and describe case studies. A well-known instantiation of the generic notions of the development process depicted in Figure 5.1.1, is the comparatively comprehensive Methontology methodology [GPFLC04], which has been applied to various subject domains since its development in the late 1990s (e.g., the chemicals [FGPPP99] and legal domain [CMFL05]). This methodology is for single ontology development and while several practicalities are superseded with more recent and even newer languages, tools, and methodologies, the core procedure still holds. Like Figure 5.1.1, it has a distinct flavor of a waterfall methodology. The five main steps are:

1. Specification: why, what are its intended uses, who are the prospective users
2. Conceptualization: with intermediate representations such as in text or diagrams
3. Formalization: transforms the domain-expert understandable 'conceptual model' into a formal or semi-computable model
4. Implementation: represent it in an ontology language
5. Maintenance: corrections, updates, etc.

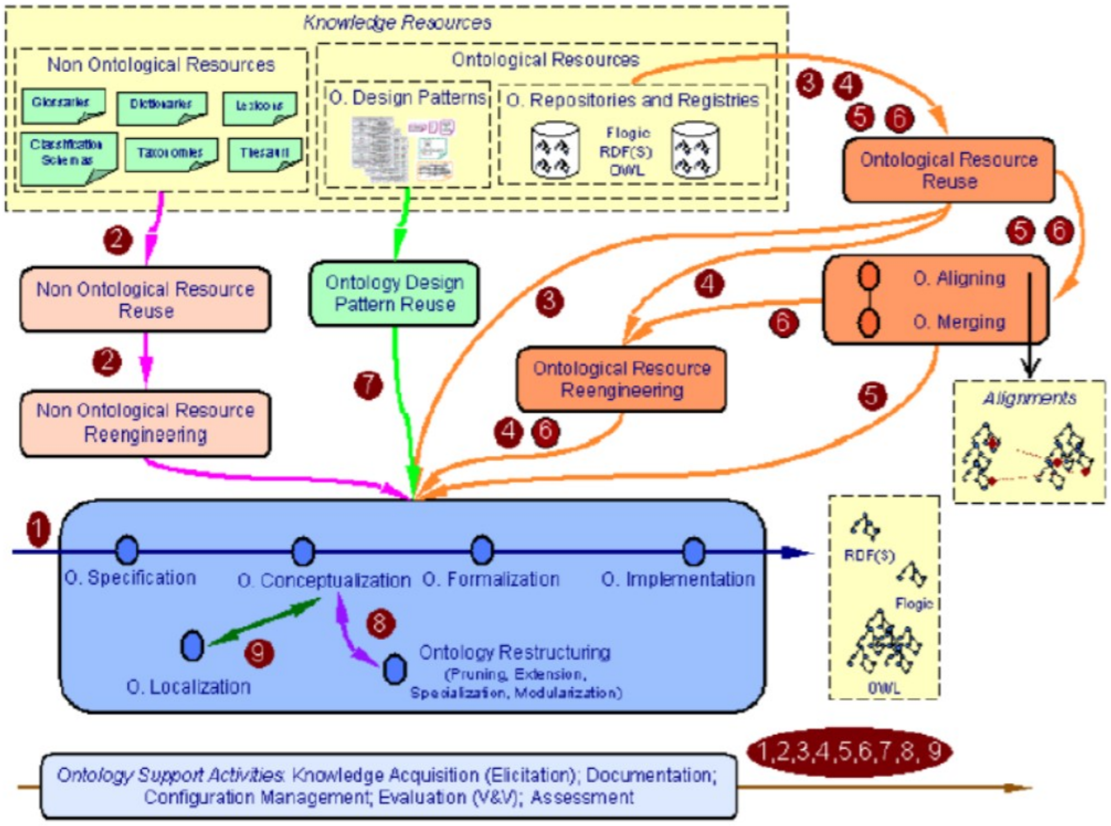
Knowledge Resources
Non Ontological Resources
Glossaries
Dictionaries
Lexicons
Classification Schemas
Taxonomies
Thesauri
Ontological Resources
O. Design Patterns
O. Repositories and Registries
Flogic
RDF(S)
OWL
Ontological Resource Reuse
O. Aligning
O. Merging
Alignments
Non Ontological Resource Reuse
Ontology Design Pattern Reuse
Ontological Resource Reengineering
Non Ontological Resource Reengineering
O. Specification
O. Conceptualization
O. Formalization
O. Implementation
O. Localization
Ontology Restructuring
(Pruning, Extension, Specialization, Modularization)
RDF(S)
Flogic
OWL
Ontology Support Activities: Knowledge Acquisition (Elicitation); Documentation; Configuration Management; Evaluation (V&V); Assessment
1,2,3,4,5,6,7,8, 9

Figure 5.1.2: *Graphical depiction of several different steps in ontology development, where each step has its methods and interactions with other steps (Source: [SFdCB$^+$08])*

In addition, there are various management activities, such as planning activities, control, and quality assurance, and supporting tasks, such as documentation and version control. Ontology management may vary somewhat across the methodologies, such as helping with development of a Gantt chart for several ontology development scenarios. A refinement over the years is, among others, the better provision of 'intermediate representations'; e.g., the MOdeling wiKI MoKi [GKL$^+$09] has a feature for automatic translation between formal and semi or informal specifications by the different experts, which is also reflected in the interface so as to let domain experts, ontologists, and logicians work together on a single project, which is further facilitated by chat-like features where discussions take place during the modeling stage [DGR12].

Methontology is, practically, superseded by the NeON methodology. Instead of the straight-forward five steps, there are many possible routes composed of multiple steps; see Figure 5.1.2. Various development scenarios are then specified by combining a subset of those steps and in some order, which results in a different planning of the ontology activities. Each number in Figure 5.1.2 denotes a scenario. For instance, Scenario 4 is that of "Building ontology networks by reusing and reengineering ontological resources." [SFdCB$^+$08]. How to actually do this Scenario 4, is another matter. For instance, one may/will have to

a) be able to find ontologies in the domain of interest, evaluate them on the relevance and choose which would be the best fit—a research problem of its own (see, e.g., [KG17, KK12, McD17]);

b) extract only a module from an ontology, rather than reusing the whole ontology (see, e.g., [Daw17] for a recent overview, new methods, and tools);

c) convert the representation language of the ontology; e.g., from OWL 2 DL to OWL 2 QL, or from OBO to OWL; and

d) align ontologies, which is even a sub-field within ontology engineering that is large and active enough for a second edition of a handbook [ES07].

Each of these tasks have their own theoretical foundations, methods, and tools.

NeON also includes more details for the specification stage, especially with respect to so-called *Competency Questions* (CQs). CQs, first introduced in [GF95], specify the questions one's ontology should be able to answer and therewith what knowledge the ontology should contain. For instance, with the AWO, one may want the ontology to be able to answer "Which animal eats which other animal?" and "Which animals are endangered?". The AWO you have inspected does contain some information to answer the former (lions eat impalas), but not the latter, for it does not contain information about endangered species.

NeOn also has a "Glossary of Activities", identifying and defining 55 activities when ontology networks are collaboratively built, such as ontology localization (for another natural language), alignment (linking to another ontology), and diagnosis (of errors), which are divided into a matrix with "required" and "if applicable" [SFdCB$^+$08].

Not even the NeON methodology covers all options—i.e., all the steps and all possible permutations at each step—that should be in an ontologist's 'tool box', though. For instance, some mention "non-ontological resource reuse" for bottomup ontology development (number 2 in Figure 5.1.2), and note NLP and reuse of thesauri, but lack detail on how this is to be done—for that, one has to search the literature and look up specific methods and tools and the other bottom-up routes (the topic of Chapter 7) that can, or have to be, 'plugged in' the methodology actually being applied. A glaring absence from the methodologies is that none of them incorporates a 'top-down' step on foundational ontology use to enforce precision and interoperability with other ontologies and reuse generic classes and object properties to facilitate domain ontology development. We will look at this in some detail in Chapter 6. For the older methodologies this may be understandable, given that at the time they were hardly available, but it is a missed opportunity for the more recent methodologies.

Lifecycles

A recent addition to the ontology development methodology landscape is the Ontology Summit 2013 Communiqué's[2] take on the matter with the *ontology lifecycle model*; see Figure 5.1.3. Each stage has its own set of questions that ought to be answered satisfactorily. To provide a flavor of those questions that need to be answered in an ontology development project, I include here random selection of such questions at several stages, which also address evaluation of the results of that stage (see the communiqué or [N$^+$13] for more of such questions):

- Requirements development phase
  - – Why is this ontology needed? (What is the rationale? What are the expected benefits)?
  - – Are there existing ontologies or standards that need to be reused or adopted?
  - – What are the competency questions? (what questions should the ontology itself be able to answer?)
- Ontological analysis phase
  - – Are all relevant terms from the use cases documented?
  - – Are all entities within the scope of the ontology captured?
- System design phase
  - – What operations will be performed, using the ontology, by other system components? What components will perform those operations? How do the business requirements identified in the requirements development phase apply to those specific operations and components?
  - – How will the ontology be built, evaluated, and maintained? What tools are needed to enable the development, evaluation, configuration

management, and maintenance of the ontology?

At this point, you are not expected to be able to answer all questions already. Some possible answers will pass the revue in the remainder of the book, and others you may find when working on the practical assignment.

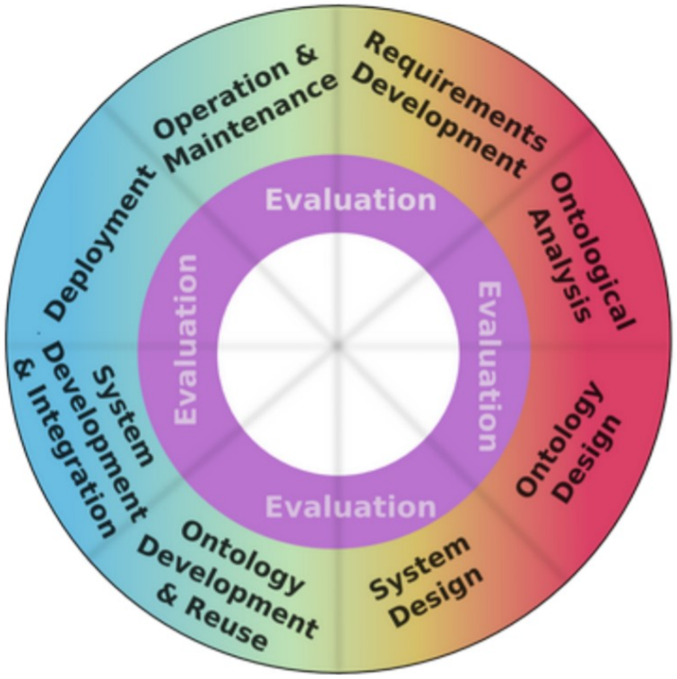

Figure 5.1.3: *Ontology Summit 2013's lifecycle model (Source: ontolog.cim3.net/cgi-bin/wiki...2013Communique)*

Agile

Agile approaches to ontology development are being investigated at the time of writing this book. That is, this is in flux, hence, perhaps, too early at this stage to give a full account of it. For some preliminary results, one may wish to have a look at, e.g., the Agile-inspired OntoMaven that has OntoMvnTest with 'test cases' for the usual syntax checking, consistency, and entailment [PS15], simplified agile [Per17], a sketch of a possible Test-Driven Development methodology is introduced in [KL16], and eXtreme Design was added to NeON [PD+09].

It is beyond the current scope to provide a comparison of the methodologies (see for an overview [GOG+10]). Either way, it is better to pick one of them to structure your activities for developing a domain ontology than using none at all. Using none at all amounts to re-inventing the wheel and stumbling upon the same difficulties and making the same mistakes developers have made before, but a good engineer has learned from previous mistakes. The methodologies aim to prevent common mistakes and omission, and lets you to carry out the tasks better than otherwise would have occurred without using one.

Micro-Level Development

OntoSpec, OD101, and DiDOn can be considered 'micro-level' methodologies: they focus on guidelines to formalize the subject domain, i.e., providing guidance *how* to go from an informal representation to a logic-based one. While this could be perceived to be part of the macro-level approach, as it happens, such a 'micro-level view' actually does affect some macro-level choices and steps. It encompasses not only axiom choice, but also other aspects that affects that, such as the following ones (explained further below):

1. Requirements analysis, with an emphasis on purpose, use cases regarding expressiveness (temporal, fuzzy, n-aries etc.), types of queries, reasoning services needed;
2. Design of an ontology architecture (e.g., modular), distributed or not, which (logic-based) framework to use;
3. Choose principal representation language and consider encoding peculiarities (see below);
4. Consider and choose a foundational ontology and make modeling decisions (e.g., on attributes and n-aries as relations or classes; Chapter 6);
5. Consider domain ontology, top-domain level ontology, and ontology design pattern ontology reuse, if applicable, and any ontology matching technique required for their alignment;
6. Consider semi-automated bottom-up approaches, tools, and language transformations, and remodel if needed to match the decisions in steps 3 and 4 (Chapter 7);
7. Formalization (optionally with intermediate representations), including:
    1. examine and add the classes, object properties, constraints, rules taking into account the imported ontologies;
    2. use an automated reasoner for debugging and detecting anomalous deductions in the logical theory;
    3. use ontological reasoning services for ontological quality checks (e.g., OntoClean and RBox Compatibility);
    4. add annotations;
8. Generate versions in other ontology languages, 'lite' versions, etc., if applicable;
9. Deployment, with maintenance, updates, etc

Some of them are incorporated also in the macro-level methodologies, but do not yet clearly feature in the detail required for authoring ontologies. There is much to say about these steps, and even more yet to be investigated and developed (and they will be revised and refined in due time); some and its application to bio-ontologies can be found in [Kee12b].

For the remainder of this section, we shall consider briefly the language choice and some modeling choices on formalizing it, in order to demonstrate that the 'micro' is not a 'single step' as it initially might seem from the macro-level methodologies, and that the 'micro' are not simply small trivial choices to make in the development process.

The Representation Language

Regarding formalization, the first aspect is to choose a suitable logic-based language, which ought to be the optimal choice based on the required language features and automated reasoning requirements (if any), that, in turn, ought to follow from the overall purpose of the ontology (due to computational limitations), if there is a purpose at all [Kee10a]. Generalizing slightly, they fall into two main group: light-weight ontologies—hence, languages—to be deployed in systems for, among others, annotation, natural language processing, and ontology-based data access, and there are 'scientific ontologies' for representing the knowledge of a subject domain in science, such as human anatomy, biological pathways, and data mining [D+10, HND+11, KLd+15, RMJ03]. More importantly for choosing the suitable language, is that the first main group of ontologies require support for navigation, simple queries to retrieve a class in the hierarchy, and scalability. Thus, a language with low expressiveness suffices, such as the Open Biological and biomedical Ontologies' obo-format, the W3C standardized Simple Knowledge Organization System (SKOS) language [MB09], and the OWL 2 EL or OWL 2 QL profile [MGH+09]. For a scientific ontology, on the other hand, we need a very expressive language to capture fine-grained distinctions between the entities. This also means one needs (and can use fruitfully) more reasoning services, such as satisfiability checking, classification, and complex queries. One can choose any language, be it full first order predicate logic with or without an extension (e.g., temporal, fuzzy), or one of the very expressive OWL species to guarantee termination of the reasoning services and foster interoperability and reuse with other ontologies. The basic idea is summarized in Figure 5.1.4, which is yet to be refined further with more ontology languages, such as the OWL 2 RL profile or SWRL for rules and the $\mathcal{DLR}$ and $\mathcal{CFD}$ families of DL languages that can handle n-ary relationships (with $n \geq 2$) properly.

The analysis of the language aspects can be pushed further, and one may wish to consider the language in a more fine-grained way and prefer one semantics over another and one ontological commitment over another. For instance, assessing whether one needs access to the components of a relationship alike UML's association

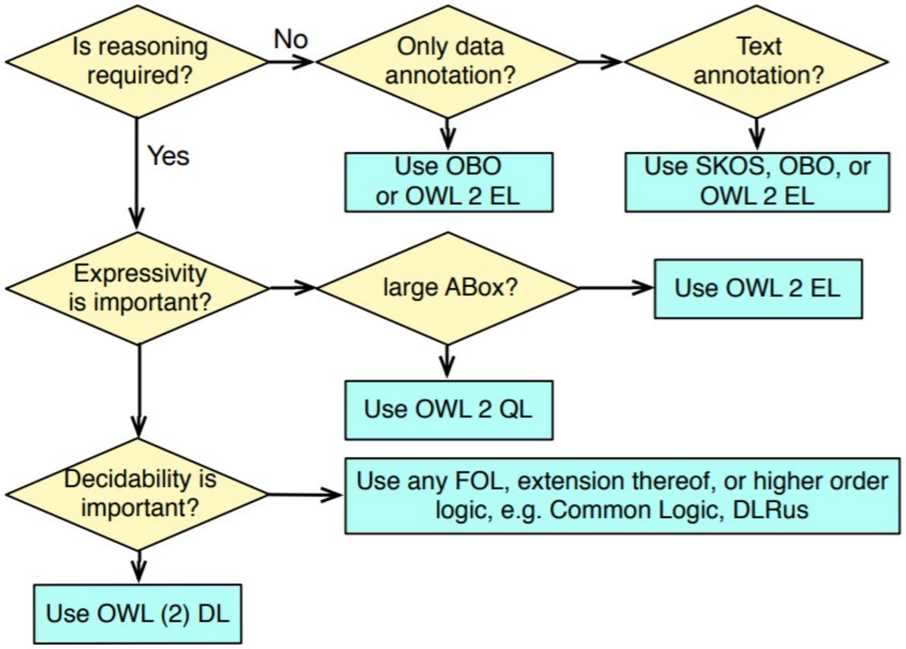
Is reasoning required?
No
Only data annotation?
Text annotation?
Yes
Use OBO or OWL 2 EL
Use SKOS, OBO, or OWL 2 EL
Expressivity is important?
large ABox?
Use OWL 2 EL
Use OWL 2 QL
Decidability is important?
Use any FOL, extension thereof, or higher order logic, e.g. Common Logic, DLRus
Use OWL (2) DL

Figure 5.1.4: *A preliminary decision diagram to choose a suitable ontology language for one's prospective ontology, with indications of current typical usage and suggestions for use. (Source: extended from [Kee12b])*

ends, the need for n-aries, or whether asymmetry is essential, and, e.g., graph-based versus model-theoretic semantics. This is interesting from a logic and philosophical perspective at a more advanced level of ontology engineering and research, which we will not cover in this introductory course to a practically usable detail.

Encoding Peculiarities

This is tricky to grasp at the start: there may be a difference between what the domain expert sees in the tool—what it is 'understood to represent'—and what you, as the computer scientist, know how it works regarding the computational representation at the back-end that a domain expert need not know about. Ontologies need not be stored in an OWL file. For instance, it may be the case that a modeler sees an ontology in the interface of a software application, but those classes, relations, and constraints are actually stored in a database, or an n-ary relationship is depicted in the diagrammatic rendering of the ontology, but this is encoded as 3 binaries behind the scenes. The former plays a trick logically: in that case, mathematically, classes are stored as instances in the system (not classes-as-instances in the ontology!). For instance, `Chair` may be represented in the OWL ontology as the class `Chair` , but one equally well can store `Chair` in a database table, by which it mathematically has become an instance when it is a tuple or a value when it is stored in a cell, yet it is 'thought of' and pretended to be a universal, class, or concept in the graphical interface. This is primarily relevant for SKOS and OBO ontologies. Take the Gene Ontology, among others, which is downloadable in OBO or OWL format—i.e., its taxonomy consists of, mathematically, classes—and is available in database format—i.e., mathematically it is a taxonomy of instances. This does not have to be a concern of the subject domain experts, but it does affect how the ontology can be used in ontology-driven information systems. A motivation for storing the ontology in a database, is that databases are much better scalable, which is nice for querying large ontologies. The downside is that data in databases are much less usable for automated reasoning. As an ontology engineer, you will have to make a decision about such trade-offs.

There is no such choice for SKOS 'ontologies', because each SKOS concept is always serialized as an OWL individual, as we shall see in Chapter 7. One has to be aware of this distinction when converting between SKOS and OWL, and it can be handled easily in the application layer in a similar way to GO.

One also could avail of "punning" as a way to handle second-order logic rules in a first-order setting and use the standard reasoners instead of developing a new one (that is, not in the sense of confusing class as instance, but for engineering reasons), or 'push down' the layers. This can be done by converting the content of the TBox into the ABox, encode the second-order or meta rules in the TBox, and classify the classes-converted-into-individuals accordingly. We will come across one such example with OntoClean in Section 5.2: Methods to Improve an Ontology's Quality "Philosophy-Based Methods: OntoClean to Correct a Taxonomy".

In short: one has to be careful with the distinction between the 'intended meaning' and the actual encoding in an implemented system.

On Formalizing It

The '*how to formalize it*?' question is not new, neither in IT and Computing [Hal01, HP98] nor in logic [BE93], and perhaps more of those advances made elsewhere should be incorporated in ontology development methodologies. For ontologies, they seem to be emerging as so-called *modeling styles* that reflect formalization choices. These formalization choices can have a myriad of motivations, but also consequences for linking one's ontology to another or how easily it is to use the ontology in, say, an OBDA system. The typical choices one probably has come across in conceptual modeling for database systems or object-oriented software also appear here, as well as others. For instance:

– will you represent 'marriage' as a class `Marriage` or as an object property `isMarriedTo` ?

– will you represent 'skill' as a class `Skill` , as an object property `hasSkill` , or as a data property (attribute) with values?

At this stage, it may look like an arbitrary choice of preference or convenience. This is not exactly the case, as we shall see in Chapter 6. Others have to do with a certain axiom type or carefulness:

– On can declare, say, `hasPart` and `partOf` and state they are inverses, i.e., adding two vocabulary elements to the ontology and the axiom $\texttt{hasPart} \equiv \texttt{partOf}^-$. In OWL 2, one also could choose to add only one of the two and represent the other through an inverse directly; e.g., with only `hasPart` , one could state $\texttt{C} \sqsubseteq \texttt{hasPart}^-.\texttt{D}$ "Each C is part of at least one D", i.e., `partOf` would not be a named object property in the ontology.

– The Pizza Ontology Tutorial cautions against declaring domain and range axioms, mainly because the inferences can come as a surprise to novice ontology developers. Should one therefore avoid them? Ideally, no, for this results in a lower precision and actually may hide defects in the ontology.

– n-aries ($n \geq 3$) cannot be represented fully, only approximated, in OWL and there are different ways to manage that, be it through reification or choosing another logic.

Whichever way you choose to represent a particular recurring pattern, do try to do it consistently throughout. There are some methods and tools to assist with such matters, which will be introduced gradually, starting with the next section.

Footnotes

[1]details can be found in [FGPPP99, SSSS01, SFdCB+08, GOG+10, Kas05, Kee12b, KL16, NM01], respectively.

[2]ontolog.cim3.net/cgi-bin/wiki...013_Communique