UML+OCL Specification of the Dexter Hypertext Reference Model*

Nora Koch[1]

Ludwig-Maximilians-Universität München
Institut für Informatik
Oettingenstr. 67
D-80538 München
Tel. +49 89 2178 2151
Fax +49 89 2178 2152
kochn@informatik.uni-muenchen.de

1 Introduction

The Dexter Hypertext Reference Model was the result of the discussions of a small workshop on hypertext at the Dexter Inn, Sunapee, New Hampshire in October 1988. The purpose was to find a common language for the people involved in hypermedia development and to obtain common abstractions to the hypermedia systems existing at that time (Halasz and Schwartz, 1994). It has been proven to be so useful and stable, that it has been used since then as basis for discussions and to build hypermedia systems. No doubt that the Dexter Reference Model is one of most the important milestones in the hypermedia development history.

The Dexter Model was formalised by Halasz and Schwartz (1990) in the specification language Z (Spivey, 1992), a specification language based on the set theory. Since then, object-oriented models and programming have increased their importance and dissemination. In addition, more emphasis is put in visual modeling languages, that improve intuitive comprehension of models.

The specification developed in this work is an object-oriented specification approach based on the Unified Modeling Language (UML) and the Object Constraints Language (OCL). UML (Booch, Rumbaugh & Jacobson, 1999) provides the notation and the object-oriented modeling techniques for the visual representation of the model. OCL (Warmer & Kleppe, 1999) is used for the formal specification of invariants on the model elements and attributes as well as of pre-conditions and post-conditions on the functions of the Dexter Reference Model.

---

* Technical Report 0008, Ludwig-Maximilians-Universität München, Dezember 2000.

[1] also working at F.A.S.T. Applied Software Technology GmbH, Arabellastr. 17, D-81925 München, Germany, koch@fast.de.

The visual representation has the advantage that it shows at one glance the relevant concepts, how they are organised and how these concepts are related. UML has been chosen as it has become a standard modeling language. The semi-formal graphical representation is complemented with semantic information formally written in OCL. The use of OCL improves the model precision as it is stresses by Richter and Gogolla (1999) compared to constraints described as English text. But in this work it also allows for an object oriented formal specification that is equivalent to the original Z specification of Dexter Model.

This work is structured as follows. The second section gives a brief textual description of the Dexter Hypertext Reference Model. The third section details the object-oriented specification of the reference model. It is based on UML class diagrams of the layers of the Dexter model and the OCL descriptions of invariants, pre- and post-conditions for classes, attributes and operations. Finally, some conclusions and further work is outlined. The Dexter Model uses the word "hypertext" to refer to both text-only and multimedia systems; so is done in this work.

2 The Dexter Hypertext Reference Model

The Dexter Reference Model divides a hypertext system into three layers. They are the *run-time* layer, the *storage* layer and the *within-component* layer connected by the interfaces *presentation specification* and *anchoring*. The model focuses mainly on:

- the storage layer,
- the mechanisms of anchoring (interface between the storage layer and within-component layer),
- the presentation specification (interface between the storage layer and run-time layer), and
- some aspects of the run-time layer.

The within-component layer is purposely not elaborated within this reference model.

Figure 1 shows these layers as presented in the work of Halasz & Schwarz (1994).

The main goal of the reference model is to describe the network of nodes and links of the storage layer, i.e. the mechanisms by which these links and nodes are related. In this layer the nodes are treated as general data containers. The content and structure within the hypertext nodes are described in the within-component layer. The run-time layer contains the description of how nodes and links are presented, how interaction can be performed, i.e. the description of the dynamics of the application. But the Dexter Model only provides the realisation of a set of interfaces, it does not attempt to cover all the details of the user interaction with the hypertext.

For the general containers of data of the within-component layer, no details are given about their content, such as text, graphics, animation, etc. as well about the structure and the mechanism to deal with this structure.

In addition, the model describes the interfaces between the run-time layer and the storage layer (presentation specification) and between the storage layer and the within-

component layer (anchoring). It can be observed that this separation of the contents, structure and presentation aspects of hypermedia systems is the basis of most of the hypermedia design methods.

The Dexter Model describes the storage layer as the structure of a hypertext system that consists of a finite set of *components*. A component is either an *atom*, a *link* or a *composite* entity. Atoms in the Dexter Model terminology are the "*nodes*" of the hypertext system. Links, also called link components, are entities that represent relations between components. Each component includes a component information and a content specification. The component information consists of a list of attributes, a presentation specification and a list of anchors.

- With *attributes* arbitrary properties can be included, as for example to attach keywords to a component.
- The *list of anchors* provides a mechanism to specify the end points of the links that relate this node with other nodes of the network.
- The *presentation specification* is used as interface to the run-time layer.
- The *content specification* is used as interface to the within-component layer.

Every component has associated a unique identifier (UID). These UIDs are assumed to be unique in the whole universe of discourse.

The content of a *link component* is a list of two or more specifiers. Each specifier contains a *component specification*, a *presentation specification*, *an anchor identification* (id) and a *direction*. Direction can either have the value "from", "to", "bidirect" or "none" with the following semantic: source of a link, destination, both or neither source nor destination.

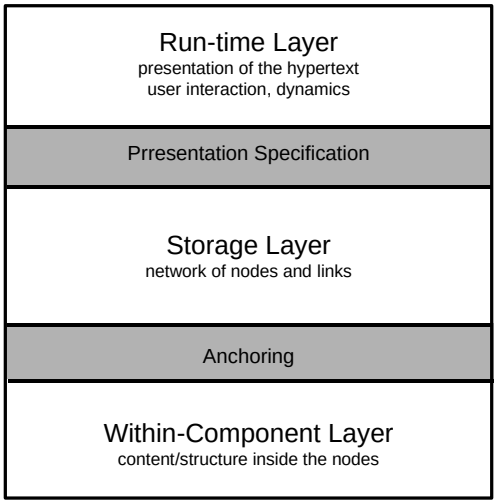

Figure 1: Layers of the Dexter Hypertext Reference Model

Anchoring is the mechanism that provides the functionality to allow for linking between nodes or documents but also for addressing (referring) to locations within the content of a component. An *anchor* is an indirect addressing entity, which has two parts: *anchor id* and *anchor value*. The *anchor value* is an arbitrary value specifying a location, an item or a region. This anchor value only is a variable and interpretable field within the content of the component. It is part of the within-component layer. Otherwise, the *anchor id* remains constant and identifies its anchor uniquely within the scope of its component or uniquely across the whole universe through a pair "UID-anchor id".

The functionality of the storage layer is supported by a *resolver* function and an *accessor* function. Together they are responsible for mapping specifications of components into the components themselves, i.e. retrieving the components. The resolver function "resolves" the component specification into a component UID or set of UIDs, which is used by the accesor function to "access" the correct component(s). The accessor function may find out that no component exits for a UID. We are in presence of a *dangling* link.

In addition to the data model, the Dexter Model defines a set of operations to access or modify the hypertext structure. These operations are: create an atom, a link or a composite component, delete or modify components, set values of attributes and get a component (using the accessor function) as well an operation to get all attributes of a component. Two other operations help to determine the accessibility of the network. They are the *linkTo* and *linkToAnchor* operations. The first one, given a component and an anchor contained in the component, it returns the set of links that resolve to this anchor. The second one, given a hypertext and a component UID, it returns all links resolving to that component.

The Dexter Model requires link consistency. Therefore, when a component is deleted, the system has to guarantee that also all links resolving to that component are deleted. This requirement has been widely criticised.

The run-time layer describes how the components are presented to the user. The presentation is based on the concept of instantiation of a component, i.e. a copy of the component is cached to the user. If the user modifies the instantiation, it is written back into the storage layer. The copy receives an instantiation identifier (IID). To note is, that simultaneously there may exits more than one instantiation for a component and that a user may be viewing more than one component. In order to keep track of all these instantiations the run-time layer uses an entity *session.* The user will open a session by the action *present Component* of a given hypertext, she can edit the instantiation, save the modifications, create a new component or delete a component. The most common action is *follow Link*, which takes the IID of an instantiation together with the link marker contained within that instantiation and presents then to the user any component resolved according the content of a link component specifier, i.e. components that are the end point destination of links. The user is allowed to remove an instantiation and close the session.

3 UML+OCL Specification of the Dexter Model

The run-time layer, storage layer and the within-component layer, into which the Dexter Model divides a hypertext system, are represented in this work as packages in a UML class diagram. This diagram is shown in Figure 2.

The following sections present the UML class diagrams and the OCL specification of the storage layer, run-time layer and the description of the functionality of these layers. The description of the within-component layer is not within the scope of Dexter hypertext reference model.

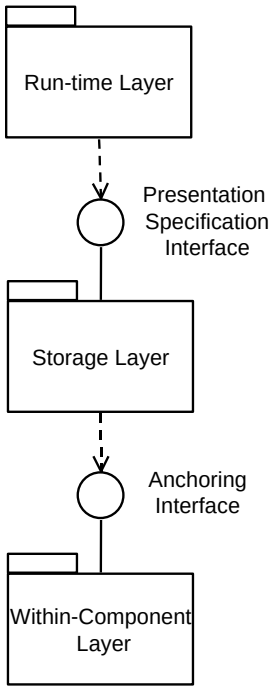

Figure 2: UML-Model for the Layers of the Dexter Hypertext Reference Model

3.1 The Storage Layer

The storage layer describes the structure of a hypertext *(class Hypertext)* as a finite set of components *(class Component)* together with two functions, a *resolver* and an *accessor* function. With these two functions it is possible to "retrieve" components. Every component has a globally unique identity *(class UID)*, that is assumed to be unique in the entire hypertext universe. The accessor function allows for the "access" to a component given its UID. UIDs provide a guaranteed mechanisms for addressing any component in a hypertext.

In the Dexter model this addressing is accomplished in a indirect way based on the entities called anchor *(class Anchor)* consisting of two parts: an anchor ID and anchor value *(classes AnchorID and AnchorValue).* The anchor value is an arbitrary value that specifies some location within a component. The anchor ID is an identifier that uniquely

identifies the anchor within the scope of the component. Together with the UID it permits to uniquely identify the anchor within the scope of the hypertext.

A component is composed by two parts: a component base *(class ComponentBase)* and a component information *(class ComponentInfo).* The component information consists of attributes *(class Attribute),* a presentation specification *(class PresentSpec)* and a sequence of anchors *(class Anchor).* The component base can be either an atom *(class Atom),* a link *(class Link)* or a composite of other components *(class Composite).* A link is a set of two or more specifiers *(class Specifier).* Specifiers are a composed by a component specification, an anchor specification and a presentation specification.

Figure 3 shows the Storage Layer represented by a UML class diagram. All the classes depicted are part of the package "Storage Layer" with exception of *Content* and *Anchor Value* that are classes of the package "Within-Component Layer".

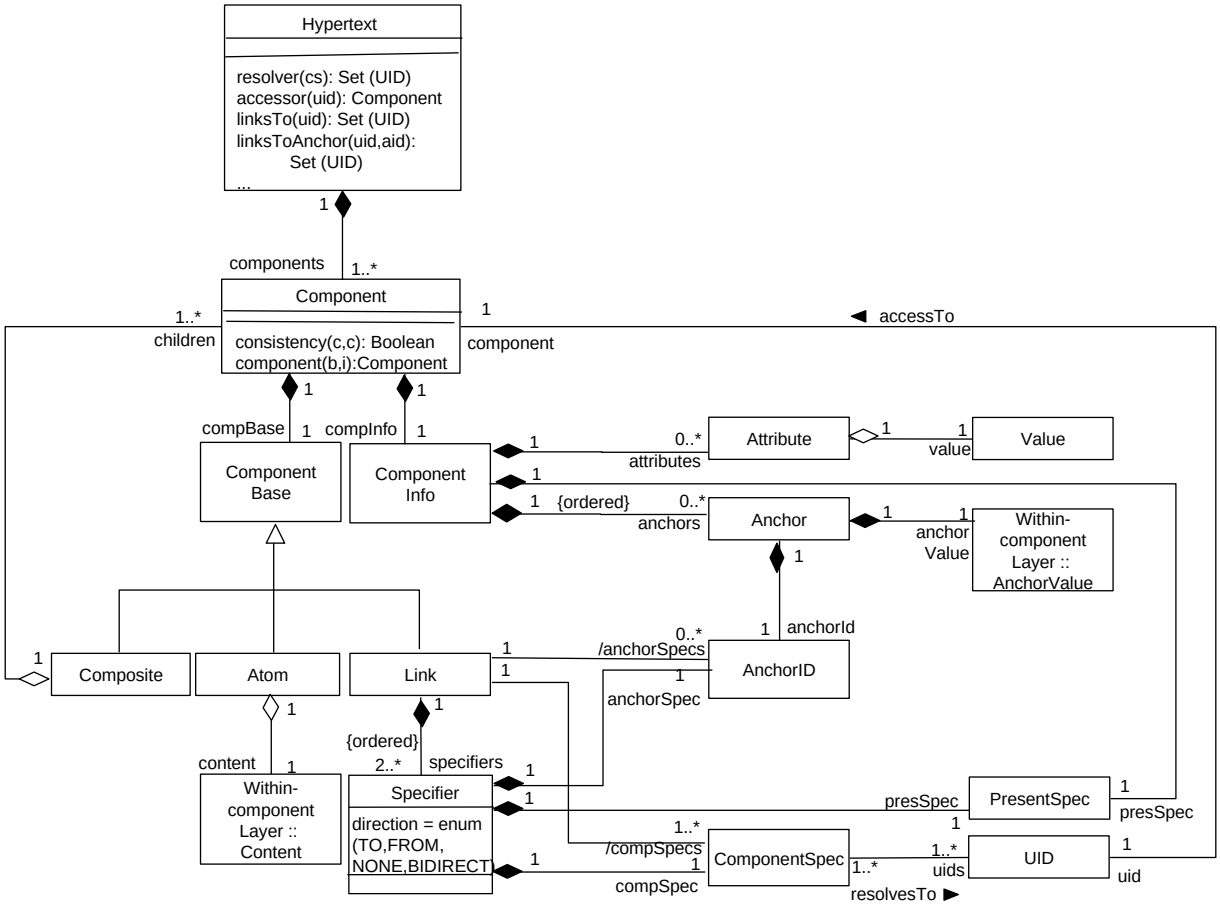

Figure 3: UML Class Diagram for the Storage Layer

Component

A component has associated a base component (*ComponentBase*) and a component information *(ComponentInfo).* It is represented with an abstract class *Component.*

The base component describes the content of the component. It can be either an atom *(Atom)*, a link *(Link)* or a composite *(Composite)*, as it is shown in the UML class diagram (see Figure 3).

The model defines a "type consistency" relationship between components: two components are "type consistent", if they are both atoms or both links or both composites. This relationship is specified by the following constraint.

context Component :: consistency (c1:Component, c2: Component): Boolean
pre: - - none
post: result = c1.oclIs TypeOf (Atom) and c2.oclIs TypeOf (Atom) or c1.oclIs TypeOf (Link) and c2.oclIs TypeOf (Link)
or c1.oclIs TypeOf (Composite) and c2.oclIs TypeOf (Composite)

The following function builds a component given its component base and associated information.

context Component :: component (base:ComponentBase, info: ComponentInfo): Component
pre: - - none
post: let c = self.oclIsNew
in c.compBase = base
and c.compInfo = info
and result = c

The component information instead describes the properties of the component, that are different to the content of the component. These properties are a sequence of anchors *(Anchor)*, a presentation specification *(PresentSpec)* and optionally a set of arbitrary attribute/value pairs *(Attribute and Value)*. The last one can be used to define any arbitrary property for a component and assign a value to it. The presentation specification contains information specifying how this component should be presented at run-time. It is part of the interface between the storage layer and the run-time layer. Anchors are part of the interface between the storage layer and the within-component layer.

Note that a presentation specification always has some value. Therefore, a component information (*ComponentInfo*) is initialised with no attributes, no anchors and a presentation specification which is given as argument. The post-condition of the operation *init* indicates that a component instance has to fulfil these constraints.

context ComponentInfo :: init (ps:PresentSpec)
pre: - - none
post: attributes → isEmpty
and anchors → isEmpty
and presSpec = ps

Anchor

Anchoring is the mechanism that provides the functionality to allow for linking between nodes or documents but also for addressing (referring) to locations within the content of a component.

An anchor is defined as a pair of an anchor ID *(AnchorID)* and an anchor value *(AnchorValue)*. The anchor ID is an identifier which uniquely identifies its anchor within the scope of its component. Through the pair component UID - anchor ID an anchor can therefore be uniquely identified across the whole universe. The anchor value is an arbitrary value that indicates some location, item or substructure within the component. The anchoring process is made possible by this decomposition of the anchor in two parts: the anchor ID is used by the storage layer while the anchor value is a variable field for use by the within-component layer.

Thus, to ensure that the anchor identifiers are unique within a component the following invariant constraint must be fulfilled: The number of anchors must be equal to the number of different anchor identifiers.

context ComponentInfo
inv number of anchors**:**
anchors → size = anchors.anchorID → asSet → size

Atom

An atom has a content which represents the data of the component. The content of an object is a primitive of the model. It is concern of the within-component layer, therefore no details are described in the storage layer. The operation *init* connotes that a atom instance has no content after initialisation.

context Atom :: init ()
pre: - - none
post: content → isEmpty

Specifier

Another type of component is a link. It consists of a sequence of at least two specifiers. A specifier defines one single end point of a link. A specifier consists of a component specification (*ComponentSpec*) and an anchor identification (*AnchorID*) as well as two additional fields: a presentation specification and a direction.

The component specification together with the anchor identification specifies a component and an anchor within the component. The use of the component specification instead of the UID has the advantage that it allows for indirect addressing, i.e. the UID of destination is resolved at run-time.

The direction encodes whether the end point is the source of the link (FROM), the destination (TO), both a source and a destination (BIDIRECT), or neither a source nor a destination (NONE). The direction of a specifier instance is initialised with NONE.

context Specifier :: init ()
pre: -- none
post: direction = #NONE

The presentation specification (*PresentSpec*) is a primitive value that forms part of the interface between the storage layer and the run-time layer.

Link

As already defined, a link consists of a sequence of at least two specifiers. Thus, the Dexter Model excludes dangling links, but allows for links with arity greater than two. Binary links are the standard in hypertext systems.

All links should have at least one destination. The following invariant ensures this as it requires at least one specifier with value TO for the direction.

context Link
inv at least one specifier with direction TO**:**
specifiers**.**direction → exists (s: Specifier | s**.**direction = #TO)

Links are "first class citizen" as they inherit from component, that implies that links to a link component may be defined in the same way as to an atom or composite component.

Link includes two derived associations (*compSpecs* and *anchorSpecs*) establishing a direct association to ComponentSpec and to AnchorID. These associations are annotated with a "/". The association */compSpec* results in the set of component specifications for a link and */anchorSpec* in the set of anchors IDs for the link.

context Link
inv derived association /compSpecs**:**
/compoSpecs = specifiers**.**compSpec → asSet

context Link
inv derived association /anchorSpecs**:**
/anchorSpecs = specifiers**.**anchorSpec → asSet

Composite

A composite component is constructed recursively out of other components. It is restricted to be a directed acyclic graph, i.e. a component may be sub-component of more than one composite and no composite may directly or indirectly contain itself as a sub-component. Here the composite pattern (Gamma, Helm, Johnson & Vlissides, 1995) is used for modeling a component structure.

The "no existence of children" is a constraint that has to be fulfilled by a new composite instance.

context Composite :: init ()
pre: - - none

post: children → isEmpty

Hypertext

A hypertext system represented by the class model (Figure 3) consists of three parts:

- a set of components that represent "nodes" and "links",
- a function called "resolver" that returns the UID for a given component specifier (more than one specifier may return the same UID), and
- an "accessor" function which given a UID returns a component.

The resolver function is responsible for "resolving" a component specification into a UID. The UIDs are primitives in the model with attribute ID. The accessor function is responsible to access the component corresponding to the resolved UID. The resolver is a partial function; the accessor a total and invertible function.

```
context Hypertext :: resolver ( cs : ComponentSpec ) : Set (UID)
pre: components → exists ( c: Component |
          c.compBase.oclIsTypeOf (Link)
          and c.compBase./compSpecs → includes (cs)
         )
post: result = UID.allInstances → select ( u: UID | cs.uid → includes (u) )

context Hypertext :: accessor ( uid : UID ) : Set (Component)
pre: components → exists ( c: Component |
          c.compBase.oclIsTypeOf (Link)
          and c.compBase./compSpecs.uids → includes (uid)
         )
post: result = uid.component
```

To identify the set of links resolving to a component, the Dexter Reference Model introduces the function linksTo which, given a hypertext system and the UID of a component in the system, returns the UIDs of all links resolving to that component.

```
context Hypertext :: linksTo ( uid : UID ) : Set (UID)
pre: self.components → exists ( c : Component | accessor (uid) = c )
post: result = UID.allInstances → select ( lid : UID |
           Component.allInstances → exists (link : ComponentBase |
           link.oclIsTypeOf (Link)
          and link = accessor (lid).compBase
            and ComponentSpec.allInstances → exists ( cs : ComponentSpecs
              | link./compSpec → includes (cs)
                 and uid = resolver (cs)
          ) ) )
```

There are four constraints which must be satisfied by every instance of the class *Hypertext* (invariants):

- The *accessor* function must yield a value for every component. As this function is invertible, every component must then have a UID.
- The *resolver* function must be able to produce all possible valid UIDs, i.e. the range of the *resolver* has to be equal to the domain of the *accessor*. Thus, dangling links are not allowed in the model.
- The anchor ID of a component must be the same as the anchor IDs of the component specifiers of the links resolving to the component.
- There are no cycles in the component/sub-component relationship, that is no component may be a sub-component (directly or transitively) of itself.

The first constraint is the "components accessibility" and ensures that all hypertext components are accessible by means of the accessor function. It can be formalised as follows:

```
context Hypertext
inv  components accessibility:
        components → forAll ( c:Component |
        UID.allInstances → exists (uid:UID | c = accessor (uid)
    ) )
```

The second constraint states that the set of UIDs obtained "resolving" component specifications (resolver range) is equal to the set of valid documents that can be retrieved by the accessor (accessor domain).

```
context Hypertext
inv  range of resolver = domain of accessor:
    ComponentSpec.allInstances → forAll (cs: ComponentSpec |
       UID.allInstances → exists (uid:UID |
         resolver (cs) → includes (uid)
         and Component.allInstances → exists  (c:Component |
              accessor (uid) = c
              and  components → includes (c)
         ) ) )    and
     UID.allInstances → forAll (uid: UID |
        Component.allInstances → exists (c:Component |
           accessor (uid) = c
           and  ComponentSpec.allInstances → exists  (cs:ComponentSpec |
                   resolver (cs)→ includes (uid)
                 and  componentsSpec→ includes (cs)
         ) ) )
```

The third constraint can be described in OCL using the previously defined operation *linkTo*. This constraint assures that the set of anchors identifiers of a component should always be equal to the set of anchors identifiers of the links resolving to that component.

context Hypertext
inv anchors IDs of a component = anchors IDs of the links resolving to the component**:**

components → forAll (c : Component |
UID → exists (uid:UID |
c = accessor (uid)
and AnchorID → exists (aid : AnchorID |
c.compInfo.anchors.anchorID → includes (aid) implies
Component → exists (link:Link | UID → exists (lid: UID |
linksTo (uid) → includes (lid)
and link./anchorSpecs → includes (aid)
)))))

The fourth constraint guarantees that a component is not included in the transitive closure of sub-components of this components.

It has to be proved that the transitive closure of the relation children does not contain a pair with two equal elements. To calculate the transitive closure, first the association *children* is transformed into an association class as depicted in Figure 4.

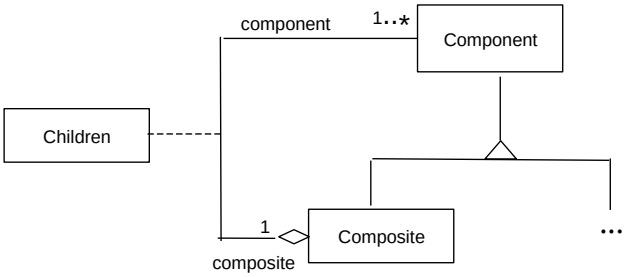

Figure 4: The children association class

The OCL constraint that we are looking for is the following, where *transClos* is the transitive closure of the pairs of composites related by a *children* relationship:

not transClos → exists (ch: Children | ch.component = ch.composite)

Unfortunately, OCL collections of collections are flattened, i.e. we define the *transClos* as a sequence as proposed by Mandel and Cengarle (1999), of an even number of elements, where even positions belongs to components and odd positions to composites. The expression written above can be replaced by:

not transClos → exists (i : Integer | transClos → at (i*2-1) =
transClos → at (i*2))

The transitive closure can be calculated in two steps. First an operation called *subcomponents* is defined that builds a sequence of pairs of components (sub_comp) including all components that have children of type composite.

```
context Hypertext:: subcomponents(): Sequence (Composite)
pre:  --none
post:  Children.allInstances → iterate ( pair: Children;
          sub_comp : Sequence (Composite) =  Sequence{} |
            if  pair.component.oclTypeOf  (Composite)
            then sub_comp →  append (pair.composite)
                                → append (pair.component)
            else  sub_comp
            endif
)
```

In the second step an operation *transitiveClosure* is defined. It applies the Warschall's algorithm (Lang, 1988) to a given sequence of composites (pair of related composites) to calculate the transitive closure (*transClos)* . The result is a sequence of all pair of composites included in the transitive closure of the initial sequence.

```
context Hypertext :: transitiveClosure(initial:Sequence (Composite)):
Sequence (Composite)
pre:  --none
post: Composite.allInstances → iterate ( c3 : Composite;
          aux3 : Sequence (Composite) =  initial |
          Composite.allInstances → iterate ( c2 : Composite;
            aux2 : Sequence (Composite) =  aux3 |
                  Composite.allInstances → iterate ( c1 : Composite;
                  aux1 : Sequence (Composite) =  aux2 |
                           if  Sequence {1..(aux1 → size) / 2}→ exists ( i,j : Integer |
                          aux1 →  at (2*i-1) = c1   and   aux1 →  at (2*i) = c3
                          and  aux1 →  at (2*j-1) = c3   and   aux1 →  at (2*j) = c2
                           then  aux1 →  append (c1) →  append (c2)  else  aux1
                          endif
          ) ) )
```

The fourth invariant is obtained using the above defined operation *subcomponents* and *transitiveClosure*. Thus, the constraint specifying that a composite may not contain itself as a sub-component can be formalised as follows:

```
context Hypertext
inv notItselfAsSubcomponent:
          let  transClos : Sequence (Composite)  =
                    transitiveClosure (self.subcomponents())
          in  not  transClos  →  exists ( i : Integer | transClos →  at (i*2-1)
= transClos →  at (i*2) )
```

3.2 The Storage Layer Functions

The *Hypertext* class includes two operations to access to links and anchors, i.e. ensuring the navigability functionality of the hypermedia system. The first one determines the set of links that resolve to a specific component (*linksTo*). The second one obtains the set of links that resolve to a specific anchor (*linksToAnchor*). Given a hypertext system and the UID of a component in the system, the function *linksTo* (defined above) returns the UIDs of all links resolving to that component.

The operation *linksToAnchor* returns the link components that are associated to a particular anchor of a component. The following is the OCL expression for *linksToAnchor*.

```
context Hypertext :: linksToAnchor (uid:UID, aid:AnchorID) : Set (UID)
pre:   -- none
post:  result = linksTo (uid) → select (lid: UID |
                    accessor (lid).compBase.oclIsTypeOf (Link)
                    and  accessor (lid).compBase./anchorSpecs → includes (aid)
                    )
```

Operations are provided to update a hypertext: *createComponent, modifyComponent* and *deleteComponent*. Operations that modify nodes and links of the hypertext must assure "link consistency", i.e. that is all the component specifiers resolve to existing components. It is proven with the following invariant.

```
context Hypertext
inv    linkConsistency:
       components.compBase.allInstances → forAll ( cb : ComponentBase |
            cb.oclIsTypeOf (Link)   implies
            Components.allInstances → exists ( c : Component |
                 accessor(resolver(cb./compSpec)) = c
        ) )
```

In addition to the functions to manipulate anchors and links, the Dexter Reference Model defines functions for the creation, modification, removing and retrieval of a component as well as for the manipulation of attributes. Creation is supported by a set of operations described below, modification *by modifyComponent*, removal by *deleteComponent*, retrieval by *getComponent* and manipulation of attributes by *attributeValue, setAttributeValue* and *allAttributes*.

Creating a New Component

The operation *createNewComponent* is the function invoked from the run-time layer to incorporate a new component to the hypertext. It calls one of the following operations: *createAtomicComponent, createLinkComponent* or *createCompositeComponent*. These three operations make use of are the operation *addComponent* (createComponent is the original name given by the authors of the Dexter Model to this operation).

The operation *addComponent* adds a new component to the hypertext. It ensures that the range of the accessor function is extended to include the new component. The resolver function is also extended so that there is at least one component specifier for this new component that resolves to this unique identifier.

The constraints related to this operation are:

```
context  Hypertext :: addComponent (c : Component)
pre:    -- none
post:  components   =  components @pre → including (c)
          and UID.allInstances  →  exists ( uid:UID  |  accessor (uid) = c
           and  ComponentSpec.allInstances  →  exists (cs:ComponentSpec  |
           resolver (cs) = uid  →  includes  (uid)
           ) )
```

createAtomicComponent takes an atom and a presentation specification and uses *addComponent* to create a new atomic component.

```
context  Hypertext :: createAtomicComponent (a: Atom, ps: PresentSpec) :
Component
pre:    -- none
post:  Component. allInstances →  exists ( c: Component  |
                    c.oclIsNew
                    and  c.compBase =  a
                    and  c.compInfo.presSpec  =  ps
                    and  self.addComponent (c)
                   and  result =  c
          )
```

createLinkComponent takes a link and a presentation specification and utilises *addComponent* to create a new link component. Link consistency has to be proven.

```
context  Hypertext :: createLinkComponent (link:Link, ps: PresentSpec) :
Component
pre:     -- none
post: Component. allInstances →  exists ( c: Component  |
                    c.oclIsNew
                    and  c.compBase =  link
                    and  c.compInfo.presSpec  =  ps
                    and  self.addComponent (c)
                    and  result =  c
          )
post:    linkConsistency
```

createCompositeComponent takes a collection of base components and a presentation specification and utilises *addComponent* to create a new composite component. It must be ensured that any subcomponent of the new composite are already in the hypertext.

```
context  Hypertext :: createCompositeComponent (cp: Composite, s: Set
(Component), ps: PresentSpec) :  Component
pre:    s.oclIsTypeOf (Sequence)
post: Component. allInstances → exists ( c: Component |
                c.oclIsNew   and
                and  c.compBase =  cp
                and  c.compInfo.presSpec  =  ps
                and  self.createComponent (c)
                and  s.allInstances → forAll ( s: S | components → includes (s)
                and  c.children  = c.children@pre → including (s)
        )
              and  result =  c
        )
```

createNewComponent is the function that will ultimately be invoked from the run-time-layer. Here an abbreviated form is used as OCL requires expressions in the body of an if-then-else.

```
context  Hypertext :: createNewComponent (bc:BaseComponent, ps:PresentSpec, s:
Set (Component)) : Component
pre:    -- none
post:  result = if  bc.oclIsTypeOf (Atom)
                then  createAtomicComponent (bc, ps)
                 else  if  bc.oclIsTypeOf (Link)
                       then createLinkComponent (bc, ps)
                       else  if bc.oclIsTypeOf (Composite)
                             then createCompositeComponent (bc, s, ps)
                             else   -- none
                               endif
                     endif
                 endif
```

Removing a Component

The operation *deleteComponent* eliminates a component from the hypertext ensuring that all links whose specifiers resolve to that component are removed.

```
context  Hypertext :: deleteComponent (uid:UID)
pre:    components → includes (accessor (uid))
post:  let  lIDs =  linksTo (uid) → including (uid)
        in  lIDs → iterate ( lid:lIDs |
        components  =  components@pre →  excluding (lid)
        )
post:  linkConsistency
```

Modifying a Component

Components are modified by the operation *modifyComponent* that ensures that the associated information as well as the type (atom, link or composite) remains unchanged and that the resulting hypertext remains link consistent. The resolver is not modified when modifying a component as the new component overrides the old one.

```
context  Hypertext :: modifyComponent (uid:UID, new:Component)
pre:     components → includes (accessor (uid))
post:  let    old =  accessor (uid)
       in    components = components@ pre→  excluding (new)
                          including (old)
             and  oclType (new.compBase)  = oclType (old.compBase)
             and  new.compInfo  =  old.compInfo
post:  linkConsistency
```

Retrieving a Component

The operation *getComponent* takes a UID and uses the accessor function to return a component. If the UID represents a link component, it returns either a source or a destination specifier for that component.

```
context  Hypertext :: getComponent (uid:UID) : Component
pre:    components → includes (accessor (uid))
post:  result =  accessor (uid)
```

Accessing and Modifying Attributes

The Dexter model includes the following three operations that allow for manipulation of attributes of components. These operations are *attributeValue*, *setAttributeValue* and *allAttributes*.

The first one takes a component UID and an attribute and returns the value of the attribute.

```
context  Hypertext :: attributeValue (uid:UID, a:Attribute) : Value
pre: components → includes (accessor (uid))
post:  Components.allInstances → exists ( c:Component | c =  accessor (uid)
          and   c.attributes → select (at:Attribute | at = a
           implies   result  =  at.value
         ) )
```

The second operation is *setAttributeValue*, that given a component UID, an attribute and a value, it sets the value of the attribute.

```
context  Hypertext :: setAttributeValue (uid:UID, a:Attributes, v:Value)
pre: components → includes (accessor (uid) )
post:  Components.allInstances → exists ( c:Component | c =  accessor (uid)
           and  Attributes.allInstances → exists (at:Attribute |
                at = c.attributes  and   at =  a
```

implies at.value = v
))

The third one, *allAttributes* returns the set of all component attributes.

context Hypertext :: allAttributes () : Set (Attribute)
pre: -- none
post: result = Attributes.allInstances → select (at:Attribute |
Component → exists (c:Component | c.compInfo.attributes → includes
(at)
))

3.3 The Run-time Layer

The run-time layer describes the mechanisms supporting the user's interaction with the hypertext. The fundamental concept of this layer is the instantiation. An instantiation is a presentation of the component to the user. It can be considered as a kind of run-time cache of the component as the user sees and edits a copy of the component. Thus, more than one instantiation for any given component can coexist.

Figure 5 shows the classes of the Run-time Layer that are described in the Dexter Reference Model and part of the Storage Layer.

Instantiation of a component also results in instantiation of its anchors. An instantiated anchor is known as a link marker *(LinkMarker*). In order to follow the same structure as in the storage layer, the instantiation is a complex entity that consists of a base instantiation *(BaseInstantiation)*, a sequence of link markers and a function mapping link markers to the anchors they instantiate. Base instantiation is a primitive in the model and represents the presentation of a component to the user.

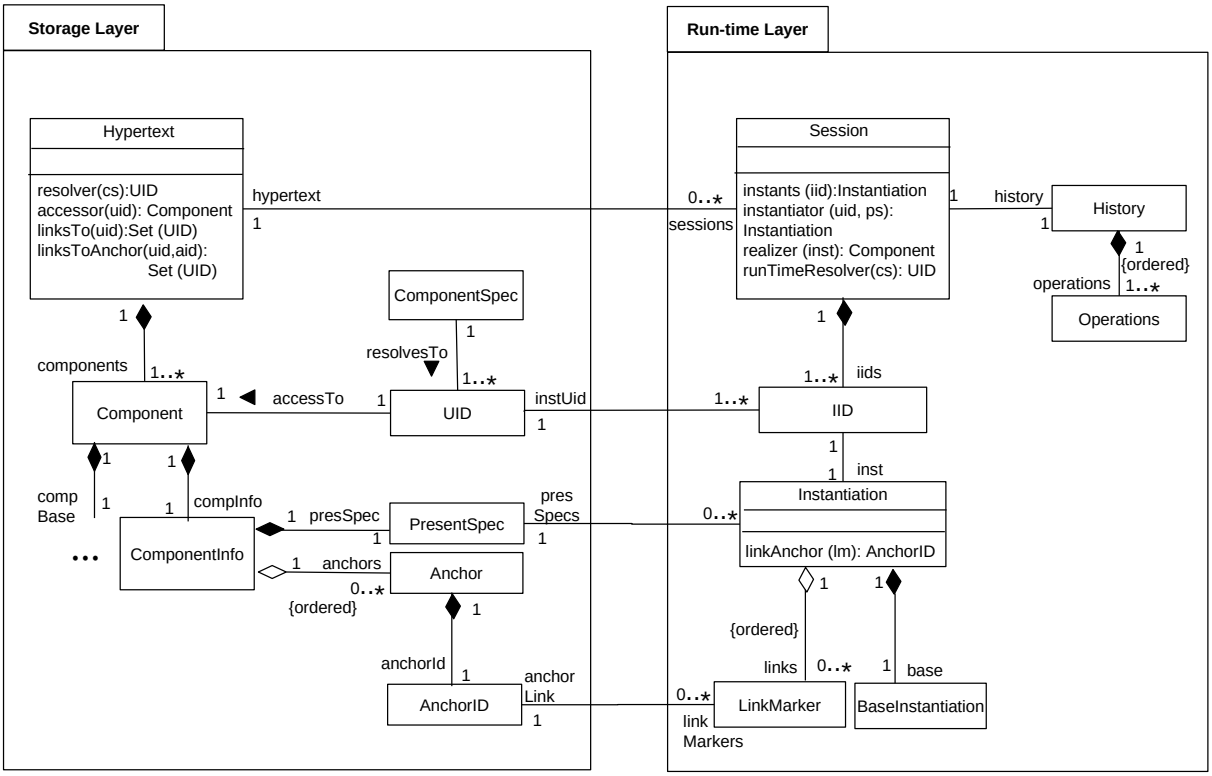

Figure 5: ***UML Class Diagram for the Run-time Layer and Part of the Storage Layer***

Instantiation

Each instantiation has a unique instantiation identifier from a given set of instantiations ID *(IID)*. In addition, an instantiation consists of, according to Halasz and Schwarz (1994), a base instantiation which "represents" a component, a sequence of link markers which "represents" the anchors of the component, and a function mapping link markers to anchor IDs called "link anchor" *(operation linkAnchor).*

context Instantiation : linkAnchor (lm : LinkMarker) : AnchorID
pre: links → include (lm)
post: result **=** lm.anchorLink

The invariant " dom linkAnchor = ran links" for the operation link anchor demands that for every link marker the function link anchor maps the link marker to an anchor ID.

context Instantiation
inv dom linkAnchor = ran links:
links → forAll (lm: LinkMarker | links → includes (lm)
implies AnchorID.allInstances → exists (aid : AnchorID |
linkAnchor (lm) = aid
))
and LinkMarker.allInstances → exists (lm : LinkMarker |
linkAnchor (lm) = aid
implies links → includes (lm)

)

Session

In the Dexter Reference Model the session contains the hypertext being accessed, a history, a mapping from IIDs of the session's current instantiations to the corresponding components of the storage layer, an *instantiator* function, a *realizer* function and a *run-time resolver* function. This is represented by a class *Session* with an association to the class *Hypertext* and to a class *History.*

The history records all the operations a user performs during a session, i.e. since the last open session. There are seven different types of operations that a user can perform during a session. This operations are: open and close a session, present and unpresent an instantiation of a component, create a new instantiation during a session as well as edit, save or delete an instantiation.

For the manipulation of instantiations a mapping function is defined from instantiations to components. Instantiations are generated for a session. Given an instantiation identification, the function *instants* returns the instantiation of the component and the function *instantsUid* the UID of the corresponding component.

context Session :: instants (iid: IID) : Instantiation
pre: iids → includes (iid)
post: result = iid.inst

context Session :: instantsUid (iid: IID) : UID
pre: iids → includes (iid)
post: result = iid.instUid

The *instantiator* is the core of the run-time model. This function returns, given a UID of a component and a presentation specification, an instantiation of the component that is part of the session. The presentation specification is a primitive in the model that contains information about how the component is to be presented by the system during instantiation.

context Session :: instantiator (uid: UID, ps: PresentSpec) : Instantiation
pre: hypertext.components → includes (accessor(uid))
and accessor (uid).compInfo.presSpec = ps
post: result = Instantiation.allInstances → select (ins:Instantiation |
ins.instPresSpec = ps and ins.iid.instUid = uid
)

The inverse function to the instantiator is the *realizer*. It takes an instantiation and returns a "new" component reflecting the recent changes due to editing the instantiation. This returned component is the input for the modifyComponent operation of the hypertext of the storage layer.

```
context  Session :: realizer (inst: Instantiation) : Component
pre:     Instantiation.allInstances → includes (ins)
post:   hypertext.components  → exists (c:Component |
          c.oclIsNew
          and  c.compBase =  ins.Base
          and  ins.links → forAll ( lm:LinkMarker | links → includes (lm)
                implies   c.compInfo.anchors.anchorID  = ins.linkAnchor (lm)
               )
          and  ( let  uid  = instantsUID (ins.iid)
                in  hypertext.modifyComponent (uid,c)
                )
          and  result  =  c
         )
```

The session's run-time resolver is the run-time version of the storage's layer resolver operation. It maps component specifiers into component UIDs. The run-time resolver is needed when run-time information is used for the resolution process, i.e. when history or time aspects are taken into account in the process. The storage layer resolver wouold not be able to handle this specification. The run-time resolver is a superset of the storage layer resolver.

```
context  Session :: runTimeResolver (cs: ComponentSpec) : UID
pre:    -- none
post:   result =  resolver (cs)
    -- cs is a component specification that may have changed during run-time
```

The following invariants have to be fulfilled for every Session:

```
context  Session
inv   first operation in a session is OPEN:
         history.operations → first  = #OPEN

inv    the set of components accessible by the accessor function is equal to set of
components realised from instantiations:
       UID.allInstances  → forAll ( uid : UID |
          PresentSpec.allInstances → exist ( ps : PresentSpec |
                  accessor (uid)  =  realizer (instantiator(uid,ps))
       ) )

inv   storage layer resolver is a subset of the run-time resolver:
       ComponentSpec.allInstances → forAll ( cs : ComponentSpec |
           UID → exists  ( uid : UID |
              resolver (cs) = uid   implies   runTimeResolver (cs) = uid
           ) )
```

3.4 The Run-time Layer Functions

A set of functions are included in the Run-time-Layer at least to fulfil the presentation of the components of the Storage Layer.

Opening a Session

A Session starts with an existing hypertext (storage layer) and neither instantations nor history. The *openSession* has to fulfil the following constraint:

```
context  Session :: openSession (h: Hypertext)
pre:     self.oclIsNew
post:    h.sessions = h.sessions@pre → including (self)
          and  history → isEmpty
          and  history.operations → append (#OPEN)
          and  iids → isEmpty
```

Opening an Instantiation

There are several operations which can open a new instantiation: opening components, presenting a component, following a link and creating a new component.

The first operation is called *openComponents* and it opens up a set of new instantiations based on a set of existing components. The function uses as input a sequencce of specifiers and a sequence of present specifications.

```
context  Session :: openComponents (specs: Seq (Specifier), pspecs: Seq
(PresentSpec))
--   two sequences are defined instead of a set of pairs as in OCL all collections --
are flat
pre:  specs → size > 0  and  pspecs → size  =  specs → size
post: history.operations → append (#PRESENT)
post:  let  newiids: Set (IID) → isEmpty
              and  newinst : Set(Instantiation)  → isEmpty
         in  specs  → iterate ( j : Integer; r = true |
                  let  s = specs →  at (j)  and  ps = pspecs →  at (j)
                  in Instantiation.allInstances →  exists (ins:Instantiation |
                       ins.oclIsNew
                       and   IID.allInstances →  exists ( iid:IID |
                            iid.oclIsNew
                            and   instants (iid) = ins
                            and   ComponentSpec.allInstances →  exists ( cs:
                                     ComponentSpec |  s.compSpec = cs
                                     and  UID.allInstances →  exists ( uid:UID |
                                                  runTimeResolver (cs) = uid
                                                  and  instantiator (uid, ps) = ins
                                                  and  instantsUID (iid) = uid
                                     ) )
                       and  newiids →  including (iid)
```

```
                and  newinst → including (ins)
        ) ) )
        and  iids → union (newiids)
        and  iids.inst → union (newinst)
```

The second one, *presentComponent*, is the operation that calls openComponents to present just one component given one specifier and one presentation specification.

```
context  Session :: presentComponent (spec:Specifier, pspec:PresentSpec)
pre:    -- none
post:  openComponents ({spec}, {pspec})
```

Another way to open a component is to follow a link from a given link marker in a given instantiation and present all the components for which the associated links have specifiers with a direction that has value "TO". There may be more than one link involved because there may be more than one link associated with a particular anchor.

```
context  Session :: followLink (iid:IID, lm:LinkMarker)
pre:    -- none
post:   let   specs =  Specifier.allInstances → select (s:Specifier |
          s.direction = #TO
          and  AnchorID.allInstances → exists ( aid: AnchorID |
                  aid = instants (iid).linkAnchor (lm)
                  and   UID.allInstances  → exists ( uid :UID |
                        LinksToAnchor (instantsUid (iid), aid) → includes (uid)
                        and  accessor(uid).compBase.oclTypeOf (Link)
                      and  accessor(uid).compBase./anchorSpecs
                              → includes (aid)
                        and accessor(uid).compBase.specifiers → includes (s)
              ) ) )
        in   Sequence { 1.. (specs → size)} → iterate (i:Integer ;
                  result : pspecs: Sequence (PresentSpec) |
                  pspecs → at (i)  = (specs → at (i)).presSpec
                        )
                and  openComponents (specs,pspecs)
```

The *newComponent* operation models the opening of a new instantiation when a new component is created.

```
context  Session :: newComponent (base:baseComponent, ps:PresentSpec,
presentSpec :PresentSpec) : Component
pre:    -- none
post:   history.operations → append (#CREATE)
post:   result = hypertext.createNewComponent (base, ps)
            and  UID.allInstances → exists (uid : UID  |
                    Instantiation.allInstances → exist ( ins : Instantiation |
                        IID.allInstances → exists (iid : IID  |
```

```
                                iids → excludes (iid)
                                and ins = instantiator (uid,presentSpec)
                                and accessor (uid) = c
                                and instants (iid) = ins
                                and instantsUid (iid) = uid
        ) ) )
```

Removal of an Instantiation

The operation *unPresent* models the removal of an instantiation.

```
context Session :: unPresent (iid:IID)
pre:    iids → includes (iid)
post:   history.operations → append (#UNPRESENT)
post:   iids = iids@pre → excluding (iid)
```

Modifying an Instantiation and/or a Component

An edit operation is used to modify instantiations (*editInstantiation*). The editing of an instantiation has no effect on the component. An explicit operation to save the changes result of an edit is required. This operation is the *realizeEdits*.

```
context Session :: editInstantiation (ins:Instantiation, iid:IID)
pre:    iids → include (iid)
post:   history.operations → append (#EDIT)
post:   let oldIns = instants(iid)
        in  iids.inst = iids.inst@pre → excluding (oldIns) → including (ins)
```

```
context Session :: realizeEdits (iid:IID)
pre:    iids → include (iid)
post:   history.operations → append (#SAVE)
post:   Components.allInstances → exists ( c: Component |
                Instantiation.allInstances → exists ( ins: Instantiation |
                        UID.allInstances → exists ( uid: UID |
                                instants (iid) = ins
                                and instantsUid (iid) = uid
                                and realizer (i) = c
                                and hypertext.modifyComponent (uid,c)
        ) ) )
```

Deleting a Component

To delete a component this component has to be instantiated. Any other instantiations of the same component have also to be deleted.

```
context  Session :: deleteComponent (iid:IID)
pre:     iids → includes (iid)
post:    history.operations → append (#DELETE)
post:    UID.allInstances → select ( uid : UID |  uid = instantsUid (iid)
                implies  hypertext.deleteComponent (uid)
         )
post:  iids = iids@pre  → excluding (iid)
```

Closing a Session

A session ends when it is closed out, i.e. the last operation registered in the history has value CLOSE. All instantiations of components are deleted. Changes to instantiations that have not explicitly be saved will be lost.

```
context  Session :: closeSession()
pre:    history → size > 1   and  history.operation → first = #OPEN
post:  history.operation → append (#CLOSE)
post:  iids → isEmpty
```

Read-only Session

A read-only session can be modelled as follows:

```
context  Session
inv    read only session:
        history.operations → iterate ( op: Operation; result : Boolean = true | result
        and  ( op <> #SAVE' and  op <> #CREATE  and  op <> #DELETE
        ) )
```

4 Conclusions

An object-oriented formal specification of the Dexter Hypertext Reference Model is presented in this work. It is based on the graphical notation of the UML and it makes intensive use of the OCL for the specification of invariants for the model elements and for the specification of the pre- and post-conditions on operations. These operations describe the functionality of a hypertext system.

UML class diagrams allow for a visual representation of the reference model which show the concepts of the model and how they are related. This graphical representation is missing in the Z and ObjectZ specification. The addition of some additional constructs to the OCL would improve the readability of the specification. With constructs like *domain* and *range* some constraints can be simplified. An important difficult arise by the definition of the transitive closure. . Even though the length of the computing of the transitive closure presented by Mandel and Cengarle (1999) has been reduced by the use of the construct “let ...in ...” that has been included in the UML version 1.3., it remains unnecessarily complex. Besides some little improvements that would optimise the specification, it has turned out that OCL is adequate for the specification of the Dexter Hypertext Reference Model.

References

Gamma E., Helm R., Johnson R. and Vlissides J. (1995). *Design Pattens.* Addison Wesley.

Gronbaek K. and Trigg R. (1994). Design Issues for a Dexter-Based Hypermedia System. *Communications of the ACM 37(2), Gronbaek K. and Trigg R. (Eds.),* 40-49.

Halasz F. and Schwartz M. (1990). The Dexter Hypertext Reference Model. *NIST Hypertext Standardization Workshop.*

Halasz F. and Schwartz M. (1994). The Dexter Hypertext Reference Model. *Communications of the ACM 37(2), Gronbaek K. and Trigg R. (Eds.),* 30-39.

Hardman L., Bulterman C. and van Rossum G. (1994). The Amsterdam Hypermedia Model. *Communications of the ACM 37(2), Gronbaek K. and Trigg R. (Eds.),* 50-62.

Mandel L. and Cengarle M.V. (1999). On the expressive power of OCL.

Richters M. and Gogolla M. (1999). A Metamodel for OCL. In Proceedings of the Conference The Unified Modeling Language beyond the standard (UML´99). LNCS 1723, Springer Verlag.

Spivey J. (1992). *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall International, second edition.

UML Version 1.3 alpha R5 (1999). *Object Constraint Language*. http://www.rational.com/uml/resources/documentation/index.jtmpl

Van Ossenbruggen J. and Eliëns A. (1995). The Dexter Reference Model in Object-Z. http://www.cs.vu.nl/~dejavu/papers/dexter-full.ps.gz

Warmer J. and Kleppe A. (1999). *The Object Constraint Language: Precise Modeling with UML*. Object Technology Series. Addison-Wesley.