

Hyperform: An Extensible Hyperbase Management System

Uffe Kock Wiil
John J. Leggett

Hypermedia Research Lab

JULY 1992
TAMU-HRL 92-003

Table of Contents

1. INTRODUCTION	1
1.1 The HAM Generation	2
1.2 Extensibility	3
1.3 The Hyperform Approach	4
2. HYPERFORM	5
2.1 Tool Integrator	6
2.2 Hyperform Server	7
2.2.1 Interfaces	8
2.2.2 Object-Oriented Database Extensions	8
2.2.2.1 Object Concept	9
2.2.2.2 Object Persistency, Identity and Encapsulation	9
2.2.2.3 Object Specialization	10
2.2.2.4 Message Passing	11
2.2.2.5 Class Evolution	11
2.2.2.6 Database Mechanisms	12
2.2.2.7 Basic Mechanisms	13
2.2.3 Object Subclasses	13
2.2.3.1 Concurrency Control Object	14
2.2.3.2 Notification Control Object	15
2.2.3.3 Access Control Object	16
2.2.3.4 Version Control Object	17
2.2.3.5 Query and Search Object	19
3. USING HYPERFORM	21
3.1 Hyperform as HyperBase Development Tool	21
3.1.1 Overview of Simulating the HAM	21
3.1.2 Simulating the Danish HyperBase	22
3.1.2.1 Developing the HyperBase Data Model	23
3.1.2.2 Initial Performance Test	26
3.1.2.3 Porting EHTS Editor to Hyperform	27
3.2 Other Applications of Hyperform	28
3.2.1 What is Hyperform?	28
3.2.2 Scaling Up the Hyperform Architecture	29

Table of Contents

4. RELATED WORK	32
5. SUMMARY	34
6. STATUS	35
Acknowledgments	35
7. REFERENCES	36
APPENDIX A. HYPERFORM SPECIFICATION	41
A.1 Meta Class	41
A.2 System Object	42
A.3 Object	44
A.4 Concurrency Control Object	45
A.5 Notification Control Object	46
A.6 Access Control Object	46
A.7 Version Control Object	47
A.8 Query and Search Object	48
APPENDIX B. HYPERFORM BASICS	50
B.1 Running the Hyperform Architecture	50
B.2 Internal Lists and Variables	51
B.3 Source Code	52
APPENDIX C. CLASS DESCRIPTIONS FROM HYPERBASE SIMULATION	53
C.1 Entity Class Description	53
C.2 Node Class Description	56
C.3 Link Class Description	59

Hyperform: An Extensible Hyperbase Management System

UFFE KOCK WIL
JOHN J. LEGGETT

1. INTRODUCTION

Most hypertext systems are divided into three layers [11]: a storage layer providing persistence to the system, an application layer providing the functionality of the system and a presentation layer enabling users to interact with the system. There seems to be a growing consensus in the field on separating the storage layer from the application and presentation layers by creating separate hyperbase (hypertext database) servers [8,15,35,36,37,38,47,55]. Several reasons exist for this separation. First, it is necessary if one wants to support fully distributed access and controlled data sharing. Second, it is more economical to develop hypertext systems if programmers can attend to presentation and application issues rather than the intricacies and complexity of the storage layer. Finally, it makes it possible to present hypertext on a wide spectrum of machines, some with advanced interfaces and some with primitive interfaces.

The hypertext community has not been able to agree on a single storage model for hypertext. The Dexter model [18] and other formal approaches by Garg [14], Tompa [43], Lange [25] and Afrati and Koutras [1] are examples of (mostly) paper data models that have not succeeded in becoming the standard hypertext storage model. This has led to the situation in which each hypertext system is based on its own data model instead of a commonly agreed upon model. Despite, or rather because of, the diversity of hypertext data models many of the fundamental questions are still open issues, such as: should links be separate objects, processes or attributes to the node? Should anchors be separate objects, processes or attributes to either the node or link? What is the right way to do composites? What is the right versioning model for hypertext? The need for calculated attributes and links has also been put forward [17], but no existing hyperbase supports this in a coherent way.

The hyperbase support in most hypertext systems is geared toward the specific needs of the application and presentation layers of the system and usually provides a fixed data model. Even the general-purpose hyperbases that have been developed such as HAM [8], the German HyperBase [37] and the Danish HyperBase [47] have a fixed data model. Throughout the paper we will refer to these hyperbases as the HAM generation, since HAM was the first general-purpose hyperbase. The term hypertext will be used to cover both hypertext and hypermedia. The term data model includes both the mathematical abstraction of data and operations on data.

1.1 The HAM Generation

It is our experience that the HAM generation of hyperbases does not support the work habits of many researchers and system developers. Systems are often developed using an experimental approach in which developers start building the kernel of the system and then add on more and more of the intended features in a number of design-experiment iterations, learning by their experiences. This is generally known as prototyping. The HAM generation is simply not designed to deal with requests from an evolving application for fundamental changes to the data model of the hyperbase. Relying on fixed services in the hyperbase can force developers to make some undesirable compromises in the design of the application and presentation layers; this was a major lesson learned from the development of EHTS [50] – a collaborative hypertext system based on the Danish HyperBase. The developers of KMS also believe the data model has great influence on the design of the user interface [2].

Since the HAM generation is characterized by providing fixed hyperbase support, developers have to deal with the question: "How do I make the best use of the services provided in the hyperbase?" This is true even if they use a traditional approach and design the whole system before doing the implementation. In other words, the developer has to think in terms of the provided hyperbase support when designing the other layers of the system. This paper presents an approach to hyperbases based on the concept of extensibility. By having extension facilities in the hyperbase, developers can avoid making undesirable design trade-offs due to fixed hyperbase support and turn the above question into "Which services would I like the hyperbase to provide."

1.2 Extensibility

Extensibility is not a new concept. A number of well-known extensible systems already exist. An example from operating systems is Unix which can be extended using a shell script language. An example from text editing is the GNU Emacs editor [41] which can be extended using the provided extension language GNU Emacs Lisp. Both systems are based on a kernel that provides the basic functionality of the system. An extension language interpreter is layered on top of the kernel to provide the extension facilities. User defined commands look just like the existing functions of the system. Examples of hypertext systems providing extension facilities are NoteCards [16] and KMS [2]. NoteCards provides a programmer's interface based on Interlisp and KMS provides an internal action language in which new programs can be created.

To our knowledge, no one has introduced the concept of extensibility at the storage level in hypertext systems. The presented hypertext platform, Hyperform, is implemented as a server which can be accessed over a network. Normally one can communicate with a server using a fixed set of commands. Hyperform is a server in which one can extend and modify the set of server commands. Examples of servers using fixed sets of commands are the MIT X Window System and the whole HAM generation of hyperbases. A well-known example of an extensible server is Sun's NeWS.

The HAM generation of hyperbases can only be extended by combining the existing hyperbase functions at the application level. These external extensions have a serious drawback – they require extra network communication. The Hyperform architecture provides both external and internal extension facilities, but extensibility alone is not enough to ensure usability of a system. Users want tools and basic building blocks so they do not have to start from scratch the first time they use the system. Both Unix and Emacs provide a number of basic building blocks and tools on top of their kernels. These facilities, combined with the extensible and tailorable nature of the systems, are the major reason these systems are so popular. Also, developers often share their extensions with other users via ftp sites, newsgroups and mailing lists; this helps new users and allows the basic features of the systems to evolve over time.

1.3 The Hyperform Approach

Hyperform implements basic hyperbase services that can be tailored to provide specialized hyperbase support. The system is based on an internal computational engine that provides an object-oriented extension language in which new data model objects and operations can be added at run-time. These capabilities provide design autonomy to the hyperbase designer. Hyperform provides a small class hierarchy (a library of reusable object-oriented software routines) which can be specialized using multiple inheritance to support many different hyperbase configurations. The built-in classes provide basic services such as concurrency control, notification control (events), access control, version control and search and query. Since there are no restrictions on the number of classes and objects, it is possible to have more than one data model and hyperbase configuration running in Hyperform at the same time. This allows heterogeneous hyperbases to coexist concurrently and makes Hyperform the first multi-hyperbase system.

Hyperform can be used as a research engine for future hypertext applications since it provides flexible support for data storage and data sharing. The object-oriented extension facilities ease experimentation with new data models and new hyperbase configurations. Hyperform is designed to deal with changes and be extended and tailored to match specific requirements of hypertext applications. Hyperform provides hyperbase support beyond that found in the HAM generation. It also provides a platform for addressing the major issues of the next generation of hypertext systems through a rapid prototyping approach.

In Section 2 of this paper we give an overview of the architecture of Hyperform and the services of the built-in classes. Section 3 describes the use of Hyperform in simulating the HAM and the Danish HyperBase and Section 4 compares Hyperform to the HAM generation of hyperbases and other related work. Sections 5 and 6 give a summary and status of the project.

2. HYPERFORM

Hyperform is a centralized hyperbase server implemented around the Elk (Extension Language Kit) Interpreter [28]. Elk is based on a Scheme dialect compatible (to a very high degree) with the Scheme standard [33]. In addition, Elk contains many features that make it suitable as the basis for Hyperform. Since Elk supports dynamic loading of object files, it is possible to extend Elk with features written in other languages. Elk can be compiled to run on a wide range of Unix architectures and Hyperform is currently running on Sun Sparcstations. The performance critical parts of Hyperform are written in an efficient language (C), while the rest of the system has the flexibility of an interpreted language (Scheme). Figure 1 shows a diagram of the Hyperform server.

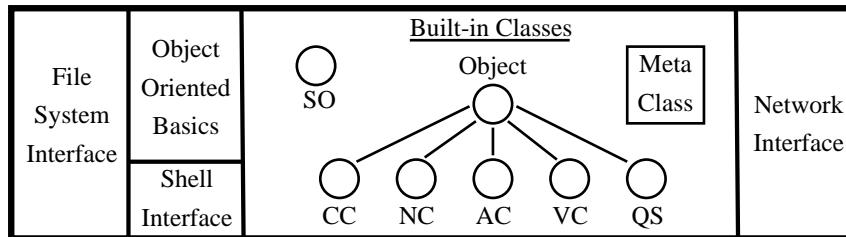


Figure 1. A diagram of the Hyperform server showing the inheritance hierarchy (class lattice) of the built-in classes: Meta Class, System Object (SO), Object, Concurrency Control (CC), Notification Control (NC), Access Control (AC), Version Control (VC) and Query and Search (QS).

The heart of Hyperform is an object-oriented data modeling facility implemented in Scheme as an extension to the Scheme language. Basic object-oriented database features such as object persistency, object identity, attribute and procedure encapsulation, object specialization by multiple inheritance, class (type) evolution by class versioning and method invocation by message passing are supported. Hyperform treats all instances and classes as objects and is self-contained since all class descriptions are saved as objects in the database. Hyperform is based on the extended client-server architecture presented in [50] which supports the development of dynamic, open and distributed systems (see Figure 2).

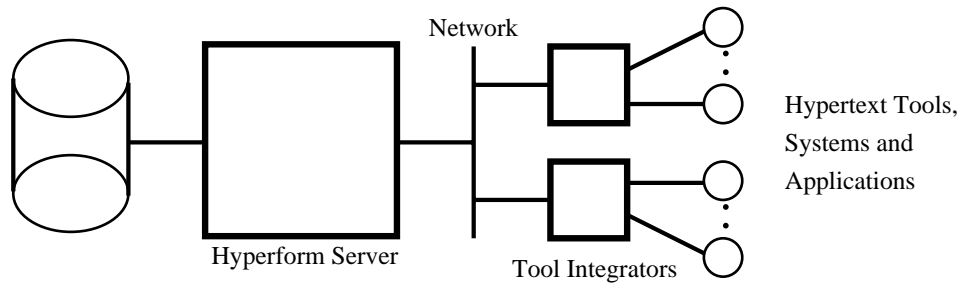


Figure 2. The Hyperform extended client-server architecture. The tool integrator is an extensible and tailorable interface between the Hyperform server and participating hypertext tools, systems and applications.

In the next two sections we concentrate on the extensibility and tailorability features of the Hyperform architecture. Section 2.1 will describe the facilities of tool integrators while Section 2.2 will describe the services of the built-in classes of the Hyperform server and explain how they can be used to address most of the major issues of future hypertext systems.

2.1 Tool Integrator

The tool integrator (TI) is an extensible, tailorable interface between the Hyperform server and participating hypertext tools, systems and applications. The TI is also based on Elk, giving it extension features similar to those of the Hyperform server. Since Elk can be extended on the fly, the TI supports dynamic integration of new tools into the environment making the Hyperform architecture dynamic, open and distributed.

The TI supports integration of both external and internal tools. External tools can be integrated via the shell and socket interfaces or tools can run internally by loading them into Elk via the dynamic object file loading facility. The Scheme extension language is used to facilitate communication among different tools and between tools and Hyperform. It might be difficult to integrate independently written applications [53] since they have been developed as stand-alone programs. Although, non-interactive applications can be invoked via the shell interface and interactive applications can be extended to communicate with the TI via the socket interface. It would be faster to implement some hypertext systems (e.g., those with only one tool) by bypassing the TI and interfacing directly with the Hyperform server through the provided network protocols.

A major advantage of the TI is that it can maintain a shared representation (cache) of important hyperbase data and structure (based on events from the server) to be used by the different hypertext tools. Different views (editor, graphical browser, other browsers, etc.) on the data and structures can be based on the common cache which speeds operations in the tools, saves internal memory and network communication and reduces the server load in event-driven approaches to data distribution [50].

The extended client-server architecture supports extensibility at two levels in systems: storage and application, leaving it up to the application programmer to decide in each case where extensions give best results in terms of flexibility and efficiency. In some cases it might be best to place the extensions in the server (if they require access to many objects) and in other cases closer to the tools (to save network communication).

2.2 Hyperform Server

The built-in classes of Hyperform provide a general set of services without introducing special design policies. The classes specify a well-defined method interface to encapsulated data and behavior, enabling application programmers to abstract from basic implementation details. The application programmer can concentrate on tailoring and extending the provided services into a configuration that fulfills the chosen design policies. The classes are implemented in the inheritance hierarchy (class lattice) shown in Figure 1. Appendix A contains a specification of attributes and methods of built-in classes of Hyperform.

We start by describing the three interfaces (file system, network and shell). Then, we describe the object-oriented extensions of the Scheme language implemented in Meta Class and Object and the basic mechanisms of System Object used to extend the ELK interpreter into an object-oriented database. Finally, we describe the five subclasses of Object.

2.2.1 Interfaces

The file system, network and Unix shell interfaces are implemented in C and interfaced to the Scheme language. These three parts of the Hyperform server are updated and improved versions of similar interfaces used in the FENRIS system [6].

The file system interface allows chunks of data (objects) to be stored on disk in large collections (containers). The interface maintains an index file for each container providing direct manipulation (storage and retrieval) of chunks. Each chunk has a unique identifier associated with it. Hyperform uses one such container to store all its objects.

The network interface allows chunks of data to be transmitted over the network based on a TCP/IP protocol (sockets). The interface provides functions to open and close sockets, send and receive chunks and monitor sockets for arrival of chunks. Hyperform keeps a socket open to accept messages from clients. After processing a message, the answer is returned to the client (internet address) specified in the message.

The Unix shell interface enables two types of shell commands: interactive and non-interactive. The interface creates a Unix process, initiates the command, returns the answer (if in interactive mode) and kills the process. For example, the shell interface can be used to get the "date" for initialization of a *created* attribute in objects.

2.2.2 Object-Oriented Database Extensions

The section explains in detail the extension of the ELK Scheme interpreter into an object-oriented database. We will deal with important object-oriented issues, such as object concept, object persistency and identity, object specialization, attribute and procedure encapsulation, class evolution and message passing. We will also touch upon some basic database mechanisms to deal with system matters, such as access and caching.

2.2.2.1 Object Concept

The object approach taken in Hyperform is based on the fact that objects in hypertext systems basically consist of a number of default attributes, some dynamically allocated attributes and a number of operations on these attributes. The basic object class of Hyperform supports four categories of attributes: *read-only* (similar to class variables), *read-write* (similar to instance variables), *calculated* and *dynamic*. There is no restriction on the size and contents of attributes and all attributes can be assigned an initial value in the class description. The value of read-only attributes remains the same in all instances of the class, read-write attributes can differ from instance to instance, calculated attributes can contain any Scheme expression (which is evaluated when reading the attribute) and dynamic attributes can be added on the fly. The former two categories of attributes have their origin in object-oriented programming languages and databases, while the latter two are added to deal with special requirements of hypertext systems. The object class contains five basic methods, two instance methods – **get-instance** and **delete-instance** and three attribute methods – **get-attribute**, **set-attribute** and **delete-attribute**. The five basic methods of the Object class maintain an instance cache to speed operations. The size of the cache can be changed by invoking a System Object method (default is 100 instances).

2.2.2.2 Object Persistency, Identity and Encapsulation

Every new object in Hyperform is assigned a unique identifier (uid) by the file system interface. Currently, uid's are reused due to disk space efficiency, but a new version of the file system interface will change this. Objects are saved on disk every time they have been changed. The structure of instances on disk are:

(class-uid (read-only attributes) (read-write attributes)
(calculated attributes) (dynamically allocated attributes))

while the structure of classes are:

(ancestor descendant super-classes sub-classes class-description)

ancestor and *descendant* are used to deal with versions of classes, while *super-classes* and *sub-classes* are used to keep track of the class hierarchy.

Every class in Hyperform has its own environment in which the methods of the class are evaluated (and bound). Class environments are built when Hyperform starts up and when new classes are loaded into Hyperform with the **set-class** method of Meta Class (see later). This solution enables fast method look-up and evaluation, since class descriptions are cached in the server and do not have to be retrieved from disk every time they are used. Since not all instances in Hyperform are cached, we maintain a number of internal association lists (alists) to speed operations by minimizing disk access. We associate class uid's with names, environments, ancestors, super classes, sub-classes and uid's of instances of the class (see Appendix B).

2.2.2.3 Object Specialization

The Object class is the root of the class inheritance hierarchy. The object class can be specialized in two different ways to define new object types in Hyperform: (1) by adding new methods, which is similar to adding methods in a subclass; and (2) by adding default attributes. These attribute names will always be present in instances of that specific object type (subclass). Default attributes can be specified as either read-only, read-write or calculated. A subclass inherits all methods and default attributes from its super class(es) with the restriction that each method and attribute have a unique name. Notice that an attribute can be specified as read-only at one level and be changed to calculated in a subclass.

The multiple inheritance schema of Hyperform gives highest priority to the leftmost super class branch (specified in the class description) in case of conflicts on (multiple definitions of) default attributes and methods. If there is more than one default attribute with the same name, when traversing the super class lattice, only one attribute will be allocated and the initial value of the first met attribute specification will be used. The same conflict resolution strategy is used for methods. The first matching method, when traversing the super class lattice from left to right, is invoked when sending a message to an object.

2.2.2.4 Message Passing

Messages can be sent to objects in the following way:

(send object message . args)

The send mechanism performs method look-up and initiation. There are two send variants: **send** and **send-super**. Normally, users will only know of **send**. **send** can be used transparently over the network from the tool integrator or from within methods of Hyperform classes. **send** is invoked from the server loop of the system when a message arrives. **send-super** is intended for special cases where classes specialize the behavior of methods located in super classes and can only be used within the Hyperform server. **send-super** allows the original method of the super class to be invoked instead of the specialized method. **send-super** starts looking for method definitions in the environment of the super class(es) of the instance in contrast to **send** which starts looking in the class environment. Other aspects of the send mechanism are described in Appendix A.

2.2.2.5 Class Evolution

Meta Class provides basic operations to operate on the class lattice: **get-class**, **set-class**, **delete-class** and **create-instance**, making it possible to retrieve class descriptions, create new classes and new versions of existing classes, delete classes and create instances of classes. Multiple versions of classes are maintained to allow classes, except the three basic classes (Meta Class, System Object and Object), to evolve over time. In other words, Hyperform not only supports object-oriented data modeling, it also supports tailoring of basic database features such as concurrency control and versioning control.

Meta Class methods maintain internal alists holding the inheritance hierarchy. Different versions of a class have identical names but different uid's. Old versions of classes are kept to avoid inconsistency between instances of different versions of the class. When there is more than one version of a class, the class name refers to the most recent version. Old versions can be referenced through their uid. Class references in instances remain the same during the entire life of the instance. In this way instances of old class versions will still be operational when a new version of their class is created.

A new (version of a) class is created by sending the **set-class** message to Meta Class containing a class description consisting of two parts: a list of default attributes and specifications of methods operating on the attributes (see Figure 3).

```
((class-name AC-object)
 (read-write-att (owner ()) (created ()) (modified-by ()) (modified ()) (permission "gsdg__g__") (group ()))
 (super-class object)
 (methods (define (AC-init self . args) ... )          (define (change-permission self permission) ... )
           (define (change-group self group) ... )      (define (get-permission? self) ... )
           (define (set-permission? self) ...)         (define (delete-permission? self) ...)))
```

Figure 3. The class description for AC Object (simplified).

The syntax for class descriptions is explained in Appendix A.2 and detailed examples of class descriptions can be found in Appendix C.

2.2.2.6 Database Mechanisms

The System Object defines methods that deal with system matters such as database access, caching, class creation permissions, user groups, etc. The **connect** and **disconnect** methods are located here along with methods for specifying access control including which users (super users) are allowed to change the information kept by the System Object. The database administrator can use as many constraints as desired, but by default Hyperform is an open system allowing all users access to all features.

Basically, the System Object manages access control at the database level by storing information on users, super users and groups of users and providing methods to maintain this information. System information is kept persistently in an instance of System Object. The connect and disconnect methods maintain an internal list of clients connected to the system and their location in the network. This list is currently used by the lock mechanism to uniquely identify clients and by the event mechanism to generate event messages, but it can also be used as basis for queries. System Object also provides methods to check for permissions to operate on classes and to maintain the

cache. The size of the cache and the maximum size of instances can be specified. Instances larger than the maximum size will not be cached.

2.2.2.7 Basic Mechanisms

Hyperform implements basic features written in Scheme to provide low-level support for the extension of the ELK interpreter into an object-oriented database. In this section, we will describe three of these features: the server loop, the start-up mechanism and the conversion functions.

Basically, the server loop collects arriving messages from the server socket, evaluates them and returns the answers. In doing so, the server loop uses the network interface. Arriving messages can be of different type: *load*, *eval* and *send*. *Load* reads a file into the interpreter and evaluates the content. Load files can contain Scheme code or object code. *Eval* can be used to evaluate expressions, that is, define new global variables and functions in Hyperform. *Send* uses the send mechanism described in Section 2.2.2.4 and Appendix A.

When the Hyperform server starts up, it reads the contents of the container which must be located in the directory where the server is started. Otherwise the server will not know of the objects in the container and instead create a new container with the description of the built-in classes and start all over loading in the classes and creating class environments and internal alists.

Both the file system and the network interface operate on strings but internally Hyperform objects consist of lists of Scheme objects. Therefore, before sending objects over the network or storing them on disk they must be converted into strings and, likewise, when objects are retrieved from disk and received over the network, they are converted from strings to Hyperform objects.

2.2.3 Object Subclasses

The five subclasses of Object (Figure 1) have been implemented using the basic object-oriented features of Meta Class and Object.

2.2.3.1 Concurrency Control Object

Concurrency control is an important issue for databases and much work has been done in this area [5], but the results cannot be directly adopted by hyperbases. Hyperbases must provide special support for collaborative work [17], requiring adjustments to normal concurrency control techniques. In particular, concurrency control for hyperbases must allow very long transactions and granularity of locking is a critical issue.

The CC Object adopts the locking mechanism from the Danish HyperBase which was successfully designed to support collaborative work [48]. Locking can be done at either the attribute-level or instance-level and it is possible to read locked attributes and instances, thereby enabling browsing through a network containing locked objects. Locks provide support for long duration updates to objects in the database.

The locking mechanism of CC Object maintains an internal list of granted locks; locks do not survive from server crashes. The lock list holds the user (not the client), the object and the attribute (which can be 'ALL'). The reason for granting locks to users and not to clients (tool integrators) is that one user should be able to have more than one client running without having to remember from which client he initiates locks.

The CC Object also provides deadlock-free transactions (based on a *two-phase locking* (2PL) protocol [5] to ensure *serializability*) to be able to recover from client and server crashes. The transaction mechanism (start, commit, abort) provides the application programmer with the possibility of grouping database methods together into atomic operations.

The transaction mechanism maintains an internal list of running transactions. Again, transactions do not survive server crashes, but if they are cached in the clients, they can be retried when the server is running again. Each user can only have one transaction running; we do not support nested transactions. The transaction list holds the user, granted locks and messages in the transaction. The user explicitly has to specify which (part of) objects he wants to lock, since we do not want to introduce policies such as: should we lock when read operations are performed? We leave such decisions up to the application programmer. Each transaction begins with the **start** command, after

which all messages, except locks, from the specific user are cached in the transaction list, until either the **commit** or **abort** command is used. Locks can be performed during a transaction, with the restriction that if one lock request fails, the transaction is aborted and granted locks are released. It is then up to the client to decide whether to redo the transaction or not. If **commit** is used, all messages cached in the transaction list are evaluated and all locks are released. If **abort** is used, all locks are released.

Atomic operations involving more than one Hyperform method can also be supported by moving the operation from the application to the hyperbase using the provided extension facilities of Hyperform. This technique would speed the operation since internal operations are much faster than operations over the network. The concurrency control mechanisms provided by the CC Object overlap to some extent, leaving it up to the application programmer to decide which policy to use in each configuration of Hyperform.

2.2.3.2 Notification Control Object

Notification control allows users to be notified of important actions on the shared network of nodes and links performed by other users of the system [17]. A notification control mechanism is necessary for supporting collaboration [50]. Hyperform provides a very flexible notification mechanism that can be tailored by creating a new version or subclass of the NC Object class. The design of NC Object is based on the experiences with the Danish HyperBase gained in EHTS. EHTS used the attribute-level event and lock mechanisms of the Danish HyperBase to provide powerful support for joint authoring. In the Danish HyperBase users can subscribe to events using the function: **Event** (object, operation, attribute). NC Object goes a step further in its event support by using the expressive power of the Scheme language in event subscriptions. Events can be any Scheme expression and have access to all variables and functions in Hyperform.

```
(event (lambda ()
      (if (and (= NC-entity 5)
              (equal? user "leggett"))
          NC-operation
          #f)))
```

Figure 4. An example of event subscription using the Scheme language.

The expression in Figure 4 stipulates that all operations performed by a user named "leggett" on object number 5 will cause an event specifying the type of the operation. Event expressions are evaluated when the **send-events** method is invoked. All expressions evaluating to anything other than false cause an event to be generated and transmitted to the subscribing user.

Event subscriptions are stored in an internal list maintained by NC Object; events subscriptions do not survive a server crash. The event list holds information on the user, where he is located and the specified Scheme expression used to trigger events. Each event subscription is assigned a unique id, which is returned to the user. Whenever an event is transmitted to the user, he receives the event id and the results from the evaluation of the event expression. The (**send-events** object method attribute) message assigns the value of its three arguments to global Hyperform variables named NC-entity, NC-operation and NC-attribute. These variables are the prime candidates to be used in event subscriptions allowing users to monitor all operations on all (parts of) objects. Appendix B gives an exhaustive list of internal Hyperform lists and variables that can be used to trigger events and as return values from events.

send-events can be included in all methods in subclasses created in Hyperform and is (by default) invoked when using the methods of Meta Class and System Object since these cannot be versioned or subclassed. **send-events** can also be invoked with specific parameters allowing "high-level" events to be generated; for instance, events describing actions performed in the tool integrator instead of Hyperform.

2.2.3.3 Access Control Object

As hypertext systems evolve from single user to multiuser systems the need for controlling user access to shared objects arises. The simple read/write protection scheme is not appropriate and should be augmented with at least a third level of protection, annotate, allowing users to attach links to objects to which they have read (but not write) access.

The design of AC Object general services was influenced by the protection mechanism in Unix. We support three different levels of object protection: *get*, *set* and *delete* which correspond to the basic operations on instances and attributes and we adopt the notion of "user/group/others" from

Unix.

The object level access control is provided by maintaining a number of attributes in AC Object instances: *owner*, *created*, *modified-by*, *modified*, *permission* and *group*. An initialization function allows users to assign appropriate access permissions to a group of users defined in the System Object. If nothing is specified, the default permissions (*gsdg__g__*) and group (the user himself) will be used. AC Object provides methods to check if a user has appropriate permissions. These methods can be used to restrict access to instances by including them in all operations of the data model to see if the user has the required permission to perform the operation in progress.

Since there is no notion of nodes, links and composites in Hyperform, it is not possible to talk about annotation rights as being part of the general services of the AC Object. Basically, annotation rights can be implemented with *set* access. Once a data model is loaded into Hyperform, the AC Object can easily be updated to include an annotate access right (or other levels of protection) by creating a new version of AC Object and reusing most of the original class description (Figure 3).

2.2.3.4 Version Control Object

Versioning is an important feature for hypertext systems [17], yet little research has been done to uncover appropriate versioning models for hypertext [19]. The requirements of versioning in hypertext systems can be categorized into two main areas: versioning *data* and versioning *structure* [20]. Versioning data has its parallel in version control, while versioning structure has its parallel in configuration management [34].

Versioning structure is heavily dependent on the data model used in the system. It is impossible (at this point in time) to provide general hyperbase support for versioning structure in hypertext without introducing a fixed data model. Versioning data is independent of the data model but still requires policies to be introduced; for instance, the versioning model to be used: timeline, tree, network or a new hypertext versioning model. The VC Object was created to address data versioning issues. Since structure is contained in the attributes of Hyperform it is likely that versioning structure would be done by subclassing the VC Object (or one of its variants) once a particular data model is introduced.

The data versioning support in VC Object is inspired by RCS [42] and Gypsy [10] and is based on the tree model (see Figure 5). VC Object supports revisions, variants and releases of objects in delta or fully constituted form; VC Object maintains certain attributes in its instances in doing so: *versiongroup*, *version*, *previous*, *next* and *variants*. Users can check-out instances of objects for update and later check them in as new revisions, variants or releases of the instance. VC Object takes care of version numbering based on the type of instance versioning requested at check-out time. At check-in time the user may specify whether the previous version of the object should be stored as a delta or as a complete copy. At any time a message can be sent to objects that changes the method of storage. Versions of an instance are kept in version groups and earlier versions can be retrieved through the query mechanism. A message can be sent to any object (delta or complete) that returns the complete version of the object. In addition, VC Object provides a method returning all the objects in a version tree in preorder.

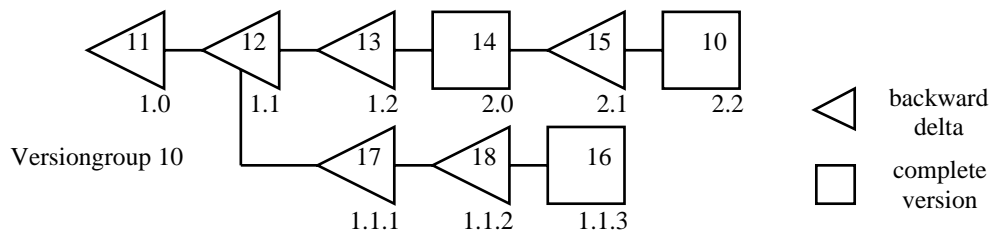


Figure 5. The tree version model used in Hyperform supporting revisions, variants and releases.

The latest version of every branch in the tree will always be complete, but VC Object also allows intermediate versions in fully constituted form. To give an idea of how the version mechanism works, we describe how the tree of Figure 5 was created. Object number 10 was created and initialized (hence versiongroup 10). Two versions of the object were checked out (1.1 to 1.2) and subsequently checked in using the delta storage option. A release was created (2.0) and two versions of the release were created (2.1 to 2.2). Then, we checked out a variant of object #12 (1.1.1) and created two versions of the variant (1.1.2 to 1.1.3). As can be seen in Figure 5, the object #10 will always be the latest version of the main branch and, likewise, object #16 will always be the latest version of the variant branch, since they were both the first versions in their particular branch. In this way we only have to remember one object id from each branch to always know the latest version of the branch, and the latest version of the main branch will be the name of the versiongroup.

We use a specialized form of backward delta storage (geared toward the storage method in Hyperform) to reduce the amount of storage required and to allow flexible access to previous versions of the instance. The delta mechanism operates at the attribute level. Four of the five version attributes (*version*, *previous*, *next* and *variants*) are present in all versions (delta and complete), but all other attributes are only present in delta versions if they have been changed in the following version. The delta mechanism could probably be more storage efficient if we also applied "diff" to the attributes that have been changed and only saved the difference between changed attributes, and not the whole attribute like in this version of Hyperform. But, the storage efficiency gain depends very much on the size of the attribute values. With small sized attribute values, applying "diff" could in fact increase the amount of storage, while with large sized attribute values, "diff" would definitely decrease the amount of storage required. Note that it is very easy to use Unix tools, such as "diff", via the shell interface in Hyperform.

A small number of researchers are currently addressing hypertext versioning issues [20,21,44,45,56], but much work still needs to be done. Hyperform, due to its extensible nature, can be used as a research vehicle in determining the right versioning model for hypertext. In particular, Hyperform's rapid prototyping facility can be used for addressing the issue of partitioning versioning support among the various architectural layers (e.g., hyperbase vs. application).

2.2.3.5 Query and Search Object

The basic information access metaphor in hypertext is navigation. In general, navigation becomes inefficient as the hypertext information space grows larger and larger [17]. Techniques from database and information retrieval (IR) systems should be incorporated into hyperbase technology to provide alternate ways of accessing information in hypertext. Halasz suggests query-based mechanisms supporting both *content* and *structure* search. Content search involves IR techniques such as keyword, index and full-text retrieval and database techniques such as query-based retrieval. Structure search involves a special structural query language capable of retrieving specified subgraphs of the network and therefore depends very much on the data model in the system.

We provide basic content search facilities in Hyperform that can be extended to provide support for more powerful IR methods. The content search mechanism of Query and Search Object is list-based and supports the use of all list manipulating functions of the Scheme language. In addition, we provide a general filtering function capable of matching specific values in attributes (remember, all information in Hyperform is stored in attributes) and basic list operations such as union, intersect and subtract.

Two variants of the filtering function are provided; one for classes, **match**, and one for instances, **filter**. **match** can retrieve information on attributes and methods defined in class descriptions, class names, class ancestors, class descendants, super classes and sub-classes. **filter** can retrieve information on attributes present, attribute values present, attributes with specific values (exact match) or attributes including specific values (member match). It can be specified which parts of the instances should be used for the match: all attributes, read-only attributes, read-write attribute, calculated attributes or dynamic attributes.

We can also access all internal lists of Hyperform, retaining information on classes, instances, etc. (see Appendix B). These basic methods (inspired by [13]) can be composed into very powerful content searches using the composing mechanisms of the Scheme language. We do not provide direct support for structure search since we do not want to introduce a fixed data model. Although, since all structural information is stored in attributes in Hyperform, structure search mechanisms can be built on top of the basic content search facilities once a data model is introduced.

3. USING HYPERFORM

In this section we show how Hyperform can be used as a hyperbase development tool by giving examples of developing specific hyperbase configurations: the HAM and the Danish HyperBase. We also discuss the use of Hyperform in a more general context.

3.1 Hyperform as HyperBase Development Tool

Development of hyperbases involves addressing certain critical issues. A data model must be determined and decisions concerning the degree of concurrency control, notification control, access control, version control and query and search support must be made. Once these policies are established the actual implementation must be done. Hyperform can be valuable in the policy-setting stage as well as the implementation stage. In the policy-setting stage one might use the rapid prototyping features of Hyperform to quickly experiment with alternate designs. In the implementation stage Hyperform provides the basic building blocks for implementation of the hyperbase.

The next two sections contain examples of modeling two HAM generation hyperbases. Section 3.1.1 gives an overview of the steps necessary to simulate the HAM and Section 3.1.2 gives a detailed example of simulating the Danish HyperBase.

3.1.1 Overview of Simulating the HAM

In this section we describe how the facilities of Hyperform would be used to simulate the HAM. The first step is to tailor the functionality by creating new versions of five Hyperform classes to simulate the functionality of the HAM in the same five areas. In particular, the following should be done (Figure 6):

- CC:** Tailor transaction mechanism and shadow locking mechanism (HAM does not provide user-controlled locking);
- NC:** Tailor event mechanism to simulate demon mechanism;
- AC:** Tailor access control mechanism and extend with annotate access rights;
- VC:** Tailor version control mechanism to use time line model and extend with copy and merge semantics;

QS: Tailor query and search mechanism.

One of the above classes should also tailor the attribute mechanism provided in the Object class to simulate HAM attribute operations. The data model should then be created as new classes inheriting features from the five tailored classes to provide the basic operations of the HAM. The final step is to create a C client library that provides the HAM interface. HAM clients (application tool or end-user) would not notice the difference.

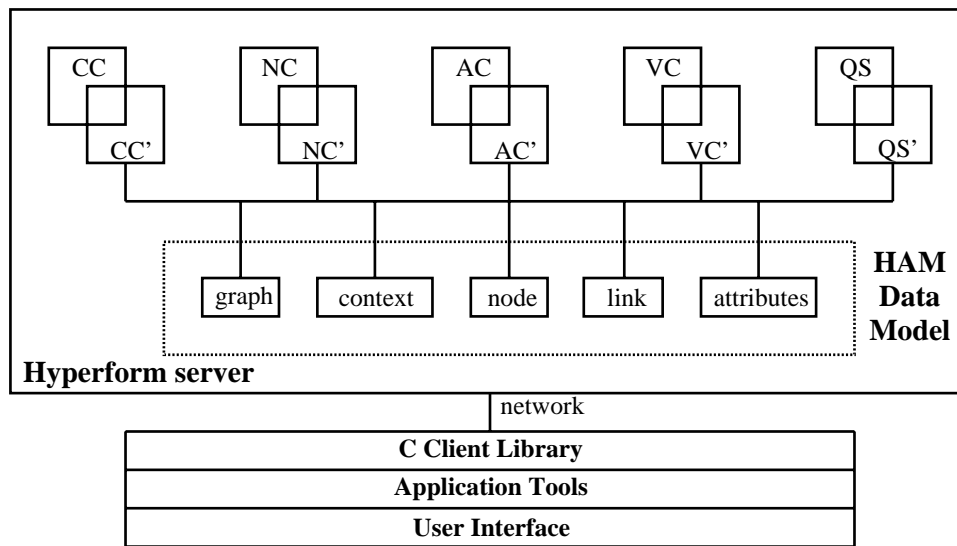


Figure 6. Overview of simulating the HAM with Hyperform.

3.1.2 Simulating the Danish HyperBase

A question that automatically comes up when introducing a flexible system such as Hyperform is "How efficient is the system?" Flexibility and efficiency are often counteracting factors in system development. To answer this question we used Hyperform to simulate all the functionality of the version of the Danish HyperBase used in the EHTS experiment (Figure 7). This allows comparisons between Hyperform and a hyperbase developed in an efficient language, in this case C++.

Briefly described, the simulation includes three steps: (1) development of the HyperBase data

model, (2) initial performance test of the simulated data model and (3) porting the EHTS Editor from HyperBase to the Hyperform architecture. The final step also includes a performance test of the data model using the EHTS Editor.

3.1.2.1 Developing the HyperBase Data Model

We start by analyzing the data model in HyperBase. The HyperBase operation set, specified in [49], contains four categories of operations:

1. Structure Operations:

CreateNode () ----> (Entity)
CreateLink (ToNode) ----> (Entity)
UseLink (FromNode,Link) ----> ()
MoveLink (Link,NewNode) ----> ()
RemoveLink (FromNode,ToLink) ----> ()
Delete (Entity) ----> ()

2. Read and Write Operations:

Read (Entity,Key) ----> (Value)
Write (Entity,Key,Value) ----> ()
Enumerate (EntityType) ----> (EntityList)

3. Multiuser Operations:

Lock (Entity,Key) ----> ()
UnLock (Entity,Key) ----> ()
ShowLock (Entity,Key) ----> ()
Event (Entity,Operation,Key) ----> ()
UnEvent (Entity,Operation,Key) ----> ()
ShowEvent (Entity,Operation,Key) ----> (Users)

4. Other Operations:

Connect (Server,UserID) ----> ()
Disconnect () ----> ()

Analyzing the operation set from an object-oriented point of view, it can be divided into three types: operations that can be performed on both entity types and specific node and link operations. HyperBase operations in category 3 and Enumerate are entity operations, while operations in category 1 and Read and Write are node and link operations. Therefore, we created three classes in Hyperform: the Entity, Node and Link classes (see Figure 7). Read and Write belong in the Node and Link classes, since both operations are closely related to the actual attributes in node and links; they check to see if specified attributes exist before carrying out the rest of the operation. The connect and disconnect methods of System Object are used as HyperBase category 4 operations.

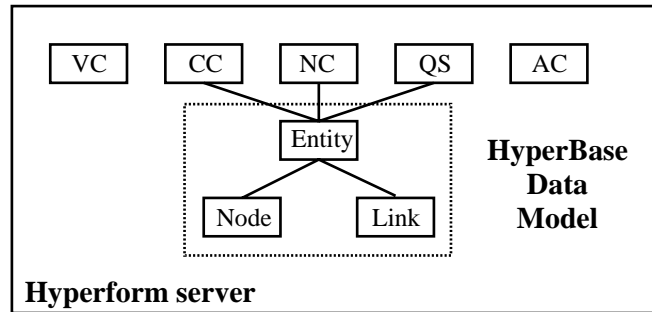


Figure 7. Simulating the HyperBase data model with Hyperform.

In the following sections, we go into detail with each of the three HyperBase data model classes. In the development of all three classes we had to consult with [46] to determine possible return values (error codes) from corresponding HyperBase operations.

3.1.2.1.1 The Entity Class

As depicted in Figure 7, the Entity class inherits functionality from CC Object, NC Object and QS Object. These classes are described in detail in Appendix A. Enumerate combines two QS Object methods, lock operations tailor CC Object methods, while event operations tailor NC Object functionality.

Enumerate uses the **filter** and **get-all** methods of QS Object to retrieve all instances belonging from either nodes or links. The locking mechanism in CC Object is based on the locking mechanism in HyperBase; the two mechanisms have almost identical functionality. The tailoring of the CC Object locking mechanism is, therefore, only a matter of changing the method names and converting the return values into HyperBase error codes. The event mechanism of NC Object is more powerful than the corresponding HyperBase event mechanism and can with little effort be tailored to simulate the HyperBase style event mechanism. Based on the arguments to the Event method, we use the three Hyperform variables NC-entity, NC-operation and NC-attribute (see Appendix B) to create a Scheme expression which is passed on to the event mechanism of NC Object. The returned event-id is associated with the user, entity, operation and attribute to be used in both the UnEvent and ShowEvent methods, since Hyperform events have a unique number and

HyperBase events are quadruples: (user, entity, operation, attribute). In this way we can include **send-events** in all operations of the HyperBase data model to trigger events.

The class description for the Entity class is listed in Appendix C.1.

3.1.2.1.2 The Node Class

The Node class inherits functionality from the Entity Class and provides specific node type operations: CreateNode, Delete, UseLink, RemoveLink, Read and Write. The Node class description is listed in Appendix C.2.

Node operations are mostly based on basic Meta Class and Object methods: **create-instance**, **delete-instance**, **get-attribute** and **set-attribute**, but we also use methods from CC Object to check for locked attributes and instances. The methods of the Node class simulate the corresponding operations of the HyperBase data model which, in addition to using basic Hyperform methods, includes maintaining different attributes in nodes and connected links.

3.1.2.1.3 The Link Class

The Link class description (listed in Appendix C.3) inherits features from the Entity Class and adds the following specific link type operations: CreateLink, Delete, MoveLink, Read and Write. Like Node operations, Link operations can be simulated by using basic Hyperform methods and maintaining affected attributes in links and connected nodes.

3.1.2.1.4 Simulation Results

The experiment showed the time it takes to simulate HyperBase in Hyperform is **two days of effort**, compared to the **two man years** of effort it took to develop the original HyperBase. The simulated operations in Hyperform each consist of 4 to 16 lines of Scheme code which gives a total of 8 KBytes of code for the 3 HyperBase classes in Hyperform compared to 350-400 kByte of C++ code to develop HyperBase.

3.1.2.2 Initial Performance Test

Initially, the efficiency of Hyperform was tested by comparing the speed of simulated HyperBase operations invoked from the tool integrator with the original HyperBase operations invoked from a HyperBase client. The results are shown in Table 1.

Seconds for 100 operations	Hyperform	HyperBase	Difference
Structure	43.9	20.0	- 54.4%
Write	26.6	20.0	- 24.8%
Read	13.1	20.0	+ 52.8%
Enumerate	76.1	61.6	- 19.1%
Lock	15.0	19.6	+ 30.7%
Event	11.3	20.2	+ 78.8%
Average	31.0	26.9	- 13.2%

$$\text{Difference} = \frac{\text{HB-HF}}{\text{HF}}$$

Table 1. Results from the Hyperform performance test.

The tests showed that half of the simulated operations in Hyperform were faster (read, lock, event) and half were slower (structure, write, enumerate) than the corresponding operations in HyperBase. Overall, the performance of this non-optimized version of Hyperform was only 13.2% slower on average than the original C++ implementation of HyperBase. The tests were made with almost ideal conditions (best case), that is, low load on the tool integrator/HyperBase client machine, the network and the Hyperform/HyperBase server machine. In most cases the time difference would be a much smaller percentage as the load on the network and server are likely to increase in real use situations.

3.1.2.3 Porting EHTS Editor to Hyperform

The next step in the HyperBase simulation experiment involves porting the EHTS Editor from HyperBase to the Hyperform architecture. We thought about two different approaches: (1) we could leave both the Hyperform server and the EHTS Editor exactly the way they are and use the tool integrator to transform the HyperBase protocol into the Hyperform protocol, or (2) we could change the EHTS Editor to conform to the Hyperform protocol. The first solution requires that the tool integrator communicate with the EHTS Editor using the HyperBase network protocol, transform HyperBase function calls into Hyperform messages and transform return values from Hyperform into HyperBase error codes and format. The second solution requires that the network communication part of the EHTS Editor is changed to use the Hyperform network protocol.

We chose the second solution. Once the EHTS Editor is interfaced to the Hyperform architecture, we can use Hyperform extension facilities to speed operations. Since the experiment involves only one tool we could gain efficiency by bypassing the tool integrator (Figure 8A). However, the EHTS Editor can also run in cooperation with the tool integrator (Figure 8B).

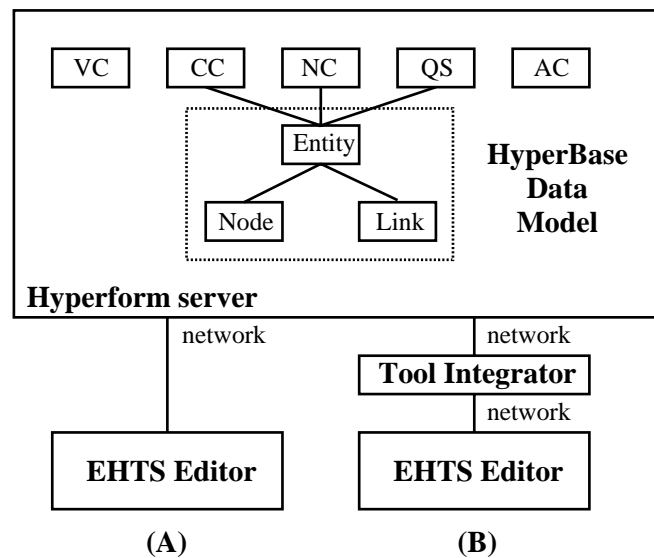


Figure 8. Simulating HyperBase with Hyperform, two configurations: (A) and (B).

It took less than a week to port and optimize the EHTS Editor. By using the extension facilities of Hyperform we were able to make all operations of the EHTS Editor run at least as fast on Hyperform as on HyperBase. Every operation in the EHTS Editor that requires more than one HyperBase operation can be performed by one new Hyperform operation thus saving network communication time. For example, eight HyperBase operations are required for the EHTS Editor to create and initialize a node – CreateNode and seven Write operations to initialize the attributes. In Hyperform, we were able to do this with one new operation in the Node class (CreateInitNode) using the CreateNode and Write operations.

The extensions to the three HyperBase data model classes are listed in Appendix C. With these extensions the three class descriptions increase from 8 to 13 KByte of code.

3.2 Other Applications of Hyperform

In this section we look at different application areas where Hyperform can be useful. In doing so we touch upon the question "What is Hyperform?" and discuss issues in scaling up the Hyperform architecture.

3.2.1 What is Hyperform?

In Section 4 we argue that Hyperform can be classified as an extensible object-oriented database management system. Another way of classifying the Hyperform architecture is as an interpreter supporting a Scheme dialect with object-oriented data modeling facilities and persistent, sharable objects.

The extensible object-oriented database (object server) view opens a wide range of possible applications for using Hyperform as a database: office information systems, CSCW systems, software engineering systems and programming systems, to mention a few. Since Hyperform can be extended and tailored on the fly, it can also be useful as a research engine in addressing future database issues (as discussed in [39]) in a fast prototyping manner.

The Scheme interpreter view opens new types of Scheme applications since the Hyperform architecture supports persistent, sharable objects that can be composed of all kinds of Scheme types and functions.

Both of the above views are applicable, but Hyperform's intended application area is hypertext. Within the area of hypertext we can identify a number of possible applications of Hyperform:

- Link engine for inter-application linking [24,31,32,35,36].
- Data interchange between existing hypertext data models [29], since more than one data model can be simulated in the platform simultaneously. Transformation (interchange) objects can be created, inheriting functionality from both data models and providing operations to convert from one format to the other (and vice versa).
- Research engine for future hypertext applications, since Hyperform provides flexible and efficient support for data storage and data sharing. The extension facilities eases experimentation with data models; new data models can be derived from existing and shaped towards new applications.

We have also shown that Hyperform can be used to simulate HAM generation hyperbases. Finally, we have identified Hyperform as the first multi-database (hyperbase) system, since Hyperform allows heterogeneous database configurations to coexist concurrently.

3.2.2 Scaling Up the Hyperform Architecture

The hyperbase configurations developed so far for Hyperform (e.g. the Danish HyperBase data model) have one thing in common, they are not intended to handle the enormous amounts of data required in next generation software systems. In this section we discuss how Hyperform can be extended to match these storage requirements.

The EHTS experiment shows that small objects are handled very efficiently in Hyperform; at least as efficient as in the C++ coded Danish HyperBase, but Hyperform becomes progressively slower

when handling large objects. This is partly due to the relatively simple file system interface and limitations in the implementation of the ELK Scheme interpreter.

The file system interface uses indexing techniques to provide direct access to objects. All objects are saved in a single file using single, double and triple pointers between equal sized blocks of data. This is similar to how Unix manages its i-nodes. The Danish HyperBase uses a similar way of storing its objects, but does not get progressively slower with the size of the objects, so our problem is mostly caused by the problems with ELK.

Before compiling the ELK Scheme interpreter you can specify a number of constants in a configuration file. Two of these are a fixed heap size and a fixed maximum size of strings. These two constants posed a great number of problems in the implementation of Hyperform. It seems that whatever heap size you specify in the configuration file, there are problems related to it. Small heaps run out of storage (which causes ELK to stop) and large heaps take up a lot of memory on the Hyperform server machine (which causes a lot of swapping). Strings larger than the maximum size have to be split into smaller units, which seems to be very time consuming in ELK. The string problem is probably related to the fixed heap size causing a lot of garbage collection to take place when handling large objects.

Despite the problems with ELK, Hyperform can certainly be extended to handle large amounts of data. One obvious way is to save all small objects (e.g. text and graphic objects) and only save parts of large objects (e.g. sound and video objects) using the provided file system interface. The data attribute of large audio and video objects could be a reference to where the actual data can be retrieved (e.g. file system, CD-ROM, optical disk). This solution preserves object identity, since the basic attributes of specialized objects (those that bypass the Hyperform file system interface) are maintained by the Hyperform storage manager.

It is also possible to address the scale issue in more fundamental ways. We can replace the simple Hyperform storage manager with an efficient storage manager from an existing database project. If later implementations of ELK do not solve the constant value problems, we can port Hyperform to another interpreter (Lisp or Scheme) that does not have similar problems, e.g. GNU Emacs (the present version of GNU Emacs does not allow dynamic loading of object code, which is a major

reason why we chose ELK). These solutions would allow larger objects and a larger number of objects to be stored efficiently in Hyperform. Other storage devices are still necessary to deal with large sound and video objects.

Finally, we can distribute the Hyperform server across several machines in the network to deal with the scale issue. The proposed architecture has been designed with multiple servers in mind. The tool integrator can easily be extended to deal with multiple Hyperform servers. In this way we will not have to change a single line of code in the Hyperform servers to provide a truly distributed hyperbase system of either homogeneous or heterogeneous hyperbases (homogeneous and heterogeneous in the sense of the hyperbase data model).

4. RELATED WORK

Hyperform is an extensible, tailorable hyperbase system based on an internal computational engine that provides an object-oriented extension language based on Scheme. Other approaches to hyperbases include the HAM [8], GMD-IPSI's HyperBase [37], University of Aalborg's HyperBase [47], HRL's HB1 [35] and HB2 [36], UNC's Graph Server [38] and RMIT's Hyperion [55]. The referenced hyperbase system approaches differ in their intended application areas, but have one thing in common: they are all general-purpose hyperbase systems intended to provide application independent hypertext support and mostly they are based on the client-server model.

HAM was designed to support hypertext based CAD and CASE applications, the German HyperBase was designed to support a hypertext based authoring system and the Danish HyperBase was designed with special support for collaboration. HRL's HB1 and HB2 feature open, distributed and extensible architectures focusing on support for inter-application linking, UNC's Graph Server services a hypertext system for structuring ideas in collaborative working sessions, such as large software development projects, and RMIT's Hyperion concentrates on efficient management (storage and retrieval) of large volumes of text.

All these approaches, the HAM generation, are characterized by having a fixed data model as described in the introduction; HAM generation data models are surveyed in [7] and [23]. In contrast, Hyperform provides a basic set of building blocks (a library of object-oriented software routines) that can be extended to provide virtually any kind of hyperbase support. The HAM generation introduces specific design policies in addressing hypertext issues, while Hyperform can be used to experiment with different design policies to allow important hypertext issues to be addressed in a fast prototyping manner.

Using object-oriented data modeling in connection with hypertext has also been suggested by a number of other researchers [51,52]. The tendency in these approaches is to suggest a data model which can be tailored to meet specific needs of applications. None have introduced extensibility and tailorability into other parts of the hyperbase system such as concurrency and notification control mechanisms.

Based on the seven Halasz issues [17,19], Lange et al. [26] identify seven requirements for the sub-issue of the storage mechanism in hypertext based systems: *openness and distribution, sharing, integrity, multimedia, querying, versioning and extensibility*. Throughout this paper we have shown how the features of Hyperform can be used to address these issues, which confirms with research performed by Lange [27] suggesting the use of object-oriented databases over relational databases, knowledge bases and file systems in providing support for the above seven issues for hypermedia storage. A case study by Smith and Zdonik [40] also suggests the use of an object-oriented database (ENCORE) over a traditional relational database (INGRES) to provide hyperbase support for the Intermedia system.

Hyperform incorporates techniques from existing fields such as concurrency control from the database field [5] and versioning control from the software engineering field [34]. Hyperform can be categorized as an extensible object-oriented database management system, since we support common object-oriented database features [54] and introduce extensibility into basic database features, such as concurrency control and versioning control. Other approaches to extensible database management systems include EXODUS [9] and GENESIS [4]. The family of object-oriented database management systems includes ORION [3], GemStone [30], Iris [12] and ENCORE [22].

5. SUMMARY

We have presented an approach to hyperbases based on the concept of extensibility. Hyperform provides basic hyperbase services in the areas of object-oriented data modeling, concurrency control, notification control, access control, version control and search and query. We introduced as little policy as possible when designing the basic services, thereby enabling Hyperform to be tailored and extended to provide virtually any hyperbase configuration. Hyperform addresses important next generation issues such as extensibility and tailorability and provides support for virtual structures and active data storage (computations) since it is based on an internal computational engine.

The use of Hyperform has shown that it greatly reduces the effort it takes to develop high quality customized hyperbases. We have described how Hyperform can be used to simulate the HAM and the Danish HyperBase, but more importantly, we showed the effort it takes to simulate a HAM generation hyperbase is less than a week. Therefore, we estimate the effort it would take to develop a fully featured "next" generation hyperbase in Hyperform is a matter of months instead of years.

Since there are no restrictions on the number of classes and objects, it is possible to have more than one data model and hyperbase configuration running in Hyperform at the same time. Hyperform is the first multi-hyperbase system and a very suitable platform for experimenting with different policies and mechanisms for the major hyperbase issues of future hypertext systems. The proposed dynamic, open, extensible architecture with the tool integrator providing extensibility at both the database and application level makes this approach equally well suited for experimentation with hypertext system configurations as for experimentation with hyperbase configurations.

6. STATUS

The first version of Hyperform presented in this paper has been implemented and tested. We have an ongoing effort to optimize the implementation to make Hyperform as fast as the Danish HyperBase without the use of the extension capabilities. We plan to release the software in the public domain after having tested the implementation thoroughly, preferably in collaboration with a number of other research sites.

Future plans include experimenting with hyperbase distribution and other issues involved in scaling up the Hyperform architecture to provide efficient support for next generation hypertext systems.

Acknowledgments

We would like to thank Kasper Østerbye for fruitful comments and ideas during the early design phase, Kurt Nørmark for inputs on implementing OO concepts in Scheme and Bent Bruun Kristensen for helpful discussions throughout the project. We are also most grateful for the help of Cindy Kunz in the preparation of this manuscript. Part of this work has been carried out during the first author's one-year visit at the Hypermedia Research Laboratory at Texas A&M University, sponsored by the Danish Research Academy - grant # S910130.

7. REFERENCES

1. Afrati, F., and Koutras, C. D. 1990. A hypertext model supporting query mechanisms. In *Hypertext: Concepts, Systems and Applications, Proceedings of the European Conference on Hypertext*, (Paris, France, November), A. Rizk, N. Streitz, and J. Andre, Eds., Cambridge University Press, 52-66.
2. Akscyn, R., McCracken D., and Yoder, E. 1988. KMS: A distributed hypermedia system for managing knowledge in organizations. *Commun. ACM*, 31, 7, (July), 820-835.
3. Banerjee, J., Chou, H., Garza, J., Kim, W., Woelk, D., Ballou, N., and Kim, H. 1987. Data model issues for object-oriented applications. *ACM Trans. Off. Inf. Syst.*, 5, 1, (January), 3-26.
4. Batory, D. S., Barnett, J. R., Garza, J. F., Smith, K. P., Tsukauda, K., Twichell, B. D., and Wise, T. E. 1990. GENESIS: A reconfigurable database management system. *IEEE Trans. Softw. Eng.*, (November), 1258-1272.
5. Bernstein, P. A., and Goodman, N. 1981. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13, 2, (June), 185-221.
6. Boel, M., Gregersen, A., Larsen, P., and Møller, F. H. 1991. Manual og dokumentation til FENRIS version 2.0. Department of Computer Science Internal Report IR 91-02, The University of Aalborg, Denmark, February, (in Danish).
7. Boone, J. 1991. A survey of data models for hypermedia. Department of Computer Science Technical Report TR91-022, The University of North Carolina at Chapel Hill, April.
8. Campbell, B., and Goodman, J. 1988. HAM: A general-purpose hypertext abstract machine. *Commun. ACM*, 31,7, (July), 856-861.
9. Carey, M., DeWitt, D., Graefe, G., Haight, D., Richardson, J., Schuh, D., Shekita, E., and Vandenberg, S. 1990. The EXODUS extensible DBMS project. Readings in object-oriented databases, S. Zdonik, and D. Maier, Eds., Morgan Kaufmann, 474-499.
10. Cohen, E., Soni, D., Gluecker, R., Hasling, W., Schwanke, R., and Wagner, M. 1988. Version management in Gypsy. *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Boston, MA, November), 201-215.

11. Delisle, N., and Schwartz, M. 1986. Neptune: A hypertext system for CAD applications. *Proceedings of the ACM International Conference on the Management of Data (SIGMOD)*, 132-143.
12. Fishman, D., Beech, D., Cate, H., Chow, E., Connors, T., Davis, J., Derrett, N., Hoch, C., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M., Ryan, T., and Shan, M. 1987. Iris: An object-oriented database management system. *ACM Trans. Off. Inf. Syst.*, 5, 1, (January), 48-69.
13. Fuller, M., Kent, A., Sacks-Davis, R., Thom, J., Wilkinson, R., and Zobel, J. 1991. Querying in a large hyperbase. *Proceedings of the Second International Conference on Database and Expert Systems Applications*, (Berlin, Germany, August), 455-458.
14. Garg, P. K. 1988. Abstraction mechanisms in hypertext. *Commun. ACM*, 31, 7, (July), 862-870.
15. Haan, B. J., Kahn, P., Riley, V. A., Coombs, J. H., and Meyrowitz, N. K. 1991. IRIS hypermedia services. *Commun. ACM*, 35, 1, (January), 36-51.
16. Halasz, F., Moran, T., and Trigg, R. 1987. NoteCards in a nutshell. *Proceedings of the CHI '87 Conference on Human Factors in Computing Systems*, (Toronto, Canada, April), J. M. Carroll and P. P. Tanner, Eds., 45-52.
17. Halasz, F. 1988. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Commun. ACM*, 31, 7, (July), 836-852.
18. Halasz, F., and Schwartz, M. 1990. The Dexter hypertext reference model. *Proceedings of the Hypertext Standardization Workshop*, (Gaithersburg, MD, January), 95-133.
19. Halasz, F. 1991. Hypertext '91 Keynote Talk. *ACM Conference on Hypertext (Hypertext '91)*, (San Antonio, Texas, December).
20. Hicks, D. L., Leggett, J. J., and Schnase, J. L. 1991. Version control in hypermedia databases. Department of Computer Science Technical Report TAMU-HRL 91-004, Texas A&M University, College Station, Texas, July.
21. Hicks, D. L., Leggett, J. J., and Schnase, J. L. 1991. Version control in hypermedia: An open systems perspective. Submitted to the European Conference on Hypertext (ECHT '92), Milan, Italy.

22. Hornick, M., and Zdonik, S. B. 1987. A shared, segmented memory system for an object-oriented database. *ACM Trans. Off. Inf. Syst.*, 5, 1, (January), 70-95.
23. Kacmar, C. J., Leggett, J. J., Schnase, J. L., and Boyle, C. 1988. Data management facilities of existing hypertext systems. Department of Computer Science Technical Report TAMU 88-018, Texas A&M University, College Station, Texas, September.
24. Kacmar, C. J., and Leggett, J. J. 1991. PROXHY: A Process-Oriented Extensible Hypertext Architecture. *ACM Trans. on Inf. Syst.*, 9, 4, (October), 399-419.
25. Lange, D. B. 1990. A formal model of hypertext. *Proceedings of the Hypertext Standardization Workshop*, (Gaithersburg, MD, January), 145-166.
26. Lange, D. B., Østerbye, K., and Schütt, H. 1992. Hypermedia storage. Submitted to the European Conference on Hypertext (ECHT '92), Milan, Italy.
27. Lange, D. B. 1992. A hypertext system is a database application. Submitted to the 26th Annual Hawaii International Conference on System Sciences (HICSS-26), Kauai, Hawaii.
28. Laumann, O. 1990. Reference manual for the ELK Extension Language Interpreter, (May). (This document is part of the ELK system available via anonymous ftp from mcsun.eu.net).
29. Leggett, J. J., and Killough, R. L. 1991. Issues in hypertext interchange. *Hypermedia*, 3, 3, 159-186.
30. Maier, D., Stein, J., Otis, A., and Purdy, A. 1986. Development of an object-oriented DBMS. *Proceedings of the OOPSLA '86 Conference*, (Portland, OR, September), 472-482.
31. Meyrowitz, N. 1989. The missing link: Why we're all doing hypertext wrong. In *The Society of Text: Hypertext, Hypermedia, and the Social Construction of Information*, Edward Barrett, Ed., The MIT Press, 107-114.
32. Pearl, Amy. 1989. Sun's link service: A protocol for open linking. *Proceedings of the ACM Conference on Hypertext (Hypertext '89)*, (Pittsburgh, PA, November), 137-146.
33. Rees, J., and Clinger, W., Eds. 1986. The revised³ report on the algorithmic language Scheme. In *ACM SIGPLAN Notices*, 21, 12, (December).
34. Rohrbach, R., and Seiwald, C. 1988. Galileo: A software maintenance environment. *Proceedings of the International Workshop on Software Version and Configuration Control*, (Grassau, Germany, January), 444-456.

35. Schnase, J. L., Leggett, J. J., and Hicks, D. L. 1991. HB1: Initial design and implementation of a hyperbase management system. Department of Computer Science Technical Report TAMU-HRL 91-003, Texas A&M University, College Station, Texas, October.
36. Schnase, J. L. 1992. HB2: A hyperbase management system for open, distributed hypermedia system architectures. Dissertation, Department of Computer Science, Texas A&M University, College Station, Texas, August.
37. Schütt, H. A., and Streitz, N. 1990. *HyperBase*: A hypermedia engine based on a relational database management system. In *Hypertext: Concepts, Systems and Applications, Proceedings of the European Conference on Hypertext*, (Paris, France, November), A. Rizk, N. Streitz, and J. Andre, Eds., Cambridge University Press, 95-108.
38. Shackelford, D. E. 1991. Requirements document for the UNC distributed graph service. Department of Computer Science Technical Report TR91-051, The University of North Carolina at Chapel Hill, December.
39. Silberschatz, A., Stonebraker, M., and Ullman, J., Eds. 1991. Database systems: Achievements and opportunities. *Commun. ACM*, 34, 10, (October), 110-120.
40. Smith, K., and Zdonik, S. B. 1987. Intermedia: A case study of the differences between relational and object-oriented database systems. *Proceedings of the OOPSLA '87 Conference*, (Orlando, FL, October), 452-465.
41. Stallman, R. M. 1984. EMACS: The extensible, customizable, self-documenting display editor. In *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds., McGraw-Hill, NY, 300-325.
42. Tichy, W. F. 1985. RCS – A system for version control. *Software -- Practice and Experience*, 15, 7, (July), 637-654.
43. Tompa, F. W. 1989. A data model for flexible hypertext database systems. *ACM Trans. Off. Inf. Syst.*, 7, 1, (January), 85-100.
44. Weber, A. 1991. Versioning issues in hypermedia publishing environments. Arbeitspapiere 542, Gesellschaft für Mathematik und Dataverarbeitung MBH, Schloss Birlinghoven, Postfach 12 40, D-5205 Sankt Augustin 1, June.
45. Weber, A. 1992. TITLE??? Submitted to the European Conference on Hypertext (ECHT '92), Milan, Italy.

46. Wiil, U. K. 1990. Design and implementation of a HyperBase. Department of Computer Science Internal Report IR 90-03, The University of Aalborg, Denmark, September.
47. Wiil, U. K., and Østerbye, K. 1990. Experiences with HyperBase – A multi-user back-end for hypertext applications with emphasis on collaboration support. Department of Computer Science Technical Report R 90-38, The University of Aalborg, Denmark, October.
48. Wiil, U. K. 1991. Using events as support for data sharing in collaborative work. International Workshop on CSCW, K. Gorling and C. Sattler, Eds., (Berlin, Germany, April), 162-176.
49. Wiil, U. K. 1991. Issues in the design of EHTS: A multiuser hypertext system for collaboration. Department of Computer Science Technical Report R 91-24, The University of Aalborg, Denmark, June.
50. Wiil, U. K. 1992. Issues in the design of EHTS: A multiuser hypertext system for collaboration. *Proceedings of the Hawaii International Conference on System Sciences (HICSS-25)*, B. Schriver, Ed., (Kauai, Hawaii, January), 629-639.
51. Woelk, D., Kim, W., and Luther, W. 1986. An object-oriented approach to multimedia databases. *Proceedings of the ACM SIGMOD '86 Conference*, (Washington, DC, May), 311-325.
52. Wuwongse, V., and Singkorapoom, S. 1991. An object-oriented data model for hypermedia databases. In *Object-Oriented Approach in Information Systems*, F. Van Assche, B. Moulin and C. Rolland, Eds., Elsevier Science Publishers B.V. (North-Holland), 403-418.
53. Yankelovich, N., Meyrowitz, N., and van Dam, A. 1985. Reading and writing the electronic book. *IEEE Computer*, 18, 10, (October), 16-30.
54. Zdonik, S. B., and Maier, D., Eds. 1990. Readings in object-oriented database systems. Morgan Kaufmann.
55. Zobel, J., Wilkinson, R., Thom, J., Mackie, E., Sacks-Davis, R., Kent, A., and Fuller M. 1991. An architecture for hyperbase systems. *Proceedings of the First Australian Multi-Media Communications, Applications & Technology Workshop*, 152-161.
56. Østerbye, K. 1992. Structural and cognitive problems in providing version control for hypertext. Submitted to the European Conference on Hypertext (ECHT '92), Milan, Italy.

APPENDIX A. HYPERFORM SPECIFICATION

The notation used for describing the behavior of methods in the built-in classes of Hyperform is of the form:

(**method** in₁, ... , in_n) ----> (out₁, ... , out_m)

Methods are invoked via the send mechanism. Sending a message to an object is done in the following way:

(**send** object message . args)

object can be either a name (if the object is a class) or an uid (instance or class). In the first case the most recent version of the class environment is used for method look-up. In other words, the uid must be used when sending a message to an old class version. The method look-up mechanism of **send** first checks whether the specified object is an instance or a class. If it is an instance, the method is looked up in the environment of the class of the instance, then in the super class environments recursively until either the method is found or the root of the class lattice is reached. If it is a class, the message is looked up in the environment of the class or in Meta Class depending on the type of the message. If the method look-up fails the error MESSAGE-NOT-UNDERSTOOD is returned; otherwise the method is invoked in the proper class environment with the given arguments.

In₁ is always *self* (the target object containing the method). *self* is assigned by the send mechanism and, therefore, should be omitted by the user. This allows messages to be sent to *self* within methods. The remaining arguments to **send** (args) are assigned to in₂ to in_n. In most cases out₁ is used to indicate whether the operation succeeded or failed. If the operation failed, out₁ will describe what went wrong. For example, the **set-class** method of Meta Class:

(**set-class** self description) ----> (OK? uid)

is invoked like this:

(**send** 'metaclass 'set-class class-description) or (**send** 1 'set-class class-description)

since Meta Class is object #1 in Hyperform. **set-class** returns an error code and the uid of the newly created class.

We start by specifying the three basic classes of the Hyperform server: Meta Class, System Object and Object and follow this with a specification of the five subclasses of Object.

A.1 Meta Class

Attributes specified by Meta Class:

class name: metaclass

system attributes: super, name, uid

super class: None

Class manipulation methods:

(**get-class** self) ----> (OK? description)

Returns the *description* of the class.

(**set-class** self description) ----> (OK? uid)

Creates a (new version of a) class specified in *description* and returns the *uid* of the class. If the class name in *description* is an unused class name a new class is created. If the class name already exists a new version of the class is created.

(**delete-class** self) ----> (OK?)

Deletes the class, all its subclasses and all instances of the class and subclasses.

(**create-instance** self) ----> (OK? uid)

Creates an instance of the class and returns the *uid* of the new instance.

Meta Class is object #1 in Hyperform. Class descriptions must obey the following syntax:

((**class-name** <name>) <options>)

name can be any Scheme symbol and *options* can be of the form:

(**read-only-att** <attribute list>) or

(**read-write-att** <attribute list>) or

(**calculated-att** <attribute list>) or

(**super-class** <class name list>) or

(**methods** <method definitions>)

attribute list can be either a Scheme symbol (the name of the attribute) or of the form: (symbol initializer) . *class name list* is a list of Scheme symbols (class names or uid's). *method definitions* are definitions of Scheme functions, which can be interfaces to loaded object code or functions entirely written in Scheme. *super-class* is default set to *object*. All options in a class description can be redefined with **set-class**.

A.2 System Object

Attributes specified by System Object:

class name: system

read-write attributes: users, super-users, groups, max-instance-size, cache-limit

super class: object

Session controlling methods:

(**connect** self user) ----> (OK?)

Initiates a session if *user* has proper access rights. A user can only have one session from each internet address (machine and socket).

(disconnect self) ----> (OK?)

Closes an open session.

User manipulation methods:

(get-user self user) ----> (OK? permission)

Returns the class manipulation permissions of *user*, which are of the form "gsd" (get, set, delete) corresponding to three meta class methods. "g__" means that the user can get, but not set and delete class descriptions.

(set-user self user permission) ----> (OK?)

Inserts a user name and permissions into the list of valid users. Also used to change permissions of a user.

(delete-user self user) ----> (OK?)

Deletes a user from the list of valid users.

(user? self user) ----> (true / false)

Returns true if *user* is a valid user, else false.

Super user manipulation methods:

(get-super-users self) ----> (OK? list)

Returns a list of all super users.

(set-super-user self user) ----> (OK?)

Inserts *user* into the list of super users.

(delete-super-user self user) ----> (OK?)

Deletes *user* from the list of super users.

(super-user? self user) ----> (true / false)

Returns true if *user* is a super user, else false.

Class permission methods:

(get-permission? self user) ----> (OK?)

Returns true if *user* has permission to use **get-class**, else false.

(set-permission? self user) ----> (OK?)

Returns true if *user* has permission to use **set-class**, else false.

(delete-permission? self user) ----> (OK?)

Returns true if *user* has permission to use **delete-class**, else false.

Group manipulation methods:

(get-group self group) ----> (OK? users)

Returns a list of the members in *group*.

(create-group self group users) ----> (OK?)

Creates a group with the specified users. *group* must be a string and *users* must be a list of strings.

(delete-group self group) ----> (OK?)

Deletes *group*.

(**add-to-group** self group user) ----> (OK?)

Adds *user* to *group*.

(**remove-from-group** self group user) ----> (OK?)

Removes *user* from *group*.

(**group-member?** self group user) ----> (true / false)

Returns true if *user* is a member of *group*, else false.

(**group?** self group) ----> (true / false)

Returns true if *group* exists, else false.

Cache manipulation methods:

(**set-cache-limit** self limit) ----> (OK?)

Sets the maximum number of instances allowed in the cache. Default is 100.

(**set-max-instance-size** self size) ----> (OK?)

Sets the allowed maximum size of instances in the cache. Default is 3000 bytes. Instances larger than the maximum size will not be cached. The cache uses the heap in ELK, which has a fixed maximum size (currently 1MB). ELK has to be recompiled to change the heap size. A too large cache will cause Hyperform (ELK) to stop with the error message: out-of-heap-space.

System Object is object #3 in Hyperform. System Object will always have exactly one instance, which is object #9. In other words, object #9 keeps system information and provides methods to maintain the information. Only super users can change the information kept by System Object. When Hyperform is first started all users have access to the system and all users are super users, because Hyperform interpretes an empty *users* and *super-users* this way. If the lists become non-empty, only users in the *user* list are allowed to use Hyperform and users in the *super-users* list are allowed to change system information.

A.3 Object

Attributes specified by the Object class:

class name: object

system attributes: uid, class-uid

super class: None

Instance methods:

(**get-instance** self) ----> (OK? instance)

Returns the contents of *instance*.

(**delete-instance** self) ----> (OK?)

Deletes the instance.

Attribute manipulation methods:

(**get-attribute** self attribute) ----> (OK? contents)

Returns the *contents* of *attribute* in instance.

(**set-attribute** self attribute contents) ----> (OK?)

Writes *contents* to *attribute* in instance. If *attribute* does not exist, it will be created. Read-only attributes cannot be written.

(**delete-attribute** self attribute) ----> (OK?)

Deletes *attribute* in instance. Only dynamic attributes can be deleted.

The Object class is object #2 in Hyperform.

A.4 Concurrency Control Object

Attributes specified by Concurrency Control Object:

class name: CC-object

super class: object

Lock methods:

(**lock** self attribute) ----> (OK?)

Locks *attribute* in instance. If attribute name is 'ALL', the whole instance will be locked.

(**unlock** self attribute) ----> (OK?)

Unlocks *attribute* in instance or whole instance if attribute name is 'ALL'.

(**who-locks?** self attribute) ----> (OK? user)

Returns the name of the *user* holding a lock on *attribute*. Again attribute can be 'ALL'.

(**instance-lock?** self) ----> (true / false)

Returns true if the user is allowed to lock the entire instance, else false. Can be used to check for locks.

(**attribute-lock?** self attribute) ----> (true / false)

Returns true if the user is allowed to lock *attribute* (or holds a lock on the attribute) in instance, else false. Can be used to check for locks.

Transaction mechanism methods:

(**start** self) ----> (OK?)

Starts a transaction. Each user can only have one transaction running.

(**commit** self) ----> (OK?)

Ends a transaction by evaluating all expressions in the transaction and releasing all locks.

(**abort** self) ----> (OK?)

Ends a transaction by aborting all expressions in the transaction and releasing all locks.

Concurrency Control Object is object #4 in Hyperform.

A.5 Notification Control Object

Attributes specified by Notification Control Object:

class name: NC-object

super class: object

Event methods:

(**event** self expression) ----> (OK? event-id)

Subscribes to an event specified in *expression*. Returns a unique identifier for the event.

(**cancel-event** self event-id) ----> (OK?)

Cancels an event specified by the unique *event-id*.

The method (**send-events**) is globally defined; it can be invoked anywhere without the use of *send*:

(**send-events** object method attribute) ----> (OK?)

Evaluates all event expressions and transmits events based on the results. The three arguments (object, method and attribute) are used to trigger events. Event expressions evaluating to anything but false causes an event to be transmitted. The event contains the event identifier and the results of the evaluated expression (which has access to all internal lists and variables of Hyperform).

Notification Control Object is object #5 in Hyperform.

A.6 Access Control Object

Attributes specified by Access Control Object:

class name: AC-object

read-write attributes: owner, created, modified-by, modified, permission, group

super class: object

Object initialization method:

(**AC-init** self . args) ----> (OK?)

Initializes attributes (see above) maintained by AC Object. The values for *permission* and *group* can be specified in *args* (*group* must be specified in System Object). If the attributes are not specified, default values will be used: *permission* = "gsdg_g__" and *group* = *owner*.

Modification methods:

(**change-permission** self permission) ----> (OK?)

Changes the *permission* of the object. Permissions are of the form "gsdgsdgsd".

(**change-group** self group) ----> (OK?)

Changes the *group* of the object. The group must be a group specified in System Object.

Access controlling methods:

(**get-permission?** self) ----> (OK?)

Returns "(OK)" if the user has permission to get (read) the object.

(**set-permission?** self) ----> (OK?)

Returns "(OK)" if the user has permission to set (write) the object.

(**delete-permission?** self) ----> (OK?)

Returns "(OK)" if the user has permission to delete the object.

Access Control Object is object #6 in Hyperform.

A.7 Version Control Object

Attributes specified by Version Control Object:

class name: VC-object

read-write attributes: versiongroup, version, previous, next, variants

super class: object

Object initialization method:

(**VC-init** self) ----> (OK?)

Initializes attributes (see above) maintained by VC Object.

Version Control methods:

(**check-out** self type . arg) ----> (OK? uid)

Checks out a new version of the object. Three types can be specified: 'revision' 'newrelease' 'variant'. If the new version is a new release, a version number can be specified in *arg*. Returns the uid of the new version.

(**check-in** self delta) ----> (OK?)

Checks in a previous checked out version. If *delta* is true, the previous version will be saved as a backward delta. If *delta* is false, the previous version will be saved as a complete version.

(**destroy-version-group** self) ----> (OK?)

Deletes old versions and variants in a version group. Only the latest version of the main branch of the version tree will be kept (the object that named the versiongroup).

(**make-delta** self) ----> (OK?)

Changes a complete version into a backward delta version.

(make-complete self) ----> (OK?)

Changes a delta version into a complete version.

(get-version self) ----> (instance)

Returns the contents of *instance*. If the instance is a delta version, the complete version will be collected.

(get-version-tree self) ----> (list)

Returns the version tree in preorder as a *list* of uid's.

(copy self) ----> (OK? uid)

Creates a copy of an instance (used in check-out). Returns the *uid* of the copy.

Status review methods:

(set-accessible? self) ----> (true / false)

Returns true if the user is allowed to change the object, else false. Only the latest version of each branch in the tree should be changed, all other versions should be frozen.

(delta? self) ----> (true / false)

Returns true if the object is a delta version, else false.

Version Control Object is object #8 in Hyperform.

A.8 Query and Search Object

Attributes specified by Query and Search Object:

class name: QS-object

super class: object

Query methods:

(get-all self type) ----> (list)

Returns a list of all objects of the specified *type*. *type* can be: 'instance', 'object', 'class' or 'cache'.

(union self list1 list2) ----> (list)

Returns the union of the first and second list.

(intersect self list1 list2) ----> (list)

Returns the intersection of the first and second list.

(subtract self list1 list2) ----> (list)

Returns the elements of the first list that are not in the second.

(top self number list) ----> (list)

Returns the first *number* elements of *list*.

(match self type value class-list) ----> (list)

Class filtering function. *type* specifies the type of information filtered and can be either: att (all attributes), read-only, read-write, calculated, method, name, descendant, ancestor, super or sub. Uid's of classes in *class-list* matching the value when filtering against the specified type are returned in *list*. Simplified examples:

(match 'name 'AC-object (get-all 'class))

Find all classes with the name AC-object (same as return all versions of AC Object).

(match 'sub 2 (get-all 'class))

Find all subclasses of the Object class (the Object class is object #2 in Hyperform).

(match 'read-write 'owner (get-all 'class))

Find all classes specifying a read-write attribute named owner.

(filter self type attributes value exact instance-list) ----> (list)

Instance filtering function. Again *type* specifies the type of information filtered and can be either: att (all attributes), read-only, read-write, calculated, dynamic or class. This function returns uid's of instances in *instance-list* that contain either specific attributes (void value argument), attributes with specific values or specific values (void attribute argument). The function is capable of either exact match (exact = true) or member match (exact = false). In the first case the values have to be identical to match, and in the latter the value just have to be a member of a list to match. Simplified examples:

(filter 'att 'owner "kock" #t (get-all 'instance))

Find all instances with an attribute named *owner* with the exact value "kock".

(filter 'read-write 'reference 10 #f (get-all 'instance))

Find all instances with a read-write attribute named *reference* containing the number 10.

(filter 'class '() 15 #t (get-all 'instance))

Find all instances of class number 15.

Query and Search Object is object #7 in Hyperform.

APPENDIX B. HYPERFORM BASICS

This appendix contains a description of practical things application programmers should know about Hyperform in order to use the platform in hyperbase and hypertext system development, such as in the EHTS Editor experiment. We start by describing how the server and tool integrator can be created and initiated. We then mention global lists and variables of Hyperform which can be useful in tailoring the event and query mechanisms. Application programmers can retrieve information on the internal state of the server, such as which users are connected to the server and where they are located in the network and which (parts of) objects are locked. Finally, we list all the source files of the Hyperform architecture.

B.1 Running the Hyperform Architecture

The Hyperform architecture consists of the Hyperform server and tool integrators, as shown in Figure 2. Hyperform has been interfaced from GNU Emacs (and Epoch) as a part of the EHTS Editor porting experiment. The server is a single program running inside the ELK Scheme interpreter. The server saves its objects in two containers, named *hyperform.data* and *hyperform.index*. If these two files are not present in the directory where the server is started, two new files will be created containing descriptions of the built-in classes. The server can be started with option *-hb*:

```
Hyperform -hb
```

in which case debug information is listed to the terminal window. To generate a log, write the debug information to a file:

```
Hyperform -hb > ../path/./log-file
```

Normally, Hyperform is initiated as a (background) proces in a terminal window:

```
Hyperform (&)
```

The server program is created by invoking the ELK Scheme interpreter and using two commands:

```
> (load 'lowlevel.scm)
> (dump-hyperform)
```

The first command loads all the C object code and Scheme source code into the interpreter and the second dumps an image of the interpreter, named Hyperform, which can be invoked as a stand-alone program. The best way to stop the server is to use CTRL-C in the terminal window to interrupt the program. Hyperform catches the interrupt, closes the files and terminates. When the server is running, different clients can be started: tool integrators, Epoch clients and EHTS Editors. Likewise, the tool integrator program is created by invoking the ELK Scheme interpreter and using two commands:

```
> (load 'integrator.scm)
> (dump-integrator)
```

which creates the binary named: Integrator.

B.2 Internal Lists and Variables

The built-in classes and basic mechanisms of Hyperform maintain a number of internal lists and variables. These can be used by the event mechanism to generate events and as return values from events. The lists can also be used as basis for queries, such as: list all users connected to the system or list all users having a transaction running. The internal lists and variables are listed after the source code file that defines them:

<u>Source File</u>	<u>Lists and Variables</u>	<u>Explanation</u>
Hyperform Server		
access.scm:	None	
event.scm:	NC-entity, NC-operation, NC-attribute event-list event-send-list event-id	(to generate events - updated by (send-events)) (event-id (user machine socket) expression) (event-id (user machine socket) result))
loadclass.scm:	class-name-alist class-env-alist class-ancestor-alist class-super-alist class-sub-alist instance-alist	(class name . class uid) (class uid . class environment) (class uid . ancestor class uid) (class uid . super class uid's) (class uid . sub class uid's) (instance uid . class uid)
lock.scm	lock-list transaction-list	(uid attribute user) (user (locks) (messages))
lowlevel.scm:	hb-debug-on	
meta.scm:	None	
object.scm	None	
query.scm:	None	
send.scm:	None	
server.scm:	user, machine, socket, message-id, message user-list	(user machine socket)
shell.scm:	None	
string-object.scm:	None	
system.scm:	max-instance-size, cache-limit, cache-size cache-alist	(instance uid . instance)
version.scm:	check-out-list	(uid type user)
Tool Integrator		
integrator.scm:	socket, machine, user, message-id server, address	(tool integrator) (of Hyperform server)

B.3 Source Code

This section lists the different source code files of the Hyperform server and tool integrator, the amount of source code in the files and their contents.

<u>Source File</u>	<u>Amount</u>	<u>Contents</u>
Hyperform Server		
access.scm:	7 KByte	AC-object
events.scm:	4 KByte	NC-object
loadclass.scm:	6 KByte	load objects (create class environments and internal lists)
lock.scm:	6 KByte	CC-object
lowlevel.scm:	7 KByte	start-up mechanism
meta.scm:	12 KByte	metaclass
object.scm:	7 KByte	object
query.scm	6 KByte	QS-object
send.scm:	4 KByte	send mechanism
server.scm:	3 KByte	server loop
shell.scm:	1 KByte	shell operations
string-object.scm:	3 KByte	conversions
system.scm:	14 KByte	system
version.scm:	12 KByte	VC-object
file.c:	67 KByte	file system interface
network.c:	7 KByte	network interface
shell.c:	1 KByte	shell interface
<u>TOTAL</u>	<u>167 KByte</u>	
Tool Integrator		
integrator.scm:	4 KByte	intergrator
shell.scm:	1 KByte	shell operations
string-object.scm:	3 KByte	conversions
network.c:	7 KByte	network interface
shell.c:	1 KByte	shell interface
<u>TOTAL</u>	<u>16 KByte</u>	

APPENDIX C. CLASS DESCRIPTIONS FROM HYPERBASE SIMULATION

Appendix C contains descriptions for the three classes used in the HyperBase data model simulation, the Entity, Node and Link classes.

C.1 Entity Class Description

```
((class-name entity)
 (read-write-att name createdby createddate lastmodifiedby lastmodifieddate)
 (super-class CC-object NC-object QS-object)
 (methods

 (define (Enumerate self type)
 (case type
 ((node link)
 (define number (cadr (assoc type class-name-alist)))
 (send-events 0 'Enumerate 0)
 (list 0 (send self 'filter 'class '()) number #t
 (send 'QS-object 'get-all 'instance))))
 (else (list -1))))

 (define HB-event-id-list '())

 (define (Event self ent ope key)
 (if (equal? ent 'ALL) (define allent #t) (define allent #f))
 (if (equal? ope 'ALL) (define allope #t) (define allope #f))
 (if (equal? key 'ALL) (define allkey #t) (define allkey #f))
 (define exp (lambda ()
 (if (and (or (equal? NC-entity ent) allent (eq? 0 ent))
 (or (equal? NC-operation ope) allope (eq? 0 ope))
 (or (equal? NC-attribute key) allkey (eq? 0 key)))
 (list user NC-entity NC-operation NC-attribute))))
 (define ID (cadr (send 'NC-object 'event exp)))
 (set! HB-event-id-list (cons (list ID user ent ope key) HB-event-id-list))
 (send-events ent 'Event key)
 (list 0))
```

```
(define (UnEvent self ent ope key)
  (define ID 0)
  (do ((a HB-event-id-list (cdr a))) ((null? a))
    (if (and (equal? user (cxr a "ada"))
              (equal? ent (cxr a "adda"))
              (equal? ope (cxr a "addda"))
              (equal? key (cxr a "adddda"))))
      (begin
        (set! ID (caar a))
        (set! a '(stop))))))
  (send 'NC-object 'cancel-event ID)
  (set! HB-event-id-list (remove-from-alist ID HB-event-id-list))
  (send-events ent 'UnEvent key)
  (list 0))

(define (ShowEvent self ent ope key)
  (define users '())
  (do ((a HB-event-id-list (cdr a))) ((null? a))
    (if (and (equal? ent (cxr a "adda"))
              (equal? ope (cxr a "addda"))
              (equal? key (cxr a "adddda"))))
      (set! users (cons (cadar a) users))))
  (send-events ent 'ShowEvent key)
  (list 0 users))

(define (Lock self key)
  (define return (car (send-super self 'lock key)))
  (cond
    ((equal? return "OK")
     (send-events self 'Lock key)
     (list 0))
    ((or (equal? return "ERROR:lock:LOCKED-BY-YOU")
          (equal? return "ERROR:lock:LOCKED-BY-OTHER-USER")
          (equal? return "ERROR:lock:CANNOT-LOCK-ALL"))
     (list 350))
    ((or (equal? return "ERROR:lock:OBJECT-IS-A-CLASS")
          (equal? return "ERROR:lock:NO-SUCH-OBJECT")
          (equal? return "ERROR:lock:NO-SUCH-ATTRIBUTE"))
     (list -1))))
```

```
(define (UnLock self key)
  (define return (car (send-super self 'unlock key)))
  (cond
    ((equal? return "OK")
     (send-events self 'UnLock key)
     (list 0))
    ((equal? return "ERROR:unlock:NOT-LOCKED")
     (list 352))
    ((equal? return "ERROR:unlock:NOT-OWNER-OF-LOCK")
     (list 351))))
```

```
(define (ShowLock self key)
  (define return (send-super self 'who-locks? key))
  (cond
    ((equal? (car return) "OK")
     (send-events self 'Showlock key)
     (list 0 (cadr return)))
    ((equal? (car return) "ERROR:who-locks?:NOT-LOCKED")
     (list 352))))
```

```
.....
;; Extra operations to speed things
.....
```

```
(define (Get-all-node-names self)
  (define outlist '())
  (define node-list (cadr (Enumerate self 'node)))
  (do ((a node-list (cdr a)) ((null? a)
    (define name (cadr (send (car a) 'get-attribute 'name)))
    (set! outlist (cons (cons name (car a)) outlist)))
    outlist)
```

```
(define (Get-all-link-names self node)
  (define outlist '())
  (define link-list (cadr (send node 'get-attribute 'linknum)))
  (do ((a link-list (cdr a)) ((null? a)
    (define name (cadr (send (car a) 'get-attribute 'name)))
    (set! outlist (cons (cons name (car a)) outlist)))
    outlist)))
```

C.2 Node Class Description

```
((class-name node)
(read-write-att (linkstome 0) linknum (data "") font geometry)
(calculated-att (size (string-length (cadr (send-super self 'get-attribute 'data))))))
(super-class entity)
(methods

(define (CreateNode self)
  (define newnode (cadr (send-super self 'create-instance)))
  (send-events newnode 'CreateNode 0)
  (list 0 newnode))

(define (Delete self)
  (cond
    ((not (send-super self 'instance-lock?))
     (list 350))
    ((not (= 0 (cadr (send-super self 'get-attribute 'linkstome))))
     (list -204))
    (else
     (define linknum (cadr (send-super self 'get-attribute 'linknum)))
     (do ((a linknum (cdr a))) ((null? a))
       (define usecount (cadr (send-super (car a) 'get-attribute 'usecount)))
       (set! usecount (- usecount 1))
       (send-super (car a) 'set-attribute 'usecount usecount))
     (send-super self 'delete-instance)
     (send-events self 'Delete 0)
     (list 0))))

(define (UseLink self link)
  (define linknum (cadr (send-super self 'get-attribute 'linknum)))
  (set! linknum (append linknum (list link)))
  (send-super self 'set-attribute 'linknum linknum)
  (define usecount (cadr (send-super link 'get-attribute 'usecount)))
  (set! usecount (+ usecount 1))
  (send-super link 'set-attribute 'usecount usecount)
  (send-events self 'UseLink 0)
  (list 0))
```

```
(define (RemoveLink self link)
  (define linknum (cadr (send-super self 'get-attribute 'linknum)))
  (set! linknum (remove-from-list link linknum))
  (send-super self 'set-attribute 'linknum linknum)
  (define usecount (cadr (send-super link 'get-attribute 'usecount)))
  (set! usecount (- usecount 1))
  (send-super link 'set-attribute 'usecount usecount)
  (send-events self 'RemoveLink 0)
  (list 0))

(define (Read self key)
  (case key
    ((linkstome linknum data size font geometry name createdby createddate lastmodifiedby lastmodifieddate)
     (define return 0)
     (define person (locked? self key))
     (if (and person (not (equal? person user)))
         (set! return 350)
         (send-events self 'Read key)
         (list return (cadr (send-super self 'get-attribute key))))
    (else
     (list 201))))

(define (Write self key value)
  (case key
    ((linkstome linknum size)
     (list 208))
    ((data font geometry name createdby createddate lastmodifiedby lastmodifieddate)
     (define return 0)
     (define person (locked? self key))
     (if (and person (not (equal? person user)))
         (set! return 350)
         (begin
          (send-super self 'set-attribute key value)
          (send-events self 'Write key)))
     (list return))
    (else
     (list 201))))
```

```
.....  
.....  
;; Extra operations to speed things  
.....  
.....  
  
(define (CreateInitNode self name person date font geometry)  
  (define newnode (cadr (CreateNode self)))  
  (Write newnode 'name name)  
  (Write newnode 'createdby person)  
  (Write newnode 'createddate date)  
  (Write newnode 'lastmodifiedby person)  
  (Write newnode 'lastmodifieddate date)  
  (Write newnode 'font font)  
  (Write newnode 'geometry geometry)  
  newnode)  
  
(define (UpdateNodeData self data person date geometry)  
  (Write self 'data data)  
  (Write self 'lastmodifiedby person)  
  (Write self 'lastmodifieddate date)  
  (Write self 'geometry geometry))  
  
(define (UpdateNodeName self name person date)  
  (Write self 'name name)  
  (Write self 'lastmodifiedby person)  
  (Write self 'lastmodifieddate date))  
  
(define (UpdateNodeFont self font person date)  
  (Write self 'font font)  
  (Write self 'lastmodifiedby person)  
  (Write self 'lastmodifieddate date))  
  
(define (UpdateNodeGeom self geometry person date)  
  (Write self 'geometry geometry)  
  (Write self 'lastmodifiedby person)  
  (Write self 'lastmodifieddate date))  
  
(define (ReadNodeBuffer self)  
  (define font (cadr (Read self 'font)))
```

```
(define geometry (cadr (Read self 'geometry)))  
(list font geometry))
```

```
(define (ReadAllAtt self)  
  (define node (cadr (send self 'get-instance)))  
  (define size (cadr (send self 'get-attribute 'size)))  
  (define attlist (list (list 'size size)))  
  (set! attlist (append attlist (caddr node)))  
  (set! attlist (remove-from-alist 'data attlist))  
  (remove-from-alist 'linknum attlist))))
```

C.3 Link Class Description

```
((class-name link)  
(read-write-att (usecount 0) tonode)  
(super-class entity)  
(methods
```

```
(define (CreateLink self node)  
  (define link (cadr (send-super self 'create-instance)))  
  (send-super link 'set-attribute 'tonode node)  
  (define temp (cadr (send-super node 'get-attribute 'linkstome)))  
  (set! temp (+ temp 1))  
  (send-super node 'set-attribute 'linkstome temp)  
  (send-events link 'CreateLink 0)  
  (list 0 link))
```

```
(define (Delete self)  
  (cond  
    ((not (send-super self 'instance-lock?))  
     (list 350))  
    ((not (= 0 (cadr (send-super self 'get-attribute 'usecount))))  
     (list -204))  
    (else  
     (define tonode (cadr (send-super self 'get-attribute 'tonode)))  
     (define linkstome (cadr (send-super tonode 'get-attribute 'linkstome)))  
     (set! linkstome (- linkstome 1))  
     (send-super tonode 'set-attribute 'linkstome linkstome)
```

```
(send-super self 'delete-instance)
(send-events self 'Delete 0)
(list 0))))
```

```
(define (MoveLink self node)
  (define oldnode (cadr (send-super self 'get-attribute 'tonode)))
  (define linkstome (cadr (send-super oldnode 'get-attribute 'linkstome)))
  (set! linkstome (- linkstome 1))
  (send-super oldnode 'set-attribute 'linkstome linkstome)
  (set! linkstome (cadr (send-super node 'get-attribute 'linkstome)))
  (set! linkstome (+ linkstome 1))
  (send-super node 'set-attribute 'linkstome linkstome)
  (send-super self 'set-attribute 'tonode node)
  (send-events self 'MoveLink 0)
  (list 0))
```

```
(define (Read self key)
  (case key
    ((usecount tonode name createdby createddate lastmodifiedby lastmodifieddate)
     (define return 0)
     (define person (locked? self key))
     (if (and person (not (equal? person user)))
         (set! return 350)
         (send-events self 'Read key)
         (list return (cadr (send-super self 'get-attribute key))))
    (else
     (list 201))))
```

```
(define (Write self key value)
  (case key
    ((usecount tonode)
     (list 208))
    ((name createdby createddate lastmodifiedby lastmodifieddate)
     (define return 0)
     (define person (locked? self key))
     (if (and person (not (equal? person user)))
         (set! return 350)
         (begin
```

```
(send-super self 'set-attribute key value)
(send-events self 'Write key)))
(list return))
(else
(list 201))))

:.....
;; Extra operations to speed things
:.....

(define (CreateInitLink self tonode fromnode name person date)
  (define newlink (cadr (CreateLink self tonode)))
  (send fromnode 'UseLink newlink)
  (Write newlink 'name name)
  (Write newlink 'createdby person)
  (Write newlink 'createddate date)
  (Write newlink 'lastmodifiedby person)
  (Write newlink 'lastmodifieddate date)
  newlink)

(define (UpdateLinkMove self node person date)
  (MoveLink self node)
  (Write self 'lastmodifiedby person)
  (Write self 'lastmodifieddate date))

(define (UpdateLinkName self name person date)
  (Write self 'name name)
  (Write self 'lastmodifiedby person)
  (Write self 'lastmodifieddate date))

(define (DeleteAll self list1)
  (do ((a list1 (cdr a)) ((null? a)
    (Delete (car a))))))

(define (ReadAllAtt self)
  (cxr (send self 'get-instance "addad"))))
```