Towards Evolvable Documents with a Conceptualization-Based Case Study

Marek Suchánek and Robert Pergl

Faculty of Information Technology
Czech Technical University in Prague
Prague, Czech Republic
Email: marek.suchanek,robert.pergl@fit.cvut.cz

***Abstract*—Documents surround us in our everyday lives and affect us even without noticing it. Information technology brought an evolution to documents in terms of flexibility and efficiency in their composing, processing, and sharing. However, in these days, an electronic document lacks the evolvability and reusability of its parts. Maintaining the consistency across one or even several documents and their versions makes it a very complicated task. We encounter a similar problem in the software development domain where, however, effective principles and techniques have been developed and adopted. Incorporating modularity, design patterns, loose coupling, separation of concerns, and other principles are being successfully applied to achieve evolvability. Results are proven in decades by scientific research and countless practical applications. Hypothetically, such principles may be used also for documents in order to achieve reliable and easy-to-maintain documents. This paper presents our generic conceptualization leading to evolvable documents and which is applicable in any documentation domain based on related work in the electronic documents, as well as the evolvable software development domains. Advantages and core ideas of our conceptualization are then demonstrated in a case study – prototype design of OntoUML modelling language documentation. Finally, possible next steps for generic evolvable documents are proposed, as we perceive our contribution as the first step in the journey towards evolvable documents in the scientific point of view. The results from this paper can be used for further research and as the first boilerplate for designing custom evolvable documentation.**

Keywords–Electronic Documents; Evolvability; Modularity; Conceptualization; OntoUML; Case Study; Separation of Concerns.

I. Introduction and Motivation

Documents are a vital carrier for storing and distributing knowledge – the precious result of various human activities. The number of documents grows rapidly primarily due to their "cheapness" in the digital era. However, an interesting observation may be made: In spite of various means of storing, retrieving, and sharing documents in electronic forms, the foundations did not change, and the documents are the same hard-to-maintain and evolve structures as they always were. Imagine, for example, a document capturing regulations of a study program enrolment. Such a document is issued and maintained by the Dean of a faculty. However, it must be compliant with the university's regulations document, which in turn must be compliant with the regulations of the Ministry of Education. We have three levels of documents where the more specific ones contain parts of the more general ones, take them as-is or elaborate more specific versions, add further regulations, and so on. Now, imagine that there is a change in the Ministry's regulations, which must be appropriately dealt with in the referring documents. This situation affects at least dozens of Faculty's agendas which results in inefficiency, inconsistency, and other related problems.

This paper is an extension of the previous conference paper [1] by extending related work, broadening the initial conceptualization, and (the most importantly) introducing a conceptualization-based case study – draft of the evolvable OntoUML documentation. OntoUML is an ontology-based modelling language used also for expressing the conceptualization in this paper [2].

The first observation is that documents are seen as monolithic wholes or wholes composed of highly coupled parts which cannot be separated or even reused. If we would be able to decouple parts of documents, make them loosely coupled just by higher concerns and design them as reusable, it would significantly help in many domains, such as teaching materials, corporate documents, manuals, or regulations. The practice of software engineering suggests that if done properly, evolvability may be significantly improved, the efficiency of document management gained, and error rate decreased [3].

In Section II, we first briefly introduce a wide variety of related work affecting documents domain in terms of the modularity and evolvability. Section III is divided into three steps of our approach to create a generic conceptualization, i.e., independent on a type of enterprise or domain involved. We apply concepts from theories used in computer science and software engineering verified by practice. Furthermore, we build our approach on the Normalized Systems (NS) theory [3], which is dealing with evolvability of information systems and it has been reported to be successfully applied in other domains than software development including documents [4]-[5]. In Subsection III-A, we split the domain into key parts and then, in Subsection III-B, we introduce conceptual models for them using the ontologically well-founded conceptual modelling language OntoUML [2]. After this exploratory and inductive part, Section IV demonstrates the case study that applies the previously described ideas and findings from the related work. Finally, Section V contains deduced possible and potentially suitable next steps and future work.

II. Related Work

Over the years of Information and Communication Technologies (ICT) field development, many solutions for working with documents and documentation emerged [6]. In this section, we discuss some key areas and approaches related to electronic documents. This review of the current state-of-the-art provides a foundation for our conceptualization of the documents problem domain in general.

Nowadays, there are many different text processing tools, syntaxes and complex systems for dealing with documents within their whole life-cycle [6]. The goal of this part is not to describe particular existing solutions, but to emphasize essential and interesting approaches or ideas that should be considered before developing new solutions. All of the mentioned approaches strive to make dealing with documents simpler and more effective. In the following conceptualization and the case study, we will take those observations into account.

A. Formats and Syntax

There is a plethora of markup languages and document encoding possibilities providing different advantages: some are focused to be easily readable in plain text, and others provide ways to encode complex document elements [6]. From using basic annotations to mark headings and lists emerged simple markups such as Markdown or more complex as AsciiDoc or reStructuredText. Interesting ly, last two named share a concept of extensions that is similar to LaTeX commands/environments or Word macros. Such feature is essential for building complex documents. But when compared to LaTeX and office suites, it is much more flexible and easier to process due to the good human and machine-readability in a plain text [7][8].

Another interesting concept of versatility and evolvability is represented by the Pillar markup language for the Pharo environment [9]. It consists of a document model which is easily extensible by implementing new classes and visitors defining syntactic constructs meaning and handling. Furthermore, the provided tool allows export in many other formats and markups. Sadly, the syntax is not very common and Pillar is widely used only in the Pharo community.

When it comes to a specific format, community size and available well-maintained tooling are crucial. Converting between formats is also important to mention. A great example of a markup converter is Pandoc, which enables conversion from over 20 formats to more than 30 formats [10]. The smaller number of input formats illustrates the fact that some of them are harder to process. At the same time, an output format of a document should be expressive and extensible. For example, LaTeX has mechanisms of custom packages and commands, environments, and macros. It is then a considerable challenge to convert it to another format lacking these extensions [11].

B. Templates and Styles

Separation of a graphical design and content is the first notion of separation of concerns in documents. This separation – as well as splitting the document into parts (or modules) – makes it easier to maintain and to keep track of changes. Dealing with style when writing a document is extra overhead that should be done separately. Only the meaning should be expressed by the text, for example, marking text as important instead of decorating it as bold. Then, in some template, such important text then can be rendered in red colour, underlined or different font without stating that it should not be bold. This applies to every possible semantics in a text [6].

A document, or any piece of data in general, can be rendered using an independent template associated with one or various styles. This approach can be seen in many documentation systems and languages, such as Extensible Markup Language (XML), HyperText Markup Language (HTML) and Cascading Style Sheets (CSS), LaTeX or even in various *What-you-see-is-what-you-get* (WYSIWYG) Office suites. This separation leads to good evolvability of document structure and style without touching the content itself [6][11].

Using templates with styles to easily form and design complex structures is well observable in the field of web development. Many web frameworks are supplied with one of many template engines, namely, Twig, Jinja, JavaServer Pages, Mustache, or other. Template engine takes structured data and a template as input and produces a rendered document, e.g., query result in the form of HTML document with table or JavaScript Object Notation (JSON) array based on the request. Moreover, it is usually possible to extend and compose templates together, and to create reusable components and macros [12].

C. Sharing and Collaboration

Documents are often written by more than one person. Collaboration possibilities are related to the format used. If the document files are in plain text, then one of the solutions is to use Git or other version control system (VCS) [13]. There are also many cloud services allowing users to create and edit documents collaboratively, for instance, Google Documents, Dropbox Paper, Overleaf, or Microsoft Office Online. Both types of solutions help maintain consistency of document versions in a distributed authoring set-up.

When mentioning Git and other VCSs, it is important to emphasize that they already provides a lot of functions that a powerful document system needs [13][14]. Such features are among others:

- tracking of history and comparing changes of version,
- tagging a specific version,
- signing and verifying changes,
- looking up who changed a particular line of text,
- working with multiple sources/targets and linking other projects submodules,
- logging and advanced textual or binary search within the changes,
- allowing changes in multiple branches,
- merging or combining changes.

Moreover, services like GitLab, GitHub, or BitBucket provide more collaborative tools for issues, change reviews, project management, and other services integrations. One of the important related services types is continuous integration (CI), which allows the building, checking, and distribution of results seamlessly. It can be used for example to compile the LaTeX document and send the Portable Document Format (PDF) to a file server or email address [14][15].

D. Document Management Systems and Wikis

A document management system (DMS), as explained in [6] and [16], is an information system that is able to manage and store documents. Most of them are capable of keeping a record of the various versions created and modified by different users. The term has some overlap with the notion of content management systems. It is often viewed as a component of enterprise content management (ECM) systems and related to digital asset management, document imaging, workflow systems and records management systems.

One of the leading current DMS is an open-source system named Alfresco that provides functionality such as storing, backing up, archiving, but also ISO standardization, workflows, advanced searching, signatures and many others [17]. From our perspective, the problem is that DMSs are mainly focused just on working with a document as a whole that lacks finer-grained modularity necessary for evolvability itself.

Knowledge can be gathered, formatted, and maintained in a Wiki – a website allowing users collaboratively modify content and structure directly from the web browser [18]. Wikis are extensible and simple-to-use sets of pages that can be edited in a WYSIWYG editor or manually with some simple or custom syntax, e.g., Markdown, reStructuredText, or DokuWiki. The system keeps track of changes within pages as well as the attachments, so it enables the comparison differences and see who changed the document and when they did. Common extensions of Wikis are tools for exporting to various formats or extending syntax and other user-friendly functionality [19]. There are many diverse commercial and open-source solutions with slightly different functionality. Commercial solutions are often called enterprise content management and consist of a Wiki system and a DMS to manage documents in a better way than just with a plain DMS [16].

E. The Normalized Systems Theory

The Normalized Systems theory [3] deals with modularity and evolvability of systems and information systems specifically. It introduces four principles in order to identify and eliminate combinatorial effects (i.e., dependencies that are increasing with the system size):

- Separation of Concerns
- Data Version Transparency
- Action Version Transparency
- Separation of States

Applying the principles leads to evolvable systems composed of fine-grained and reusable modules. In the documents domain, mainly the first two principles are applicable [5], because actions and states are workflow-related. There is no workflow “inside” the document as we know from information systems, but documents are often subject or object in some workflow (e.g., passing a document between activities or approving draft to the final version).

The principles and concepts of the theory have been reported to be used in other domains, such as study programs [4] and documents [20]. In the paper [5], it is shown in a form of the prototype, how the theory can be used (especially the separation of concerns and creating modular structures) in the domain of documents for study programs. The prototype is able to combine selected fine-grained independent modules and to generate a resulting LaTeX document.

Theoretical foundations of the Normalized Systems theory are applicable also in other domains, even those that are non-ICT related. A typical example of such domain is streets made of building blocks or multi-stage rockets as described in [3]. It is fairly easy to find countless examples of everyday-use systems that could or should be normalized and it gives opportunities to further research in various domains.

F. Aspect-Oriented programming

Aspect-oriented programming (AOP) is a programming paradigm that uses the separation of cross-cutting concerns to modularize software. It has similarities to the Normalized Systems theory and their solution in terms of so-called *join points*. Using this paradigm allows adding new behaviour to an existing code without changing it directly but plug it into a specific place in the code. One of the typical examples is logging as a cross-cutting concern; there can be many different logging implementations and the chosen one can be plugged into the code without changing anything else. The final code is then composed by the *weaver* that generates object-oriented code with integrated aspects from the aspect-oriented code. Apparently, for successful AOP, appropriate and high-quality tooling and language are necessary [21].

The core ideas of AOP are interesting also for other domains than software development. Having tools and solution allowing seamlessly plugging-in additional functionality undoubtedly increases efficiency. For the documents domain, this could be reflected as a possibility to add new document-parts and even new types of document-parts into the existing document with tools composing everything together in a similar way as, for example, it is in LaTeX.

G. Source Code Documentation

Basically, for every widely-used programming language, there are one or more systems for building a documentation from annotations and comments that are placed directly in a source code. Such systems are, for example, Javadoc for Java, Doxygen for C/C++, Sphinx for Python (see Listing 1), or Haddock for Haskell.

Listing 1. Documentation of Python source code

```
class Person:
"""This is simple example Person class

You can create new person like this:

.. code::

bd = datetime.datetime(1902, 1, 1)
p = Person("Peter Pan", bd)

:ivar name: Full name of the person
:vartype name: str
:ivar birthdate: Birthdate of the person
:vartype birthdate: datetime
"""

#: Number of people instantiated
people = 0
...

@property
def age(self):
"""Age of the person (birthdate-based)"""
t = date.today()
b = self.birthdate
return self._age_diff(t, b)
```

The fundamental idea is to place parts of documentation directly into a documented artefact (a variable, a function, a class, a module, a source file, etc.). The resulting documentation is as modular and evolvable as the writer creates it according to guidelines and with *Don't Repeat Yourself* (DRY) principle. It is then indeed easy to edit just a part of documentation related to one concern if the concern is separated in the source code. Another observation is that such documentation is composed of reusable parts. Linking the source file to different project results in its inclusion to a documentation of a different project, too. [22]

On top of the modularity and evolvability of such documentation, other advantages can be observed in such systems. The used style and resulting format (for instance, HTML or PDF) are picked and specified independently of the textual content. Furthermore, as it is usually a part of software development, the tooling and community around these systems are on a very good level including support in version control systems [6][22].

The already mentioned Sphinx is a tool designed for creating intelligent and easy-to-read documentation that is using reStructuredText syntax together with many customizations. Originally, it was created for the Python documentation, but it has excellent support for the documentation of software projects in other languages as well. However, it is not limited to software documentation. It is possible to find online courses, personal websites, or even theses composed using Sphinx [8][23].

III. Our Approach

Our approach to investigate and understand the problem domain of evolvable documents is to split it into *four separate key areas* and to build conceptual models of the domain in an ontologically rich language OntoUML based on them. Next, we suggest possible solutions that can be based on them and could lead to improvement of documents evolvability.

A. Key Document Viewpoints

After the brief overview of current approaches in the ICT support for documents, this section introduces various key viewpoints that are limited to electronic documents but are typical for documents in any form. Each of them is briefly described, and a possible implication in the computer science domain follows. The viewpoints are defined with respect to the semiotic ladder that introduces several steps from the social world to the physical world: pragmatics, semantics, syntactics, and empirics [24].

Pragmatics and semantics, that are related to the meaning and intentions, are covered within the first three subsections. Syntactics is related to the last subsection called Structure. Encoding the document in the physical world, as other parts of empirics, is out of the this work's scope as we are on different abstraction level than character encoding or printing.

1) Meaning: Apparently, the meaning is the key part of a document, as the purpose of the document is to store and carry a piece of information that can be retrieved in the future [25]. As the well-known triangle of reference [26] says, the meaning is encoded in symbols of some language via concepts. The common problem is that in the case of documents, the language is a natural language. Because of that, documents are hard to be understood by computers effectively in the sense of their true meaning, i.e., lacking a property nowadays called as machine-actionability as opposed to human-readability. Advanced methods in data mining and text processing disciplines try to address this [27]; however, sometimes the meaning is hard to be decoded even by human beings themselves.

Meaning, purpose, concern, and other content information may be provided as metadata of the document or its standalone part. Considering such metadata, there should be a simple, single and flexible model for the description of documents for an easy automated processing. If a meaning of a text is captured in a machine-readable way, then it is possible to extract desired information, compare the meaning of different documents, find logical dependencies, and many others with an automated processing. [28][29]

The most basic form of captured meaning are *triplets* that consist of subject, predicate, and object [29]. Such an assertion is very simple but powerful. For specific languages, it is possible to derive them more easily than from the others (e.g., English with its stable sentence structure vs. Slavic languages); text mining may also be used for derivation [27]. The assertions can naturally have relations between themselves and form a swarm of assertions, which is helpful for comparing different sources of information. The information storing based on triplets is typical especially for life sciences. Of course, encoding a natural sentence into several connected assertions is excellent for machine-actionability but not suitable for a regular reader that is used to enjoy the beauty of fluffy sentences.

2) Concerns: Writing a document happens with a concern in mind, and typically there are multiple concerns across a document. We can understand a concern in a document as a principle that binds sentences in a paragraph, paragraphs in a section, and sections in a document together. The whole document then speaks about the highest-level concern that is then split into parts recursively, until we reach some atomic level such as paragraphs containing a set of statements. Lower-level concerns can act as a separator of document modules, and higher level concerns are then composed by multiple submodules. It indicates that splitting the concerns further is not intended by the author.

For example, considering a manual for a product, the top-level concern is about the product in general with sub-concerns installation, usage, license, and warranty. The usage can be then again split into concerns related to usage of specific parts of the product. On the other hand, the warranty might not have any further sub-concerns.

3) Variants: Apart from the primary concerns in a document, there are also cross-cutting concerns that are not related to meaning and information inside a document, but rather to its usage. Such cross-cutting concerns are an intended audience, specific ways to describe the concern in respect to the essence of the document, a language, a form of document (slides, handout, book, etc.), and so on. Those represent *variants* of a single document. They are a source of possible combinatorial effects and also highly affect the content of the document.

For instance, teaching a course requires a textbook and lecture slides which are, of course, very closely related. When you do some update in the textbook, you need to update the affected slides. Now, imagine teaching the course in two languages with some classes for seniors and some for juniors. So, you have 8 different documents and adding one more language would lead to another 4. Apparently, it is becoming

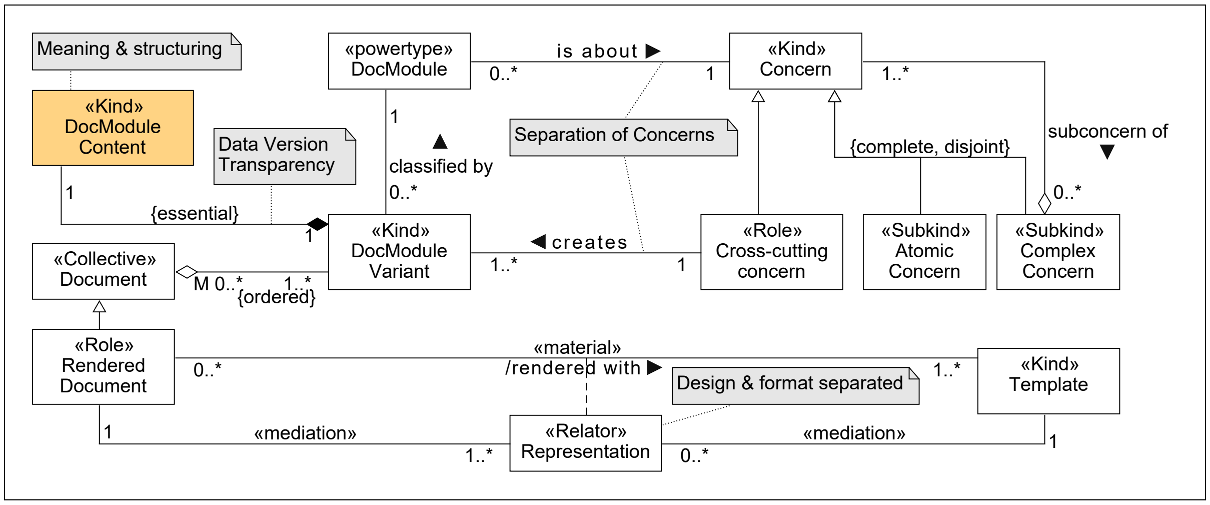

Figure 1. Conceptualization of concern-based document modularization in the OntoUML ontologically well-founded language

hard to manage these separate documents correctly. This is the core challenge, where combinatorial effect-free documents should help. Ideally, we want to work on sum of reusable variants ($2 + 2 + 3 = 7$) but get a product of possible final documents ($2 \cdot 2 \cdot 3 = 12$). Although that is hard to manage for documents, it is crucial to use variants for the lowest level concerns to maximize the advantage.

4) Structure: A structure of a document is essentially a hierarchy of the document composition: chapters, sections, subsections in various levels, paragraphs, and parts of paragraphs. Then there are also other *block elements*, such as lists, tables, figures, code examples, equations and similar. Next, we distinguish so-called *inline elements*, which are parts of text inside a block to capture the different meaning of words (e.g., a link, important, math, a quote, a superscript, etc.) or to provide additional information, for example, a reference or a footnote. Notice that we do not state anything about the style here.

The naming of document parts or structural elements can be different based on the template. For example, in a template for presentation, we can expect to have a group of slides, slide, slide section, and bullet instead of sections and paragraphs. It can be totally custom and innovative but always with the same purpose to encode the text within a logical structure that reflects the composition of concerns that are carried in the content, not always necessarily by natural language.

The flexibility of a document structure is an enabler of evolvability. Aligned with the notion of modules in programming, every modular unit should be loosely coupled with remaining parts and allowed to be moved to a different place even in a different document. A heading level represents a typical problem: there is a level of the unit involved, and it gets more complicated with cross-references. Cross-reference to a different internal document part can be easily switched to external reference pointing to a separated document part or even its labelled encapsulated content. It goes even deeper when we consider that its position in a document may form a list of prerequisites that the reader should know beforehand.

Finally, we would expect a possibility to define a new custom element, based on those already specified in the structure, to increase usability and flexibility. That indicates the need for multilevel modelling in the document structure. For example, a table with predefined rows and columns can be used for invoices, a link to some resources that changes to the best possible mirror server, or a special type of paragraph can indicate the higher importance of content for readers.

B. Conceptualization of Documents

Based on the previous considerations, we can now assemble the conceptual models. We use already mentioned language OntoUML which uses high-level and well-defined terms from the Unified Foundational Ontology (UFO) as stereotypes and significantly enhances semantics and expressiveness of basic Unified Modelling Language (UML). Details about the language and the ontology are fully explained in [2]. The connector of all the introduced models is the *document content*, the carrier of information. All models are connected, compatible, and describe different viewpoints introduced in the previous section. Moreover, NS patterns and modularization are well observable in the following models.

1) Concern-Based Document Modularization: Figure 1 shows the diagram of the conceptual model with the separation of concerns pattern for documents. A document is a modular structure composed of module variants that encapsulated the content. Module variants are instances of document module, thus we use the *powertype* stereotype [30]. Concerns as the drivers of modularization are naturally binding elements of documents to groups. A concern can be composed of subconcerns and that makes it complex concern, otherwise, it is atomic.

Cross-cutting concerns are then the special case of general concerns in case they produce variants of document modules, i.e., for one or more module variants the concern is in a role of the cross-cutting concern. The model allows a case when module variant is about a concern that is also the cross-cutting concern for the very same module variant.

Documents can be rendered using many templates, while the content is still the same. That separates a used style and typography from the actual content. We call the document rendered if it is represented by using a certain template.

For example, in a manual for a software product, there are the following concerns: installation, usage, and warranties. Some of those have sub-concerns, which creates submodules, e.g., installation for various platforms. A cross-cutting concern, in this case, can be the language. Variants of "installation" are formed by using different natural languages. The manual is an ordered collection of various variants. Thus it is possible to have a multi-language manual, but also language-specific manual, or just installation manual in English and then reuse these module variants easily. Finally, the manual can be then rendered with a template for printing, website, annotated XML, eBook, and so on.

Language is a typical cross-cutting concern in documents, but it can also be a case of general concern for creating modules. Consider a document about some ancient language. Probably some top-level module will be about the language concern with sub-concerns related to different parts of the language. Such document can be published in many languages as a cross-cutting concern as well.

2) Document Content Structuring: The task of document content structuring has been addressed many times through syntax for composing documents and systems like the already-mentioned Pillar, LaTeX, or Sphinx. For our purpose, the conceptualization is designed on a higher abstraction level, as shown in Figure 2. A content of the module is composed of document elements that can be either block or inline. Block element contain other block elements and/or plain content. Plain content can be a decorated part by some inline elements, for example, marked as important or quoted.

Those types of elements are similar to document modules powertypes [30] in the conceptualization, and their instances are particular usages of them. For example, the most common instance of a block element is a paragraph, and an instance of a paragraph is a particular paragraph containing a particular text, which is covered by the atomic content kind that is not further subdivided in our model. It works similarly for figures, pieces of data, file imports, and so on.

Element type instances can be the well-known unordered or ordered lists, tables, definition lists, links, forms, cross-references, figures, quotes, external references, and others. On top of that, using powertypes allows defining new structural elements with different semantics, e.g., an important paragraph, specific table combined with a form, or external file. Metadata for each module content and document element can be provided. Content may be maintained as revisions that allow keeping track of changes.

3) Meaning in Nanopublications: The way a meaning is encoded within a document module content is shown in Figure 3. A content is formed by natural sentences, which are essential for the content as a whole, for a writer to express thoughts, and for a reader to perceive them. In a sentence, there can be one or more encoded and usually tightly connected assertions, which are triplets in a simplified view: subject, predicate, and object. It is possible to form multiple assertions with the same meaning by using synonyms, and by switching subject with the object while using predicate for opposite direction (e.g., *Peter likes sushi* and *sushi is liked by Peter*).

Knowlet, or so-called nanopublication, is such an assertion with additional information and provenance as characterizations. Nanopublications are widely used within semantic webs

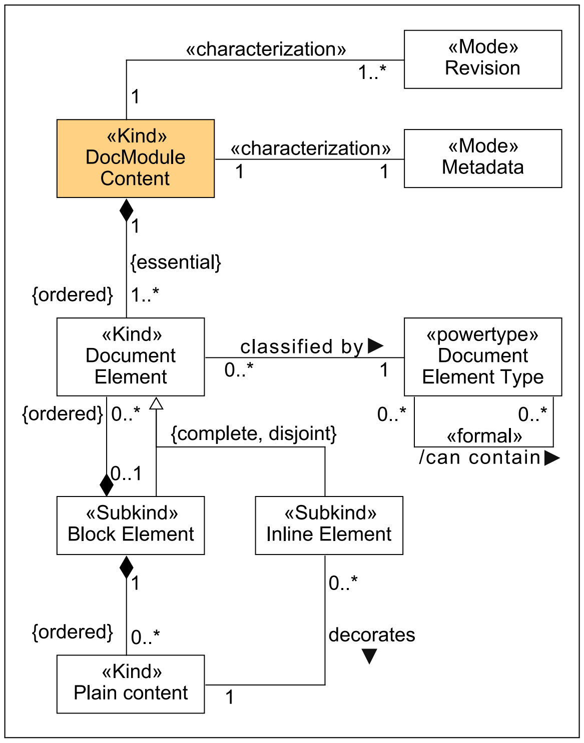

Figure 2. Conceptualization of structuring document module content

and Resource Description Framework (RDF) in general, as described in [27] and [31]. Each instance of a word should be uniquely identifiable, in semantic web this problem is solved by the use of Uniform Resource Identifier (URI). For example, even with a simple assertion like *cat is white*, we need to know which cat the assertion is about, or if it is about all cats. The context is crucial for assertions, but it is hard to be adequately captured [27].

This expression of meaning could allow machines to read and understand the content in a more efficient way rather than it is possible with text mining. Moreover, a semantic search, comparison, or reasoning can be built in a more straight-forward way. It could lead to easier work with the documents, their parts and changes, and significant resource savings. With a definition of opposite words, a contradiction in sentences, for example, *cat is dead* and *cat is alive* with same-URI cats, can be indicated.

IV. CASE STUDY: EVOLVABLE ONTOUML DOCUMENTATION

In this part, we are going to demonstrate the previously described ideas from our conceptualization on a very specific sort of document – the OntoUML documentation. We chose this topic for several reasons. First, it is needed in the OntoUML community, since the information is spread among various papers and theses. The documentation of OntoUML has many concerns including cross-cutting concerns, can be semi-structured, and it describes some solid assertions that must be valid and consistent across the documentation. Also, we have used OntoUML for our conceptualization, it might help the reader to understand how OntoUML works.

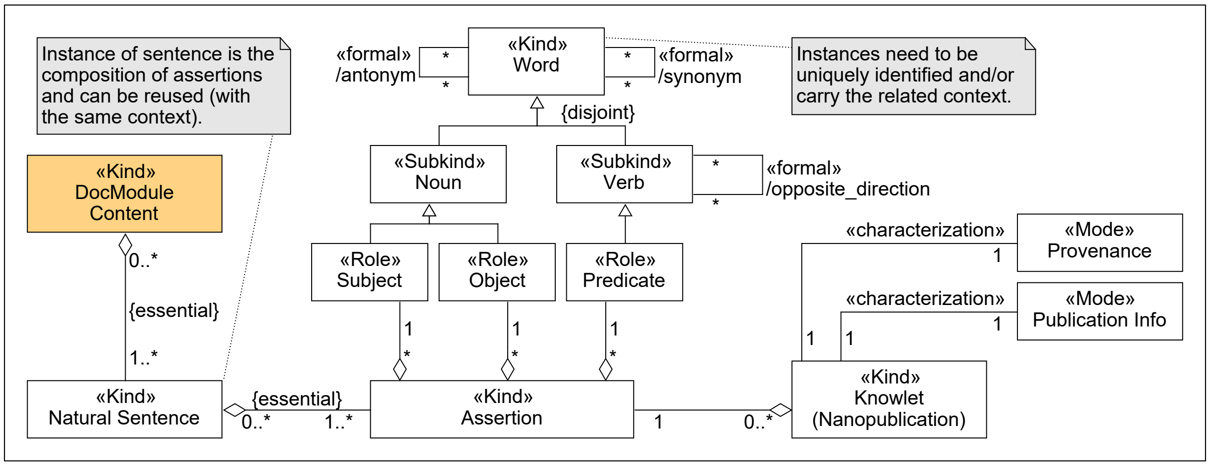

Figure 3. Conceptualization of meaning encoded in nanopublications

A. *Concerns of OntoUML*

The OntoUML documentation should cover many different top-level concerns such as introduction with general information about OntoUML, core concepts, class stereotypes, relationship stereotypes, and complex examples. They represent the very first level modularization of the answer to “What is the OntoUML documentation about?”. The selection and order are mainly given by the OntoUML properties, but reader experience should be considered as well.

1) Concerns Composition: Those top-level concerns are (in accordance with the conceptualization) further composed of sub-concerns in multiple levels. For example, class stereotypes concern consists of Kind, Subkind, Role, Phase, and other stereotypes as leaves that can be used when defining an entity type, but there is a hierarchy separating sortals and non-sortals and then there is grouping by rigidity. It can be expected that concerns can be added or removed as well as change its position in the hierarchy in the future.

2) Cross-Cutting Concerns: For each of the stereotypes we have concerns that “cut through” these sub-concerns:

- textual description,
- metamodel fragment (structured information),
- constraints,
- frequently asked questions (FAQ),
- assertions (rules and logical laws),
- examples,
- related patterns and anti-patterns.

Those special cross-cutting concerns apply also to each of the relationship stereotypes. Also, some are applicable to other top-level concerns, such as examples of core concepts or FAQ about OntoUML in general. At some positions a cross-cutting concern can be mandatory and at other positions the same cross-cutting concern can be optional. Again, cross-cutting concerns may be added, removed, or changed in the future.

B. *Architecture of the Prototype*

The most challenging part of the prototype is to devise the prototype’s architecture. It must allow simple usage with standard tools (to avoid reinventing the wheel), capture concern-based modularization described in the conceptualization (Section III-B1). The document content structuring conceptualization (Section III-B2) affects the choice of document composer that allows separation of graphical design and markup functionality extensions, as well as composing document from various parts together. Such extensibility then easily enables implementation of the meaning encoding captured with nanopublications from the final one of our conceptual models (Section III-B3). The architecture is depicted in Figure 4 as further described.

1) Directory Structure and Files: The core principle, separation of concerns, is realized in a very straight-forward way – the directory structure and files. Folders represent concerns as they are also an example of the composite pattern, they allow sub-concerns as subfolders and form internal nodes of the document tree. As leaves, there are files of various types and purposes representing the lowest-level cross-cutting concerns, i.e., *DocModule Variant*. They encapsulate submodular structuring and carry the specific *DocModule Content* as its essential part.

Aside from these files with content such as plain text, figures, tables, datasets, laws, or code fragments, two special types of files need to be present to describe the structure and metadata. First is a classical index file that acts just as the table of contents for the single atomic concern. The second type – called descriptor – contains the definition of *DocModule* (modelled as a powertype in the conceptualization).

2) DocModule Descriptors: The DocModule descriptor is a definition with basic information about the module. It specifies the content using easy-to-read and to process Ain’t Markup Language (YAML). The main information included in a descriptor is:

- name and a short description of the concern,
- list of cross-cutting concerns involved and their role for the current concern or sub-concerns,
- list of sub-concerns and their restrictions (multilevel specification).

An important thing to emphasize is that those lists containing both types of related concerns are ordered. The order is important for the reader’s understanding, for example, a description should come before an example in the most cases. For the sub-concerns, a logical order is preferred: from basic to more advanced concepts that also contain more references to the previously described. This part of descriptors also enables simple reordering.

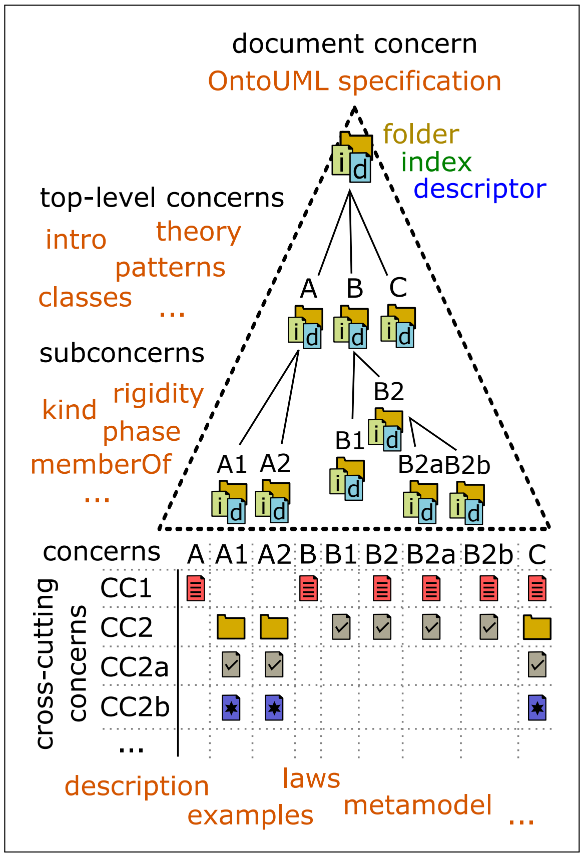

Figure 4. OntoUML evolvable documentation architecture

3) Text Parts: As described already in Section II, there are many markup languages and – in terms of evolvability – simple and extensible formats with plain-text human readability that are a suitable option. Thus, all textual modules will be encoded in reStructuredText. It allows all basic markup, figures, references, various block types, and tables. Using Sphinx tool, it can be easily extended with custom document elements (i.e., define *Document Element Type*) as shown in Figure 2 of the conceptualization [8].

4) Metamodel fragments: The main part of the OntoUML documentation is the definition of the modelling language metamodel. For this purpose, YAML files will be used to specify the metamodel fragments based on atomic concerns such as single stereotypes. For these YAML files, appropriate schema provides a simple way of validation. The core idea is to provide multilevel modelling – class stereotype schema describes what properties can be described in the class stereotype descriptor. This part of the work is highly influenced by the structure of UML profiles.

5) Assertions and Laws: In the OntoUML specification, a lot of assertions are made that need to be consistent across whole documentation. A simple markup extension should allow defining triples in the text parts of the document. It should clearly define what assertions there are as a conclusion from a paragraph or a figure. It allows semantic queries over the documentation with widely used tools, but also simplifies understanding for the reader.

On top of that, important OntoUML properties are specified as modal logic formulas. Similarly to assertions, those formulas need to be presented as rendered mathematical expressions to a reader but also kept in a machine-readable format for reasoning and contradictions revealing. An example of such reasoning is a validation of OntoUML models that uses a specific version of metamodel from the documentation. In the OntoUML 2.0 paper [32], the TPTP Logic Specification Format is used to encode the formulas and it can be used for the evolvable documentation, as well.

6) Figures and Model Examples: Since model examples are essential for the OntoUML documentation, it needs to be done in an evolvable way, as well. Problems appear when an exported diagram is used as a figure in the document, as it gets separated from the model in an editable format. Instead of exported graphics, there needs to be a way how to connect the model into the document through its source file (e.g., XML) and to generate a figure when the document is composed. A new custom extension should solve this problem.

C. Tooling

After the design of the document encoding into concern-based modules of various types and supporting metafiles, the next step is to propose a workflow of how to build and work with the document using appropriate tools. Many useful services and tools that were already mentioned in Section II are going to cover requirements for this use case. Other custom tools have to be designed and implemented because of the uniqueness and novelty of the solution (for instance, exporting an encoded metamodel as a UML profile). Figure 5 shows the realization with specific formats and tooling.

1) Documentation Weaver: Using reStructuredText leads to using Sphinx as the tool for building the documentation. It acts similarly to the weaver in AOP for our case study. It takes all linked *DocModule Variant*, forms internally a merged *Document* that is then rendered using selected *Template* and output format as described in Figure 1. Predefined or custom-made *Template* can be used, including possibility of creating a document *Representation* in form of a classical document, website, or presentation [23].

2) Sphinx Extensions: Sphinx easily allows to develop custom extensions in the Python programming language and since it is a widely-used tool, many useful extensions already exist. First, `autosectionlabel` provides a simple way how to reference concerns simply by using headings. More interesting is `ifconfig` that allows incorporating conditional blocks in the documentation, for example, some parts of textual description visible just when building presentations from it. This extension will also be used to easily exclude concerns from the document composition. Other community extensions, such as builders for different formats including docx, will be used, as well [23].

3) Concern Query Tools: In the documentation constructed as described above, there is a precise definition of concerns, cross-cutting concerns, and their relations. Thanks to that, another support tool can be developed to inform the writer

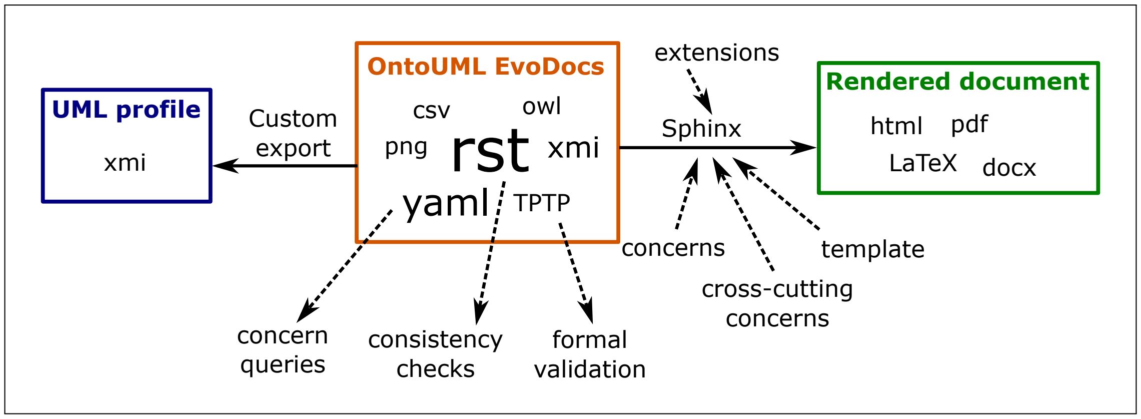

Figure 5. Selected formats and tools in overall architecture

about possible change propagation through referencing. For example, if the metamodel specification of the Role stereotype is changed, then the writer should be notified to revisit textual description of Role, examples, and possibly other stereotypes tightly related to the Role. There are important questions then, for example, how much are they related, can it be simply measured by the number of references or denoted with special annotation. Other similar queries are straightforward to implement with this architecture.

4) Custom Exports: OntoUML is a UML profile and all the needed information is part of the documentation. One of the custom tools will enable to build UML profile specification in XML Metadata Interchange (XMI) standard from the YAML descriptors. Basically, the tool is a simple translator between two formats. In the future, new exports may be easily made from existing specification or with enriching it by additional cross-cutting concern.

D. Sharing and Collaboration

Everything needed is encoded in multiple simple text formats structured nicely with directories and subdirectories according to document outline. On top of that, there are some metadata about the document to enable easier composing and understanding. Thanks to all that, is it possible and also beneficial to use Git for version control and collaboration on the document.

1) Editing the Documentation: Thanks to the selected formats, any text editor can be used to edit the documentation. Support for auto-completion, syntax highlighting, on-the-fly preview, and other nice-to-have features is very good due to the selected formats that are standard and widely-used. For reStructuredText, there are also WYSIWYG editors, even in an online in-browser version.

2) Branches and Versions: OntoUML specification is expected to be developed in various branches as new proposals based on scientific researches or practical use cases emerge. Then there should be also the main branch with the official specification. Separate branches can be developed by anyone and mechanism for incorporating changes in the main branch should be possible with allowing discussion and reviews of experts. In all branches, version tags are needed to enable referencing for models (e.g., this model is designed using OntoUML v1.0.5). All of these features are covered by Git and GitHub as already discussed in Section II.

3) External Services: Choosing GitHub as a hosting for the Git repository with the documentation enables the use of a lot of integrated external services. Aside from Travis CI for automatic building the documentation and sending it to a web server, for Sphinx documentation, there is readthedocs.org service that directly publishes the documentation that is currently in the repository. Another example of a very useful integration for OntoUML specification case is Zenodo that provides Digital Object Identifier (DOI) assignment to the repository content. With the GitHub API, it is possible to build custom integrations in the future, when they are needed.

E. Prototype Implementation and Evaluation

According to the previously described conceptualization and the proposed solution, we implemented the prototype of the OntoUML evolvable documentation and published it in the repository github.com/OntoUML/OntoUML with automatic tests checking consistency and on-change documentation publishing on ontouml.readthedocs.org.

1) Implementing the Solution: After setting up the repository with Sphinx boilerplate, as the initial documentation, the previous OntoUML Wiki from the community portal was translated from HTML into reStructuredText and split by the mentioned concerns and sub-concerns. For this translation, Pandoc tool was very useful although it had to be completed by person mainly because of relative links and particular reStructuedText environments giving the text more semantics than previous HTML encoding. Concerns and sub-concerns are directly linked via a table of contents or include features of Sphinx. Consistency and ability to build the complete documentation was set up through Travis CI and automatic deployment of the documentation as a website through ReadTheDocs (both free thanks to the open-source license of the project).

2) Previous Solutions and Comparison: Previous to this solution, we ran the community portal ontouml.org that was intended to be the central point of OntoUML knowledge (that is spread across multiple papers and websites). At our faculty, students used the portal to study OntoUML and reported possible mistakes by email to teachers. These reportings needed a discussion and duplicate reports were regularly occurring. Although over a hundred of users were registered in the portal, almost no one preferred to use the forum for discussion of problems nor reporting them. After two months of new OntoUML documentation, students are communicating instensively by creating and maintaining issues on GitHub and some even directly propose changes via *pull requests*.

All changes are transparently visible and are possible to discuss widely. Thanks to the separation into smaller modules by concerns and the use of suitable file formats, changes are easy-to-do directly in the GitHub editor within a web browser. Tracking changes and comparison of OntoUML documentation versions and branches is now possible. All changes including those proposed by externals are automatically tested for consistency and can be reviewed by an expert before merging to the official branch. Those advantages resulted in dropping original Wiki in the portal and linking the evolvable documentation instead.

3) Community Adoption: As described, the new documentation brought advantages in its evolvability and ability to validate and also the adoption by the OntoUML community is positive. Since the prototype is still under development and as such we have not notified broader community yet, it is a positive sign that we have more than three times more visitors in our new documentation system over the same time span according to Google Analytics; of course there may be other aspects involved.

4) Future Development Plans: As it is a prototype, there is still a lot to implement and enhance. In the near future, more cross-cutting concerns, their validations, and additional tools will be added to the documentation making it more complex but still easily evolvable. Examples are: temporal logic description and its validation, generating OntoUML hierarchy from the specification of stereotypes, and improved automatic checks of documentation consistency. The community feedback provided is also about the form as well: We will improve, for example, rendering of stereotype overviews and implement customized documentation templates for presentations and PDF export. Such plans would be impossible or inadequately hard to do with the previous Wiki-based technology in the portal.

F. Case Study Summary

The solution described in this case study is an early prototype implementing our initial conceptualization with a focus on the evolvability patterns and principles. Thanks to that, the contribution in form of simple but smart evolvable documentation is promising and can be enhanced or used for other domains in the future.

1) Evolvability of OntoUML documentation: The introduced design and selected tools bring advantages in terms of evolvability. Adding new or editing existing concerns and cross-cutting concerns (for example, new class stereotype or a new aspect of all relationship stereotypes) to the existing OntoUML specification is easy and will not cause problems via combinatorial effects.

2) Usability: Proposing new changes or variants of the OntoUML is using well-known Git (and GitHub) workflows with branches and pull requests including community discussion, authorized peer reviews, incorporating suggested improvements, and automatic checks. Encoding of the documentation is suitable for human readers and it is machine-actionable without any vendor locking to specific text editors or text processing tools. On the other hand, the support of selected formats in form of libraries, parsers, or editors is very good thanks to their global popularity [6][8].

3) Future Work: This case study describes the overall evolvable design, encoding schemas, and basic implementation of the needed tools and the sketched further work will follow. First, the OntoUML community including the authors must be involved to incorporate their knowledge and use it for designing future version and branches of the language specification. For this, a lot of communication and setting up contribution guidelines is necessary. Thanks to inviting other contributors, new additional ideas for necessary extensions will likely emerge.

V. Next Steps Towards Evolvable Documents

The final part of this paper is about the next steps that are suggested to be done in the near future as a sequel to the introduced conceptualization. Of course, the domain of documents is changing rapidly and so is the computer science that affects it significantly. Therefore, there is not just a single possible way how to achieve evolvability in documents and other options can be explored and evaluated. The described steps seem to us very promising based on an extensive review and our own experience.

1) A Prototype of Evolvable Documents System: Designing and developing a prototype of an extensible DMS for evolvable documents based on ideas in this paper would be a suitable next step. The result should be generally usable in any domain. The prototype would serve to find proof(s) of concept and to uncover new challenges.

The process of prototype development would be based on the provided conceptualization and it could explore missing, incorrect or unnecessary concepts using standard well-known design science method (Figure 6). It is desirable that the system itself is evolvable and developed according to the Normalized Systems theory. A simple user interface is also important for daily usage.

The case study as an example of a domain-specific application could be used as an initial step for gathering generic needs for domain specific extensions. On the other hand, it should simplify the work with an evolvable document for regular users who have no expertise in programming, command line, and various markup languages when compared to the case study.

2) A Methodology for Evolvable Writing: During the research cycles of the prototype, some form of generic guidelines for creating evolvable documents may emerge. However, the possibility of writing evolvable documents is highly affected by selected tools and formats. It is desirable to strive for a modular solution based on existing open standards and tools such as the mentioned Git, Pandoc, LaTeX, Markdown, XML, GitHub, and Pillar. The presented case study is an example of such an implementation for a specific domain.

VI. Conclusion

In this paper, we present our extended approach to evolvable documents based on the principles of Normalized Systems theory but, compared to the related work, our approach is applicable for any domain thanks to avoiding any domain-specific aspects. The presented conceptualization is the basis

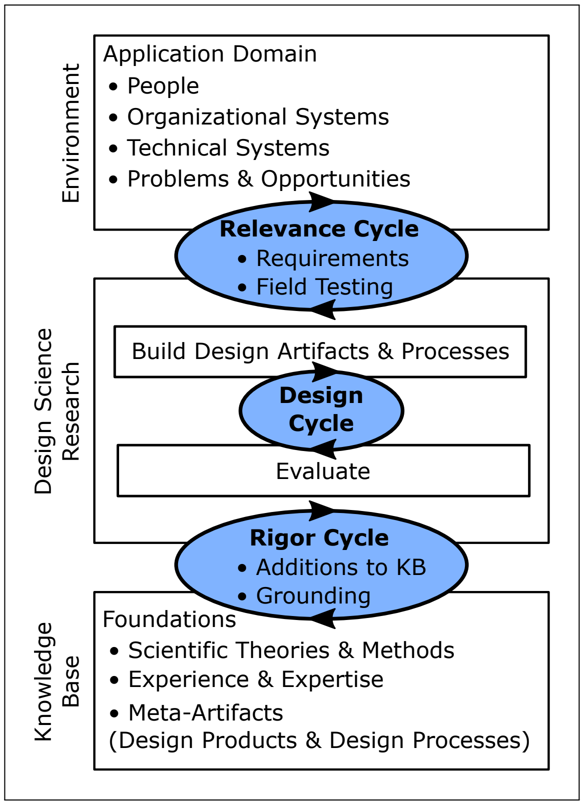

Figure 6. Hevner's design science research cycles [33]

of this generality. By incorporating modularization based on the semiotic ladder and the NS concepts together with the ontology-driven conceptual modelling language OntoUML, we uncovered different aspects and challenges in the documents domain. The described tightly-related conceptual models demonstrate the power of modularization and they can become a foundation for further discussion and building of a methodology or a system prototype using the model-driven development (MDD) methods. As we have shown in the case study, the ideas from the conceptualization together with neatly selected tooling can be used to devise a simple but smart solution for domain-specific evolvable documents. Advantages of the proposed solution over classical monolithic documents are self-evident and hopefully will be used in the OntoUML community for building the language specification. Research topic of applying Normalized Systems theory in the documents domain is very broad and our contribution is one of the first steps towards achieving evolvable documents.

Acknowledgments

This research was supported by the CTU grant No. SGS17/211/OHK3/3T/18. The included case study was supported as a project in the field of pedagogical activities (RPP) at the Faculty of Information Technology, CTU in Prague, and was partially done during Normalized Systems Summer School '18 organized by the University of Antwerp. This work also contributes to the CTU's ELIXIR CZ Service provision plan.

References

[1] M. Suchánek and R. Pergl, "Evolvable documents – an initial conceptualization," in *Proceedings of the Tenth International Conference on Pervasive Patterns and Applications (PATTERNS)*. IARIA, 2018, pp. 39–45.

[2] G. Guizzardi, *Ontological foundations for structural conceptual models*. CTIT, Centre for Telematics and Information Technology, 2005.

[3] H. Mannaert, J. Verelst, and P. De Bruyn, *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design*. Kermt (Belgium): Koppa, 2016.

[4] G. Oorts, H. Mannaert, P. De Bruyn, and I. Franquet, "On the evolvable and traceable design of (under) graduate education programs," in *Enterprise Engineering Working Conference*. Springer, 2016, pp. 86–100.

[5] G. Oorts, H. Mannaert, and I. Franquet, "Toward evolvable document management for study programs based on modular aggregation patterns," in *PATTERNS 2017: the Ninth International Conferences on Pervasive Patterns and Applications, February 19-23, 2017, Athens, Greece/Mannaert, Herwig [edit.]; et al.*, 2017, pp. 34–39.

[6] B. Duyshart, *The Digital Document*. Taylor & Francis, 2013.

[7] P. Lord, "Adventures in text land," *An Exercise in Irrelevance*, 2014. [Online]. Available: http://www.russet.org.uk/blog/3020

[8] D. Goodger, "reStructuredText Markup Specification (rev. 8205)," *Docutils Project Documentation Overview*, 2017. [Online]. Available: http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html

[9] T. Arloing, Y. Dubois, S. Ducasse, and D. Cassou, "Pillar: A versatile and extensible lightweight markup language," in *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*. ACM, 2016, p. 25.

[10] M. Dominici, "An overview of pandoc," *TUGboat*, vol. 35, no. 1, pp. 44–50, 2014.

[11] S. Kottwitz, *LaTeX Cookbook*. Packt Publishing, 2015.

[12] Wikipedia, *Template Engines: JavaServer Pages, WebMacro, ASP. NET, Template Engine, Web Template System, Web Template Hook Styles, Haml, Template Processor*. General Books, 2011.

[13] K. Ram, "Git can facilitate greater reproducibility and increased transparency in science," *Source code for biology and medicine*, vol. 8, no. 1, p. 7, 2013.

[14] S. Chacon and B. Straub, *Pro Git*, ser. The expert's voice. Apress, 2014.

[15] E. Westby, *Git for Teams: A User-Centered Approach to Creating Efficient Workflows in Git*. O'Reilly Media, 2015.

[16] K. Roebuck, *Document Management System (DMS): High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Lightning Source, 2011.

[17] V. Pal, *Alfresco for Administrators*. Packt Publishing Ltd, 2016.

[18] B. Leuf and W. Cunningham, *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley, 2001.

[19] A. Porter, *WIKI: Grow Your Own for Fun and Profit*. XML Press, 2013.

[20] G. Oorts, H. Mannaert, and P. De Bruyn, "Exploring design aspects of modular and evolvable document management," in *Enterprise Engineering Working Conference*. Springer, 2017, pp. 126–140.

[21] R. Filman et al., *Aspect-oriented Software Development*, 1st ed. Addison-Wesley Professional, 2004.

[22] C. Bunch, *Automated Generation of Documentation from Source Code*. University of Leeds, School of Computer Studies, 2003.

[23] G. Brandl, "Sphinx documentation, release 1.8.0+," 2018. [Online]. Available: http://sphinx-doc.org/sphinx.pdf

[24] R. K. Stamper, "Applied semiotics," in *Proceedings of the Joint ICL/University of Newcastle Seminar on the Teaching of Computer Science, Part IX: Information*, B. Randell, Ed., 9 1993, pp. 37–56.

[25] B. Frohmann, "Revisiting "what is a document?"," *Journal of Documentation*, vol. 65, no. 2, pp. 291–303, 2009.

[26] C. K. Ogden and I. A. Richards, "The meaning of meaning: A study of the influence of thought and of the science of symbolism," 1923.

[27] B. Mons, H. van Haagen, C. Chichester, J. T. den Dunnen, G. van Ommen, R. Hooft *et al.*, "The value of data," *Nature genetics*, vol. 43, no. 4, pp. 281–283, 2011.

[28] E. Duval, W. Hodgins, S. Sutton, and S. L. Weibel, "Metadata principles and practicalities," *D-lib Magazine*, vol. 8, no. 4, 2002. [Online]. Available: http://www.dlib.org/dlib/april02/weibel/04weibel.html

[29] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 concepts and abstract syntax. W3C Recommendation," 2014. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/

[30] G. Guizzardi, J. P. A. Almeida, N. Guarino, and V. A. de Carvalho, "Towards an ontological analysis of powertypes," in *JOWO@IJCAI*, 2015. [Online]. Available: http://ceur-ws.org/Vol-1517/JOWO-15_FOfAI_paper_7.pdf

[31] T. Kuhn, P. E. Barbano, M. L. Nagy, and M. Krauthammer, "Broadening the scope of nanopublications," in *Extended Semantic Web Conference*. Springer, 2013, pp. 487–501.

[32] G. Guizzardi et al., "Endurant types in ontology-driven conceptual modeling: Towards ontouml 2.0," 2018. [Online]. Available: https://www.inf.ufes.br/~gguizzardi/ER2018-OntoUML.pdf

[33] A. Hevner, "A Three Cycle View of Design Science Research," *Scandinavian Journal of Information Systems*, vol. 19, no. 2, Jan. 2007.