Semantic Interrelation of Documents via an Ontology*

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

Bremen Institute for Safe and Secure Systems, Universität Bremen

DRAFT

Abstract. This tutorial describes the rationale for and use of an ontology for extensive semantic interrelation of documents to increase "sustainable development", i.e. continuous long-term usability of the contents. The ontology is structured via Packages (corresponding to whole documents); these may be imported 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.
The LaTeX commands for the declaration and definition of classes, objects and relations, and References to them, are described in detail, such that these can be used in standard LaTeX documents. Moreover, structuring via Packages is introduced and the benefits of change management in a Repository are explained.

1 Introduction

This document is a draft and not complete.
A Quick Reference to the structural entities is included as the end.

2 Semantic Interrelation

2.1 Simple Semantic Interrelation

The *semantic interrelation* of documents in its simplest form can be compared to the marking of a particular position in a document by a label; it provides for referencing and macro-like abbreviation as well. Moreover, it will be used in the future for establishing an index[1].

A semantic term denotes an identifier that combines the function of a LaTeX label and a macro for a phrase, resp.; since it may be related to other semantic

* The MMiSS project has been supported by the German Ministry for Research and Education, bmb+f, in its programme "New Media in Education".

[1] not implemented yet.

terms via an ontology (p. 4), a "semantic net", the semantic connotation is justified.

We will proceed by first describing the function of a Declaration of a semantic term in the prelude of a document, then the Definition of it somewhere in the body (the "defining occurrence", corresponding to the major entry in an index), and finally means of referring to this Definition by various kinds of References (the "applied occurrences", corresponding to minor entries in an index).

Simple Declaration of a Semantic Term. Consider the simple *Declaration* of a semantic term using the LaTeX command `Declare`, e.g.

`\Declare{XMLDocTypeDef}{document type definition}`.

The first parameter, `XMLDocTypeDef`, denotes the particular *semantic term* we want to declare; the second, `document type definition`, the (textual) *phrase* that should appear in the text by default when we later refer to this semantic term (see below). The textual effect of a `Declare` is nil; however,

- the default phrase, `document type definition`, has been associated with the semantic term, `XMLDocTypeDef`, as if a new macro had been defined for `XMLDocTypeDef`, expanding to "document type definition"; and
- a commitment has been made that this semantic term will be defined later on somewhere in the document.

A `Declare` may only appear in the so-called OntologyPrelude (p. 11) before the proper beginning of a document. The new name, `XMLDocTypeDef`, must not be declared elsewhere, as a semantic term, a LaTeX command, or otherwise. To avoid name clashes with LaTeX commands, we adopt the convention that the name representing a semantic term starts with a capital letter.[2]

Definition of a Semantic Term. The simple *Definition* of a semantic term uses the LaTeX command `Def`, e.g.

`\Def{XMLDocTypeDef}` $\rightsquigarrow$ "*document type definition*"[3]

`Def` marks the position by a LaTeX label `XMLDocTypeDef`. Note that the previously declared phrase, `document type definition`, is delivered and emphasized. Alternatively, an optional parameter may be given instead of the default phrase, e.g.

`\Def[\XML{} document type definition]{XMLDocTypeDef}` $\rightsquigarrow$
"XML *document type definition*"

Warning. LaTeX commands with optional parameters must not be nested, e.g.
`\Def[\Ref[XML-]{XML}document type definition]{XMLDocTypeDef}`
would be illegal and lead to strange behaviour by the LaTeX system.

[2] Following LaTeX conventions, such a name may only contain letters.
[3] We use the symbol "$\rightsquigarrow$ " to denote "yields the formatted text", which then follows in quotes.

References to a Semantic Term. Various alternatives for *Reference*s are possible, e.g.

1. `\XMLDocTypeDef{}` ⇝ "document type definition";
2. `\Ref{XMLDocTypeDef}` ⇝ "document type definition"; and
3. `\Reference{XMLDocTypeDef}` ⇝ "document type definition (p. 2)".

The first alternative denotes the easiest and probably most common usage: the semantic term is used as if it was the name of a parameterless macro-like LaTeX command; we call this a *macro reference*. 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. Moreover, it does provide the opportunity of macro-abbreviation, as in the example above; consider also

`\Declare{JAVA}{\GivenName{Java}}`

where `JAVA` is formatted using a macro `GivenName`, such that `\Ref{JAVA}` yields "Java". If the formatting of `JAVA` should be changed, then it suffices to make this change only once in the phrase of the Declaration (or, in this case, `GivenName`).

The possibility of declaring an extra phrase will also come in useful when translating the document into a different natural language: one would first translate the Declarations of semantic terms by giving a different phrase for each semantic term — the text then has a framework of the most important terms already translated; after the complete translation, the source will look a bit funny since the semantic terms will still be in the original language, but this is a reasonable comcommitment for standardisation of terminology across languages (the semantic terms may also be renamed in the ImportPrelude (p. 12), if desired).

For `Ref` and `Reference`, an optional parameter may be given as for `Def`. Thus `\Ref{XMLDocTypeDef}` is equivalent to `\XMLDocTypeDef{}`, except for this possibility of an alternative text, e.g.

`\Ref[\XML{} document type definition]{XMLDocTypeDef}` ⇝

"XML document type definition"

The third alternative, `Reference`, also generates a reference to a page number[4] as well as the hyperlink[5].

A Note on Empty Parameters and Braces. We write a LaTeX command with no parameters (a 0-ary command) with an empty parameter, e.g.

`\XML{}` instead of `\XML`.

in many cases the formatting effect is the same, but there are some situations (often strange to the novice) where the second notation "swallows" a following

[4] The complete package (rather than ontology.sty) will generate this page reference only for the LayoutAttribute value Paper

[5] References only work within a document so far; work is under way to allow references to other documents as well. The first two alternatives (i.e. the macro reference expansion) will also work for semantic terms imported from other Packages.

character (or the like);[6] thus the first notation is safer. It always delimites the LaTeX command from whatever follows it, cf. e.g.

`\SemanticTerm{}s` instead of `\SemanticTerms`

where the plural "s" has been mistakenly added, leading to a different LaTeX command name `SemanticTerms` instead of `SemanticTerm`.

If a LaTeX command application is nested another, i.e. in braces or square brackets, e.g. `\Ref[\XML{}]{XMLDocTypeDef}` or `{\XML{}}`, then the extra braces are superfluous, i.e. `\Ref[\XML]{XMLDocTypeDef}` or `{\XML}` are actually sufficient. However, we use the former notation for uniformity. An extra pair of braces (as an "empty parameter" or around a term) is always permitted and harmless.

2.2 Semantic Interrelation via 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. Ontologies are becoming increasingly important because they also provide the critical semantic foundations for many rapidly expanding technologies such as software agents, e-commerce and knowledge management. An ontology consists of concepts and relations between these concepts. Properties of a concept are specified by describing its various features and attributes. For the diagrammatic representation of an ontology, we use a subset of the modeling language UML [**?**], which is an actual de facto standard language for software development.

Example: An Ontology of Users Before we explore the variety of LaTeX commands in the sequel, let us consider a little example of an ontology in Fig. 1 (p. 5).

The example shows an ontology of potential *user*s of an authoring system and e-learning environment and their *role*s, resp. A *professor*, as an *academic user*, may assume the role of a *teacher*, thereby only having reading access to the material, or of an *author* with a particular kind of writing access. This ontology is of course much simplified (there are also developer and administrator roles, etc.). It shows, however, the general principles:

- a taxonomic hierarchy of *class*es, ordered by the *is a subclass of* relation (denoted by the fat arrow with a triangular head), e.g. "professor is a subclass of academic user", reading "a professor is an academic user", with the meaning "the class academic user subsumes the class professor" (all objects of professor are also objects of academic user; the properties of academic user are inherited to professor);
- *object*s ("individuals") of these classes (a dashed arrow denotes the *is an object of* relation, not shown here); and

[6] The ugly trick of introducing spaces in the definition of commands, used by some authors, is avoided here.

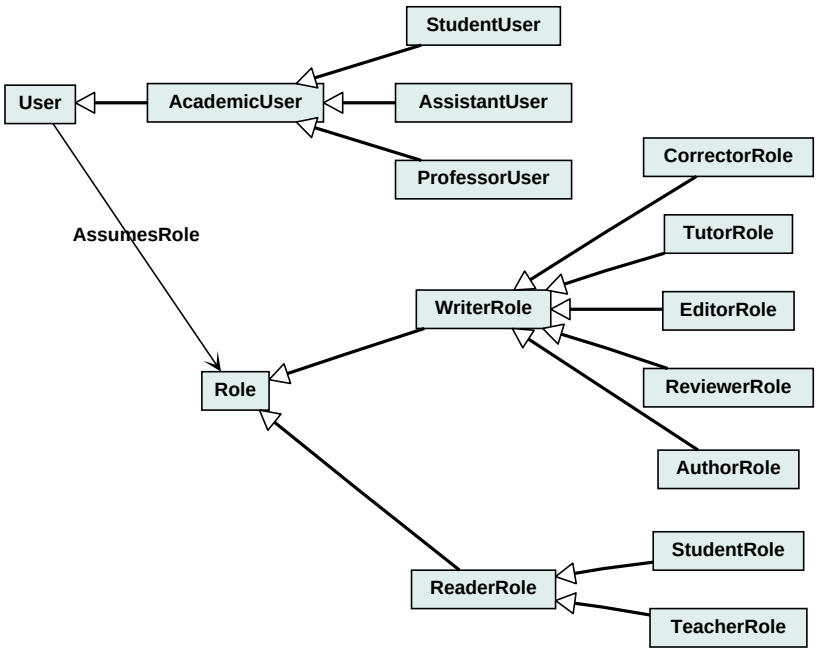

Fig. 1. Ontology of Users and Roles.

– a (hierarchy of) *relations*[7], declared between the classes and applied to objects, e.g. userX assumes the role roleY.

Example: Declaration of an Ontology Recall Fig. 1 (p. 5), the example of an ontology of users and roles. Such an ontology would be declared in the so-called OntologyPrelude (p. 11) using the Ontology Style (p. **??**), the LaTeX extension to represent the ontology commands, as follows (partially shown here):

```
\Class{User}{user}{}
 \Class{AcademicUser}{academic user}{User}
  \Class{ProfessorUser}{professor}{AcademicUser}
  \Class{AssistantUser}{assistant}{AcademicUser}
  \Class{StudentUser}{student}{AcademicUser}
\Class{Role}{role}{}
 \Class{ReaderRole}{reader}{Role}
  \Class{TeacherRole}{teacher}{ReaderRole}
  \Class{StudentRole}{student}{ReaderRole}
 \Class{WriterRole}{writer}{Role}
```

[7] called associations in UML

```
   \Class{AuthorRole}{author}{WriterRole}
   \Class{CorrectorRole}{corrector}{WriterRole}
\Relation{*-*}{assumesRole}{assumes the role}{}
\RelType{assumesRole}{User}{Role}
```

Declaration of a Class. Consider the Declaration of a class using the command `Class`, e.g.

`\Class{StudentRole}{student}{ReaderRole}`.

As for `Declare`, the first LaTeX parameter, `StudentRole`, denotes the particular semantic term we use in the ontology; the second, `student`, the phrase that should appear in the text by default; the third, `ReaderRole`, the superclass of `StudentRole`. The textual effect of a `Class` is nil; however, an entry in the ontology has been made:

- the default phrase, `student`, has been associated with the semantic term, `StudentRole`, as if a new macro had been defined for `StudentRole`, expanding to "student";
- a commitment has been made that this semantic term will be defined later on somewhere in the document; and
- the new class, `StudentRole`, has been related, as a new subclass, to the existing class `ReaderRole` (i.e. "`StudentRole subclassOf ReaderRole`" holds), it inherits all its properties.

A `Class` may only appear in the OntologyPrelude (p. 11). The new identifier, `StudentRole`, must not be declared elsewhere, as a semantic term, a LaTeX command or otherwise.

Subclass Hierarchy. In most cases, the hierarchy of the `subclassOf` relation is strict, i.e. a class can only be a direct `subclassOf` one other class (if `A subclassOf B` and `A subclassOf C` then `B subclassOf C`). This restriction holds especially for the systems ontology (cf. Sect. 2.5 (p. 10)). In general, however, a class may be a direct `subclassOf` more than one class, i.e. may inherit from several other classes. In this case, the same semantic term appears in more than one class Declaration [8]. The associated phrase should then be the same.

Resolution of Ambiguities. There are at least three reasons for having a semantic term as an an extra parameter of a Declaration :

- the semantic term (the first parameter in a Declaration) may be renamed upon Import from another package to avoid name clashes while the phrase (the second parameter) remains the same,
- the phrase may be translated into a different language variant of the ontology, assuming that the semantic term remains the same for uniformity of language variants,

[8] The LaTeX system will issue a warning message that the same LaTeX label has been declared twice.

– apparent ambiguities regarding same phrases may be resolved by having two different semantic terms with the same phrase .

To illustrate the ambiguity issue consider the following example and its source:

a student assumes the role of a student
`a \StudentUser{} \assumesRole{}{} of a \StudentRole{}`

An apparent ambiguity (which is usually resolved by context in natural language) is resolved since there are two different semantic terms in the example ontology. Note that a hyper-reference references the appropriate Definition correctly.

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

`\Object{ProfBKB}{Prof. B. Krieg-Br\"uckner}{ProfessorUser}`, or
`\Object{LecturerWADT}{lecturer at WADT}{TeacherRole}`.

As for `Class`, the first parameter, `BKB`, denotes the particular semantic term we use in the ontology, in this case an identifier denoting an object of the class `ProfessorUser` (third parameter); the second parameter denotes the phrase that should appear in the text by default, `Prof. B. Krieg-Br\"uckner`.

The other effects are as for a command `Class`, except that

– the semantic term, `ProfBKB`, has been declared as an object of the existing class `ProfessorUser` (i.e. "`ProfBKB objectOf ProfessorUser`" holds).

Definition of a Class or an Object. The Definition of a class or an object is as described in Sect. Definition of a Semantic Term (p. 2), e.g.

`\Def{ProfBKB}` ⇝ "*Prof. B. Krieg-Brückner*"
`\Def[lecturer at the WADT Workshop]{LecturerWADT}` ⇝
"*lecturer at the WADT Workshop*"

References to a Class or an Object. Similarly, References to a class or an object are as described in Sect. References to a Semantic Term (p. 3), e.g.

`\ProfBKB{}` ⇝ "Prof. B. Krieg-Brückner"
`\Ref{ProfBKB}` ⇝ "Prof. B. Krieg-Brückner"
`\Reference[BKB]{ProfBKB}` ⇝ "BKB (p. 7)"

Declaration of a Relation. Again, the Declaration of a relation is analogous to the Declaration of a semantic term in Sect. Declaration of a Class (p. 6), using the command `Relation`, e.g.

`\Relation{*-*}{assumesRole}{assumes the role}{}`.

Here there is an extra, the first, parameter that denotes the standard properties (p. 8) of the relation, in this particular case, an arbitrary relation with the standard property denoted by *-*. As for `Class`, the second parameter, `assumesRole`, 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, `assumes role`; the last the superrelation of `assumesRole`, empty here.

The other effects are as for a command `Class`; the entry in the ontology is as follows:

- the default phrase, `assumes the role`, has been associated with the semantic term, `assumesRole`, as if a new macro had been defined for `assumesRole`, see Sect. References to a Relation (p. 9);
- a commitment has been made that this semantic term will be defined later on somewhere in the document;
- the new relation, `assumesRole`, has been related, as a new subrelation, to the relation given as the last parameter, if any (empty here) — it inherits all its properties; and
- the new relation has the standard property (p. 8) given in the first parameter, in addition to the inherited properties; if it is empty, no (additional) properties are declared, i.e., if it has no superrelation, it is an arbitrary relation by default, corresponding to `*-*`.

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

`\RelType{assumesRole}{User}{Role}`.

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.

Moreover, a relation type is inherited along the subclass relations of the domain and range; e.g.

`\RelType{assumesRole}{ProfessorUser}{TeacherRole}`.

is already implied.

Standard Properties of a Relation. *Standard properties* of relations and their denotations are given by the following table:

`*-*` denotes an arbitrary relation,
`>` denotes a strict order relation,
`>=` denotes a partial order relation,
`=` denotes an equivalence relation,
`->` denotes an onto relation,
`<->` denotes a one-to-one relation,

(to be continued);
similarly for the inverses.

Declaration of the Inverse of a Relation. The *Inverse* of a relation may be declared as follows:

`\InverseRel{superclassOf}{is a superclass of}{subclassOf}`.

As for `Relation`, the first parameter, `superclassOf`, denotes the particular semantic term we use for the inverse relation in the ontology; the second parameter, `is a superclass of`, the phrase that should appear in the text by default; the last parameter, `subclassOf`, the relation it is an inverse relation of. The type or standard property need not be declared as they are the inverses, resp., by definition.

Definition of a Relation. The Definition of a relation is as described in Sect. Definition of a Semantic Term (p. 2), e.g.

`\Def{assumesRole}` ⇝ "*assumes the role*"

`\Def[assumes the role of a]{assumesRole}` ⇝

"*assumes the role of a*"

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

`\Relate{assumesRole}{ProfBKB}{LecturerWADT}` ⇝ ""

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[9]. 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 macro references (p. 3) to other semantic terms, which are "constants" and have an empty parameter, a relation has two parameters, thus

`\assumesRole{\ProfBKB{} }{ of a \LecturerWADT{}}`[10] ⇝

"Prof. B. Krieg-Brückner assumes the role of a lecturer at WADT", or

`a \StudentUser{} \assumesRole{}{} of a \StudentRole{}`, alternatively

`a \StudentUser{} \Ref{assumesRole} of a \StudentRole{}`, ⇝

"a student assumes the role of a student".

Note that in the case of `\Ref{assumesRole}`, the semantic term refers to its default phrase only, it is not a new macro with two parameters, thus no two extra empty parameters are needed. This facilitates its use in the case of optional alternative texts, as e.g. in

`\Ref[may assume the role of a]{assumesRole}`, ⇝

"may assume the role of a", similarly for

`the relation \Reference[assumesRole]{assumesRole}`, ⇝

"the relation assumesRole (p. 9)".

[9] These tools are not operational yet.

[10] Since the relation is used in infix notation here, one needs to introduce blanks on either side unless brackets are used in larger terms.

2.3 Structure Links

Links *(present implementation).* *Link*s point to structural entities, in a way analogous to References (p. 3), e.g.

`Sect.~\Link{SimpleDefinition}` ⇝ "Sect. 2 (p. 2)", or
`Sect.~\Link[Declaration of a Class]{ClassDeclaration}` ⇝
"Sect. Declaration of a Class (p. 6)"

Links *(future).* *Link*s point to structural entities, in a way analogous to References (p. 3), e.g.

`\Link{Figure}{FigUsersOntology}` ⇝ "Fig. 1 (p. 5)",
`\Link{Section}{SimpleDefinition}` ⇝ "Sect. 2 (p. 2)", or
`\Link[Declaration of a Class]{Section}{ClassDeclaration}` ⇝
"Sect. Declaration of a Class (p. 6)", also
`\Link{}{SimpleDefinition}` ⇝ "2 (p. 2)", or
`\Link[there]{}{ClassDeclaration}` ⇝ "there (p. 6)".

As can be seen in the examples, Links already include the leading text, resp. This is definitely preferred since it will e.g. allow to generate special language variants automatically.

2.4 Tools for the Ontology

Some tools have already been implemented to feed the ontology of a document into ontology-related tools based on the emerging OWL standard. Moreover, tools are available to display (part of) an ontology in a graphic way[11], cf. Fig. 1 (p. 5).

... more ...

2.5 Example: the Systems Ontology

As an example, consider an extract from the MMiSS systems ontology.

Inheritance of Standard Properties of Relations. Let us first look at the relations:

```
\Relation{*-*}{comprises}{comprises}{relatesDocConstructs}
  \Relation{<-}{embeds}{embeds}{comprises}
  \Relation{<-}{contains}{contains}{comprises}
\Relation{>}{reliesOn}{reliesOn}{relatesDocConstructs}
  \Relation{}{imports}{imports}{reliesOn}
\Relation{->}{pointsTo}{pointsTo}{relatesDocConstructs}
  \Relation{}{designates}{designates}{pointsTo}
  \Relation{}{references}{references}{pointsTo}
  \Relation{}{linksTo}{linksTo}{pointsTo}
```

[11] It is planned to generate an index from the ontology, similar to the corresponding LaTeX facilities.

These form a hierarchy, where each relation inherits the standard properties (p. 8) from its super-relation.

3 Structuring via Packages

Note: this section has not been finally edited and will be updated. It is included here for further background information.

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 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.

Packages are organised in a folder hierarchy. Every package can be identified uniquely by its *paths* in this hierarchy, called its *absolute path*.

It should be noted that the package mechanism relies on the repository to implement the separate namespaces and the imports between, since we cannot do this in LaTeX. Whenever we check out a document from the repository, all imports are resolved and checked for name spaces. After that, the import statements are replaced with the resulting LaTeX definitions. Thus, without the repository the package mechanism cannot be used to its full advantage. The details of this are explained below.

3.1 Packages

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 contains a prelude that contains a kind of "global declarations" for it, e.g. a BibliographyPrelude, or an ImportPrelude for other Packages ("structuring

In particular, it may contain an *OntologyPrelude*, where the elements of an ontology may be declared (cf. the Package for semantic interrelation, promising these elements to be defined in this Package, such that they become available when imported by another.

The *local name*s are those defined in this package (as opposed to names imported into the package). By default, all local names are exported. Imported names can be re-exported, if stated so in the import (see below). It is not possible to restrict the export; rather, name clashes and restrictions are resolved on import.

Which names are imported is specified by the *ImportPrelude*, which will be described next.

3.2 Import Prelude

A Package specifies the imported packages in the *ImportPrelude*. The ImportPrelude contains a number of Import Prelude Declarations; each specifies a Package to be imported, plus a number of *import directives*. Import directives can introduce package renamings, rename, hide or reveal specific identifiers, and specify qualify or unqualified import.

Paths and Path Aliases Every package has a name. Since packages are organised in a folder hierarchy, they can be identified by *paths*. A path is a list of package names, separated by dots. Since paths can get quite long, paths can be renamed by *Path* declarations (*alias*es) of the form

$$\mathtt{a} = \mathtt{p}_1.\mathtt{p}_2.\ldots.\mathtt{p}_n$$

where `a` is the new alias, and $\mathtt{p}_1,\ldots,\mathtt{p}_n$ are either folder names or previously defined aliases, subject to the condition that each alias is defined exactly once, and Path declarations are acyclic.

There are three *special aliases*: `Current` refers to the folder of the package it is used in; `Parent` refers to the parent folder; and `Root` refers to the root folder of the folder hierarchy (thus, the three special aliases correspond to '`.`', '`..`' and '`/`' in Posix systems). Users are discouraged to use the `Root` alias, since it makes reorganising the folder hierarchy difficult; it is mainly intended for usage by tools.

Qualified Import After importing a package, we can use the semantic terms, classes and structural entities it defines. If we import *qualified*, we have to prefix the name with the (possibly aliased) package name; if we import *unqualified*, we can use them as they are.

Semantic terms and classes are *always* imported unqualified. The default behaviour, if nothing else is specified, is to import structural elements qualified. The rationale here is that semantic terms and classes may define LaTeX command names, and a name like `A.\Plus` is not a valid LaTeX command name, so we do not allow the qualified import of semantic terms.

Local and 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 a name is imported under a different name.

It is illegal to specify a name both to be hidden and revealed. It is allowed to specify that a name is both hidden and not imported.

3.3 Writing it down: LaTeX Syntax

This section introduces the actual LaTeX syntax in which the Import Prelude Declarations are written.

- A *package name* is given by the following grammar

$$\begin{aligned} \mathit{PackageName} ::= {} & \mathit{Identifier} \mid \mathit{Alias} \\ & \mid \mathit{PackageName}\,.\,\mathit{Identifier} \end{aligned}$$

  where an identifier is a package name, and an alias is path alias introduced with `\Path`. Package names are non-empty sequence of Unicode characters, starting with a letter.

- `\Path{`*A*`}{`*PackageName*`}`

  denotes a *path alias* $A = Path$.

- `\Import[`*directives*`]{PackageName}`

  denotes an import of the package specified by *PackageName*. The *directives* are as follows:
  - `Qualified` specifies the package is imported qualified;
  - `Unqualified` specifies the package is import unqualified;
  - `Hide{`*names*`}` names to be hidden as a comma-separated list;
  - `Reveal{`*names*`}` specifies that only the names in the comma-separated list are imported.
  - `Rename{`*names*`}` specifies renamings as a comma-separated list of equations `newName=oldName`;
  - `Global` specifies a global import, and `Local` specifies local import. Local import is the default.

  Note that the import directives are processed from the *right* to the *left*, and so the order of the directives is relevant.

3.4 Implementation in the Repository

As has been mentioned above, LaTeX itself has neither name spaces nor a sophisticated export/import mechanism. Hence, the package mechanism is implemented on the level of the repository as follows.

When we check out a package from the repository, a MMiSSLaTeX document is generated, and the import prelude is supplemented by the MMiSSLaTeX code which is actually imported. The import clauses itself, and the generated MMiSSLaTeX preamble, are contained in the document, and marked by comments (pragmas):

```
%% MMiSSLaTeX import statements:

... import statements ...

%% End of MMiSSLaTeX import statements.

%% MMiSSLaTeX document prelude:
%% - Generated - do not edit! -

... included definitions ...

%% End of MMiSSLaTeX document preamble.
```

When a package is commited back to the repository, the generated document 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 import statements themselves are void LaTeX commands; they only serve to generate a prelude.

This mechanism depends on users not wantonly deleting the pragmas marking begin and end of import prelude and generated document prelude, but on the other end offers them a far more flexible and versatile name space handling than one could hope to implement in plain LaTeX.

4 Quick Reference

4.1 Structural Entities

All structural entities are written as LaTeX environments in the following form:

`\begin{`*NameSE*`}[`*AttributeSettings*`] ... \end{`*NameSE*`}`

where `NameSE` denotes the structural entity, i.e. the LaTeX environment, resp. Unless otherwise indicated, these LaTeX environments have no extra parameters.

Packages. Each document corresponds to exactly one Package.

`Package` specifies all Attributes relevant for the document, as the root structural entity of the nesting hierarchy.

Sections. Sections are contained in a Package. They may be nested.

Each section should contain an Abstract, an Introduction, and a Summary, either as specialized sections or special paragraphs (see below).

`Section` generalizes, by appropriate nesting, the LaTeX commands `section`, `subsection`, `subsubsection`, etc. Note that the LaTeX counterparts do not specify appropriate ends.

Units. Units are contained in Sections. A unit embodies a concept; it should ideally fit on one page (or slide). The outermost unit (immediately contained in an enclosing Section) starts a new slide and provides the Title for it; nested units are presented on the same slide or follow-up slides with the same Title.

Closed Units. A closed unit must not contain other closed units; it may contain other composite units or atoms and enclose them as an obvious frame of reference. For example, the relation Theorem X lives in Theory Y is implicitly given when Y contains X.

It is assumed that a closed unit of a particular Formalism is always complete. For example, the Code contained in a Program, when collected and stripped of all other content, can be correctly submitted to a compiler (possibly related, as a module, to other Program modules).

`Paragraph` embodies a concept primarily described by text.

`Abstract` a special Paragraph of a Section giving a concise description of its contents.

`Introduction` a special Paragraph at the beginning of a Section giving an introduction to its contents, relating it to previous material ("upper left context").

`Summary` a special Paragraph at the end of a Section summarizing its contents, relating it to subsequent material ("lower right context").

`Theory` embodies a mathematical concept of a particular formalism; the Formalism attribute should be specified accordingly.
`Program` embodies a Program, (sub)Program or module of a particular formalism; the Formalism attribute should be specified accordingly.

Composite Units. Composite units are contained in Sections or closed units.

`Illustration` provides a "wrapper" for a Figure or Table, e.g. adding a legend. *To be defined.*
`Example` for a concept, often given by a Definition in the same closed unit,
`Exercise` self-paced for the reader.
`Assignment` for a student.
`Solution` to a Problem (Exercise or Assignment).
`Description` analogous to its LaTeX counterpart `description`; contains `Item`s.
`Itemize` analogous to its LaTeX counterpart `itemize`; contains `Item`s.
`Enumerate` analogous to its LaTeX counterpart `enumerate`; contains `Item`s.
`Definition` of a concept, usually in connection with a `Def`, i.e. an ontology definition.
`Theorem` special case of an assertion.
`Proposition` special case of an assertion.
`Lemma` special case of an assertion.
`Corrollary` special case of an assertion.
`Conjecture` special case of an assertion.
`FalseConjecture` special case of an assertion.
`Proof` of a term, e.g. some Formula, often stated in an assertion.
`Development` of a program fragment, e.g. some Code.

Unit Components. Unit components are contained in units but are not themselves units. They may have a special syntax differing from the usual one for structural entities given above, as indicated.

`Item` *Special syntax!* Identical to LaTeX `item`; to be used in lists.

Atoms. Atoms are contained in units.

`Text` Running text, formatted in normal font, i.e. Sans Serif on slides, Roman otherwise
`Source` Source text, formatted in `Typewriter` font
`Table` Default: `FormatAttribute = LaTeX`
`Figure` Default: `FormatAttribute = LaTeX`, i.e. the body contains LaTeX commands such as
```
\includegraphics[scale=0.6]{img/UserOntology6}
\caption{Ontology of Users and Roles.}
```
Positioning default is `[htb!]`.
To be improved by LayoutAttributes.

`Formula` Default: `FormatAttribute = LaTeX`, i.e. LaTeX Math mode.
To be improved: label as in LaTeX equation.
`Step` *To be defined more precisely.*
`Code` (At the moment,) identical to LaTeX `verbatim`;
no Attributes may be given at the moment; to be extended.
Default: `FormatAttribute = ASCII`; LaTeX `verbatim` is used.
(In the future,) the `Formalism` should be given, unless it is inherited.

Included Structural Entities. Included structural entities are written as a special LaTeX environment with two extra parameters, in the following form:

`\begin{Included}[`*`AttributeSettings`*`]{`*`NameSE`*`}{`*SEName*`}` ... `\end{Included}`

Usually, this environment needs no Attributes; thus the first parameter, giving the kind of structural entity, will follow immediately, e.g.

```
\begin{Included}{Section}{SimpleDefinition}
  \begin{Section}[% Label=SimpleDefinition,
                   Title={Definition of a Semantic Term},
                   Authors={...}]
    ...
  \end{Section}
\end{Included}
```

An included structural entity contains, as its body, the structural entity identified by the *SEName*, with the help of the Repository: it basically copies the source of the corresponding structural entity. *This generated body, i.e. the copy, must no be edited.*

If the Repository is not available, this effect can be simulated by manually inserting a copy, but then an update of the original will not automatically affect the included copies. Note that it may be necessary, in such a simulation, to add some of the Attributes that have been inherited in the context of the original, such as Authors in the example above; the tools of the Repository will take care of this automatically. *This is future work; please also comment out Labels for the moment.*

Outdated Structural Entities. The structural entities below (from previous versions of MMiSSLaTeX) are now outdated; they are accepted in a transitory period but will be rejected in the near future.

`Script` *outdated as unit*; use `Proof`, or `Development` instead;
functionally to be replaced by a Script attribute.
`List` *outdated as atom*; use `Description`, `Itemize`, or `Enumerate` instead.
`TextFragment` *outdated as atom*; use `Text` instead.
`ProgramFragment` *outdated as atom*; use `Code` instead.
`Clause` *outdated as atom*; use `Formula` instead.
`AuthorEntry` *outdated as atom; to be redefined as a prelude declaration*
`BibEntry` *outdated as atom; to be redefined as a prelude declaration*

Outdated Embedded Operations. The embedded operations below (from previous versions of MMiSSLaTeX) are now outdated; they are accepted in a transitory period but will be rejected in the near future.

`IncludeSection` (and all other Include-operations, resp.) *outdated as embedded operation*; use `included structural entity` instead.

4.2 Attributes.

AttributeSettings are written as a comma-separated list of pairs using an equal sign, in the following form:

```
NameAttribute = NameAttributeObject
```

Except for some structure attributes, attributes are inherited.

Structure Attributes. Except for the authors and the short authors attribute, structure attributes are not inherited and have to be specified anew for each structural entity, if so desired.

`Label` letters and underscore only; unique for each structural entity in a package; will also be used in the Repository and should therefore *always* be present.
`Title` suggestive for the corresponding structural entity. *Non-inheritance not yet implemented, so please specify as empty, if applicable, e.g. for atoms.*
`ShortTitle` abbreviation of the Title; Title as default.
`Authors` ...
`ShortAuthors` abbreviation of the Authors; Authors as default.

System Attributes. System Attributes are generated by the Repository.

`Date` ...
`Version` ...
`PreviousVersion` ...
`PriorAuthors` to be appended to the present list of authors

Variant Attributes.

`Language` for natural languages, according to the standards ISO 639 / ISO 3166.
- `en` English
- `en-GB` British English
- `en-US` AmericanEnglish
- `de` Deutsch
- `fr` Francais
- ...
- `ANY` any language

`FormatAttribute` for different document and exchange formats. *not yet used*

`ASCII` ASCII only
`ISOLatinOne` ISOLatin1 only
`LaTeX` LaTeX
`MMiSSXML` MMiSS-XML
...
`NONE` no format
`ANY` any format
`Formalism` for some formalism given in the (extensible) ontology.
...
`NONE` no formalism
`ANY` any formalism

Outdated Attributes. The attributes below (from previous versions of MMiSS-LaTeX) are now outdated; they are accepted in a transitory period but will be rejected in the near future.

`Caption` for Figure; use LaTeX.
`Notation` use FormatAttribute.
`LevelOfDetail` use options in the LaTeX `documentclass` declaration.
`InteractionLevel` use options in the LaTeX `documentclass` declaration; to be replaced shortly by script animation attributes.
`ListType` use `Description`, `Itemize`, or `Enumerate`.
`DescItem` use `Description` and `Item`.
`LinkText` no more in use, see new syntax for `Link`.