Journal of Universal Computer Science, vol. 8, no. 10 (2002), 892-912
submitted: 24/4/02, accepted: 15/10/02, appeared: 28/10/02

An Object-oriented Approach to Design, Specification, and Implementation of Hyperlink Structures Based on Usual Software Development

Alexander Fronk
(Software-Technology, University of Dortmund, Germany
fronk@LS10.de)

Abstract: Different models and methodologies for the development of hypermedia systems and applications have emerged in the recent years. Software-technical methods and principles enriched with ideas mainly driven from the applications' needs are often sponsor to those models and methodologies. Hence, they deal with very specific problems occurring in the hypermedia domain, thereby extending design notations like UML or State Charts and adapting them to modeling this domain. In the present paper, we propose a very usual software-technical approach to the development of hyperlink structures which form the basis for navigation in hyperdocuments. Our approach uses standard UML, algebraic specification and object-oriented implementation to cover the construction of hyperlink structures, from design through to specification and realization. We thereby equate the development of hypermedia documents with usual software development. Instead of adopting software-engineering and notations to hypermedial concerns, we adopt the latter to the former and show the advantages of this approach.
Key Words: Hypermedia, Software Engineering, Systems Development, Methodologies, Object-oriented Languages
Category: I.7.2, D.2, K.6.1, D.2.10, D.3.2

1 Introduction

A hypermedia document, or hyperdocument for short, is understood as a collection of media objects such as texts, graphics, videos, etc. which are connected to each other in a non-linear fashion, that is, they are hyperlinked. In [Fronk, 2001], an object-oriented document description language for implementing such hyperdocuments, *DoDL* for short [Doberkat, 1996b, Doberkat, 1998], was discussed. The object-oriented implementation with *DoDL* has shown that hyperdocuments can easily be comprehended as programs, or more generally, as software, and hence inherently possess both document and program qualities. This document/program-dualism (c.f. [Fronk, 2000]) can clearly be observed by the fact that hyperdocuments not only offer non-linearly related media and hence hyperlinked information. Moreover, navigational aspects, i.e. the hyperlinks of the document, called its *hyperlink structure*, are entirely encoded within the hyperdocument itself. If we understand any browser as a runtime environment allowing navigation, a hyperdocument encompasses control structure. Hyperdocuments are often treated merely as documents leaving their characteristics as programs nearly completely aside.

Depending on the specific task a hyperdocument has to satisfy, its media objects may underlie frequent change. For example, digital libraries, information kiosks, or hypermedia information systems showing environmental conditions such as air temperature, water-levels, or radio activity, etc. need to be updated or supplemented periodically with the latest measuring results. Thereby, the link structure has to be adapted to the new data. In the view of programs, the control structure has to be reimplemented. If, however, the link structure depends on positions within these media objects, and if positions can be described without referring to specific content (for example, if a certain occurrence of a city name, regardless of where exactly this name occurs within the media object, is always linked to a map showing some city details), hyperlinks can be captured abstractly without referring to concrete media objects. In the view of programs again, the control structure can be specified. Hence, an adequate implementation of a document's link structure can be reused, if it is given without reference to concrete data. Analogously, programs are implemented without referring to their concrete input data. Exactly here, the object-oriented implementation of link structures finds its place.

In HTML/XML, hyperlinks and media objects are assembled within one single document. Using *tags* directly embedded within media objects makes it impossible to separate the description of hyperlinks from them. Our approach allows in contrast to describe the link structure abstractly and separately from media objects. Thereby, we strictly follow the principle of *separation of concerns*, and contribute to maintaining hyperdocuments [Fronk, 1999] immediately on the level of maintaining software. A suitable compiler system allows to generate a hyperdocument by binding concrete media objects to the description of a link structure. Thus, our approach is open to changing media objects frequently without neither touching the hyperdocument itself nor its implementation. Some advantages become obvious here: By exchanging the values given in a binding and hence obeying the principle of *locality*, we are able to generate arbitrary many hyperdocuments underlying a certain link structure; further, the code is written in an object-oriented fashion and is thus open to inheritance, polymorphism, overloading, etc., and thus to high-level implementation of hyperdocuments. More details can be found in [Fronk and Pleumann, 1999] and [Pleumann, 2000].

The notion of position is vital to obtain an object-oriented description of a hyperdocument. Therefore, media objects were given a simple tree-like structure which allows us to easily determine positions serving as link anchors, and thereupon hyperlinks as pairs of positions. By introducing positions only based on this structure, we do not need to use coordinate systems laid upon media objects, and establish a uniform mechanism fitting to any kind of media object. This idea is vital for the description of hyperdocuments by using an object-oriented approach, which contrasts very much to the common HTML/XML implementation of hyperdocuments.

The construction of hyperdocuments is generally not limited to structuring the media objects involved and their link structure (c.f. [Lowe and Hall, 1999], Chapter 9). Moreover, an overall view on *hypermedia systems* has to be taken into account including usability and presentation of hyperdocuments. For the time being, we restrict ourselves to the development of hyperlink structures. Hence, the visual appearance of media objects, i.e. their layout, is not a concern here, although we consider it as a very important aspect of a hyperdocument: layout is undoubtedly responsible for usability and acceptance. In addition to layout, the presentation of a hyperdocument encompasses the (animated) arrangement of its media objects in time and space (c.f. [Gloor, 1997], page 264). For our approach, capturing elements and their relative positions to each other is more important than their visual representation. This allows to talk about the composition of media objects, about the notion of subdocuments, and about positions of subdocuments in a document under consideration. Further, we do not discuss models for hypermedia development (c.f. [Tochtermann, 1994]). Moreover ,the goal of this paper, which is an extension of parts of the author's Ph.D. thesis, is to propose a development process for hypermedia documents, which is based on software-technical methods and principles employed for usual object-oriented software construction. We discuss a process that covers design, specification, implementation, testing and maintenance of hyperdocuments. The lack of testability and maintenance lowers the quality of a hyperdocument [Lowe and Hall, 1999]. Our approach supports these tasks and hence contributes to product quality.

The paper is organized as follows. First, we briefly mention some related work in [Section 2]. To comprehend the basic correlations between usual software development and hyperdocument development, we proceed backwards through a software life cycle, from implementation through to specification and design. We discuss a simple hypermedial car cockpit as an example, and show its implementation first [see Section 3], prior to considering its specification in [Section 4]. A more complex example of such a car cockpit (see [Fronk, 2001], Chapter 13 for details) is used to motivate its design with standard UML [Section 5] seamlessly leading to the discussed process encompassing specification and implementation. We conclude in [Section 6] and sketch further work.

2 Related Work

In [Fronk, 2002b], we discuss some work related to structuring media objects and locating positions in them, such as the Trellis model [Stotts and Furuta, 1989, Stotts et al., 1998], XPath and XLink (c.f. [Deitel et al., 2001]) or the COMPOSITE design pattern [Gamma et al., 1995]. Chang and Shih give a detailed survey of hypermedia process models [Chang and Shih, 2002] developed in the recent years, covering different phases in the life cycle of hypermedia applications. Costagliola et al. [Costagliola et al., 2002] survey hypermedia development methodologies, such as HDM [Garzotto and Paolini, 1993], or RMM [Isakowitz et al., 1995]. OOHDM, an object-oriented hypermedia design method, is introduced by [Schwabe and Rossi, 1998]. This methodology distinguishes between the design of content, link structure, and views, followed by the implementation. These four phases capture content and links abstractly by so called *entity* and *link types*. Entities are structured stepwise before they are connected by concrete links. Refinement steps are allowed and lead to a top-down incremental, prototype-based process model. Our development process is *not* a process model but can be embedded into any such model known in software-engineering (c.f. [Ghezzi et al., 1991]).

Another object-oriented modelling process, called OMMMA, is proposed in [Engels and Sauer, 2002]. This approach introduces a language, OMMMA-L, which extends UML for modeling many different aspects of hypermedia applications. More important, OMMMA provides a method description on how to use this language in a prototype-based software development process. As an example, they discuss an automotive information system. The OMMMA-L model developed is very close to our UML-model of the hypermedial car cockpit discussed in the present paper. In quite contrast, we do not extend UML.

The Dexter model, as described in [Halasz and Schwartz, 1994], consists of three layers, the run-time layer, the storage layer, and the within-component layer. The latter is purposely left unspecified to allow for different structuring mechanisms as well as different media types. Our tree-like structures can be seen as a formalization of this layer. The run-time layer is not a concern in our approach, since presentation and layout are not crucial for discussing the construction of hyperlink structures.

The storage layer, however, is of greater interest. The entity used here is a *component*. It is divided into an information component and a base component. The former possesses a unique component identifier, *UID* for short, and a set of *anchors*. The latter consists of atomic and compositional media objects, as well as links. Anchors serve as sources and sinks for links, and are pairs of identifiers and values. The value describes a certain position within a component. A *specifier* consists of a component specification and an anchor identifier (for the time being, we do not consider direction and presentation). A link is a sequence of specifiers, such that arbitrary linkage is feasible.

In our approach, links are modeled as pairs on positions. The link set thus forms a relation. Hence, we allow for arbitrary link structures, too. Positions are evaluated within media objects. The variable addressing a media object can be seen as a *UID* in the Dexter model, since positions are unambiguously assigned to media objects. Our approach offers operations to determine positions serving as an anchoring mechanism, and forming the interface between the storage and within-component layer. Our positions directly correspond to anchors, since an anchor value is given by a path within a media object's tree structure, and the anchor identifier relates to the node addressed by the path, or, equally, its unique number within a tree representation.

Similar to the Dexter model, concrete positions depend on concrete media objects and their structure, and the measuring system is the media object itself. In [Hardman et al., 1994], the notions of *time* and *context* are added to the Dexter model making it applicable to more complex media. Our approach is open to extensions by adding further operations and structuring mechanisms yielding to determining positions in more complex structures than trees.

3 Object-Oriented Implementation of Hyperdocuments

Throughout this paper, we discuss the realization of a hypermedial car cockpit. With the help of this example, we show how media objects and hyperdocuments are structured, designed, specified, and implemented. We chose a simple object-oriented language, *DoDL*. This language has especially been tailored for describing hyperlink structures in an object-oriented fashion. The languages only contains those elements necessary for this domain. This allows to keep things simple and comprehensive, and to concentrate on the development process rather than on technical details. We introduce *DoDL* by the aforementioned example shown in [Figure 1].

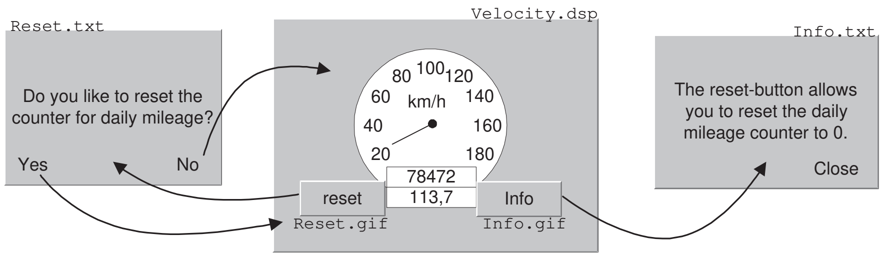

Figure 1: A simple hypermedial car cockpit

For simplicity, the sample car cockpit consists of a speedometer and a mileage counter, `Velocity.dsp`, together with two buttons, `Reset.gif` and `Info.gif`, attached to it. Each button is linked to a text. The reset-button resets the display. Pressing this button corresponds to traversing a hyperlink leading to an informative text, `Reset.txt`, announcing the reset procedure which can be canceled or confirmed. The info-button calls for some information, `Info.txt`, about the instrument's behavior. For the time being, we do not concentrate on the cockpit's functionality. We focus on composing media objects and on hyperlink structures first. [Section 5.2] gives details on how to implement functionality making this cockpit work.

We structure media objects in a tree-like manner, hereby defining a representational model for our hyperdocuments. In our approach, *atomic* media objects form the smallest units by which we compose larger ones, called *compositional* media objects.

Definition 1. Compositional media objects are recursively defined as follows:

1. Each atomic media object is a compositional media object.
2. Let m_i, $i = 1 \dots n$, $n \geq 2$, be compositional media objects, which need not to be pairwise disjoint. Then, $m = \mathit{compose}(m_1, \dots, m_n)$ is a compositional media object.

Definition 2. Let $m = \mathit{compose}(m_1, \dots, m_n)$ be a media objects. Each m_i, $i = 1 \dots n$, is called a **part** of m. The set of **constituents** of m is the set of all its parts. The set of **components** of m is recursively defined as follows:

$$\mathit{components}(m) := \begin{cases} \{m\}, \quad \text{if } m \text{ is atomic} \\ \mathit{constituents}(m) \cup \bigcup_{i=1}^{n} \mathit{components}(m_i), \\ \qquad\qquad \text{if } m = \mathit{compose}(m_1, \dots, m_n), n \geq 2 \end{cases}$$

In the sequel, we refer to atomic media objects as *atoms*, and to compositional media objects as *compositionals*. The tree representation of media objects is given as follows:

– An atom, a, is a tree consisting of a root node only; the root is labeled with a.

– A compositional, $m = \mathit{compose}(m_1, \dots, m_n), n \geq 2$, is a tree the root node of which is labeled with m. The root has exactly n children labeled with m_i, such that m_i is the i-th son of the root.

Based on this structure, paths uniquely address components within a media object. A **position** in m is such a path, a **link** a pair of positions:

Definition 3. Let $m = \mathit{compose}(m_1, \dots, m_n), n \geq 2$, and m' be two media objects. A **path** in m to m', $\mathit{path}(m, m')$ for short, is recursively defined as follows:

$$path(m, m') := \begin{cases} \epsilon_m, & \text{if } m \equiv m' \\ (i \frown path(part(m,i), m'))_m, & \\ \quad \text{if } m' \in components(part(m,i)), i = 1 \ldots n, & \\ \perp, & \text{else} \end{cases}$$

We use ϵ_m to denote the empty path referring to m itself. If m' occurs more than once in m, several paths to m' exist. For more details on compositional media objects, the reader is referred to [Fronk, 2001, Fronk, 2002b].

We define an interface to media objects which allows to compose them and to access positions within them. The interface provides the following functionality predefined in *DoDL*. A `compose`-method allows to create compositional media objects and is defined as in [Definition 1]. The method expects arbitrary many media objects which may be atomic or composed. To access positions, we restrict ourselves to referring the beginning of a media object, `getBegin`, the end, `getEnd`, and the set of occurrences, `getOcc`, of a specific component within a media object. Further methods can easily be defined if required by an application. Henceforth, let m and m' be two media objects. The methods' semantics are given in the sequel.

Definition 4. The **beginning** of m, $getBegin(m)$ for short, is the path defined as follows:

$$getBegin(m) := \begin{cases} \epsilon_m, & \text{if } m \text{ is atomic} \\ 1 \frown getBegin(part(m,1))_m, & \text{else} \end{cases}$$

The **end** of m, $getEnd(m)$ for short, is the path defined as follows:

$$getEnd(m) := \begin{cases} \epsilon_m, & \text{if } m \text{ is atomic} \\ n \frown getEnd(part(m,n))_m, & \text{if } m = compose(m_1, \ldots, m_n) \end{cases}$$

These paths lead to the left-most and the right-most atom of m, respectively. Since we allow a component to occur more than once in m, we need the set of all its occurrences:

Definition 5. The **set of occurrences** of m' in m, $getOcc(m, m')$ for short, is defined as follows:

$$getOcc(m, m') := \{p_m \mid p_m \text{ is a path in } m \text{ to } m'\}$$

Each position $p_m \in getOcc(m, m')$ is called **occurrence** of m' in m.

```
class Cockpit is
   documents  display, resetbutton, infobutton,
              resettext, infotext:             MedObj;
   construct
      MedObj composeDisplay(void){             // compose media objects
         return compose(resetbutton, display, infobutton); }

      set(link) linkDisplay(void){
         // a link from resetbutton to resettext
         composeDisplay().getBegin().setLink(resettext.getBegin());
         // a link from infobutton to infotext
         composeDisplay().getEnd().setLink(infotext.getBegin()); }

      set(link) linkResetText(void){
         resettext.getOcc("Yes").linkAll(display.getBegin());
         resettext.getOcc("No").linkAll(display.getBegin()); }
end Cockpit;
```

Listing 1: Implementing the cockpit

Notice that *getOcc*(m, m') is empty if and only if m' does not occur in m.

To define links as pairs of positions, we use a method `setLink`. The positions contained in a set of positions determined by *getOcc* may serve as anchors of links leading to the same target position. To create such a $(n : 1)$-relation, we use a `linkAll`-method.

We use *DoDL*-classes to assemble media objects and links between them. That is, class attributes, introduced by the keyword `documents`, refer to media objects, methods, introduced by the keyword `construct`, are responsible to construct links between the media objects given in the `documents`-section only. Local and generic classes can also be defined. Together with relations between classes, such as aggregation and inheritance, these concepts are employed to structure code object-oriently. The reader is referred to [Fronk, 2001] and [Fronk, 2002a] for more details on *DoDL*.

A hyperlink structure like that of the cockpit discussed above can be implemented in *DoDL* as shown in [Listing 1]. We define the beginning of a text as its first word, the end as its last, and the occurrence of a word, w, corresponds to n, if w is the n-th word in the text. For graphics, we also provide suitable implementations of these methods. That is, the realization of positions depends on the type of a media object.

The type `MedObj` represents media objects. For the time being, we support simple text, graphics, and applets to display some information given as input to the applet. Applets are here understood as a kind of animated graphics. Hence, the type `MedObj` has subtypes `Text`, `Graphics`, and `Display`, predefined in *DoDL* and equipped with a suitable realization of positions.

Class `Cockpit` introduces five media objects the place holders for which are

```
binding Cockpit is
    display: Display = Velocity.dsp;
    resetbutton: Graphics = Reset.gif;
    infobutton: Graphics = Info.gif;
    resettext: Text = Reset.txt;
    infotext: Text = Info.txt;
end;
```

Listing 2: Binding concrete media objects

collected in the `documents`-section. Further, three methods are implemented to create the hyperlink structure shown in [Figure 1]. Method `composeDisplay` creates a compositional media object using three atoms, `resetbutton`, `display`, and `infobutton`. Method `linkDisplay` creates a link from the reset button (or, equally, from the beginning of the media object composed in `composeDisplay`) to the first word of the reset text, and from the info button (or, equally, from the end of the media object composed in `composeDisplay`) to the first word of the info text. The method thus yields a set of links. The other method works analogously without the need to know the concrete structure or content of any media object. If, for example, a text for `resettext` is given that does not contain the words "Yes" and "No", method `linkResetText` cannot create any link. Testing the hyperdocument can detect such shortcomings and is briefly sketched in the next section.

To assign concrete media objects to the document variables given, we use typed bindings that assign each document variable an initial value. [Listing 2] shows which concrete media objects are assigned to which document variables. For technical simplicity, we assume that these media objects are taken from a repository, and that their content is as shown in [Figure 1].

This example clearly shows that arbitrarily many different hyperdocuments can be generated by exchanging the binding. Maintaining the hyperdocument is in this approach equated with maintaining the code implemented in usual software development.

Concluding this section, we define hyperdocuments as follows.

Definition 6. A **hyperdocument** is a triplet $\mathcal{H} = (\mathcal{M}, \mathcal{P}, \mathcal{L})$, where

- $\mathcal{M}$ is a non-empty set of composed media objects,
- $\mathcal{P}$ is any arbitrary set of positions, p_m, in media objects $m \in \mathcal{M}$,
- $\mathcal{L}$ is any arbitrary set of links, $(p_m, p'_{m'})$, such that $p_m, p'_{m'} \in \mathcal{P}$.

4 Object-Oriented Specification of Hyperdocuments

Both the composition of media objects and accessing positions is algebraically specified. That is, we provide an abstract data type, in the sequel called $HDOC$, an implementation of which was sketched in the previous section. [Specification 1] shows some excerpts of a specification $HDOC$ elaborated in [Fronk, 2001] and [Fronk, 2002b].

$\Sigma_{HDOC} =$

sorts $\mathit{MedObj}, \mathit{position}, \mathit{link}, \mathit{set}(\mathit{position}), \mathit{set}(\mathit{link})$

opns
$\mathit{compose}^n : \mathit{MedObj}^n \to \mathit{MedObj}, \quad n \geq 2$
$\mathit{part}_n : \mathit{MedObj} \times \mathit{nat} \to \mathit{MedObj}, \quad n \geq 2$
$\epsilon : \mathit{MedObj} \to \mathit{position},$
$\mathit{getBegin} : \mathit{MedObj} \to \mathit{position},$
$\mathit{getEnd} : \mathit{MedObj} \to \mathit{position},$
$\mathit{getOcc} : \mathit{MedObj} \times \mathit{MedObj} \to \mathit{set}(\mathit{position}),$
$\mathit{setLink} : \mathit{position} \times \mathit{position} \to \mathit{link},$
$\mathit{linkAll} : \mathit{set}(\mathit{position}) \times \mathit{position} \to \mathit{set}(\mathit{link}),$
$\mathit{source} : \mathit{link} \to \mathit{position},$
$\mathit{end} : \mathit{link} \to \mathit{position}$

vars $m, m_1, \ldots, m_n : \mathit{MedObj}, p, q : \mathit{position}, \mathit{pset} : \mathit{set}(\mathit{position}), l : \mathit{link}$

axms

$$\begin{aligned}
\mathit{part}_n(\mathit{compose}^n(m_1, \ldots, m_n), i) &= m_i, \\
\mathit{compose}^n(\mathit{part}_n(m, 1), \ldots, \mathit{part}_n(m, n)) &= m, \\
\mathit{getBegin}(m) &= \epsilon, \\
\mathit{getEnd}(m) &= \epsilon, \\
\mathit{getBegin}(\mathit{compose}^n(m_1, \ldots, m_n)) & \\
&= 1 \frown \mathit{getBegin}(\mathit{part}_n(\mathit{compose}^n(m_1, \ldots, m_n), 1)), \\
\mathit{getEnd}(\mathit{compose}^n(m_1, \ldots, m_n)) & \\
&= n \frown \mathit{getEnd}(\mathit{part}_n(\mathit{compose}^n(m_1, \ldots, m_n), n)) \\
(l \in \mathit{linkAll}(\mathit{pset}, p)) &= (\mathit{source}(l) \in \mathit{pset} \wedge \mathit{sink}(l) = p), \\
\mathit{source}(\mathit{setLink}(p, q)) &= p, \\
\mathit{sink}(\mathit{setLink}(p, q)) &= q
\end{aligned}$$

Specification 1: Specifying compositional media objects, positions, and links

[Specification 1] essentially specifies the functionality discussed in the previous section. For simplicity, we assume a specification for sets given as, for example, in [Padawitz, 2001]. Details on algebraic specifications can be found, for example, in [Wirsing, 1990] or in [Ehrich et al., 1989, Ehrig et al., 1999].

With specification $HDOC$, we can abstractly define the hyperdocument given in [Figure 1] [see Specification 2]. It is easy to check that Listing 1 properly implements this specification briefly described in the sequel. For more details on specifying hyperdocuments, we refer to [Doberkat, 1996b, Doberkat, 1998, Fronk, 2000, Fronk, 2001, Fronk, 2002b, Fronk, 2002a].

$\Sigma_{COCKPIT} =$ **import** Σ_{HDOC} **into**

sorts $\mathit{cockpit}$

opns $\mathit{display} : \mathit{cockpit} \to \mathit{MedObj},$
$\mathit{resetbutton} : \mathit{cockpit} \to \mathit{MedObj},$
$\mathit{resettext} : \mathit{cockpit} \to \mathit{MedObj},$
$\mathit{infobutton} : \mathit{cockpit} \to \mathit{MedObj},$
$\mathit{infotext} : \mathit{cockpit} \to \mathit{MedObj},$
$\mathit{yes} : \mathit{cockpit} \to \mathit{MedObj},$
$\mathit{no} : \mathit{cockpit} \to \mathit{MedObj},$
$\mathit{composeDisplay} : \mathit{cockpit} \to \mathit{MedObj},$
$\mathit{linkDisplay} : \mathit{cockpit} \to \mathit{set}(\mathit{link}),$
$\mathit{linkResetText} : \mathit{cockpit} \to \mathit{set}(\mathit{link})$

vars $c : \mathit{cockpit}$

axms

$$\mathit{composeDisplay}(c) = \mathit{compose}(\mathit{resetbutton}(c), \mathit{display}(c), \mathit{infobutton}(c)),$$

$$\mathit{linkDisplay}(c) = \{\mathit{setLink}(\mathit{getBegin}(\mathit{composeDisplay}(c)), \mathit{getBegin}(\mathit{resettext}(c))), \mathit{setLink}(\mathit{getEnd}(\mathit{composeDisplay}(c)), \mathit{getBegin}(\mathit{infotext}(c)))\},$$

$$\mathit{linkResetText}(c) = \mathit{linkAll}(\mathit{getOcc}(\mathit{resettext}(c), \mathit{yes}(c)), \mathit{getBegin}(\mathit{display}(c))) \cup \mathit{linkAll}(\mathit{getOcc}(\mathit{resettext}(c), \mathit{no}(c)), \mathit{getBegin}(\mathit{display}(c)))$$

Specification 2: Specifying a hypermedial car cockpit

Specification $COCKPIT$ imports specification $HDOC$ to achieve access to the elements defined there. We introduce a sort *cockpit* and represent each document declaration as a unary operation from this sort to sort MedObj. Method `composeDisplay` is represented by an operation from sort *cockpit* to sort MedObj, whereas the other methods yield a value in sort $\mathit{set}(\mathit{link})$. We introduce axioms for these methods to give them meaning. Operation *composeDisplay* is equated with a *compose*-term, whereas the other operations are specified as sets of links. Methods `linkDisplay` and `linkResetTest` create them, respectively.

A binding is respected when such a specification is interpreted by appropriate algebras. We follow a loose semantics approach. Let $\mathcal{C}$ be a $\Sigma_{COCKPIT}$-algebra. The sort *MedObj* is interpreted as the set of values given in a binding. We consider [Listing 2], and use its values together with values *Yes* and *No* to interpret the carrier-set C_{MedObj}:

$$C_{MedObj} = \{\, Velocity.dsp, Reset.gif, Info.gif, Reset.txt, Info.txt, Yes, No\}$$

The interpretations of document declarations make use of these values. They are given as follows:

$$\begin{aligned} display^{\mathcal{C}}(c) &= Velocity.dsp, & & \\ resetbutton^{\mathcal{C}}(c) &= Reset.gif & infobutton^{\mathcal{C}}(c) &= Info.gif, \\ resettext^{\mathcal{C}}(c) &= Reset.txt & infotext^{\mathcal{C}}(c) &= Info.txt, \\ yes^{\mathcal{C}}(c) &= Yes, & no^{\mathcal{C}}(c) &= No \end{aligned}$$

The term $compose(resetbutton^{\mathcal{C}}(c), display^{\mathcal{C}}(c), infobutton^{\mathcal{C}}(c))$ interprets operation *composeDisplay*, the link-creating operations as follows:

$$\begin{aligned} & linkDisplay^{\mathcal{C}}(c) = \\ & \quad \{(setLink^{\mathcal{C}}(getBegin^{\mathcal{C}}(composeDisplay^{\mathcal{C}}(c)), getBegin^{\mathcal{C}}(resettext^{\mathcal{C}}(c)))), \\ & \quad\; (setLink^{\mathcal{C}}(getEnd^{\mathcal{C}}(composeDisplay^{\mathcal{C}}(c)), getBegin^{\mathcal{C}}(infotext^{\mathcal{C}}(c))))\}, \\ & linkResetText^{\mathcal{C}}(c) = \\ & \quad linkAll^{\mathcal{C}}(getOcc^{\mathcal{C}}(resettext^{\mathcal{C}}(c), yes^{\mathcal{C}}(c)), getBegin^{\mathcal{C}}(display^{\mathcal{C}}(c))) \\ & \quad \cup \\ & \quad linkAll^{\mathcal{C}}(getOcc^{\mathcal{C}}(resettext^{\mathcal{C}}(c), no^{\mathcal{C}}(c)), getBegin^{\mathcal{C}}(display^{\mathcal{C}}(c))) \end{aligned}$$

Together with $setLink^{\mathcal{C}}(p_1, p_2) = (p_1, p_2)$ and $linkAll^{\mathcal{C}}(pset, p) = \{(p', p) \mid p' \in pset\}$, for all $p, p_1, p_2 \in position$ and for all $pset \in set(position)$, we can fix the carrier-sets for *position* and *link* by evaluating the above given interpretations. For example, the beginning of the compositional is $\epsilon_{Reset.gif}$, or, equally, $1_{composeDisplay}$, since the *compose*-operation describes a tree with a root labeled with $compose(Reset.gif, \ldots)$ together with three sons labeled with *Reset.gif*, *Velocity.dsp*, and *Info.gif*:

$$\begin{aligned} C_{position} = \{\, & \epsilon_{Reset.gif}, 1_{Reset.txt}, \epsilon_{Info.gif}, \\ & 1_{Info.txt}, 13_{Reset.txt}, 14_{Reset.txt}, \epsilon_{Velocity.gif}\}, \\ C_{link} = \{\, & (\epsilon_{Reset.gif}, 1_{Reset.txt}), (\epsilon_{Info.gif}, 1_{Info.txt}), \\ & (13_{Reset.txt}, \epsilon_{Velocity.gif}), (14_{Reset.txt}, \epsilon_{Velocity.gif})\} \end{aligned}$$

It is easy to prove that $\mathcal{C}$ is a model for $\Sigma_{COCKPIT}$. Further, the last mentioned carrier-sets can be model-checked in advance of the hyperdocuments incorporation. For example, dangling links or erroneous connections can be detected, reachability checks can be carried out formally. More details on analyzing hyperlink structures can be found in [Fronk, 2002b]. Following [Definition 6], a hyperdocument is given by the carrier-sets for *MedObj*, *position*, and *link*.

Algebraic specifications can be adopted to work hand in hand with object-oriented implementation. In [Fronk, 2001], flat specifications are implemented by simple classes as shown in this section. Specifications with hidden symbols prepare the use of local classes, whereas hierarchical specifications allow to specify subclass relations; parameterized specifications correspond to generic classes. These details are not vital to understand the development process discussed in the present paper, although they become important when the process is applied to more complex hyperlink structures. This is discussed in the next section.

5 Object-Oriented Design of Hyperdocuments

5.1 Using standard UML diagrams

The object-oriented implementation of a hyperdocument exploits classes, attributes, methods, and relations between classes. A class groups media objects and defines methods to create links between them. Thus, class relations, such as aggregation or inheritance, relate groups of hyperlinked media objects, called *subdocuments*. Apart from its hyperlink structure, a hyperdocument is thereby given an object-oriented structure.

This immediately raises the question how a complex hyperdocument can be broken down into subdocuments which are implemented within a single class. A first approach is not to consider subdocuments at all, but to implement the entire hyperdocument within one class. This kind of monolithic mapping may be enforced by certain hardware restrictions. A second approach, in quite contrast, embeds each media object into a single class. This kind of isomorphic mapping requires extensive use of aggregation, since media objects, then represented by class instances, have to be aggregated within a class such that this class can define links between the aggregated media objects. A third approach is to find reoccurring patterns of hyperlink structures in which similar link-creating methods are applied to different media objects, and to group these media objects within a class. Since *DoDL* uses bindings, we can instantiate this class by means of different bindings yielding similar link structures on different media objects.

Each group of media objects, i.e. each subdocument can be specified by means of a flat specification as shown in the previous section. Relations between those groups are captured by structured specifications. For example, inheritance allows to reuse the methods defined in a subclass and to add new link-creating methods.

Thus, a subdocument may be seen as an extension of another subdocument having similar link structures or defining additional links. Aggregation means that a hyperdocument consists of parts that are themselves hyperdocuments. The aggregating hyperdocument may define hyperlinks between those subdocuments and thus connects them.

Summing up, we identify groups of media objects each of them forms a subdocument. Relating these groups means to set hyperlinks between these documents. If a group is implemented by a *DoDL*-class, we can easily denote the entire link structure by a standard UML class-diagram (c.f. [Fowler and Scott, 1997]). The methods are specified as shown in the previous section and can be implemented using an object-oriented language like *DoDL*. Since the algebraic specifications we use respect object-oriented concepts, the UML class-diagram can be specified respecting class relations. The implementation then works straight forward. The development process is graphically captured in [Figure 2].

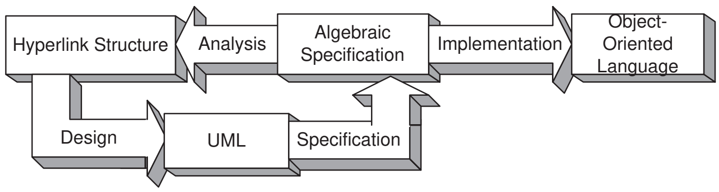

Figure 2: A hyperdocument development process

Other UML diagrams are employed as usual for software development. That is, sequence diagrams, for example, are used to model the interaction between some media objects and the driver, state transition diagram, however, show the individual behavior of media objects within such interactions. In the remainder of this section, we focus on class diagrams.

5.2 A sample hypermedial car cockpit

A car cockpit as presented in [Figure 1] assembles instruments such as a speedometer and information screens. Additionally, we allow for armature to operate, for example, a CD player or a navigation device. Each information whether displayed textually or graphically on the cockpit is represented by a media object. We assume a car to be supplied with a digital screen such as a touch-screen or some other device allowing to interact with the cockpit.

An advantage of such a digital cockpit is its flexibility. The set of media objects displayed may change in correlation with the car's surroundings. For example, when driving on a Highway, the cockpit may provide a cruise control, whereas driving in a city requires more detailed information on the destination. Even checking the car at a garage may require to display technical information on the car's motor or its safety devices the driver is normally not interested in. We call these sets of displayed media objects *screens*. Such a screen the cockpit may present is shown in [Figure 3]. It consists of a speedometer (`Velocity.dsp`), a rotation device (`Rotation.dsp`), as well as a navigation device (`Navigation.dsp`, `Address.dsp`) together with some informative texts.

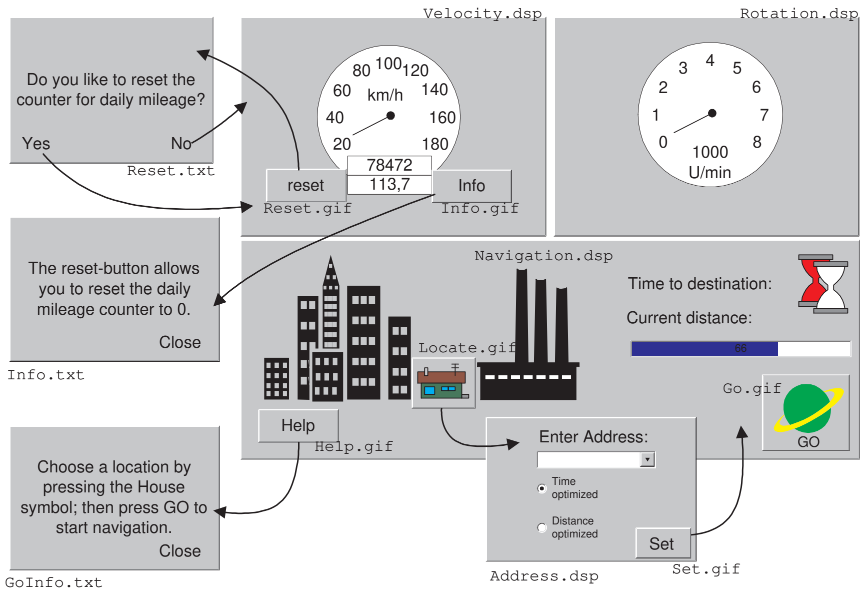

Figure 3: A complex hypermedial car cockpit

We distinguish between two different kinds of hyperlinks: (1) usual links traversed by clicking some button display on a screen, and (2) links changing the screen due to external events. For example, the navigation device may know when the car approaches a city and then automatically switches from a screen showing information relevant for driving on a Highway to a screen displaying information important in cities. Here, sensors are responsible to traverse such a hyperlink. These links may be attributed with values, thereby realizing a browsing behavior. For example, a link changing from a screen *Highway* to a screen *City* may be annotated with an attribute/value pair (*location* : *city*) and is traversed as soon as a sensor sets the attribute *location* to the desired value. Monitoring such links is left to the runtime environment operating the cockpit. Details can be found, for example, in [Doberkat, 1996a] or [Stotts and Furuta, 1989].

These ideas lead to a design process using standard UML briefly discuss in the sequel. A more elaborated version can be found in [Fronk, 2001].

We apply Use Case-diagrams to model the interaction with the cockpit on different screens. [Figure 4] grasps three of them in individual Use Cases each of them can be refined such that the interaction within these screens is precisely captured.

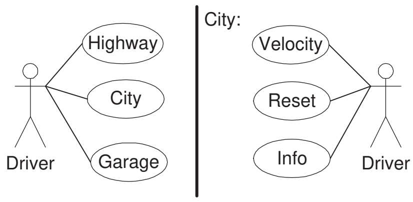

Figure 4: A Use Case-diagram captures the interaction with the cockpit

The connections between those screens are modeled by a first conceptual class-diagram. Each screen is model by a class. [Figure 5] shows how screens are reachable from each other pointed out by UML navigabilities. Navigability means that there exists at least one media object in screen Highway that is hyperlinked to a media object in screen City. In the example, a garage screen is only reachable from a city screen, that is, we do not model a garage aside a Highway.

Figure 5: A conceptual class-diagram captures reachability

The next step is to model the screens in more detail. We refer to [Figure 3] and show its corresponding conceptual class-diagram in [Figure 6]. Each class represents a media object which may be specialized through inheritance. We omit some classes and relations for better readability. Aggregation models composed media objects, whereas navigabilities again denote hyperlinks. Multiplicities indicate how many different instances of concrete media objects can be related to each other. For example, a reset button may invoke many different informative texts, and each of these texts may be invoked by many other reset buttons used in other compositional media objects. Classes in conceptual class-diagrams need not necessarily to correspond to classes in specifying class-diagrams.

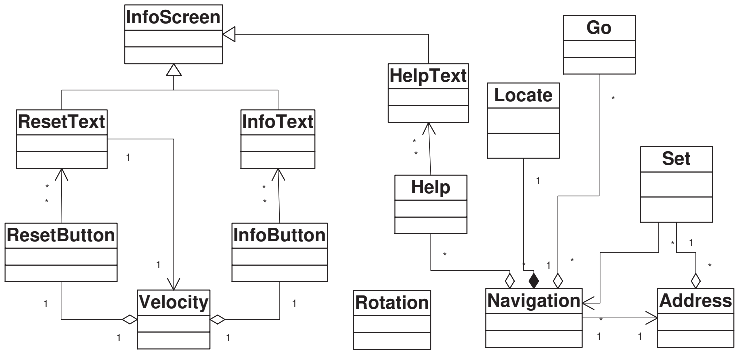

Figure 6: A conceptual class-diagram models a screen

The final step is to model the cockpit through a specifying class-diagram. [Figure 7] shows that a cockpit associates with a presentation. The attributes and methods defined in class `Cockpit` are responsible to make the cockpit work. For example, method `selectPresentation` is responsible to control sensors and to trigger a screen changing event. Screens are special presentations. Again, we omit some classes for better readability. Each screen is equipped with methods to display compositional media objects such as the speedometer, which is declared as an attribute of type `Velocity` in class `CityScreen`. Layout is encoded within methods like `connectInstruments` responsible to both put instruments on the screen and to set links between them if desired. Class `Velocity` represents such a compositional media object. It is a subclass of a generalized display containing methods to access positions, to set links, and to invoke informative texts.

The methods can be specified using structured algebraic specifications like discussed in [Section 4]. On the algebraic level, the impact of concrete media objects on the hyperlink structure can be tested in advance of the hyperdocuments implementation which then equates with usual object-oriented software development shown in [Section 3].

6 Conclusion and Future Work

We proposed a development process for hyperdocuments using standard UML, algebraic specification and object-oriented implementation. The process compares to usual software development. Thereby, we did not extend notations like UML (c.f. [Engels and Sauer, 2002, Garrido et al., 1999]) or State Charts (c.f. [Paulo et al., 1999]) and still consider specific hypermedial aspects. Moreover, software-technical principles and ideas were brought into the hypermedia domain leading to structured media objects and documents in such a way usual

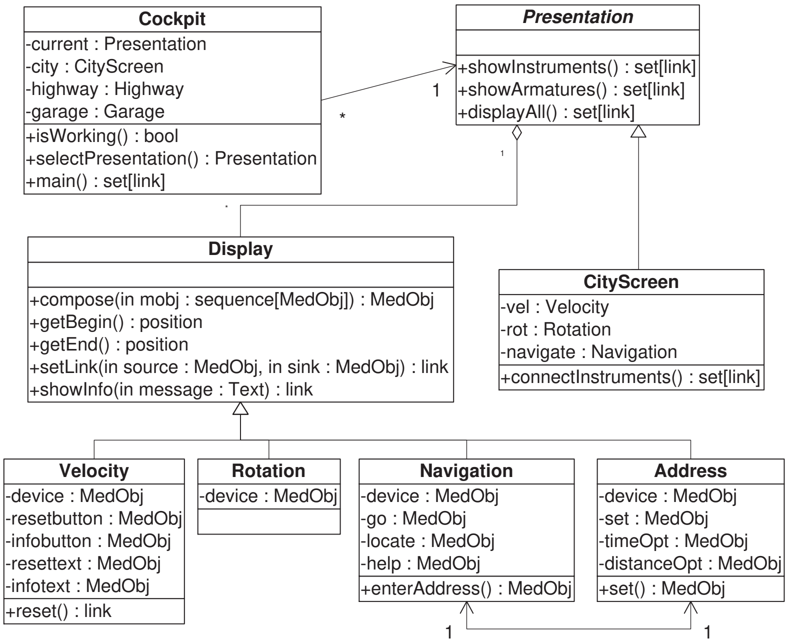

Figure 7: A specifying class-diagram models the entire cockpit

software development can carry over. That is, we rather adapt hypermedia development to software development than vice versa.

In [Fronk, 2001], both *DoDL* and hyperdocuments were subject to formal investigations concerning the language's denotational semantics given by algebras, as well as the hyperlink structure, opening properties of structured media objects and hyperlinks to verification on a formal, algebraic level. More details on this analysis phase, which we refer to as the test phase of a hyperlink structure, can be found in [Fronk, 2002b]. The specification of hyperdocuments proposed in [Section 4] develops its full power when the number of hyperlinks exceeds a level that makes it cumbersome to analyze hyperlinks by walking through the entire hyperdocument manually. Hence, formally testing link structures saves time, is less error-prone, helps to ensure certain properties, and, last not least, allows to mathematically prove structural and hyperlink properties. This is important, for example, in such environments where operating a hyperdocument relies on reasoning about its security.

Media objects and their behavior as well as browsing are currently under investigation following the same ideas as sketched in the present paper. We are developing the car cockpit mentioned in [Section 5]. Apart from implementing suitable interfaces to media objects allowing for accessing positions, layout as well as communication between media objects are considered in greater detail. We focus on "active links" which allow to exchange data between media objects, and to trigger actions. This adds behavior to links which then can become active when traversed, realizing a browsing behavior given by the link's individual implementation. Techniques to implement complex graphical user interfaces (GUI's) and hypermedial issues are brought together. This is feasible due to comparing the development of hyperdocuments with usual software development. A hyperdocument can in this approach be understood as a GUI containing hyperlinked elements. As much as the implementation of a GUI's functionality is separated from its layout, we separate both the hyperdocuments structural and navigational aspects from the presentation of its elements.

Acknowledgements

We gratefully thank Prof. Dr. E. E. Doberkat, who supervised the author's PhD thesis, for his important and constructive remarks on the thesis. The discussions with Dr. Jörg Westbomke helped to clarify some interesting points concerning the present paper.

References

[Chang and Shih, 2002] Chang, S. K. and Shih, T. K. (2002). Multimedia software engineering. In Chang, S. K., editor, *Handbook of Software Engineering & Knowledge Engineering*, volume 2, Emerging Technologies, pages 1 – 20. World Scientific.

[Costagliola et al., 2002] Costagliola, G., Ferrucci, F., and Francese, R. (2002). Web engineering: Models and methodologies for the design of hypermedia applications. In Chang, S. K., editor, *Handbook of Software Engineering & Knowledge Engineering*, volume 2, Emerging Technologies, pages 181 – 199. World Scientific.

[Deitel et al., 2001] Deitel, H. M., Deitel, P. J., Nieto, T. R., Lin, T. M., and Sadhu, P. (2001). *XML How to Program*. Prentice Hall.

[Doberkat, 1996a] Doberkat, E.-E. (1996a). Browsing a hyperdocument. Memorandum 87, Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie.

[Doberkat, 1996b] Doberkat, E.-E. (1996b). A language for specifying hyperdocuments. *Software - Concepts and Tools*, 17:163–172.

[Doberkat, 1998] Doberkat, E.-E. (1998). Using logic for the specification of hypermedia documents. In Balderjahn, J., Mathar, R., and Schader, M., editors, *Classification, Data Analysis and Data Highways*, pages 205–212. Springer.

[Ehrich et al., 1989] Ehrich, H.-D., Gogolla, M., and Lipeck, U. W. (1989). *Algebraische Spezifikation abstrakter Datentypen*. Teubner.

[Ehrig et al., 1999] Ehrig, H., Mahr, B., Cornelius, F., Grosse-Rhode, M., and Zeitz, P. (1999). *Mathematisch-strukturelle Grundlagen der Informatik*. Springer.

[Engels and Sauer, 2002] Engels, G. and Sauer, S. (2002). Object-oriented modeling of multimedia applications. In Chang, S. K., editor, *Handbook of Software Engineering & Knowledge Engineering*, volume 2, Emerging Technologies, pages 21 – 52. World Scientific.

[Fowler and Scott, 1997] Fowler, M. and Scott, K. (1997). *UML distilled: applying the standard object modeling notation*. Addison-Wesley.

[Fronk, 1999] Fronk, A. (1999). Support for hypertext maintenance. IEEE Computer. Letter to the Editor.

[Fronk, 2000] Fronk, A. (2000). A tool system for an object-oriented approach to construction and maintenance of hypermedia documents. In Gaul, W. and Ritter, G., editors, *Classification, Automation and New Media*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 265 – 272. Springer, 2002.

[Fronk, 2001] Fronk, A. (2001). *Algebraische Semantik einer objektorientierten Sprache zur Spezifikation von Hyperdokumenten.* PhD thesis, Lehrstuhl Software-Technologie, Fachbereich Informatik, Universität Dortmund, Shaker Verlag, 2002.

[Fronk, 2002a] Fronk, A. (2002a). An approach to algebraic semantics of object-oriented languages. Memorandum 128, Lehrstuhl Software-Technologie, Fachbereich Informatik, Universität Dortmund.

[Fronk, 2002b] Fronk, A. (2002b). Towards the algebraic analysis of hyperlink structures. Memorandum 126, Lehrstuhl Software-Technologie, Fachbereich Informatik, Universität Dortmund.

[Fronk and Pleumann, 1999] Fronk, A. and Pleumann, J. (1999). Der *DoDL*-Compiler. Memorandum 100, Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie. ISSN 0933-7725.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: elements of reusable object-oriented software.* Addison-Wesley.

[Garrido et al., 1999] Garrido, A., Helmerich, A., Koch, N., Mandel, L., Rossi, G., Olsina, L., and Wirsing, M. (1999). Hyper-UML: Specification and modeling of multimedia and hypermedia applications in distributed systems. In *2. Workshop on the German-Argentinian Bilateral Program for Scientific and Technological Cooperation, Königswinter, Germany.*

[Garzotto and Paolini, 1993] Garzotto, F. and Paolini, P. (1993). HDM – a model-based approach to hypertext application design. *ACM Transactions on Information Systems*, 11(1):1 – 26.

[Ghezzi et al., 1991] Ghezzi, C., Jazayeri, M., and Mandrioli, D. (1991). *Fundamentals of Software-Engineering.* Prentice Hall.

[Gloor, 1997] Gloor, P. (1997). *Elements of hypermedia design: techniques for navigation & visualization in cyberspace.* Birkhäuser.

[Halasz and Schwartz, 1994] Halasz, F. and Schwartz, M. (1994). The dexter hypertext reference model. *Communications of the ACM*, 37(2):30–39.

[Hardman et al., 1994] Hardman, L., Bulterman, D. C., and van Rossum, G. (1994). The amsterdam hypermedia model: Adding time and context to the dexter model. *Communications of the ACM*, 37(2):50 – 62.

[Isakowitz et al., 1995] Isakowitz, T., Sthor, E., and Balasubramanian, P. (1995). RMM: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34 – 44.

[Lowe and Hall, 1999] Lowe, D. and Hall, W. (1999). *Hypermedia & the Web - an engineering approach.* Wiley & Sons.

[Padawitz, 2001] Padawitz, P. (2001). Sample swinging types. `http://ls5.cs.uni-dortmund.de/~peter`. Manuskript.

[Paulo et al., 1999] Paulo, F. B., Masiero, P. C., and de Oliveira, M. C. F. (1999). Hypercharts: Extended statecharts to support hypermedia specification. *IEEE Transactions on Software Engineering*, 25(1):33–49.

[Pleumann, 2000] Pleumann, J. (2000). dodl2html - Ein Generator zum Erzeugen von graphspezifizierten Hyperdokumenten. Master's thesis, Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie.

[Schwabe and Rossi, 1998] Schwabe, D. and Rossi, G. (1998). *An object-oriented approach to web-based application design.* Wiley & Sons.

[Stotts and Furuta, 1989] Stotts, P. D. and Furuta, R. (1989). Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems*, 7(1):3 – 29.

[Stotts et al., 1998] Stotts, P. D., Furuta, R., and Cabarrus, C. R. (1998). Hyperdocuments as automata: Verification of trace-based browsing properties by model checking. *ACM Transactions on Information Systems*, 16(1):1–30.

[Tochtermann, 1994] Tochtermann, K. (1994). *Ein Modell für Hypermedia*. PhD thesis, Universität Dortmund, Fachbereich Informatik, Lehrstuhl 1.

[Wirsing, 1990] Wirsing, M. (1990). Algebraic specifications. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, pages 675 – 788. Elsevier.