Designing Dexter-based Cooperative Hypermedia Systems†

Kaj Grønbæk, Jens A. Hem, Ole L. Madsen, and Lennert Sloth

Computer Science Department,
Aarhus University,
Ny Munkegade 116, Bldg. 540
DK-8000 Århus C, Denmark.
Email: {kgronbak,nold,olmadsen,les}@daimi.aau.dk

ABSTRACT

This paper discusses issues for the design of a Dexter-based cooperative hypermedia architecture and a specific system, DeVise Hypermedia (DHM), developed from this architecture. The Dexter Hypertext Reference Model [Hala90] was used as basis for designing the architecture. The Dexter model provides a general and solid foundation for designing a general hypermedia architecture. It introduces central concepts and proposes a layering of the architecture. However, to handle cooperative work aspects, such as sharing material and cooperative authoring, we have to go beyond the Dexter model concepts. To deal with such aspects we have extended our implementation of the Dexter concepts with support for long-term transactions, locking and event notification as called for by Halasz [Hala88]. The result is a platform independent architecture for developing cooperative hypermedia systems. The architecture consists of a portable kernel that constitutes an object oriented framework for developing Dexter compliant hypermedia systems. It is a client/server architecture including an object oriented database (OODB) to store the objects implementing the Dexter Storage Layer. We use a general OODB being co-developed to support long term transactions, flexible locking, and event notification. The transaction and locking mechanism support several modes of cooperation on shared hypermedia materials, and the notification mechanism supports the users in maintaining awareness of each others' activity. The portable kernel was used to implement the DHM system on two quite different platforms: UNIX/X-windows and Apple Macintosh.

KEYWORDS

Dexter model, Open Hypermedia, CSCW, Shared materials, Object Oriented Database.

1. INTRODUCTION

The hypermedia work discussed here is part of the DeVise project at the Computer Science Department, Aarhus University, Denmark. The DeVise project is among other things developing tools to support cooperative design in a variety of application areas including large engineering projects. A large engineering project, constructing one of the worlds largest tunnel and bridge "links", is the primary user

---

† In Proceedings of the ACM conference on Hypertext, Seattle, USA, November 14-18, 1993 (Hypertext '93).

organization in the Esprit projects EuroCoOp and EuroCODE, from which the DeVise group gets parts of its funding. The use settings for our tools are characterized by cooperative work distributed over time, space and hardware platforms. Cooperative work in engineering projects raises several requirements for hypermedia, such as: shared databases, support for awareness among users of shared materials, access from multiple platforms, open architecture for integration of applications, portability, extensibility and tailorability. In particular the possibility for integrating applications, already developed for the engineering domain, with hypermedia facilities was an important requirement to meet. For a detailed discussion of our use setting, the engineering project, and its CSCW and hypermedia requirements, see [Grøn93].

No existing hypermedia system to our knowledge met these requirements on the platforms we needed to support. Having to build our own, we nonetheless wanted to benefit from the experience and expertise of past and present hypermedia designers. Thus, we decided to use the Dexter Hypertext Reference Model [Hala90] (called "Dexter" in the rest of this paper) as a basis for our development. Dexter attempts to capture the best design ideas from a group of "classic" hypermedia systems, in a single overarching *data* and *process* model. Although these systems have differing design goals and address a variety of application areas, Dexter managed to combine and generalize many of their best features.

We took the Dexter model as the starting point and developed an object oriented architecture from its concepts. A working hypermedia system prototype (called DeVise Hypermedia, or just "DHM") was developed for both a Unix and a Macintosh platform. The development environment is the Scandinavian Mjølner BETA System (MBS), see [Knud93; Mads93]. An object oriented database (OODB) [Ande92; Hem91] based on MBS is being developed in *parallel* with our hypermedia development. It is a general purpose OODB, i.e. the facilities work for any type of object independent of the application domain. The OODB design has, however, been highly influenced by the requirements from the parallel hypermedia development. Design issues related to the development of kernel hypermedia functionality are discussed in detail in [Grøn92b]. In this paper, we focus on design issues related to hypermedia support for cooperative work activities, e.g. cooperative authoring and sharing of materials in large design projects such as bridge construction and software development. These design issues also include a discussion of tailoring the OODB to meet cooperative hypermedia requirements.

COOPERATIVE WORK AND HYPERMEDIA SUPPORT

Design and authoring in large design projects involves cooperative work among individuals who contribute to the overall design task. Such work involves both explicit communication and coordination, and implicit coordination through shared materials [Sørg87]. For instance, work on different parts of shared materials needs to be coordinated and related. Cooperative design and authoring in such settings may be supported in many different ways. In cooperative design situations, a number of users are manipulating a large body of shared material using a variety of editors. We assume the shared materials to be hypermedia networks with large sets of components (nodes, links and composites). Such hypermedia networks may be subdivided into parts identified by composites containing a subset of the components in the hypermedia network. The kind of computer support to provide depends on needs for coordination of the work on different parts. To describe the kind of support we aim at providing, we have identified six different modes of cooperation on shared materials:

1) Separate responsibilities. The design material is divided into disjoint parts. Each part is manipulated by at most one user. Other users may inspect parts manipulated by others. The cooperation here is quite loose and will mainly consists of one user making use of designs developed by others.

2) Turn taking. As mode 1, but each part may alternate between different users. At most one user at a time is allowed to modify a given part. This mode of cooperation requires more support, to coordinate the work between users manipulating the same parts.

3) Dynamic exchange. During a session, users may exchange parts dynamically. One user A may want to modify a part currently being locked by another user B. User A may then ask user B to transfer his lock to user A during the session.

4) Alternative versions. Different users may develop alternative versions of the same part. Such parts may then have to be merged later.

5) Mutual sessions. Two or more designers may work on the same part at the same time (synchronously) with some direct communication channel open. All operations made by each designer are immediately updated on a shared copy of the part. A *cooperative commit* will update the part in the OODB. A variant of this cooperation mode is when each user makes changes that are not immediately committed, but may be undone without other users seeing them.

6) Fully synchronous sessions. As mode 5, except that several users work on the same part using a shared (global) window. In this mode all users share exactly the same view of the shared material (WYSIWIS) and they may have telepointers.

Our EuroCODE project aim at supporting this variety of cooperation modes on shared materials. Among these modes the hypermedia development mainly focuses on supporting the *implicit* and *asynchronous* cooperation on shared materials. Modes 1-3 are typical asynchronous cooperation modes that we aim at supporting directly by our hypermedia system. These modes call for support to create awareness among users about who is doing what in the shared body of materials. Chunks of the materials may be related by means of links and cooperation may take place through linked annotations to parts developed by others. These modes require a flexible locking scheme by the underlying database storing the hypermedia objects, in our case an OODB. The versioning approach represented by mode 4 is also an asynchronous mode of cooperation that may be supported for some kinds of materials to be shared [Magn93]. We are planning to provide such versioning support at the general OODB level, and utilize this for developing versioning support in hypermedia systems developed from our framework.

Modes 5 and 6 both represent variants of synchronous modes of cooperation on shared materials, they correspond to the *tightly-coupled* cooperation mode introduced by [Stre92]. The main difference between mode 5 and 6 is whether a shared view is maintained or not. In synchronous sessions all users have the same view of the hypermedia component being edited. In mutual sessions, several users may edit the same component in the hypermedia without maintaining the same view. These synchronous cooperation modes require extensions to the hypermedia system to support shared commitment of changes to the OODB. Support for multicasting updates and maintaining shared views will be provided by other EuroCODE sub projects developing a shared window system and a computer conferencing system providing voice or video communication channels. Hypermedia support for modes 5 and 6 will be provided by integrating the hypermedia architecture with the shared window system and the computer conferencing system being developed in the EuroCODE project.

SHARED MATERIALS IN ENGINEERING PROJECTS

The materials being shared in the bridge construction project include CAD drawings, pictures and videos of bridge elements, letters, procedure handbooks, scanned documents, spreadsheets, case records, reports, etc. One area where the use of hypermedia was considered useful was in maintaining the rich set of relationships among case records, letters, reports and work procedure handbooks. The hypermedia can support the engineers' navigation in the material and cooperation on cases, e.g. acceptance of changes to the construction process for specific bridge elements. Changes to construction processes are quite frequent, and it is important for the engineers to be notified about addenda being added to work procedure handbooks and annotations being made to drawings, reports, etc. Another area is writing of reports and collection of materials for reports, these tasks are typically organized with a responsible editor and several persons contributing and commenting.

These characteristics of the use setting indicated that hypermedia support for the asynchronous modes of cooperation needed primary attention.

STRUCTURE OF THE PAPER

The structure of the paper is as follows. Section 2 briefly introduces the Dexter model. Section 3 describes and discusses our Dexter based cooperative hypermedia architecture using an object oriented database (OODB). The OODB was augmented to support our hypermedia development and Section 4 delves into a detailed discussion of the OODB locking and event notification mechanism that has been developed to support our hypermedia. Section 5 discusses how we applied the augmented OODB and the Dexter based architecture for developing the cooperative hypermedia system DHM. Section 6 concludes the paper.

2. THE DEXTER MODEL

The *Dexter Hypertext Reference Model* [Hala90] separates a hypertext system into three layers having well-defined interfaces as shown in Figure 1.

The *Storage layer* captures the persistent, storable objects making up the hypertext. The basic object provided in the Storage layer is the *component*. Components are divided into *contents*, corresponding to the component's data, and *component information*. The component information includes a general purpose set of *attributes*, a *presentation specification* and a set of *anchors*. The *atomic component* is an abstraction replacing the widely used but weakly defined concept of 'node' in a hypertext. *Composite components* provide a hierarchical structuring mechanism. The content of a *link component* is a list of *specifiers*, each including a presentation specification as well as component and anchor identifiers. A *hypertext* is simply a set of components.

The *Within-component layer* corresponds to the data objects, the contents of components, and the individual editors to handle the data objects. The editors are responsible, e.g. for supporting content selection for link anchoring.

The interface between the storage and within-component layers is based on the notion of anchors. Anchors consist of an identifier that can be referred to by links and a value that picks out the anchored part of the material.

The *Runtime layer* is responsible for handling links, anchors, and components at runtime. Objects in the runtime layer include sessions, managing interaction with particular hypertexts, and instantiations, managing interaction with particular components. The runtime layer provides editor independent user interface facilities through operations like NewComponent, AddLinkEndpoints, and FollowLink.

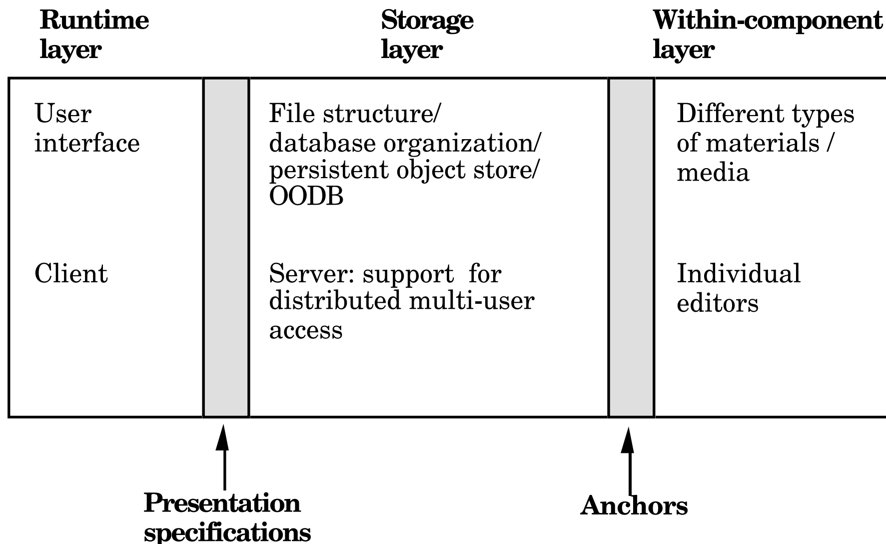

Figure 1: The Dexter model layers and interfaces.[1]

The interface between the Storage layer and the Runtime layer includes *presentation specifications* that determine how components are presented at run-time. Presentation specifications might include information on screen location and size of a presentation window, as well as a “mode” for presenting a component. Halasz & Schwartz [Hala90] use the example of an animation component that can be opened in either run mode or edit mode.

The Dexter terminology provides a solid framework for discussing the design of hypermedia systems, but the formal specification leaves certain design decisions open. For instance, how do we support sharing of hypertexts and components among several users? How do the Dexter layers relate to a multi-user distributed hypermedia architecture? Where do we place the responsibility for locking, and event notifications? The following sections discuss how we extended the Dexter model’s notion of hypermedia systems to deal with these issues.

3. A DEXTER BASED ARCHITECTURE FOR COOPERATIVE HYPERMEDIA

The DeVise hypermedia architecture is a process architecture providing several types of servers and clients that correspond to the Dexter Model layers shown in Figure 2. In the object oriented framework the processes wrap objects. Below we outline some important features of the DeVise Hypermedia architecture, capturing the full details is outside the scope of this paper.

COOPERATIVE HYPERMEDIA ARCHITECTURE

The cooperative DHM is structured in a client/server architecture as shown in Figure 2.

Figure 2 also shows how we interpret the role of the processes in relation to the Dexter model layers. The following three types of processes are present in the architecture:

---

[1]Some of the text appearing in the figure represent our own interpretation of the model.

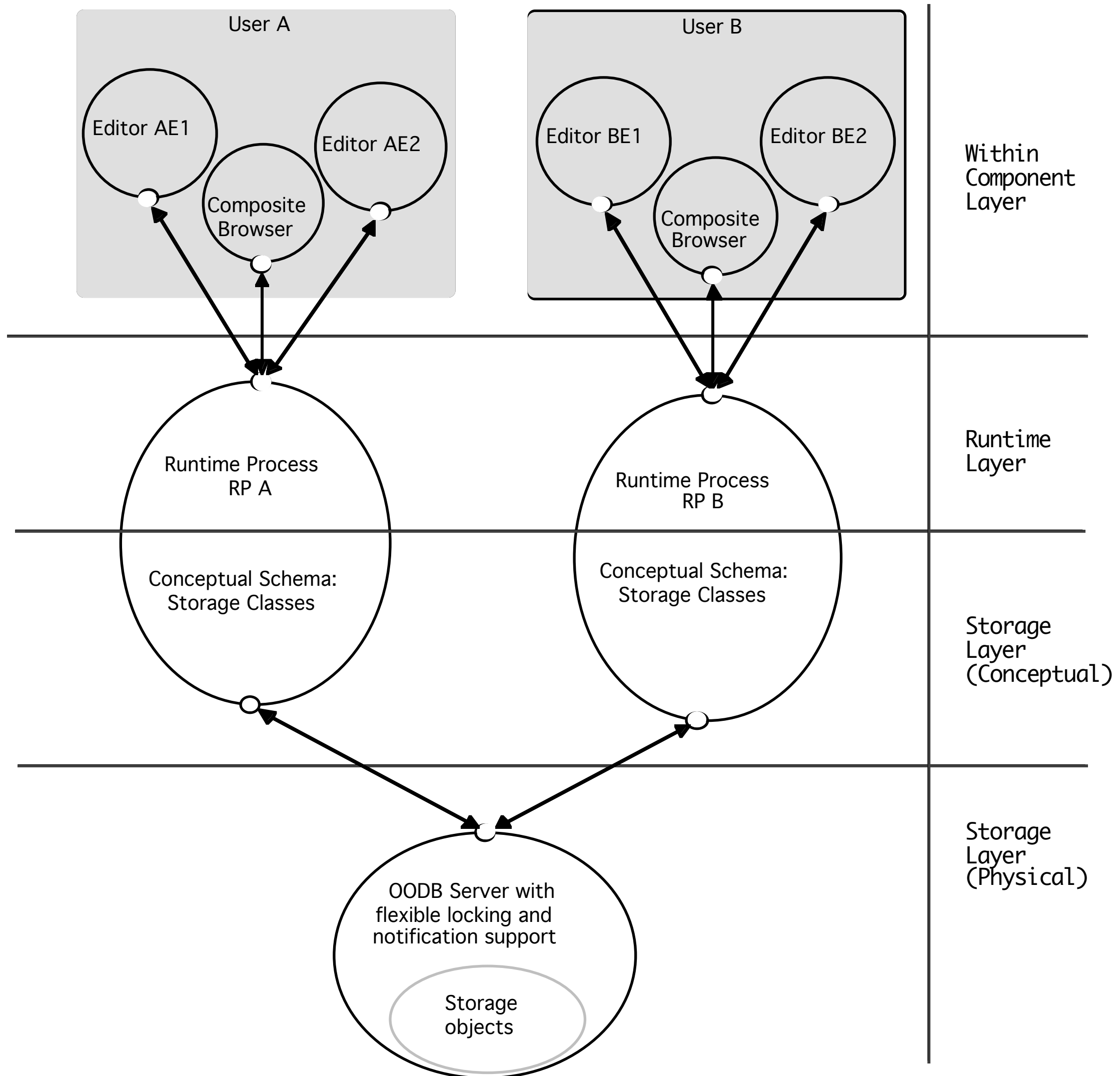

Figure 2: Multi-user hypermedia client/server architecture. The small ellipses represent protocols that the processes support.

1. Editor Process: These processes are end-user editors integrated with the hypermedia, and they may include text editors, graphical editors, video players/editors and hypermedia browsers. An editor takes care of a specific type of data objects, e.g. text objects which constitute the contents of textComponents. The data objects, corresponding to the Dexter within-component-layer objects, may be stored by the editors in separate files outside the OODB. The editors thus represent the runtime handling of the component content objects. This also holds for the Hypermedia browsers, which are implemented by means of composites, see [Grøn92b]. Thus a browser edits the within component objects of composites. Editors communicate anchor values to Storage objects (through a Runtime Process), and they interpret Presentation specifications provided by a Runtime Process.

2. Runtime Process: A Runtime Process (RP) provides the hypermedia service for a set of editor processes currently in use by a user. The RP is responsible for handling links, anchors and components at run-time. The RP is a server that communicates with the editors and it is a client of the OODB server. The RP creates instances of the objects defined by the generic and specific classes implementing the Dexter Runtime Layer concepts, and it provides editor independent operations for creating

and manipulating components and links objects implementing the Dexter Storage Layer concepts. The RPs are also responsible for distributing event notifications received from the OODB server to the editors. These facilities are described in further detail in Sections 4 and 5. The RPs serve a similar role as the Tool Integrators proposed in the HyperForm architecture [Will92] and the Link Hub proposed in the IRIS Hypermedia Services [Haan92].

3. OODB server. The OODB server process provides permanent physical storage for the hypermedia objects. The objects being stored are instances of classes specialized from the generic classes implementing the Dexter Storage Layer concepts. The Storage class structure, which is declared at the client level, becomes the conceptual schema for the hypermedia objects stored by the OODB server. In the MBS OODB, the conceptual schema is defined in the client processes, i.e. at the logical storage level. There may be several OODB servers running at the same time, and in a future version the OODB distribution facilities will make it possible to link between hypertexts stored by different OODB servers.

INTEGRATION OF TOOLS WITH THE HYPERMEDIA

Needs for an open hypermedia architecture were identified in our analysis project [Grøn93]: the engineers wanted to be able to continue using their favorite editors and have the hypermedia functionality integrated with these editors. Such demands for open hypermedia have been recognized by several authors in the field, e.g. [Davi92; Kacm91]. Sharing these concerns, we have designed an open hypermedia architecture. To integrate a new application with DHM, a component type, instantiation type, presentation specification, anchor specification and linkMarker specification corresponding to the material maintained by the application must be defined. This is done by specialization of the generic classes of the DHM kernel. In addition, the new editor must be interfaced to the protocol of the RP. Tools are classified according to the extent they may be integrated with DHM, e.g. to support local anchoring: Fully open editors, semi-open third party editors and closed third party editors. For a further discussion of integration in DHM, see [Grøn92b].

DISTRIBUTION OVER DIFFERENT PLATFORMS

The OODB server and the RPs may run on different computers in a distributed environment. There is one active RP for each active user of the hypermedia. The RP is a client of an OODB server, and it may run on a Macintosh while the OODB server runs on, e.g. a Sun Sparc station, or vice versa. The RP and the editors may in principle also run on different computers, but in practice they will usually run on the user's workstation. Distribution of editors on different platforms could support hypermedia integration of, e.g. ordinary office programs running on one workstation and a powerful CAD system running on another workstation in the same office. The distributed multi-user hypermedia architecture is depicted in Figure 2. To support cooperative design and authoring by the hypermedia, users need support to coordinate their work on the shared materials. Technically such coordination is supported through event notifications distributed by the OODB server. The OODB server is able to inform its clients about events occurring on the stored objects, and the clients of the OODB server may subscribe to various types of events. The next section describes how these facilities are supported by the OODB.

4. COOPERATION SUPPORT: OODB BASED EVENT NOTIFICATION AND FLEXIBLE LOCKING

The DHM system was developed to support the asynchronous cooperation modes (1-3) described in Section 1. This requires on the one hand support for creating

awareness among users about what happens to the material being shared; and on the other hand support for exchanging responsibility for parts of the material, i.e. exchanging locks on hypertexts, components, anchors and attributes.

EVENT NOTIFICATIONS FOR OODB CLIENTS

The idea of supporting awareness notifications was proposed by Halasz [Hala88] and an example of a system providing such support is given by Wiil [Wiil91]. The idea is that users via their editors or a browser are able to subscribe to a variety of events occurring on the shared materials. Finding the approach promising, we have developed an event notification mechanism for our OODB. The fact that it was developed directly within the OODB implies that we can support event notifications for arbitrary objects and classes independent of their declaration, i.e. event notifications are not bound to objects inheriting from a special superclass. A Runtime Process (RP) may ask the OODB server to be informed about changes to objects, which are made by other RPs associated with other users. In a given situation, several users may be accessing the same 'hypertext'(in Dexter terms). If one user makes changes to a component in the hypertext, these changes will be made visible for the other users who have opened this component with read access and subscribed to notifications about changes. Subscriptions may be made *automatically* for some eventtypes in a specific hypermedia application or they may be made *manually* by the users. This section describes the OODB notification mechanism. A notification object provides a feedback from the OODB server about an event generated by this or other clients. A client subscribes to notifications identified by an event type, a target object or class, and a user group specification. Currently it is possible to choose among users identified by user name, and all users connected to the server. The group "all users" changes dynamically as users start/commit transactions. The operation `getActiveClients` returns the usernames of the clients with started transactions. The operation can be used to get information on which users may be specified in the user-restriction, when subscribing to a notification. The event types currently supported are: `startTransactionEvent`, `commitTransactionEvent`, `abortTransactionEvent`, `getEvent`, `updateEvent`,`createEvent`, and `lockChangeEvent`.

Events generating notifications occur in the OODB server when a client performs a checkpoint or a commit operation on a transaction. Exceptions to this are `startTransactionEvent` and `abortTransactionEvent` which occur at transaction start and transaction abort, respectively. Finally, `lockChangeEvents` are distributed when calls to `changeLock` have been completed successfully. Subscriptions belong to a given transaction, and Table 2 summarizes the `Transaction` class interface; the main operations are described below.

The `SubscribeToNotification` operation subscribes to events of type `EventType` related to the object or class specified by `targetObjectOrClass` and generated by the users specified by `usersSpec`. To unsubscribe to a previously subscribed notification the `UnSubscribeToNotification` operation must be used.

Notifications are sent from the OODB server to its clients. In the case where a `Notification` is pending, the virtual operation `Notify` on the client is called: The client process has to decide how a notification should be interpreted. This is done by a further binding of the `Notify` virtual;[2] here the contents of `Notification-ref`

---

[2]The BETA programming language supports virtual procedures similar to SIMULA or C++ virtuals. A virtual procedure is common for the whole inheritance hierarchy of the enclosing class, but its attributes and action may be specialized (further

can be interpreted and used to trigger appropriate actions. Finally, the operation `DisplayNotification` is available to display event notifications textually, for instance in a console or a log file.

OBJECT ACCESS AND LOCKING

The OODB provides support for fine grained access and locking of objects. This section gives an overview of the facilities, illustrating how they can be used to support hypermedia development.

The operations described in this section belong to the interface of a transaction (see Table 1) which can either be committed, aborted or checkpointed (When checkpointing, the current status of the transaction is stored, notifications are distributed, but no locks are freed). A transaction may be of arbitrary length as called for by Halasz [Hala88]; in addition, Halasz's call for a more flexible locking protocol is supported as described below.

```
Transaction: Class
(# ...
   Start: Ø -> Ø
   Checkpoint: Ø -> Ø
   Commit: Ø -> Ø
   Abort: Ø -> Ø
   SubscribeToNotification: Subscription-ref -> Status
   UnSubscribeToNotification: Subscription-ref -> Status
   Notify virtual: Notification-ref-> Ø
   DisplayNotification: Notification-ref -> Ø
   Create: (name, Obj-Ref) -> Ø
   Get: (name, ClassDescriptor, LockSpec) -> Obj-Ref
   ReGet: (Obj-Ref, LockSpec) -> Ø
   Update: Obj-Ref -> Ø
   ChangeLock: (Obj-Ref, LockSpec) -> Status
   ...
#)
```

Table 1: An excerpt from the interface to the Transaction class.

The `Create` operation makes an object (given by a reference) into a persistent root, i.e. a persistent object with a specified name to be used when retrieving it from the OODB server with the `Get` operation. `Create` tells the database to store the object and its transitive closure, the next time the transaction is committed or checkpointed. Every object in the closure of the root object thereby becomes persistent.

A persistent root object and its transitive closure of objects are retrieved from the database by means of the `Get` operation. Locks for all objects retrieved from the OODB server during the `Get` operation are specified by the `LockSpec` parameter. Currently there are only two lock values (`write` and `read`), but the OODB is open for adding other lock values, e.g. those described in [Ahme91].[3] Note that retrieving the transitive closure of an object is a logical operation. The physical retrieval is implemented by an incremental retrieval algorithm ensuring that only the objects

---

bound) at each level in the hierarchy. Many virtuals in the hypermedia system are called by the system, giving programmers hooks to have their own code called automatically in specialized classes.

[3] We use ‘read’ lock here in the relaxed notion that multiple readers have read access to objects retrieved with a read lock. Future version may also support an ‘exclusive read’ lock allowing only one reader at the time.

actually being accessed are read into memory. It is also possible to retrieve arbitrary objects (with their transitive closure). This is useful when a client is notified about a change to an object and needs to retrieve the new version of the object. Using the `ReGet` operation, all objects in the transitive closure from Obj-Ref are re-read from the database. The parameter `LockSpec` may specify a lock other than the current one for Obj-ref. If the current lock is `read` and the `LockSpec` is `write` it may not be possible to obtain write permission. In this case an exception is raised allowing an application to start a dialog asking the user what to do.

If an object, fetched from the database using the `Get` operation, is changed, and those changes are to be stored in the database, the `Update` operation is invoked. `Update` operates on some persistent object (including its closure). Invoking the update operation tells the database that every change made to these objects during the transaction should be stored persistently, the next time the transaction is either checkpointed or committed. The ability to specify with such fine granularity exactly which objects must be stored is important, because this specification is used directly in the distribution of notifications on update events. In case an update operation is invoked on an object which is only 'read-locked' an exception is raised.

The lock for an object that has been retrieved from the database may be changed dynamically by the `ChangeLock` operation. The `ChangeLock` operation changes the locks for all objects in the transitive closure of `Obj-Ref`. Changing the lock to one with higher permissions than the current lock implies an implicit `ReGet`[4] of the objects to be locked. Changing the lock to one with lower permissions, implies an implicit checkpoint, since the objects may have been changed. If a write lock is abandoned, another client may obtain a write lock, change the objects, and commit these changes to the OODB.

A GENERAL OODB VERSUS A DEDICATED HYPERBASE

The use of a general OODB distinguishes our hypermedia architecture from the Ålborg HyperBase [Wiil91] and the HyperForm [Will92]. The OODB we use is being developed in parallel with our hypermedia architecture and it is designed to meet our hypermedia requirements, but it is a general OODB in the sense that it supports locking and notifications for all types of objects independent of whether they are hypermedia objects or say CAD objects. This implies that we do *not* have to predict which objects or classes of objects we would like to support locking and event notifications for. Any application can at any stage be tailored to subscribe to notifications on events on some object or class that is used as part of some other types of objects. Said in other words: locking and event notification are completely independent of the *declaration* of the objects stored in the OODB. Since notification and locking is the responsibility of the OODB, Storage classes need *not* be extended to support this.

In the Ålborg HyperBase [Wiil91] notifications are tied to the specific data model that the HyperBase supports, hence it may be a major change to start getting notifications on other types of objects at a different level of detail than that captured in the data model. In the HyperForm approach [Will92], the lock and notification handling is supported in generic classes such as Concurrency Control (CC) and Notification Control (NC). When implementing a specific data model, classes that support notification and locking need to inherit from the NC and the CC classes, respectively. When using the general OODB approach, locking and notification handling are meta properties that need not be decided when designing the data

---

[4]The user may be asked for confirmation before the `ReGet` is performed.

model. This implies that a system can be developed to use a large existing database and to subscribe to notifications on events not anticipated when the conceptual schema for the stored objects was designed.

5. UTILIZING THE AUGMENTED OODB TO BUILD COOPERATIVE HYPERMEDIA

In Section 1, a number of modes of cooperation on shared materials were described, and it was pointed out that awareness of other users' activities on the shared materials should be supported directly in the cooperative hypermedia system. This section describes how we used the OODB facilities described in the previous section to develop the DHM cooperative hypermedia system.

EXTENDING RUNTIME OBJECTS TO HANDLE LOCKING AND NOTIFICATION

Event notifications are interpreted by the RPs which again propagate the notifications to their clients (the editors). The Runtime classes, session and instantiation, have (compared to the similar Dexter model concepts) been extended with operations to handle and propagate event notifications received from the OODB server. The notifications are typically displayed in the applications and/or the hypermedia composite browser. The applications may also provide a user interface to subscribe to event notifications that the user is interested in, or the application may *automatically* subscribe to and handle certain kinds of event notifications. This section describes how the Dexter Runtime classes were extended to treat locking and event notification.

The Dexter concepts of Session, Instantiation and LinkMarker were transformed into classes with operations corresponding to the Dexter model functions manipulating these objects. Since the programming language used [Mads93] supports block structured nesting of classes, our Runtime classes are encapsulated in a sessionMgr class and the RP consists of an object instantiated from that class. Table 2 summarizes the operations included in the Runtime classes to handle locking and notification.

For the lock handling we have defined a `changeLock` operation on the session and instantiation classes making it possible to change the lock for an entire hypertext or a single component, respectively. However, the point is that further `changeLock` operation can easily be added offering the possibility to change locks on objects of a hypertext at an arbitrary level of granularity, e.g. `linkMarkers` may be extended to support change of locks on individual anchors.

The notification handling is designed similarly. The sessionMgr, session, and instantiation classes are extended with a `subscribe`, an `unSubscribe`, and except for the `sessionMgr`, a `ReGet` operation.

The `ReGet` operations are introduced to enable retrieval of a new version of a Storage object, e.g. a hypertext or a component from the OODB server. `ReGet` on a session object retrieves the newest version of all persistent objects encapsulated in that session. The need to do a ReGet typically occurs when the RP receives an update notification telling that the hypertext of the session has been changed. Reget may also be called automatically by Runtime objects as a reaction to an event notification.

In order to perform a subscribe operation, a `subscription` object consisting of a target, an event type, and a user restriction, need to be specified. The user restriction specifies users from whom event notifications are wanted, the target is either a specific object or a class reference specifying that all objects of that class are

of interest, and the event type specifies the relevant type of operations to be informed about (i.e. update events).

```
SessionMgr: Class
(#...
   subscribe: Subscription-ref -> Ø
   unSubscribe: Subscription-ref -> Ø
   session: Class
   (#...
      subscribe: Subscription-ref -> Ø
      unSubscribe: Subscription-ref -> Ø
      reget: Ø -> status
      changeLock: lock-spec -> Status
      instantiation: Class
      (#...
            subscribe: Subscription-ref -> Ø
            unSubscribe: Subscription-ref -> Ø
            reget: Ø -> status
            changeLock: lock-spec -> Status
            linkMarker: Class(#....#)
            (* instantiation private *)
            instantiationReaction: sessionReaction(# ...#)
            updateR: instantiationReaction(#...#)
            getR: instantiationReaction(#...#)
            lockChangeR: instantiationReaction(#...#)
            createR: instantiationReaction(#...#)
      #)
      compositeInstantiation: instantiation(#...#)
      (* session private *)
      sessionReaction: Reaction(# ...#)
      updateR: sessionReaction(#...#)
      getR: sessionReaction(#...#)
      lockChangeR: sessionReaction(#...#)
      createR: sessionReaction(#...#)
   #)
   (* sessionMgr private *)
   Reaction: Class(#...#)
   StartR: Reaction(# ...#)
   CommitR: Reaction(# ...#)
   AbortR: Reaction(#...#)
#)
```

Table 2: The new operations introduced for notification handling.

In addition, to requesting notifications from the OODB server, the subscribe operation also register an object pair: the `subscription` object and a `reaction` object, in a table maintained at the sessionMgr level. A reaction is an object instantiated from one of the classes shown in the "private parts" of Table 2. The reaction object is executed when the RP receives a notification matching the corresponding subscription object in the (subscription, reaction) - pair. The table always contains such a pair for each pending subscription made by the RP. The reaction stored depends on the event type of the subscription object and each Runtime class contains a reaction class for each event type of relevance at that level. For instance, the sessionMgr level has three reaction classes `AbortR`, `CommitR` and `StartR` that are instantiated in case a subscription is made to either abort, commit or start events. On the instantiation level the reactions `CreateR`, `UpdateR`, `GetR` and `LockChangeR` exist in order to handle create, update, get, or change-lock events on a particular instantiation.

As seen in Table 2, the reactions are organized in an inheritance hierarchy reflecting the block structured nesting of Runtime classes. Each of the main

Runtime classes has an abstract reaction superclass collecting the similarities of reactions for that class. The detail level of the reactions increases with the level of nesting of the Runtime classes; hence having reactions for a nested class being a specialization of the abstract super reaction of the enclosing class is an efficient solution. In addition, this specialization hierarchy allows a reaction to be handled at all intermediate levels, e.g. if a component receives a notification after its instantiation has been closed, then the enclosing session will handle the notification, and ultimately the sessionMgr will handle it.

EXAMPLES OF DHM NOTIFICATION AND LOCKING SUPPORT

The following user interface examples, are from the UNIX/X-windows prototype version of DHM. The example data is a `hypertext` developed together with the engineers from the bridge construction project described in [Grøn93]. This particular hypertext covers materials for two cases that were selected for experimentation on organizing the engineering materials in hypermedia structures.

Runtime Processes (see Section 3, Figure 2) stores and retrieves hypertexts as persistent roots via the OODB server. Hypertexts, components and anchors in DHM possess attributes with information about, e.g. who was the creator and who was the last modifier. There are also attributes indicating whether the Storage objects are public, belong to a group, or belong to a specific user. These attributes allow a session for a hypertext to selectively present only the objects that the current user would like to use or has the rights to use. The basic event notification mechanism described in the previous section makes it possible to keep track of higher-level events on shared hypertexts such as:

- Creating, deleting and updating hypertexts
- Creating, deleting, updating components (atomic, link or composite) in a hypertext
- Creating, deleting, updating anchors and attributes in a component.
- Lock changes on hypertexts and components.

It is also possible to subscribe to notifications about start of transactions as well as commit and abort of active transactions by other RPs. Notification on such events makes it possible to support users' awareness of both changes to status and contents of shared hypertexts.

Notifications being passed through the Runtime objects will appear in the user interface in a fashion chosen by the user, e.g. graphical indication and sound, and it can be inspected when and how other users access the same hypertext as the current user. In Figure 3 a ‘RW’ mark on the icon for the 'GB' hypertext indicates that this user possesses a read lock on the hypertext, but another user has obtained a write lock on it. Similar marks indicate, e.g. when no one has a write lock on the hypertext and when there are no other users accessing the hypertext. See [Grøn92a] for more details. Users can inspect a notification log as well as attributes on hypertexts and components to get informed about modifications to objects.

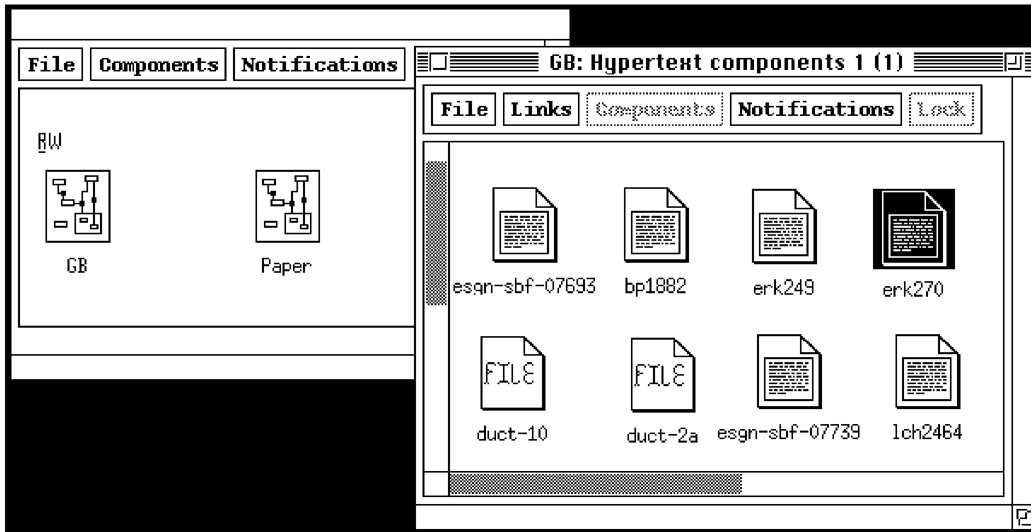

Figure 3: A snapshot of the browser interfaces to hypertexts and components. The browser window to the left displays an icon for each open hypertext and it provides an interface to hypertext level notifications. The small mark on top of the 'GB' hypertext icon indicates that this user has a read lock and another user has a write lock. The Component browser to the right has an icon for each component and it has an interface to component level notifications.

A user who has obtained a write lock on a (part of a) hypertext may during a transaction modify the hypertext. To keep track of such changes, it is possible for other users through their RP to subscribe to notifications about object retrieval, creation, update and access changes caused by other users of the hypertext. As mentioned objects are retrieved with either a read lock or a write lock, and such lock information is also part of the event notification.

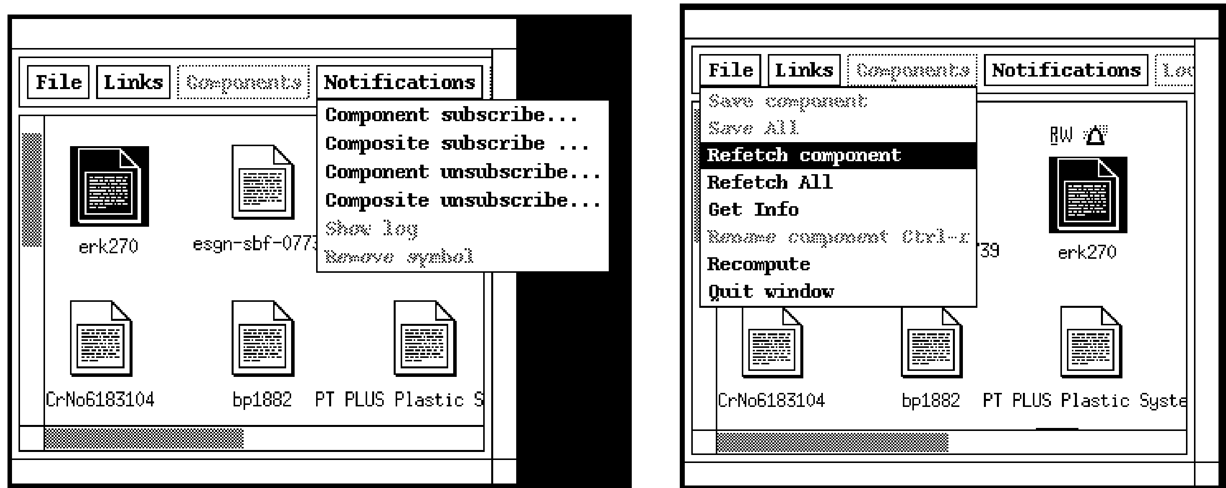

Figure 4: The user interface for subscribing to notifications and refetching on components, represented in a Browser window. Selecting the 'Component subscribe...' item in the menu to the left brings up the dialog shown in Figure 5. In the window to the right the 'erk270' component is marked with a bell indicating that an update notification was received. To examine the changes the user performs a 'Refetch component' operation on the erk270' component.

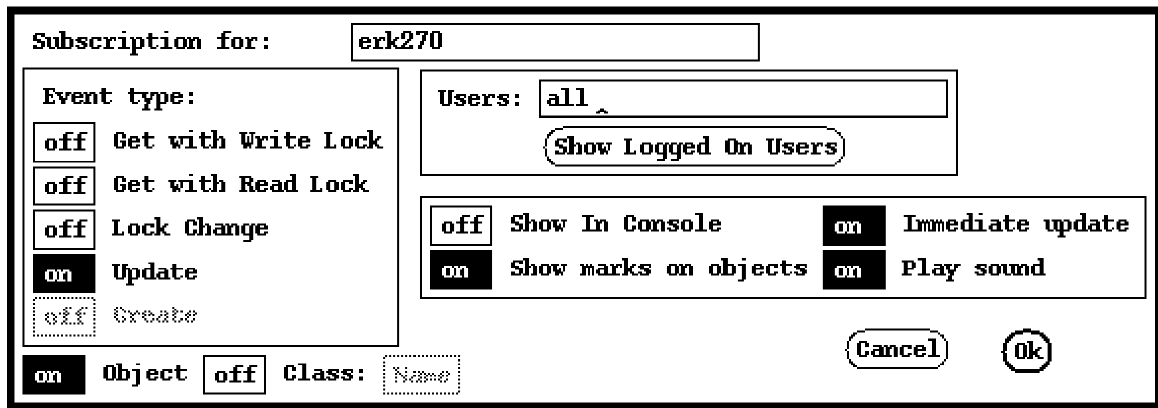

Figure 5: The dialog to subscribe to notifications at Instantiation/Component level. Event type, user restriction and the kind of reaction wanted is specified.

Subscriptions may be made for specific objects or entire classes of objects. Figures 4-5 show how component level notification subscription and reception appear to the user. The subscriptions are made through a component browser or a specific editor. The user interaction shown illustrates the situation where the user has chosen to receive a notification and then perform a Refetch. If the 'Immediate Update' option in Figure 5 is checked, the Refetch takes place automatically.

SCENARIOS FOR COOPERATIVE HYPERMEDIA USAGE

The previous section showed an example of how the notification and locking mechanisms appear in the prototype user interface. To give a more comprehensive description of the kind of cooperation support DHM provides, this section contains a set of abstract use scenarios. The scenarios illustrate typical use of the support developed for cooperation modes 1–3 discussed in Section 1. The scenarios illustrate interactions with the DHM system that were abstracted from work situations analyzed in the engineering project described in [Grøn93]; they are formulated here in Dexter and OODB terms.

Scenario 1: Immediate update:

Peter and Susan both start a session on hypertext H1. Peter obtains a write lock on component C1. Susan opens C1 with read lock and subscribes to immediate updates when C1 is changed by other users. The instantiation for C1 now automatically updates the instantiation by regetting the most recent version of C1 stored in the OODB whenever an update event notification appears. Peter makes changes and commits them to the database, forcing immediate updates to happen on Susan's screen.

Scenario 2: Logging events:

Several users (Peter, Susan, and John) have started a session on hypertext H1. Susan has opened C1 with a read lock and subscribed to logging of changes to C1. Peter opens C1 with a write lock - Susan is notified with a message in the console saying: "Peter opened C1 with write lock Thursday 26.11.92 at 11:28:08". Peter makes changes and saves C1 - Susan is notified with a message in the console saying: "Peter modified C1 Thursday 26.11.92 at 12:00:11". Peter releases the write lock - Susan is notified: "Peter released the write lock for C1 Thursday 26.11.92 at 12:01:00". Later John opens C1 with a write lock - Susan is notified: "John opened C1 with write lock Thursday 26.11.92 at 13:10:08"

Scenario 3: Awareness notification for hypertexts:

Peter and Susan both start a session on hypertext H1, subscribing to notifications about who uses H1. The result is a console showing a list of users having performed a 'Get' operation on H1. Now John also starts a session on H1, making an identification of John appear in Peters and Susan's consoles.

Scenario 4: Awareness notification for components and composite components:

Several users (Peter, Susan, and John) are working on the same “case” for which Susan is responsible. They have started a session on the corresponding hypertext H1. Susan makes a Composite CS1 containing components C1, C2, C3, and C4, corresponding to the currently active documents in the “case”. Susan uses the ‘Composite subscribe...’ menu command to subscribe to notifications on changes occurring to components contained in CS1. The result is that Susan is notified whenever another user performs update, lock change, etc. on C1, C2, C3, and C4.

Scenario 5: Notification about creation/deletion of specific types of objects:

Several users (Peter, Susan, and John) start sessions on hypertext H1. Susan subscribes to logging of textComponent creation in H1. Peter creates a new textComponent C1 for H1, edits the contents and saves it - Susan is notified with a message in the console saying: “Peter created textComponent C1 Thursday 26.11.92 at 11:28:08”.

Scenario 6: Lock exchange:

Peter, Susan and John start a session on hypertext H1. Peter obtains a write lock on component C1. Susan and John open C1 with read lock. John subscribes to logging of changes to C1. At some point Susan uses the menu command "Change lock...", which informs her that Peter has a write lock on C1. Then Susan calls Peter on the phone and asks him whether he is willing to save his changes and release the write lock on C1. Peter agrees to do that, saves his changes, and changes the write lock to a read lock. Susan immediately obtains a write lock. During this exchange John has received notification messages that: Peter has saved changes, Peter has released write lock on C1, Susan has obtained write lock on C1. Peter subscribes to logging of all changes to C1. Susan then makes some changes and commits them to the OODB triggering notification messages to both Peter and John.

*Scenario 7: S*imultaneous linking:

Peter and Susan starts a session on hypertext H1, and opens the textComponent C1 with read lock and subscribes to immediate update on C1. Peter creates a public link from a text region in C1 to a text region in component C2. Peter commits the change making a short upgrade to a write lock on the anchor list of C1, immediately updating Susan's view of C1 with the new linkMarker. Susan makes another public link from C1 to component C3, commits the changes, Peter's view of C1 is immediately updated in a similar fashion.

The scenarios described in this section illustrate examples of the kind of support for cooperation on shared hypertexts that is provided currently with the augmented OODB and the Runtime class extensions described earlier. Experiences from the engineering project and other upcoming use settings are contributing to ongoing development of support for a richer set of cooperation scenarios. Among the future developments we also expect to support scenarios where users gracefully move from asynchronous modes of cooperation to synchronous modes still inheriting the the general Dexter based hypermedia features.

6. CONCLUSION

The paper discussed issues for the design of a general architecture for cooperative hypermedia systems based on the Dexter Hypertext Reference model [Hala90]. The architecture provides a generic framework for developing Dexter compliant hypermedia systems. The architecture consists of an extended object oriented implementation of the generic concepts proposed by the Dexter model. The client and server processes of the architecture are designed to correspond to the layering proposed by the Dexter Model. The architecture includes an object oriented database (OODB) to store the objects implementing the Dexter Storage Layer concepts. The OODB has, in course of the project, been augmented to support long term transactions, flexible locking and notifications as called for by Halasz [Hala88]. Developing such support within the OODB makes it general and independent of changes and extensions to the Dexter based process and data models. Our working prototype, DeVise Hypermedia (DHM), utilizes the power of this architecture. DHM was developed and used to explore the possibilities of providing hypermedia support for engineering projects. Inspired from these experiments a set of abstracted use scenarios is described to illustrate examples of the kind of cooperation support that can be provided by systems developed from the Dexter-based hypermedia architecture.

The Dexter-based architecture constitutes the basis for further hypermedia development in the EC funded Esprit III project, EuroCODE - CSCW open development environment (1992-1995). This involves further development of the Dexter-based architecture, development of a tailoring environment, and implementation of specific hypermedia prototype systems for the primary user organizations involved in the project.

ACKNOWLEDGEMENTS

We greatly thank Randy Trigg for his invaluable inspiration and for his comments on earlier drafts of this paper. We also thank Søren Brandt and Kim J. Møller for their work on the OODB, Jørgen Nørgård for his work on browsers, and the rest of our group: Niels Damgaard, Jørgen L. Knudsen, Morten Kyng, Preben Mogensen, and Elmer S. Sandvad for their contributions to the design of DHM. The work is supported by the Danish Research Programme for Informatics, grant number 5.26.18.19, and the Esprit projects EuroCoOp and EuroCODE.

REFERENCES

[Ahme91] Ahmed, S., Wong, A., Sriram, D., & Logcher, R. (1991). *A Comparison of Object-Oriented Database Management Systems for Engineering Applications* (Research Report No. R91-12).

[Ande92] Andersen, P., Brandt, S., Hem, J. A., Madsen, O. L., Møller, K. J., & Sloth, L. (1992). *Workpackage WP5 Task T5.4, Deliverable D5.4: Distributed Object- Oriented Database Interface.* (EuroCoOp deliverable No. ECO-JT-92-3). Jutland Telephone and Aarhus University.

[Davi92] Davis, H., Hall, W., Heath, I., Hill, G., & Wilkins, R. (1992). Towards an Integrated Information Environment with Open Hypermedia Systems. In *European Conference on Hypertext (ECHT '92)*, (pp. pp. 181-190). Milano, Italy: ACM.

[Grøn93] Grønbæk, K., Kyng, M., & Mogensen, P. (1993). CSCW Challenges: Cooperative Design in Engineering Projects. *Communications of the ACM, 36*(6), 67-77.

[Grøn92a] Grønbæk, K., Madsen, O. L., Møller, K. J., Nørgaard, J., & Sandvad, E. (1992). *EuroCoOp Workpackage WP5 Task T5.3 Distributed Hypermedia Design Tool.* (EuroCoOp Deliverable No. ECO-AU-92-14). Aarhus University.

[Grøn92b] Grønbæk, K., & Trigg, R. H. (1992). Design issues for a Dexter-based hypermedia system. In *European Conference on Hypertext 1992 (ECHT 92)*, (pp. 191 - 200). Milano, Italy.: ACM, New York.

[Haan92] Haan, B. J., Kahn, P., Riley, V. A., Coombs, J. H., & Meyrowitz, N. K. (1992). IRIS Hypermedia Services. *Communications of the ACM, 35*(1), 36-51.

[Hala90] Halasz, F., & Schwartz, M. (1990). The Dexter Hypertext Reference Model. In *Hypertext Standardization Workshop*, (pp. 95-133). Gaithersburg, Md.:

[Hala88] Halasz, F. G. (1988). Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM, 31*(7), 836 -852.

[Hem91] Hem, J. A., Madsen, O. L., Møller, K. J., Nørgaard, C., & Sloth, L. (1991). *Workpackage WP5 Task T5.2, Deliverable D5.2: Object-Oriented Database Interface*. (EuroCoOp Deliverable No. ECO-JT-91-2). Jutland Telephone and Aarhus University,.

[Kacm91] Kacmar, C. J., & Leggett, J. J. (1991). PROXHY: A Process-Oriented Extensible Hypertext Architecture. *ACM Transactions on Information Systems, 9*(4), 399-419.

[Knud93] Knudsen, J. L., Löfgren, M., Madsen, O. L., & Magnusson, B. (1993 (forthcoming)). *Object-Oriented Software Development Environments - The Mjølner Approach*. Englewood Cliffs, NJ: Prentice Hall.

[Mads93] Madsen, O. L., Møller-Pedersen, B., & Nygaard, K. (1993). *Object-Oriented Programming in the Beta Programming Language*. Reading, MA: Addison-Wesley.

[Magn93] Magnusson, B., Asklund, U., & Minör, S. (1993). *Fine-Grained Version Control for Cooperative Software Development* (Research Report No. LU-CS-TR:93-112). Lund University, Department of Computer Science.

[Sørg87] Sørgaard, P. (1987). A cooperative work perspective on use and development of computer artifacts. In *10th Information Systems Research Seminar in Scandinavia (IRIS)*. Vaskivesi, Finland, August 10-12,1987:

[Stre92] Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schütt, H., & Thüring, M. (1992). SEPIA a Cooperative Hypermedia Authoring Environment. In *European Conference on Hypertext (ECHT '92)*, (pp. 11-22). Milano, Italy: ACM.

[Wiil91] Wiil, U. K. (1991). Using events as Support for Data Sharing In Collaborative Work. In K. &. S. Gorlin C. (Ed.), *Proceedings of the International workshop on CSCW*. Berlin: Institut für Informatik und Rechentechnik.

[Will92] Wiil, U. K., & Leggett, J. J. (1992). Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems. In *European Conference on Hypertext (ECHT '92)*, (pp. 251-261). Milano, Italy: ACM.