An Object-Oriented Hypermedia Reference Model formally specified in UML

Nora Koch[1]
F.A.S.T. Applied Software Technology GmbH, Germany
E-mail: koch@fast.de

INTRODUCTION

The Dexter Hypertext Reference Model and some of its variants gained wide acceptance as a basis for the design of hypermedia systems and interoperability tools. It was formalised by Halasz and Schwarz (1990) in Z, a specification language based on set theory. Since then, the object-oriented paradigm is widely adopted in design and implementation of information systems. In addition, more emphasis is now put on visual modeling languages that make models more intuitive. A first object-oriented specification for the Dexter Model was presented by Van Ossenbruggen and Eliëns (1995). It is an Object-Z approach without graphical representation.

This work presents an object-oriented formal specification of a Dexter-based reference model for hypermedia systems in the Unified Modeling Language (UML). The specification consists of a visual representation with UML class diagrams supplemented with formal constraints on model elements, i.e. invariants on elements as well as pre-conditions and post-conditions on operations written in the Object Constraint Language (OCL). UML has been chosen because it is the standard modeling language; OCL is part of the UML (1999).

A visual representation has the advantage of showing at a glance the relevant concepts, how they are organised and how they are related to each other. This semi-formal graphical representation is supplemented with semantic information formally written in OCL. The use of OCL improves the model precision – as stressed by Richter and Gogolla (1999) – compared to constraints imposed when written as text. In this work it allows for an object-oriented formal specification that is comparable to a Z (Halasz &

[1] She is also an invited researcher at the University of Munich (LMU), Germany.

Schwarz, 1990) or a VDM specification (Tochtermann & Ditrich, 1996), for example.

This work is structured as follows: The second section gives an overview of the state-of-the-art in the field of reference models for hypermedia systems. The third section presents the goals of the current approach. Section four describes the argument of using UML and OCL. The fifth section presents the core of the model. Section six summarises the extension possibilities of this model. Finally, in the last section some conclusions and future steps are outlined.

BACKGROUND

For the specification of models, formal, semiformal or informal techniques can be used. Formal techniques are those based on mathematics, logic or algebra and for which syntax, semantics and manipulation rules are explicitly defined. Semiformal techniques are mainly diagram-based and tabular-based techniques, which present information in a structured form. Informal techniques are those that only use the natural language. Several reference models have been developed in the area of hypermedia.

The formal approaches include amongst others the Trellis Model (Furuta & Scotts, 1990), the Dexter Model (Halasz & Schwartz, 1994) and the Dortmund Family of Models (Tochtermann & Dittrich, 1996). The former is a specification based on the Petri net formalism, that defines five different levels of abstraction. The Dexter Hypertext Reference Model is the most well known model. It is formally specified in Z and was often used as a discussion basis and for extensions. DHM (Devise Hypermedia Model) of Grønbæk and Trigg (1996) is one of this extensions. The Dortmund Family of Models is formalised in the Vienna Development Method (VDM) with the goal of introducing flexibility using alternative data type specifications. Most of these models focus on the static hypertext or hypermedia structure.

To the semi-formal models we count the aforementioned DHM, the Amsterdam Hypermedia Model (AHM) of Hardman, Bulterman, and van Rossum (1994) and the Adaptive Hypermedia Application Model (AHAM) of De Bra, Houben and Wu (1999). In contrast to the Dexter Model, DHM is an object-oriented and semi-formal approach. It presents a class diagram in the notation proposed by Coad and Yourdon (1991) of part of the hypermedia model (mainly classes of the Storage Layer). It introduces the concepts of location and reference specification. It does not model the Run-Time Layer and operations are informally described. AHM is also an informal extension of the Dexter Model, which supports the modeling of dynamic media, such as audio, video and animation to allow for specification of temporal relationships between the data items. It introduces concepts, such as channel and synchronisation arcs. AHAM is a Dexter-based approach for adaptive hypermedia systems, which specification is tuple-based.

GOALS OF THIS APPROACH

Given such a variety of reference models for hypermedia the question should be asked: Why should another reference model for hypermedia systems be defined? The main reasons for such a project are:

- To produce an approach that is both object-oriented and formal,
- To obtain an easily extendable core reference model,
- To supplement semi-formal specification techniques with formal specification,
- To use a standard notation for the visual representation of object-oriented models,
- To define a less mathematical and widely comprehensible formal specification.

After analysis of amongst others the approaches mentioned above, the Dexter Model proved to be suitable for use as a basis of the approach developed and outlined in this work. It is a strong Dexter-based model that includes a visual representation of the reference model in the standard UML supplemented with OCL constraints that formally specify invariants, and pre- and post-conditions for operations.

Just as in the case of the Dexter Model, the hypermedia space is modelled as a three layer architecture. The layers 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 description of static and dynamic aspects of the Storage Layer and the Run-Time Layer, and the mechanisms of the interfaces, Anchoring and Presentation Specification (see Figure 1). The Within-Component Layer is purposely neither elaborated within the Dexter Model nor within our reference model.

The main goal of the reference model is to describe the network of nodes and links in the Storage Layer, i.e. the mechanisms by which these links and nodes are related. The nodes are treated in this layer as general data containers. The content and structure within the hypermedia nodes are described in the Within-Component Layer. The Run-Time Layer contains the description of the presentation of nodes and links focusing on user interaction.

As it is a Dexter-based reference model we decided to use the Dexter terminology, resisting the temptation to use more expressive names, such as “end point” instead of specifier. End point was introduced by DHM that also replaced Dexter’s anchor value by location specification. Although, we introduce some changes – mostly simplifications – to the Dexter Model, such as only allowing atoms to have content, to aggregate components information, i.e. attributes, anchors and present specifications directly to component, links have exactly two specifiers, an end point “from” and end point “to”.

USING UML FOR A VISUAL AND FORMAL SPECIFICATION

The standard UML is used in this chapter to visualise and specify the reference model. UML diagrams and modeling elements allows for a graphical representation of the reference model and the use of OCL (part of UML) allows for a formal description of the functionality of the model. The Run-Time Layer, Storage Layer and the Within-Component Layer, into which the reference model divides a hypermedia system, are represented as UML packages. The UML class diagram of Figure 1 shows the architecture model of hypermedia systems composed by these packages and the interfaces between them, i.e. the Anchoring and Presentation Specification interfaces.

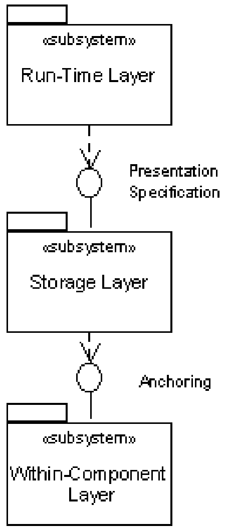

Figure 1: Architecture Model of Hypermedia Systems

In the following sections the object-oriented representation with UML of the Storage Layer and Run-Time Layer are presented as well as an OCL specification of some operations of these layers. These operations are functions to access components and anchors, authoring and presentation functions. Note that a complete specification of the reference model is not within the scope of this chapter. The UML-based Specification Environment (USE, 2001) was used to check the specification of the reference. In this chapter some of these expression are simplified as follows with the objective of augmenting readability: spaces are included in invariants' names, non-side effect-free operations are included in constraints – for abbreviation – whenever they can be replaced by an expression, and the operation "oclAsType" is sometimes omitted to increase clarity (Warmer & Kleppe, 1999).

The use of a visual representation has the advantage that some invariants do not require an OCL constraint as they are included in the visual specification. In contrast, in a Z specification they must be explicitly included. For example, the constraint "a link is a sequence of two specifiers" is given by the aggregation association of type composition between the *classes Link* and *Specifier* with multiplicity 2 and property {ordered}.

THE CORE OF THE HYPERMEDIA REFERENCE MODEL

The Storage Layer

The Storage Layer describes the structure of a hypermedia as a finite set of components together with two functions, a *resolver* and an *accessor* function. These concepts are modelled by a *class Hypermedia* and a *class Component*. With the two operations *resolver* and *accessor* defined for the *class Hypermedia*, it is possible to "retrieve" components. Every component has a globally unique identity *(class UID)*. The accessor function allows for the "access" to a component given its UID. UIDs provide a guaranteed mechanisms for addressing any component in a hypermedia.

As with the Dexter model, addressing is accomplished in a indirect way based on the entities called anchor *(class Anchor)* consisting of two parts: an anchor ID *(class AnchorID)* and anchor value *(class 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 unique identification of the anchor within the scope of the hypermedia.

Figure 2 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. In the remainder of this section some of these classes are described in detail.

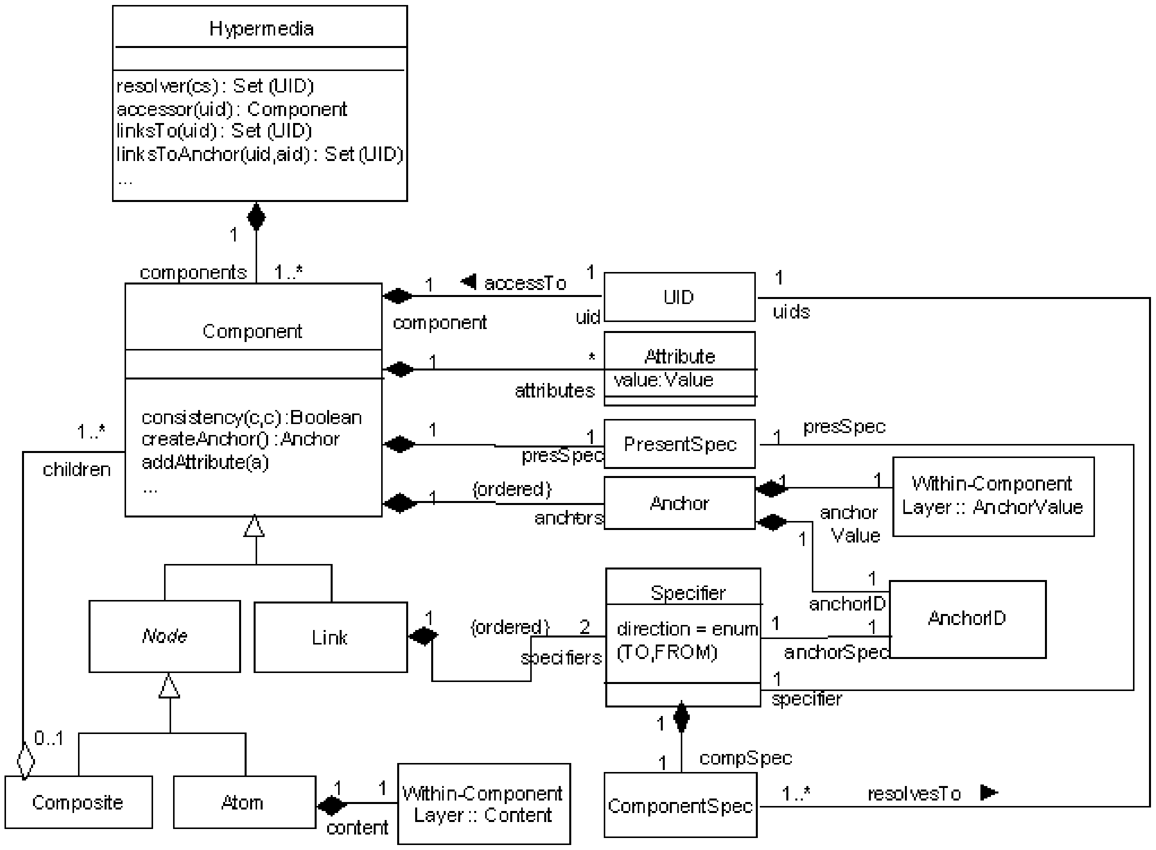

Figure 2: Storage Layer Model

Component

A component is an abstract representation of an information item from the application domain. It is represented with an *abstract class Component*. Every component has a unique identifier (UID) associated with it. These UIDs are assumed to be unique in the whole universe of discourse. A component can either be a node *(class Node)* or a link *(class Link)*. A node, in turn, can either be an atom *(class Atom)*, or a composite of other components *(class Composite)*. The composite pattern (Gamma & et. al., 1995) is used to represent this structure as shown in Figure 2. For links description see subsection below.

A component has associated component information that describes the properties of the component. These properties are a set of attributes *(class Attribute)*, a presentation specification *(class PresentSpec)* and a sequence of anchors *(class Anchor).* Attributes allow definition of arbitrary properties, such as keywords attached to a component. The list of anchors provides a mechanism for specifying the end points of the links that relate this node to other nodes in the network. The presentation specification is used as the interface to the Run-Time Layer.

Compared to the Dexter model there is no need in the object-oriented specification to include classes for component base and component information. Attributes, anchors and presentation specifications aggregate directly to class component. In addition we introduce the concept of node in the class hierarchy *Component-Node-Composite-Atom-Link.* Composites, in contrast to the Dexter Model cannot have content.

The model assures "type consistency" between components, i.e. two components are "type consistent", if they are both atoms or both links or both composites. The "type consistency" is specified by the following OCL constraint.

```
context  Component :: consistency (c1:Component, c2: Component): Boolean
post:    result  = c1.oclIsTypeOf (Atom) and c2.oclIsTypeOf (Atom)
                   or  c1.oclIsTypeOf (Composite) and c2.oclIsTypeOf (Composite)
                  or   c1.oclIsTypeOf (Link) and c2.oclIsTypeOf (Link)
```

Anchor

Anchoring provides a mechanism that allows for linking between nodes but also for addressing (referring) to locations within the content of a component. An anchor is defined as a pair consisting of an anchor ID *(class AnchorID)* and an anchor value *(class AnchorValue).* The anchor ID is an identifier which uniquely identifies its anchor within the scope of the component, of which it is part. 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. The UML diagram (Figure 2) includes the name of the package for classes that do not belong to the

"Storage Layer" package.

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  Component
inv  number of anchors:
     anchors → size  =  anchors.anchorID → asSet  → size
```

Link

A link consists of a sequence of two specifiers. A specifier defines one single end point of the link. In comparison links of the Dexter Model, are allowed to have more than two specifiers. A link is modelled by a *class Link* and an aggregation association of type composition to a *class Specifier*. The *class Specifier* has an attribute *direction.* The direction encodes whether the end point is the source of the link (from) or the destination (to). All links should have one source and one destination. The following invariant formalises this, i.e. at least one specifier with value "from" and one specifier with value "to" for the direction.

```
context Link
inv one specifier with direction FROM and one with direction TO:
     specifiers.direction →  exists ( s: Specifier |  s.direction = #FROM)  and
     specifiers.direction →  exists ( s: Specifier |  s.direction = #TO)
```

Hypermedia

A hypermedia system consists of a set of components, i.e. nodes and links, a function *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 hypermedia is represented in the object-oriented model by a *class Hypermedia*, which is a composition of objects of type *Component*. A hypermedia requires at least one component (This constraint is formalised by the multiplicity in the diagram).

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

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

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

Access to Components and Anchors

The *Hypermedia* class includes two operations for links and anchors, i.e. ensuring the navigation functionality of the hypermedia system. They are the *linkTo* and the *linkToAnchor* functions. The *linkTo* function returns the set of links that resolve to a specific component. The *linkToAnchor* obtains the set of links that resolve to a specific anchor. The following are the OCL pre- and post-conditions for the operations *linksTo* and *linkToAnchor*.

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

context Hypermedia :: linksToAnchor (uid:UID, aid:AnchorID) : Set (UID)
post:  result = linksTo (uid) → select ( lid: UID |
                   accesor(lid).oclIsTypeOf (Link)
                   and accessor (lid). /anchorSpecs → includes (aid) )
```

Hypermedia Invariants

Every class *Hypermedia* must satisfy four constraints. The first constraint is called the “components accessibility”. It assures that all components of a hypermedia system are accessible by means of the accessor operation. This can be formalised as follows:

```
context Hypermedia
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 nodes that can be retrieved by the *accessor* (accessor domain), i.e. the *resolver* function must be able to produce all possible valid UIDs. This can be proved proving the following two inclusions:

range of resolver $\subseteq$ domain of accessor and
range of resolver $\supseteq$ domain of accessor

The range of the *resolver* is included in the domain of the *accessor* by definition. The following OCL constraint thus proves then that the domain of the *accessor* is included in the range of the *resolver*.

```
context Hypermedia
inv range of resolver ⊇ domain of accessor:
       UID.allInstances → forAll (uid:UID |
         components.specifiers.compSpec → exists
             ( cs:ComponentSpec | resolver (cs) → includes (uid) ) )
```

The third constraint ensures 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. It is specified in OCL using the previously defined operation *linkTo.*

```
context Hypermedia
inv  anchors Ids of a component =  anchors IDs of the links resolving
    to the component:
        components →  forAll  ( c : Component |
c.anchors.anchorID = Link.allInstances → select ( l:Link |
        UID.allInstances → exists ( uid: UID |
            l.specifiers.anchorSpecs =
            linksTo(uid).component.anchors.anchorID
            and  accessor (uid) = c ) ) )
```

The fourth constraint guarantees that a hypermedia systems does not include cycles in the component/sub-component relationship, i.e. no component may be a sub-component (directly or transitively) of itself. Formally, it means that a component is not included in the transitive closure of this component, i.e. it has to be proved that the transitive closure of the relation children does not contain a pair with two equal elements.

The OCL specification of this constraint has to deal with some difficulties. To calculate the transitive closure, first the association *children* must be transformed into an association class. Unfortunately, OCL collections of collections are flattened, Mandel and Cengarle (1999) propose as a solution to define the *transClos* as a sequence of an even number of elements, where even positions belongs to components and odd positions to composites. The transitive closure can be calculated then in two steps. First an operation called *subcomponents* is defined that builds a sequence of pairs of components including all components that have children of type composite. In the second step an operation *transitiveClosure* is defined. It applies the Warshall's algorithm to a given sequence of composites (pair of related composites) to calculate the transitive closure. The result is a sequence of all pair of composites included in the transitive closure of the initial sequence. Due to space limitations, the OCL specification of this invariant is not included here; for more details see Koch (2001).

The Authoring Functions

In addition to the functions to manipulate anchors and links, the hypermedia reference model includes authoring functions. They are mainly required to update the model, i.e. to create components (*createAtomicComponent*, *createCompositeComponent*, and *createLink*), to modify components *(modify Component)* and to remove components *(deleteComponent)* as well as to manipulate attributes *(attributeValue, setAttributeValue* and *allAttributes)*.

These functions are defined as operations of the *class Hypermedia*. OCL constraints are used to specify the pre- and post-conditions that have to satisfy these authoring functions. Two examples of OCL expressions associated to authoring methods are presented below.

Modifying a Component

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

```
context  Hypermedia :: modifyComponent (uid:UID, new:Component)
pre:     components  →  includes (accessor (uid) )
post:   let    old =  accessor (uid)
        in    concistency (new, old)
            and components = components@pre  →  excluding (old)
                                                 →  including (new)
```

Modifying Attributes of a Component

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

```
context  Hypermedia :: setAttributeValue (uid:UID, a:Attributes, v:Value)
pre:    components  →  includes (accessor (uid) )
        and  components.attributes →  includes (a)
post:  let  atr = Attributes.allInstances →  select (at:Attribute  |  at = a
             and  components →  exits ( comp:Component  |  comp =  accessor (uid)
                  and comp.attributes (at) )   →  includes (at) )
                                             →  asSequence →  first
       in   atr.value = v
```

The Run-Time Layer

The Run-Time Layer describes how the mechanisms supporting the user's interaction with the hypermedia system, which comprise how the components are presented to the user. This 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). It should be noted that more than one instantiation for a component may exist simultaneously and that a user may be viewing more than one component. Thus, the fundamental concepts of this layer are the instantiation and the session, which are modelled by a *class Instantiation* and a *class Session*. Figure 3 shows the classes of the Run-Time Layer and a partial visualisation of the Storage Layer, mainly including classes that are related to the classes of the Run-Time Layer.

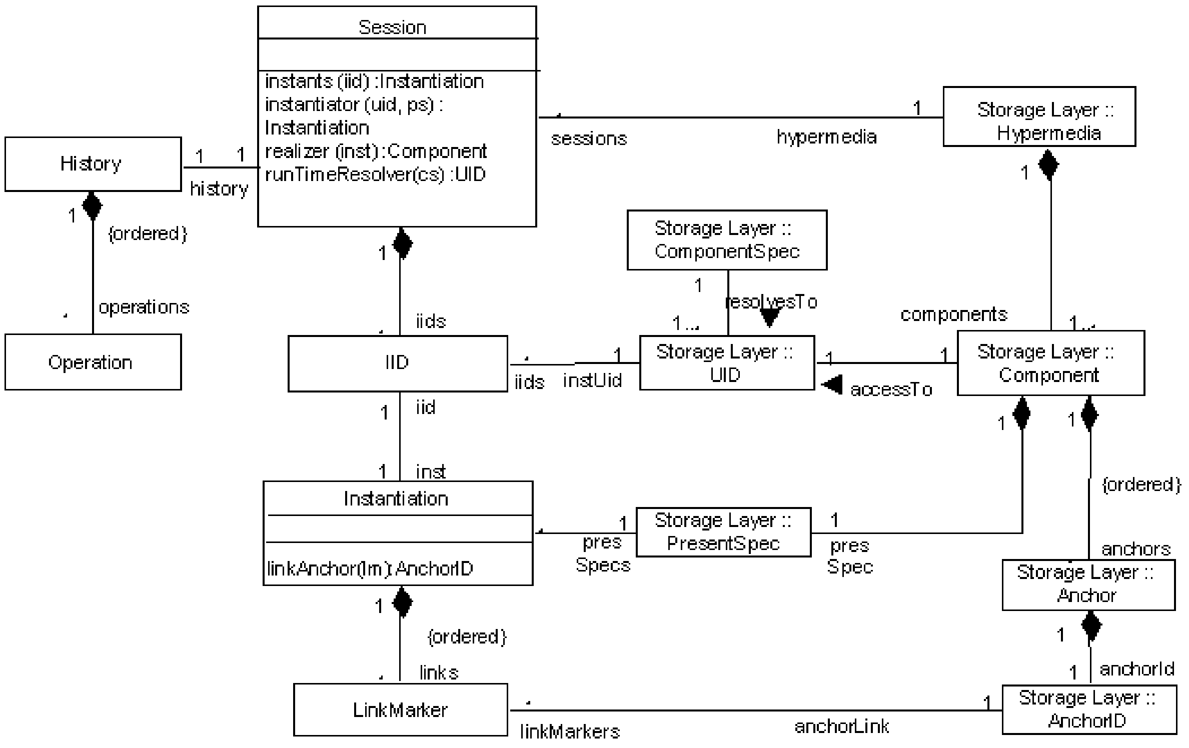

Figure 3: Model of the Run-Time Layer

Instantiation

Each instantiation has a unique instantiation identifier from a given set of instantiations ID *(class IID).* Instantiation of a component also results in instantiation of its anchors. Therefore, an instantiation has associated a set of link markers which "represents" the set of Run-Time anchors of the component, and a function maps instantiated anchors to anchor IDs. This function is called "link anchor" *(linkAnchor)* and an instantiated anchor is known as a link marker. This mapping is modelled in the UML diagram with aggregation associations between *class Instantiation* and *class LinkMarker* as well as an association between *class LinkMarker* and *class AnchorID* (see Figure 3).

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

The invariant "the domain of linkAnchor is equal to the range of links" for the operation link anchor demands that for every link marker the function link anchor maps the link marker to an anchor ID. The following is the OCL specification of this invariant.

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

Session

In order to keep track of all these instantiations the Run-Time Layer uses an entity session. In a session the user can open a component that results in the creation of an instantiation, edit an instantiation, save the modifications, create a new component or delete a component. The most common action is to follow a link, which takes the IID of an instantiation together with the link marker contained within that instantiation and then presents to the user the component resolved according to the content of a link component specifier, i.e. components that are the end point destination of all links.

The Run-Time package includes therefore a class *Session* with an association to the class *Hypermedia* and an aggregation association of type composition to a class *History*. The history records all the operations (interactions) a user performs during a session, i.e. since the last open session. Similarly to the Dexter Model seven basic types of actions are included that a user can perform during a session. These actions 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 one session. Given an instantiation identification, the method *instants* of the *class Session* returns the instantiation of the component and the method *instantsUid* returns the UID of the corresponding component. The following is the formalisation of pre- and post-condition for the operation *instantsUid* in OCL:

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

The *instantiator* is the core of the Run-Time model. Given a UID of a component and a presentation specification, the function returns, an instantiation of the component that is part of the session. The presentation specification is a primitive in the model, which contains information about how the component is to be presented by the system during instantiation. The following OCL constraint expresses the pre- and post-conditions of the *instantiator* method.

```
context  Session :: instantiator (uid: UID, ps: PresentSpec) : Instantiation
pre:    hypermedia.components →  includes  (accessor(uid) )
                    and   accessor (uid).presSpec =  ps
post:   result =  iids.inst   →  select (ins:Instantiation  |
                    ins.instPresSpec = ps   and  ins.iid.instUid = uid )
                                  →  asSequence →  first
```

The inverse function to the *instantiator* is the *realizer*. This takes an instantiation and returns a “new” component reflecting the recent changes due to editing the instantiation. The following invariant assures that the set of components accessible by the accessor function is equal to the set of components realised from instantiations. Thus, every session fulfils the following invariant:

```
context  Session
inv  range of accessor = range of realizer:
       UID.allInstances  →  forAll ( uid : UID  |
               PresentSpec.allInstances →  exist
             ( ps :  PresentSpec  |  accessor (uid)  =  realizer (instantiator(uid,ps) ) ) )
```

A read-only session can be modelled as follows:

```
context  Session
inv    read only session:
        not  history.operations →  forAll (op: Operation  |
              op.opn = #CREATE  or op.opn = #EDIT
    or op.opn = #SAVE  or op.opn = #DELETE)
```

The Presentation Functions

A set of functions is included in the Run-Time Layer with the purpose of fulfilling the presentation of the components of the Storage Layer, e.g. *opening a session, opening an instantiation, removal of an instantiation, modifying an instantiation* and/or *a component, deleting a component and closing a session*. There are several operations which can open a new instantiation: *opening components, presenting a component, following a link and creating a new component*. Due to limited space we only include OCL expressions of the first three functions.

Opening a Session

A session starts with an existing hypermedia (Storage Layer) and neither instantiations nor history. The *openSession* has therefore to fulfil the following constraint:

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

Opening an Instantiation

There are several operations which can open a new instantiation: open an instantiation, present a component, follow a link and create a new component. Browsing is the most common user activity. It starts with a user mouse click that activates the *followLink* function of the Run-Time Layer. The *resolver* and *accessor* functions are responsible for identifying the node to be accessed. A set of objects are involved in this open function. The operation *openInstantiation* opens up a new instantiation based on a existing component. The function uses a specifier and the present specification of the component as input.

```
context Session::openInstantiation (spec:Specifier,pspec:PresentSpec) : Instantiation
post: let newins = Instantiation.allInstances -> select (ins:Instantiation |
          ins.oclIsNew and IID.allInstances -> exists ( iid:IID | iid.inst = ins
          and ComponentSpec.allInstances -> exists (cs:ComponentSpec |
                    spec.compSpec = cs
          and UID.allInstances -> exists (uid: UID | cs.uids -> asSequence -> first = uid
          and uid = iid.instUID
          and ins.presSpec = pspec ) ) ) ) -> asSequence -> first
      in  iids.inst = iids.inst@pre -> including (newins)
          and result = newins
```

Removal of an Instantiation

The operation *unPresent* models the removal of an instantiation. The post-condition indicates that after the operation *unPresent* is completed a new operation is included in the history list and that the instantiation identifier is no longer included in the set of instantiation identifiers of the corresponding session.

```
context  Session :: unPresent (iid:IID)
pre:   iids →  includes (iid)
post: : let  op = Operation.allInstances  →  select ( o:Operation  |
                o.opn = # UNPRESENT )  →  asSequence →  first
        in  op.oclIsNew  and  history.operations →  asSequence =
                history.operations@pre →   asSequence ) → append (op)
      post:  iids = iids@pre  →  excluding (iid)
```

EXTENDING THE OBJECT-ORIENTED REFERENCE MODEL

The object-oriented specification of the hypermedia reference model presented in the previous section allows for an easy extension to model special hypermedia systems, such as adaptive or mobile hypermedia. The UML diagrams show a visual representation of the metamodel augmenting intuitive comprehension. OCL has the advantage of requiring a less in-depth mathematical background than other specification languages. Extensions are obtained enhancing the core reference model with new packages and classes, as well as adding attributes, operations and associations.

The Munich Reference Model for adaptive hypermedia systems is such an extension (Koch, 2001). It extends the core hypermedia reference model with user modeling and adaptive functionality. The three-layer structure is kept unchanged. Though the Storage Layer includes, in addition to the domain information, a user profile and a set of adaptation rules. In the UML representation the Storage Layer subsystem includes three packages: the Domain, the User and the Adaptation Metamodels. The Domain Metamodel corresponds – slightly adapted – to the Storage Layer of the core hypermedia reference model.

The User Metamodel includes a user manager and a model for each user, consisting of user attributes and attributes values. Attributes may be dependent on the domain or domain-independent. The former build an overlay model for each user, i.e. current appropriateness of each node. The later describes the user profile, i.e. the preferences, interests and tasks of the user.

The Adaptation Metamodel is defined as a set of rules, which can be classified into construction, acquisition and adaptation rules. Adaptation rules describe the implementation of techniques to dynamically adapt content, navigation and presentation of nodes. The dynamic generation of pages adapted to the current state of the user model is guaranteed by operations, such as *constructor, evaluator* and *trigger* that are applied by navigation in addition to the *accessor* and the *resolver* function.

The Run-Time Layer of the Munich Reference Model is also an extended version of the Run-Time Layer of the core of the hypermedia model shown in Figure 3. It observes the user behaviour and provides this information to the adaptation engine.

CONCLUSIONS AND FUTURE TRENDS

This chapter presents an object-oriented specification of a Dexter-based reference model. The UML notation is used for the visualisation of the reference model and OCL (defined as part of UML) is used for the formal specification of invariants for the model elements and for the specification of the pre-conditions and post-conditions on operations describing the functionality of hypermedia.

The visualisation with UML diagrams has the advantage of showing the concepts of the model at a glance and how these concepts are related. This graphical representation is lacking in the Dexter Hypertext Reference Model specified in Z or in the tuple-based description of AHAM. OCL is quite a new language and only few articles report on experiences using OCL (Baar, 2000). Apart from some minor improvements that would optimise the specification, it transpires that OCL is adequate as formal complementary language to the visual UML specification.

The object-oriented formal specification describe the basic structure and functionality to be addressed in the design of hypermedia systems (Hennicker & Koch, 2000). It was defined with the goal of easily allowing for extensions of this model. An extension for adaptive hypermedia systems is presented in Koch (2001). An important future step will be to specify a reference model for mobile software applications. In addition, new formal reference models will be required to cope with technological changes in hypermedia, such as the case of the Semantic Web – World Wide Knowledge, of Berners-Lee (1999), a Web based on typed and semantic links. An appropriated visualisation and formalisation could be obtained extending the object-oriented specification in UML/ OCL presented in this chapter.

REFERENCES

Baar T. (2000). Experiences with the UML/OCL-Approach in Practices and Strategies to Overcome Deficiencies. *Net.ObjectDays 2000*, Germany.

Berners-Lee T. (1999). *Weaving the Web*: *The Original Design and Ultimate Destiny of the World Wide Web.* Harper San Fransisco.

Coad P. & Yourdon E. (1991). *Object-Oriented Analysis*. Prentice Hall.

De Bra P., Houben G.-J. & Wu H. (1999). AHAM: A Dexter-based Reference Model for Adaptive Hypermedia. *Proceedings of the ACM Conference on Hypertext and Hypermedia*, 147-156.

Furuta R. & Stotts P. (1990). The Trellis Hypertext Reference Model. *Proceeding NIST Hypertext Standardization Workshop.*

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

Grønbæk K. & Trigg R. (1996). Towards a Dexter-based Model for Open Hypermedia: Unifying embedded references and link objects. *Proceedings of the Hypertext´96 Conference.*

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

Halasz F. & 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. & van Rossum G. (1994). The Amsterdam Hypermedia Model. *Communications of the ACM 37(2), Grønbæk K. and Trigg R. (Eds.),* 50-62.

Hennicker R. & Koch N. (2000). A UML-based Methodology for Hypermedia Design. *Proceedings of the Unified Modeling Language Conference, UML´2000*, Evans A. and Kent S. (Eds.). LNCS 1939, Springer Verlag, 410-424.

Koch N. (2001). Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modeling Techniques and Development Process. PhD. Thesis, LMU Munich, Uni-Druck, in print.

Mandel L. & Cengarle M.V. (1999). On the expressive power of OCL, World Congress on Formal Methods, 854-874.

Richters M. & 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.

Rumbaugh J., Jacobson I. & Booch G. (1999). *The Unified Modeling Language Reference Manual.*

Addison Wesley.

Tochtermann K. & Dittrich G. (1996). The Dortmund Family of Hypermedia Models. *Journal of Universal Computer Science*, 2(1), Springer Verlag.

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

USE: UML-based Specification Environment. University of Bremen. http://www.db.informatik.uni-bremen.de/projects/USE

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

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

APPENDIX: UML CONCEPTS

This appendix is organised as an alphabetical list of entries, each giving a brief description of one UML concept. The list only includes UML concepts used in this chapter. For more details see UML complete documentation (1999) or the UML Reference Manual (Rumbaugh, Jacobson & Booch, 1999).

Abstract class: A class that may not be instantiated. The name of the abstract class is shown in italics.

Aggregation: A form of association that specifies a whole-part relationship between an aggregate and a constituent part. An aggregation is shown as a hollow diamond adornment on the end of the association line at which it connects to the aggregate class.

Association: The semantic relationship between two or more classes that involves connections among their instances. An association is shown as a solid path connecting the borders of two classes.

Class: The descriptor for a set of objects that share the same attributes, operations, methods, relationships, and behaviour. A class represents a concept within the system being modelled. A class is shown as a solid-outline rectangle with three compartments separated by horizontal lines. The compartments hold the class name, list of attributes and list of operations, respectively. The middle and bottom compartments can be suppressed.

Class diagram: A graphical representation of the static view that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

Composition: A form of aggregation association with strong ownership and coincident lifetime of parts of a whole. Composition is shown by a solid diamond adornment on the end of the association line attached to the composite element.

Inheritance: A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. Inheritance between classes is shown as a solid-line path from the child element (subclass) to the parent element (superclass), with a large hollow triangle at the end of the path where it meets the more general element.

Interface: A named set of operations that characterised the behaviour of an element. An interface is displayed as a small circle with the name of the interface placed below the symbol.

Invariant: A semantic condition or restriction represented as an expression (constraint) that must be true at all times (or, at least, at all times when no operation is incomplete).

Multiplicity: A specification of the range of allowable cardinality values that a set may assume. Multiplicity is specified by a text expression consisting of a comma-separated list of integer intervals, each in the form “minimum .. maximum”.

Role: A named slot within an object structure that represents behaviour of an element that participates in a context. The rolename is shown by a graphic string placed near the end of an association path, at which it meets a class box.

Object: A discrete entity with a well-defined boundary and identity that encapsulates state and behaviour;

an instance of a class. The canonical notation for an object is a rectangle showing the name of the object and its class, all underlined.

Package: A general-purpose mechanism for organising elements into groups. Packages may be nested within other packages. A package is shown as a large rectangle with a small rectangle attached on one corner.

Precondition: A constraint that must be true when an operation is invoked.

Postcondition: A constraint that must be true at the completion of an operation.

Subsystem: A package of elements treated as a unit, including a specification of the behaviour of the entire package contents treated as a coherent unit. A subsystem has a set of interfaces that describe its relationship to the rest of the system and the circumstances under which it can be used. A subsystem is notated as a package symbol containing the keyword «subsystem».