D1.2
Design of the SKO structural model and evolution Version 2

Maintainer/Editor-in-chief	Ronald Chenu
Core authors	Fausto Giunchiglia, Ronald Chenu, Hao Xu, Aliaksandr Birukou, Enzo Maltese
Maintainers/Editors	Ronald Chenu
Reviewers	Maurizio Marchese, Nardine Osman, Judith Simon, Aliaksandr Birukou
LiquidPub research group leaders	Fabio Casati, Roberto Casati, Ralf Gerstner, Fausto Giunchiglia, Maurizio Marchese, Gloria Origgi, Alessandro Rossi, Carles Sierra, Yi-Cheng Zhang
LiquidPub project leader	Fabio Casati

Grant agreement no.	213360
Project acronym	LiquidPublication
Version	v2.0
Date	June 15, 2010
State	Solid
Distribution	Public

Disclaimer

This document is part of a research project funded by the European Community as project number 213360 acronym LiquidPublication under THEME 3: FP7-ICT-2007-C FET OPEN. The full list of participants is available at `http://project.liquidpub.org/partners-and-contributors/liquidpub-teams`.

Abstract

This deliverable is intended to provide a refined version of the design of the SKO structural model and evolution. It includes variable granularity components and relations between them (e.g. citations). It also includes considerations on evolution, search and navigation, ownership and control models. All these features are oriented towards enabling the LiquidPub project's use cases (Liquid Journals, Liquid Books and Liquid Conferences).
Keyword list: **liquid**, **knowledge**, **data**, **semantic**, **science**, **publication**, **Web 2.0**, **artifact**, **reputation**, **tagging**, **SKO**

Executive Summary

Following the format introduced in the first version of this deliverable, the deliverable D1.2v2 presents three related works joined together as a single document.

Part 1 of this deliverable represents the direct evolution to the SKO structural theory presented on the first version of this document. The most significant change to be found from last year is the transition of the SKO model from being scientific representation model (competing in certain aspects with some of the existing ones like [2] or [6]) to becoming an open knowledge bus and repository that is able to support not only several types of file types as mentioned in version1 but it would be also able to support other semantic ontologies, visualization styles and becoming a communication channel between the LiquidPub use cases and external platforms or services.

The current version identifies four layers (that subsume the previous three-leveled approach from SKOv1), each of the which is used to capture a particular aspect of the scientific artifacts they encode:

- *File Layer*: includes the data level from SKOv1 and, much like its predecessor, this is the model's link to general content (e.g. text, pictures, tables). Formats accepted for this layer include document ones (txt, doc, tex, pdf) and picture ones (jpg, png) among others.

- *Semantic Layer*: includes the knowledge and collection levels from the first version of SKO, this layer includes all metadata and relational information related the objects from the previous layer. Just like it would be unreasonable to ask for a single file format in the file layer, the semantic layer does no such requirement. As such, it aims to not only to support widely used metadata representation formats (RDF, OWL, Microformats, Dublin core) but also upcoming formats for discourse representation in scientific documents (ABCDE and SALT).

- *Serialization Layer*: created by extending the serialization structure concept introduced in SKOv1, this new layer is used for enabling and tracking aggregation and reuse of the information from the previous two layers. The ultimate objective of the serialization layer is to become the blueprint that is followed to order and choose the components from the graph-like content and knowledge repository (represented in the two previous layers), into a single linear document (much in the same way a software program's makefile compiles libraries and code into a running executable).

- *Presentation Layer*: visual and display format representing directives are introduced for the first time in this version of the SKO theory. The presentation layer's main objective is to detach visual format and display consideration from the actual content and meta-information as much as possible. This would enable content creators to generate artifacts that are able to be represented in several display formats (e.g. ACM, LNCS in the case of paper representation formats) or aimed at different display devices (e.g. computer screen, printer, mobile computer screen) without much additional work. To achieve this, the presentation layer aims to support widely used display and typesetting formats (e.g. CSS, LaTeX).

With this, the SKO four-layered model, aims to become a multi-format aggregation resource that is not only able to access and aggregate content but also meta-content from the various supported and widely available formats. This aggregation and "'unlocking of formats"' is important for allowing access to information (which is relevant to the LiquidJournal and LiquidConferences use cases) and for the creation of new artifacts (which is relevant to the LiquidBook use case). Finally, Appendix B proposes an XML notation to represent all the concepts from Part 1 of this deliverable and thus enable the creation of significant examples of the application of the SKO model structure.

Part 2 of this deliverable covers applications various additional subjects related to the SKO structural model from Part 1. The following are the main highlights:

- *The State Dimension*: a direct evolution of the SKOv1 work that used a metaphor based on the states of matter to describe the evolution of artifacts. The current version of this state-based evolution model is still based on characterizing artifacts as belonging to the Gas, Liquid and Solid states and assigning them properties based on this. However, to accommodate the changes to the SKO structure and the separate introduction of version control, the state-based evolution model now focuses itself on capturing evolution and certification of the artifacts.
- *Version Control, Search and Navigation*: the following two sections of this part introduce version control separately from the state dimension and search/navigation considerations. These sections mainly explore the implementation and new possibilities for version control, search and navigation that are enabled by the new SKO model.
- *Ownership, Licensing and Control Models*: this section explores the possibilities for licensing, copyright and credit attribution that are supported by the SKO model and it also serves as a connection to the work from WP2 and WP4.

The sections of this Part 2 that are relative to the evolution of SKO-based artifacts are specially aimed to enable the LiquidBook use case. On the other hand, the sections of this part based on finding, accessing and other rights of these artifacts are specially relevant for the LiquidJournal and LiquidConferences use cases. In general, Part 2 focuses not only on explaining how the SKO model can (in the worst-case scenario) keep up with current strategies for commonly used artifact operations/processes. Furthermore, it additionally introduces how to extend and propose innovative alternatives to these that would ideally improve how scientific publications are tracked, disseminated, shared and reused. Of special importance is the Appendix A, which aims to present a practical example of most of the processes dealt with in this part by implementing a LiquidBook exercise through widely available resources.

Part 3 contains besides a brief recap of the typing and attribute definition work from first year, along with initial study of patterns in scientific papers that aims to find the common way in which the sub-atomic components of a paper come together to form a paper.

It is worth mentioning that during the second year, echoing the new shift of the project to a bottom-up approach (i.e. the use cases decide the concepts and services to be implemented and

the model adapts to that), the SKO model has become more general and flexible to accommodate the always evolving requirements from the use cases. The current version of the SKO model also presents a sort of "'future feature bazaar"' with different probable applications and features that the SKO model could offer to the use cases in the future. The Appendix C contains in more details how the requirements from the use cases and the reputation model have influenced the SKO model design, and also contains a section that details some of the features and applications of the SKO that still have to be picked up by any of the use cases. Nevertheless, with the use cases much better defined and a clear scope of how far their implementation wants to be taken, D1.2v3 from the third year will attempt to present a much more concise and focused description of the back end of the LiquidPub Platform.

Contents

Part 1

Scientific Knowledge Objects v2

1.1 Introduction

A Knowledge Artifact is an object created as a result of an activity which encodes knowledge, which is the understanding or awareness gained beyond data. On the other hand, Complex Artifacts are those that are composed of several, simpler artifacts that have been made into a single coherent unit. Examples of these Complex Knowledge Artifacts include scientific papers, books and journals, among others.

This document defines and details the Scientific Knowledge Objects (abbreviated as SKOs) introduced in the LiquidPub vision document [4]. These SKOs will become the unit and our particular representation of the simple and complex knowledge artifacts in the context of the LiquidPub project.

In the light of all the known problems and limitations, explored in [1], it is natural to wonder what would make an ideal scientific artifact format? One that is able to cope with the demands from creators, reviewers, publishers and consumers of scientific content while also enabling new interactions between these groups. More specifically, some of the requirements for this 'ideal' artifact format would be the following:

- *It is Composable and Complex*: allowing the definition of new artifacts from simpler ones, aggregating several types of data (e.g texts, spreadsheets, images, videos, etc) to facilitate the production of new knowledge or simply different executions or presentations of already existing knowledge.
- *It allows continuous evolution*: allows continuous creation and distribution of evolutions of the scientific artifact, while also supporting the current model of discrete releases at key points in time.
- *It facilitates collaborative work*: the previous properties like evolvability and composability, along with Internet-enabled interaction, greatly facilitate the creation of SKOs as collaborative effort between different actors.
- *It introduces improved models for certification and credit attribution*: provides easy-to-understand certification and credit attribution models with improved accuracy and that can

be adopted gradually.

- *It is easy to find and classify*: making the large volume of information easier to filter and sort, by the use of semantics along with smart search algorithms, can be used to fight off the information overload.

- *It has a reduced work overhead*: provided suitable editor environments are created, it introduces no or very little additional work for the human actors.

The SKO-based representation system is proposed as an evolution and extension of the currently used scientific artifacts. As an evolution of the three-leveled approach presented in the previous version of this work, the SKOs separate in layers the different types of information that are normally or tacitly present in current scientific artifacts. Fig. 1.1 shows a diagram of the SKO layers.

Figure 1.1: The different layers of the SKO model.

This multi-layered approach is mainly aimed at enabling and facilitating the composition, reuse and collaborative creation of scientific artifacts. Furthermore, it provides the base for related works on improving the evolution, credit attribution, and search/navigation of these artifacts. An additional dimension extending this layered organizational approach by the introduction of three states: Gas, Liquid and Solid. This metaphor to the states of physical matter, along with version control and other considerations of the model will be expanded on Part 2 of this deliverable. The validation of the approach specified in this deliverable and its compliance with the previous requirements will be mainly tested through the LiquidPub project use cases (Liquid Books, Liquid Journals and Liquid Conferences). Additionally, an XML-based scripting language for SKOs will be provided at Appendix B; these SKO scripting appendix is provided as a starting point for a pssible SKO-based content creation system (should the need to create one is ever established by the three use cases).

This part will first explain the four layers of the proposed SKO approach, starting from the file layer (Sect. 1.2) and then going through the semantic (1.3), serialization (1.4) and the presentation (1.5) layers. More information about SKOs and the LiquidPub project can be found at the LiquidPub project homepage[1].

[1] http://project.liquidpub.org/research-areas/scientific-knowledge-objects-sko

1.2 The File Layer

The file layer is the first and the most basic layer of the approach. Furthermore, it is also the approach's foundational layer and the main connection to well-established and commonly-used content and standards. This layer points to the actual content or data from the artifact. As such, its description starts with the definition of a widely used concept.

1.2.1 URL Definition

URL stands for Uniform Resource Locator and provides the location or address of a particular file. By using this address it is possible to reference, use and even edit the file that it points to. Furthermore, by using HTML, multiple files (each identified by its URL) may be joined and displayed as a single page. An example of this is shown on Fig. 1.1.

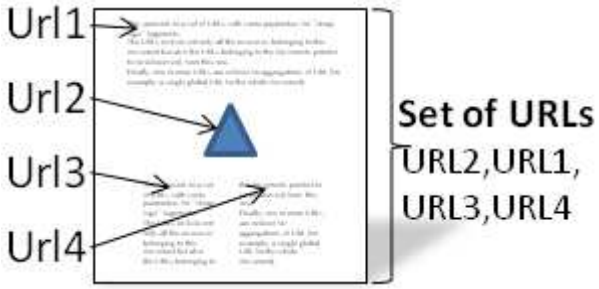

Figure 1.2: URLs involved in displaying a composite page.

Much like the citation system used in scientific publications, the URLs can also be used to reference other files or pages that, while not being included or displayed, are relevant or related to the current artifact. The URL will be used as the core component of the file layer objects, how these are defined and how they map to file-layer components will be explained in the next subsection.

1.2.2 File-layer Objects

This section introduces the two objects that are used by the approach for representing the file layer.

File Nodes

File nodes are the approach's internal representation of already existing files. As shown in Fig. 1.3, a file node is simply defined by its URL.

The file node can be considered as a "'pointer"' to the actual content that is found by using the URL. Thus, every resource that has an assigned URL, can be converted into a file node. For example, we could define a file node for the GNU GLP version 3, simply by having the URL "http://www.gnu.org/licenses/gpl-3.0.odt", which points to that file.

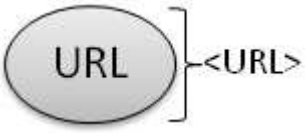

Figure 1.3: File node definition.

File Fragment Definition

While file nodes refer to the full content pointed by the URL, there may be cases (like the ones from Fig. 1.4) where there is the need of referencing or rendering only a subset of that content.

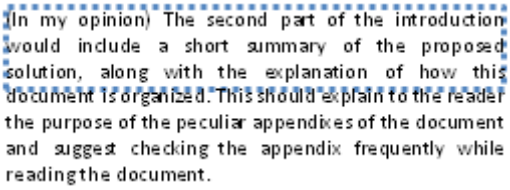

Figure 1.4: Text fragment example, the dashed lines show the selected part.

In the same way as file nodes, file fragments can be considered as "'pointers"' to existing resources. Nevertheless, the file fragments do not point to the whole resource but to a sub-part of it. HTML solves this issue by using the '#' (number) sign as the fragment delimiter. For example, in "http://en.wikipedia.org/wiki/Fragment_identifier#Examples" the word "Examples" would be used as the fragment identifier (which is normally interpreted as a scrolling directive for web browsers). Note that this sort of fragment delimitation only defines the start of the fragment. However, in order to implement a finer-grained delimitation (and in accordance with other works on the subject like [5]) we will also define the end of the fragment. Fig. 1.5 shows the fragment definition.

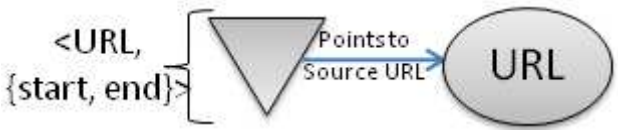

Figure 1.5: File fragment definition.

The the URL element is used to determine the object to which the fragment refers, while $\{start, end\}$ element is a non-empty set of start/end pointers within that same object. Depending of their content, some file fragments may need more than start/end pair for their definition. For example, a fragment of an image file, like the one from Fig. 1.6, requires two start/end pairs (one for length and the other for height).

Likewise, in another example, a video fragment would need three pstart/end pairs (length, height and duration) to be completely defined. The practical use of selecting parts of for files (i.e. defining fragments) will be made clear in Sec. 1.3.2 where semantic annotations are defined.

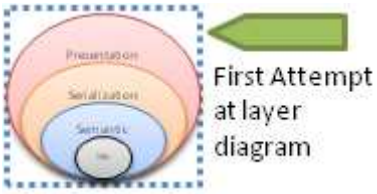

Figure 1.6: Image fragment example, the dashed lines show the selected part.

1.3 The Semantic Layer

The semantic layer is the second layer of the approach. Directly situated on top of the file layer, the semantic layer adds attributes and relations to the URLs, which are used to specify the specific context and concepts to which they refer. Metadata-based components from documents like the abstract, introduction and keywords can be represented or linked by usign this layer.

1.3.1 SURL Definition

This approach introduces the term SURL or Semantic URL, as a means to refer to the location where the semantic information of an object is being stored. As such, the SURL of a given object would point to the semantic metadata (i.e. the attributes) of that object, in the same way that the URL from that same object would point to its data or content. For the purposes of this work, SURLs will be represented as regular URLs that point to files that with the extension ".sem.xml". Future iterations of this work will deal with the extension of SURL into a full-fledged semantics addressing system.

1.3.2 Semantic-layer Objects

This section introduces the three objects that are used by the SKO model for representing the semantic layer. Fig. 1.7 exemplifies the uses of some of these objects to capture the semantic information related to a document.

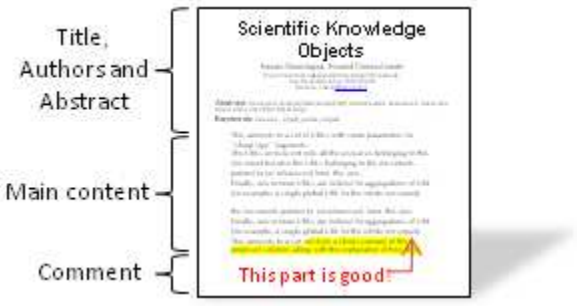

Figure 1.7: Example of semantic information within the first page of a document.

Notice that Fig. 1.7 is a visual representation of a first page of a document, with clearly delimited title, authors, main content and an added content referring to the highlighted part. All

this information can be represented as shown in Fig. 1.8.

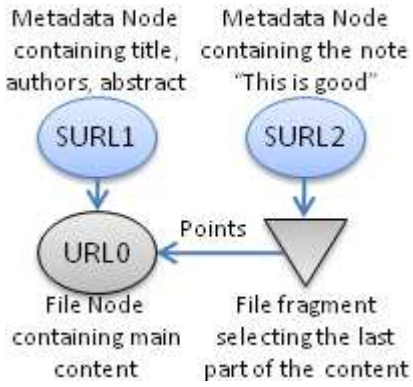

Figure 1.8: Representation of the semantic information of the first page example.

In Fig. 1.8, the main content from Fig. 1.7 is represented with a file node (URL0) and a file fragment within URL0 corresponding to the last few sentences marked in yellow at the end of the document. Furthermore; a semantic object (SURL1) is defined containing the title, authors, abstract that correspond to the file node; while another semantic object (SURL2) containing the comment "this is good" is created referring to the previously mentioned file fragment.

Semantic-layer SKOnodes

Semantic SKOnodes are the approach's main way to enrich a file node with semantics that apply specifically and exclusively to it. Semantic SKOnodes are attached directly on top file nodes from the previous layer, and their definition is given in Fig. 1.9.

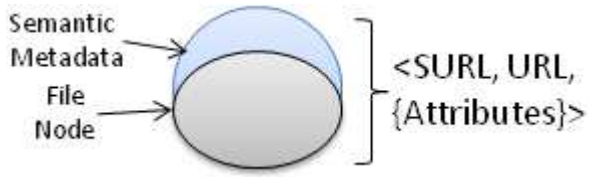

Figure 1.9: Semantic SKOnode definition.

In Fig. 1.9 $SURL$ and URL define the link between the file layer and the semantic layer. The possibly empty $\{Attributes\}$ set is, on the other hand, used to define basic semantic information that is found on most of the SKOnodes (for a list of these attributes refer to Appendix B). An example of a SKOnode, would be the result of the objects with SURL1 and URL0 from Fig. 1.8. More generally, any combination of data and metadata that refers exclusively and specifically to that data can be represented as a SKOnode.

By using semantic-layer SKOnodes it is then possible to enrich the objects from the file layer with attributes; however the the true advantages of doing so will be revealed when the last two objects from the the semantic layer are explained.

Semantic Annotations

Semantic SKOnodes are used to capture fairly-common and basic attributes for specific file nodes. However, there is also the need to capture semantics that exist between objects (e.g. object A is better than object B) or that apply to more than one object at the same time (e.g. I like objects A and B). Furthermore, the need for more unique and specialized attributes is also necessary to capture less common concepts. Semantic annotations, as defined in Fig. 1.10, are the approach's response to offer the customization beyond the definition of standard attributes from SKOnodes.

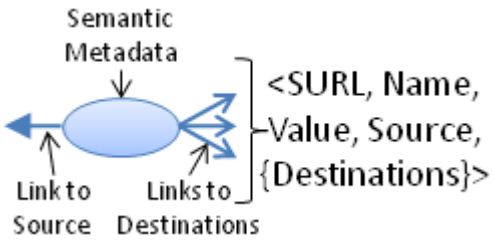

Figure 1.10: Semantic annotation definition.

The $SURL$ element from Fig. 1.10 points to the current annotation, while the $Name$ and $Value$ elements are respectively used to identify the annotation and assign it a value. The $Source$ attribute is used to store the address of the object to which the annotation mainly refers to and the possible empty $\{Destinations\}$ set specifies the address of the secondary objects that also are affected by the annotation (examples of this below). Additional rules may be attached to each specific type of semantic annotations (e.g. the "Is related" annotation must have at least one destination). However for simplicity's sake this will not be elaborated in this document but on the, more implementation oriented, D1.3v1 deliverable. Finally, also note that neither $Source$ nor $Destinations$ define any restriction about the objects they refer to. As such, they may contain URLs, SURLs (or the other address types that will be introduced in the next sections). This enables annotations to enrich with semantics almost any aspect of an object, for example:

- *Content*: to leave a comment on the content of a document, create a semantic annotation with "Name"="Comment" targeting its file node and containing the actual comment in its "Value" attribute.

- *Semantics*: to complain about having a work being unfairly related to a given author, create a semantic annotation targeting the disputed semantic relation with the "Name"="Complaint" and containing the actual complaint in text in its "Value" attribute.

- *Style*: to suggest some visual changes to a specific paper, point to the presentation-layer object (see Section 1.5) and leave a semantic annotation with the presentation suggestions.

More concrete uses and examples of semantic annotations may be found on Part 2 of this deliverable.

Semantic-layer SKOs

With semantic SKOnodes covering the individual and most common attributes and semantic annotation covering the other cases, it would seem that no other object would be needed to capture semantics. However, as shown in Fig. 1.11, there is a (quite common) case in which the semantics to capture emerge from the aggregation of objects.

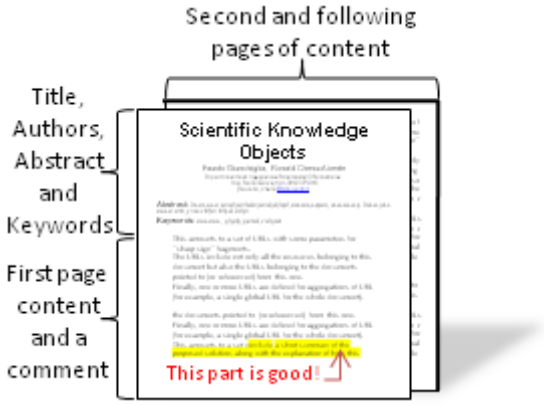

Figure 1.11: Example of semantic information within the full document.

Similarly to the previous Fig. 1.7, Fig. 1.11 contains a visual representation but, this time, of a full document. Fig. 1.12 shows a representation of this full document.

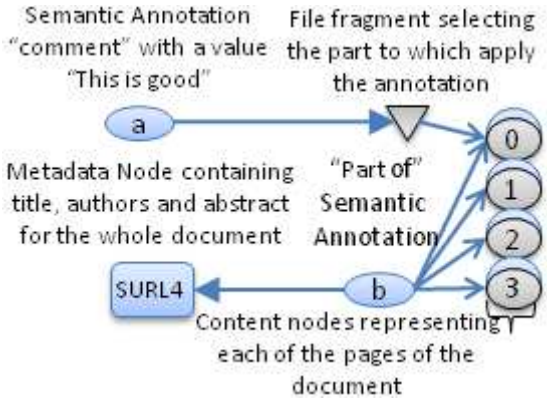

Figure 1.12: Representation of the semantic information of the full document example.

The SKOnodes: SKOnode 0 (URL0, SURL0), SKOnode 1 (URL1, SURL1), SKOnode 2 (URL2, SURL2) and SKOnode 3 (URL3, SURL3) were added in Fig. 1.12 to represent both the content and the semantics of all the pages of the document[2]. Furthermore, notice that a new semantic object (SURL4) containing the title, authors and abstract of the document was added along with a "part of" semantic annotation b joining it to the page SKOnodes. The reason for this is that the title, authors and abstract of the document, apply globally to all the pages of the document

[2]We are aware that is unrealistic to assume that people will separate their work by pages and not by sections or concepts. This assumption is only made here to help highlighting the specific concepts being explained but note that the concepts also hold for more realistic examples.

(represented by SKOnodes). It is key, to notice that these attributes do not apply individually to each of the pages (each page may have its own lower level title); they apply only to the whole aggregated from the pages. Thus, more than the sum of their parts, the semantic SKOs (defined at Fig 1.13), capture the gestaltic (i.e. that emerge from the combination of the parts) semantics of other semantic objects.

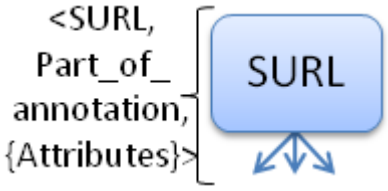

Figure 1.13: Semantic SKO definition.

The $SURL$ from Fig. 1.13 points to the current SKO and $Part\_of\_annotation$ is a SURL to the semantic annotation entity representing the "part of" relation, which connects the SKO with its immediate components (in the graphical representation of SKOs, the annotation is not drawn separately for simplicity's sake). As in the case of semantic SKOnodes, the possibly empty, $\{Attributes\}$ set is used to define basic semantic information that is found on most of the SKOs (for a list of these attributes refer to Appendix B). Consider Fig. 1.14 as an example of semantic SKOs.

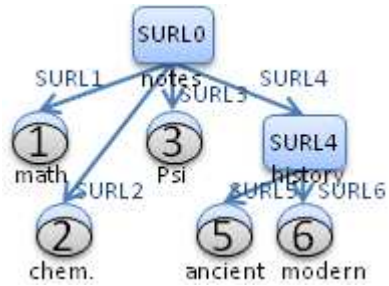

Figure 1.14: Semantic tree example.

Figure 1.14 shows a semantic tree in which two semantic SKOs and five semantic SKOnodes participate. These structures can be used to represent a categorization hierarchy as shown there or also the semantic organization of a complex document as demonstrated in Fig. 1.12.

Finally, while it is true that the aggregation of semantic elements could have been represented simply by using the "part of" semantic annotation alone, the semantic SKO was introduced to convey the idea that the aggregation itself is considered to be a new entity that emerged from the aggregation.

1.4 Serialization Layer

On Fig. 1.12 and Fig. 1.14 from the previous section we have seen how the content and semantics of a complex object and each of its parts could be represented as a semantic tree. However, if

there is the need of creating an artifact (e.g. a document) that contains all of the information in the semantic tree, it is not immediately clear how to order the data and metadata contained in it in a meaningful manner. Moreover, depending on the nature of the document that we want to create, some of the content or metadata stored in the semantic tree may not be needed (e.g. a picture used in the lecture presentation may not want to be repeated in the paper version of that same work), so we would also need a way to filter out some content and/or metadata. As the layer between the semantic layer and the presentation layer, the main objective of the Serialization Layer is to organize the content and the semantic metadata from the previous layers into a more human-friendly sequential list. This sequential list can be then converted into document (or other artifact) that effectively becomes a new whole or unit aggregated from the original parts. For example, document sections like the table of contents and bibliography (which are normally constructed semi-automatically from the full document by common word-processing tools) are introduced in this layer. By varying the information in this layer it is possible to create artifacts (e.g. documents, slides, or blogs) from the same basic semantic layer objects. These are called, different executions of the same SKO. While it is possible for executions of the same SKO to present slightly different data and knowledge, they all come from the same semantic base so they should convey approximately the same message.

1.4.1 LURL definition

Similarly to the previously defined SURL, the approach introduces the term LURL or Serialization URL as a means to refer to the location where the serialization information of an object is being stored. For the purposes of this work, LURLs will be represented as regular URLs that point to files that with the extension ".serial.xml".

1.4.2 Serialization-layer Objects

This section introduces the three objects that are used by the SKO model for representing the serialization layer. A key factor to consider is that all the serialization-layer objects are presentation-neutral. That is, they define which content or metadata will be shown. How this information will be presented (e.g. fonts, colors, what format of reference to use, etc.) will be defined when we speak about the presentation layer in the next section. In Fig. 1.15, to exemplify the use of serialization metadata, we will assume that each paragraph is represented by a different SKOnode.

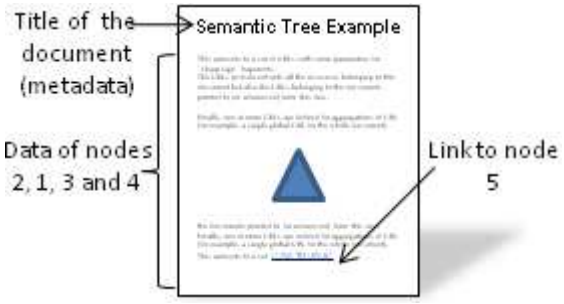

Figure 1.15: Example of the serialization information within the first page of a document.

Now, in Fig. 1.16, we see an example of the representation of the serialization information from Fig. 1.15.

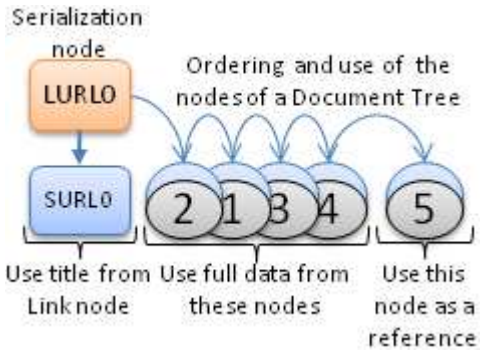

Figure 1.16: Representation of the serialization information within the first page of a document.

Fig. 1.16 shows that the documents can be represented by a SKO containing the top level metadata (title, for example), five SKOnodes (1,2,3 and 4) containing the text and images of the page and an additional SKOnode (5) that is referenced to but without actually displaying its content. Note however, that the semantic-layer SKOs and SKOnodes do not, contain neither information about what SKOnode is actually showing in the page (e.g. full content, title, references), nor the actual order in which the different SKOnodes appear. As shown in Fig. 1.11, this information is contained by a serialization layer object (LURL0).

Serialization-layer SKOnodes

Serialization SKOnodes are used to define serialization information about a specific semantic-layer SKO or SKOnode. Thus, as shown in Fig. 1.17, serialization SKOnodes are defined directly above semantic SKOs or SKOnodes from the previous layer.

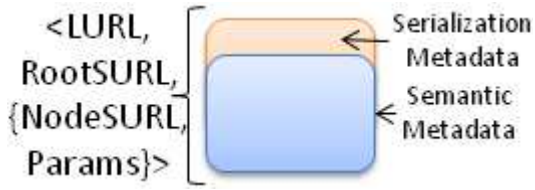

Figure 1.17: Serialization SKOnode definition.

The first two elements from Fig. 1.17 (LURL and RootLURL) are used to define the link between the semantic SKOnode or SKO from the semantic layer and the serialization SKOnode from the serialization layer. The third element is an ordered list of pairs (NodeSURL and Params), that defines which semantic objects are part of the serialization and how. The current supported values that the element Params can take are:

- *full*: includes the full content of the pointed SURL in the serialization.
- *abridged*: includes in the serialization a previously defined (via annotations, that would mark the key ideas on the text) abridged version of the pointed SURL. For example, this

command would be used to include the abstract or key ideas of the document in its serialization.

- *title*: includes in the serialization the defined (in attributes) title of the pointed SURL.
- *author*: includes in the serialization the defined (through relations) authors of the pointed SURL. For example, this command would be used to include the authors of the document in its serialization.
- *reference*: with this parameter the serialization would only include a reference of the pointed SURL.

Consider, Fig. 1.18 as a concrete example of serialization SKOnodes, where different results are obtained from the same semantic-layer object (SURL0) by changing the serialization-layer information from the serialization SKOnodes (LURL1, LURL2, LURL3).

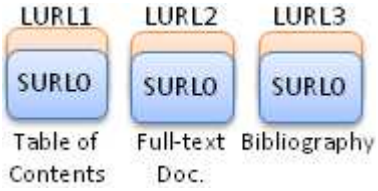

Figure 1.18: Example of different serialization SKOnodes.

Serialization-layer Fragment Definition

Previously to the serialization-layer we could define semantic trees (like the one in Fig. 1.14) to convey that multiple nodes and their aggregations were semantically related. However, it was at the serialization-layer where we specified exactly how these subparts actually came together to form a new whole.

While serialization-layer SKOnodes refer to the full content aggregation, just like in the case of file fragments, there may be cases where there is the need of referencing only a subset of that aggregation. It is for such situations that we introduce the serialization fragment structure, as defined on Fig. 1.19.

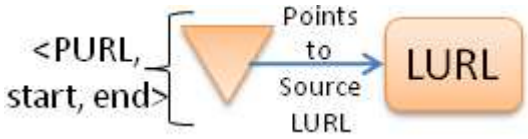

Figure 1.19: Serialization fragment definition.

There are a lot of similarities to be found between the serialization-layer fragments defined at Fig. 1.19 and the file-layer fragments defined at Fig. 1.5. These similarities stem from the fact that the objective of both fragments is to define a continuous subpart of a unitary object. In the case of the serialization fragment however, only a single start and end are defined (whereas file fragments may have many). The reason for this is that the serialization fragments operate over

elements that are always linear (as they have been serialized), so only one pair of start and end arguments is needed to specify the selection of these fragments. As a concrete example of the use of serialization fragments, suppose that back at the example from Figure 15 a concerned reviewer would like to leave an annotation that applies to the contents of node 2 and node 1. This is shown in more detail in Fig. 1.20.

Figure 1.20: Serialization fragment example.

Once selected, the reviewer is able to use the fragment to leave a comment related to the serialization of the document (for example, containing something along the lines of "these two paragraphs do not form a logical progression").

Serialization-layer SKOs

Serialization-layer SKOs perform the aggregation of other serialization-layer objects. As a motivation for this, consider the example from Fig. 1.21.

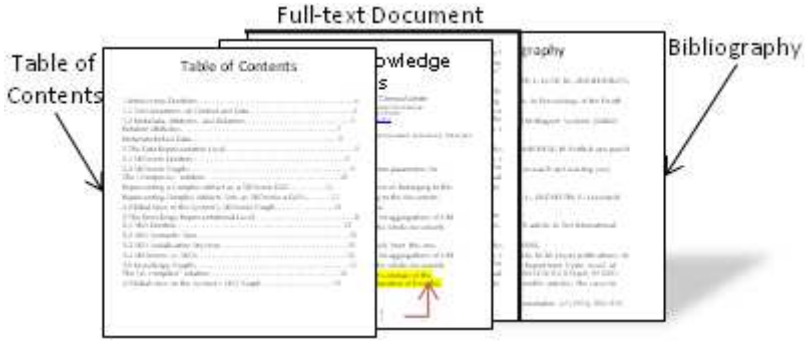

Figure 1.21: Example of aggregation of serialization SKOnodes.

Fig. 1.21 represents a full document, that has a table of contents at the beginning, then the full contents of the document and finally its bibliography. This is represented in Fig. 1.22 by using a global serialization object.

The three elements from Fig. 1.22 (table of contents, full contents and bibliography) are represented just by three serialization SKOnodes that all have the same semantic components and only differ in their serialization. Furthermore these three serialization SKOnodes are aggregated into a single serialization structure with a single LURL. This LURL will represent the whole document and its equivalent everything in the visual representation of Fig. 1.21.

The structure used in the previous object is a serialization SKO. Note that, while serialization SKOnodes are used to compose parts into new wholes (e.g. sections into a paper), serialization

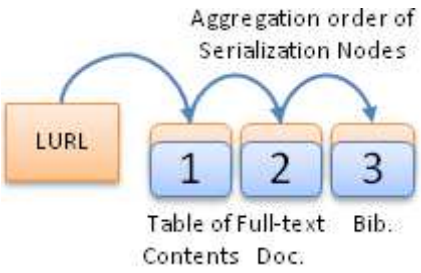

Figure 1.22: Representation of the aggregation of serialization SKOnodes.

SKOs are used to compose wholes into new wholes (e.g. papers into journals). Serialization SKOs are defined in Fig. 1.23.

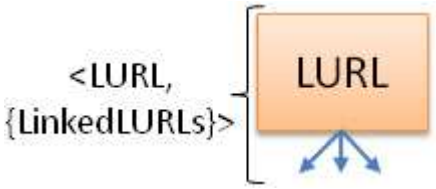

Figure 1.23: Serialization-layer SKO definition.

The $LURL$ element from Fig. 1.23 defines the address for the aggregation of serialization objects, which defined by the $\{LinkedLURLs\}$ element. The $\{LinkedLURLs\}$ contains an ordered list of the LURLs of the aggregated serialization objects mentioned before. Note that unlike serialization SKOnodes, the serialization SKO limits itself to define which serialization objects to aggregate and does not define parameters to alter the presented content. The definition and customization of the serialization is then left to each individual SKOnode in its entirety. Consider Fig. 1.24 as an example of semantic SKOs.

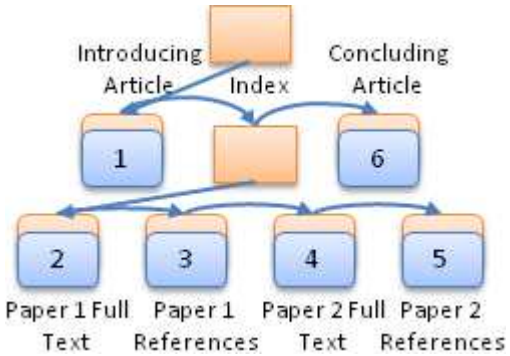

Figure 1.24: Serialization tree example.

The serialization tree shown in Fig. 1.24 is formed from the aggregation of two papers and two articles. Following the arrows we see that the order in which such document will first show the 'Introducing Article', then the full text of 'Paper 1' followed by its references, then full text of 'Paper 2' followed by its references and finally the 'Concluding Article'.

1.5 Presentation Layer

This layer introduces presentation oriented directives that enable the rendering of the previous layer's output into several presentation styles and formats. In particular, it deals with two main properties:

- *Presentation styles*: refers to colors, font types, etc. used for each type of text. For example, the headers of each section will be displayed in Times new Roman size 18, color blue. Choosing to display text in single or double column also belongs to the Presentation Layer.

- *Formats*: refers to the final output format in which all the information from the structures can be assembled into (e.g. pdf, doc, etc.). Furthermore, this layer also includes improvements in the visualization according to the display target (e.g. computer screen, mobile phone, web, etc.)

By varying the information in this layer it is possible to create different styles of the same artifact (e.g. single column, double column, text to speech, etc.) from the same basic serialization layer objects. These are called, different presentations of the same SKO. All presentations of the same SKO always contain exactly the same data and knowledge. Finally something worth noting is that, while presentation information is normally defined on top of the serialization structure (e.g. each node that represents a chapter should be rendered with the font Arial).Note that presentation information may also refer to information from other layers (e.g. render in italics all text marked as "important" at the semantic layer).

1.5.1 PURL Definition

Similarly to the previous layers, the term PURL or Presentation URL is introduced as a means to refer to the location where the presentation information of an object is being stored. For the purposes of this work, PURLs will be represented as regular URLs that point to files that with the extension ".pres.xml" (where the serialization information is contained).

1.5.2 Presentation-layer Objects

This section covers the main objects that are used by the SKO model for representing presentation information Fig. 1.25 shows, as an example, two very different presentations or styles for the same document.

Note how in the second style from 1.25 even the comment is presented differently and that the author wanted the word "objects" from the title to be printed in red (unlike the rest of the document). Fig. 1.26 shows the representation of this style's presentation information.

In Fig. 1.26, a serialization SKO (LURL0) determines the information to be displayed, while a presentation object (PURL1) contains information about the style used to present the document. Furthermore, a serialization fragment is selected to have a particular style (PURL2) that overwrites the default style defined and results in the rendering of the word "objects" in the color red.

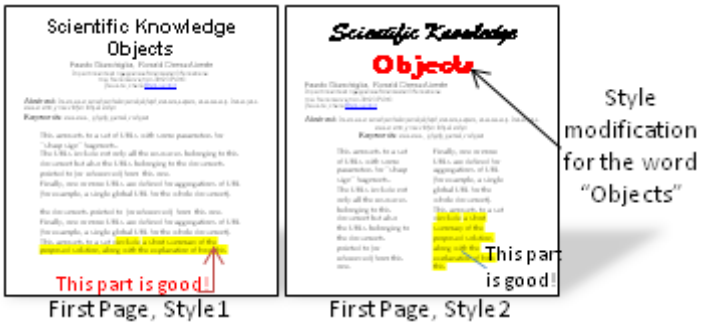

Figure 1.25: Example of the same document in two different styles.

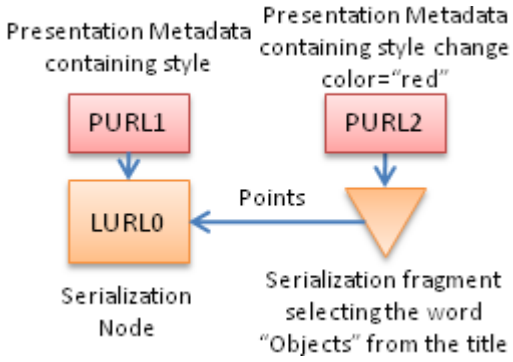

Figure 1.26: Representation of the presentation information for the 'Style2' document.

Presentation-layer SKOnodes

Presentation SKOnodes are used to define the specific link between the serialization layer object (choosing the content) and a presentation layer object (which contains the style in which the content will be presented). The definition of presentation SKOnodes is shown in Fig. 1.27.

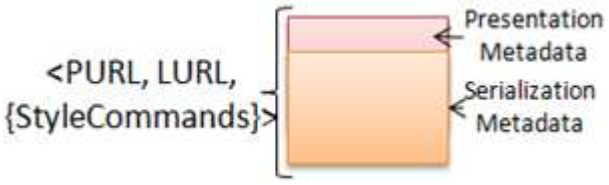

Figure 1.27: Presentation SKOnode definition.

The $PURL$ and $LURL$ elements from Fig. 1.27 define the link between the serialization and presentation objects. The possibly empty $\{StyleCommands\}$ set is, on the other hand, used to define the presentation information specific for the current SKOnode (for a list of these attributes refer to Appendix B). A concrete example of a presentation SKOnode was given in Fig. 1.26, where it was shown that the same serialization information could be presented in very different styles just by attaching different presentation-layer information.

Presentation Annotation

Much like the previously defined semantic annotations, the presentation annotations are used to capture specific and specialized presentation commands that may be applied to specific sections and that may even contradict the style defined in the more general presentation SKOnodes. These presentation annotations are defined in Fig. 1.28.

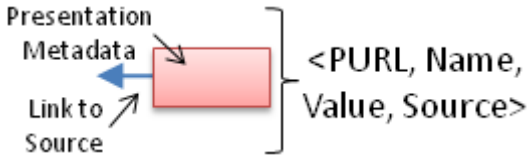

Figure 1.28: Semantic annotation annotation.

Note that by having; the $PURL$ element, along with the $Source$ elements (which provides a single target to for the application of the annotation) and finally the pair $Name$ and $Value$ (which encode the style change that its being applied); the presentation annotation definition from Fig. 1.28 almost mirrors the semantic annotation definition from Fig. 1.10. The only difference is the lack of a destination set, which is missing because, unlike semantic annotations, presentation annotations always refer to a single target. An example of a presentation annotation was given in Fig. 1.26, where an annotation was used to change the color of word "Objects" from black (defined in the SKOnode style) to red.

Presentation-layer SKOs

While presentation SKOnodes define style directives that are specific to their linked serialization object, presentation SKOs are used to define general styles that may be applied to several different documents. The definition of the presentation SKOs is done in 1.29.

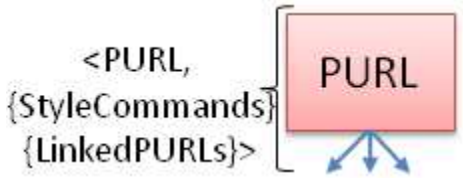

Figure 1.29: Presentation SKO definition.

The $PURL$ element from Fig. 1.29 contains the presentation address for the SKO, the $\{StyleCommands\}$ defines presentation directives, and the $\{LinkedPURLs\}$ is a set of PURLs that apply those presentation directives defined. Furthermore, a single serialization may have more than one presentation-layer attached to it, as shown in Fig. 1.30.

Note that, in the example from Fig. 1.30, the presentation SKO A is the most general style, which is modified individually for the nodes 1 and 2. Then, the substyle B adds its modifications to style A, while the nodes 3 and 4 introduce their own individual modifications for substyle B.

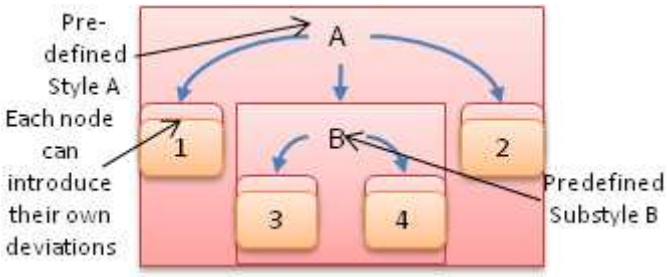

Figure 1.30: Presentation tree example..

Part 2

SKO Model Applications: Evolution, Search and Control

2.1 Introduction

With the structural definition of the SKO covered in Part 1, the discussion now focuses on the processes, operations operations and services that can be carried out using the SKO model. Notice that while the first part of this deliverable was a basically a model definition and specification, this part takes a much more experimental approach (i.e. has several proposals of possible features and implementations for the use cases to consider).

This part will propose an updated state-based evolution model (Sect. 2.2), to then introduce the new Version control and Branching considerations (Sect. 2.2), along with the Search and Navigation Considerations (Sect. 2.2). Finally, considerations on the ownership, licencing and control models that would apply to SKOs will be discussed (Sect. refsec:ownership-licensing).

2.2 The SKO State Dimension

In order to abstract away some of the more abstract and complicated properties from scientific artifacts (e.g certification, persistence), this subsection introduces an easy-to-understand metaphor based on the most well-known physical states of matter. As in the physical world, an object may have very different properties depending whether it is in one of the three discrete states (Gas, Liquid and Solid) introduced for scientific artifacts (as shown in Fig. 2.1).

The key properties that define each of these states are contained in Tbl. 2.1.

In more detail the main three properties abstracted away in this model are:

- *Maturity*: despite being a basically subjective property, maturity is the most representative and intuitive of all of state properties. Gas objects are normally used for highly fluctuating work-in-progress and deemed with not enough maturity to be considered serious. On the other hand, liquid objects are considered to have all their basic knowledge or science in place but still undergoing some adjustments, while solid objects are considered mature enough for

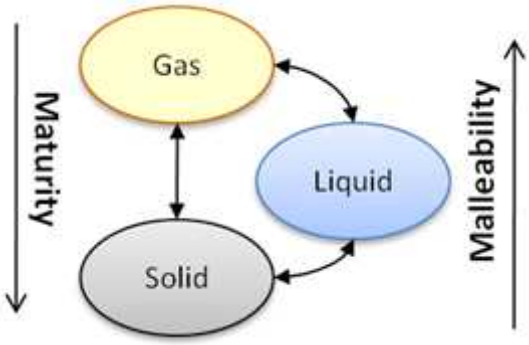

Figure 2.1: The SKO states.

Property/State	Gas	Liquid	Solid
Maturity	Unfinished, Work-in-progress	Draft, request-for-comments	Final
Certification	None	Author	Certifying Authority
Persistence	Unwarranted	Web	Web and Digital Libraries

Table 2.1: Main properties of the three SKO states.

being candidates for extending human knowledge or science.

- *Certification*: refers to the person or entity that takes responsibility and, eventually, credit for the content of the object. As shown in the previous table, it is the author that certifies liquid objects; which means that he assumes responsibility for the content in it. For solid objects, both the author and a certifying authority (e.g. a publisher a board of reviewers, etc.) assume responsibility for the artifact.

- *Persistence*: refers to the method used to warrant the availability of the objects over a period of time. This is important, for example, for citation-based processes; which need their objects to be relatively persistent to perform adequately. In a platform warranted persistence (liquid state) the SKO platform itself warrants that all liquid objects will remain available. On the other hand, in a "'platform and digital libraries"' warranted persistence, the object may be distributed and duplicated in digital libraries to further improve its availability.

From the point of view of the project's use cases, this state model is specially relevant to:

- *LiquidBook use case*: evolution and reuse of components and documents are the key factor of this use case and, as such, the state-based model becomes very useful for capturing the overall state of the very complex artifacts that this use case deals with. For example, if the state is able to be displayed as a visual cue (e.g. special icons or colors) just a general look at the subcomponent repository of a LiquidBook would be enough to have a general idea of their progress and maturity.

- *LiquidConference use case*: the liquid/solid states could be implemented into the paper submission life-cycle of the LiquidConferences site. Additionally, a special section of pre-publication papers or "'liquid"' documents could be created in the LiquidConferences web page. These documents would be subjected to a more lenient review process and would allow early dissemination of ideas and the obtaining of valuable feedback.

- *LiquidJournal use case*: the LiquidJournal use case would become the key to finding and giving access to the newly introduced liquid-state documents. Thus, based on the user's query, the LiquidJournal may contain the more stable but normally outdated solid-state content or the more up-to-date but still being updated liquid-state content.

The following subsections will further clarify and exemplify the three introduced states.

2.2.1 The Gas State

By creating objects in the gas state the authors themselves admit that the content of such objects is a work-in-progress, immature or untested. As even its authors do not certify the content of the objects in this state, it is considered having the lowest maturity and is not generally deemed important enough to ensure its persistence. This means that no reliable reputation (credit/blame) or citations can be derived from this type of objects. Nevertheless, these properties also allow gas-state objects to be the most malleable of all states; making them the default starting point for new ideas. The following list explores additional details about the gas state:

- *Main Purpose*: objects in the gas state are mainly used as the starting point for new ideas or for introducing changes to already established work. Frequent and unordered changes, are expected in this state.

- *Evolution*: since no persistence is warranted for gas state objects, they are in general the least constrained objects in the system. As such it is common to allow the deletion and modification of objects in the gas state, despite the fact that it may be referenced or used by other objects.

- *Target audience*:gas objects are normally aimed to a very reduced number of trusted collaborators. This is caused by the the frequent and loosely tracked modifications, along with the preliminary and evolving nature of ideas contained in this state.

A current example of what we call a gas state object would be a work-in-progress document stored in on local media and being worked on by the author exclusively or by a small group of collaborators over a LAN. The software equivalent of this state would be a development version of an application.

To start a more concrete example, consider the fairly common case of a scientific author writing a paper for submitting it to a conference. At the beginning author would create a first version of the paper that he wants to eventually submit (version 0.1). The author would then keep working on this gas state document (through versions 0.2, 0.3 and 0.4) until he finally reaches a point (version 0.5) where he feels that the theory and ideas of the paper are relatively stable and conveyed in a good enough manner. The author at this point would be interested in receiving feedback of his

current progress, he is however concerned about protecting his ideas and giving this work a little more of persistence (which would enable interested parties to reference or contribute this work). This is when the author would need the liquid state, introduced in the next subsection.

2.2.2 The Liquid State

Authors creating objects in the Liquid State generally acknowledge their work as not being final but still they deem it stable enough for partial dissemination. To protect the correct credit attribution of the ideas and content that they contain, liquid-state objects demand personal certification from its authors (authors take the responsibility and credit of the liquid-state content they have produced). Finally, the persistence of liquid-objects is enhanced to allow their possible (according to the access-control configurations set by the creators) citation and reuse.

The following list explores additional details about the liquid state:

- *Main Purpose*: much like a "'beta"' or "'candidate"' release in the software world, the liquid-state objects constitutes an important chance for the creators of getting important feedback that will be used to produce a final version. Another important purpose is updating the members of your community about the progress you have made on a given subject. So in that sense, liquid objects could also be considered as temporal increase in the human knowledge until a solid version is released.

- *Evolution*: persistence of the liquid-state objects is now warranted by the platform they have been published on and each liquid release will probably be kept for reference and archival even though a newer one may have been released (unlike in gas-objects in which their low persistence requirement allowed them to be freely deleted and overwritten). Furthermore, because of their maturity standards, the author is expected to release liquid-state versions of their objects in a somewhat less frequent manner (unlike internal gas-object that may get daily or hourly releases). Finally, providing the equivalent of "'patch notes"', or a list of the main changes done since the release of the last liquid version of an object could be either mandatory or strongly suggested (depending the the platform's requirements).

- *Target audience*: the previous characteristics enable to disseminate the object within a reference community in order to obtain feedback. Note also that it is the a finalized liquid-state object what will be submitted to a certifying authority/reviewers when the authors want to obtain the permission to release a solid state version.

Currently there are no widely-used examples of liquid-like scientific objects. However, software equivalent would be a Beta testing release of an application. This is why, one of the main objectives in the approach presented in this document is the introduction of the Liquid state objects, as a bridge between currently existing and widely used gas-like and solid-like objects.

Continuing the example of the author evolving a paper for submission to a conference, the author has just released a liquid version of his latest document (version 0.5). This liquid document quickly gets some feedback from the author's friend, as shown in Fig. 2.2[1].

[1]Note that in Fig. 2.2 the arrows actually represent actions and not relations between the entities as it was previously the case in the document.

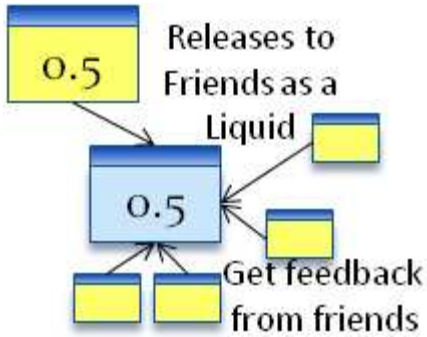

Figure 2.2: A liquid version is released to obtain feedback from friends.

While he/she waits for feedback the author continues to work and creates an internal (i.e. for his own use only) gas version (version 0.6). Later the author decides to include the feedback obtained into creating a new internal version (version 0.7), as shown in Fig. 2.3.

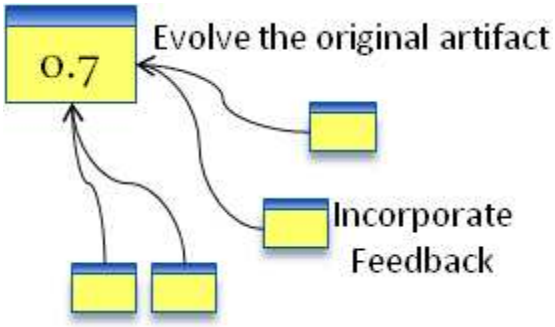

Figure 2.3: The feedback is used to evolve the original document in progress document.

After creating a new internal version (0.8) for addressing some finishing touches, the author is satisfied with the current iteration of his work and decides to release a liquid state document. This time (as shown in Fig. 2.4) he decides to submit it directly to the conference, where he gets reviews for that version.

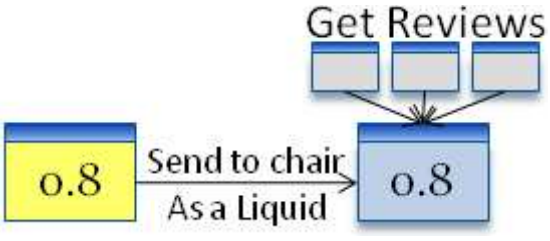

Figure 2.4: A liquid version is sent to the chair of a conference and gets reviewed.

The reviews that the author receives include interesting feedback and suggestions that he addresses in a new internal version (0.9), as shown in Fig 2.5. After the review round and the implementation of the reviewers' suggestions, the author addresses the last pending issues by releasing

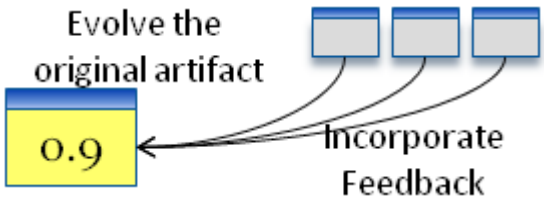

Figure 2.5: The review feedback is used to make correction and evolve the original document.

a final version (1.0). At this point, the author is confident that he/she has a high-quality and stable version of his work (version 1). Therefore, he/she is now interested in releasing and disseminating this work so it can be discussed and used by the general public. These are properties that belong to the solid-state objects that will be discussed in the next subsection.

2.2.3 The Solid State

As the state that is characterized by having the highest maturity, solid-state objects represent a milestone of "'maturity"' in the evolution of an artifact. To ensure this maturity and their quality, solid-state objects cannot be self-certified and are normally certified by higher-level organization (like a journal, conference or a publisher). Furthermore, solid-state objects are considered very important reference material. Therefore, their persistence is normally ensured by duplication in one or more (digital) library repositories.

- *Main Purpose*: the main purpose of solid-state objects is the increase of science or, more generally, of the human knowledge. As such, solid-state objects are the basic unit for dissemination and consumption for the general public.
- *Evolution*: as in the liquid case, state object persistence is also warranted for solid state objects, which allows every version of a solid object is remains available for reference. The only difference here would be that solid state objects generally tend to have far less versions that liquid state objects, as the certification procedure involved to produce a solid version is normally far more complicated/expensive.
- *Target audience*: the maturity and certification properties of solid objects ensure that they are ready for general public releases and wide dissemination.

The documents released through current publication practices (e.g papers, books, journals) are current examples of solid state objects.

As the conclusion of conference submission example, the author once again submits to the conference (this time the version 1.0 of his work). There, as shown in Fig. 2.6 the conference organizers finally approve its work and release a solid version (1.0) of this work.

Based on the previous definitions it is clear that solid state artifacts (or artifacts with equivalent properties) are already widely used and distributed (e.g. published papers, hard-back books, etc.). At the same time, gas state artifacts are also ubiquitous if one also considers the storage of personal computers (e.g. work-in-progress document, unordered notes). It is however much more rare to

Figure 2.6: A new liquid version is sent to the conference chair. The chair approves and releases the certified solid version.

find (especially in scientific environments) the use of artifacts with properties similar to those described for the liquid state.

As such, one of the key points of this model is the introduction of the liquid state objects (e.g. request-for-comment and intermediate-level documents) as a way to improve the collaboration and early dissemination in the scientific process. Furthermore, by introducing a new less demanding "'personal-level"' certification and allowing the reviewing/feedback loops to start from that point and thus enabling early dissemination.

2.3 Version Control and Branching

Using the already defined structures it is possible to create semantic tags that would be useful to identify when an artifact is a version of other artifact. As in other version control systems, the same mechanisms could be used to determine whether an artifact was merged or split from previous ones. However, when considering the SKO layered structure system, interesting options like part/whole and layered version control are also enabled.

From the point of view of the project's use cases, version control and branching is specially relevant to:

- *LiquidBook use case*: in the standard examples from the LiquidBook use case object related components and releases are constantly being added and updated. As such, a robust approach to version control is key for managing of the evolution of the base material (e.g text, images, exercises, lectures) and full artifacts (e.g. exams, books, courses) that emerge from them (and all the intermediate steps also).

- *LiquidConference use case*: conventional publishing platform and scientific repositories normally only store a single version of each paper. However with the proposed version control systems implemented, LiquidConferences would be able to also keep intermediate versions of the submissions. Furthermore, the SKO model would allow to capture and show the conceptual differences between these versions and to enable the users of the platform to discuss and refer to them.

- *LiquidJournal use case*: particular LiquidJournals could be configured to show the different versions of the same documents. This would offer a sort of "'insider look"' into the evolution of the documents, along with the conceptual evolutions related to the document's changes.

2.3.1 Basic Features and Implementation

Version control and branching are implemented in the SKO model by the use of the semantic annotations defined at Subsec. 1.3.2).

The "'is version"' annotation

Fig. 2.7, exemplifies a situation where the use of such annotations would be necessary to track the evolution between objects.

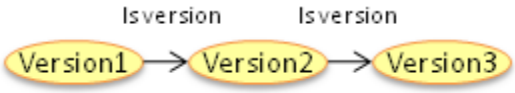

Figure 2.7: A simple version control evolution example.

As all semantic annotations from the SKO model, the "'is version"' annotation is essentially defined by the following elements:

- *Name*: in this case, this value is always set to "'is version"', this helps identify the semantic annotation as being in charge of version tracking.
- *A single source*: for version tracking annotations, the source points to the original or source object that was evolved into one or more additional objects.In Fig. 2.7 the source of the first version control operation from the left would be "'Version1"'.
- *Multiple destinations*: the destination set contains all the objects that were versioned from the object pointed by the source property. This includes both direct versioning (like the one existing between Version1 and Version2 in Fig. 2.7) and indirect versioning (like the one existing between Version1 and Version3). It is not known whether this redundancy will be found practical at the system's implementation time but it is kept for the moment, as it makes easier to determine the version control history of a given objects.
- *Value*: unused for version control annotations.

By using the previously defined version annotations it would be possible to represent the example from Fig. 2.7 into whats found in Fig. 2.8.

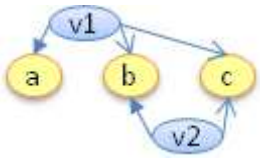

Figure 2.8: Representation of the previous version control example.

Fig. 2.8 has three nodes (a, b and c) representing the artifacts and two annotations (v1 and v2) representing the actual "'is version"' annotations. Note that the annotation v1 conveys that both b and c are *newer* versions of a, while the annotation v2 conveys that c is a newer version of b.

The following are some additional properties that the "'is version"' annotations have in their current implementation from D1.3:

- *Unique source*: this means that the only one "'is version"' annotation is allowed to exist for each source object. This restriction has the implication that all the tracking of the derived versions of a single object will be tracked in a single annotation (this may change however in the future due to implementation and scalability issues).
- *Fragments are allowed*: both the source and destination of a "'is version"' annotation may refer to fragments of objects (as defined in Subsec. 1.2.2) instead of full objects.
- *Recursion not allowed*: a single object is not allowed to be a version of itself.
- *Source and destination entities*: the source and destination from a "'is version"' annotation must both refer to the same type of entity (e.g. if source is a SKOnode, destination must be a set of SKOnodes).

The "'is split"' and "'is merge"' annotation

As exemplified in Fig. 2.9, two additional semantic annotations are necessary to help identify branches, forks and splits in the version evolution of an object.

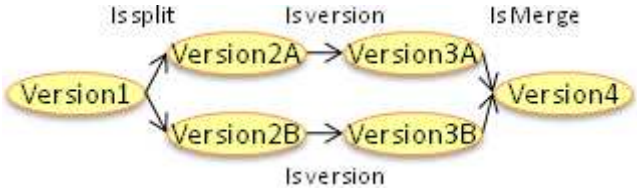

Figure 2.9: A simple branch/fork and merge example.

The first annotation type to be introduced is "'is split"', which is used to represent branches and forks. Split tracking is implemented as a semantic annotation, essentially defined by the following elements:

- *Name*: in this case, this value is always set to "'is split"', this helps identify the semantic annotation as being in charge of tracking conceptual fork and branches.
- *A single source*: for split tracking annotations, the source points to the original or source object that was split into one or more additional objects. The source of the split found in Fig. 2.9 would be the "'Version1"' object.
- *Multiple destinations*: the destination set contains all the objects that were created as a split from the object pointed by the source property. The destinations of the split found in Fig. 2.9 would be the "'Version2A"' and "'Version2B"' objects.
- *Value*: unused for "'is split"' annotations.

The second annotation is "'is merge"', which is used to represent the merge of two objects into a single one. Merge tracking is implemented as a semantic annotation, essentially defined by the following elements:

- *Name*: in this case, this value is always set to "'is merge"', this helps identify the semantic annotation as being in charge of tracking the conceptual merging of entities.
- *A single source*: for merge tracking annotations, the source points to the actual merged entity. The source of the merge found in Fig. 2.9 would be the "'Version4"' object.
- *Multiple destinations*: the destination set contains all the objects that were merged into the object pointed by the source property. The destinations of the merge found in Fig. 2.9 would be the "'Version3A"' and "'Version3B"' objects.
- *Value*: the value of "'is merge"' is used for storing an identifying tag or a reason for the merge. For example, the value for the merge annotation from Fig. 2.9 could be "'reunification of the work done by group A and B"'.

By using the previously defined version annotations it would be possible to represent the example from Fig. 2.9 into what is found in Fig. 2.10.

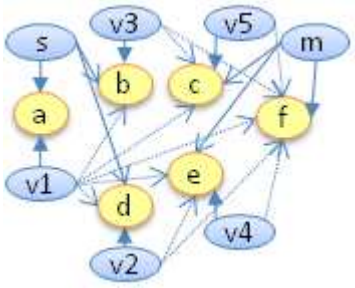

Figure 2.10: Representation of the previous branch/fork and merge example.

Fig. 2.10 has six nodes (a, b, c, d, e and f) representing the artifacts and seven annotations (v1 to v5, s and m) representing the version control relations. The list below will clarify the meaning of each of these annotations:

- *v1*: "'is version"' annotation that conveys that b, c, d, e and f are all newer versions of the artifact a (i.e. a is the version predecessor of them all).
- *v2*: "'is version"' annotation that conveys that e and f are both newer versions of the artifact d.
- *v3*: "'is version"' annotation that conveys that c and f are both newer versions of the artifact b.
- *v4*: "'is version"' annotation that conveys that f is a newer version of the artifact e.
- *v5*: "'is version"' annotation that conveys that f is a newer version of the artifact c.

- *s*: "'is split"' annotation that conveys that b and d are both splits from the artifact a.
- *m*: "'is merge"' annotation that conveys that f is a merging of c and e.

Similarly to the "'is version"' annotation both, "'is split"' and "'is merge"', have additional properties and limitations applied to them in the current D1.3 implementation. However, this information will not be repeated here, please refer to [2] for the details.

A final key point to note is that all the annotations introduced in this section do not actually capture how the artifacts are split, merged or changed. These annotation are more concerned on capturing the occurrences and reasons for this events at the conceptual level. This information will later be put to use to enable platform features like search and navigation, among others. The actual copying, splitting and merging of objects could be implemented into the platform by using common and open source version control systems. However the details of the actual implementation of version control for the model are still to be defined at the latter stages of this research.

Inverse tracking annotations

With the creation of each of the three of the previously defined version control annotations ("'is version"', "'is split"' and "'is merge"') the system will automatically create it's corresponding "'inverse annotation"'. These inverse annotations are mainly used to link the related entities in both directions and also to capture some additional information about the relation.

More specifically, the inverse relations related to version control are:

- *"'versioned from"'*: this is the inverse relation of "'is version"'. In "'versioned from"' the source component of the annotation points to the newly created version, while the destination component points to the original object (or objects in case of a merge) for this new version. Additionally, the value component of the "'versioned from"' annotation contains the version number or tag (e.g. "'5.3"', "'draft-0.6"') that belongs to the newly created object.
- *"'split from"'*: this is the inverse relation of "'is split"'. In "'split from"' the source component of the annotations points to one of the new splits created, while the destination component points to the original object (or objects in case of a merge) for this new split. Additionally, the value component of the "'split from"' annotation contains an identifying tag or phrase (e.g. "'changes added by John"', "'new solution tested"')that belongs to the newly created object.
- *"'merged from"'* this is the inverse relation of "'is merge"'. In "'merged from"' the source component of the annotation points to one of the original objects that were merged, while the destination component points to the new merged object (or objects in case of split) created. The value component of "'merged from"' is not used in this annotation.

To further explain the inverse version control annotations, consider the example on Fig. 2.11. This figure represents the object evolution presented in Fig. 2.9 but it uses inverse annotations to do so (as opposed to Fig. 2.10 that uses regular annotations).

[2] http://project.liquidpub.org/sko/API.html

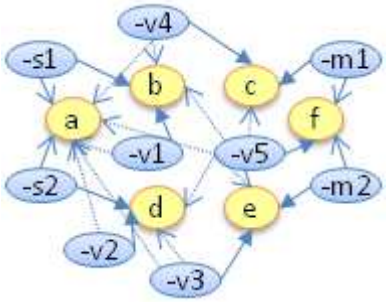

Figure 2.11: Representation of the inverse relations of the previous example.

Fig. 2.11 has six nodes (a, b, c, d, e and f) representing the artifacts and nine annotations (-v1 to -v5, -s1 to -s2 and -m1 to -m2) representing the inverse version control relations. The list below will clarify the meaning of each of these annotations:

- *-v1*: "'version of"' annotation that conveys that a is the previous version of the artifact d.
- *-v2*: "'version of"' annotation that conveys that a is the previous version of the artifact b.
- *-v3*: "'version of"' annotation that conveys that a and d are both previous versions of the artifact e.
- *-v4*: "'version of"' annotation that conveys that a and b are both previous versions of the artifact c.
- *-v5*: "'version of"' annotation that conveys that a, b, c, d, and e are all previous versions of the artifact f.
- *-s1*: "'split of"' annotation that conveys that a is the original source of the artifact b (that was created from a split).
- *-s2*: "'split of"' annotation that conveys that a is the original source of the artifact d.
- *-m1*: "'merge of"' annotation that conveys that c is one of the components that was merged into the artifact f.
- *-m2*: "'merge of"' annotation that conveys that e is one of the components that was merged into the artifact f.

The defined version control annotations and their inverse annotations are the basic building blocks which are then are used in the rest of the section to implement simple (state of the art) version control and also the version control for aggregation and layered version control explained in the following subsections.

2.3.2 Version Control for Aggregation

The clear identification of parts and wholes for each layer that the model includes, allows the version control system to track the evolution of each artifact and its aggregated artifacts almost

independently. Software artifact families are a clear example of this type of part/whole evolution, and examples of these can be found at [10].

Changes on aggregated objects

Introducing changes on aggregating objects refers to the action of adding or modifying information from the aggregated objects (or parts) and to the consequences this may bring over the objects aggregating them (or parts).

Changes in the aggregated objects may happen in any of the four layers defined in the SKO model, the changes happening to them would affect the aggregating objects from each of these layers in various ways. For example, if the text of a paragraph that it is used in both a paper and a book is modified this will cause the content of the paper and the book to change too. Expanding the previous example, if that same paragraph is linked to by annotation that states that it is used to prove a theory introduced in a presentation; then both the aggregating book and paper (up to certain degree) can be said to also help to prove the theory on the presentation.

Since it is clear that changes in the parts affect the wholes deeply, the following control methods could be implemented (according to the needs of the particular use case):

- *Auto-forking*: the owner of the aggregating object may choose to ask the system to automatically fork its components into a separate copies should they get modified in anyway (as exemplified on Fig. 2.12). By doing this the owner of the whole guarantees that the information of the wholes remains the same despite changes in the aggregated objects (but at the risk of pointing to outdated information while newer and more correct information is already available).

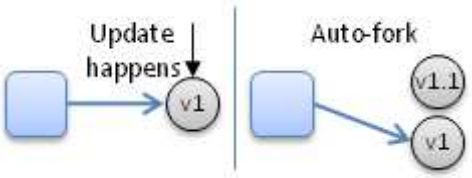

Figure 2.12: Auto-fork example: as soon as a modification is introduced for the pointed component, a fork is done by the system to preserve the information being aggregated.

  Note that this case would be the preferred for the creation of very complex and collaborative artifacts, like the ones from the LiquidBook use case.

- *Auto-updating*: the owner of the aggregating object may choose to ask the system to always point to the latest version of its components (as exemplified on Fig. 2.13). By doing this the owner of the whole guarantees that the most up-to-date information will always be propagated to the whole (but at the risk of pointing to less stable or finalized information). Note that this case would be specially relevant for dynamic aggregation objects like the ones from the LiquidJournal use case.

Note that these behaviors may be configured to operate in conjunctionwith the maturity-based states introduced in Sec. 2.2. For example, if modifications revert a component to the gas state;

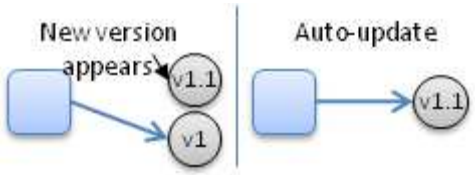

Figure 2.13: Auto-update example: as soon as a new version of the component appears, the system changes the "'part of"' annotation to point to it.

then it is probably better to just auto-fork to avoid introducing unstable or immature information into the aggregating object. On the other hand, if the system notices that a new available version of a component is liquid or a solid component; then it would become safer to auto-update to it. In any case, all changes and new versions of components should be always notified to the aggregating object owners. This enables the them to avail the impact of the change and to evaluate the proper course of action to take (much like a software programmer would be interested in being notified about updates to the libraries that his software uses).

Changes on aggregating objects

Similarly to the previous subsection, introducing changes on aggregating objects refers to the action of adding or modifying information from the aggregating objects (or wholes) and to the consequences this may bring over their objects being aggregated (or parts).

According to the SKO model introduced in Part 1, there is no file-layer aggregation (i.e. file-layer SKOs) defined. This means that in the SKO model the aggregating objects never introduces new file-layer content (e.g. the aggregating object cannot add a new picture that is not present on any of the aggregated parts); it can however introduce new information in the semantic-layer (i.e metadata and relations that emerge from the aggregation), the serialization-layer (i.e. content choosing and ordering) and in the presentation-layer (i.e styles to be applied to all original content).

But even if aggregation does not add new content to each whole, changing the metadata and the relations from the aggregating wholes could have implications on the aggregated parts. For example, if a reviewer adds his comments to a paper; these may also be relevant to at least some of the components of this paper.

In general however, at least for the moment, these types of modifications are deemed more as an informative tool than a content altering one. As such, having the system send notifications to the component owners about changes in the aggregating object should be enough to regulate these interactions (much like a software library creator would like to get notified when other applications start using or modify the usage of their libraries).

2.3.3 Layered Version Control

As in the SKO the artifacts are composed of several objects in different layers, it would be possible to version and change only some layers of a given artifact without introducing any changes to the rest. For example, changing serialization-layer objects would create a different execution of the original artifact (with more or less the same concepts but different granularity/order). On the other

hand, changing only the presentation-layer objects of a document would create new presentations and styles of the same concepts.

Like in the regular version control, the layered versions are represented by semantic annotations. Nevertheless, whereas in regular version control both the source and destination are in the same SKO layer, in the layered version control the source and destination are from different layers. The following subsections will further clarify the layered version control relations "'is execution"' and "'is presentation"' that were introduced on Part 1.

Executions

The execution relation is the main connection between the semantic-layer and the serialization-layer objects. It is mainly used to create different documents from the same general repository of content and knowledge.

Fig. 2.14 contains an examples on how the serialization-layer objects interact a semantic-layer objects to create different types of artifacts from the same knowledge and content.

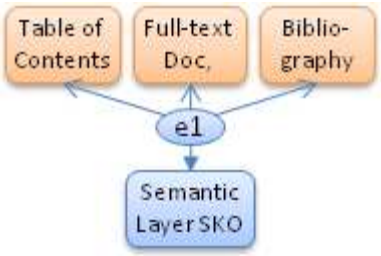

Figure 2.14: Representation of the example from Fig. 1.18, using the "'is execution"' relation.

As all semantic annotations from the SKO model, the "'is execution"' annotation is essentially defined by the following elements:

- *Name*: in this case, this value is always set to "'is execution"', this helps identify the semantic annotation as being in charge of execution tracking.
- *A single source*: for execution tracking annotations, the source points to a semantic-layer object that will be executed into one or more of serialization-layer objects.
- *Multiple destinations*: the destination set points to one or more serialization-layer objects that are the execution (i.e. contain the serialization-layer information for that object) of the semantic-layer object from the source.
- *Value*: unused for is execution annotations.

Much like the other relation-encoding annotations, "'is execution"' has an inverse annotation named "'execution from"'. In the "'execution from"' annotation the source points to a serialization-layer SKOnode and the destination set points to the semantic-layer objects that are are going to be executed in the serialization-layer SKOnode. Furthermore, the value element of "'Execution from"' contains an identifying tag or phrase (e.g. "'Abridged version of Open

Archives book"', "'Presentation for conference"')that belongs to the serialization-layer SKOnode. An example of such relation can be seen in Fig 2.15.

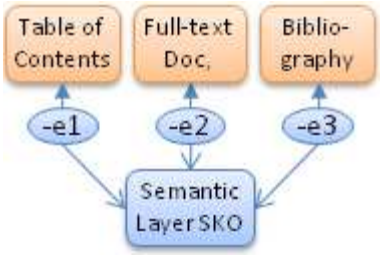

Figure 2.15: Representation of the example from Fig. 1.18, using the "'execution from"' relation.

More information about this annotation and several additional implementation-based restrictions can be found in the D1.3 deliverable.

Presentations

The presentation relation is the main connection between the the serialization-layer and presentation-layer objects. A presentation cannot change in any way the content and other information of the document so it is mainly used to create different visual styles and formats for the same document.

Fig. 2.16 contains an example on how presentation-layer objects interact with serialization-layer objects to produce two documents that look very different from each other.

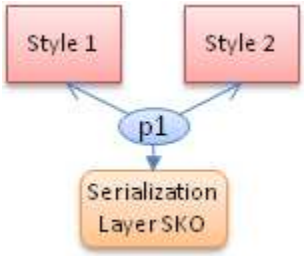

Figure 2.16: Representation of the objects joined by the "'is presentation"' relation needed for creating two different styles of the same document (as shown in Fig. 1.25).

The "'is presentation"' annotation is defined by the following elements:

- *Name*: in this case, this value is always set to "'is presentation"', this helps identify the semantic annotation as being in charge of presentation tracking.
- *A single source*: for presentation tracking annotations, the source points to a serialization-layer object whose presentation will be given in one or more of presentation-layer objects.
- *Multiple destinations*: the destination set points to one or more presentation-layer objects that are the presentation (i.e. contain style and format information for that object) of the serialization-layer object from the source.

- *Value*: unused for is execution annotations.

Much like the other relation-encoding annotations, "'is presentation"' has an inverse annotation named "'presentation from"'. In the "'presentation from"' annotation the source points to a presentation-layer SKOnode and the destination set points to the serialization-layer objects that are affected by the visual directives found in the presentation-layer SKOnode. Furthermore, the value element of "'presentation from"' contains an identifying tag or phrase (e.g. "'ACM paper format"', "'LCNS format for cell phones"')that belongs to the presentation-layer SKOnode. An example of such relation can be seen in Fig 2.17.

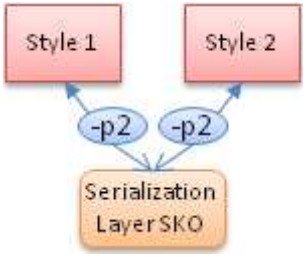

Figure 2.17: Representation of the objects joined by the "'presentation from"' relation needed for creating two different styles of the same document (as shown in Fig. 1.25).

Despite the fact that the presentation layer is not fully implemented in the SKO API from the D1.3v1, better specifications and details for the "'is presentation"' and "'presentation from"' annotations can be found at that deliverable.

2.4 Search and Navigation

This section will discuss the searching and navigation possibilities that are enabled by the SKO model. However, please note that the actual searching and navigation procedures are still not implemented in D1.3. These need to be specially tailored to the upcoming Liquid Journals, and they should also prove to be useful for Liquid Conferences and Liquid Books. As such, this section will receive a complete overhaul when these requirements are specified by the corresponding use case pillars.

2.4.1 Search options

The following are search strategies based on each of the four layer of the SKO model. These are not really meant to be carried out separately but to aid each other in order to offer the possibility of searching for unique aspects and allowing the return of the most adequate results to the user's query.

Concept-based

Concept search is based on structures from the Semantic Layer (SKOs, SKOnodes and annotations). At its most basic implementation, this search can be used to query for specific values in

metadata and relations that are common between the searched entities. Some examples of this can be found in the following list:

- *Keyword search*: the user's query is used for searching matches in the "'keywords"' attributes of the artifacts from the search space. For instance, the user could do a keyword search for the word "'knowledge"', and this search would return papers that have "'knowledge representation"', "'Knowledge evolution"' in its keywords. Note that a similar approach could be used for searching in the artifact's title, abstract or any other of its attributes.

- *Related document search*: the user's query may also include the relations existing between the different entities from the search space (e.g. comments, citations, older/newer versions)

Once a good body of material is in the system and enriched with semantics, more advanced versions of this concept-based search could be used by querying discourse representing relations between entities. For example, it would be possible to perform searches asking for artifacts that support or contradict a given artifact/claim.

Pattern-based

Pattern-based search is centered around serialization-layer objects that represent common patterns of complex objects. Searches of this type would involve searching for documents that follow specific structural or discourse patterns. For example, pattern-based search query would be "'documents that follow a Deductive pattern"' or "'documents that follow the Introduction-Body-Conclusion structure"'. More information about patterns can be found in Part 3 of this deliverable.

Again, this kind of querys are meant to be used along the querys from the other layers. As such, a more complete and realistic query would be "'documents from Fausto Giunchiglia that follow Deductive Reasoning"'.

Style-based

Style-based search is centered around presentation-layer objects that that represent the visual aspects of a document. Searches of type would involve searching for documents that are expressed in a common style or format. For example, a style-based search query would be "'documents with the ACM style"' or "'documents that are formatted to be displayed in a cellphone"'.

Content-based

Content-based search is mainly based on the structures from the file layer. This represents the, normally computing expensive, conventional full-text search. As such, this should be only used after narrowing down the search space with the previous methods. Note that for the moment, this type of search is only restricted to text-based artifacts.

2.4.2 Navigation options

In the SKO model, navigation is based around following the semantic-layer annotations that are created to keep track of the conceptual relation between objects and their evolution. Some very significant annotations whose navigation may prove interesting include:

- *Citation-based*: with enough data bootstrapped, citation networks that include not only papers and books but also other types of documents (e.g. presentations, webpages, etc.) would emerge. Being internal, these would be straight to update and navigate.
- *Version-based*: an evolution line, tree or graph (depending on the branch/merge history) would be able to be created based on the different existing versions of the same artifact and its components. Furthermore by using the state-based SKO evolution model as a filter and semantic-encoding evolutions, the evolution of knowledge along its different maturity levels would be much easier to see and navigate.
- *Components-based*: massively complex components (that have several levels of composition) will be made easier to navigate thanks to the creation of composition trees (which are not unlike the folder trees of current operating systems).
- *Executions-based*: when a single idea or base document is converted in several executions (e.g. conference paper, presentation, journal paper, one-page abstract, book), having a navigable execution-based graph would greatly help the user to find the granularity-level that he or she is interested in reading.
- *Presentation-based*: this would enable the navigation of related style patterns, which would help in choosing the most suitable visualization of an object according to several factors (e.g. output display, personal preferences, etc).
- *Discourse-based*: once enough semantic information is in the system, it would also be possible to create and navigate "'knowledge networks"' that will represents ideas and points of view as nodes and link them relations such as "'supports"', "'is opposed to"', etc.

2.5 Ownership, Licensing and Control Models

The following sections will explore how the previously detailed SKO model could be used to enable the concepts from the deliverable D2.1v1 (Process, role and licensing models for SKO lifecycle management).

2.5.1 Credit Attribution

The basic way in which the SKO model tracks authorship is by having an "'author"' metadata annotation (like the semantic annotations explained in Subsec. 1.3.2) applied to both SKOs and SKOnodes of each layer. Furthermore, in the current implementation of the SKO API (D1.3v1), the "'author"' is a default and mandatory attribute for both SKOs and SKOnodes.

However, this straight-forward authorship tracking is not sufficient to cover all the details of the options enabled by the different granularities of structures and their different layers introduced in the SKO theory. As such this section will discuss these new options and the possible ways of handling them.

Credit Attribution for Non-authoring Roles

Currently, roles like providing experimental data, writing, editing and typesetting of the document are traditionally all "'rolled-up"' into the the author role. However, in addition of allowing a higher granularity for content, the SKO theory is also able to provide a higher granularity of roles for all the contributors that participate in the creation of a given document. More specifically, the different layers of the SKO structure, presented in Part 1, are particularly aimed at some of these of these roles. For example:

- *File and semantic layers*: the contributors that work on the entities from these levels create content and semantic material related to it. As such, they normally fall in the role of data providers and author categories. The field scientific that obtains data and parses it into data and tables, along with the scientific writer that actually produces the document; would be concrete examples of contributors in these roles.

- Serialization layer: the contributors that work on the objects from this level normally take editing roles. This is the case, if they only select and order already existing content without actually changing the content themselves (much like in the case of journal editors, that do not really have control on the content they are ordering). In the case that the editor also introduces changes, he would additionally participate in the author/creator role in for the current document.

- Visualization layer: contributors working in these roles normally concern themselves more about the actual style or looks of the document, while detaching themselves from its actual content and their semantics. Known internally with the role of typesetters (although the names document stylist or visual designer would also be appropriate), these contributors are in charge of making the document comply with certain styling requirements (like for example, the ones set on a conference call) or of designing a attractive visualization style for it (like for example, the designer of a web page). Note however that if the typesetter also introduces changes in content or serialization (for example, reformatting to make things fit better with a style), he would also participate in the author/creator and editor roles for the current document.

In addition to the previous, it would also be possible to use annotations and metadata to encode further subtleties of the creation process. In particular, these could be used to:

- *Qualify the interaction*: annotations could be used when introducing changes for registering how mayor a contribution is. For example, this could be used to capture the fact that the typesetter only changed the colors of a figure to harmonize it better with the rest of the document (but he did not really participate in the creation of the figure).

- *Add new roles or types of interaction*: new roles that do not directly correspond to the layers of the SKO theory or even with the document creation (i.e. they happen before or after the creation process of the document) can also be captured through the use of annotations. These roles include funder, motivator, reviewer, data provider, secretary and supporter among others.

This separation of roles during the document creation and evolution is again particularly useful managing credit attribution in the LiquidBook use case; nevertheless, it is also relevant for the LiquidJournal use case. The creator LiquidJournal is, in fact, mostly doing editing work (he or she works mostly on the serialization layer in SKO theory terms) without actually having any power on the content itself.

Finally, note that by introducing new granularities and facets of credit attribution we are not implying that these are relevant for the reputation calculations related to scientific merit or value for researchers and/or papers. The intention of this section was mainly to highlight the possibilities and extensions to credit attribution that the SKO model is able to offer.

2.5.2 Licensing and Copyright

SKOs are also able to represent original works. As such, copyright laws and licensing regulations are also applicable to them. Much in the same way as authorship the basic copyright and licensing information is kept at each of the SKOs' and SKOnodes' metadata.

Conventionally, documents (like papers and books) are treated with "'blanket licenses"' that cover the whole document and all the content (e.g. figures, text, tables) that is part of it. Nevertheless, as it is in the case of credit attribution, the SKO model allows for licensing and copyright information to be defined at lower granularities. This would be similar the way a software artifact is composed of modules and libraries that may have different (and even contradicting) licensing information defined for them.

This is further complicated by the additional segmentation of contributions introduced in the SKO theory. For example with the SKO theory in place, it would be possible to define different licensing and copyright rules for the actual content of the document, another for the document pattern being used and a third one for the style and visualization aspects of the document.

The following subsections will propose two scenarios in which SKOs could be applied and will analyze approaches to deal with the previous issues. Note however, that it is not the purpose of this deliverable to decide which one of these is the "'best"' one or the one that matches better to the spirit of the project.

Closed Platform Scenarios

On an closed platform scenario all artifacts are created, viewed and handled exclusively through a regulating platform forbidding direct or external manipulation of the content. In this sort of platform, all downloads are normally forbidden and content may not be modified or even accessed without going through the platform. An example of such platform would be the smart phone operating systems (e.g. Apple's Iphone) that only serve and interact with content through closed systems.

This scenario would, in fact, allow easier enforcing of all the licensing and copyright restrictions of both the whole documents and its composing parts. It would enable both; authors who want to disseminate and make their content freely available and authors who want to apply restrictions for the access and (re) use of their content. Furthermore, with its freely available structure, the SKO-based structures could also be used between several of these platforms.

To implement such platform it would be necessary the additional definition of a locked SKOs format (by using encryption and other safeguard methods), on top of the conventionally open-format SKO, to prevent the information contained in the SKO from being directly accessed outside of its managing platform.

However, despite the all the implementable safeguards and legal measures, it is impossible to guarantee that any platform would be immune to unauthorized access and copying of their content (i.e. piracy); which is an on-going problem for all types of content. Furthermore, depending on its specific details, a system perceived as restrictive may not be well-received by a community that is currently pushing for more open standards.

Open Platform Scenarios

On an open platform scenario, all artifacts are detachable from the platform and it is possible to download, access and modify them by using any third-party application.

This scenario offers more content manipulation flexibility to the users but it also makes the licenses belonging to said content harder to enforce. In addition, this openness may be perceived as a liability by some authors and publishers; specially because SKOs are meant to be an open and low granularity standard for encoding scientific content. Because of these properties, being able to download the SKO representation of a document would be equivalent to downloading the source code of a software artifacts. More specifically, by having a complete SKO, the user would have access and editing rights not only to the whole document but also to each of its sub elements (e.g. figures, tables), the patterns used for building the document and the presentation styles used for is visualization.

Note that both, the scenarios presented here and in the previous subsection, do not really need to be mutually exclusive. Some authors may prefer their work to be accessible and editable anywhere (much like some authors create open-source software) while others would be much more comfortable with a more controlled environment. Regardless of which choice is made or whether both are chosen, and as previously explained, the SKO structure would be able to store all the information necessary for implementing these scenarios.

Part 3

SKO Typing and Patterning

This research contains some work-in-progress results of our ongoing research in the internal structure and composition of scientific articles or papers. Based on works like [9], [7] and [8]; the motivation of this research is to improve the SKO model so it can better capture all the key information about scientific artifacts and also enable the model to offer useful services. The next two sections will contain a quick summary of the progress in this subject.

3.1 Paper Patterning in Computer Science

In this section, we propose a pattern approach for scientific discourse representation grounded on general Scientific Method in Computer Science [3]. In Fig. 3.1 we modularize a paper according to the logical function of each of its component and we reorganize it by using a widely-used pattern for scientific papers.

Note that we divide the discourse of the paper into Data (i.e the main body of a paper that is constructed via the Scientific Method in Computer Science) and Metadata (e.g. bibliographic information, abstract, reference set, annotation). More in particular:

- *Global Metadata*
  - *Bibliographical Information*: Topic, Title, Author/Editor (Name, Affiliation, Email), Keywords, Category, Source (Journal, Conference, Inproceedings, Inbook, Article, Thesis, Techreport, Misc, Other), Publisher, Year, Volume (optional), Number (optional), Pages (optional), Series (optional), Edition (optional), Month (optional), Document Type, etc.
  - *Abstract*: a brief description of paper including Purpose, Method, Result and Content Map.
  - *Reference Set*: a set of referenced entities, such as a list of "'References"', Persons and Projects mentioned in "'Related Work"' and "'Acknowledgement"', a set of URLs or other entities in the Footnotes and Endnotes, etc.
  - *Annotation*: Comment, Review, Tag, or other Annotation.

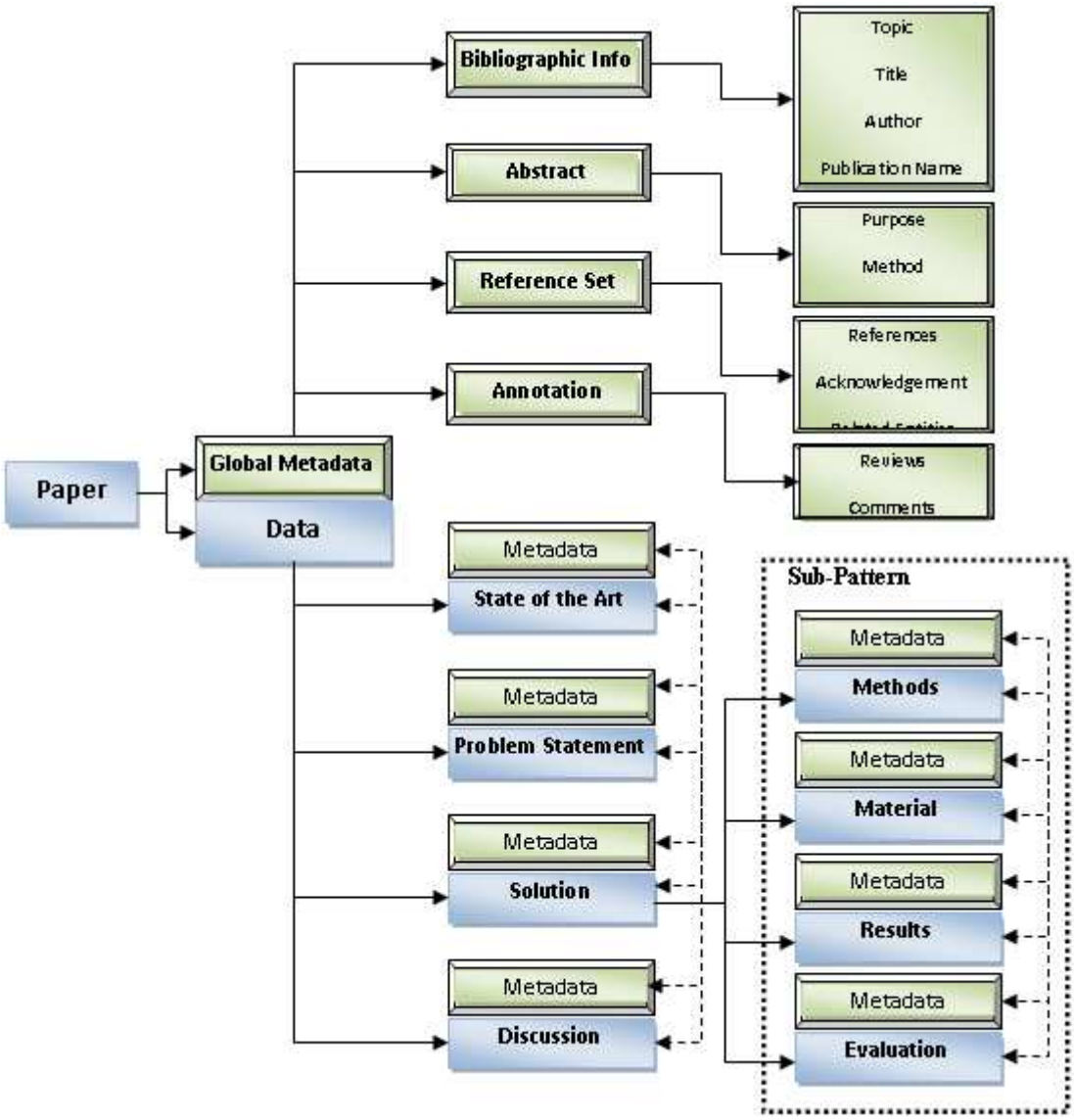

Figure 3.1: Example of patterned data and metadata of a scientific paper.

- *Data*
  - *State of the Art*: observations of phenomena, situation, foundational theories and related work, where the contextualized scientific problem addressed.
  - *Problem Statement*: the description and an active challenge faced by researchers and aimed to be solved in the discourse.
  - *Methods*: the specific techniques or methodology used in conducting a particular experiment.
  - *Material*: data collection, pretreatment, and analysis.
  - *Results*: the outcome or the findings of the research.
  - *Evaluation*: the evaluation methodology and its associated results.
  - *Discussion*: comparison results with related solutions or observations.

Focusing attention on the data part, we can observe several elements that are, again, made of the composition of metadata and data. We will call these compositions rhetorical blocks and, as we showed, their contents can be used to represent all the discourse of the scientific paper. While several types of relations can be thought to link these rhetorical blocks (e.g this block explains the other, this block justifies this, etc.), they could be thought as being essentially unordered. This comes from the fact that each person chooses their own ordering (or serialization in SKO terms), according to what they think is better for conveying their claims. Nevertheless, by collecting and analyzing data about current publication, we will try to find common and widely used serializations patterns for each type of paper. The following subsections present three of these extracted patterns.

3.1.1 Deductive pattern

In deductive reasoning the effort is focused on determining the conclusion when given the precondition and the rule that proves a the precondition implies a conclusion (i.e. $precondition \stackrel{rule}{\rightarrow} conclusion?$). For example, "'When it rains, the road gets wet. It rains. Therefore, the road is wet."'. Thus, the deduction method works from a general rule or principle towards a specific solution (as shown in Fig. 3.2).

The deductive pattern for scientific papers is then organized in:

1. *State of the Art*: investigate existing Theories and Observations. Related phenomena, development and analysis construct the Initial State S_i. Selected theories and techniques will support inference and argumentation as part of $rule$.

2. *Problem Statement*: predict a Target State S_t as a hypothesis for further test and confirmation. The problem statement presents the gap between S_i and S_t.

3. *Methods*: the way of design/refine/apply a $rule$, which leads $Si \stackrel{rule}{\rightarrow} St$. The method could be experimental method, numerical method, or theoretical method, etc.

4. *Material*: all the raw data, intermediary data, and pretreated data collected from the State of the Art that are used for Experimentation by proposed Method.

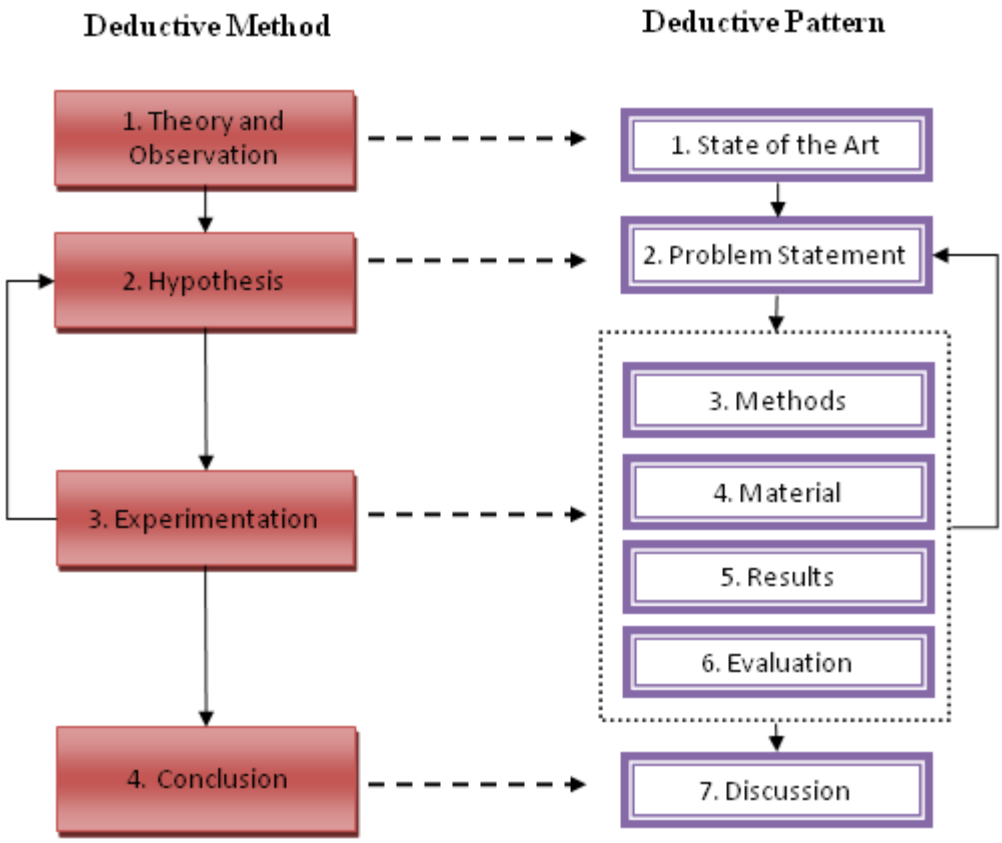

Figure 3.2: Deductive methodology diagram, along with its extracted deductive pattern for papers.

5. *Results*: present Middle State S_m, where $S_m = rule(S_i)$.

6. *Evaluation*: compare S_m with S_t. If S_m is not satisfied as expected, repeat the loop 3-4-5-6 with the modifications of $rules$ until the ideal S_m is obtained. Bote that some new problems may arise during the looping of 3-4-5-6; if this happens, return to 2 making a new sub problem statement and continue the recursion. When S_m (approximately) equals to S_t, break and go down to step 7.

7. *Discussion*: compare $rule$ and S_t with related observations from other scientists (S_i), always along with an old theory confirmed or applied within a new context.

3.1.2 Inductive pattern

In inductive reasoning the effort is focused on determining the rule that proves that a precondition implies a conclusion when given the specific precondition and the conclusion (i.e. $precondition \overset{rule?}{\rightarrow} conclusion$). For example, "'The road has been wet every time it has rained. Therefore, when it rains, the road gets wet."'. Thus, the inductive method works from specific observations towards general theories and principles (as shown in Fig. 3.3).

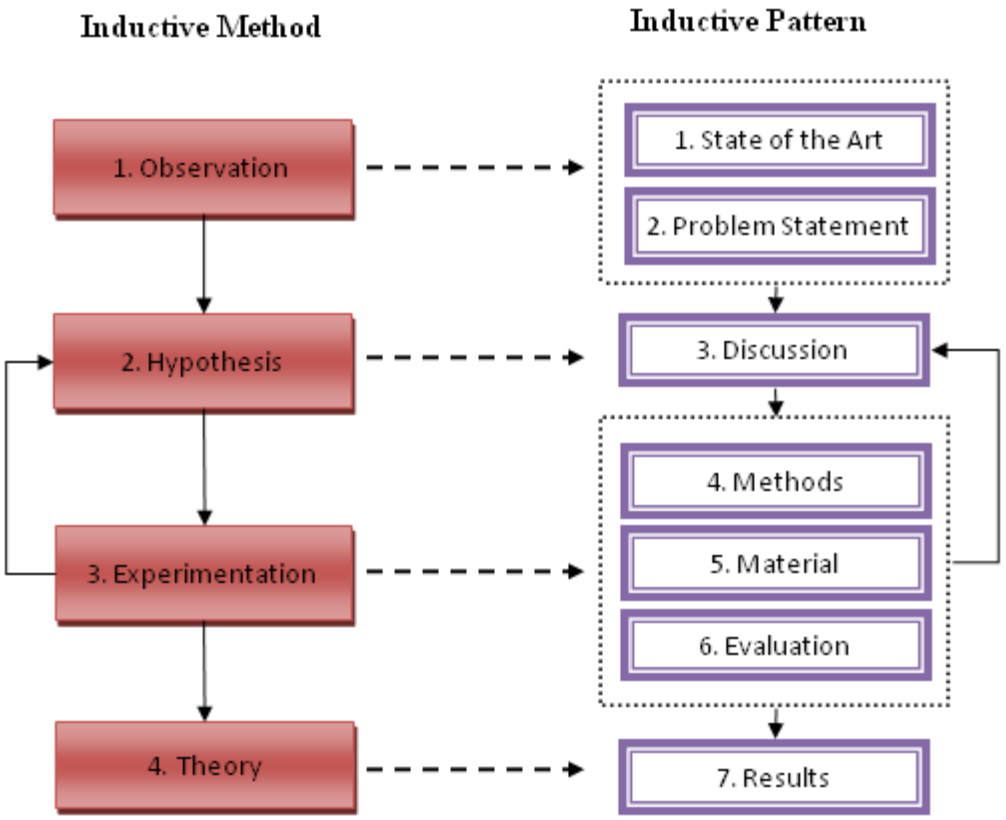

Figure 3.3: Inductive methodology diagram, along with its extracted inductive pattern for papers.

The inductive pattern for scientific papers is then organized in:

1. *State of the Art*: investigate existing Observations along with their theoretical explanations, and set them as the Initial State S_i.

2. *Problem Statement*: pose some phenomena as a Target state S_t, which can not be explained by existing theories or described by existing models. The problem statement aims at finding a $rule$, where it possible implies $S_i \rightarrow S_t$.

3. *Discussion*: observe and analyze the specific phenomena and particular scenario in S_i and S_t. The generalize and look for patterns of a general solution in a series of separated problems.

4. *Methods*: the scientific methodology, logic, or philosophic approach for deriving Rule from transmission SiSt.

5. *Material*: evidences, data, observations, etc, which support analysis and evaluation via proposed Method.

6. *Evaluation*: compare $Rule(S_i)$ with S_t. Repeat the loop 3-4-5-6 with the modifications of rules and methods until the ideal $rule$ is obtained.

7. *Results*: a new theory, $rule$, is proposed.

3.1.3 Abductive pattern

In abductive reasoning the effort is focused on determining the precondition when given the conclusion and the rule that proves that proves that a precondition implies a conclusion (i.e. $precondition? \stackrel{rule}{\rightarrow} conclusion$). For example, "'When it rains, the road gets wet. The road is wet, therefore, it may have rained."'. Thus, the abductive methods is a process of inference that produces an hypothesis as its end result (as shown in Fig. 3.4).

1. *Problem Statement*: pose a problem such as to derive explanations E out of observations O according to theories T, namely:
   (a) $T \cup E \models O$ and
   (b) $T \cup E$ is consistent.

2. *State of the Art*: investigate related observations, phenomena, facts, and set them as the target state S_t, while supported theories become the $rule$.

3. *Discussion*: observe and analyze the set of seemingly unrelated facts, and discuss various possibilities that an initial state S_i could be an explanation of S_t, where $S_i \stackrel{rule}{\rightarrow} S_t$.

4. *Methods*: the way of deriving S_i, for example, enumerative method, exclusive method, etc.

5. *Material*: evidences, facts, observations, etc, which support analysis and backtracking according to existing $rule$.

6. *Evaluation*: compare $Rule(S_i)$ with S_t. Repeat the loop 2-3-4-5-6 with the method modifications rules replacement until the ideal S_i is obtained.

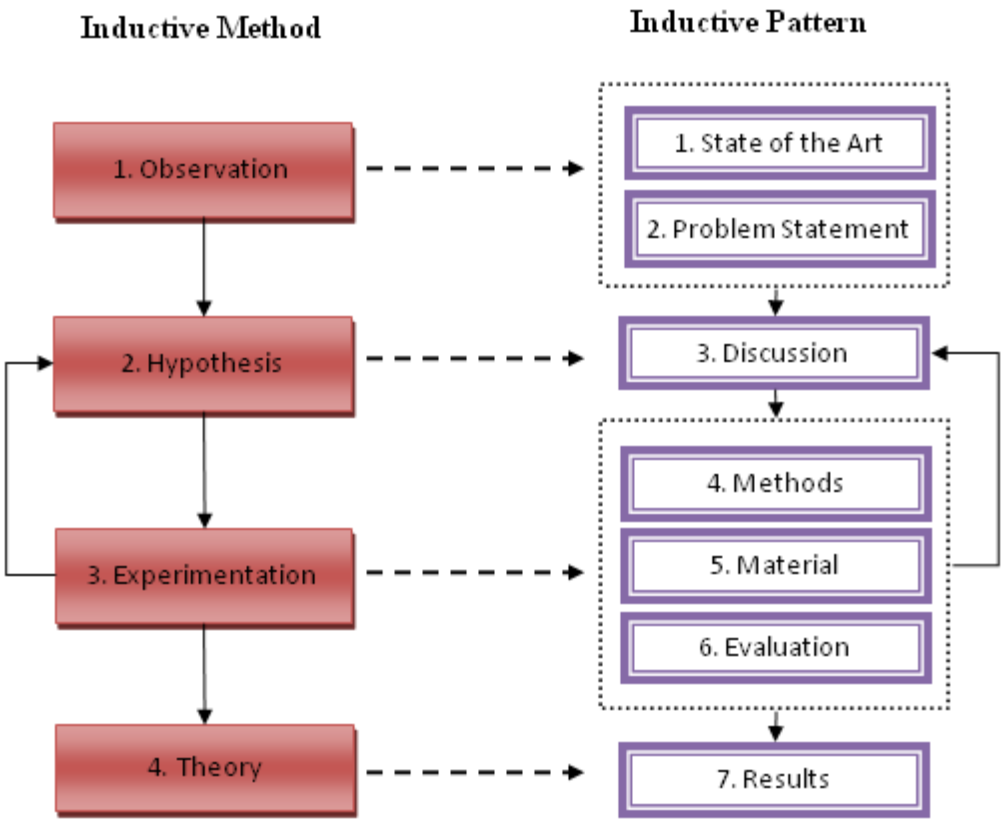

Figure 3.4: Abductive methodology diagram, along with its extracted abductive pattern for papers.

7. *Results*: phenomena detection or theory generation/development/appraisal.

In general the patterns focus on how the individual subparts of a paper come together to form a coherent whole. This research is specially interesting for the third layer of the current SKO model (described in Sec 1.4 from Part 1),as the patterns here explored could potentially be used for suggesting a flow or a linear structure for a set of components or ideas. Future iterations of this work would focus on expanding the presented patterns and validating them with a wide set of examples of existing papers.

3.2 Metadata Types for SKOs

More than how the different components of the SKO come together in patterns to form a paper, the typing section of this work is focused on the specific metadata attributes that both the parts and wholes of scientific artifacts should have to capture the key information about them. A first approach that analyzed several existing standards (e.g Dublin Core, Learning Object Metadata) at this was given on D1.2v1, in particular in the parts 3 and 4 of that document.

The specification found there was used for the specification of the fixed metadata attached to each of the entities described in the D1.3. In particular each of the attributes from the SKOtypes from D1.2v1 was either:

- *was included as a fixed attribute*: the attributes and relations that were considered to be of most common use were hard-coded (i.e. included manually) into the database used for representing the Scientific Knowledge Objects; or
- *was left to be encoded by free annotations*: the developed annotation systems allows additional attributes and relations to be added according the needs of the system and its users.

As the culmination of this line of work, we aim to prepare a final report detailing the LiquidPub Core Platform's most used attributes and annotations. With this information we would be able to determine what is the most useful set of meta-information (SKOtypes) to consider for both scientific artifacts and their components.

Appendix A

LiquidBook: Additional Details and Exercises

This appendix will cover an ongoing exercise that aims to apply the main concepts behind the LiquidBook (LB) use case by using already existing and widely-available tools. We will describe the details of an experiment carried out to test the concepts of the LiquidBook (explained in D5.1v2) and, more generally, the SKO layered structure. The final objective of this exercise is to measure the potential for improvement offered by LiquidBooks both in the artifact creation process and in the experience offered to students in the course.

A.1 A Course Framework through LiquidBooks

The exercise is based around a Mathematical Logics course, where the slides that compose the lecture can be freely annotated and contributed to by the students. All of the materials of the course (e.g. slides, references, complementary material, exercises) are stored in a repository, in such a way to support the creation of personalized versions of Mathematical Logics (or even other types of) courses that may be taught by different people in different places. The material in the repository should be organized according to its topic and kind of material and, potentially, across different versions in their evolution. The other important part of the experiment are the actual crystallized artifacts that are created (or serialized in SKO theory terms) from the repository in other to present content to other persons (in the case of our exercise, to the course's students). A diagram of this framework and of the carried out exercise is shown in Fig. A.1.

The following are the key elements of Fig. A.1:

- *LiquidBook Repository*: the center of the figure is occupied by the main content repository that we will call LiquidBook repository, this is where all the resources that are considered relevant for the LiquidBook are stored[1].

[1]For simplicity, we will assume that this central repository indeed exists; nevertheless,according to the SKO structural and LiquidBook definitions, this repository may not be centralized or even it may even be the "'Internet as a whole"'. This means, as previously implied in the SKO definition,that keeping the url and metadata each of the used resources should be enough to manage them and used them in LiquidBooks.

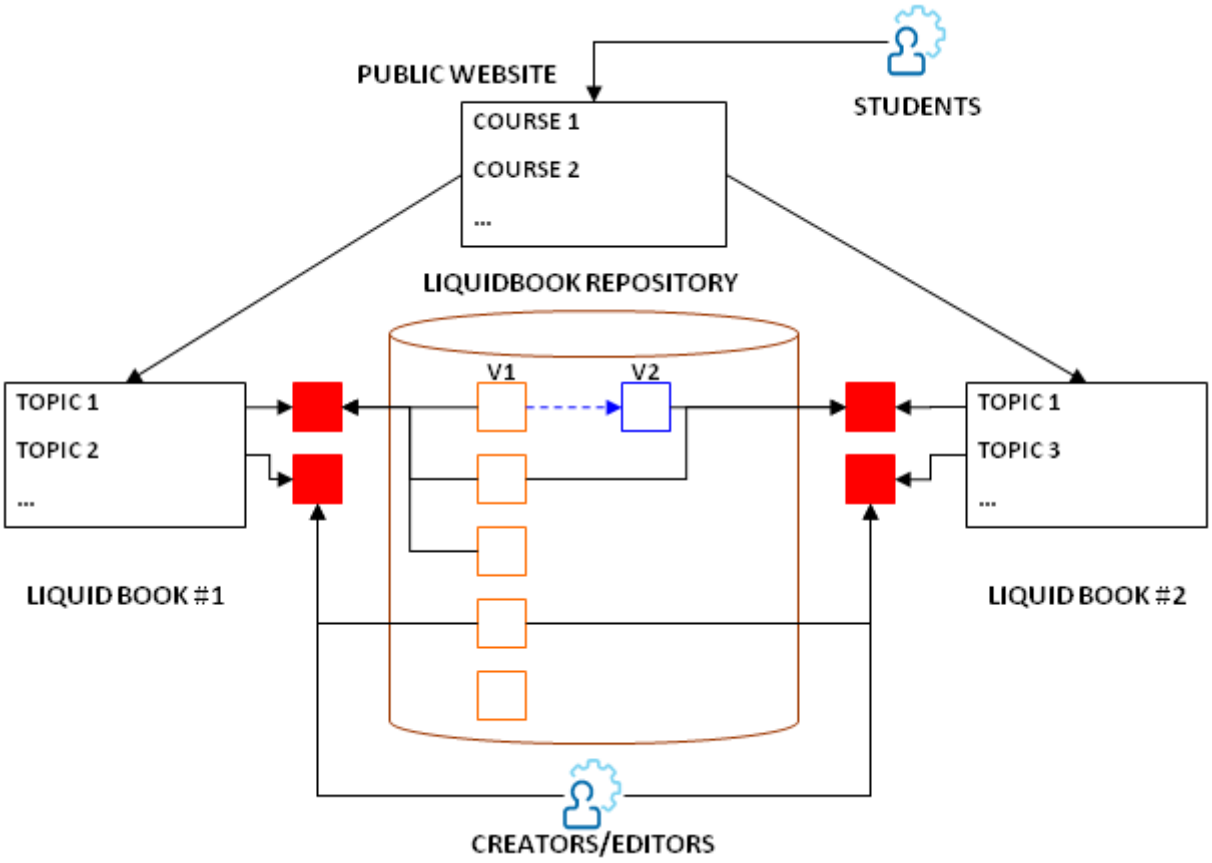

Figure A.1: Different elements of the LiquidBook exercise.

- *Resources*: the resources may be internal (like the squares drawn inside the repository), which means that the the creators have the ability of modifying or producing new versions of them; or they can be external (like the red squares drawn outside the repository), which means that they may be referred to but not modified.

- *Creators/Editors*: the creators produce new versions of the Resources used in the LiquidBook Repository, while the editors add metadata and serialize the existing resources into the crystallized LiquidBooks.

- *LiquidBooks*: these represent the crystallized LiquidBooks that are serialized from the resources existing in the LiquidBook repository to present the this content to other persons in an ordered fashion. Furthermore, these LiquidBooks may even be printed and become physical solid books.

- *Public Website*: technically this is another Liquidbook (i.e. an artifact created from the resources in the repository) but in this example it has the special function of becoming the bridge for the contributions and feedback from the students.

- *Students*: normally students only consume the knowledge offered to them in a course. However, thanks to the website and the LiquidBook model, they are also able to add their sub-

missions and corrections to the course.

A.2 Features Offered and Experiment Levels

- *Evolution management through version control*: this refers to gradually evolving content while keeping track of all the steps or previous versions that the content went through before reaching its current state.
- *Branching and Forking*: this refers to the ability of defining different evolution progress paths from a common original content. These different evolution paths may be later merged back to the original version (as in the case of branches) or remain separate and evolve separately (as in the case of forks).
- *Semantic annotations and relations*: allows defining semantic annotations and relations on every element and between them.
- *Note and Comment Repositories*: forum-like repository that can be used as unmoderated archival of student comments and discussions.
- *Management of complementary content (e.g. exercises, related links, etc.)*: used for categorizing, construction and keeping track of complementary materials. These can be mix and matched into different end results.
- *Serialization of repository into Courses and Lectures*: the repository is ”compiled” into actual lectures and courses (that are aggregation of lectures and other materials). The courses and lectures themselves can have their own version and branch history (independently of their originating repository).
- *New mechanisms for student collaboration*: offer methods for the students to contribute to the course and get rewarded for it.

A.3 Execution of Level0 exercise

This section contains the explanation of the procedure to carried out for implementing the Level0 version of the this exercise and the preliminary results obtained during the first instance carried our earlier this year.

A.3.1 General Procedure

1. All the materials (e.g. presentations, exercises, references, solutions, and professor notes) are put into a specially prepared SVN repository. This SVN repository has concrete naming and placing regulations defined for files to keep it easy to access and understand.
2. The lecturer gives the course lecture normally in class.

Level/Feature	Level0 - in use	Level1 - cont. management	Level3 - possible LP platform
Evolution	SVN	SVN	SKO-tracked version control
Branching	Manual tagging	Manual tagging	SKO-tracked version control
Semantics	Txt files	Txt files	SKO Semantic Layer metadata/annotations
Comments	None	Forums	SKO Semantic Layer metadata/annotations
Extra Content	Stored at Repos.	Stored at Repos.	Linked through semantics
Serialization	Manual + Txt files	Manual + Txt files	Automated, compilation-like
Student Collab	Email, manual selection	Forums	Community selection

Table A.1: Exercise implementation Levels

3. After the lecture, the lecturer creates a sub directory in the SVN repository, where he places, the presentation and others materials that he used during his lecture. Furthermore, he can add notes and comments related to this lecture like for example "'students had problems with this or that part"'

4. All the materials from the course are converted into webpages and published in the web page of the course. It is important to note that the original files are NOT given to the students. For example, each slide of the presentation is converted into an image file and posted independently in the course's web page. This is done mainly to encourage students to keep coming back to the course's webpage (as opposed to visiting the course webpage once to download all the slides and never coming back)

5. Students are encouraged to submit their class notes, corrections, links, solutions to exercises, new exercises. These notes are directed to a previously prepared email address, where course administrators select the most insightful and interesting ones and publish at the course's webpage (directly under the slide or resource to which their refer to). A substantial incentive (5 points out of the scale) is offered to students as an incentive for submitting notes.

6. For creating tests, the course manager use the course's webpage as the basic resource.

7. By the end of the course, two important resources have been created:

   (a) The course repository: that contains the artifacts (and more generally the knowledge) from all courses.

   (b) The course webpage: that represents how that particular course was given.

8. For future iterations of the course, the course administrators can use and evolve the course repository as they see fit. For example, the same repository could be used to teach the course in different languages and different countries (e.g. China and Paraguay), on different times (e.g. the 2010 course, the 2011 course) and different subject depth (e.g. introductory course, advanced course). As such, each instance in which a course is used will produce a new serialization of the original repository (i.e. the course's webpage) and while the repository itself does not need to be duplicated, it will evolve each time it is used to give a course.

9. After several courses, the managers of the repository may decide to create a physical book on the subject matter. New materials may be created for this purpose but also some material already existing at the repository may be reused (e.g slide text, exercises and student comments/exercises from the students). Again the creation of the book would create a separate book artifact (much like the webslides from the course) and also evolve the repository of the course.

A.3.2 Preliminary Results

The experiment was carried out in the Mathematical Logics course[2] that started in February of 2010 and ended in June of the same year. More than 40 students were asked to send in their submissions to improve the course and, over the five months that the course lasted we compiled over 150 submission entries from 30 different participants. The most common type of submissions were (in no particular order):

- *Corrections to the course's slides*: this went from simple typos and pointing out corrections to some exercises, to discussing conceptual approaches and the order in which the lessons were organized.

- *Class notes*: basically, the notes that the students take during the lecture. This kind of submission is interesting for gaging what the students understand and take away from the lectures.

- *Links to external sources*: one of the most common submissions, this included links to Wikipedia articles and sometimes even papers that contained additional explanations or exercises related to the subject of the lecture.

- *Exercises and exercises with answers*: exercises related to the subject of the lecture or extending the exercises given during the lecture. Unfortunately, few of these submissions had also the answer.

- *Study Notes*: these were sent towards the end of the course and normally contained summary of concepts and formulas from the lessons

All the submissions were subjected to a review by the course owners and finally about 40 of them were selected and approved to make part of the LiquidBook repository of the course. Furthermore, a subset of these accepted submissions were published at the course's webpage; each submission placed below the slide that they actually referred to. The objective of this was

[2] http://disi.unitn.it/ ldkr/ml2010/

to make available these corrected and approved resources to the students that had create them, in order to help them better understand the concepts of the course and prepare better for the upcoming exams.

A.4 Final Words

This section has presented a preliminary account of the results of the first of the exercises related to a SKO-enabled LiquidPub. We have still to perform a rigorous analysis on the data we have obtained[3] but the reaction from the students has been positive and we believe that the information we have received through the submissions will become useful for new iterations of the Mathematical Logics course.

To continue this exercises the following tasks are planned for next year:

- *Finish data processing for the first exercise*: update and order the course's repository based on the latest submissions, while performing an in-depth validation of the submission quality and utility for the evolution of the course. To close this task a final document detailing all the results of the first exercise will be created.
- *Perform a second Level0 course exercise*: this consists in using the updated repository to create a new (hopefully better and more evolved) version of the course to be presented later this year.
- *Perform a Level1 course exercise*: approximately by the end of the year, attempt to have a basic content repository tool set up, in order to enable an easier and more organized interaction with the students.
- *Create a final report of the exercise*: based on the previous two exercises create a final report on the LiquidBook use case exercise, try to cover details about its usefulness and applicability and give recommendations to possible future implementations.

[3] It is worth noting that this appendix section was written two week after the end of the course so a more in-depth analysis could not be performed in time to be included in this deliverable.

Appendix B

SKO XML proposal

This appendix proposes an XML notation to represent all the concepts from Part 1 of this deliverable. This XML representation would be specially important for a SKO-based content and metadata creation application (frequently referred to as "'SKO editor"') .The need for actually implementing this XML format for representing the SKO concepts will be indeed decided by the project use cases. Nevertheless, creating example artifacts using this XML format (e.g. have year 3 deliverables and related presentations) may prove to be an interesting exercise that could provide insights on several aspects of the SKO theory and in general scientific publications.

The following sections will propose the XML representation for each of the described SKO layers, with applied example of a concrete artifact using this XML format to be added later to the SKO section of the LiquidPub project page[1].

B.1 File-layer XML

Since almost no additional information or features are added on top of the existing file systems, this section contains the fairly basic markup elements used to implement the file layer into XML.

B.1.1 The file_node Element

Attributes:

- *url*: mandatory attribute that specifies the URL of the file node.

B.1.2 Referencing a File Fragment

To reference a file fragment in any element, add the optional 'fragment' attribute along with the 'url' argument. The format of the value of this argument is '#start1,end1#start2,end2#...#startn,endn'. Where the numbered start and end represent:

[1] http://project.liquidpub.org/research-areas/scientific-knowledge-objects-sko

- *start*: position in content units (e.g. characters for text, pixels for images) marking the start of the fragment within the original file (or zero if the fragment starts at the beginning of the file).

- *end*: position in content units marking the end of the fragment within the original file (or it has the maximum size if the fragment ends at the end of the file).

B.2 Semantic-layer XML

Several specifications like Dublin Core[2] , ABCD [2] and SALT [6] have been considered for the standardization of the XML schema for this layer. Future versions of the SKO specification may even include direct support and interaction with some of these standards.

B.2.1 The sem_skonode Element

Arguments:

- *surl*: mandatory attribute that specifies the SURL of the semantic SKOnode.

- *url*: mandatory attribute that specifies the URL of the file node to which the current SKOnode refers to.

Furthermore, as its attribute set definition it may have the following possible simple sub-elements: 'title', 'abstract', 'keywords', 'language', 'creation_date', 'pub_date', 'state'.

B.2.2 The sem_annotation Element

Arguments:

- *surl*: mandatory attribute that specifies the SURL of the semantic annotation.

- *name*: name of the annotation. The "name" argument can be one of the following values: 'is version', 'versioned from', 'execution from', 'related to', 'comments', 'Review of', 'part of', 'serialization of', 'split from', 'merged from', 'execution of', 'abridged content of'.

- *value*: a text value containing the actual metadata property.

- *source*: address of the main object to which apply the annotation

Furthermore, as the destination set definition, it may have 0 or more 'destination' sub-elements, where each has an 'address' argument. Each of these 'destination' sub-elements define additional objects to which apply the annotation.

[2] http://dublincore.org/

B.2.3 The sem_sko Element

Arguments

- *surl*: mandatory attribute that specifies the SURL of the semantic SKO.
- *part_of_annotation*: SURL of the 'part of' annotation that links this SKO with its immediate components.

Furthermore, as its attribute set definition may have the following possible simple sub-elements: 'title', 'abstract', 'keywords', 'coverage', 'language', 'creation_date', 'pub_date', 'state'.

B.3 Serialization-layer XML

As fewer works on the subject for artifact aggregation and content/metacontent selection were available, the serialization-layer XML schema was created specifically tailored to the SKO approach.

B.3.1 The serial_skonode Element

Arguments:

- *lurl*: mandatory attribute that specifies the LURL of the serialization SKOnode.
- *root_surl*: mandatory attribute that specifies the SURL of the semantic object to which the current serialization SKOnode refers to.

Furthermore, one or more 'item' sub-elements may be defined, with the following attributes:

- *surl*: mandatory attribute that specifies the SURL of the semantic object that is being included in the serialization.
- *param*: one of the following values; full, abridged, title, author, reference. Used to determine what information from the included objects is going to be included in the serialization.

B.3.2 The serial_sko Element

Arguments:

- *lurl*: mandatory attribute that specifies the LURL of the serialization SKO.

Furthermore, one or more 'item' sub-elements may be defined with the following attributes

- *lurl*: mandatory attribute that specifies the LURL of one of the serialization objects that this serialization SKO aggregates

B.3.3 Referencing a Serialization Fragment

To reference a serialization fragment in any element, add the optional 'fragment' attribute along with the 'lurl' argument. The format of the value of this argument is '#start,end'. Where the start and end represent:

- *start*: position in serialization units marking the start of the fragment within the original serialization.
- *end*: position in serialization units marking the end of the fragment within the original serialization.

B.4 Presentation-layer XML

‘

For the purposes of this document, we establish a simple and custom XML schema for the presentation elements. But future versions of the specifications may include partial or full support for presentation-semantics markup languages like CSS.

B.4.1 The pres_skonode Element

- *purl*: mandatory attribute that specifies the PURL of the presentation SKOnode.
- *lurl*: mandatory attribute that specifies the LURL of the serialization object to which the current presentation SKOnode applies.

Furthermore, as its attribute set definition it may have the following possible simple sub-elements: 'font', 'color', 'font_style', 'font_size', 'paragraph_style', 'column_style' among others.

B.4.2 The pres_annotation Element

- *purl*: mandatory attribute that specifies the PURL of the presentation annotation.
- *name*: name of the annotation. The "'name"' argument can be one of the following values: "'font"', "'color"', "'font_style"', "'font_size"', "'paragraph_style"', "'column_style"', among others.
- *value*: a text value containing the actual metadata property.
- *source*: address of main object to which apply the annotation.

B.4.3 The pres_sko Element

- *purl*: mandatory attribute that specifies the PURL of the presentation SKOnode.

- *attribute set*: may have the following possible simple sub-elements: 'font', 'color', 'font_style', 'font_size', 'paragraph_style', 'column_style' among others.
- *item_purl* one or more sub-elements named "'item"'that each specify the PURL of the presentation object that this presentation SKO aggregates.

Appendix C

SKO Model Changes and Requirement Compliance

With the introduction of the three use cases (LiquidJournals, Liquid Books and Liquid Conferences) during year two, the SKO had to accommodate and answer to a new set of evolving requirements. In addition to that, "'The SKO model and the LiquidPub System"' was a part of D1.2v1 that contained requirements and restrictions that the SKO model should comply with to allow the application of particular reputation metrics.

The objective of this appendix is to better cover the evolution of the SKO model from from year one to year two and to a provide (or point to) a direct answer to its use case and reputation requirements. Finally it will also identify the features of the SKO model that are yet to be used by any use case or application.

C.1 Main Evolutions of the SKO theory

This section will quickly go through the main changes that the SKO model and theory have went through in the last year. In particular, each of the subsection will focus on a main aspect of the SKO theory.

C.1.1 SKO Structure

Originally the SKO theory was split in three levels: SKOnodes, SKOs and SKOnodes. Nevertheless, the rising importance query-based collections (i.e. SKOsets) in the Liquid Journal use case led to considering them also as "'first-class citizens"' (i.e. a central concept) to the model and caused the second and third layers from the year1 SKO theory to be merged. Furthermore, the need of managing a wide variety of artifacts in different stages of evolution in time and with different representations and formats required by the Liquid Book use case (and also, but in a somewhat less central way, by the Liquid Journal use case) led to the addition of two new layers for capturing executions (same basic ideas different end artifact or concept granularity) and pre-

sentations (same ideas, same artifact and organization, different visual/format representation). In any case, if anything the SKOs were reorganized and extended from the year one three-leveled model to the more general four-layered model described in part 1

C.1.2 SKO Evolution

The refinement of the evolution model was driven by the requirements of greater flexibility from the use cases. As such, the first year evolution model was split into two independent subcomponents for the second year. The first component is the state-based evolution model that it aims to capture the more human/subjective maturity aspect of scientific artifact evolution while also offering new certification and persistence opportunities. On the other hand the second component is focused on capturing the different evolution steps (version control) and paths (branch/fork control) that the scientific artifacts go through. Information about the evolution model may be found in the part 2 of D1.2v2.

C.1.3 SKO Types and Patterns

While the work on the first year focused more on types or the internal attributes that are useful to be captured in scientific artifacts and their components, during the second year a significant push towards patterns or the way in which these components are aggregated into full artifacts was made. The results of this work may be found on part 3 of D1.2v2.

C.2 Matching Requirements to the New SKO

This section will provide or point to the matching of concepts and requirements from the main two sources of requirements considered for year2.

C.2.1 Reputation Requirements

The second part of D1.2v1 states that architecturally the Liquid Publications should be composed of person entities, SKO entities and the relations between these two groups. This is shown Fig. C.1 where we reprint a diagram from that deliverable.

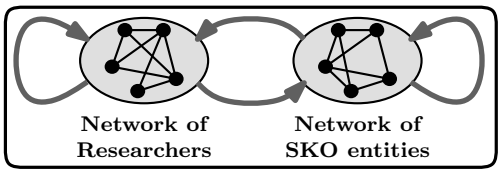

Figure C.1: Different elements of the LiquidBook exercise.

It is worth noting that the interactions required by the three main elements from Fig. C.1 (i.e. SKOs, Persons and Relations) are supported by the update of the SKO theory that is contained in

part 1 of this document. In particular, the following are the three entities from the new model that correspond to these requirements:

- Persons: researchers are represented by person entities. It is worth noting that the current version of the SKO model does not define its own entity specification for persons, authors or researchers; leaving this instead to each of its use case pillars. This has, however proven to be a common feature for all three use cases so its internalization to the model will be considered for year 3.
- SKOs: as explained before, the year one concept SKOset was included into the SKO structure. Because of this, the new SKOs and SKOnodes explained in this deliverable are the ones that comply with all the needs of scientific artifact and knowledge representation of the reputation requirements.
- Annotations: also known as SKO annotations, these implement all the relationship types specified in the reputation requirements (e.g. Author, Reviewer,Downloaded, Part of, Version of, Review of). Furthermore the implementation of the SKO annotations offers the possibility of extending it with new relations and type-checking each relation for integrity/consistency.

The formal specification defined at that document can also be easily adapted to hold for the new SKO theory. Nevertheless, the internal integrity constraints maintained by the current SKO model are fairly lenient. The SKO model basically limits itself to defining the basic rules and conditions for the relations and annotations, while leaving the definitions of correct use up to the pillar use cases.

As a final consideration of compliance, the motivational example from the final sections of the reputation requirements describes a conference example. The Liquid Conference use case can be considered as a simplified answer to this motivational example (with some of the processes implemented "'offline"').

C.2.2 Matching with Use Cases and other Features

The matching done between the SKO model, use cases and other external features/concepts effectively makes the SKO theory and its implementation (the LiquidPub core platform) one of the main convergence points for all the work done in the project. The conceptual matching of the SKO model with the three use cases may found on D5.1v2 and the support for externally developed functionalities and concepts (e.g licensing and reputation metrics) through an annotation-based metadata handling may be found on part 2 of D1.2v2.

C.3 Aspects of the SKO theory not yet Applied

While the changes and features of the SKO model have been affected in a bottom-up fashion, some features from the SKO model where also planned as anticipation of future feature requests and future extensions or use case.

In particular, the following features from the SKO model has yet to be applied directly on any of the use cases:

- *Serialization Layer*: the serialization of subcomponents is still not crucially required by Liquid Journals or Liquid Conferences. A "'is serialization"' relation is currently supported between SKOs to identify artifacts that just contain a different ordering or granularity of the same content. This feature would become much more important if the Liquid Book use case is decided to be implemented during year three.
- *Presentation Layer*: similarly to the serialization, apart from tagging similar SKOs, this layer is not importantly needed by any of the currently implemented use cases. The implementation of these concepts is pending to the needs of the Liquid Book use case. However, some partners have already commented on how they think this is mostly out of the current scope of the project.
- *XML representation*: the XML representation of the SKO model has still to be used on any of the use cases. This would become important if in the future we have a need to implementing a "'SKO editing platform"' that would enable the creation of data and metadata for scientific artifacts.

These features are naturally given the least possible importance for implementation (i.e. none of the previous features are currently directly supported by the implementation in D1.3v1). Their theory is however already defined and available for comments in case that they want to be extended and implemented during year three.

Bibliography

[1] CASATI, F., GIUNCHIGLIA, F., AND MARCHESE, M. Publish and perish: why the current publication and review model is killing research and wasting your money. *ACM Ubiquity* (11 2006).

[2] DE WAARD A., AND G., T. The abcde format enabling semantic conference proceedings. In *SemWiki* (2006).

[3] DODIG-CRNKOVIC, G. Scientific methods in computer science. In *Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden* (2002).

[4] FABIO CASATI, FAUSTO GIUNCHIGLIA, M. M. Liquid publications: Scientific publications meet the web. Tech Report from Unitn, found at http://eprints.biblio.unitn.it/archive/00001313/01/073.pdf, 09 2007.

[5] GROSSO, P., MALER, E., MARSH, J., AND WALSH, N. Xpointer element() scheme, 2003.

[6] GROZA, T., HANDSCHUH, S., MÖLLER, K., AND DECKER, S. Salt - semantically annotated latex for scientific publications. *Lecture Notes in Computer Science 4519* (2007), 518–532.

[7] HARMSZE, F. *A modular structure for scientific articles in an electronic environment (ISBN 90-9013486-7)*. PhD thesis, University of Amsterdam, 2000.

[8] MANN, W., AND THOMPSON, S. Rhetorical structure theory: A theory of text organization. Tech. rep., Information Science Institute, 1987.

[9] SWALES, J. Genre analysis: English in academic and research settings, 1990.

[10] THAO, C., MUNSON, E. V., AND NGUYEN, T. N. Software configuration management for product derivation in software product families. Cambridge University Press, 2008.