An Algebraic Dexter-Based Hypertext Reference Model

Volker Mattick, Claus-Peter Wirth

volker.mattick@cs.tu-dortmund.de,
wirth@logic.at
http://ls1-www.cs.uni-dortmund.de/cms/mattick.html
http://www.ags.uni-sb.de/~cp

Research Report 719/1999
http://www.ags.uni-sb.de/ cp/p/gr719

November 6, 1999

Universität Dortmund
Fakultät für Informatik
44227 Dortmund
Germany

Abstract: We present the first formal algebraic specification of a hypertext reference model. It is based on the well-known Dexter Hypertext Reference Model and includes modifications with respect to the development of hypertext since the WWW came up. Our hypertext model was developed as a product model with the aim to automatically support the design process and is extended to a model of hypertext-systems in order to be able to describe the state transitions in this process. While the specification should be easy to read for non-experts in algebraic specification, it guarantees a unique understanding and enables a close connection to logic-based development and verification.

Contents

1 Introduction

The number of hypertext applications is growing. What started as an idea of Vannevar Bush more than half a century ago (cf. [Bus 45]) has now become one of the most rapidly growing fields in software engineering. The reason for this rapid development is the World-Wide Web (*WWW*).

Nearly all of the web sites used nowadays are hypermedia applications and only a few are mere hypertexts. In this paper we will refer the term*hypermedia* to a combination of*hypertext* and*multimedia*, as suggested e.g. in [HBR 94]. If the textual or multimedial nature is not relevant, we will speak of*hyperdocuments*. As hypermedia is an open approach, there are infinitely many different types of media-objects in principle. In a closed reference model, these different types of media-objects can only be modeled with an abstract interface. Therefore, it seems to be justified to speak of a*hypertext* reference model even for models of hypermedia like the one we are going to specify in this paper.

Our hypertext reference model is*Dexter-based* because it deviates from the Dexter Hypertext Reference Model (cf. [HS 90]) only in some aspects that had to be corrected in order to be compatible with the WWW. A detailed comparison with the Dexter model, however, is not subject of this paper.

The hypertext model (cf. § 2) was developed as a product model with the aim to support the design of the product "hyperdocument" automatically. It is extended to a model of hypertext-systems (cf. § 3) in order to describe the state transitions of the design-process.

To our knowledge, our hypertext reference model is the first[1]*formal algebraic* modeling approach for hypertexts, hypermedia, or hypertext-systems. Algebraic specification came up in the seventies based on concepts of universal algebra and abstract datatypes. Due to the technical complexity of the subject, it is still an area of ongoing research on the one hand. On the other hand, there is still a gap between what practice demands and what theory delivers. One motivation for our work is to make this gap a little smaller and we hope that our specification is quite readable for non-experts in algebraic specification. Due to its origin, algebraic specification is superior to other specification formalisms in its clear relation to logic and semantics that guarantees a unique understanding and enables a close connection to logic-based development and verification.

1.1 Product Models for Hyperdocuments

In the domain of hyperdocuments there are three fundamental different kinds of product models (cf. [LH 99, p. 221 ff.]). Programming language based, information-centered and screen-based models. The programming language based approach, which applies any general purpose programming language starting from scratch, was used in former days due to the lack of any other sophisticated models, and has nearly no importance in the presence.

For a long time the*information-centered model* has dominated. The most popular product model for hyperdocuments, the "Dexter Hypertext Reference Model" [HS 90], is information-centered. Dexter or one of its modifications, e.g. [GT 94] or [OE 95], describe the structure of a hyperdocument, divided into its logical structure, its linkage, and its

[1] Note that we do not consider Z to be a formal algebraic specification language.

style. A hyperdocument can import components from a "within-component layer" via an anchor mechanism and specify how the document should be presented in a "presentation specification".

Similar ideas are presented in an object-oriented style in the so-called "Tower Model", cf. [BH 92]. Additionally a hierarchization is added. It is described that components could include other components. But no restrictions on how to compose hyperdocuments are mentioned. Thus, you can produce a lot of components not used in any actual hypermedia system.

In both models there is no possibility to describe strategies how to navigate through a set of hyperdocuments. But this is a design goal of increasing importance in the rapidly growing world of hypermedia. The Dexter-based reference model for adaptive hypermedia *(AHAM)* (cf. [BHW 99]) describes first steps towards this direction.

Even the wide-spread Hypertext Markup Language (*HTML*) has obviously its roots in the information-centered paradigm, even though many designers use it in an other way, namely as a screen-based design language. "Screen-based" means that the focus is not the logical structure of the document, enriched with some display attributes, but the display of the document itself. With the upcoming of the WWW and the WYSIWYG-editors the*screen-based model* became more important in hyperdocument design, because sometimes it is easier to think in terms of the produced view on the screen. As far as we know, there are only two models for this approach: The Document Object Model (*DOM*, cf. [W3C 98b]) and the Document Presentation Language (P Language) of*THOT* ([Qui 97]). The goal of DOM is to define an application programming interface for XML and HTML. Thus it is limited to the features used in that languages. The P Language of THOT, used by the W3C-test-bed client browser Amaya ([GQV 98]), is more general, but lacks device-independence; i.e. the presentation only describes a function of the structure of the documents and the image that would be produced on an idealized device.

1.2 Semantics for Hyperdocument Models

All hyperdocument models have in common that no explicit semantics is given. Some information-centered models try to treat the structural part of a hyperdocument as a data type and assign a semantics, but no semantics for the attributes is given. E.g., DOM reduces the DOM-semantics to the semantics of HTML, but up to now there is no unique semantics for HTML, but only device- and browser-dependent semantics.

But there are two widely accepted device-independent description formalisms for documents: The postscript- and the PDF-format ([BCM 96]). Postscript is very mighty but lacks the hyperlinks. Hence, we will use PDF as a screen-based model for hyperdocuments.

Both kinds of models, the screen-based and the information-centered, have in common that they abstract from the contents to be displayed. In practice the gap between both is bridged by a user agent, often called*browser*, cf. Fig. 1 on the facing page. A browser is a mapping between the syntax of the information-centered model of hyperdocuments and the semantics of the screen-based model. It should be equal to the concatenation of the translation (alg2pdf) from the algebraic signature of hyperdocuments into the language of PDF and a display mapping ($[\,]_{\mathrm{PDF}}$) assigning the semantics to the screen-based PDF-

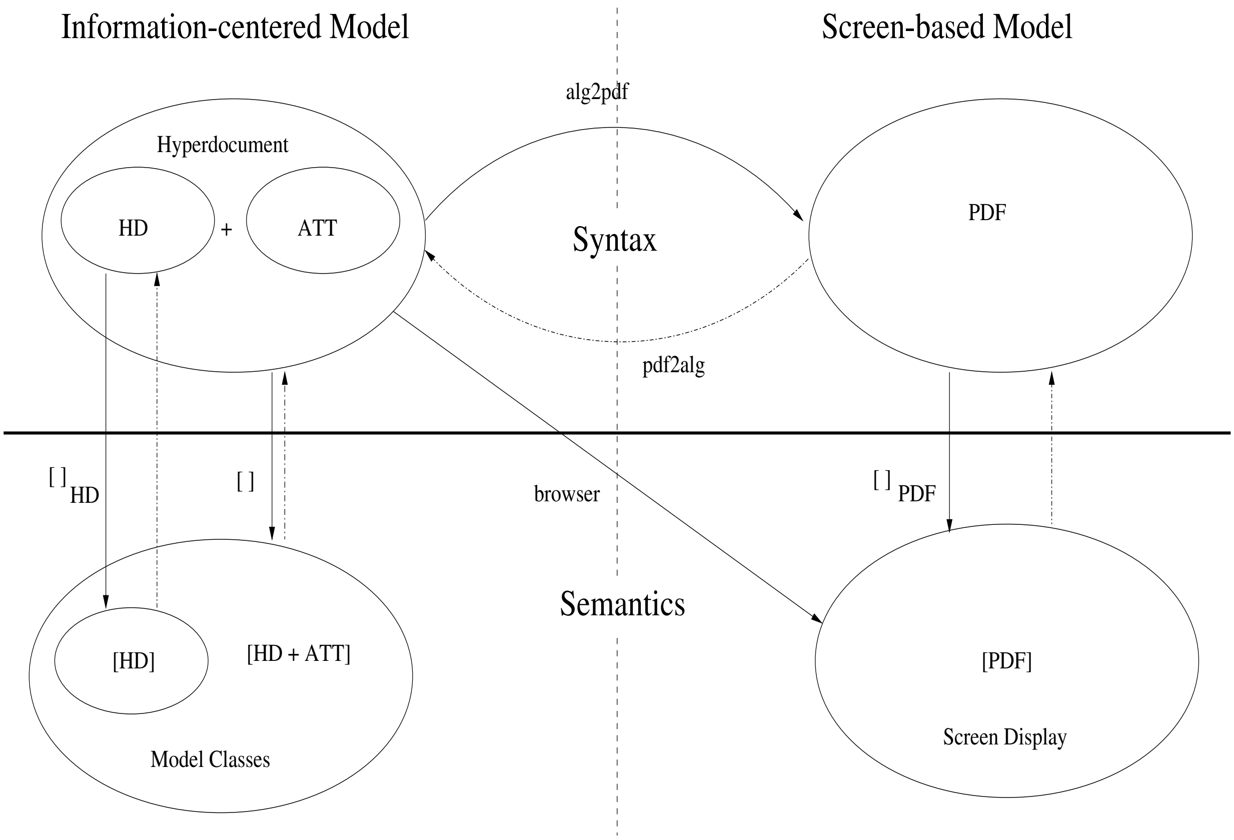

Figure 1: Browser

model. Thus, the semantics of an information-centered description of hyperdocuments is defined in terms of the semantics of a well-known description language for documents. Up to now it is an enormous problem both for browser developers and for designers that there is no unique meaning for a hyperdocument, but only meanings together with particular browsers and output devices. Note that the lower left corner of Fig. 1 denotes some model class providing the algebraic or logic semantics of the information-centered model.

Another problem with the existing models is that they do not reflect the actual state of hypertext technology. Two of the three information-centered models mentioned above come from the "pre-WWW" times.

Therefore we will not formalize the models as they are, but use their crucial ideas, add some new ones coming up with the WWW and structure a document in analogy to classical linear text. We will describe all this in an algebraic specification language (cf. [Pad 2000]) enriched with a modularity concept, cf. ASF$^+$ ([LW 94]) or cTLA ([MK 95]).

This paper deals mainly with the upper left part of Figure 1 and the relation to its neighbors. The formalization of the screen-based model as well as the formal description of the browser defining mapping will be left to another paper. In § 2 we start with the formal description of the general hyperdocument data-structure and identify different hierarchy levels, similar to the levels in linear texts. In § 3 the extension to a model for hypertext-systems is presented.

2 The Information-Centered Model for Hypermedia

> "It is essential to have a solid understanding of the kinds of information present during a design process before the design process itself can be studied." [Sal 96].

The*product model* (sometimes called "object model") is a formal representation of exactly the above-mentioned kinds of information. For our description formalism we choose the constructor-based algebraic approach (cf. [Pad 2000], [KW 96]). In section 2.1 we describe how that formalism can be transferred to our domain. In section 2.2 we develop our product model for hypermedia documents and compare it with existing reference models, like the Dexter Model [HS 90] or the Tower Model [BH 92], and with certain standards as, e.g., those used by the World Wide Web Consortium (W3C) for defining XML [W3C 98c].

2.1 Algebraic Specifications for Describing Product Models

In classical first-order algebraic specifications, the world is represented with the help of a signature. A*signature* $\mathrm{sig} = (\mathbb{F}, \alpha)$ consists of an (enumerable) set of function symbols $\mathbb{F}$ and a (computable) arity function $\alpha : \mathbb{F} \to \mathbf{N}$, saying that each function symbol $f \in \mathbb{F}$ takes $\alpha(f)$ arguments. A corresponding sig-algebra (or sig-structure) consists of a single homogeneous universe (or carrier) and, for each function symbol in $\mathbb{F}$, a total function on this universe.

Heterogeneous, however, is the world we have to model.[2] We have at least three different sorts of objects: *anchors* (cf. § 2.4), *links* (2.5) and *documents* (2.7). Therefore, an adequate structural representation should contain different universes for different sorts. This leads us to the following refinement of the notion of a signature.

A*many-sorted signature* $\mathrm{sig} = (\mathbb{S}, \mathbb{F}, \alpha)$ consists of a (finite) set of sorts $\mathbb{S}$, an (enumerable) set of function symbols $\mathbb{F}$ and a (computable) arity function $\alpha : \mathbb{F} \to \mathbb{S}^{+}$, saying that each function symbol $f \in \mathbb{F}$ with $\alpha(f) = s_1 \ldots s_n s'$ takes n arguments of the sorts $s_1, \ldots, s_n$ and produces a term of sort s'. A corresponding sig-algebra $\mathcal{A}$ consists of a separate universe $\mathcal{A}_s$ for each sort $s \in \mathbb{S}$ and, for each function symbol $f \in \mathbb{F}$ with $\alpha(f) = s_1 \ldots s_n s'$, a total function $f^{\mathcal{A}} : \mathcal{A}_{s_0} \times \ldots \times \mathcal{A}_{s_n} \to \mathcal{A}_{s'}$.

Typically, certain function symbols are called "constructors" because they construct the data domains (or domains of discourse) of an algebra. More precisely, the constructor (ground) terms[3] are used for designating the data items of an algebra*completely* and*uniquely*; the popular catchwords being*no junk* and*no confusion*, resp.. E.g., zero '0' and successor 's' may construct the sort of natural numbers 'nat', 'nil' and 'cons' the lists, 'true' and 'false' the Boolean sort, &c.. For the sort 'nat' of natural numbers, each data item of the sort 'nat' is to be denoted by some constructor term of the sort 'nat' (no junk), and two different constructor terms of the sort 'nat' describe two different data objects (no confusion). Note that the latter is special for constructor terms: E.g., for a non-constructor function symbol '+', the terms $\mathrm{s}(0)+0$, $0+\mathrm{s}(0)$, and $\mathrm{s}(0)$ may well denote the same data object, but only the last one is a constructor term.

[2] For a more detailed discussion cf. [LP 92].

[3] I.e. the well-sorted terms built-up solely from constructor function symbols.

Since we are strongly convinced that the notion of a "constructor function symbol" must be based on the signature only (and not on the axioms of a specification), this leads us to the following refinement of the notion of a many-sorted signature.

sig' is a *subsignature* of sig if sig' and sig are many-sorted signatures and, for $(\mathbb{S}',\mathbb{F}',\alpha') := \mathrm{sig}'$ and $(\mathbb{S},\mathbb{F},\alpha) := \mathrm{sig}$, we have $\mathbb{S}'\subseteq\mathbb{S}$, $\mathbb{F}'\subseteq\mathbb{F}$, and $\alpha'\subseteq\alpha$. The sig'*-reduct* of a sig-algebra $\mathcal{A}$ consists only of the universes for the sorts of $\mathbb{S}'$ and of the functions for the symbols in $\mathbb{F}'$. Formally, when a sig-algebra $\mathcal{A}$ is seen as a total function with domain $\mathbb{S}\uplus\mathbb{F}$,[4] the sig'-reduct can be seen as the restriction of $\mathcal{A}$ to the domain $\mathbb{S}'\uplus\mathbb{F}'$, which we generally denote in the form ${}_{\mathbb{S}'\uplus\mathbb{F}'}{\upharpoonleft}\mathcal{A}$. For a subset $\mathbb{C} \subseteq \mathbb{F}$ we denote with $\mathrm{sig}^{\mathbb{C}}$ the subsignature $(\mathbb{S},\mathbb{C},{}_{\mathbb{C}}{\upharpoonleft}\alpha)$ of sig.[5] If the function $\mathcal{B}$ that differs from the sig-algebra $\mathcal{A}$ only in that the universe of each sort $s\in\mathbb{S}$ contains only the values of the $\mathbb{C}$-terms of the sort s under the evaluation function of $\mathcal{A}$, is a sig-algebra again, then we call $\mathcal{B}$ the $\mathbb{C}$*-generated subalgebra* of $\mathcal{A}$. We call $\mathbb{C}$ a set of *constructors for* sig if $\mathbb{C} \subseteq \mathbb{F}$ and the signature $\mathrm{sig}^{\mathbb{C}}$ is *sensible* (or "inhabited"), i.e., for each $s\in\mathbb{S}$, there is at least one constructor ground term of sort s.

Definition 2.1 (Data Reduct)
If $\mathbb{C}$ is a set of constructors for sig, then, for each sig-algebra $\mathcal{A}$, the $\mathbb{C}$-generated subalgebra of the $\mathrm{sig}^{\mathbb{C}}$-reduct of $\mathcal{A}$ is a $\mathrm{sig}^{\mathbb{C}}$-algebra, which is called the $\mathbb{C}$*-data reduct of* $\mathcal{A}$.

A *constructor-based specification* $\mathrm{spec} = (\mathrm{sig},\mathbb{C},\mathcal{AX})$ is composed of a set of constructors $\mathbb{C}$ of the signature sig and of a set $\mathcal{AX}$ of axioms (over sig).

Definition 2.2 (Data Model)
Let $\mathrm{spec} = (\mathrm{sig},\mathbb{C},\mathcal{AX})$ be a constructor-based specification.
$\mathcal{A}$ is a *data model* of 'spec' if $\mathcal{AX}$ is valid in the sig-algebra $\mathcal{A}$ and the $\mathbb{C}$-data reduct of $\mathcal{A}$ is isomorphic to the term algebra over $\mathrm{sig}^{\mathbb{C}}$.[6]

Note that the latter is just a formal way to express the catchword "no confusion" from above. The catchword "no junk" can formally be realized by variables ranging only over the constructor ground terms or the $\mathbb{C}$-data reduct of $\mathcal{A}$. For technical details cf. [KW 96].

Let $\mathbb{N} := \mathbb{F}\backslash\mathbb{C}$ denote the set of *non-constructor* (or *defined*) function symbols. Note that by Definition 2.2, the data reduct of data models of a consistent specification 'spec' is uniquely defined (up to isomorphism) as the constructor ground term algebra. Data models for 'spec' may differ, however, in the way partially specified functions from $\mathbb{N}$ behave in the unspecified cases. E.g., suppose that the operator '-' is partially specified on 'nat' by the two equations $x-0=x$ and $\mathrm{s}(x)-\mathrm{s}(y)=x-y$. In this case, data models may differ on the evaluation of the term $0-\mathrm{s}(0)$, which may evaluate to different values of the $\mathbb{C}$-data reduct or even to different "junk" or "error" values. Note that in this way we can model partial functions with total algebras.

This possibility to model partiality is also the reason why we prefer characteristic functions (i.e. functions of Boolean sort) to predicates: the result of the application of a characteristic function can be true, false or possibility neither true nor false (undefined, unspecified). With predicates we do not have the latter possibility.

[4]We use '$\uplus$' for the disjoint union of classes.
[5]Note that ${}_{\mathbb{C}}{\upharpoonleft}\alpha$ denotes the restriction of the function α to the domain $\mathbb{C}$.
[6]I.e. isomorphic to the initial $\mathrm{sig}^{\mathbb{C}}$-algebra.

The constructor ground terms of the sorts of some subset $\mathbb{S}_{\mathrm{P}} \subseteq \mathbb{S}$ will be used to describe the fixed unchanging parts of a product. The constructor ground terms of the remaining sorts in $\mathbb{S}\backslash\mathbb{S}_{\mathrm{P}}$ statically describe the dynamic states of the product without its dynamic behavior. The dynamic functions from $\mathbb{N}$ will change the static description of the product w.r.t. the constructor ground terms of these sorts. As we do not have final algebra domains or state sorts in our application by now, we have not treated these subjects explicitly here.

It is useful to further classify the function symbols from $\mathbb{N}$. E.g., functions that inspect a data item may be called "selectors" or "observers", functions that manipulate may be called "mutators" or "editors", &c.. More important here is the classification of a function symbol as belonging to the product of the design process; contrary to functions for the design process itself, auxiliary functions for the implementation, &c.. Thus, let $\mathbb{P} \subseteq \mathbb{N}$ be a set of *product function symbols*. $(\mathrm{sig}, \mathbb{C}, \mathbb{S}_{\mathrm{P}}, \mathbb{P}, \mathcal{AX})$ is a *product specification* if $(\mathrm{sig}, \mathbb{C}, \mathcal{AX})$ is a constructor-based specification, $\mathbb{S}_{\mathrm{P}} \subseteq \mathbb{S}$ is non-empty and $\mathbb{P}\subseteq\mathbb{N}$, for $(\mathbb{S}, \mathbb{F}, \alpha) := \mathrm{sig}$ and $\mathbb{N} := \mathbb{F}\backslash\mathbb{C}$.

Definition 2.3 (Product Model)
Let $\mathrm{sig} = (\mathbb{S}, \mathbb{F}, \alpha)$ be a many-sorted signature.
Let $\mathrm{spec} = (\mathrm{sig}, \mathbb{C}, \mathbb{S}_{\mathrm{P}}, \mathbb{P}, \mathcal{AX})$ be a product specification.
Let C be the set of those function symbols $c \in \mathbb{C}$ whose argument and result sorts in $\alpha(c)$ do all belong to $\mathbb{S}_{\mathrm{P}}$.
A *structural product model* of 'spec' is the $(\mathbb{S}_{\mathrm{P}}, C, {}_{C}\!\upharpoonright\!\alpha)$-reduct of the $\mathbb{C}$-data reduct of a data model of $(\mathrm{sig}, \mathbb{C}, \mathcal{AX})$.
A *behavioral product model* of 'spec' is the $\mathrm{sig}^{\mathbb{C}\cup\mathbb{P}}$-reduct of a data model of $(\mathrm{sig}, \mathbb{C}, \mathcal{AX})$.

Note that by this definition, a structural product model of a consistent specification is uniquely defined (up to isomorphism). Behavioral product models, however, may differ in the way partially specified functions from $\mathbb{P}$ behave in the unspecified cases.

The present situation of our application is not very complicated because at first only the structural product model is of interest. Moreover, since $\mathbb{S}_{\mathrm{P}} = \mathbb{S}$, the $(\mathbb{S}_{\mathrm{P}}, C, {}_{C}\!\upharpoonright\!\alpha)$-reduct of the $\mathbb{C}$-data reduct is the $\mathbb{C}$-data reduct itself. Therefore, the whole universe of discourse, namely all possible descriptions of products, can and will be represented by constructor ground terms. To simplify the description of the structural product model we use some predefined data types, like 'nat' and 'bool', some of them generic, like 'set', 'function', 'list', and 'tree'. For the understanding of the product model, it suffices to assume that these data types do what their mathematical counterparts do. For a deeper understanding a detailed description can be found in [Pad 2000]. For the presentation of our specification we use the fairly intuitively readable style from [Pad 2000].[7] The only further remark that may be necessary here is the way the structured specification is meant to be put together: The union of two specifications is the element-wise non-disjoint union of sort symbols, function symbols, arity functions, constructors symbols, and axioms. When parameters of a specification are bound to some actual name of a specification, we take the union of both specifications and replace the parameter with the actual name everywhere. Although this approach is not perfect,[8] we have chosen it for its simplicity, power and conciseness.

[7] This style is constantly improved. Thus, there can be little differences in the notation, which should not disturb the understanding of the presented specifications.

[8] E.g., the approach is error-prone and does not provide any proper modularization, i.e. the specification can only be checked or properly understood as a whole.

2.2 The Object under Consideration: Hyperdocuments

In the domain of hyperdocuments there are three fundamental different kinds of product models (cf. [LH 99, p. 221 ff.]): *Programming language based*, *information-centered* and *screen-based* models. The programming language based approach, which applies any general purpose programming language starting from scratch, was used in former days due to the lack of any other sophisticated models, and has nearly no importance in the presence.

For a long time the *information-centered model* has dominated. The most popular model for hyperdocuments, the "Dexter Hypertext Reference Model" [HS 90], is information-centered. Dexter or one of its modifications, e.g. [GT 94] or [OE 95], describe the structure of a hyperdocument, divided into its logical structure, its linkage, and its style. A hyperdocument can import components from a "within-component layer" via an anchor mechanism and specify how the document should be presented by a "presentation specification".

Similar ideas are presented in an object-oriented style in the "Tower Model", cf. [BH 92]. Additionally a hierarchy is added. It is described that components could include other components. But there are not mentioned any restrictions how to compose hyperdocuments. So you can produce a lot of components not used in any actual hypermedia system. In both models there is no possibility to describe strategies how to navigate through a set of hyperdocuments. But this is a design goal of increasing importance in the rapidly growing world of hypermedia. The Dexter-based reference model for adaptive hypermedia *(AHAM)* (cf. [BHW 99]) describes first steps toward this direction.

Even the wide-spread Hypertext Markup Language (*HTML*) has obviously its roots in the information-centered paradigm, even though many designers use it in another way, namely as a screen-based design language. "Screen-based" means that the focus is not the logical structure of the document, enriched with some display attributes, but the display of the document itself.

With the upcoming of the WWW and the WYSIWYG-editors the *screen-based model* became more important in hyperdocument design, because sometimes it is easier to think in terms of the produced view on the screen.

A simple and common characterization of our object under consideration is:

Definition 2.4 (Informal Description of a Hyperdocument)
A *hyperdocument* is a *basis document*, sometimes called *lineardocument*, consisting of a fixed set of basic contents, organized according to a *media structure*, enriched with a pointer concept, called *anchors*, to access a specific content inside the document, and a reference concept, called *hyperlinks*, to access another document by its *address*. If the only medium in a hyperdocument is text, then we speak of a *hypertext document*, or else of a *hypermedia document*.
Moreover, device independence is often formulated as a hypermedia requirement. This is only possible if you disjoin the structural description and the *presentation attributes*, as e.g. in HTML or $\TeX$.

In the following sections, we will examine how the five crucial elements of a hyperdocument,

- the basis document (2.3),
- the set of anchors (2.4),
- the set of links (2.5),
- the presentation attributes[9] and
- the addresses (2.6)

can be specified.

2.3 Basis Documents

According to the Dexter Model the structure of the basis is not known. It is only assumed, that each basis element has a fixed set of properties (which can be observed by some special observer functions, which are not part of the product model) and a particular structure, which can be accessed by the anchor-mechanism via a *location*. Accordingly we model basis documents as a parameter.

Definition 2.5 (Parameter "Basis Document")

DOCUMENT_P[document,location] =
`sorts` document
location

Note that in the boxes like the one above we do not present the full specification (cf. § A) but only an essential part of it that should be easy to understand.

2.4 Anchors

Originally a hyperdocument used to have no layout at all. It was seen only as an arbitrary collection of atomic basis elements. The*anchor* was the only possibility to get access to one of these atoms. It had a name and a method which could be interpreted by the underlying database. When hypertext evolved, more complex construction mechanisms came up and the need to control the layout became more important. The anchor-method depended no longer only on the data base but also on the document structure. This method to access an element at a given position is a bit confusingly named *location*. We adopt that name, because it is used in most hypermedia models.

In contrast to Dexter, our anchors are enriched with an anchor-type. So you may not only mark a special element, but you also mark it as a possible start-point (*source*) or end-point (*target*) of a link or both (*label*). Note that in our specification the anchor-types are

[9]Because of the fact that presentation attributes are meaningful only in connection with a screen-based-model, we leave them undefined at the moment. They will be added later.

part of anchors and therefore*local* to the hyperdocuments. In Dexter this feature is included into the*global* specifier-mechanism of hyperlinks, however (cf. § 2.5). In pre-WWW times, where hypertext was usually a non-distributed system, this made no difference. But in a distributed system like the WWW it becomes important that the anchor types can be found without searching the whole WWW and must therefore be stored local to the document they are related to.

Considering all these facts and adding the attributes, as discussed previously, we come to the following specification for anchors:

Definition 2.6 (Structural Product Model "Anchor")

ANCHOR[location] = DOCUMENT_P[document,location] `and` ATT_ANCHOR `then`
`vissorts`
 anchor_type
 anchor = anchor(location)
`constructs`
 Source, Target, Label : anchor_type
 Mkanchor. location $\times$ anchor_type $\times$ att_anchor $\rightarrow$ anchor

2.5 Hyperlinks

A *hyperlink* (or*link* for short) is a reference from a fixed set of contents (*source*) to a fixed set of contents (*target*). Each of these sets of contents are described by a set of *specifiers*. Our model differs from the Dexter Model insofar as no links to links are possible. But our view is compatible to most other hypermedia models. A specifier consists of a global address of sort 'uri' and a local name of sort 'anchor_id'. 'uri' is the abbreviation for "Unified Resource Identifier"[10] [BFI 98]. The anchor-name is to be mapped to an anchor of the hyperdocument under the global address. This mapping is not global but part of the hyperdocument. In the Dexter Model, specifiers have also a direction. We split this direction into the anchor_type and the link_type. Hence we get uni- and bi-directional links.

Moreover links are classified according their intended behavior. This idea goes back to [Eng 83], where *jump-* and *include*-links were introduced. Often the term "jump-link" is used synonymous with link at all. It denotes that kind of link where the system is waiting for a user action (e.g. a mouse-click) and then the old source-document is replaced by the new target-document. The term "include-link" denotes a class of links which are to be automatically evaluated and presented inside a previously defined location. These "traditional" kinds of links do not suffice since systems work with multiple windows. A third kind of link is necessary, namely one that can open new windows to present the target-document and leave the source-document untouched in its old place.

This kind of presentational behavior is represented in the*show-type*, as we will call it according to [W3C 98d]. Links of show-type 'Embed' embed their target into the context of their source. Links of show-type 'Replace' replace the hyperdocument of their source

[10] The well-known URLs in the WWW are a subset of URIs.

with the hyperdocument of their target. Finally, links of show-type 'New_window' open a new window with the document of their target.

The second distinction is whether a user interaction is required or not. This is represented by the*actuate-type*, as we will call it according to [W3C 98d]. Links of actuate-type 'User' are followed upon user interaction. Links of actuate-type 'Auto' are followed automatically.

If we combine all the named possibilities we get twelve different types of links. But, what sense makes e.g. a bi-directional link of show-type 'Embed'? Or a bi-directional link of actuate-type 'Auto'? We think that the only meaningful bi-directional links are of show-type 'Replace' and of actuate-type 'User'. Therefore, uni-directional links ('Uni$(*, *)$') are modeled with two parameters (show-type, actuate-type), but no arguments are given to the bi-directional links ('Bi').

The previously mentioned jump-link has the type Uni(Replace, User) and the include-link Uni(Embed, Auto).

Definition 2.7 (Structural Product Model "Links")

LINK = ANCHOR_ID `and` URI `and` ATT_LINK `and` SET[entry$\mapsto$specifier] `then`
`vissorts`
link_type
show
actuate
specifier
link

`constructs`
Embed, Replace, New_window : show
User, Auto : actuate
Uni. show $\times$ actuate $\rightarrow$ link_type
Bi : link_type
Mkspecifier. uri $\times$ anchor_id $\rightarrow$ specifier
Mklink. set(specifier) $\times$ set(specifier) $\times$ link_type $\times$ att_link $\rightarrow$ link

The generic abstract data type 'set' in Definition 2.7 is assumed to be predefined, cf. p. 25 for its signature.

2.6 Addresses

In order to be referenced, each hyperdocument must have an address. In general, this address space is described by the already described sort 'uri'. But we will allow to define special address subspaces for local addresses where the type of a hyperdocument can be inferred from the type of its address. Thus, we have a second parameter.

Definition 2.8 (Parameter "Addresses")

ADDR_P[addr] =
`sorts` addr

2.7 Hyperdocuments

We have now modeled all parts of our product, but as often, the product is more than the sum of its parts. It is not very convenient to access specific parts of the basis via a possibly cryptic location-description. That is the reason why hyperlinks deal only with anchor-names, instead of their values. Therefore each anchor, if it is used in a document, must be combined with a name. We model this by using a function, thereby ensuring that no anchor name can be used twice inside the same document. We get a product model for a class of hyperdocuments that vary in the underlying documents and the address space. These open parameters will be instantiated in the following section.

Definition 2.9 (Structural Product Model "Hyperdocuments")

HD[document,location,addr] = DOCUMENT_P[document,location] `and`
ADDR_P[addr] `and`
ANCHOR[location] `and`
LINK `and`
ATT_HD `and`
FUNCTION[domain↦anchor_id,range↦anchor] `and`
SET[entry↦link]

`then`
`vissorts`
hd = hd(document, location, addr)
`constructs`
Mkhd. document × function(anchor_id,anchor) × set(link) × att_hd × addr → hd

2.8 The Hierarchy of Hyperdocuments

Most hypermedia models end here with the definition of hyperdocuments. Some of these models give no further information about the structuring of hyperdocuments at all, others define new kinds of objects, e.g. views. We suggest another approach, based on the classical organization of texts. They are structured by a hierarchy of at least three levels, shown in the left column of Table 1.

Linear Text	Hyperdocument	
Book	Site	(Section 2.8.2)
Chapter[11]	Frameset-Document	(Section 2.8.3)
Page	Hypermedia-Document	(Section 2.8.4)

Table 1: The Levels of a Document

The only basic element of a linear text is the character. Together with the media-structures like paragraphs, tables or lists, they build the structured basis for documents. Arranging these structured elements sequentially leads to a page. Now you have the possibility to combine pages into a document of a higher level. We believe that this hierarchy is a good strategy to organize hyperdocuments as well, because these levels can also be found, when you examine the most popular application for hyperdocuments, the WWW, and the wide spread Hypertext Markup Language ([W3C 98a]) or some of its relatives out of the SGML-family[12]. The right column of Table 1 shows the hypermedial counterpart in terms of the most prominent hypertext application, the WWW.

Thus, we will define three typical levels of hierarchy for a hyperdocument. These levels belong to the "storage layer" in the Dexter Model, cf. Table 2. The media-objects belong to the "within-component layer" of the Dexter-Model. This is not the focus of our work and it will not be viewed in detail.

Dexter Model	**Our Product Model**
Run-time Layer	—
Presentation Specifications	*Attributes*
Storage Layer	Site Frameset-Document Hypermedia-Document
Anchoring	*Anchor*
Within-Component Layer	Media-Object

Table 2: Comparison with the Dexter Model (Interfaces in italics)

[11] *Wall news sheet* may be intuitionally closer to "frameset document" because it describes a multi-dimensional combination of pages.

[12] SGML is the Structured Generalized Markup Language (ISO-Norm 8779)

2.8.1 Media-Objects

Media-objects are not hyperdocuments. They only provide the interface to the Within-Component-Layer in the Dexter Model. As hypermedia is an open approach, there are infinitely many different types of media-objects in principle.

Our interface to media-objects is quite simple because we are not interested in modeling their internal behavior. The only thing we require is that they have some unified resource identifier of sort 'uri' and a set of anchor identifiers to which links may refer. Thus, a media-object basically introduces a legal set of specifiers referring to it.

Definition 2.10 (Structural Product Model "Media-Objects")

MO = URI `and` ANCHOR_ID `and` SET[entry$\mapsto$anchor_id] `then`
`vissorts`
 $\mathrm{mo} = \mathrm{mo}(\mathrm{uri}, \mathrm{anchor\_id})$
`constructs`
 Mkmo. $\mathrm{uri} \times \mathrm{set}(\mathrm{anchor\_id}) \rightarrow \mathrm{mo}$

2.8.2 Pages and Hypermedia-Documents

Pages are at the lowest level in the hierarchy. As mentioned before the basic contents, represented by the media-objects, is hierarchically structured. Some models (cf. e.g. [Dob 96]) introduce a sub-document relation for this purpose, which only describes which document is part of another. The way in that they are related is left to the presentation attributes. This strategy is adequate to examine the navigational structure of a document, but it is not sufficient to describe "real-world" hyperdocuments. We believe that presentation attributes must be reserved for simple lay-out purposes only, and that a change of presentation attributes must not change the document in a fundamental way. E.g., if you re-arrange a table into a linear list, you change the information. Of course, the distinction between structural elements and lay-out attributes is not sharp in general. To avoid a discussion about this topic here, we pragmatically follow the HTML-definitions. Note that our product model allows both, a description solely with the predefined structural elements or solely with presentation attributes of an unstructured text. We think that our proposed mix of both is the best way, but the model does not enforce this.

Pages are simple linear texts, with a fixed set of logical structuring elements, such as paragraphs, lists or tables. Of course, one can imagine more functions than we define here, but we tried to model the minimal necessary set of functions.

Besides the basic elements, we introduce a set of level-dependent symbols, which are simply characters on the first level. We differentiate them for practical reasons. Generally, symbols differ from basic elements in that they do not have an individual address, but are immediately handled by the browser.

Definition 2.11 (Structural Product Model "Page")

PAGE = MO `and` PAGE_SYMBOLS `and` ATT_PAGE `and`
TREE[entry↦page_struct] `and`
LIST[entry↦page] `and` LIST[entry↦nat] `then`
`vissorts`
page
page_struct
page_location = list(nat)
`constructs`
Basic, Symbol, Emptypage, Page_list, Table, Tableline, Headline, Minipage, Text,
Br, Footnote, Paragraph, Copyright : page_struct
$[\![]\!]$: page
$[\![\_]\!]$. mo $\to$ page
$''\_''$. page_symbols $\to$ page
Mkpage. page_struct $\times$ list(page) $\times$ att_page $\to$ page

To construct a hyperdocument of our first level we now only have to combine our product models for page and the address space and instantiate the parameters 'document', 'location', and 'addr'.

Definition 2.12 (Structural Product Model "Hypermedia-Documents")

HMD = PAGE `and` HMD_ADDR `and`
HD[document↦PAGE.page,
location↦PAGE.page_location,
addr↦HMD_ADDR.hmd_addr] `then`
`vissorts`
hmd = hd(PAGE.page, PAGE.page_location, HMD_ADDR.hmd_addr)

2.8.3 Chapters and Frameset Documents

The following specifications are essentially incomplete and have to be completed in the future!!!

At the second level, our basic elements are the structured hyperdocuments (Definition 2.12). From this point of view, the name "lineardocument", mentioned previously, is not quite right. Though it is organized without links on the discussed level (and hence "linear"), its basic documents might obviously be hyperdocuments already. The symbols at this level are geometrical forms, such as lines, rectangles or bars.

Definition 2.13 (Structural Product Model "Chapter")

CHAPTER = HMD `and` CHAPTER_SYMBOLS `and` ATT_CHAPTER `and`
TREE[entry↦chapter_struct] `and`
LIST[entry↦chapter] `and` LIST[entry↦nat] `then`
`vissorts`
chapter
chapter_struct
fsd_location = list(nat)
`constructs`
Horiz_frameset, Vert_frameset, Alt_frameset : fsd_struct

Analogous to the previous section, we must instantiate the parameters.

Definition 2.14 (Structural Product Model "Frameset Document")

FSD = CHAPTER `and` FSD_ADDR `and`
HD[document↦CHAPTER.chapter,
location↦CHAPTER.chapter_location,
addr↦FSD_ADDR.fsd_addr]

`then`
`vissorts`
fsd = hd(CHAPTER.chapter, CHAPTER.chapter_location, FSD_ADDR.fsd_addr)

2.8.4 Books and Sites

The following specifications are essentially incomplete and have to be completed in the future!!!

The third level is the aggregation of chapters to a book. A book consists of “hyperchapters”.

Definition 2.15 (Structural Product Model “Book”)

BOOK = FSD `and` BOOK_SYMBOLS `and` ATT_BOOK `and`
TREE[entry↦book_struct] `and`
LIST[entry↦book] `and` LIST[entry↦nat] `then`
`vissorts`
book
book_struct
book_location = list(nat)
`constructs`
sitemap : book_struct

Definition 2.16 (Structural Product Model “Site”)

SITE = BOOK `and` SITE_ADDR `and`
HD[document↦BOOK.book,
location↦BOOK.book_location,
addr↦SITE_ADDR.site_addr]

`then`
`vissorts`
site = hd(BOOK.book, BOOK.book_location, SITE_ADDR.site_addr)

3 Extending the Product Model

In § 2 we introduced an algebraic Dexter-based product model for hyperdocuments. We now extend this model with observer and editing functions to an algebraic model for *hyperdocument systems*. By "hyperdocument system" we mean, as suggested e.g. by [LH 99], functions of tools used by a developer to create and modify a hyperdocument. *Observer functions* supply information about the objects, e.g. which elements a document contain. *Editing functions* can modify a concrete object, but of course not the domain. The remaining functions are merely *auxiliary functions*. They are not discussed in detail, but documented in the appendix.

In the constructor-based algebraic approach the set of functions is divided into a set of *constructors* (cf. § 2.1) and a set of *non-constructors* or *defined functions*. Defined Functions are defined via axioms on the basis of the constructors. Observer functions and editing functions are both represented by defined functions.

In our domain we have parameter specifications (*document*), object-classes (*anchor* and*hyperdocument*), and concrete objects (*link*, *page*, *hypermedia document*, *chapter*, *frame*, *book*, and*site*). For each of these we will explain at first the observer functions (§ 3.1) and then the editing functions (§ 3.2).

3.1 Observer Functions

Objects are represented by tuples, build up with the help of the constructors. Observer functions are characterized by their ability to extract information out of these tuples. Historically they are sometimes called *destructors*, because they can deconstruct objects. As the term "destructor" has already been used with so many connotations and it is not clear whether it includes the Boolean functions, we prefer the term "observer functions" here.

The*observer functions* include the following two special cases:

Boolean functions will be marked with a question mark '?' at the end of their names.

Projections extract exactly one component of a composite object. Names of projections will be prefixed with 'get_'.

Observer functions must not be mixed up with *display functions*. Even though both help the user or developer to observe an object, the latter transforms the logical description into a 'physical' and visible description, in our case a notation that can be displayed by a user agent or browser. Display functions are much more sophisticated in their algebraic representation and a part of our ongoing work.

3.1.1 Document

The parameter specification for *documents* has only one Boolean function, namely 'embed_link_ok?'. It tests whether an embed link can be positioned at a given location in the document. All other observer and editing functions belong to the documents on the corresponding level.

3.1.2 Page

At the first level are the *pages*. A page is either an empty page, some media object of lower level, some page symbol of the corresponding level, or a triple constructed by 'Mkpage' (cf. § 2.8.2) from a structure name ('page_struct'), a list of pages ('list(page)'), and some attributes ('att_page').

Definition 3.1 (Observer Functions "Page")
`defuns`
atomic?. page → bool
has_pnth?. nat × list(page) → bool
has_location?. page_location × page → bool
embed_link_page_ok?. page_location × page → bool
get_struct. page → tree(page_struct)
get_pages. page → list(page)
get_att. page → att_page
locate. page_location × page → page
page_dimension. page → list(nat)

A page is called *atomic* ('atomic?') iff it is empty, a media object, or a symbol.

'has_location?' is a partially defined boolean function, which tests whether a location occurs in a page. The empty location means the whole page and therefore it exists in every page.

'embed_link_page_ok?' returns 'true' if a given location exists in the page and the document located there is an empty page. If the location does not exist, it returns 'false'.

As a page is a nested structure, the adequate result of the observer function 'get_struct' is the tree of structures in the page under consideration.

Similarly the result of 'get_pages' is the list of all pages that a given page includes on top level.

'get_att' returns merely the top level attributes of the page.

'locate' returns the sub-page located at a given position in a given page.

'page_dimension' returns the list of natural numbers of the sizes of the page in all its dimensions. E.g., a two dimensional table with m lines and a maximum of n columns in one of these lines has a dimension of (m, n). This means that the smallest two dimensional cube around it will have hight m and breadth n. A three dimensional table with dimension (m, n, p) will fill a cube of depth p. If the objects are not atomic, the element-wise maximum of its dimensions will be appended at the end of the dimension list of the table. Generally speaking, a page object represented as an Mkpage-node tree of depth d has the dimension $(n_1, \ldots, n_d)$ where n_i is the maximum number of children of a node at depth i.

3.1.3 Anchor

An *anchor* is a triple constructed by 'Mkanchor' (cf. § 2.4) from a location ('location'), a type ('anchor_type'), and some attributes ('att_anchor').

Definition 3.2 (Observer Functions "Anchor")
`defuns`
get_location. anchor → location
get_type. anchor → anchor_type
get_att. anchor → att_anchor
suptype. anchor × anchor → anchor_type

We need a projection for each component, called 'get_location', 'get_type' and 'get_att'.

The last observer function, 'suptype', returns the supremal type according to '$\forall x.\ x \leq$ Label' because an anchor of type 'Label' can serve both as source and as target, while the types 'Source' and 'Target' are incomparable.

3.1.4 Link

A *(hyper) link* is a quadruple constructed by 'Mklink' (cf. § 2.5) of a two sets of specifiers denoting the source and target ('set(specifier)'), a type ('link_type'), and some attributes ('att_link'). Specifiers again are pairs consisting of a global address ('uri') and local name ('anchor_id').

Definition 3.3 (Observer Functions "Link")
`defuns`
get_uri. specifier → uri
get_id. specifier → anchor_id
get_source. link → set(specifier)
get_target. link → set(specifier)
get_type. link → link_type
get_att. link → att_link
get_specifier. link → set(specifier)

We need projections, 'get_uri', 'get_id' for the specifiers and 'get_source', 'get_target', 'get_type' and 'get_att' for the links.

'get_specifier' returns the set of all specifiers in the source and the target of a link.

3.1.5 Hyperdocument

A *hyperdocument* is a quintuple constructed by 'Mkhd' (cf. § 2.7) from a document ('document'), a function mapping anchor names to anchors ('function(anchor_id,anchor)'), a set of links ('set(link)'), some attributes ('att_hd'), and an address ('addr').

Definition 3.4 (Observer Functions "Hyperdocument")
`defuns`
$\|\_\|$. hd $\rightarrow$ document
get_anchors. hd $\rightarrow$ function(anchor_id,anchor)
get_link. hd $\rightarrow$ set(link)
get_att. hd $\rightarrow$ att_hd
get_addr. hd $\rightarrow$ addr
get_anchor_id. anchor $\times$ function(anchor_id,anchor) $\rightarrow$ set(anchor_id)
get_anchor. anchor_id $\times$ function(anchor_id,anchor) $\rightarrow$ anchor

Of course we get five projections, namely $\|\_\|$, get_anchors, get_link, get_att and get_addr. The first one extracts the (linear) document from the hyperdocument. Because this function will be used very often, we use the short notation '$\|\_\|$' instead of the name 'get_document'.

'get_anchor', returns the anchor to a given anchor name.

'get_anchor_id' returns the set of all anchor names referring to a given anchor.

3.2 Editing Functions

The*editing functions* are the most interesting functions for the user. With the help of these functions a hyperdocument can be designed and modified.

3.2.1 Page

We will start with the functions for working with *pages*.

Definition 3.5 (Editing Functions "Page")
`defuns`
ch_struct. page_struct $\times$ page $\rightarrow$ page
insert_at. page $\times$ page_location $\times$ page $\rightarrow$ page
place_at. page $\times$ page_location $\times$ page $\rightarrow$ page
add_attribute. att_page $\times$ page $\rightarrow$ page
del_attribute. att_page $\times$ page $\rightarrow$ page
mktable. nat $\times$ nat $\rightarrow$ page
mklist. nat $\rightarrow$ page

'ch_struct' is a kind of converter function. The components of the page are left untouched, but arranged in another structure.

Inserting one page into another at a special place is the most important editing action a designer might need. We give two different functions to do that: 'insert_at' and 'place_at'. Both replace a part of an existing page, residing at a given location, with a new page. A location is represented by a node position. 'insert_at' moreover extends the page with sufficiently many child nodes, if this location does not yet exist. The type of these child nodes may depend on the parent node. E.g., if the parent node is a table then the child nodes will be of type table-line. If no special knowledge is given, the child nodes will be simply of type empty page.

'add_attribute' and 'del_attribute' add or remove attributes resp.. Editing functions for attributes exist for every object and are not mentioned in the further sections anymore.

A special kind of editing functions are 'mklist' and 'mktable'. They are syntactic sugar for very often used construction mechanisms. 'mklist' produces a list with a given number of items, containing an empty page in every item. 'mktable' produces a $m \times n$-table, containing an empty page in every cell.

3.2.2 Anchor

Anchor has only editing function that change the values of the location ('ch_location') or the type ('ch_type') resp..

Definition 3.6 (Editing Functions "Anchor")
`defuns`
ch_location. location × anchor → anchor
ch_type. anchor_type × anchor → anchor
add_attribute. att_anchor × anchor → anchor
del_attribute. att_anchor × anchor → anchor

3.2.3 Link

According to the construction of links, we have editing functions for specifiers and for links, which are very simple functions for changing the value of a component.

Definition 3.7 (Editing Functions "Specifier")
`defuns`
ch_uri. uri × specifier → specifier
ch_id. anchor_id × specifier → specifier

Definition 3.8 (Editing Functions "Link")
`defuns`
insert_source. set(specifier) $\times$ link $\rightarrow$ link
delete_source. set(specifier) $\times$ link $\rightarrow$ link
insert_target. set(specifier) $\times$ link $\rightarrow$ link
delete_target. set(specifier) $\times$ link $\rightarrow$ link
ch_type. link_type $\times$ link $\rightarrow$ link
add_attribute. att_link $\times$ link $\rightarrow$ link
del_attribute. att_link $\times$ link $\rightarrow$ link

3.2.4 Hyperdocument

At the first glimpse, things seem to be as easy with hyperdocuments as with the other objects. For the most functions, 'del_anchor', 'del_link', 'add_attribute', 'del_attribute' and 'ch_addr', this is true. But 'add_anchor' and 'add_link' are much more sophisticated in their details.

Definition 3.9 (Editing Functions "Hyperdocument")
`defuns`
add_anchor. anchor_id $\times$ anchor $\times$ hd $\rightarrow$ hd
del_anchor. anchor_id $\times$ hd $\rightarrow$ hd
add_link. link $\times$ hd $\rightarrow$ hd
del_link. link $\times$ hd $\rightarrow$ hd
add_attribute. att_hd $\times$ hd $\rightarrow$ hd
del_attribute. att_hd $\times$ hd $\rightarrow$ hd
ch_addr. addr $\times$ hd $\rightarrow$ hd

'add_anchor' produces a hyperdocument after a given anchor with given name has been added to the anchors of the original hyperdocument, provided that an anchor with this name does not exist before. If an anchor with this name does exist in the original document at the same location it is updated to an anchor with supremal type and attributes. Otherwise the function is not defined.

'add_link' is the most complex editing function because we must consider several different cases in that the addition of a link can be accepted. A link of the type 'Uni(Replace, $*$)' or 'Uni(New_window, $*$)' may be added when its source contains a specifier that refers to an anchor in the the given hyperdocument of type 'Source' or 'Label'. For a link of the type 'Uni(Embed, User)' we additionally require that this anchor must point to a location that may carry an embed link. For a link of the type 'Uni(Embed, Auto)' we additionally require that the link has exactly one target. Finally, a link of the type 'Bi' may be added when its source contains a specifier that refers to an anchor in the the given hyperdocument of type 'Label'.

3.2.5 Hypermedia Document

The hyperdocument at level 1 is called*hypermedia document*. It is an instantiation of the hyperdocument object-class and therefore it includes all functions given there. Besides that, it provides the two insertion functions 'place_at' and 'insert_at'.

Definition 3.10 (Editing Functions "Hypermedia Document")
`defuns`
place_at. hmd $\times$ page_location $\times$ hmd $\times$ hmd_addr $\rightarrow$ hmd
insert_at. hmd $\times$ page_location $\times$ hmd $\times$ hmd_addr $\rightarrow$ hmd

'insert_at' replaces the part of a given hyperdocument, located at a fixed existing location, with a new hyperdocument. The replacement is only possible when the names of the anchors in the two hyperdocuments are disjoint and the replaced part does not carry any anchors. The result gets the address given in the last argument of the function and all links referring to any of two input hyperdocuments are changed in order to refer to the the resulting hyperdocument.

'place_at' has the same result as 'insert_at' provided that the location actually exists in the given hyperdocument. Otherwise, it generates this location just as 'place_at' from "Page", cf. § 3.2.1.

4 Conclusion and Outlook

To our knowledge we have presented the first[13] *formal algebraic* hypertext reference model. It guarantees a unique understanding and enables a close connection to logic-based development and verification. With the exception of some deviations in order to be compatible with the WWW it follows the Dexter Hypertext Reference Model (cf. [HS 90]) and could be seen as an updated formally algebraic version of it. Additionally, three different levels of hyperdocuments, namely hypermedia documents, frameset documents, and sites are introduced — although the specification of the latter two is still essentially incomplete and has to be completed in future work.

The hypertext model (cf. § 2) was developed as a product model with the aim to support the design of the product "hyperdocument" automatically. It is extended to a model of hypertext-systems (cf. § 3) in order to describe the state transitions of the design-process. The whole specification is in the appendix and a prototypical implementation in ML will be found under `http://www.ags.uni-sb.de/ cp/ml/come.html`.

In this paper we have algebraically specified the information-centered model and the interfaces to the screen-based model. Before we can start the formalization of the screen-based model, we need to study the numerous existing, non-formalized, screen-based approaches. Up to now the favorite idea is to use PDF as a reference model. The mapping between the formalized information-centered model and the formalized screen-based model will then provide an abstract kind of reference user agent (browser), cf. Fig. 1 on page 3.

[13] Note that we do not consider Z to be a formal algebraic specification language.

A The Algebraic Specification

A.1 Basic Specifications

The specifications for BOOL (for the Boolean functions), NAT, CHAR, STRING, TREE, LIST, LISTPAIR, SET, MAPSET, and FUNCTION are assumed to be given, but we will present some of their signatures below.

The maximum operator $\max(n, n')$ must be defined in the module 'NAT'. The standard boolean function is_proper_prefix(l, l') and the functions repeat(n, x) (which returns a list containing x n-times), and map(f, l) must be defined in the module 'LIST'.

The following parameter specification provides only one single sort. Note, however, that for any specification we tacitly assume the inclusion of the module 'BOOL' and the existence of an equality and an inequality predicate which exclude each other and are total on objects described by constructor ground terms (data objects).

ENTRY
`sorts` entry

Since SET is so fundamental, we present its signature here.

SET = ENTRY `and` NAT `then`
`sorts` set = set(entry)
`funs`

'{}' is the empty set.

{}. → set

'null' test whether a set is empty.

null. set → bool

Is first argument contained in the second argument?

_ ∈ _. entry × set → bool

'|_|' returns the cardinality (i.e. the number of elements) of a set.

\|_\|. set → nat

'insert' inserts its first argument as an element into its second argument.

insert. entry × set → set

'dl' deletes its first argument as an element from its second argument.

dl. entry $\times$ set $\to$ set

'_ $\cup$ _' returns the union of its arguments.

_ $\cup$ _. set $\times$ set $\to$ set

'_ $\cap$ _' returns the intersection of its arguments.

_ $\cap$ _. set $\times$ set $\to$ set

'exists' tests whether its second argument contains an element satisfying its first argument.

exists. (entry $\to$ bool) $\times$ set $\to$ bool

MAPSET will be use to map sets to sets. Note that it cannot be a part of SET because it needs two sort parameters (one for the domain and one for the range of the mapping function) instead of one.

MAPSET = SET[entry$\mapsto$entry1] `and` SET[entry$\mapsto$entry2] `then`
`funs`

'map_set' replaces all elements of its second argument by their values under its first argument.

map_set. (entry1 $\to$ entry2) $\times$ set(entry1) $\to$ set(entry2)

LISTPAIR provides operations on pairs of lists and is similar to the Standard ML Basis Library module of the same name, but we need the following non-standard function:

LISTPAIR = LIST[entry$\mapsto$entryD1] `and`
LIST[entry$\mapsto$entryD2] `and`
LIST[entry$\mapsto$entryR] `then`
`funs`

'map_default' maps two input lists (fourth and fifth argument) into a new list by applying a binary function (third argument). In case one of the input lists is shorter than the other, default values (first and second argument) are appended to the shorter list.

map_default. entryD1$\times$
entryD2$\times$
(entryD1 $\times$ entryD2 $\to$ entryR)$\times$
list(entryD1)$\times$
list(entryD2)
$\to$ list(entryR)

Since FUNCTION is non-standard, we present its signature here.

FUNCTION = SET[entry$\mapsto$domain] **and** SET[entry$\mapsto$range] **then**
sorts function = function(domain, range)
funs

empty_function is the function with empty domain.

empty_function. $\rightarrow$ function

'upd' returns its third argument but with its second argument being the new value of its first argument. UPDate.

upd. domain $\times$ range $\times$ function $\rightarrow$ function

'apply' applies its first argument to its second argument and is undefined if the second argument is not in the domain of the first argument.

apply. function $\times$ domain $\rightarrow$ range

'rem' returns its second argument but now undefined for its first argument. REMove from domain.

rem. domain $\times$ function $\rightarrow$ function

DOMain of a function.

dom. function $\rightarrow$ set(domain)

RANge of a function.

ran. function $\rightarrow$ set(range)

'rev_apply' applies the reverse relation of first argument to the singleton set containing its second argument. REVerse-APPLY.

rev_apply. function $\times$ range $\rightarrow$ set(domain)

'union' unites its first argument with its second argument in such a way that first argument wins in case of conflicts.

union. function $\times$ function $\rightarrow$ function

'map_range' replaces the range elements of its second argument with their values under its first argument.

map_range. (range $\rightarrow$ range) $\times$ function $\rightarrow$ function

A.2 Parameter Specifications

The specifications for URI, HMD_ADDR, FSD_ADDR, SITE_ADDR, ANCHOR_ID, as well as for HMD_SYMBOLS, FSD_SYMBOLS, SITE_SYMBOLS and ATT_HMD, ATT_FSD, ATT_SITE are left open and are subject of future work.

DOCUMENT_P below is merely a parameter specification. Intuitively you would expect a rudimentary structure here characterizing the genre “document”. For the first level, the*pages*, this structure is obvious, for the second level, the*frames*, it seems to be very similar. For the third level, the*sites*, it is far from clear, however, whether this modeling is actually adequate. We therefore have chosen a parameter specification to ensure sufficient flexibility.

DOCUMENT_P = ENTRY[entry↦document] `and`
ENTRY[entry↦location] `then`

`sorts` document
location

`funs`

embed_link_ok?(l, b) tests whether an embed link can be positioned at location l in document b.

embed_link_ok?. location × document → bool

The following parameter specification provides us with a sort ‘addr’ of addresses for local storage of hyperdocuments.

ADDR_P = ENTRY[entry↦addr]

A.3 Anchors

ANCHOR[location] = DOCUMENT_P[document,location] `and` ATT_ANCHOR `then`
`vissorts`
anchor_type
anchor = anchor(location)
`constructs`
Source, Target, Label : anchor_type
Mkanchor. location × anchor_type × att_anchor → anchor
`defuns`
− − − `Observer Functions` − −−
get_location. anchor → location
get_type. anchor → anchor_type
get_att. anchor → att_anchor
suptype. anchor × anchor → anchor_type
− − − `Editing Functions` − −−
ch_location. location × anchor → anchor
ch_type. anchor_type × anchor → anchor
add_attribute. att_anchor × anchor → anchor
del_attribute. att_anchor × anchor → anchor
`vars` o, o'. location
t, t'. anchor_type
att, att'. att_anchor
c, c'. anchor
`axioms`
− − − `Observer Functions` − −−
get_location(Mkanchor$(o, t, att)) = o$
get_type(Mkanchor$(o, t, att)) = t$
get_att(Mkanchor$(o, t, att)) = att$

suptype(c, c')
Returns the supremal type according to '$\forall x.\ x \leq$ Label' because 'Label' can serve both as source and as target, while 'Source' and 'Target' are incomparable.

$$
\begin{array}{ll}
\text{suptype}(\text{Mkanchor}(o, \text{Label}, att), c') = \text{Label} & \\
\text{suptype}(c, \text{Mkanchor}(o', \text{Label}, att')) = \text{Label} & \\
\text{suptype}(\text{Mkanchor}(o, t, att), \text{Mkanchor}(o', t', att')) = \text{Label} & \Longleftarrow t \neq t' \\
\text{suptype}(\text{Mkanchor}(o, t, att), \text{Mkanchor}(o', t', att')) = t & \Longleftarrow t = t'
\end{array}
$$

− − − `Editing Functions` − −−
ch_location$(o'$, Mkanchor$(o, t, att))$ = Mkanchor(o', t, att)
ch_type$(t'$, Mkanchor$(o, t, att))$ = Mkanchor(o, t', att)
add_attribute$(att'$, Mkanchor$(o, t, att))$ = Mkanchor$(o, t$, concat$(att', att))$
del_attribute$(att'$, Mkanchor$(o, t, att))$ = Mkanchor$(o, t$, remove$(att', att))$

A.4 Links

LINK = ANCHOR_ID `and` URI `and` ATT_LINK `and`
MAPSET[entry1↦specifier, entry2↦specifier] `then`

`vissorts`
link_type
show
actuate
specifier
link

`constructs`

Links of show-type ‘Embed’ embed their target into the context of their source. Links of show-type ‘Replace’ replace the hyperdocument of their source with the hyperdocument of their target. Finally, links of show-type ‘New_window’ open a new window with the document of their target.

Embed, Replace, New_window : show

Links of actuate-type ‘User’ are followed upon user interaction. Links of actuate-type ‘Auto’ are followed automatically.

User, Auto : actuate

Links may be uni-directional (‘Uni($*, *$)’) or bi-directional (‘Bi’). Since bi-directional links are always of show-type ‘Replace’ and of actuate-type ‘User’, no arguments are given to ‘Bi’.

Uni. show × actuate → link_type
Bi : link_type

A specifier consists of a global address of sort ‘uri’ and a local name of sort ‘anchor_id’ that is to be mapped to an anchor by the hyperdocument under the global address.

Mkspecifier. uri × anchor_id → specifier
Mklink. set(specifier) × set(specifier) × link_type × att_link → link

```
defuns
 --- Observer Functions ---
 get_uri. specifier → uri
 get_id. specifier → anchor_id
 get_source. link → set(specifier)
 get_target. link → set(specifier)
 get_specifier. link → set(specifier)
 get_type. link → link_type
 get_att. link → att_link
 --- Editing Functions for Specifier ---
 ch_uri. uri × specifier → specifier
 ch_id. anchor_id × specifier → specifier
 replace_uri_sp. uri × uri × specifier → specifier
 --- Editing Functions for Link ---
 insert_source. set(specifier) × link → link
 delete_source. set(specifier) × link → link
 insert_target. set(specifier) × link → link
 delete_target. set(specifier) × link → link
 ch_type. link_type × link → link
 add_attribute. att_link × link → link
 del_attribute. att_link × link → link
 replace_uri_li. uri × uri × link → link
vars  S, S', S'', S'''. set(specifier)
      s, s'. specifier
      l, l'. link
      L. set(link)
      t, t'. link_type
      n, n'. anchor_id
      att, att'. att_link
      a, a', a''. uri
```

`axioms`

$---$ `Observer Functions` $---$

$\text{get\_uri}(\text{Mkspecifier}(a, n)) = a$
$\text{get\_id}(\text{Mkspecifier}(a, n)) = n$
$\text{get\_source}(\text{Mklink}(S, S', t, att)) = S$
$\text{get\_target}(\text{Mklink}(S, S', t, att)) = S'$
$\text{get\_specifier}(\text{Mklink}(S, S', t, att)) = S \cup S'$
$\text{get\_type}(\text{Mklink}(S, S', t, att)) = t$
$\text{get\_att}(\text{Mklink}(S, S', t, att)) = att$

$---$ `Editing Functions for Specifier` $---$

$\text{ch\_uri}(a', \text{Mkspecifier}(a, n)) = \text{Mkspecifier}(a', n)$
$\text{ch\_id}(n', \text{Mkspecifier}(a, n)) = \text{Mkspecifier}(a, n')$
$\text{replace\_uri\_sp}(a', a'', \text{Mkspecifier}(a, n)) = \text{Mkspecifier}(a'', n) \quad \Longleftarrow \quad a'{=}a$
$\text{replace\_uri\_sp}(a', a'', \text{Mkspecifier}(a, n)) = \text{Mkspecifier}(a, n) \quad \Longleftarrow \quad a'{\neq}a$

$---$ `Editing Functions for Link` $---$

$\text{insert\_source}(s, \text{Mklink}(S, S', t, att)) = \text{Mklink}(\text{insert}(s, S), S', t, att)$
$\text{delete\_source}(s, \text{Mklink}(S, S', t, att)) = \text{Mklink}(\text{dl}(s, S), S', t, att)$
$\text{insert\_target}(s, \text{Mklink}(S, S', t, att)) = \text{Mklink}(S, \text{insert}(s, S'), t, att)$
$\text{delete\_target}(s, \text{Mklink}(S, S', t, att)) = \text{Mklink}(S, \text{dl}(s, S'), t, att)$
$\text{ch\_type}(t', \text{Mklink}(S, S', t, att)) = \text{Mklink}(S, S', t', att)$
$\text{add\_attribute}(att', \text{Mklink}(S, S', t, att)) = \text{Mklink}(S, S', t, \text{concat}(att', att))$
$\text{del\_attribute}(att', \text{Mklink}(S, S', t, att)) = \text{Mklink}(S, S', t, \text{remove}(att', att))$

$\text{replace\_uri\_li}(a', a, l) = l'$
Replaces any reference to the URI a' in the specifiers of the link l with the URI a. Note that we can use 'replace_uri_sp' as a binary function in the definition because we consider all functions to be curried and argument tupling just to be syntactic sugar. Finally, note that 'map_set' is from MAPSET[entry1$\mapsto$specifier, entry2$\mapsto$specifier].

$\text{replace\_uri\_li}(a', a, \text{Mklink}(S, S', t, att)) =$
$\text{Mklink}(\text{map\_set}(\text{replace\_uri\_sp}(a', a), S), \text{map\_set}(\text{replace\_uri\_sp}(a', a), S'), t, att)$

A.5 Hyperdocuments

HD[document,location,addr] = DOCUMENT_P[document,location] `and`
ADDR_P[addr] `and`
ANCHOR[location] `and`
LINK `and`
ATT_HD `and`
FUNCTION[domain↦anchor_id, range↦anchor] `and`
SET[entry↦link]

`then`
`vissorts`
hd = hd(document, location, addr)
`constructs`
Mkhd. document × function(anchor_id,anchor) × set(link) × att_hd × addr → hd
`defuns`
− − − `Observer Functions` − − −
‖_‖. hd → document
get_anchors. hd → function(anchor_id,anchor)
get_link. hd → set(link)
get_att. hd → att_hd
get_addr. hd → addr
get_anchor. anchor_id × function(anchor_id,anchor) → anchor
get_anchor_id. anchor × function(anchor_id,anchor) → set(anchor_id)
− − − `Editing Functions` − − −
add_anchor. anchor_id × anchor × hd → hd
del_anchor. anchor_id × hd → hd
add_link. link × hd → hd
del_link. link × hd → hd
add_attribute. att_hd × hd → hd
del_attribute. att_hd × hd → hd
ch_addr. addr × hd → hd
− − − `Converter Functions` − − −
embed. addr → uri
`vars` d, d'. document
L, L'. set(link)
l. link
act. actuate
sp, sp'. specifier
A, A'. function(anchor_id,anchor)
c, c'. anchor
a, a', a''. addr
att, att'. att_hd
n. anchor_id

`axioms`

`— — — Observer Functions — — —`

$\|\text{Mkhd}(d, A, L, att, a)\| = d$
$\text{get\_anchors}(\text{Mkhd}(d, A, L, att, a)) = A$
$\text{get\_link}(\text{Mkhd}(d, A, L, att, a)) = L$
$\text{get\_att}(\text{Mkhd}(d, A, L, att, a)) = att$
$\text{get\_addr}(\text{Mkhd}(d, A, L, att, a)) = a$

$\text{get\_anchor}(n, A)$
Returns the anchor referred to by the name n by calling the function 'apply' from FUNCTION.

$\text{get\_anchor}(n, A) = \text{apply}(A, n)$

$\text{get\_anchor\_id}(c, A)$
Returns the set of all names referring to the anchor c by calling the function 'rev_apply' from FUNCTION.

$\text{get\_anchor\_id}(c, A) = \text{rev\_apply}(A, c)$

`-- Editing Functions --`

$\text{add\_anchor}(n, c, h) = h'$
h' is the hyperdocument after the anchor c with name n has been added to the anchors of hyperdocument h, provided that an anchor with this name does not exist in h before. If an anchor with name n does exist in h at the same location as anchor c, then h' is updated to an anchor with supremal type and attributes. Note that we use 'upd' from FUNCTION and write long argument lists vertically instead of horizontally.

$$\text{add\_anchor}(n, c, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, \text{upd}(n, c, A), L, att, a)$$
$$\Longleftarrow \quad (n \in \text{dom}(A)) = \text{false}$$

$$\begin{aligned}
&\text{add_anchor}(n, c, \text{Mkhd}(d, A, L, att, a)) = \\
&\quad \text{Mkhd}(d, \\
&\qquad\quad \text{upd}(n \\
&\qquad\qquad \text{Mkanchor}(\text{get_location}(c), \\
&\qquad\qquad\qquad\qquad \text{suptype}(c, c'), \\
&\qquad\qquad\qquad\qquad \text{concat}(\text{get_att}(c), \text{get_att}(c'))), \\
&\qquad\qquad A), \\
&\qquad\quad L, \\
&\qquad\quad att, \\
&\qquad\quad a) \\
&\quad \Longleftarrow \quad (n \in \text{dom}(A)) = \text{true} \wedge \text{get_anchor}(n, A) = c' \wedge \text{get_location}(c) = \text{get_location}(c')
\end{aligned}$$

$\text{del\_anchor}(n, h) = h'$
h' is the hyperdocument after the anchor with the name n has been removed from the hyperdocument h.

$\text{del\_anchor}(n, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, \text{rem}(n, A), L, att, a)$

$\text{add\_link}(l, h) = h'$
h' is the hyperdocument after the link l has been added to the set of links in h. A link of the type 'Uni(Replace, $*$)' or 'Uni(New_window, $*$)' may be added when its source contains a specifier sp that refers to an anchor c in the the given hyperdocument of type 'Source' or 'Label'. This is expressed in the first four rules. For a link of the type 'Uni(Embed, User)' we additionally require that this anchor c must point to a location that may carry an embed link. This is expressed in the next two rules. Note that 'embed_link_ok?' comes from DOCUMENT_P. For a link of the type 'Uni(Embed, Auto)' we additionally require that the link has exactly one target. This is expressed in the next two rules. Finally, a link of the type 'Bi' may be added when its source contains a specifier sp that refers to an anchor c in the the given hyperdocument of type 'Label'.

$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Uni(Replace}, act) \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Source}$
$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Uni(Replace}, act) \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Label}$
$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Uni(New\_window}, act) \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Source}$
$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Uni(New\_window}, act) \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Label}$
$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Uni(Embed, User)} \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Source} \wedge$
$\quad\quad \text{embed\_link\_ok?}(\text{get\_location}(c), d)$
$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Uni(Embed, User)} \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Label} \wedge$
$\quad\quad \text{embed\_link\_ok?}(\text{get\_location}(c), d)$
$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Uni(Embed, Auto)} \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Source} \wedge$
$\quad\quad \text{embed\_link\_ok?}(\text{get\_location}(c), d) \wedge |\text{get\_target}(l)|\text{=}1$
$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Uni(Embed, Auto)} \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Label} \wedge$
$\quad\quad \text{embed\_link\_ok?}(\text{get\_location}(c), d) \wedge |\text{get\_target}(l)|\text{=}1$
$\text{add\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{insert}(l, L), att, a)$
$\quad \Longleftarrow \text{get\_type}(l)\text{=Bi} \wedge sp \in \text{get\_source}(l) \wedge$
$\quad\quad \text{get\_uri}(sp)\text{=embed}(a) \wedge \text{get\_anchor}(\text{get\_id}(sp), A)\text{=}c \wedge \text{get\_type}(c)\text{=Label}$

$\text{del\_link}(l, h) = h'$
h' is the hyperdocument after the link l is removed from the hyperdocument h.

$\text{del\_link}(l, \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, \text{dl}(l, L), att, a)$

$\text{add\_attribute}(att, h) = h'$
h' is the hyperdocument after the hyperdocument h is enriched with the attributes att.

$\text{add\_attribute}(att', \text{Mkhd}(l, A, L, att, a)) = \text{Mkhd}(l, A, L, \text{concat}(att', att), a)$

$\text{del\_attribute}(att, h) = h'$
h' is the hyperdocument after the attributes att are removed from the hyperdocument h.

$\text{del\_attribute}(att', \text{Mkhd}(l, A, L, att, a)) = \text{Mkhd}(l, A, L, \text{remove}(att', att), a)$

$\text{ch\_addr}(a', h) = h'$
h' is the hyperdocument after the address of h is replaced by address a'.

$\text{ch\_addr}(a', \text{Mkhd}(d, A, L, att, a)) = \text{Mkhd}(d, A, L, att, a')$

A.6 Media Objects

MO = URI `and` ANCHOR_ID `and` SET[entry↦anchor_id] `then`
`vissorts`
mo = mo(uri, anchor_id)
`constructs`

> Our interface to media-objects is quite simple because we are not interested in modeling their internal behavior. The only thing we require is that they have some unified resource identifier of sort 'uri' and a set of anchor identifiers to which links may refer. Thus, a media-object basically introduces a legal set of specifiers referring to it.

Mkmo. uri $\times$ set(anchor_id) $\rightarrow$ mo

A.7 Hypermedia Document Level

A.7.1 Page

PAGE_SYMBOLS = STRING `then`
`vissorts`
 page_symbols

PAGE = MO `and` PAGE_SYMBOLS `and` ATT_PAGE `and`
 TREE[entry↦page_struct] `and`
 LIST[entry↦page] `and`
 LISTPAIR[entryD1↦nat, entryD2↦nat, entryR↦nat] `then`
`vissorts`
 page
 page_struct
 page_location = list(nat)
`constructs`
 Basic, Symbol, Emptypage, Page_list, Table, Tableline, Headline, Minipage, Text,
 Br, Footnote, Paragraph, Copyright : page_struct
 ⟦⟧ : page
 ⟦_⟧. mo → page
 ″_″. page_symbols → page
 Mkpage. page_struct × list(page) × att_page → page
`defuns`
 − − − `Observer Functions` − −−
 atomic?. page → bool
 has_pnth?. nat × list(page) → bool
 has_location?. page_location × page → bool
 embed_link_page_ok?. page_location × page → bool
 get_struct. page → tree(page_struct)
 get_pages. page → list(page)
 get_att. page → att_page
 pnth. nat × list(page) → page
 locate. page_location × page → page
 page_dimension. page → list(nat)
 page_list_dimension. list(page) → list(nat)
 − − − `Editing Functions` − −−
 ch_struct. page_struct × page → page
 mklist. nat → page
 mktable. nat × nat → page
 mktableline. nat → page
 place_at. page × page_location × page → page
 place_at_help. page × nat × page_location × list(page) × page → list(page)
 insert_at. page × page_location × page → page
 add_attribute. att_page × page → page
 del_attribute. att_page × page → page

`vars` h. mo
$symb$. page_symbols
p, p', p'', p'''. page
s, s'. page_struct
P. list(page)
n. nat
o. page_location
att, att'. att_page

`axioms`

`--- Observer Functions ---`

$\text{atomic?}([\![]\!]) = \text{true}$
$\text{atomic?}([\![h]\!]) = \text{true}$
$\text{atomic?}(''symb'') = \text{true}$
$\text{atomic?}(\text{Mkpage}(s, P, att)) = \text{false}$

has_location?(o, p)
Tests whether location o occurs in page p. The empty location [] means the whole page and therefore it exists in every page. has_pnth?(o, P) is an auxiliary function for it.

$\text{has\_pnth?}(\text{s}(0), []) = \text{false}$
$\text{has\_pnth?}(\text{s}(0), p :: P) = \text{true}$
$\text{has\_pnth?}(\text{s}(\text{s}(n)), p :: P) = \text{has\_pnth?}(\text{s}(n), P)$
$\text{has\_location?}([], p) = \text{true}$
$\text{has\_location?}(\text{s}(n) :: o, p) = \text{false}$
$\Longleftarrow \text{atomic?}(p) = \text{true}$
$\text{has\_location?}(\text{s}(n) :: o, p) = \text{false}$
$\Longleftarrow \text{atomic?}(p) = \text{false} \wedge p = \text{Mkpage}(s, P, att) \wedge \text{has\_pnth?}(\text{s}(n), P) = \text{false}$
$\text{has\_location?}(\text{s}(n) :: o, p) = \text{has\_location?}(o, \text{pnth}(\text{s}(n), P))$
$\Longleftarrow \text{atomic?}(p) = \text{false} \wedge p = \text{Mkpage}(s, P, att) \wedge \text{has\_pnth?}(\text{s}(n), P) = \text{true}$

embed_link_page_ok?(o, p)
Returns 'true' if the location o exists in page p and the document located at o is an empty page $[\![]\!]$. If location o does not exist in page p it returns 'false'.

$\text{embed\_link\_page\_ok?}(o, p) = \text{false} \Longleftarrow \text{has\_location?}(o, p) = \text{false}$
$\text{embed\_link\_page\_ok?}(o, p) = \text{true} \Longleftarrow \text{has\_location?}(o, p) = \text{true} \wedge \text{locate}(o, p) = [\![]\!]$

get_struct(p)
Returns the tree of structures in page p. Notice that it uses the function 'map' from LIST that runs the function in its first argument over the list in its second argument.

$$
\begin{aligned}
\text{get\_struct}([\![]\!]) &= \text{Mktree}(\text{Emptypage}, [])\\
\text{get\_struct}([\![h]\!]) &= \text{Mktree}(\text{Basic}, [])\\
\text{get\_struct}(''symb'') &= \text{Mktree}(\text{Symbol}, [])\\
\text{get\_struct}(\text{Mkpage}(s, P, att)) &= \text{Mktree}(s, \text{map}(\text{get\_struct}, P))
\end{aligned}
$$

get_pages(p) $= P$
P are the top level elements of page p.

$\text{get\_att}(\text{Mkpage}(s, P, att)) = P$

get_att(p) $= att$
att are the top level attributes of page p.

$\text{get\_att}(\text{Mkpage}(s, P, att)) = att$

pnth(s(n), P)
Computes the n[th] element of the list P, but starts with 1 (instead of 0).

$\text{pnth}(\text{s}(n), P) = \text{nth}(n, P)$

locate(o, p) $= p'$
p' is the the page located at position o in page p.

$\text{locate}([], p) = p$
$\text{locate}(\text{s}(n) :: o, \text{Mkpage}(s, P, att)) = \text{locate}(o, \text{pnth}(\text{s}(n), P))$

page_dimension(p)
Returns the list of natural numbers of the sizes of the page object p in all its dimensions. E.g., a two dimensional table with m lines and a maximum of n columns in one of these lines has a dimension of (m, n). This means that the smallest two dimensional cube around it will have hight m and breadth n. A three dimensional table with dimension (m, n, p) will fill a cube of depth p. If the objects are not atomic, the element-wise maximum of its dimensions will be appended at the end of the dimension list of the table. Generally speaking, a page object represented as an Mkpage-node tree of depth d has the dimension $(n_1, \ldots, n_d)$ where n_i is the maximum number of children of a node at depth i. Note that it uses the function 'map_default' from LISTPAIR on page 26.

$$
\begin{array}{l}
\text{page\_dimension}(p) = [] \\
\quad \Longleftarrow \ \text{atomic?}(p) = \text{true} \\
\text{page\_dimension}(p) = \text{length}(P) :: \text{page\_list\_dimension}(P) \\
\quad \Longleftarrow \ \text{atomic?}(p) = \text{false} \wedge p = \text{Mkpage}(s, P, att) \\
\text{page\_list\_dimension}([]) \quad = \quad [] \\
\text{page\_list\_dimension}(p :: P) = \\
\quad \text{map\_default}(0, 0, \max, \text{page\_dimension}(p), \text{page\_list\_dimension}(P))
\end{array}
$$

`-- Editing Functions --`

ch_struct$(s', p) = p'$
p' is the page containing the same documents and attributes as p, but with a different structure s'.

$$\text{ch\_struct}(s', \text{Mkpage}(s, P, att)) = \text{Mkpage}(s', P, att)$$

mklist$(n) = p$
p is a list with n items, containing an empty page $[\![]\!]$ in every item.

$$\text{mklist}(n) = \text{Mkpage}(\text{Page\_list}, \text{repeat}(n, [\![]\!]), []_{\text{Att}})$$

mktable$(m, n) = p$
p is a $m \times n$-table, containing an empty page $[\![]\!]$ in every cell. mktableline(n) is an auxiliary function for it.

$$
\begin{array}{l}
\text{mktable}(m, n) = \text{Mkpage}(\text{Table}, \text{repeat}(m, \text{mktableline}(n)), []_{\text{Att}}) \\
\text{mktableline}(n) = \text{Mkpage}(\text{Tableline}, \text{repeat}(n, [\![]\!]), []_{\text{Att}})
\end{array}
$$

$\text{place\_at}(p', o, p) = p''$
If the location o occurs in the page p, then p'' is the page p with its part at location o replaced with the page p'. If o does not exist in p because a node ν in p has not enough children, then p is first extended with sufficiently many child nodes for ν. The type of these child nodes may depend on the parent node ν. E.g., if the parent node is a table then the child nodes will be of type table-line. If no special knowledge is given, the child nodes will be simply of type empty page ('$[\![]\!]$'). The default child node is the last argument of a helper function 'place_at_help' that is very similar to 'place_at' but works on children lists instead of single nodes.

$$
\begin{aligned}
&\text{place\_at}(p', [], p) = p'\\
&\text{place\_at}(p', n :: o, \text{Mkpage}(s, P, att))\\
&\qquad\qquad = \text{Mkpage}(s, \text{place\_at\_help}(p', n, o, P, \text{mktableline}(0)), att)\\
&\qquad\qquad \Longleftarrow\ s = \text{Table}\\
&\text{place\_at}(p', n :: o, \text{Mkpage}(s, P, att))\\
&\qquad\qquad = \text{Mkpage}(s, \text{place\_at\_help}(p', n, o, P, [\![]\!]), att)\\
&\qquad\qquad \Longleftarrow\ s \neq \text{Table}
\end{aligned}
$$

$$
\begin{aligned}
\text{place\_at\_help}(p', \text{s}(0), o, p :: P, p'') &= \text{place\_at}(p', o, p) :: P\\
\text{place\_at\_help}(p', \text{s}(\text{s}(n)), o, p :: P, p'') &= p :: \text{place\_at\_help}(p', \text{s}(n), o, P, p'')\\
\text{place\_at\_help}(p', \text{s}(0), o, [], p'') &= \text{place\_at}(p', o, p'') :: []\\
\text{place\_at\_help}(p', \text{s}(\text{s}(n)), o, [], p'') &= p'' :: \text{place\_at\_help}(p', \text{s}(n), o, [], p'')
\end{aligned}
$$

$\text{insert\_at}(p', o, p) = p''$
p'' is the page after p' has been inserted at location o if o exists in p.

$$\text{insert\_at}(p', o, p) = \text{place\_at}(p', o, p) \ \Longleftarrow\ \text{has\_location?}(o, p) \ = \ \text{true}$$

$\text{add\_attribute}(att, p) = p'$
p' is the page after p is enriched with the attributes att.

$$\text{add\_attribute}(att', \text{Mkpage}(s, P, att)) = \text{Mkpage}(s, P, \text{concat}(att', att))$$

$\text{del\_attribute}(att, p) = p'$
p' is the page after the attributes att are removed from p.

$$\text{del\_attribute}(att', \text{Mkpage}(s, P, att)) = \text{Mkpage}(s, P, \text{remove}(att', att))$$

A.7.2 HyperMedia Document

HMD_ADDR = STRING `then`
`vissorts`
 hmd_addr

HMD = PAGE `and` HMD_ADDR `and`
HD[document↦PAGE.page,
location↦PAGE.page_location,
embed_link_ok?↦PAGE.embed_link_page_ok?,
addr↦HMD_ADDR.hmd_addr] `and`
MAPSET[entry1↦anchor, entry2↦PAGE.page_location] `and`
MAPSET[entry1↦link, entry2↦link]

`then`
`vissorts`
 hmd = hd(PAGE.page, PAGE.page_location, HMD_ADDR.hmd_addr)
`defuns`
 − − − `Editing Functions` − −−
 place_at. hmd × page_location × hmd × hmd_addr → hmd
 insert_at. hmd × page_location × hmd × hmd_addr → hmd
 combine_link. hmd_addr × hmd_addr × hmd_addr × set(link) × set(link) → set(link)
 sinkloc. page_location × anchor → anchor
`vars` m, n. nat
 p, p'. page
 h, h', h''. hmd
 o, o'. page_location
 A, A'. function(anchor_id,anchor)
 L, L'. set(link)
 a, a', a''. hmd_addr
 t. anchor_type
 att. att_anchor

`axioms`

`-- Editing Functions --`

$\text{place\_at}(h, o, h', a'') = h''$
Replaces the part of hyperdocument h' at location o with hyperdocument h, resulting in a new hyperdocument h'' under address a''. This is only possible when the names of the anchors in h and h' are disjoint and when h' does not have any anchors in the part replaced with h. sinkloc(o, c) is an auxiliary function that appends o to the front of the location of the anchor c, i.e. it lets c sink below the location o. Note that we can use 'sinkloc' as a unary function in the definition of 'place_at' because we consider all functions to be curried and argument tupling just to be syntactic sugar. The functions 'map_range' and 'union' are from FUNCTION[domain↦anchor_id, range↦anchor] from HD. Note that the application of 'map_range' is unproblematic here because the domains of A and A' are required to be disjoint. 'combine_link' is an auxiliary function that changes all references of links to h and h' to refer to h''. It is defined via 'map_set' from MAPSET[entry1↦link, entry2↦link]. Moreover, 'replace_uri_li' from LINK is called (like 'sinkloc') with one argument less than defined, in order to yield a function of type 'link → link'. Finally, note that in the condition of the definition of 'place_at' the 'map_set' is from MAPSET[entry1↦anchor, entry2↦PAGE.page_location] and the 'exists' is from SET[entry↦PAGE.page_location], which again is part of MAPSET[entry1↦anchor, entry2↦PAGE.page_location].

$$
\begin{aligned}
&\text{place\_at}(\text{Mkhd}(p, A, L, att, a), o, \text{Mkhd}(p', A', L', att', a'), a'') = \\
&\quad \text{Mkhd}(\text{place\_at}(p, o, p'), \\
&\qquad\quad \text{union}(\text{map\_range}(\text{sinkloc}(o), A), A'), \\
&\qquad\quad \text{combine\_link}(a, a', a'', L, L'), \\
&\qquad\quad \text{concat}(att, att'), \\
&\qquad\quad a'') \\
&\quad \Longleftarrow \;\; \text{dom}(A) \cap \text{dom}(A') = \{\} \wedge \\
&\qquad\quad \text{exists}(\text{is\_proper\_prefix}(o), \text{map\_set}(\text{get\_location}, \text{ran}(A))) = \text{false}
\end{aligned}
$$

$$\text{sinkloc}(o, \text{Mkanchor}(o', t, att)) = \text{Mkanchor}(o@o', t, att)$$

$$
\begin{aligned}
&\text{combine\_link}(a, a', a'', L, L') = \\
&\quad \text{map\_set}(\text{replace\_uri\_li}(\text{embed}(a'), \text{embed}(a'')), \\
&\qquad\qquad \text{map\_set}(\text{replace\_uri\_li}(\text{embed}(a), \text{embed}(a'')), \\
&\qquad\qquad\qquad L \cup L') \quad)
\end{aligned}
$$

$\text{insert\_at}(h, o, h', a'') = h''$
h'' is the hypermedia-document with address a'' after h has been inserted at location o into hypermedia-document h', provided that o exists in h.

$$\text{insert\_at}(h, o, h', a'') = \text{place\_at}(h, o, h', a'') \;\Longleftarrow\; \text{has\_location?}(o, \|h'\|) = \text{true}$$

A.8 Frameset Document Level

The following specifications are essentially incomplete and have to be completed in the future!!!

A.8.1 Chapter

CHAPTER_SYMBOLS =
`vissorts`
 chapter_symbols

CHAPTER = HMD `and` CHAPTER_SYMBOLS `and` ATT_CHAPTER `and`
 TREE[entry↦chapter_struct] `and`
 LIST[entry↦chapter] `and`
 LIST[entry↦nat] `then`
`vissorts`
 chapter
 chapter_struct
 fsd_location = list(nat)
`constructs`
 Horiz_frameset, Vert_frameset, Alt_frameset : fsd_struct

A.8.2 FrameSet Document

FSD_ADDR = STRING `then`
`vissorts`
 fsd_addr

FSD = CHAPTER `and` FSD_ADDR `and`
 HD[document↦CHAPTER.chapter,
 location↦CHAPTER.chapter_location,
 embed_link_ok?↦CHAPTER.include_link_chapter_ok?,
 addr↦FSD_ADDR.fsd_addr]

`then`
`vissorts`
 fsd = hd(CHAPTER.chapter, CHAPTER.chapter_location, FSD_ADDR.fsd_addr)

A.9 Site Level

The following specifications are essentially incomplete and have to be completed in the future!!!

A.9.1 Book

BOOK_SYMBOLS =
`vissorts`
book_symbols

BOOK = FSD `and` BOOK_SYMBOLS `and` ATT_BOOK `and`
TREE[entry↦book_struct] `and`
LIST[entry↦book] `and`
LIST[entry↦nat] `then`
`vissorts`
book
book_struct
book_location = list(nat)
`constructs`
sitemap : book_struct

A.9.2 Site

SITE_ADDR = STRING `then`
`vissorts`
site_addr

SITE = BOOK `and` SITE_ADDR `and`
HD[document↦BOOK.book,
location↦BOOK.book_location,
embed_link_ok?↦BOOK.include_link_book_ok?,
addr↦SITE_ADDR.site_addr]

`then`
`vissorts`
site = hd(BOOK.book, BOOK.book_location, SITE_ADDR.site_addr)

References

[BCM 96] T. Bienz, R. Cohn, J. Meehan (1996). *Portable Document Format Reference Manual.* Version 1.2, Adobe Systems Incorporated. `http://partners.adobe.com/supportservice/devrelations/PDFS/TN/PDFSPEC.PDF` (May 14, 1999).

[BFI 98] T. Berners-Lee, R. Fielding, U.C. Irvine, L. Masinter (1998). *Uniform Resource Indentifiers (URI): Generic Syntax.* RFC 2396.

[BH 92] Paul de Bra, Geert-Jan Houben (1992). *An Extensible Data Model for Hyperdocuments.* Proc. ACM Conf. on Hypertext'92, pp. 222–231. `http://wwwis.win.tue.nl/~debra/echte92/final.ps` (March 29, 1999).

[BHW 99] Paul de Bra, Geert-Jan Houben, H. Wu (1999). *AHAM: A Dexter-based Reference Model for Adaptive Hypermedia.* Proc. ACM Conf. on Hypertext '99, pp. 147–156.

[Bus 45] Vannevar Bush (1945). *As we may think.* The Atlantic **176(1)**, pp. 101–108. `http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm` (March 30, 1999).

[Dob 96] E.-E. Doberkat (1996). *A Language for Specifying Hyperdocuments.* Software — Concepts and Tools **17**, pp. 163–173, Springer.

[Eng 83] Douglas C. Engelbart (1984). *Authorship Provisions in AUGMENT.* COMPCON '84 Digest: Proceedings of the COMPCON Conference, San Francisco, pp. 465–472. `http://www.bootstrap.org/oad-2250.htm` (Nov. 6, 1999).

[GQV 98] R. Guetari, V. Quint, I. Vatton (1998). *Amaya: an Authoring Tool for the Web.* MCSEAI'98 International Conference. `http://www.inrialpes.fr/opera/people/Ramzi.Guetari/Papers/Amaya.html` (May 17, 1999).

[GT 94] K. Grønbaek, R. H. Trigg (1994). *Design Issues for a Dexter-Based Hypermedia System.* Comm. ACM **37(2)**, pp. 40–49, ACM Press.

[HBR 94] Lynda Hardman, Dick C.A. Bulterman, Guido van Rossum (1994). *The Amsterdam Hypermedia Model.* Comm. ACM **37(2)**, pp. 50–62, ACM Press.

[HS 90] F. Halasz, F. Schwartz (1990). *The Dexter Hypertext Reference Model.* Proc. Hypertext Standardization Workshop, National Institute of Technology (NIST), pp. 95–133.

[ISB 95] T. Isakowitz, E. A. Stohr, P. Balasubramanian (1995). *RMM: A Methodology for Structured Hypermedia Design.* Comm. ACM **38(8)**, pp. 34–44, ACM Press.

[KW 96] Ulrich Kühler, Claus-Peter Wirth (1996). *Conditional Equational Specifications of Data Types with Partial Operations for Inductive Theorem Proving.* SEKI-Report SR–96–11, FB Informatik, Univ. Kaiserslautern. Short version in: 8^{th} RTA 1997, LNCS 1232, pp. 38–52, Springer. `http://www.ags.uni-sb.de/~cp/p/rta97` (Oct. 13, 1999).

[LH 99] David Lowe, Wendy Hall (1999). *Hypermedia & the Web. An engineering approach.* Wiley.

[LP 92] Mihaly Lenart, Ana Pasztor (1992). *Knowledge Based Specifications of the Design Process Using Many-Sorted Logic.* Ulam Quarterly **1(4)**. `http://www.ulam.usm.edu/VIEW1.4/pasztor.ps` (May 17, 1999).

[LW 94] Rüdiger Lunde, Claus-Peter Wirth (1994). *ASF$^+$ — eine ASF-ähnliche Spezifikationssprache.* SEKI-Working-Paper SWP–94–05 (SFB), FB Informatik, Univ. Kaiserslautern. `http://www.ags.uni-sb.de/~cp/p/swp9405` (Oct. 13, 1999).

[MK 95] A. Mester, H. Krumm (1995). *Composition and Refinement Mapping based Construction of Distributed Algorithm.* Proc. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Aarhus.

[OE 95] J. van Ossenbruggen, A. Eliens (1995). *The Dexter Hypertext Reference Model in Object-Z.* Unpublished Paper, Vrije Universiteit Amsterdam. `http://www.cs.vu.nl/~dejavu/papers/dexter-full.ps.gz` (May 17, 1999).

[Pad 2000] Peter Padawitz (2000). *Swinging Types = Functions + Relations + Transition Systems.* Theoretical Computer Sci. **243**, pp. 93–165, Elsevier.

[Qui 97] V. Quint (1997). *The Languages of Thot.* INRIA 1996, Version April 1997. `http://www.eda.bg/docs/packages/amaya/languages.html` (May 17, 1999).

[Sal 96] F.A. Salustri (1996). *A formal theory for knowledge-based product model representation.* 2nd IFIP WG 5.2 Workshop on Knowledge Intensive CAD, Carnegie-Mellon Univ., pp. 59–78, Chapman & Hall. `http://salustri.esxf.uwindsor.ca/~fil/Papers/kicII/reprint.html` (May 17, 1999).

[W3C 98a] W3C (1998). *HTML 4.0 Specification.* W3C Recommendation, revised on 24-Apr-1998. `http://www.w3.org/TR/1998/REC-html40-19980424` (May 17, 1999).

[W3C 98b] W3C (1998). *Document Object Model (DOM) Level 1 Specification.* W3C Recommendation 1 October, 1998. `http://www.w3.org/TR/REC-DOM-Level-1` (May 17, 1999).

[W3C 98c] W3C (1998). *Extensible Markup Language (XML) 1.0.* W3C Recommendation 10-February-1998. `http://www.w3.org/TR/REC-xml` (August 1, 1999).

[W3C 98d] W3C (1998). *XML Linking Language (XLink).* W3C Working Draft 3-March-1998. `http://www.w3.org/TR/WD-xlink-19980303` (August 1, 1999).

[WD 99] Jörg Westbomke, Gisbert Dittrich (1999). *Ein Ansatz zur formalisierten Beschreibung von Hypermediadokumenten in XML.* Report 708/1999, FB Informatik, Univ. Dortmund. `http://lrb.cs.uni-dortmund.de/~westbomk/Homepage/Forschbericht_HMD-XML.pdf` (Oct. 14, 1999).