A Language for Specifying Hyperdocuments

Ernst-Erich Doberkat
Chair for Software Technology
University of Dortmund

Abstract

The architecture of hyperdocuments is described using an object-oriented approach and first order logic. The architectural description focusses on the static behavior of linkable document, it is based on persistent data the properties of which are described using mechanisms from logic programming. It is shown how to build up specialization hierarchies and generic document classes. The semantics of a hyperdocument is given compositionally through the semantics of its document parts.

1. Introduction

A hyperdocument is characterized by the underlying documents and the possible order in which these documents may be perused. This suggests a separation in the specification of these two aspects: the structure of a document should be specified separately from the way the entire document may be read. Specifying the document structure and the control flow separateley, however, requires a way of linking these descriptions for a specific hyperdocument, hence a third component is required which links these specifications, yielding a perusable hyperdocument.

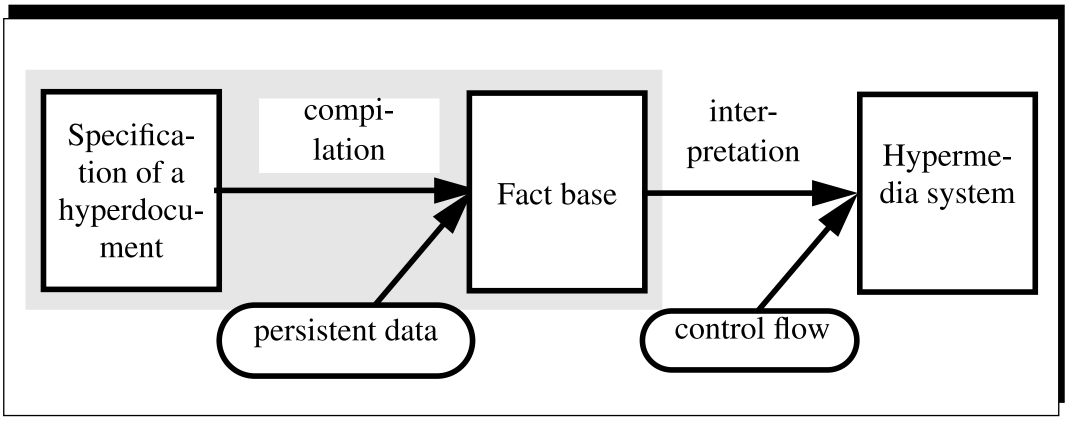
This way of approaching the construction of large hypertext systems is somewhat similar to the way software systems are specified: here the specification of the data is separated from the specification of the control flow, the description of program units (procedures, modules etc.) is separated from the decription of the program architecture.

When considering hypermedia systems, we distinguish the static aspects from the dynamic ones. The former are described when the data for the system are compiled, they are depending on the data proper and the static relations a designer wishes to constitute for them, they do not depend on the way a user peruses a document, and are independent of the way the document is actually displayed. In this sense, the description of data is similar to entity relationship models which indicate what kind of data are to be processed and how the data may be related to each other (cf. [GJM, 5.5.1]). A descriptive view of the data in a hypermedia system specifies the data and their properties, factoring out similarities and leaving aside specific attributes like storage representation etc. (which, in turn, will receive attention when necessary through suitable abstractions). This calls for an abstract description of documents, leading in turn to document classes, from which the documents of a hypermedia system are obtained through instantiation, resulting in an object-oriented approach to document description, since document classes may be related to each other through a specialization relation. Intra- and interdocument relations are also described; this is done via a subset of first order logic, to be more specific, using Horn clauses.

This paper will describe an approach to the specification of documents based on a blend of logic and specialization hierarchies. The description will be on an abstract level, thus concentrating on the conceptual aspects of document specification, neglecting the implementation and focussing on finding suitable abstractions instead. Hence this approach has to be complemented by a run time component (answering questions like: how are documents displayed? how are links to be followed?) and, equally important, by a specification of the browsing behavior of a hyperdocument. This is similar to software construction, where the static view of the data is complemented by the dynamic behavior of the functions operating on them, and, eventually, by the implementation. It turns out, however, that some aspects of the dynamic behavior of a document can already be captured through a static description, and this is what will be described here.

Concentrating on these aspects, we leave aside any considerations regarding the implementation. We will also neglect data base issues: the document descriptions will be made persistent eventually. But since we are for the time being interested in specifying documents rather than maintaining them, we think for the time being of the descriptions as volatile. So we do not discuss a data base schema for an OODB containing hyperdocuments here!

The run time aspects of a program, however, are captured in this paper not by constructing an executable hypermedia system, but rather by logically describing the non-linear structure of a set of documents using a set of facts. The compilation of a hyperdocument's specification will result in a set of clauses from which the facts are deduced. The following diagram shows that a



hypermedia system may be constructed through a sequence of additional steps, among others by interpreting these facts and by adding a control component that completes the specification of the browsing semantics (this exhibits a remote similarity to the *use cases* and *control objects* proposed in Jacobson's OOSE [Jac]). We will focus here on the concepts underlying the gray shaded part of the diagram. Executing the hypermedia system requires browsing tools; it is conceivable doing the interpretation with standard browsers used in the well known HTML/WWW context in mind. The approach outlined in this paper is rather flexible in permitting to concentrate on the deduction of facts rather than on directly representing the relationship between e.g. textual or graphical documents. New document types (like video or audio) together with new relations may easily be added on this level, requiring an independent extension to the interpretation and eventually the addition of the corresponding rendering tools. Hence it is justified to assume for the following discussion that a rather small set of basic facts may be made available when specifying hyperdocuments.

The object-oriented approach to document specification proposed here permits reusing document classes – once a class is specified, it may be instantiated in different contexts, hence building blocks for compound documents may be constructed. This is of course one of the ususal advantages of an object-oriented approach. The inheritance mechanism makes the customization of documents to particular user groups possible. Consider e.g. a manufacturer of machine parts who wants to make his maintenance procedures known to his customers and to the employees of his maintenance department. This implies that descriptions have to be provided one of which leaving aside technical details to the outside, the other one providing these details for internal use. Although the specific documents to be displayed may differ, the structure will be basically the same. Hence a document class can be specified from which classes for external and for internal documents may be derived through specialization. An abstract class description may in addition help in automatically constructing documents through indexing techniques, cp. [AMC].

The rest of this paper is organized as follows: we first have a look at the structure of simple documents, introducing our basic approach of modeling a document through persistent data (text, graphics) with facts associated with it. This view is extended by discussing the structure of complex documents that are composed of components. This poses questions of document specialization and of type parametrization, opening the road to inheritance and to generic document classes. Finally, the question of parameter passing to documents is discussed in the context of hyperdocuments, which may be thought as a main program enclosing all the local document structure. Some technical details and an example are given in appendices.

Acknowledgements: The author gratefully acknowledges helpful discussions with Stefan Dißmann and Udo Kelter.

2. Document Structure

Each document is assumed to be based on persistent data coming from some data base; during peruse of a hyperdocument, new volatile or persistent data may be created by the user. We will assume that we deal with those persistent data which are used for the construction of a hyperdocument at the time the document is specified, thus neglecting run time aspects. Given a persistent datum, we assume that it is typed, so that it can be worked upon by specific tools (like an editor, a browser etc.). The same data may be treated by a hypertext system in different ways (e.g. a text file may be annotated, so that a user may invoke a particular action by clicking on specific strings, but it may also be read linearly so that the reader may decide what document to read next). Thus a hypertext document will not only consist of a persistent datum, but its further use should be adumbrated. This observation leads to document classes which serve as abstractions describing the behavior of many concrete documents, and which permit building up specialization hierarchies by abstracting similar behavior and differentiating on specific properties. Moreover, a hyperdocument may consist of layers of other hyperdocuments, consequently we have to take the mechanisms of association and aggregation into account. In other words, we aim at an object oriented description of documents.

Given a document class, a specific behavior is associated with it. When specifying the class, the specifier may wish to describe abstractly some local pieces of control flow. An example of this is linking a portion of a graphics to an explaining text. This behavior is local since it concerns two instances of the class under consideration, rather than the behavior of the hypertext system in its entirety. The observation implies that with each instance of a document class there is associated a set of facts which, when interpreted suitably, determine the local behavior at runtime. Thus a class description has to cater for the specification of assumptions, which in turn are specified as facts. Facts may depend on conditions which are apparent only when the instance of a class is created. For describing conditions on facts we use Horn clauses and choose `PROLOG` [Bra] with a set of predefined clauses as a vehicle. In particular we adhere to `PROLOG`'s naming conventions. We will have to make some additional predicates available, their definition is given in an appendix.

Using `PROLOG`'s simple and elegant formalism with the well defined semantics of Horn clauses has the advantage that the specifications are concise, and that unification may be used, thus simplifying the control flow on the specification level. Moreover, `PROLOG` works on a set of facts, it permits adding or withdrawing facts with the `assert`, and the `retract` predicate, resp.

The maintenance of a fact base as a set of facts is important to our approach, and an alternative would have been to use `Linda`'s tuple spaces [CaGe]. In one of its extensions [Has], this would even permit multiple tuple spaces, thus enabling information hiding by dedicating tuple spaces to particular purposes. This, however, increases the descriptive overhead (explicit control structures have to be provided in this case), and instead of unification, only name based matching is available.

When describing the local control flow, we want to make use of *positions* in a document. This is a somewhat tricky notion, since its meaning may depend on the type of the piece of persistent data on which the document is based: the position of a string in an `ASCII`-text can be described uniquely by the number of character preceding the string, and by its length (i.e., by its coordinates), a description that is structurally different from a topologically connected region in a graphics document, which boils down to describing a set of pixels. Positions are akin to anchors in the Dexter model [HaSc]. We use the abstract notion of a position which will be interpreted according to the type of the persistent data underlying a document. Positions may be linked through the `link` primitive: a link for positions `one` and `two` is indicated by the fact `link(one, two)`. Note that the specific kind of link is left open. There are no dangling links, however.

The data base from which the data are drawn is of no concern to us, we talk about a `dbUnit` when referring to one of its elements. For the sake of simplicity, this paper will deal only with persistent data of external type `text` and of type `graphics`, resp. This restriction to ASCII texts and to graphis is not a severe one, since it will be easy to add new document types by extending the fact base. Hence given a `dbUnit x`, either of the following facts is associated with it:

```
extType(x, text).
extType(x, graphics).
```

We talk here about external types since our type system will permit user defined types, and these types should not be confused: the external type is imposed by the construction of the `dbUnit`, which is external to the hypermedia system, the internal type (indicated by `type`) is either primitive or user defined.

Positions have to be described for these external types. Given a string `s` and a text document `t`, we associate with the string its position as the ordered list `L` of all its occurrences. Each occurrence may be described e.g. by its coordinates in the text, but this is an implementation detail which for the time being is not relevant to us. Formally,

```
occ(x, t, L)
```

unifies `L` with all occurrences of the string `x` in text document `t`. Similarly, a position `P` in a graphics document `g` is described by a set `s` of pixels, whichever description of `s` may be chosen:

```
occ(s, g, P)
```

With each `dbUnit x` we associate its begin and its end as two distinguished positions, indicated by,

```
begin(x, B)
end(x, E)
```

which unify `B` and `E` with the begin and the end of `x`, resp.

We will describe first the specification of simple and flat documents, then we will discuss the inheritance mechanism for these documents, before we give a general description of structured documents.

2.1 Simple Document Classes

A simple document class is comprised from `dbUnits` as the building blocks of a hypertext document. In addition to a `dbUnit`, constraints may be given which characterize the document's class on the behavioral level. These constraints may deal with the `dbUnit` itself, but it may also provide information about the links emanating from an instance of that class. The constraints section is comprised of a set of facts and clauses, it is executed when a document class is instantiated, as explained below.

Considering an example, we may wish to specify that the dbUnit under consideration is a text, and that we want to link each occurrence of a given string with the first occurrence of another one in another dbUnit which is also a text. This example may be formulated as follows:

```
class text_to_text is
     documents one, two: dbUnit; x, y: string;
     constraints
          bindStrings(x, one, y, two) :-
               extType(one, text), extType(two, text),
               occ(x, one, L1), occ(y, two, L2), first(F, L2),
               bindAll(L1, [F]).
end text_to_text;
```

This class operates on two dbUnits one and two, two strings x and y are also given. The predicate bindStrings succeeds, if one and two are texts, if L1 and L2 are the list of occurrences of x in one and y in two, resp., and if the predicate bindAll succeeds in binding all elements of L1 to F, the first element of list L2. The predicate bindAll is predefined: it takes two lists of positions and executes assert(link(r, s)) for each element r of the first and each element s of the second argument. If one or two are no texts, or if y does not occur in two (hence letting first fail), the clause bindStrings fails.

Suppose text_to_text is instantiated with text documents such that one contains the text of a program, b its specification, x and y are both the name of that module, the latter heading the module's specification. Then the facts about this object include link(ff, f) for each occurrence ff of the module name in the program and the position f of its specification.

The documents part declares the names local to the class declaration, these names are visible throughout the class and are accessible from outside through a qualification clause (see below). The constraints section contains Horn clauses or facts constraining the use of the entities named in the documents section formulated in PROLOG. The names of the clauses are either local (such as bindStrings) or built-in (bindAll demonstrates this). Visibility of PROLOG variables is restricted to their respective clauses.

Thus we have for the moment three separate name spaces:

* names declared as documents: they act as value parameters and have to be provided when the class is instantiated. Their visibility follows the scope rules from ALGOL,
* PROLOG clauses, which are either local to the constraints section they are statically located in, or are visible throughout,
* PROLOG variables the visibility of which is restricted, as indicated above.

Names declared as documents are typed. The following primitive types are provided:

* dbUnit, indicating a persistent entity,
* string,
* pixel, indicating a set of pixels.

The primitive types string is associated with a list of positions, pixel with a position. These associations depend of course on the dbUnit under consideration. Given a type t, the type constructor list of t is provided. Both primitive and compound types may be accessed in the constraints section: primitive types may be used as constants, accessible through the type predicate, lists may be manipulated by the usual PROLOG predicates (e.g., if x is declared as list of string, then as with PROLOG lists, first(F, x) either unifies with F the first element of this list, or it fails, if `x` is the empty list).

2.2 Specializing Simple Document Classes

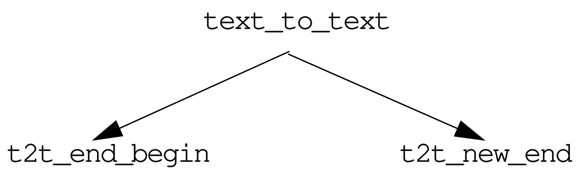
A document class may be used as a basis for other classes through specialization. This can be done by adding parameters and constraints in order to adequately model documents based on a more general specification. Consider again the following example:

```
class text_to_text is
     documents one, two: dbUnit; x, y: string;
     constraints
          inner;
          bindStrings(x, one, y, two) :-
               extType(one, text), extType(two, text),
               occ(x, one, L1), occ(y, two, L2), first(F, L2),
               bindAll(L1, [F]).
end text_to_text;
```

Note that we have augmented the `constraints` section with the innocently looking keyword `inner`. This time we want to model the former situation and in addition the following classes:

* a class `t2t_end_begin` of documents in which the end of `one` is linked to the begin of `two`,
* a class `t2t_new_end` of documents in which the end of `two` is linked to the begin of a third document `three`, provided `three` is of type `graphics`,

thus building up the following classification hierarchy



The way these extensions are declared is as follows:

```
class t2t_end_begin is text_to_text with
     constraints
          endToBegin(one, two) :-
               end(one, E), begin(two, B),
               assert(link(E, B)).
end t2t_end_begin;
```

and for the class `t2t_new_end`:

```
class t2t_new_end is text_to_text with
     documents three: bdUnit;
     constraints
          endToGraphicsBegin(two, three) :-
               extType(three, graphics),
               end(two, B), begin(three, C),
               assert(link(B, C)).
end t2t_new_end;
```

Let us have a look at the class `t2t_end_begin` first. The declaration of this class is semantically equivalent to the body of the declaration for the class `text_to_text`, with the `inner` part replaced by the definition of the predicate `endToBegin`. Thus instantiating the class `t2t_end_begin` will have the effect of executing the predicates `endToBegin` and `bindStrings` in this order. No new parameters are added in contrast to the declaration of the class `t2t_new_end`, where a fifth parameter (aptly called `three`) as well as the new predicate `endToGraphicsBegin` has been added. Instantiating this class will require values for the parameters `one`, `two`, `x`, `y` (imposed by `text_to_text`) and the added parameter `three`. It will have the effect of executing the predicates `endToGraphicsBegin` and `bindStrings` in this order.

Neither `t2t_end_begin` nor `t2t_new_end` may be extended further, since the respective constraint sections do not contain an `inner` part.

More formally, a `constraints` part may consist of facts and clauses, an extended `constraints` part may add the keyword `inner`. The extension of a class definition consists then of adding a `declare` part (to be discussed with compound documents), a `documents` part and an extended `constraints` part. The semantic effect of a class extension through the elaboration of an `inner` part is akin to a macro expansion: it is equivalent to a class definition which augments the declaration and the `documents` part of the parent class definition with the respective declaration and `documents` part of the extension, and which replaces each occurrence of the `inner` in the `constraints` section of the parent with the extended `constraints` parts of the extension.

The position of the `inner` part thus determines the extension's position. An extension may contain an `inner` part in turn, thus making multi-level hierarchies possible. Syntactically, an `inner` part may be placed in any position in which a predicate may occur. This implies that more than one `inner` may be placed into the `constraint` section of one class declaration; since elaborating the `inner` will have the effect that the added predicates will be plugged into each slot indicated by an `inner`, which will probably make not much sense. If new names are introduced to the `documents` section, this will do no harm. Note the visibility in the example above: the names introduced in the superclass `text_to_text` are visible in the subclasses `t2t_end_begin` and `t2t_new_end`. This follows from the explanation given above: when elaborating an `inner`, all names visible at the position of the `inner` are visible in the extension (this applies to names declared as `documents` and to predicates defined before the `inner` occurred); visibility may thus be propagated along many levels in a classification hierarchy. Not elaborating an `inner` does not have an effect at all, thus the class `text_to_text` may still be instantiated, in this case the `inner` part is neglected (i.e., treated as a no_op).

A general superclass `generalDocument` is provided, its definition is given by

```
class generalDocument is
    constraints
        inner
end generalDocument;
```

Each document class may be derived by specialization from `generalDocument`. Serving no practical purpose, this class is conceptually useful for providing an upper bound to the specialization hierarchy.

The present proposal of building up specialization hierarchies is adapted from the way the object-oriented programming language BETA introduces inheritance ([MMN,KLMN]). It permits only simple inheritance, thus avoiding the problems and pitfalls of multiple inheritance. BETA's mechanism is simple and elegant, yet surprisingly powerful.

2.3 Structured Document Classes

A simple document is characterized by the fact that it does not have local documents or document classes at its disposal. Thus we are not in a position declaring local documents in a simple one. In contrast, a compound document is composed of other documents, similar to a book which consists of chapters, sections, subsections and paragraphs (an observation which is not really new). Let us have a look at a book: suppose we have a novel which is supposed to be read from start to finish[1]. We would describe the corresponding document classes modeling paragraphs, subsections, sections, chapters and finally books so that each class depends on the

1. Well, probably not: when the Austrian novelist A. Stifter published his long novel *Der Nachsommer* in 1857, the German poet C.F. Hebbel promised the crown of Poland to everybody who would voluntarily read it from start to finish. Anyway.

previous one, again constructing a hierarchy, which in contrast to the classification hierarchy from above is an associtation, i.e., a *uses*-hierarchy modeling the use of (sub-) documents. We will return to this example later when discussing generic classes.

2.3.1 Compound Documents

Documents are in general logically composed of other documents, thus an architectural description for documents must be prepared to model this aspect. The components of such a document carry their own characteristics, i.e. have their own type. Thus we extend the type system by classes and lists of classes, hence we are permitted to state e.g. that a variable is of type, say `list of text_to_text`.

Objects are created by binding their name to a document class, e.g., `ttt: text_to_text` yields a document of type `text_to_text`.

A compound document consists of the following components:

* a `declare` part, in which types are declared. A type declaration may be a class definition or a list definition referencing another visible type. The classes declared may be generic, and they may be obtained through the specialization of other visible classes. Each declaration forms its own scope, visibility follows the `ALGOL` scope rules,
* a `documents` part, which gives the value parameters to the class; the types of the parameters should be visible,
* an extended `constraints` part describing the characteristics of the class; it may refer to the class itself as usual as `self`.

The compound document may be generic, see below. Each instance of a document class yields some facts associated with it and emanating from the corresponding `constraints` section. The facts for the instance of a compound document are obtained by combining the facts obtained from the parameters with the facts generated from the `constraints` section in the class definition.

If a parameter `f` is declared as having type `g`, each instance of type `g` having a parameter `x`, then `f` may access `x` through qualification as `g.x`.

To illustrate compound documents, let us specify a table of texts such that for each row we have a `graphics` document the end of which which is to be linked to the begin of the first entry.

```
class table is
     declare
          class tabEntry is
               documents t: dbUnit;
               constraints
                    isProper(t) :- extType(t, text).
          end tabEntry;
          row is list of tabEntry;
     documents
          tt: list of row;
     constraints
          -- make sure that each row has the same length
          equalLength([]).
          equalLength(LL) :-
               first(FL, LL), rest(RL, LL), first(SL, RL),
               length(FN, FL), length(FS, SL),
               FN = FS, equalLength(RL).
          equalLength(tt).
          inner
end table;
```

We have encapsulated the definition of a table entry as well as the representation of the table in the class definition. The constraints merely check that we indeed have a well-formed table. Now we add graphics:

```
class table_with_graphics is table with
     declare
          class graphicsEntry is
               documents g: dbUnit;
               constraints
                    isGraphics(g) :- extType(g, graphics).
          end graphicsEntry;
          graphRow is list of graphicsEntry;
     documents
          gr: graphRow;
     constraints
          -- make sure the lengths of tt and gr coincide
          lengthOk :-
               length(Nt, tt), length(Ng, gr), Nt = Ng.
          -- link the entries
          linkThem([], []).
          linkThem(TT, GR) :-
               first(Tab, TT), first(Gra, GR),
               rest(RTab, TT), rest(RGra, GR),
               begin(Tab, BTab), end(Gra, EGra),
               assert(link(EGra, BTab)),
               linkThem(RTab, RGra).
          linkThem(tt, gr).
end table_with_graphics;
```

2.3.2 Transactions

The `constraints` section may change the set of facts, hence from this point of view it is considered as a transaction. In particular it is atomic, so that either all changes are committed or none of them is. The `constraints` section is said to succeed iff each goal attached to a parameter succeeds, otherwise the section is said to fail. If it fails, the changes will be discarded and the document description itself fails. If it succeeds, however, and if the document under consideration is contained in another document, its success is propagated one level up in the nesting hierarchy. This implies a tree structure among the transactions – a transaction succeeds iff all child transactions succeed, and if the corresponding constraints section succeeds. Changes to the fact base will only be committed if the root transaction succeeds.

2.3.3 Generic Documents

A generic document is parametrized by a document type which will be provided when using this class. The semantics of generic replacement is given by a macro mechanism: replacing a generic type yields a class definition, where each occurrence of the type parameter is replaced by the type proper. Thus, after the specification of

```
generic class xxx[tt] is <body> end xxx;
```

the generic instantiation

```
class xxx_dd is xxx[dd];
```

semantically equivalent to the declaration

```
class xxx_dd is <body*> end xxx_dd;
```

where `<body*>` is `<body>` with each free occurrence of the type parameter `tt` replaced by `dd`.

The declaration of a generic document's body looks very much like the declaration of a document class proper: it consists of a `declare` part, a `documents` part, and an extended `constraints` part. The header of a generic class definition indicates the type parameter.

Inheritance (i.e., specialization) is available also on the generic class level; the semantics of the `inner` mechanism extends to the generic case. In particular, specializations of a generic class all work on the same type parameter, and after generic instantiation, the `inner` mechanism is available. Generic instantiation and specialization are commutative, provided the code plugged into the `inner` slot does not depend on the generic type parameter. Thus, instantiating a generic parameter and then doing the specialization results in this case semantically in the same class as doing the specialization first and then instatiating the generic parameter.

Thus the generic class definition

```
generic class listClass [T] is
     documents x: list of T;
     constraints
               linkThem(L) :-
               first(F, L), rest(R, L), first(R, G),
               end(F, EF), begin(G, BG),
               assert(link(EF, BG)),
               linkThem(R).
          doLink(x) :-
               first(F, x), last(L, x),
               assert(begin(x, F)), assert(end(x, L)),
               linkThem(x).
          inner;
end listClass;
```

specifies a class of documents which is based on a list of documents of class `T` such that the end of each document in this list is linked to the beginning of its successor, moreover the `begin` and the `end` markers are set for this list. Using this generic class, we may describe a linearly perusable book.

The basic building blocks are paragraphs:

```
class paragraph is
     documents x: dbUnit;
     constraints
          isParagraph(x) :- extType(x, text).
end paragraph;
```

Subsections are lists of paragraphs, similarly for sections and chapters:

```
class subsection is listClass[paragraph];
class section is listClass[subsection];
class chapter is listClass[section];
```

Let us add a preface and a conclusion to the book, both are supposed to be `dbUnits`:

```
class book is
     documents preface, conclusion: dbUnit;
                    chapts: list of chapter;
     constraints
          begin(self, B) :- assert(begin(preface, B)).
          end(self, E) :- assert(end(conclusion, B)).
          do(chapts) :-
               begin(chapts, B), end(chapts, E),
               begin(conclusion, CB), end(preface, PE),
               assert(link(PE, B)), assert(link(E, CB)).
end book;
```

Note that the classes under consideration carry an `inner`, thus may be specialized. This follows from the semantics of generic instantiation. For demonstrating the versatility of our approach, we will show in an appendix how to add a table of contents to the book.

3. Hyperdocuments

A hyperdocument correspondeds to a main program. It is a sequence of compound documents followed by a `binding` section which contains assignments of values to parameters.

We assume for simplicity that the persistent store is the file system of the machine the hypermedia system is constructed on, so that a `dbUnit` may be referenced by the name of a file which carries additional information about the persistent entity's type (e.g., the suffix `.txt` and `.gra` for an entity of type `text` and `graphics`, resp.). Working with a more complex persistent environment may require other assumptions; it is crucial, however, that we may identify an object and find out its type. A string is given as a string of characters enclosed in hyphens, a set of pixels is for the time being specified as a rectangular area, specified by a triplet `rect(lowerLeft, xCoord, yCoord)` indicating its lower left corner `lowerLeft` and its extension `xCoord` and `yCoord` in the *x*-, and the *y*-coordinates, resp.

3.1 The `binding` Section

The `binding` section associates a value to the parameters provided in the `documents` section of the hyperdocument. This is a complex procedure and will be discussed now informally. It will eventually result in a `PROLOG` program, i.e., in a set of facts and clauses which will be collected for the hyperdocument, yielding the `PROLOG` program assigning semantics to the hyperdocument.

Value assignment together with the definitions of the value-function *val* and the type structure function *struct* works recursively as follows. Suppose the document `x` is declared as `x: t`. Then we distinguish these cases:

1.) if `t` is a primitive type and the value to be assigned is *v*, value assignment is expressed by

```
let r:: v in x = r;
```

with the fresh name `r`. The value *val(x)* associated with `x` is defined as *(x, v)*, and *struct(x)* is assigned the external type of *v*.

2.) if `t` is a compound, but not a list type, and if the `documents` section of `t` contains (possibly after elaborating generic definitions and `inner` sections) y_1: s_1, ..., y_n: s_n, value assignment is expressed by

```
let
     let r1:: <value assignment for r1: s1> in y1 = r1;
     ...
     let rn:: <value assignment for rn: sn> in yn = rn;
in x;
```

Again, r_1, ..., r_n are fresh names. We put

$$val(x) := (x, (y_1, \pi_2(val(r_1))), ..., (y_n, \pi_2(val(r_n))))),$$

where π_2 denotes the projection to the second component. The type structure of `x` is determined as

$$struct(x) := (struct(r_1), ..., struct(r_n)).$$

3.) if `t` is a list type, say `list of s`, and we want to be `x` a list of *n* elements, the value assigned is

```
let
     {
          <value assignment for r: s> -- first value
          ...
          <value assignment for r: s> -- nth value
     }
in x;
```

Let v_i be the value resulting from value assignment to the *i*th list element, define the value as

$$val(x) := (x, (\pi_2(v_1), ..., \pi_2(v_n))),$$

the type structure is defined as

$$struct(x) := struct(r)^n$$

For primitive types,

```
let r :: v in x = r
```

may be abbreviated as

```
x := v
```

For compound documents, the somewhat heavy notation for assigning the component $y_i : s_i$ of a primitive type s_i

```
let r_i :: let r :: v in r_i = r; in y_i = r_i
```

may be abbreviated by

```
y_i := r_i.
```

As an example, consider class `text_to_text` again. Suppose we have a variable `t2t` of that type, and we wish to perform the following assignments:

one	„program.txt"
two	„docu.txt"
x	„sort"
y	„module sort"

(hence each occurrence of the string `sort` in the text file `program.txt` is to be linked to the first occurrence of the string `module sort` in the text file `docu.txt`). This assignment is done through

```
let
      one := „program.txt"; two := „docu.txt";
      x := „sort"; y := „module sort";
in t2t;
```

resulting in, respectively,

value(t2t) = (t2t, ((one, „program.txt“), (two, „docu.txt“), (x, „sort“), (y, „module sort“))),

struct(t2t) = (text, text, string, string)

For demonstrating the syntactic structure of a hyperdocument, consider the following declaration of the hyperdocument demo:

```
hyperdocument demo is
     declare
          class text_to_text is ... end text_to_text; -- as above
     documents
          t3t: list of text_to_text;
          s4: string;
     bindings
          let
               {
               let
                    one := „pascal.txt"; two := „p_docu.txt";
                    x := „h_sort"; y := „procedure h_sort";
               in r;
               let
                    one := „modula.txt"; two := „m_docu.txt";
                    x := „q_sort"; y := „PROCEDURE q_sort";
               in r;
               let
                    one := „beta.txt"; two := „b_docu.txt";
               x := „sort"; y := „ sort: (#";
               in r;
               }
```

```
            in t3t;
            s4 := „Demo Document";
    end demo;
```

Document `t3t` is declared as list of `text_to_text`, it is assigned three elements according to the following table:

one	„pascal.txt"	one	„modula.txt"	one	„beta.txt"
two	„p_docu.txt"	two	„m_docu.txt"	two	„b_docu.txt"
x	„h_sort"	x	„q_sort"	x	„sort"
y	„procedure h_sort"	y	„PROCEDURE q_sort"	y	„sort: (#"

The assignment above yields the respective values

value(t3t) = (t3t,
(
((one, „pascal.txt“), (two, „p_docu.txt“), (x, „h_sort“), (y, „procedure h_sort“)),
((one, „modula.txt“), (two, „m_docu.txt“), (x, „q_sort“), (y, „PROCEDURE q_sort“)),
((one, „beta.txt“), (two, „b_docu.txt“), (x, „sort“), (y, „ *sort : (#“)),*
)

and

value(s4) = (s4, „Demo Document“),

hence

value(demo) = (demo, (value(t3t), value(s4)))

and

$$struct(demo) = ((text, text, string, string)^3, string)$$

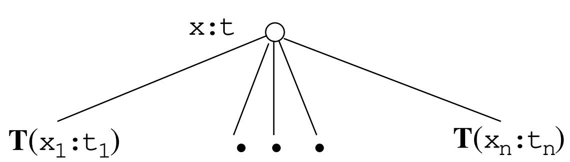
are eventually obtained.

3.2 Evaluating the Assignments

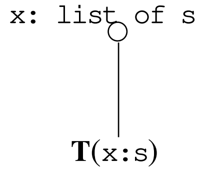
Using the value and the struct function, resp., the assignments are evaluated with the goal of obtaining a logical description of the hyperdocument in terms of facts and Horn clauses. Thus a `PROLOG` program will be constructed.

A *type tree* **T** is recursively constructed from the hyperdocument and the `documents` declarations (for simplifying the discussion, the declaration of a hyperdocument is considered a `documents` declaration `name:hyperdocument`, where `name` is the identifier given to the hyperdocument). Let `x:t` be a class definition, then

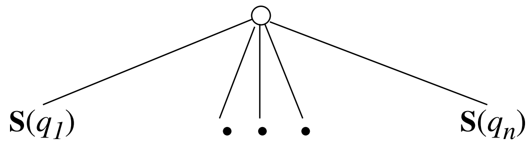
* if `t` is primitive, **T**(`x:t`) is a single node with label `x:t`
* if `t` is not a list type, and if it contains the declaration
  `documents` x_1`:` t_1`, ...,` x_n`:` t_n
  then **T**(`x:t`) is



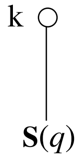
* if `t` is a list type, say, `t = list of s`, then **T**(`x:t`) is the tree



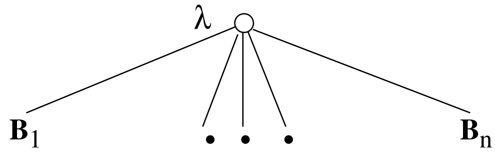
In a similar way, the *structure tree* **S**(q) is defined. If q is a primitive type, then **S**(q) is a single node with label q, **S**(q_1, ..., q_n) is defined as



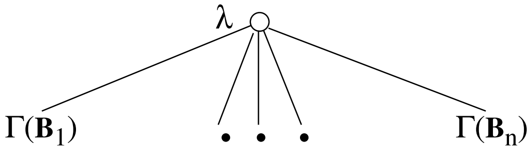
and **S**(q^k) is the tree



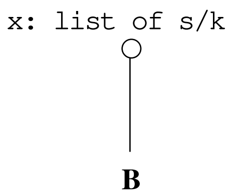
Consequently, the type tree of a hyperdocument is isomorphic to its structure tree, provided only the tree structure is considered (i. e., if labels are disregarded); let ϕ be this isomorphism. It should be noted that if a node ν in the type tree that indicates a list construction, the node $\phi(\nu)$ in the structure tree carries a non-negative integer indicating the actual length of the list, i.e., the number of list elements. From **S** and **T** a new tree **A** is constructed. For each node ν in **T** indicating a list construction `/k` is appendes to the label, where `k` is the non-negative integer labeling $\phi(\nu)$; note that ν has only one offspring. **A** is obtained from **T** by the following tree transformation Γ: Γ copies a node to a fresh one with the same label. It maps a tree of the form



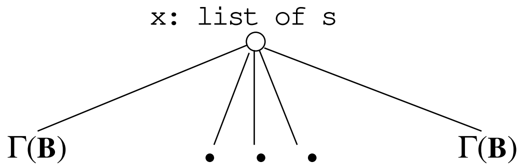
to the tree



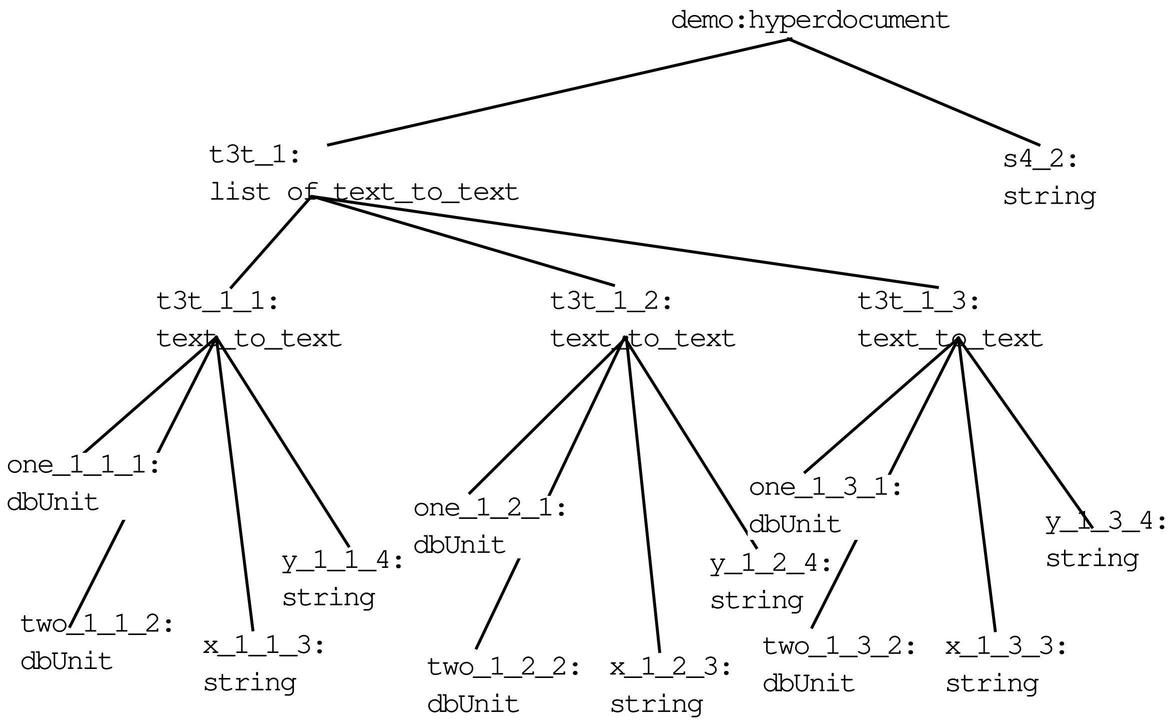
with the root as a fresh node, carrying the same label λ as its inverse image. Finally Γ maps a tree of the form



to the tree (again the root is a new node)



The root of the subtree has now *k* offsprings, in particular it is a leaf if *k = 0*. A breadth first traversal of **A** makes the `documents` names unique. This is be done by relacing `x` by x^d, the latter denoting `x` with the Dewey notation[2] for the node that is labeled by this particular instance of `x` appended.
Continuing the example from above, we the following tree **A** for the hyperdocument `demo` is obtained:



Denote by `VAL(`x^d`)` the `PROLOG` list constructed from *val(x)* in which each parameter name is replaced by its unique mutation.
The stage is thus set up for the next step. Each non-root node ν that is labeled `y: t` is decorated with a label Λ(ν) that is constructed from the constraints section for class `t`. Suppose first that `t` is a primitive type, then ν is a leaf, and the bindings section has associated a value to `y`; we will need the list `VAL(y)` later on. Λ(ν) is empty for the leaf ν, or if `t` is a list type. If ν is not a leaf, and `t` is not a list type, then `t` is a type the parameters of which are indicated by the subtrees of ν. The raw version of Λ(ν) is given by the `constraints` section for `t`, which is manipulated as follows:

2. The Dewey notation assigns unique coordinates to the nodes of a tree. The root is assigned the empty word ε, if a node has the notation δ, then its first offspring is assigned δ_1, its second one δ_2 etc. The Dewey notation is introduced and discussed in [Knu], p. 310.

* the local names are replaced by the lists according to `VAL`: a local name `z` is replaced all over the `constraints` section by `VAL(`z^d`)`,
* the types for these lists are asserted according to their declaration: a declaration

  ```
  z: s
  ```

  (where `s` is not a list type) results in the predicate

  ```
  assert(type(VAL(z^d)), s)).
  ```

  If a declaration reads

  ```
  z: list of s
  ```

  the resulting predicate will read

  ```
  assert(type(VAL(z^d)), listOf(s))).
  ```

* the local predicates are made unique through prefixing them by `y_`

$\Lambda(\nu)$ is obtained from this manipulated constraints section by wrapping it into `startTransaction` and `endTransaction`. These constructs are provided by the system, they are used to signal whether or not the transactions associated with the `constraints` section and the lower level transactions associated with the `constraints` sections for the parameters succeed, cp. [Wol]. If ρ is the root of **A**, then $\Lambda(\rho)$ consists of `startTransaction` and `endTransaction` only (this represents the root transaction).

3.3 Interpreting the Fact Base

The tree **A** is traversed in postorder, when a node ν is visited, $\Lambda(\nu)$ is written out. This eventually results in an executable PROLOG program. The facts asserted by the program's execution are committed to the fact base only if the root transaction succeeds. The fact base generated will be interpreted by suitable tools (which e.g. map abstractly specified links to physical ones). This, however, is left to an implementation.

4. Related Work

The work on HDM (see e.g. [GPS]) stresses the analogy of hypertext construction and software construction. The document architecture proposed in HDM is also a hierarchical one, it encodes certain aspects of the browsing semantics into the document by associating so called *aspects* to the root node of a document. An aspect is propagated down to all components of a document, forcing the document to behave in a particular way. Aspects may be represented in the present proposal through logical disjunction. HDM derives links that are implicitly given in the document structure from the specification of a document. This is modelled explicitly in the present approach. HDM does not separate completely the description of the data from that of the control flow.

HB1 [SLHS] focusses on semantic data modeling for hypermedia systems by specifying the storage and manipulation of the links which together with the data form a hypermedia system. This is done essentially through the use of a special semantic data model akin to an Extended Entity Relation model in which anchors and links have first-class status. Hence the specification given in [SLHS] is closer to an implementation, entities comparable to facts we use here are already encoded in the data model.

Tochtermann [Toc] models essentially the dynamic behavior of a hypermedia system through VDM. The specification given concentrates on an axiomatic description of the interplay between the major structuring components. The specification of documents is not object-oriented, thus classification hierarchies cannot be constructed, the static and the dynamic behavior are not completely separated.

Hamakawa and Rekimoto [HaRe] propose an object-oriented approach to document construction which uses an analogy of TeX's glue to the construction of composite components. The

work focusses on temporal relationships between documents, and it does not separate the specification of a document from its runtime behavior.
Finally, Gibbs and Tsichritzis [GiTs] provide an overview of approaches to the object oriented construction of multimedia systems on the level of system construction (rather than high level specification).

5. Further Work

The static view may be augmented by adding e. g. bidirectional links and other link types (Tochtermann [Toc] provides an overview of the link types discussed in the literature), furthermore it has to be investigated where user annotations fit into this view. Another aspect to be investigated is a graphical way of specifying document classes; here we expect that some lessons can be learned from the graphical presentation used by approaches like Objectory [Jac] or OMT [Rum].
Document descriptions as well as the hyperdocuments generated from them should be persistent. This opens several avenues of work: the data base of descriptions should be kept in a consistent state. Describing consistency will use formalisms from Software Development Environments, e.g. abstract syntax graphs and graph grammars [ELMSS]. Furthermore, we need tools for manipulating the documents, i.e., a Hypermedia Development Environment needs to be constructed. Here also the formalism of graph grammars will be used. Version and configuration control is needed for updating, and constructing hyperdocuments specifically for particular users. The approach outlined in [WaGr] may be helpful here.
The static view has to be complemented by a dynamic one which gives a browsing semantics [FuSt] to the hypermedia system, so that the control flow can be adjusted individually the a particular user or a user group. The browsing semantics given in [FuSt] is provided through colored timed Petri Nets the construction of which is supported by simple visual formalisms. We are currently investigating the specification of browsing semantics through statecharts. A fully dynamic view should also take views [Tom] into account, i.e. customization of the dynamic behavior to the needs of a user.

6. References

[AMC] Agosti, M., Melucci, M., Crestani, F.: *Automatic Authoring and Construction of Hypermedia for Information Retrieval*. Multimedia Systems 3, 1, 1995, 15 - 24

[Bra] Bratko, I.: *PROLOG Programming for Artificial Intelligence*. Addison-Wesley, 1986

[CaGe] Carriero, N., Gelernter, D.: *How to Write Parallel Programs.* The MIT Press, 1990

[ELMSS] Engels, G., Lewerentz, C., Nagl, M., Schäfer, W., Schürr, A.: *Building Integrated Software Development Environments, part I.* ACM Transactions on Software Engineering and Methodology 1, 2, 1992, 135 - 167

[FuSt] Furuta, R., Stotts, P.D.: *Petri-Net-Based Hypertext: Document Structure with Browsing Semantics.* ACM Transactions on Information Systems 7, 1, 1989, 3 - 29

[GPS] Garzotto, F., Paolini, P., Schwabe, D.: *HDM – A Model-Based approach to Hypertext Application Design.* ACM Transactions on Information Systems 11, 1, 1993, 1 - 26

[GJM] Ghezzi, C., Jazayeri, M., Mandrioli, D.: *Fundamentals of Software Engineering.*

Prentice-Hall, 1991

[GiTs] Gibbs, S. J., Tsichritzis, D.C.: *Multimedia Programming – Objects, Environments and Frameworks.* Addison-Wesley, 1995

[Has] Hasselbring, W.: *Prototyping Parallel Algorithms in a Set-oriented Language.* Ph. D. Thesis, Chair for Software Technology, University of Dortmund, 1994

[HaSc] Halasz, F., Schwartz, M.: *The Dexter Hypertext Reference Model.* Communications of the ACM 37, 2, 1994, 30 - 39

[HaRe] Hamakawa, R., Rekimoto, J.: *Object Composition and Playback Models for handling Multimedia Data.* In Garcia-Luna, J.J. and Rangan, P.V., (Eds): ACM Multimedia 93, ACM, 1993, 273 - 281

[Jac] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G.: *Object-Oriented Software Engineering.* Addison-Wesley, 1992

[KLM] Knudsen, J. L., Löfgren, M., Madsen, O. L., Magnusson, B. (Eds.): *Object-Oriented Environments – The Mjølner Approach.* Prentice-Hall, 1993

[Knu] Knuth, D.E.: *The Art of Computer Programming, vol. I.* Addison-Wesely, 1973

[MMN] Madsen, O. L., Møller-Pedersen, B., Nygaard, K.: *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, 1993

[Rum] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: *Object-Oriented modeling and Design.* Prentice-Hall, 1991

[SLHS] Schnase, J. L., Leggett, J. J., Hicks, D. L., Szabo, R. L.: *Sematic Data Modeling of Hypermedia Associations.* ACM Transactions on Information Systems 11, 1, 1993, 27 - 50

[Toc] Tochtermann, K.: *Ein Modell für Hypermedia – Beschreibung und integrierte Formalisierung wesentlicher Hypermediakonzepte.* Ph. D. Thesis, Chair for Automata and Switching Theory, University of Dortmund, 1995

[Tom] Tompa, F. W.: *A Data Model for Flexible Hypertext Database Systems.* ACM Transactions on Information Systems 7, 1, 1989, 85 - 100

[WaGr] Wagner, T., Graham, S.L.: *Integrating Incemental Analysis with Version Management.* In W. Schäfer, P. Botella (Eds.): Proc. 5th ESEC, Sitges, Spain, Sept. 1995. Lecture Notes in Computer Science 989, Springer-Verlag, 1995, 205 - 218

[Wol] Wolf, S.: *Unterstützung kooperativer Softwareentwicklung – ein transaktionsbasierter Ansatz.* Ph. D. Thesis, Chair for Software Technology, University of Dortmund, 1994

A. Appendix: Clauses

This appendix contains descriptions of the `PROLOG` constants and predicates mentioned above. Given a list `li`, `first(F, li)` unifies `F` with the first element of the list, `rest(R, li)` unifies `R` with the list for which `[first(_, li) | R] == li` holds, and `last(LL,li)` unifies `LL` with the last element of `li`. These predicates fail if `li` is the empty list. `length(N, li)` unifies `N` with the length of `li`. The members of a list may be extracted by their position.

A.1 Constants

`string`	indicates a string
`text`, `graphics`	type of a `dbUnit`
`dbUnit`	persistent primitive
`pixel`	indicates a set of pixels

A.2 Predicates

These predicates are built in, their realization is left to an implementation.

`begin(x, B)`	`B` unifies with the begin of `x`
`bindAll(L1, L2)`	fails, if one of the lists `L1` or `L2` is empty; otherwise, it ha the effect of `assert(link(x, y))` for each `x` in `L1` and each `y` in `L2`
`define(a, b)`	definition clause
`end(x, E)`	`E` unifies with the end of `x`
`extType(X, Y)`	`X` unifies with a `dbUnit`, `Y` with the external type of `X`, i.e., `text` or `graphics`
`link(x, y)`	`x` and `y` are positions. The fact indicates the these positions are linked
`occ(a, b, C)`	`a` is a string, `b` a `dbUnit` of type text, `C` gives an ordered list of all occurrences of `a` in `b`, or `a` is a set of pixels in a graphic document `b`, and `C` gives its position
`rect(leftC, b, c)`	specifies a rectangle in a graphics document with origin `left C`, length b and height c
`type(X, Y)`	if `X` is unified with a document, `Y` gives its type

B. Appendix: An Example

The example dealing with document class `book` is continued by adding a table of contents to a book. This is done first statically by taking a list of strings and constructing links between these strings and the corresponding chapters. The second approach is dynamic because a fact `tableOfContents` is generated from the book.

B.1 A Static Table of Contents

We assume that a list of strings is given so that each chapter may be associated with the corresponding string. Hence the number of strings on this list should equal the number of chapters; each string is linked to the beginning of its chapter.

This is done best by defining a specialization of the class `book` (which is possible due to the inner in its `constraints` section):

```
class bookWithTOC is book with
     documents tOC: list of string;
     constraints
          linkStringToChap(X, Y) :-
               begin(Y, B),
               assert(link(X, B)).
          makeTOC([], []).
          makeTOC([X|L], [Y|M]) :-
               linkStringToChap(X, Y),
               makeTOC(L, M).
          -- now, do it
          doIt :-
               -- compare lengths
               length(tOC, LgTOC), length(chapts, LgChap),
               LgTOC = LgChap,
               -- link them
               makeTOC(tOC, chapts).
end bookWithTOC;
```

Similarly, sections and subsections may be linked to strings. This static approach is appropriate when a list of header strings is constructed e.g. by a tool like grep from the first lines of each chapter.

B.2 Generating a Dynamic Table of Contents

The second approach assumes that each subsection, section and chapter is augmented by a header string that is provided as an additional parameter to the corresponding document. These strings are then dynamically collected into a list which in turn is `assert`ed as the table of contents for the book.

This approach is modelled through first specializing the generic class `listClass` by adding a header string:

```
generic class listClassWithHeader[T] is listClass[T] with
     documents str: string;
     constraints
          linkString :-
               begin(x, B), assert(link(str, B)).
          inner;
end listClassWithHeader;
```

The class `paragraph` is defined as above, subsections are specialized, however, so that they have a small table of contents:

```
class subsectionWithHeader is listClassWithHeader[paragraph] with
     constraints
          assert(tableOfContents(self, str)).
end subsectionWithHeader;
```

The class `sectionWithHeader` collects the table of contents from the corresponding subsections into a list, erasing these facts as it goes:

```
class sectionWithHeader is listClassWithHeader[subsectionWithHeader] with
     constraints
          makeTocAux([], []).
          makeTOCAux([X|L], [Y|M]) :-
               type(X, subsectionWithHeader),
               retract(tableOfContents(X, Y)),
               makeTOCAux(L, M).
          makeTOC :-
               makeTOCAux(self.x, L),
               assert(tableOfContents(self, L)).
end sectionWithHeader;
```

A similar approach is chosen for the class `chapterWithHeader`:

```
class chapterWithHeader is listClassWithHeader[sectionWithHeader] with
     constraints
          makeTOCAux([], [], []).
          makeTOCAux([X|L], [Y|M], N) :-
               type(X, sectionWithHeader),
               retract(tableOfContents(X, Y)),
               makeTOCAux(L, M, Q),
               concatenate(Y, Q, N).
          makeTOC :-
               makeTOCAux(self.x, _, L),
               assert(tableOfContents(self, L)).
end chapterWithHeader;
```

Note that a section constructs a list of strings as a table of contents; all these lists are concatenated to yield the table of contents for the chapter.

The definition of class `bookWithTableOfContents` may now proceed as above, except that the declaration

```
chapts: list of chapter
```

should be replaced by

```
chapts: list of chapterWithHeader,
```

and that predicates serving the very same function as `makeTOCAux`, and `makeTOC`, resp., should be added.

An instance of the latter class will enjoy a dynamically generated table of contents. For example, the set of facts for a document

```
winnie-the-Pooh: bookWithTableOfContents
```

will contain

```
tableOfContents(winnie-the-Pooh, L)
```

with `L` as a list of strings (the first one being „`CHAPTER I In Which We Are Introduced to Winnie-the-Pooh and Some Bees, and the Stories Begin`"), each of which is linked to the very first string in a subdocument.