Semantic Interrelation of Documents
via an Ontology[*]

Bernd Krieg-Brückner, Arne Lindow, Christoph Lüth, Achim Mahnke, George Russell

Bremen Institute of Safe Systems, FB3 Mathematik und Informatik, Universität Bremen

Abstract: This paper describes how to use an ontology for extensive semantic inter-relation of documents in order to achieve sustainable development, i.e. continuous long-term usability of the contents. The ontology is structured via packages (corresponding to whole documents). Packages are related by import such that semantic interrelation becomes possible not only within a document but also between different documents. Coherence and consistency are enhanced by change management in a repository, including version control and configuration management. Semantic interrelation is realized by particular LaTeX commands for the declaration and definition of classes, objects and relations, and references to them, such that they can be used in standard LaTeX documents, in particular, with a new LaTeX style for educational material (slides, handouts, annotated courses, assignments, and so on).

1 Introduction

Sharing and reuse is the key to efficient development. Unfortunately, while there has been a large body of research concerning the sharing and reuse of program developments, sharing and reuse of documents has until now been mainly done by little more than cut and paste. However, to ensure *sustainable development*, i.e. continuous long-term usability of the contents, sharing and reuse need to be supported by tools and methods taking into account the *semantic* structure of the document. In developing these methods and tools we can benefit from the experience in formal software development and associated support tools.

In this paper, we address this problem by introducing a methodoloy to specify coherence and consistency of documents by interrelation of semantic terms and structural entities, supported by a tool for fine-grained version control and configuration management including change management. Semantic interrelation explicates the meaning lying behind the textual content, and relates the semantic concepts within and across documents by means of an *ontology*. To allow change management, each document is structured in-the-small. Each document corresponds to a package, and packages may be structured in-the-large using folders and import relations.

The ideas and methods explained in this paper have been developed in the MMiSS project [KBHL+03] which aimed at the construction of a multi-media Internet-based adaptive ed-

ucational system. Its content initially covers a curriculum in the area of Safe and Secure Systems, but the ideas are applicable throughout computer science and beyond. Moreover, the approach for semantic interrelation described in this paper applies to all kinds of (LaTeX) documents. As an application area, we will concentrate on the preparation of lecture material, in particular, on slides for presentation in class.

This paper is strucured as follows: we first introduce semantic interrelation of documents by an ontology (Sect. 2), including the LaTeX commands to annotate semantic structure. We then describe structuring in-the-large (Sect. 3), and show how to manage change in our setting (Sect. 4).

2   Semantic Interrelation and Ontologies

Semantic Terms in an Example Lecture.   For a lecture, semantic interrelation is concerned with its topics, i.e. the terminology, and how they are interrelated. As our running example, we will consider an *Introduction to Functional Programming* for undergraduate students, one of more than 20 lectures developed in the MMiSS project. A lecture might start by introducing the difference between imperative and functional programming languages, e.g. HASKELL, and proceed to state that a program consist of type definitions and function definitions. A function definition is given by a (system of) recursive equation(s), and possibly a signature (associating a type to the function). This way, a number of *semantic term*s have been introduced: *function definition*, *signature*, *function*, etc.

There are certain operations on these semantic terms which we implicitly use when writing a text: a semantic term has to be *declared*, is *defined* somewhere in the text, and we can *refer* to it. This constitutes the *semantic interrelation* of a document. A typical lecture will contain a lot of semantic terms; we should impose some structure: an ontology.

Ontologies   provide the means for establishing a semantic structure. An ontology is a formal explicit description of concepts in a domain of discourse [UG96]. Ontologies are becoming increasingly important because they also provide the critical semantic foundations for many rapidly expanding technologies such as software agents, e-commerce, the "semantic web", and knowledge management. An *ontology* consists of (a hierarchy) of concepts and relations between these concepts, describing its various features and attributes.

The semantic terms for our introductory functional programming lecture are arranged in an ontology as shown in Fig. 1. semantic terms for *class*es are ordered by a relation *is a subclass of* (or "*is a*", denoted by a hollow arrow); for example, `FunctionIdH` *is a* `IdH`. Relations between classes may be defined by the user. For example, a `FunctionIdH` *denotes* a `FunctionDefinitionH`, a `FunctionIdH` *calls* a `FunctionIdH`.

Classes and relations are used for defining the abstract semantical level; they provide a vocabulary to denote the concrete entities corresponding to these concepts. Once the abstract notions are declared in terms of classes, objects can be used to identify entities of semantic
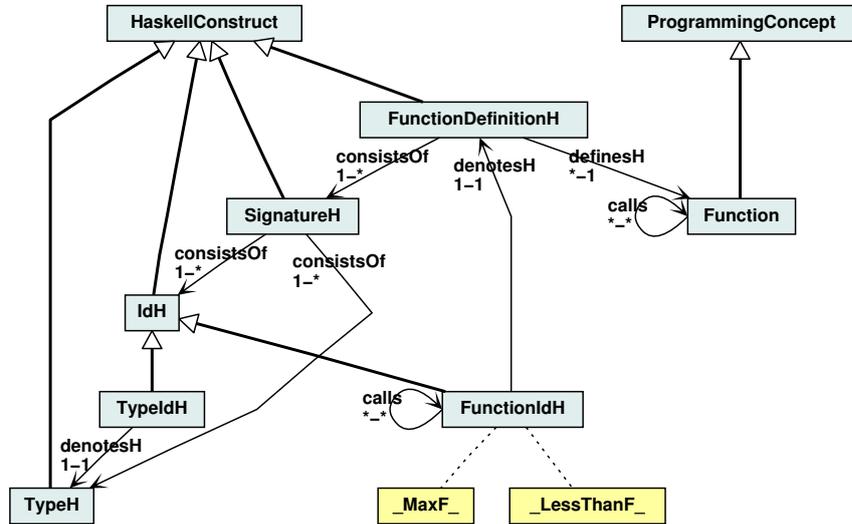
Figure 1: Example Ontology (Extract)

concepts. In our functional programming example, a function *max* could be introduced in the lecture in order to demonstrate different aspects of functional programming concepts. MaxF ("max") and LessThanF ("<") can be viewed as instances of an identifier and therefore be declared as objects of class FunctionIdH[1].

Objects can be semantically related to each other by applying the relations declared in the ontology. Suppose, we have defined a relation *calls*; then "max calls <" on the identifiers can be deduced from the function definitions they denote, and this will induce a corresponding calls relation on the functions they define.

Using the Ontology in the Example Lecture.   Fig. 2 shows an example slide taken from the above mentioned lecture about functional programming. We decided to use LATEX as the authoring format (cf. Sect. 2.1); special LATEX environments are used to structure the content. Fig. 3 shows the corresponding MMISSLATEX source.

The environment Paragraph is used to encapsulate a conceptual unit of text that is to be treated as a whole. Each Paragraph is rendered on a separate slide with its title in the headline. The information is grouped into two Itemize environments rendered as list items with bullet marks. The signature and equation are treated as atoms encapsulated in Code environments. Atoms of a particular Formalism such as HASKELL can be analyzed

---

[1]In Fig. 1, objects are depicted in yellow, and distinguished by leading and trailing underscores.

Figure 2: Example Lecture Slide

and used by other tools (e.g. for extracting all Code fragments belonging to a HASKELL program and feeding them into a compiler).

Each concept introduced on this slide is declared as a semantic term (class) in the lectures ontology. Defining occurrences (Def) are highlighted in red in the presentation format, e.g. for FunctionDefinitionH (in the title), SignatureH, etc. References to one of these concepts will point to the defining occurrence on this slide. Similarly, references to classes defined elsewhere in this lecture or in another imported lecture (e.g. to Recursion in line 19) are optionally presented as hyperlinks (in blue) and/or with section references for paper output (as in the document you are reading now).

The objects MaxF and LessThanF are defined in line 10. They are related to each other by the command \Relate{calls}{MaxF}{LessThanF} in line 11.

2.1   Ontology Commands in MMISSLATEX

MMISSLATEX as an Authoring Format for Ontologies.   The LATEX commands described in this section, and other structuring environments for structuring in-the-small, have been developed as a special LATEX style, called MMISSLATEX, to be converted to an XML representation for the Repository (cf. Sect. 4) and as an exchange format such that other tools processing ontologies may be used (cf. Sect. 2.2). The MMISSLATEX commands for ontologies and semantic interrelation can be used separately, with any LATEX document. The are used to "declare" the ontology of semantic terms used in a document, in a prelude up front. As an example consider Fig. 4, where the example ontology of Fig. 1

```
                        ─── MMISSLATEX ───
1   \begin{Paragraph} [Label=DefFunctions, Formalism=Haskell,
2      Title={\Def[Definition of Functions]{FunctionDefinitionH}}]
3   How to declare and define a \Reference{Function}:
4    \begin{Itemize}
5     \item \Def[Signature]{SignatureH} (optional):
6      \begin{Code}[Label=SigMax]
7        max :: Int -> Int -> Int
8      \end{Code}
9     \item \Def[Equation]{EquationH} (one or many):
10     \Def[]{MaxF} \Def[]{LessThanF}
11     \Relate{calls}{MaxF}{LessThanF}
12     \begin{Code}[Label=EqMax]
13       max x y = if x < y then y else x
14     \end{Code}
15     \begin{Itemize}
16       \item \Def{LhsH}: head with parameters
17       \item \Def{RhsH}: body (usually more than one line)
18       \item Typical pattern: \Reference{CaseDistinction} and
19                             \Reference{Recursion}.
20     \end{Itemize}
21    \end{Itemize}
22  \end{Paragraph}
```

Figure 3: MMISSLATEX Source of the Example Lecture Slide

is presented. This "specification" of the document contains at least a rigorous hierarchical structure of the terminology (a taxonomy, the "signature" of the document), and may be seen as an elaborate index structure. Moreover, relations between terms may be defined for more *semantic* interrelation.

The ontology serves a dual, Janus-headed purpose (just as the specification of an abstract datatype in program development): it specifies the content to be expected in the body of the document in an abstract, yet precise, manner – the content developers requirement specification; and it specifies the content for reference from the outside – the user's perspective, who may then view the body of the document as a black box. Indeed, foreign documents (not prepared in MMISSLATEX) may be encapsulated for the MMISS Repository by providing an ontology in a MMISSLATEX "wrapper".

The content developer will now use the Def command to specify the defining occurrence of a promised term, as for an index. Using the structuring in-the-large facilities via packages (cf. Sect. 3), the external user may then refer between documents using various kinds of reference commands, as the content developer may within a document.

We will now describe the commands in more detail, proceeding from the declarations in the prelude to the definitions and references in the body, for classes, objects and relations.

Declaration of a Class.   A *class declaration* uses the command Class, e.g.

```
                              MMISSLATEX
1   \Relation{1-*}{consistsOf}{consists of}{}
2   \Relation{*-*}{calls}{calls}{}
3   \Relation{*-1}{definesH}{defines}{}
4   \Relation{1-1}{denotesH}{denotes}{}
5   \Class{Function}{function}{ProgrammingConcept}
6       \RelType{calls}{Function}{Function}
7   \Class{FunctionDefinitionH}{function definition}{HaskellConstruct}
8       \RelType{consistsOf}{FunctionDefinitionH}{SignatureH}
9       \RelType{consistsOf}{FunctionDefinitionH}{EquationH}
10      \RelType{definesH}{FunctionDefinitionH}{Function}
11  \Class{SignatureH}{signature}{HaskellConstruct}
12      \RelType{consistsOf}{SignatureH}{IdH}
13      \RelType{consistsOf}{SignatureH}{TypeH}
14  \Class{IdH}{identifier}{HaskellConstruct}
15    \Class{FunctionIdH}{identifier}{IdH}
16      \RelType{denotesH}{FunctionIdH}{FunctionDefinitionH}
17      \RelType{calls}{FunctionIdH}{FunctionIdH}
18    \Class{TypeIdH}{identifier}{IdH}
19      \RelType{denotesH}{TypeIdH}{TypeH}
20  \Class{TypeH}{type}{HaskellConstruct}
21  \Object{MaxF}{max}{FunctionIdH}
22  \Object{LessThanF}{$<$}{FunctionIdH}
```

Figure 4: MMISSLATEX Source of the Example Ontology (Extract)

```
\Class{FunctionIdH}{identifier}{IdH}.
```

The first LaTeX parameter, `FunctionIdH`, denotes the particular semantic term we use in the ontology; the second, `identifier`, the phrase that is to appear in the text by default; the third, `IdH`, the superclass of `FunctionIdH`. The textual effect of a `Class` is nil; however, an entry in the ontology has been made:

- the default phrase, `identifier`, has been associated with the semantic term, `FunctionIdH`, as if a new macro had been defined for `FunctionIdH`, expanding to "identifier";

- a commitment has been made that this semantic term will be defined later on somewhere in the document; and

- the new class, `FunctionIdH`, has been related, as a new subclass, to the existing class `IdH`; it inherits all its properties.

Declaration of an Object.   Analogously, we may declare an object of a given class, e.g.,

```
\Object{MaxF}{max}{FunctionIdH}
```

The effects of this declaration are as for a command `Class`, except that the semantic term, `MaxF`, has been declared as an object of the existing class `FunctionIdH`.

Definition of a Class or Object. The *definition* of a class or object is, e.g.

`\Def{MaxF}` $\rightsquigarrow$ "*max*", `\Def{LessThanF}` $\rightsquigarrow$ "$<$"[2]

An optional parameter could be used to override the default phrase at this position, as in References below.

References to a Class or Object. Various alternatives for *reference*s are possible, e.g.

`\MaxF{}` $\rightsquigarrow$ "max", `\LessThanF{}` $\rightsquigarrow$ "$<$";

`\Ref{MaxF}` $\rightsquigarrow$ "max", `\Ref{LessThanF}` $\rightsquigarrow$ "$<$"; and

`\Reference{MaxF}` $\rightsquigarrow$ "max (p. 7)", `\Reference{LessThanF}` $\rightsquigarrow$ "$<$ (p. 7)".

The first alternative denotes the easiest, most "light-weight" and probably most common usage: the semantic term is used as if it was the name of a parameterless macro-like LATEX command. Since in many cases the identifier of the semantic term coincides with the corresponding phrase (apart from capitalisation), it provides the lowest clutter when reading the LATEX source. For `Ref` and `Reference`, an optional parameter may be given as for `Def`. Thus `\Ref{LessThanF}` is equivalent to `\LessThanF{}`, except for the possibility of an alternative text, e.g.

`\Ref[greater than]{LessThanF}` $\rightsquigarrow$ "greater than".

The third alternative, `Reference`, optionally generates a reference to a page number in a paper presentation (such as this one), in addition to a hyperlink.

Declaration of a Relation. The declaration of a relation is analogous to that of a class:

`\Relation{*-*}{calls}{calls}{}.`

Here there is an extra, the first, parameter that denotes the standard properties of the relation, in this particular case, an arbitrary relation with the standard property denoted by `*-*`[3]. As for `Class`, the second parameter, `calls`, denotes the particular semantic term we use for the relation in the ontology; the third parameter the phrase that should appear in the text by default, `calls`; the last the superrelation of `calls`, empty here.

Declaration of the Type of a Relation. The *type* of a relation may be declared:

`\RelType{calls}{FunctionIdH}{FunctionIdH}`,

`\RelType{calls}{Function}{Function}`.

The second parameter provides the domain, the third the range class of the relation.

The reason to have a separate declaration for the type of a relation is that it may be invoked several times, for different domain and range classes; thus the applicability of a relation can be declared as specifically as desired — and will be checked as specifically as possible.

---

[2] We use the symbol "$\rightsquigarrow$" to denote "yields the formatted text", which then follows in quotes.

[3] We will allow pre-defined properties, such as order (cf. Fig. 6), or user-defined properties in the near future.

In the example above, `calls` may relate syntactic or semantic objects, resp. Moreover, a relation type is inherited along the subclass relations of the domain and range.

The definition of a relation is analogous to the definition of classes or objects.

Relating Objects.   Objects may be related by the command `Relate`, e.g.

`\Relate{calls}{MaxF}{LessThanF}` ⤳ "''

This way, the relation is established between the pair of objects, constituting a definition in the ontology. `Relate` yields no text, thus it is comparable to a declaration. However, it may appear anywhere in a text (like other definitions) and is checked by specialized tools against the ontology. For example, the type of the relation is checked against the classes that the objects belong to, taking inheritance into account.

References to a Relation.   In contrast to references to other semantic terms, which are "constants" and have an empty parameter, a relation has two parameters, thus

`\calls{\MaxF{} }{ \LessThanF}` ⤳ "max calls $<$", as an alternative to

`\MaxF{} \Ref{calls} \LessThanF{}` ⤳ "max calls $<$"

As for classes and objects, it is possible to reference a relation with the `Ref` command in order to use an alternative textual phrase.

2.2   Benefits of Ontologies for Document Management

The notion of *ontology* in computer science stems from the artificial intelligence research area of knowledge representation. The general idea is to make knowledge explicit by expressing concepts and their relationships formally with the help of mathematical logics. Ontologies play a crucial role in the development of the Semantic Web, which aims at leveraging the WWW to a level on which computers can deal with the information which is – at the moment – encoded informally in the huge amount of web content ([BLHL01]).

Much like the Semantic Web idea, the semantics weaved into the textual contents of documents will enable much more advanced document management facilities. We will explain several benefits here, and more concerning change managment in Sect. 4.2.

Resolution of Ambiguities.   One simple example illustrating the gains we get from using ontologies is connected with the difficulties we often encounter when we want to reuse or share material:

- the exact meaning of a concept is unclear;

- different terms are used for the same semantic concept;

- the same term is used for different semantic concepts.

For example, consider the heavily overloaded term "process" in computer science (the process of software development, slightly different notions of parallel process, etc.). While a human user can often discriminate from the context, a tool must have unambiguous information: we would certainly expect a hyperlink to lead to the correct target definition (cf. also Sect. 3). As a more subtle example, consider the example ontology in Fig. 1 and its MMiSSLaTeX-declaration in Fig. 4. The phrase "identifier" is used for different concepts, distinguished by the terms `FunctionIdH`, `TypeIdH` (and the superclass `IdH`); it is often the case that one prefers light-weight default phrases over heavy phrases such as "function identifier", e.g. the `\FunctionIdH{} \MaxF{}` ↝ "the identifier max".

Ontology-Based Search.   Semantic mark-up and interrelation also allows more powerful searching in documents, because indexing over the semantic terms is much more accurate than textual search over possibly ambiguous phrases. A user looking for specific material can not only be provided with more adequate terms but is also able to explore properties of concepts in order to understand relations and differences between them. The ontology graph is a powerful information source for navigation, in particular when provided with tools for selective graphical presentation (under development).

Tools for Ontologies.   Tools are available to display (part of) an ontology in a graphic way (cf. Fig. 1). Moreover, some tools have already been implemented to feed the ontology of a document into ontology-related tools based on the emerging OWL [BvHH+04] standard for the Semantic Web. The connection to ontology tools opens up the possibility for efficient reasoning about ontologies developed in MMiSSLaTeX in the Description Logic of OWL. The precise formal definition of ontologies (e.g. in a First-Order Logic framework such as CASL [ABKB+02, Mo04, LMKB]) provides considerable potential for the use of formal methods and tools for proving more complex properties, e.g. with [CAT].

3   Structuring in-the-Large via Packages

Packages provide a means for modular document development by introducing *name spaces*. When writing a document, authors introduce identifiers as labels for structural entities or as semantic terms in an ontology. If these identifiers, subsumed as *names* in the sequel, are defined more than once, we say there is a *name clash*.

A *package* is the largest structural entity. A package is a document that corresponds to a whole course or book and contains all structural entities pertaining to it. A Package encapsulates the name space of a document, such that names defined in a Package do not clash with names from other packages. In order to use names from other packages, these have to be imported explicitly (see below). In other words, packages are very much like modules in programming languages such as Modula-2, Haskell, or Java. As a general rule, name clashes are resolved on import by renaming symbols, hiding, restricting or qualified import (or a combination of these), rather than by restricting the export.

A Package contains a prelude that contains "global declarations" for it. The prelude consists of the *ontology prelude* and the *import prelude*.

The *OntologyPrelude* declares elements of an ontology (cf. Sect. 2). It acts like a signature of the Package for semantic interrelation, promising these elements to be defined in this Package, such that they become available when imported by another.

The *ImportPrelude* specifies other packages to be imported. For each package to be imported, there are one or more *import directives* specifying the modalities of the import:

- *Qualified vs. Unqualified Import*: After importing a package, we can use the semantic terms directly (unqualified). Structural entities are, by default, imported *qualified*, i.e. we have to prefix the name with the (possibly aliased) package name; if we import *unqualified*, we can use them as they are. The advantage of qualified import here is that no name clashes may occur.

- *Local vs. Global Import*: A *global import* of a package is one where the imported package is reexported automatically. A *local import* is one where the package is not reexported automatically; this is the default behaviour.

- *Hiding, Revealing and Renaming*: In an import statement, we can hide, reveal or rename the imported names:

  - *Hiding* a name when importing means that it is not imported. Other names from that package are imported as usual.

  - *Revealing* means that *only* the particular names mentioned are imported, and no other names from that package.

  - *Renaming* means that an entity denoted by a name in the defining package is imported under a different name.

Resolution of Name Clashes. The problem stated in Sect. 2.2, that the same term is used for different semantic concepts, may not only occur for phrases but also for the (technical) semantic terms themselves. When importing from two different packages, two terms may well have the same name. Consider the following example:

```
MMiSSLaTeX
1  \Import[Rename{ParallelProcess=Process}]{ParallelProcesses}
2  \Import[Rename{DevelopmentProcess=Process}]{ProgramDevelopment}
```

After renaming, `ParallelProcess` and `DevelopmentProcess` coexist peacefully. If the same semantic concept has been introduced in two different packages (under the same or different names), we can hide one of them upon import.

Since LaTeX itself does not have a sophisticated handling of name spaces or import and export, we implemented the package mechanism externally (i.e. not in LaTeX) on the level of the *repository*, as we will describe in the next section.
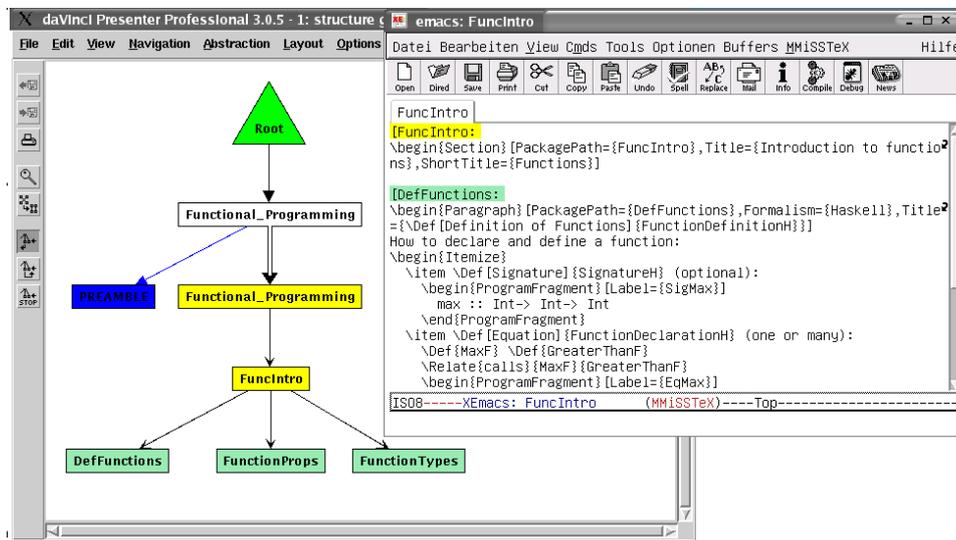
4 Change Management in the Repository

Figure 5: The MMiSS repository at work.

4.1 Fine-Grained Version Control

The *repository* provides version control and configuration management for our documents. Version-control is *fine-grained*, i.e. documents are not versioned as a whole, but broken down into their constituting structural entities and versioned on that level. Objects in the repository are organised in folders, which allow the grouping of repository objects much like directories in a file system. The objects in the repository form a graph, with structural entities such as sections, units or atoms as nodes, and the relation as edges, in particular the relation comprises (p. 12). This constitutes the *structure graph* of all objects, which is the user's view of the repository. Fig. 5 shows the user interface: the structure graph, here our example slide from Sect. 2, is displayed with the daVinci graph visualisation system [daV], which allows the user to navigate in the graph. At each node, users may select operations such as editing or calling a tool to typeset, compile or check the corresponding part of a document. For editing documents, the Emacs editor[4] is opened with the MMiSSLATEX source of this structural entity; users may edit and typeset it, and then commit it back. This is a special case of the check out-commit cycle known from other versioning tools such as CVS [CVS]. The editing context is usually limited to a unit such as a paragraph — inner or outer entities may incrementally be added to the editing screen using the buttons named

---

[4]The Emacs editor has been adapted to the system; other editors and LATEX systems may be used externally.

by the entities, such as `List2` for the nested sublist.

As has been mentioned in Sect. 3, LATEX itself has neither name spaces nor a sophisticated export/import mechanism. Hence, the package mechanism is implemented on the level of the repository. When we check out a package from the repository, a MMISSLATEX document is generated, and the import prelude is supplemented by MMISSLATEX code from imported packages. The import clauses itself, and the generated MMISSLATEX ontology prelude, are contained in the document, and marked by comments (pragmas). When a package is commited back to the repository, the generated prelude is discarded, and the import statements are used to generate a new prelude. This can also be done interactively, if users want to add import statements to their import prelude.

The repository is *distributed*; more than one repository may be running, and documents or parts of documents can be moved between them.

4.2 Change Management

The notion of *change management* is used for the maintenance and preservation of consistency and completeness of a development during its evolution. More precisely, we want to have a *consistent configuration* in which all constituents harmonise, versions are compatible, references and links refer to the proper targets, etc. At the same time, it should be a *complete configuration*: e.g. the promises of (forward) References and Links should be fulfilled, i.e. they must not be dangling.

Such notions are well-known for formal languages; in contrast, natural language used for writing teaching material does not usually possess a well-defined semantics; the notion of consistency is arguable. Different authors may postulate different requirements on the material in order to regard it as being consistent. The existence of a user-defined ontology already helps a great deal to check references.

The Systems Ontology.   As an example, consider an extract from the MMISS systems ontology, which may be extended by the user. For structuring in-the-small, (a hierarchy of) structural entities are defined, and relations between them. Let us first look at some relations, where each relation inherits the properties of its super-relation (see Fig. 6, line 1–9). Now consider some of the properties that may be defined:

comprises   An obvious structuring mechanism is nesting of individual parts of a document, leading to the *contains* relation (see Fig. 6, line 10–14): a package contains sections, a section contains another section, a section contains a Theory, etc. The contains relation is part of a family of *comprises* relations that share common properties.

reliesOn   Besides the comprises relation, there is a family of *relies on* relations, reflecting the various dependencies between different parts of a document. For example, a theorem *lives in* a theory, a proof *proves* a theorem, and so on (see Fig. 6, line 15–16).

```
                                    ─── MMiSSLaTeX ───
1   \Relation{*-*}{comprises}{comprises}{relatesDocConstructs}
2     \Relation{<-}{contains}{contains}{comprises}
3   \Relation{>}{reliesOn}{reliesOn}{relatesDocConstructs}
4     \Relation{}{imports}{imports}{reliesOn}
5     \Relation{}{livesIn}{lives in}{reliesOn}
6     \Relation{}{proves}{proves}{reliesOn}
7   \Relation{->}{pointsTo}{pointsTo}{relatesDocConstructs}
8     \Relation{}{references}{references}{pointsTo}
9
10      \RelType{contains}{PackageSE}{SectionSE}
11      \RelType{contains}{SectionSE}{SectionSE}
12      \RelType{contains}{SectionSE}{TheorySE}
13      \RelType{contains}{SectionSE}{TheoremSE}
14      \RelType{contains}{SectionSE}{ProofSE}
15      \RelType{livesIn}{TheoremSE}{TheorySE}
16      \RelType{proves}{ProofSE}{TheoremSE}
```

Figure 6: MMiSSLaTeX System's Ontology (Simplified Extract)

pointsTo The family of *points to* relations is very similar: e.g. a references defining occurrence of a semantic term.

variantOf Another structuring relation is introduced by the various notions of variants. Parts of a document may e.g. be written in various languages which gives rise to a *variant of* relation between these document parts and their constituents; it is an equivalence relation.

The change management keeps track of the various structuring mechanisms. Postulating invariant properties as requirements on the consistency and completeness of a document, and formulating these invariants as formal rules, will enable us to implement a generic and flexible change management that keeps track of the various invariants and informs the user about violations when a previously consistent document has been revised.

Properties of Individual Structuring Mechanisms. For each of the structuring mechanism described above we can formulate various invariants. Some of these are enforced by the underlying structuring language (MMiSSLaTeX), but others may be violated once the user revises a document.

Relations in the relies on family are partial orders; points to relations are many-to-one. A consistency requirement is that there is at most one target; a completeness requirement is that there is at least one target, e.g. references must not be dangling; both together require a unique target. Furthermore, for relies on relations, we require the target to be presented before. However, the ordering requirement is weaker for points to relations as we tolerate forward pointers, even to other, future packages (warnings should be given, though).

Properties of Interactions between Structuring Mechanisms. Relating the comprises and relies on relations, for example, allows us to formalize constraints regarding the closure of document parts with respect to the relies on relation. We may e.g. require that there is a proof for each theorem in a package. Furthermore, a relies on relation between two structural entities is propagated along the comprises relation towards the root of the hierarchy of nested structural entities, e.g. (for a theorem T, a proof P, and sections A, B):

B contains P and A contains T and P proves T, then B relies on A.

In any case, the more structure there is, the better are the chances for preserving consistency and completeness; any investment in introducing more relies on relations, for example, will pay off eventually. The change management will observe whether revisions by the user will affect these relations and, depending on the user's preferences, emit corresponding warnings. It is crucial to point out that, in contrast to formal developments such as in the MAYA-system [AHMS02], there is no rigorous requirement that a document should obey all the rules mentioned above. There may be good reasons, for instance, to present first a "light-weight" introduction to all notions introduced in a section before giving the detailed definitions. In this particular case, one would want to introduce forward pointers to the definitions rather than making the definitions rely on the introduction; thus the rules are covered.

The eventual aim is to allow the users to specify individual notions of consistency by formulating the rules that the relations should obey. This should be possible for the relations between the various structuring mechanisms, but also between semantic terms of the user's ontology in general.

5 Conclusion

We have presented some contributions to the important issue of *sustainable development* and management of documents, notably *semantic interrelation* via a user-defined ontology. The MMISSLATEX style is available, tools for structuring in-the-small and fine-grained version control have been implemented. Structuring in-the-large via a hierarchy of packages also allows import of document-related ontologies and thus supports the *structuring of ontologies* themselves. Similarly, the basic infrastructure for change management of documents (and ontologies as a by-product) is available; the ideas about a formal definition of *invariant properties* of relations are presently being implemented.

A considerable amount of teaching material has been produced and actively used in academic teaching; an explicit procedure for evaluation has been set up and is currently being implemented.

MMISSForum. As an open source model is used, teaching content and tools are freely available to achieve a wider national and international take-up. To assist this, a MMISS Forum has been set up to evaluate the emerging curriculum and assist its development and distribution; you are welcome to join ([MMi]).

References

[ABKB$^+$02] Astesiano, E., Bidoit, M., Krieg-Brückner, B., Kirchner, H., Mosses, P. D., Sannella, D., und Tarlecki, A.: CASL – the common algebraic specification language. *Theoretical Computer Science*. 286:153–196. 2002. www.cofi.info.

[AHMS02] Autexier, S., Hutter, D., Mossakowski, T., und Schairer, A.: The development graph manager MAYA (system description). In: Kirchner, H. und Reingeissen, C. (Hrsg.), *Algebraic Methodology and Software Technology, 2002*. volume 2422 of *Lecture Notes in Computer Science*. S. 495–502. Springer. 2002.

[BLHL01] Berners-Lee, T., Hendler, J., und Lassila, O.: The Semantic Web. *Scientific American*. May 2001.

[BvHH$^+$04] Bechhofer, S., van Hermelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., und Stein, L. A. W3C: OWL Web Ontology Language – Reference. http://www.w3.org/TR/owl-ref/. February 2004.

[CAT] CASL Tool Set web site. http://www.informatik.uni-bremen.de/cofi/CASL/CATS/index.html.

[CVS] http://www.cvshome.org.

[daV] http://www.b-novative.com/products/daVinci/.

[KBHL$^+$03] Krieg-Brückner, B., Hutter, D., Lindow, A., Lüth, C., Mahnke, A., Melis, E., Meier, P., Poetzsch-Heffter, A., Roggenbach, M., Russell, G., Smaus, J.-G., und Wirsing, M.: Multimedia instruction in safe and secure systems. In: *Recent Trends in Algebraic Development Techniques*. volume 2755 of *Lecture Notes in Computer Science*. S. 82–117. Springer. 2003.

[LMKB] Lüttich, K., Mossakowski, T., und Krieg-Brückner, B.: Developing ontologies for the semantic web with CASL. *17th Int. Workshop on Algebraic Development Techniques*. To appear.

[MMi] http://www.mmiss.de.

[Mo04] Mosses (ed.), P. D.: CASL *Reference Manual*. volume 2960 of *Lecture Notes in Computer Science*. Springer. 2004.

[UG96] Uschold, M. und Grüninger, M.: Ontologies: Principles, methods and applications. *Knowledge Engineering Review*. 11(2):93–155. 1996.