Salt

Model Guide

Florian Zipser <saltnpepper@lists.hu-berlin.de>
INRIA
SFB 632 Information Structure / D1 Linguistic Database
Humboldt-Universität zu Berlin
Universität Potsdam

Table of Contents

1. The aim of Salt

With Salt we provide a) an easily understandable meta model for linguistic data and b) an open source API to store, manipulate and represent data. Salt is an abstract model, poor in linguistic semantics. As a result, it is agnostic to linguistic schools or theories. The core model is graph based and therefore keeps the structural restrictions very low and allows for a wide range of possible linguistic annotations like syntactic, morphological, coreferential annotations and many more. You can even model your own very personal annotation as long as it fits into a graph structure (and so far we have not seen a linguistic annotation which does not). Furthermore, Salt does not depend on a specific linguistic tagset and this allows you to use every tagset you like.

Originally Salt was developed as a common meta model as part of the SaltNPepper project[1]. The aim of this project was to develop a converter framework (called Pepper) that is able to convert several linguistic formats[2] into each other. The job of Salt here was to be able to cover all kinds of different linguistic data with a single model. In the meantime, Salt was developed further into an own project and it is now

[1]see http://u.hu-berlin.de/saltnpepper
[2]for instance the PennTreebank format, TigerXML, the EXMARaLDA format, PAULA, GrAF, RST, CoNLL, the ANNIS format and many more

is part of several linguistic software solutions like ANNIS[3], Atomic[4]and of course Pepper.

This article addresses a wider range of readers. We want to satisfy readers coming from a linguistic background as well as readers coming from a technical background. As this is a balancing act between different domains, we try to provide simple additional information for specific terms and aspects of the different domains. If you get bored at some point, don't hesitate to step over these paragraphs to the more interesting parts. We always try to improve our software and guides as well. And since we are an open source community project, this is your chance to participate. So if you find typos or misleading parts of text, please let us know and the honor will be yours. Just mail to `<saltnpepper@lists.hu-berlin.de>`.

2. What is a graph?

Since Salt is a totally graph based model, it is important to have a basic understanding of what a graph is.

A graph is a very simple, but not a linguistic structure. So we need to abstract over linguistic data to press them into such a structure. To give a simple explanation of what a graph is, let us forget linguistics for a moment and think about humans and their relationships. Imagine a set of humans for instance your family or friends. In a graph, each of these humans will represent one node. The relationship, for instance between exactly two humans then is defined as a relation. In other words, a relation connects two nodes. Now the relations between humans can be very different, so for instance the relation between a couple can be described as a love relation, whereas the relation between an employee and her/his boss could be described as a work relation. This example shows us that the relations between nodes can be very different, as well as human relations could be. To differentiate the types of relations, they could be labeled. The same goes for the nodes: they also could be labeled, for instance with the human's name it represents. Returning to linguistics, this means, when we can model humans and their relationships as a graph, we can also model linguistic artifacts as a graph. For instance we can model texts, tokens etc. as nodes, linguistic categorization as labels and relations between them as relations. In the following, this will become clearer.

As we now have an informal understanding of what a graph is, we provide a formal definition of what we consider a graph to be. To model Salt, we enhanced the general directed graph structure, which is $G= (V, E)$ with:

- V being a set of nodes and
- E being a set of directed relations with $e= (v_1 \# V, v_2 \# V) \# E$.

3. What is Salt?

As already mentioned, Salt is a graph based model, therefore we expanded the graph structure presented in Section 2, “What is a graph?” with layers and labels and the mechanism to label a graph, a node, a relation, a layer or another label. The expanded graph structure is given by $G=(V, E, L, \{label_a, \dots label_b\})$ with:

[3]see http://www.sfb632.uni-potsdam.de/annis/
[4]see http://linktype.iaa.uni-jena.de/atomic/

- V being a set of nodes with $v= (\{label_c, \ldots label_d\})$ # V
- E being a set of directed relations with $e= (v_1$ # V, v_2 # $V, \{label_e, \ldots label_f\})$ # E
- L being a set of layers with $l= (V_1 \subseteq V, E_1 \subseteq E, L_1 \subseteq L, \{label_g, .. label_h\})$ # L
- and a set of labels $\{label_a, \ldots label_b\}$ the graph is labeled with.

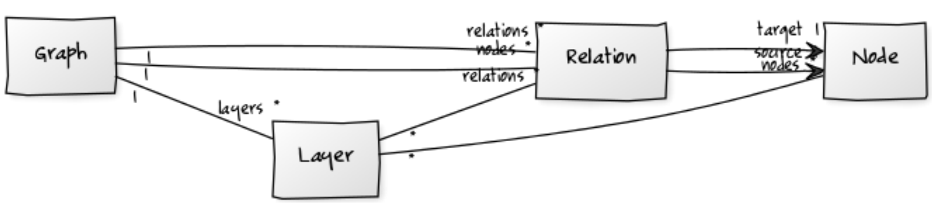
Each label is defined as $label_y= (namespace, name, value, \{label_k, \ldots label_l\})$.

A layer is a grouping mechanism for nodes and relations, and can also contain further layers (called sub layers). The containment relation implements a recursive structure for layers, to build hierarchies. In general this mechanism enables the creation of sub graphs. But note that a layer cannot be contained by itself, so cycles of layers are not possible.

A label is an attribute-value-pair contained in either a node, a relation, a graph, a layer or another label. The attribute-value-pair mechanism is realized by two components, a naming component and a value component. The naming component must be unique for one object and consists of the two values namespace and name. We expand the naming component with a namespace attribute to enable the possibility of adding more than one label with the same name to one graph, node etc. Now an object can have two labels with the same name but different namespaces. The value component is simpler and is just the value itself. The values of namespace and name are restricted to be a String value, whereas the value is unbound and can be in principle anything.

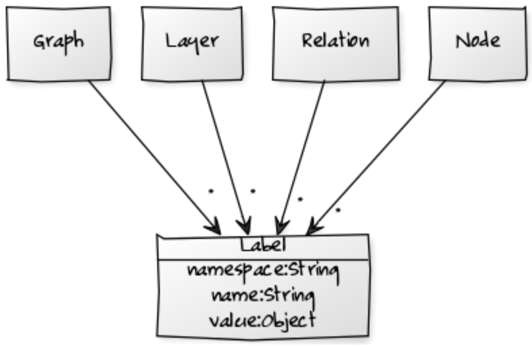
At last we want to regard Salt from a modelling point of view. Therefore Figure 1, “Salt's graph model (class diagram)” shows the base model, that we used for our abstraction for linguistic data. The figure shows, the above defined elements and their connection to each other. In Salt, we call a node `SNode`, a relation `SRelation`, a layer `SLayer` and a graph `SGraph`.

Figure 1. Salt's graph model (class diagram)



When you take a look at figure, you might wonder, what the attribute `sType` of element `SRelation` (`SRelation.sType`) means, since it is not introduced in the formal model and conflicts with the approach that everything we use in Salt must be one of the graph elements. Don't worry about, in fact the `sType` is even a label on a relation. We herefore used a specific type of label, the `SFeature` element which is described in detail in Section 3.1, “Annotations and label mechanism”. Since relations do not always need an attribute-value pair, the `sType` attribute for relations is a very useful mechanism for linguistic data. Sometimes it is enough to flag a relation being a specific type. If this sounds too abstract for now don't worry we give detailed examples in Section 4.2.6, “Loose relation” and Section 4.2.5, “Hierarchies”. Figure 2, “Label mechanism for graph, node, relation and layer (class diagram)” shows labels as model elements, which in Salt is called `SLabel`, and the connection between such a label and the other model elements.

Figure 2. Label mechanism for graph, node, relation and layer (class diagram)

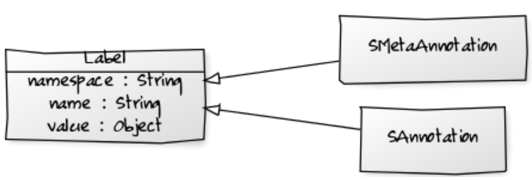


3.1. Annotations and label mechanism

Generally spoken, an annotation is an interpretation of primary data (see Section 4.2.1, “Primary data”). In Salt, we differentiate this abstract form of annotation into a "structural" and a "semantic" part. Imagine for instance a syntax tree. In Salt, the tree itself, which is modeled by nodes and relations, belongs to the structural part, whereas the assignment of a node or a relation to a category like being a sentence, a noun phrase etc. belongs to the semantic part. The semantic part is realized by labeling a node or a relation for instance by adding a label with the name '*cat*' and value '*S*' (following the TIGER scheme[5], where '*cat*' stands for category and '*S*' for sentence). Note that such a tagset is not part of Salt. Salt is poor in semantics, which means you can use every tagset you like.

We further differentiate between a linguistic annotation and a meta annotation. A linguistic annotation defines a structural element as a specific linguistic category. A meta annotation adds linguistic and non-linguistic information on a meta level to a structural element. For instance language information to a primary text, information about an annotator of a syntax tree and so on. But still both sorts of annotations are derivates of a label and are therefore a triple consisting of a namespace, a name and a value as shown in Figure 3, “Annotations in Salt are specific types of labels (class diagram)”.

Figure 3. Annotations in Salt are specific types of labels (class diagram)



Since annotations are a core essence of linguistic work, they will be used at various places in the following.

Next to `SAnnotation` and `SMetaAnnotation`, there are two further subtypes of `Label`. One is `SProcessingAnnotation` and the other is `SFeature`. The type

[5]see http://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/TIGERCorpus/annotation/tiger_scheme-syntax.pdf

SProcessingAnnotation is not part of the model, this type could be used, to add some information to any Salt object during a processing. So for instance you can store any state like '*already processed*' or other non-linguistic and non meta annotations like 'having the color red'[6] to it. Unfortunately the label SFeature is a bit more complicated. The good news is, that you normally will not need to work with it, when using the Salt API. The SFeature is a mechanism to enhance our graph elements with class attributes[7]. We will explain this in more detail in Section 4.2.3, "Annotations". We just mention it here since we have used the SFeature mechanism to model the class attribute SRelation.sType.

If it is not clear what this has to do with modeling linguistic data, we hope to make it clear in the following sections. But always keep in mind, that everything in Salt and all its power is reducible, to the here presented graph structure. And since the nodes and relations in Salt are just used as placeholders, the real power - especially the linguistic one - is in the labeling mechanism, which is widely used in Salt as you will see in the following.

4. How does Salt work?

This section addresses the single components of Salt and the linguistic aspects they are covering. It is divided into two parts, the corpus-structure and the document-stcuture. The corpus-structure is a grouping mechanism, to organize a corpus, whereas the document-structure covers the 'real' linguistic part.

4.1. Corpus-structure

A corpus-structure structures an entire corpus into smaller logical units. Such units are a corpus, a subcorpus and a document. Often the structuring goes along with the logical structure of the real data. Imagine your corpus represents a collection of writings of an author, then you may have one subcorpus per writing, which itself contains subcorpora representing the chapters or articles, which again might be divided into paragraphs etc. Dividing data has two main benefits, a logical and a practical. From a logical point of view, the corpus-structure keeps the hierarchical relation of units as given in the real world item. And from a practical point of view, it keeps things simple. For instance several human annotators can work on several units in parallel. Furthermore this will also speed up automatic processing, since data fits easier into main memory and indexes on them can be kept small.

As a quick reference Figure A.1, "corpus-structure (inheritance graph)" and Table A.1, "Overview over all elements of corpus-structure" in the appendix give a short description and an overview of the inheritance hierarchy of the elements being part of the corpus-structure.

4.1.1. Corpus

As mentioned above, a corpus is an element to organize your data. Similar to a folder in a filesystem, it groups the underlying parts (files and other folders). Abstractly spoken, a corpus is a selfcontaining structure which contains documents or further corpora. When a corpus contains another corpus, we call the container corpus the super corpus and the contained corpus the sub corpus. A corpus which is not

[6] For instance if you compute the chromatic number of the linguistic graph.
[7] In UML or object oriented programming languages an attribute is part of a class to store a state of a class instance.

contained by another corpus is called a root corpus. Each corpus can contain an unbound number of corpora. With this mechanism we now can represent a hierarchy as mentioned above. A corpus representing a collection of writings can contain further corpora, each representing a book. A book corpus itself can contain corpora representing a chapter, and so on. In Salt, a corpus is represented by the `SCorpus` element. Two `SCorpus` objects can be set into a "super corpus sub corpus"-relation by connecting them with a `SCorpusRelation` object.

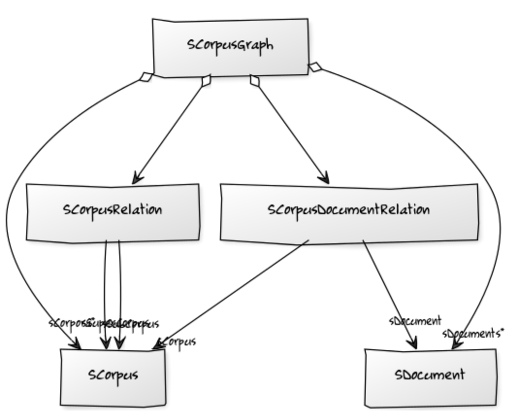
4.1.2. Document

A document is a logical partition which represents the end point of the corpus-structure hierarchy. Partitioning data means that no relations between data of two partitions are allowed. More concretely spoken, a document normally contains a single text and all annotations corresponding to it, but no interlinks between texts of different documents or their annotations. A text can be a paragraph, a chapter, an article or even an entire book. But a text can also be understood as the logical interpretation of it and be realized in several languages (called parallel text), or in case of historical texts in several normalized or diplomatic surrogates. These texts are often interlinked between same tokens (here 'same' means the same meaning, for instance in different languages). In that case all surrogates of a text **HAVE TO** belong to the same partition (document). Next to a logical partitioning, creating such documents has a high influence on processing speed and main memory. Therefore we highly recommend to keep documents as small as possible (as long as allowed by the linguistic logic behind). A document in Salt is represented by the type `SDocument` and can be grouped to a corpus or subcorpus by attaching it to a `SCorpus`. To mark a `SDocument` as being part of a `SCorpus`, just connect them via a `SDocumentCorpusRelation`.

4.1.3. Corpus graph

Since Salt is graph based over and over, the corpus-structure is represented as a graph, called the `SCorpusGraph`. This graph realizes a directed tree structure, whose nodes are corpora (`SCorpus`) and documents (`SDocument`) as shown in Figure 4, “Elements being part of the corpus-structure (class diagram)”.

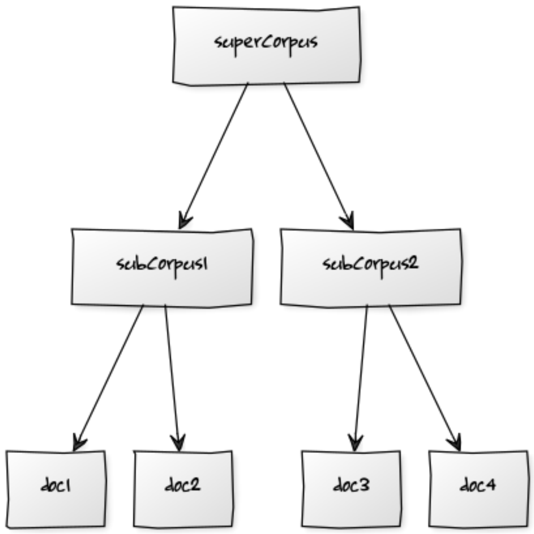
Figure 4. Elements being part of the corpus-structure (class diagram)



For those who prefer samples over UML diagrams, Figure 5, “corpus-structure sample (simplified object diagram)” shows a corpus-structure containing three

SCorpus objects *superCorpus*, *subCorpus1* and *subCorpus2* and four SDocument objects *doc1*, *doc2*, *doc3* and *doc4*. Two objects of type SCorpusRelation connect the *superCorpus* with *subCorpus1* and *subCorpus2*. Four objects of type SDocumentCorpusRelation connect the sub corpus subCorpus1 with documents *doc1* and *doc2* and sub corpus *subCorpus2* with documents *doc3* and *doc4*.

Figure 5. corpus-structure sample (simplified object diagram)



4.1.4. Meta annotations

Meta annotations are very useful for corpora to document, for instance, the creation process or the aim of the corpus. These information are supposed to give a person working with this corpus additional non-linguistic information. For instance which tools have been used, which persons have annotated the corpus, when was the corpus annotated and so on. Let's give an example: A meta annotation describing the date of the origin would have the name="date" and the value="1487" and an empty namespace can be added to a SCorpus. Salt is an open model, which means, there are no limitations on naming a meta annotation. Further, Salt does not interpret them, therefore the meta annotation for determining the author can also be named '*creator*' or something else instead of '*author*'.

The most convenient way to use meta annotations is to add a meta annotation to a document or a corpus node. But since a meta annotation is just a label of a specific type, you are free to add it to each node or relation in the Salt model.

4.2. Document-structure

In contrast to the corpus-structure, the document-structure covers the "real" linguistic data, which means primary data, linguistic structures and annotations above them. The linguistic structure contains the nodes: SSequentialDS, STextualDS, SMedialDS, SToken, SSpan and SStructure and the relations: STextualRelation, SMedialRelation, SSpanningRelation, SDominanceRelation, SPointingRelation and SOrderRelation which we will discuss in the following. All these nodes and relations are contained in a graph, the SDocumentGraph, which is the model element representing the document-structure.

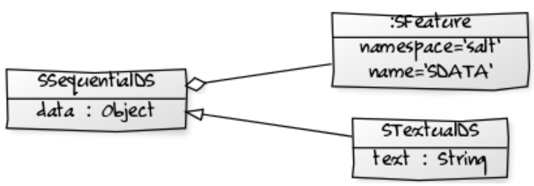
As a quick reference Figure A.2, “nodes of document-structure (inheritance graph)”, Figure A.3, “relations of document-structure (inheritance graph)” and Table A.2, “Overview over all elements of document-structure” in the appendix give a short description and an overview of the inheritance hierarchy of the elements being part of the document-structure.

4.2.1. Primary data

The primary data in linguistics are the center and the beginning of each annotation process. Every piece of language is a primary date. This includes textual data, audio-video data etc. . A special subtype of primary data is the primary text, which only covers textual data. Since in linguistics the term and the meaning of primary data and especially primary text is controversial, we here use primary data as the first digitalisation of data which comes into a Salt model.

So now the question is how to realize primary data in a graph based world. And the answer is: with graph elements, or more precisely with nodes and labels. In Salt, a specific node of type `SSequentialDS` is used as a placeholder for a primary date. The real data itself is added to that node with a label having the name `data` and the namespace *salt*. The same goes for each subtype of `SSequentialDS`, so for instance for the type `STextualDS` which represents primary texts in Salt, see Figure 6, “Primary data and primary text in Salt (class/object diagram)”.

Figure 6. Primary data and primary text in Salt (class/object diagram)



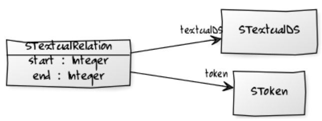
Regarding Figure 6, “Primary data and primary text in Salt (class/object diagram)” the primary text like '*Is this example more complicated than it appears to be?*' is stored as the value of the shown `SFeature`.

4.2.2. Tokenization

In general, it is not totally clear in linguistics what a token is. In most interpretations the term is used synonymously with 'word' (in the sense of graphmatics). But even here, the question what a word is, is controversial. Therefore we here use a more technical definition of what a token is. In Salt a token is the smallest countable unit of primary data. For instance in a primary text, a token could be a set of characters, just one character or even an empty character. This allows us, to use tokens free of a semantical interpretation. A token now can be a word, a syllable, a sentence or any other textual categorization.

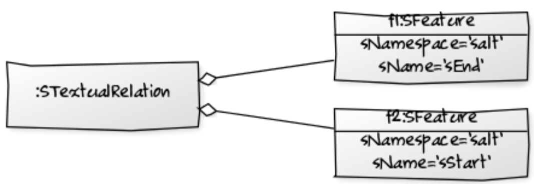
The Salt element representing a token is the type `SToken`, a specialization of the type `SNode`. Such a `SToken` object is a placeholder for annotations and a target for interlinking. The `SToken` object itself does not contain any information about the overlapped primary data. In case of the primary data being text, this is realized with a specific type of `SRelation`, the `STextualRelation`. A `STextualRelation` links a primary text (as source) with a token (as target), see Figure 7, “Representation of tokens in Salt via `SToken` and `STextualRelation` (class diagram)”.

Figure 7. Representation of tokens in Salt via `SToken` and `STextualRelation` (class diagram)



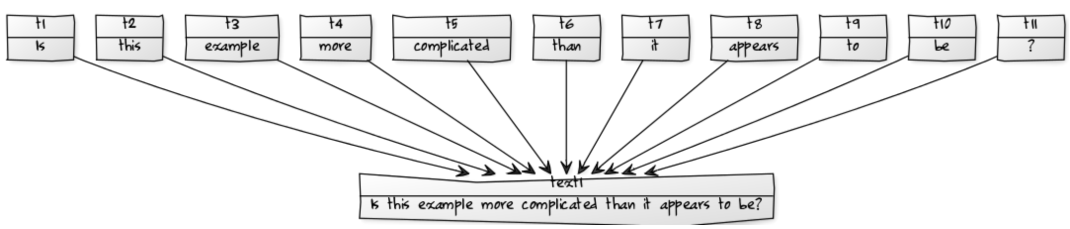
A `STextualRelation` further contains two labels (or more precisely `SFeature` objects) representing the start and the end position determining the interval of the primary text overlapped by the token. These labels are of type `SFeature` as shown in Figure 8, “Start and end position for text intervals realized with `SFeature` (f1 and f2) (object diagram)”.

Figure 8. Start and end position for text intervals realized with `SFeature` (f1 and f2) (object diagram)



Finally, Figure 9, “A sample tokenization (simplified object diagram)” gives an example of a tokenization of the primary text '*Is this example more complicated than it appears to be?*' modeled in Salt.

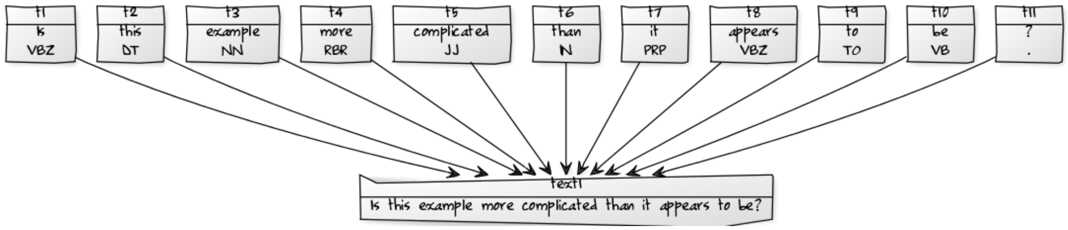
Figure 9. A sample tokenization (simplified object diagram)



4.2.3. Annotations

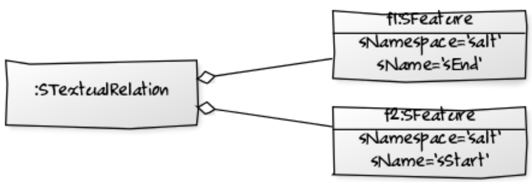
In the last sections we showed how to model the "structural" part of annotations (we have often called it placeholder). Now we want to give an impression of how to do the "semantic" part. Therefore we pick up the sample used in Section 4.2.2, “Tokenization” and especially its tokenization. We want to enhance the tokenized words with part-of-speech annotations. We already introduced the labeling mechanism in Section 3.1, “Annotations and label mechanism”. Now we want to make use of it by adding a `SAnnotation` object to each token having the `name` '*pos*' and the corresponding part-of-speech value as `value`. Figure 10, “Part-of-speech annotations of sample tokenization (simplified object diagram)” shows the annotation for the previous used tokenization sample.

Figure 10. Part-of-speech annotations of sample tokenization (simplified object diagram)



Each of these annotations are reducible to labels of type `SAnnotation` and Figure 11, "Part-of-speech annotation '*VBZ*' for token t_1 (object diagram)" exemplifies the annotation of token t_1 covering the text '*is*' with a part-of-speech annotation.

Figure 11. Part-of-speech annotation '*VBZ*' for token t_1 (object diagram)

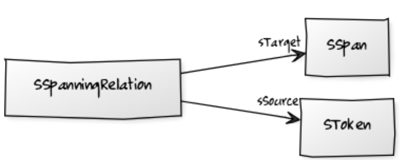


We here exemplified the creation of annotations by annotating tokens with part-of-speech annotations. But remember, that Salt is not bound to a specific set of annotations or tagsets. This means, you can use any kind of annotations with the same machanism. Furthermore adding an `SAnnotation` is not bound to tokens. Also any `SNode`, `SRelation`, `SLayer`, `SGraph` and even `SAnnotation` can be annotated in this way.

4.2.4. Spans of tokens

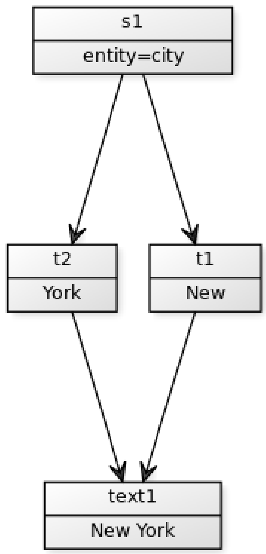
A span is used to group a couple of tokens together to give them exactly the same annotation or to connect them as a bunch with a 3rd node at once. A span therefore has the semantic of an ordered set. In a graph based world, we need to model such an ordered set as nodes and relations. Therefore Salt provides the node type `SSpan` and the relation type `SSpanningRelation`. A `SSpan` object represents the span itself and for instance could be annotated or linked with other nodes. To realize the containment of tokens in a span, each token is connected with the span with a separate `SSpanningRelation` object, see Figure 12, "Relation of spans in Salt via `SSpan` and `SSpanningRelation` (class diagram)". A `SSpanningRelation` always has a span as source and a token as target.

Figure 12. Relation of spans in Salt via `SSpan` and `SSpanningRelation` (class diagram)



Imagine a piece of a primary text like '*New York*' and two tokens t_1 (representing '*New*') and t_2 (representing '*York*'). For annotating them as an entity, you can create a span s_1 and connect t_1 with s_1 via one `SSpanningRelation` r_1 and t_2 with s_1 via a second `SSpanningRelation` r_2 as shown in Figure 13, “'New York'-sample as Salt objects (simplified object diagram)”. Since a `SSpan` is just a node, it can be further annotated for instance with an annotation '*entity= city*'.

Figure 13. 'New York'-sample as Salt objects (simplified object diagram)

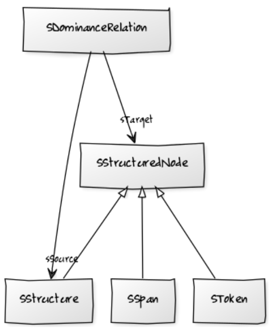


Spans can even be very helpful to annotate bigger parts of the primary text, for instance to annotate information structure or foreign language parts etc. . Also a `SSpanningRelation` can carry further annotations, but this is rather unusual since such a relation has no own linguistic semantics and is just a technical mechanism to model ordered sets in a graph.

4.2.5. Hierarchies

Hierarchies are a useful mechanism in linguistics to express a complex structure behind the surface of a text. For instance, a widely used mechanism to describe phrase structures are syntax trees. The term syntax trees implies that these hierarchies are trees, even in a graph sense. This means, they consist of nodes and relations and are therefore easy to model in Salt. Salt offers a specific type of node, the `SStructure`, and a specific type of relation, the `SDominanceRelation`. The source of an `SDominanceRelation` could be a `SToken`, `SSpan` or even a `SStructure` as shown in Figure 14, “Hierarchies in Salt are modeled with the elements `SStructure` and `SDominanceRelation` (class diagram)”. The unit of both elements enables to create unbound hierarchies above a tokenization.

Figure 14. Hierarchies in Salt are modeled with the elements `SStructure` and `SDominanceRelation` (class diagram)



The meaning of the type `SDominanceRelation` is a part-of relation. In contrast to the `SSpan` and the `SSpanningRelation`, a `SStructure` is not just a placeholder for a bunch of `SToken` objects, it is a proper element itself. The same goes for `SDominanceRelation` objects. For instance, in many cases it makes a linguistic difference whether tokens t_1, t_2 and t_3 are directly dominated by a structure s_1 or whether t_1 and t_2 are dominated by a structure s_2 which is, together with t_3, dominated by structure s_2, see Figure 15, “Syntax tree (t_3 directly dominated by s_1)” and Figure 16, “Syntax tree (t_3 indirectly dominated by s_1)”.

Figure 15. Syntax tree (t_3 directly dominated by s_1)

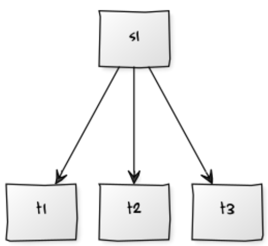
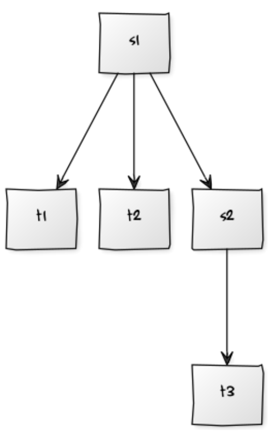


Figure 16. Syntax tree (t_3 indirectly dominated by s_1)



4.2.6. Loose relation

Sometimes it is necessary to set objects into a very loose relationship, as a kind of an interlinking, which does not influence the objects themselves. Such a relation is the type `SPointingRelation`. A pointing relation in Salt allows to relate any kind of nodes with each other. In general, this type of relation has no semantics and could be used for a wide range of annotations, which does not group or structure nodes. For instance, this could be very helpful for a dependency analysis or coreferential chains etc. .

To give an example, imagine the text '*John was a big man ... he always had to move his head*', where '*John*' (token t_1) and '*he*' (token t_i) refer to the same entity. To express that in Salt, you can create a `SPoiningRelation` object having the `sSource` t_1 and the `sTarget` t_i, or the other way around. Now let's extend this example and imagine the text '*John Doe ... he*', with the tokens t_1 ('*John*'), t_2 ('*Doe*') and t_i ('*he*'). In this case, we want to set '*John Doe*' as a whole in relation to '*he*'. This is possible by creating a `SSpan` object s_1 containing t_1 and t_2 and relating the token t_i via a `SPointingRelation` object to s_1.

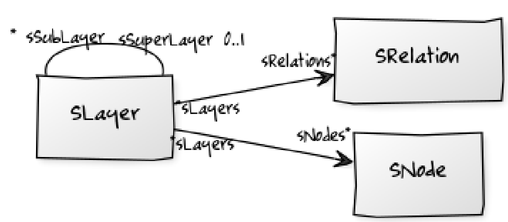
To determine a `SPointingRelation` being a coreferential relation or a dependency, you should use the `sType` attribute which is available for every `SRelation`. Like all the other relations, `SPointingRelation` can be annotated to express some linguistic semantics.

4.2.7. Layer

A layer is a simple grouping mechanism for bundling nodes and relations to a set. In graph theory, a layer is equivalent to a subgraph. Such a layer could be very helpful for linguistic data to distinguish between several kinds of annotations or annotation graphs. Imagine for instance a document-structure containing a set of tokens and a syntax tree. Now it might be helpful for your model to distinguish between these nodes. For instance when you group the tokens to a morphological layer and the nodes, being part of the syntactic tree, are grouped to a syntactic layer. Now you might ask: "Why should I? I can distinguish them by their type". And you are right in that case. But when nodes belong to the same type, but to different semantics, you need an explicit grouping mechanism. Imagine a set of spans annotating the information structure of a text and another set of spans annotating the topological fields.

In Salt, each node and each relation can belong to an unbound number of layers. Furthermore, a layer can also contain another layer. For instance a syntactic layer can contain a morphological layer. This makes all tokens being part of the syntactic layer as well. A layer is represented with the element `SLayer`, has a name and can be annotated in the same way as nodes and relations can be. Figure 17, “`SLayer` as a subgraph (class diagram)” shows the relationship between layers nodes and relations in Salt.

Figure 17. `SLayer` as a subgraph (class diagram)



4.2.8. Multiple primary texts

On the example of parallel texts

A lot of corpus projects in linguistics are more complex than handling just one primary text. They address parallel data dealing with multiple texts, for instance to compare different languages, different historical stages of language or to handle dialogue data with multiple speakers. In Salt the number of primary texts (element `STextualDS`) or primary data (element `SSequentialDS`) is not limited.

We want to demonstrate the use of multiple texts by creating a parallel corpus for the languages English and German. Demonstrations how to model dialogue data please find in Section 4.2.9, “Time management” and Section 4.2.10, “Ordering tokens”. Imagine the primary text '*Is this example more complicated than it appears to be?*' and its German counterpart '*Ist dieses Beispiel komplizierter als es zu sein scheint?*'. Creating two `STextualDs` objects '*text1*' and '*text2*' each containing one of the texts is rather simple[8]. The more interesting question is how to align the single words as being translations of each other. In case you are not so familar with German[9], we here present the translation alignment for the tokenized texts (for an easier alignment, we switched the words at the end a little):

```
Is    this    example    more complicated   than   it   appears   to   be?
Ist   dieses  Beispiel   komplizierter      als    es   scheint   zu   sein?'
```

Next to the fact, that English and German are sometimes very close to each other and for our example mostly have a word to word translation, we also have one case, where the two English words '*more complicated*' are translated to a single German word '*komplizierter*'.

To bring two tokens for instance t_{1e} and t_{1g}[10] in relation to each other, you can create a `SPointingRelation` r_1 which's source is t_{1e} and target is t_{1g}. Now they are connected, but more in a technical than in a semantic sense. To add the linguistic meaning to that relation, you can use the `sType` attribute and add for instance the type '`translation`'. For such a sample, our linguistic intention to determine that t_{1e} is the translation of t_{1g} and even the way around, the graph structure differs in detail. As our graph is a directed graph, we do not really have such a bidirectional relation. Depending on the interpretation of the corpus, it might be useful to create a second relation having t_{1g} as source and t_{1e} as target and to mark both relations as being either '*trans_en_de*' or '*trans_de_en*'.

Now coming to the more complex case of aligning the tokens t_{4e} ('*more*'), t_{5e}('*complicated*') with t_{4g} ('*komplizierter*'). To realize such an 1:n translation, we recommend using a span. With a span s_{1e} you can group the tokens t_{4e} and t_{5e}. This allows to use s_{1e} as source of the `SPointingRelation` and the token t_{4g} as its target.

If your individual case is even more complicated and needs to realize a n:m translation, just use spans on both sides. Group the tokens of the first language to a span and group the tokens of the second language to a span. Then connect the span on the one side with the tokens on the other side with a `SPointingRelation`.

In our sample we used just two languages for a better readability. Note that Salt is not bound to a fixed number of primary texts, which allows to model as many parallel texts with as many relations between their tokenizations as you like. For a well-arranged model, you can group all tokens and the primary text belonging to one language into one layer. Then you can set `SLayer.setName()` to the name of the language.

[8] Each primary text is is stored in the attribute `STextualDS.text`, see Section 4.2.1, “Primary data”
[9] By the way what is a pity.
[10] Where e stands for English and g for german

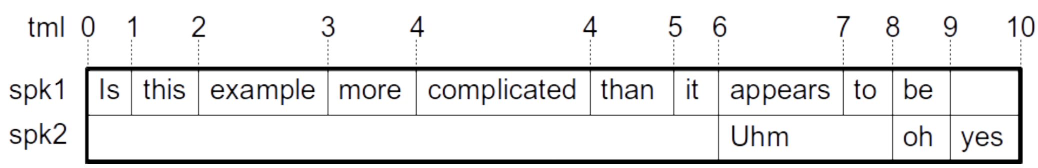
4.2.9. Time management

On the example of dialogue data

In this section, we address the time management in Salt. But what does it mean, time management? Remember, Salt is poor in semantics, so time management does not address the chronological progression of a text like "**Before** Bart went to school, he stood up."[11] With time management we mean the fact of ordering tokens in a primary text along their temporal occurance. This often becomes necessary when dealing with multiple texts. In Salt this is handled by introducing a global[12] unique timeline which is connected with each token.

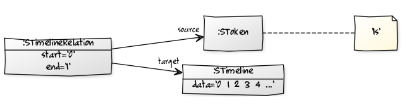
A lot of linguistic projects do not only address written texts, they further annotate spoken data. Spoken data differs in two areas from written data. First, the data source differs, since the primary data of spoken data is mostly an audio or video stream. Since Salt was developed for covering textual data, for now, we expect that there is also a textual representation of the audio or video data. But nevertheless Salt also allows to represent such data. This mechanism is addressed in detail in Section 4.2.11, "Audio data". Second, in many cases we do not have one continuous text, since there are multiple speakers, which might speak at the same time. We recommend modelling each text, belonging to one speaker, in a separate `STextualDS` object[13]. To give a more concrete example, imagine the following two texts produced by speakers '*spk1*' and '*spk2*':

Figure 18. Two speakers ('*spk1*' and '*spk2*') and the corresponding timeline (*tml*)



This sample shows the tokenized two texts and the correspondance of each token to an interval in the timeline. For instance the token '*Is*' (t_1) of speaker '*spk1*' corresponds to an interval starting at 0 and ending at 1[14]. Since we are in a graph world, this needs to be modeled by nodes, relations and labels. Therefore, the `STimeline` is a subtype of `SNode` and the `STimelineRelation` is a subtype of `SRelation` having two attributes (`SFeature` objects) `start` and `end` determining the start and end of the time interval. This is shown in Figure 19, "Token 'Is' related to common timeline (object diagram)"

Figure 19. Token 'Is' related to common timeline (object diagram)



This mechanism is quite simple, since it is the same mechanism that connects a token to a primary text. It might become more interesting in case of the time interval

[11]The modeling of such semantic issues is up to the user, for instance the ISO approach of ISO-TimeML covers things like that.
[12]Global here means global for one document-structure.
[13]In section Section 4.2.8, "Multiple primary texts" we discussed the mechanism to model multiple primary texts in Salt.
[14]In Salt the interval borders are also known as point-of-time.

between 7 and 9. In that interval the text of *spk1* covers the two tokens t_{spk1_8} ('*appears*')and t_{spk1_9} ('*to*'), and the text of *spk2* just covers the token t_{spk2_1} ('*Uhm*'). But modelling this is also very straight forward, because you can connect them via three `STimeRelation` objects: 1) t_{spk1_8} with the interval [7, 8], 2) t_{spk1_9} with the interval [8, 9] and 3) t_{spk2_1} with the interval [7, 9]. With these abstract points-of-time (like 1,2,3,4, ...), it is possible to set an unbound number of tokens in relation to the common timeline. But note, in each document-structure there can be only one `STimeline` object.

4.2.10. Ordering tokens

In this section we address the order of tokens in one or multiple texts. In a single ordinary primary text, the order might not be an issue, since the natural order of tokens is along their occurence in the primary text. In a Salt model all tokens are contained in a single list, which is sorted by their insertion. Imagine a primary text '*This is a sample*' tokenized in 4 tokens. If token t_3 ('*a*') was inserted into the list before token t_2 ('*is*') was inserted, the order of the list would be t_1, t_3, t_2, t_4. In that case, the list could be reordered in temporal order, which means along the offset in primary text (`STextualRelation.sLeft`). But what is with multiple texts? If we have multiple texts and order the corresponding tokens by their textual offset, we will end up with a mixed list of tokens of both texts. Imagine a second text '*What a nice sample*' also tokenized into 4 tokens t_{21}, t_{22}, t_{23} and t_{24}. The token list ordered by the textual offset would be the following useless list: t_1, t_{21}, t_2, t_{22}, t_{23}, t_3, t_4, t_{24}. A second option is the ordering by the offset in the timeline. Imagine both primary texts as a dialogue between two persons, with the following temporal order.

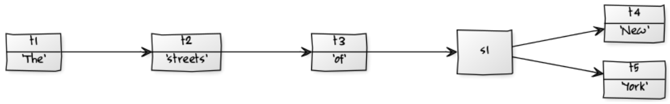
Table 1. Both texts as a dialogue.

spk1:	This	is	a	sample					
spk2:			What	a	nice	sample			

This could result in: t_1, t_2, t_3, t_{21}, t_4, t_{22}, t_{23}, t_{24}.[15] This list now represents the temporal order, but the tokens are also mixed in sense of the correspondance to the primary texts. So it depends on what you want to do with the data, to know which order might be the best.

Another way of ordering the tokens is an explicit order via the specific relation `SOrderRelation`. This relation allows to relate the node types `SToken`, `SSpan` and `SStructure`. With this relation you can create a path through the graph to represent any kind of order. But note, this path must be acyclic. When setting the `sType` of that relation, you can name the path. To give an example, imagine the text '*The streets of New York*' and a tokenization of t_1 ('*The*'), t_2 ('*streets*'), t_3 ('*of*'), t_4 ('*New*') and t_5 ('*York*'). Since you may want to annotate t_4 and t_5 as one word, you can create a span s_1 containing both tokens. A path of wordforms given by `SOrderRelations` could now be: t_1, t_2, t_3, s_1 as shown in Figure 20, "Path of word forms (simplified object diagram)"

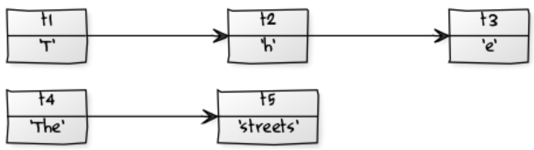
Figure 20. Path of word forms (simplified object diagram)



[15]Note, for two identical time intervals it is not clear, which token comes first.

Another use case for `SOrderRelation` objects are parallel tokenizations. For instance a tokenization of characters or letters and one by word forms. Here it may make sense, to create two pathes, one for the characters and one for the word forms as shown in Figure 21, “Path of word forms (simplified object diagram)”.

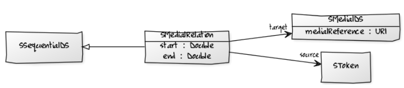
Figure 21. Path of word forms (simplified object diagram)



4.2.11. Audio data

Next to pure textual data like news paper articles, essays, internet chats etc. linguistics is also interested in spoken data like dialogues etc. . Even if Salt is a text based model, it can also deal with audio data. Audio data can be modeled as annotations or primary data. To create an audio annotation, the `SAnnotation.setValue()` must be set to an URI pointing to the audio file. To use audio data as primary data, Salt contains the model elements `SMedialDS` and `SMedialRelation`. These elements are very similar to the elements `STextualDS` and `STextualRelation`. The element `SMedialDS` is also derived from the element `SSequentialDS`, it provides the field `sMediaReference` which contains a URI referring to an audio file. With an object of type `SMedialRelation`, we now can connect such audio data with a token. To address a range in the entire audio stream, the element `SMedialRelation` contains the fields `start` and `end` to address the beginning and the end of that range as shown in Figure 22, “Audio data in Salt (class diagram)”. For instance to address a range beginning at 00:00:00 and ending 00:00:01. Since one second could be very long for spoken data, the start and end value is a very fine granular floating point number with a precision of 64-bit IEEE 754.[16]

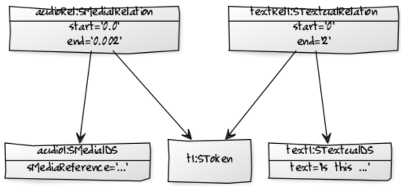
Figure 22. Audio data in Salt (class diagram)



Since Salt expects a textual transcription corresponding to audio data it is necessary to connect a token with first the audio data, second the primary text and eventually third the timeline. Imagine the primary text '*Is this example more complicated than it appears to be?*', which is tokenized by words and a corresponding audio file. Figure 23, “A sample of audio data in combination with primary text (object diagram)” shows the relation of the token *tok1* (just as a sample), the primary text *text1* and the audio date *audio1* via the `STextualRelation` object *textRel1* and the `SMedialRelation` *sAudioRel1*.

[16] A start value of 0.0 and a end value of 1.0 represents one second and an end value of 0.001 represents one millisecond.

Figure 23. A sample of audio data in combination with primary text (object diagram)



In Salt, the number of `SMedialDS` objects a model can have is not restricted and you can add as many of them as you like.

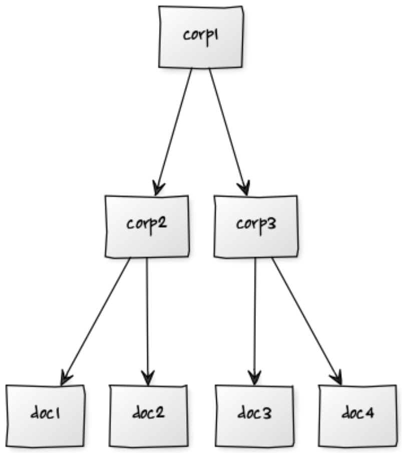
You can also combine the use of multiple audio and textual primary data with the use of the common timeline. The use of the timeline allows you to model the temporal order of tokens in such a case. This could be very useful in case of you have to model a dialogue, where each speaker is represented by an own audiofile and a separate transcription. With the timeline you can align the tokens in temporal order to each other. Imagine a dialogue consisting of two speakers, an audio recording for each speaker *audio1* and *audio2* and a corresponding transcription to each audio recording *text1* and *text2*. To connect the tokens with the common timeline, you just need a `STimeRelation` having the token as its source and the timeline as its target. For more information on that see Section 4.2.9, "Time management".

4.2.12. Names and Ids

In Salt each node, relation or layer can have a name. Each of these objects contains a field `name` (which is a String value). The `name` is added using the `SFeature` mechanism (see Section 4.2.3, "Annotations"). In Salt, there are no restrictions on that name, it even does not have to be unique. Because of that, the `name` could not reliably be used to identify exact a single object in Salt.

To identify objects, in Salt each object (except a label and therefore all types of annotations) provides a unique identifier, the `Identifier`. This identifier is unique for the whole document-structure or corpus-structure and is created by the containing graph object. An `Identifier` object is structured as an URI having the scheme *salt* and using segments for single Salt objects. Such a segment is given by the `name` of an object, if that name is unique. If not it is extended by an artificial counter to make it unique. In the corpus-structure, the `Identifier` represents the path from the root corpus to a specific object (`SCorpus` or `SDocument`). For instance imagine the corpus-structure shown in Figure 24, "A sample corpus-structure containing 3 corpora and 4 documents (simplified object diagram)"

Figure 24. A sample corpus-structure containing 3 corpora and 4 documents (simplified object diagram)



This corpus-structure results in the following `Identifier` objects:

Table 2. `Identifiers` corresponding to Figure 24, "A sample corpus-structure containing 3 corpora and 4 documents (simplified object diagram)".

salt:/corp1
salt:/corp1/corp2
salt:/corp1/corp2/doc1
salt:/corp1/corp2/doc2
salt:/corp1/corp3
salt:/corp1/corp3/doc3
salt:/corp1/corp3/doc4

An `Identifier` corresponding to an object which is contained in a document-structure, is the `Identifier` object of the document plus a fragment for the unified `name` of that object. For instance token *tok1* contained in document *doc1* gets the id: *salt:/corp1/doc1#tok1*. A second token also having the name *tok1* gets the id: *salt:/corp1/doc1#tok1_1*.

A. Appendix

1. Corpus-structure

Figure A.1. corpus-structure (inheritance graph)

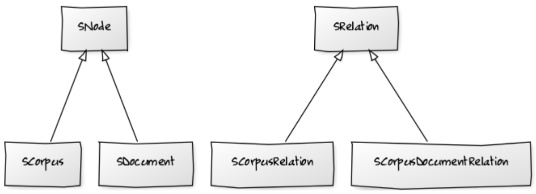


Table A.1. Overview over all elements of corpus-structure

element	short description
SCorpusGraph	A graph representing the corpus-structure itself, see Section 4.1.3, “Corpus graph”.
SNode	An abstract node, from which all other nodes are derived from, see Section 3, “What is Salt?”.
SCorpus	A container for documents or other corpora, see Section 4.1.1, “Corpus”.
SDocument	A container for primary data and annotations, see Section 4.1.2, “Document”.
SRelation	An abstract relation, from which all other relations are derived from, see Section 3, “What is Salt?”.
SCorpusRelation	Connects a super corpus with a sub corpus, see Section 4.1, “Corpus-structure”.
SCorpusDocumentRelation	Connects a corpus with a document, see Section 4.1, “Corpus-structure”.

2. Document-structure

Figure A.2. nodes of document-structure (inheritance graph)

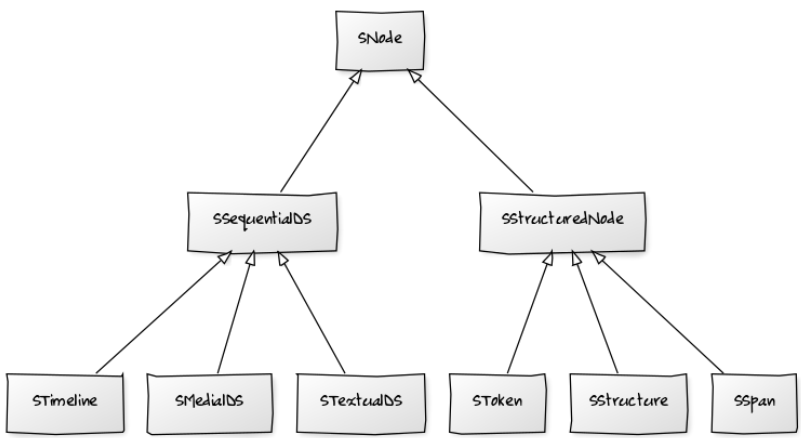


Figure A.3. relations of document-structure (inheritance graph)

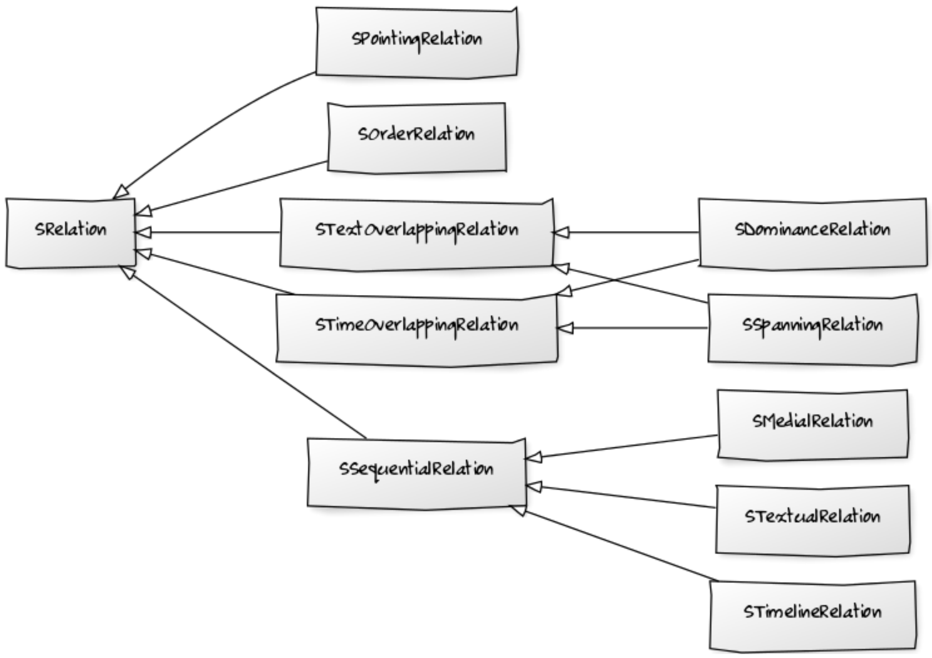


Table A.2. Overview over all elements of document-structure

element	short description
SDocumentGraph	A graph representing the document-structure itself, see Section 4.1.3, “Corpus graph”.
SNode	An abstract node, from which all other nodes are derived from, see Section 3, “What is Salt?”.
SSequentialDS	An abstract node used to model sequential datasources. Sequential here means, that you can address an interval of the datasource by determing a start and a end position.
SStructuredNode	An abstract node which is used for technical purposes, to constraint the kind of nodes being target of for instance SDominanceRelations, SPointingRelations.
STextualDS	A node representing primary texts, see Section 4.2.1, “Primary data”.
SMedialDS	A node representing audio data, see Section 4.2.11, “Audio data”.
STimeline	A common timeline for all objects in the document-structure, for instance to model time in dialogue data. There could be only one timeline object for one document-structure. See Section 4.2.9, “Time management”.

element	**short description**
SToken	Smallest annotatable unit of primary data, for instance a character, syllable, word etc., see Section 4.2.2, “Tokenization”.
SSpan	A node to model sets in a graph, to collect a number of tokens and to annotate them at once, see Section 4.2.4, “Spans of tokens”.
SStructure	A node to model hierarchies, for instance for syntax trees, see Section 4.2.5, “Hierarchies”.
SRelation	An abstract relation, from which all other relations are derived from, see Section 3, “What is Salt?”.
SSequentialRelation	An abstract relation which provides to address an interval (start and end value) in a data source, see Section 4.2.9, “Time management”.
STimeOverlappingRelation	An abstract relation marking the implementing relation to be a relation, giving the overlapped time interval from its target to its source (a kind of a contrary inheritance).
STextOverlappingRelation	An abstract relation marking the implementing relation to be a relation, giving the overlapped textual interval from its target to its source (a kind of a contrary inheritance).
STimelineRelation	A relation to connect a token with the common timeline, see Section 4.2.9, “Time management”.
STextualRelation	A relation to connect a token with the a primary text, see Section 4.2.1, “Primary data”.
SMedialRelation	A relation to connect a token with the a audio data object, see Section 4.2.11, “Audio data”.
SSpanningRelation	A relation used to create a set in a graph, this is used to connect a token with a span object, see Section 4.2.4, “Spans of tokens”.
SDominanceRelation	A relation to represent hierarchies, see Section 4.2.5, “Hierarchies”.
SPointingRelation	A loose relation to connect each kind of node with another one, see Section 4.2.6, “Loose relation”.
SOrderRelation	A relation to create an explicit order of tokens, see Section 4.2.10, “Ordering tokens”.