Mantaining a command history in LINCKS

Thomas Hall

Dept. of Computer and Information Science
at
The University of Linköping
S-581 83 Linköping, Sweden

Reg. no: LiTH-IDA-Ex-9221

Examiner:	**Lin Padgham**	**Linköping 1992-06-03**
Supervisor:	**Lin Padgham**	

Abstract

In this report we develop and implement a partially ordered command history for the multi-user LINCKS system. We examine two approaches of how to represent the temporal order in the command history (forward vs backward linking). The backward linking is chosen, because it results in a faster update of the command history and also contains the forward links indirectly. We define two types of commands: workspace commands and database commands. Workspace commands only affect objects in the workspace and database commands store or retrieve objects from the database. The different user sessions (LOGIN to LOGOUT) are grouped together in session objects. Temporal ordering between sessions exists as a temporal ordering between database commands of different sessions. This ordering is represented through backward links in the commands. A query function is implemented. The function takes two commands (A and B) and determines the temporal order between them.

Acknowledgement

I would like to express my sincerest thanks to my supervisor Lin Padgham for her guidance and help and for allowing me so much freedom in the project. I also want to thank Ralph Rönnquist for comments and proposals of improvement on draft versions of this report. Finally I want to thank the IISLAB group for an interesting and creative atmosphere in which to work, and each member for taking the time to answer my more or less stupid questions.

Linköping, May 1992 Thomas Hall

Contents

CHAPTER 1 Introduction

LINCKS stands for Linköping's Intelligent Communication of Knowledge system. The core of this system is a passive object oriented database. It is chosen to be object oriented, because that is a natural way to represent information in the real world. The system of today includes an object oriented database with some history functions.

The aim is to develop an intelligent database, information base, knowledge base system that is more appropriate to use as a database for AI-applications than most of the systems of today. Traditional database system are usually constructed to handle large amounts of data, which is also usually relatively simply structured. Traditional database systems are not so suitable for AI processing, because the information dealt with is more complex with less emphasis on repetitive information. Also, information about how the database has evolved should/must be stored, e.g. to allow the system to reason about its own behavior and improve it.

The LINCKS system has been developed to address and solve these problems. The system is intended as a research system to be used as a test bed for different AI applications. The project documented in this paper, my final project, was to extend the LINCKS system with command history management.

Much work has been done to incorporate handling of temporal information in databases.There are essentially two kinds of temporal information we want to be able to handle, temporal information about the database and temporal information about the data that are kept in the database. One example of temporal information about the database is the history of how the contents have changed over time. The time when a contents change occurs is defined by [Sno86] as the transaction time. The command history can be seen as an extension of the transaction history, since we keep track of not only how the database evolves but also of what operations were executed to change the contents and by whom.

Temporal information about the data in the database are out of the scope of this report.

1.1 The task

This my final project has been to develop and implement a command history functionality. With a command history we mean a partial order relation of commands. The command history logs the calls to some of the interface functions of the database, and stores them in a history structure. One part of the task was to determine the set of commands to log (initially) in the command history. A second part was to develop and implement a function that takes two commands in the command history and decides the temporal order between them. The history is stored as ordinary objects in the database, which demands some extra precautions to avoid unauthorized editing of the command history. The update of the command history has to be both time and space effective; time effective, since the user does not want to wait for response, and space effective, since there is not an unlimited amount of space available.

1.2 Some applications

In this section we describe some possible ways to use the command history. This has not been a major goal with the final project, but it is interesting to have some motivation and ideas on how the command history might be used.

1.2.1 Find special version

An obvious use is if you want to find some special version, or object, that has certain temporal properties. For instance, if you (say after a holiday) want to start where you left of, you ask for the objects that were in your workspace when you logged out last Friday. This can easily be done with the command history. If you do not have a command history, the query can not be answered even with a full search in the database, since you can not determine if an object was in your workspace or not.

1.2.2 Composite objects temporal history

In LINCKS each atomic object has a temporal history which is a partial order over versions of the object as it has existed in the past. A common way to work with atomic objects is to build composite objects with them.

One problem with composite objects is to determine from which component object versions the different composite object versions were built. For instance, consider a book that has two chapters,1 and 2, see figure 1. I call version 1 of chapter 1 for object 11 etc. Suppose now that A retrieves the composite object book and starts to edit chapter one. He does not change anything in chapter 2. Before user A stores the book, user B retrieves the book, i.e objects 11 and 21, and starts to edit chapter 2. If they both now store their changes back to the database, there should be two versions of the composite object book; one with the chapters 12 and 21 (from user A) and a second with chapter 11 and 22. However, the next time someone retrieves the book, he will get chapters 12 and 22, which is a merged version of the two versions. It might be interesting to know that the version you get is really a merge of several versions.

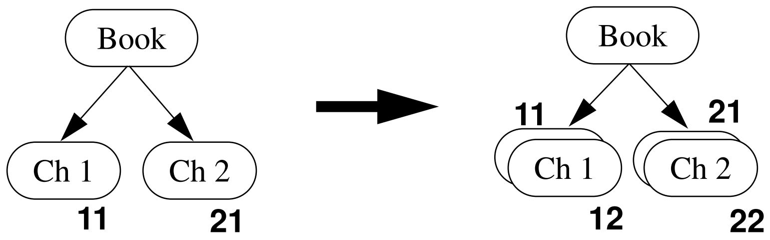


FIGURE 1 Parallel editing of composite object

This information can be found by using the command history. We can follow the link from the database object to find the command in the command history that stored the composite object, see figure 2. We can then observe what parts of the composite object were stored (and therefore changed) for each composite store. We can then easily see that 12 and 22 were not stored as part of the same composite store and thus the composite version containing both changes is really a merge of two versions.

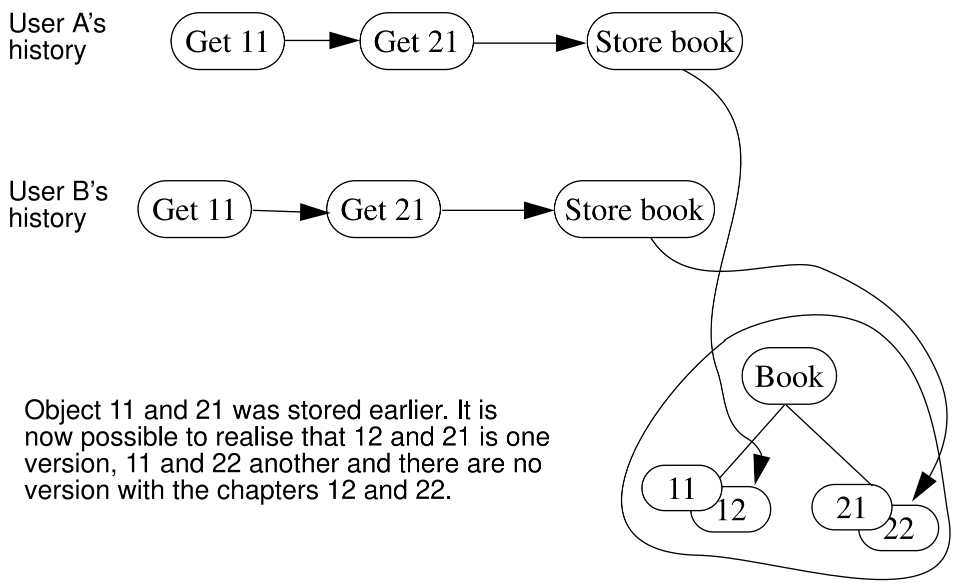


FIGURE 2 How to group parts of composite objects together.

It is thus possible to extract the temporal history for the composite object. This is however an inefficient way to get the temporal history of a composite object. In the

LINCKS system it is handled in another way, see [Lam91]. However with the command history, one can also compare the temporal histories of different composite objects.

1.2.3 Reset of the database/Rollback

Another, maybe more pragmatic use of the command history is for restoring the database to a previous state, for instance after a crash. Again one problem is in the storage of composite objects. If you reset the database to a state where only some of the low level store commands, of the composite store, have occurred, you end up with a database in an inconsistent state, i.e. only a part of the composite object has been stored in the database. This situation can be avoided with the command history by making a cut in the command history such that no composite commands are cut, see figure 3.

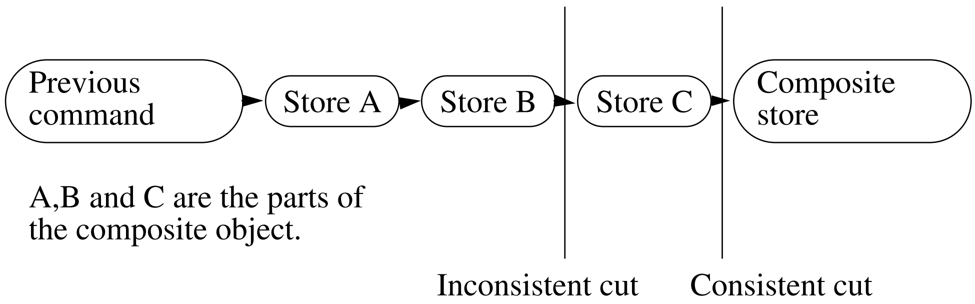


FIGURE 3 Avoiding inconsistent state with rollback.

It is also possible to use the command history for resetting the database to a state that has some property, for instance ‘reset the database to the state it had last friday, when I had stored the first version of the composite object X’. This is a special case of rollback. A rollback, according to [Sno86], is that you answer a question with the information the system contained at a time point in the past. If you after the rollback remove all information that has been added later you get a reset of the database.

CHAPTER 2 The LINCKS system

In this section I describe the LINCKS system that exists today. It is meant as a brief introduction to the system and the terminology used in this report.

2.1 The objects, nodes and versions

The node is the basic information structure in the LINCKS system. It has three parts:

Attributes: a number of name-value pairs. The attributes are understood as properties of the object independent of its relations to other objects.

Links: a number of name-pointer pairs. The links describe relationships of the object to other objects.

Image: is uninterpreted information. Usually the image contains text or graphics, typically in the size of a paragraph.

Since the system keeps track of how the database evolves over time, there are normally several versions of objects. For instance, let the object ABC123 represent the car that has the license number ABC123 and is owned by Anders. If we want to store the fact that Anders sells his car to Mikael, we can do that by creating a new version of the object ABC123 with Mikael as owner.

By keeping multiple versions of the same object, we are able to ask temporal questions about the object. In order to manage this temporal information each atomic object is actually a structure of *version nodes* (sometimes referred to simply as *versions*). The handle to the structure of versions is the *history structure node* (or *hs node*). Each atomic object has one hs node and some number of version nodes.

2.2 History functions

There are three kinds of history functions in the LINCKS system. Two of them are already implemented: the *temporal history* and the *edit history.* The third is the *command history* that is the topic of this report.

Temporal history is the history of how one object has developed over time. This gives the possibility to reconstruct an object as it was at a specific time in the past. It also makes it possible to answer questions like: What was the latest version of document X that was sent to Anders.

Edit history is the history of what version was used as a template to construct the new version of the object.

In the *command history* we keep information about the executed commands and the temporal order between the commands. From this, we can create snapshots of the database at any time point in the past and answer temporal questions, such as 'Did user α know that user β had executed command X when he executed command Y?'.

The temporal history is implemented by having the hs node as a handle for all versions of an object. This handle contains a directed acyclic graph of links to the versions.

The edit history is implemented as a link 'parent' from the version to the template/parent version.

The command history is implemented with a global structure of command objects. The commands in a users *session* (a sequence of commands that starts with a login and ends with a logout) are grouped together because it is a natural composition. Between the command objects there are links to reflect the temporal order.

2.3 Labels and Links

The nodes get unique numbers when they are brought to the workspace. By using the node numbers, we identify an object with the pair (x,y), where x represents the *hs node* and y the *version node*. Such pairs are used as references to objects and are called labels.

When we want to store a link from object A to object B, we store the label for the object B in object A, under some appropriate name.

In the LINCKS system there are two kinds of links (or labels): *bound links* and *unbound links.* A bound link is a link to a specific version of an object, i.e a link to an object with reference to time. An unbound link is a link to an object without reference to time, that is the version node part of the label has no value. In this case when the link is followed the version obtained is the current version of the object.

2.4 Atomic objects - composite objects

So far we have described what we call *atomic objects*. It is also common to use the atomic objects as building blocks for *composite objects*. For instance, a document consists of chapters, that in their turn consist of paragraphs, as illustrated in figure 4.

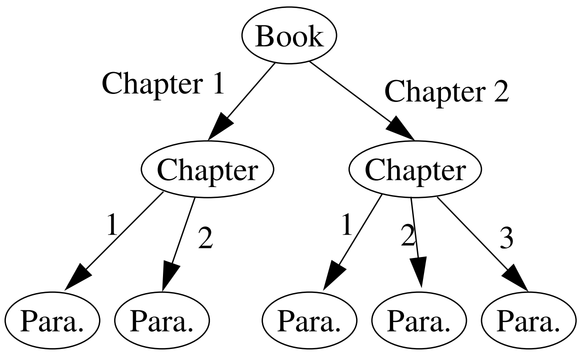


FIGURE 4 An example of a composite object.

It is possible for composite objects to share parts with other composite objects. For instance there could be two books that share some chapters, as in figure 5. This sharing of parts may result in a change in one composite object also affecting another composite object.

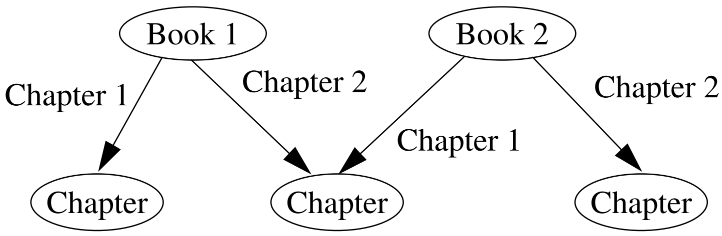


FIGURE 5 An example of sharing between composite objects.

2.5 Run-Time architecture

To support parallel, co-operative and distributed work, all changes to an object are done on a local copy in a user workspace. While an object is being updated in some local workspace, it is not locked in the database. Other users may therefore fetch the same object and edit it in their own workspaces. When they have stored their versions back to the database there may be parallel versions of that object.

The possibility of parallel versions of an object is also the reason that there may be parallel commands in the command history, as the update of the command history is done in the local workspace.

To keep track of the objects front versions, i.e. the versions that are not parent of any other version, the *hs node* has a set of links, called *latest*, to all front versions. There is also a link called *current* to the version that is the most recent.

The database is managed by a server (DBS), that communicates with the user workspace over a communication network. New versions created in the workspace must be stored explicitly, or else the changes will be lost.

The system is divided into four logical blocks:

- DBS: Databaseserver, handles the actual database.
- LIBLINCKS: Interface functions to the DBS. Handles the communication for the workspace.
- AIM: Application Interface Manager.
- UIM: User Interface Manager

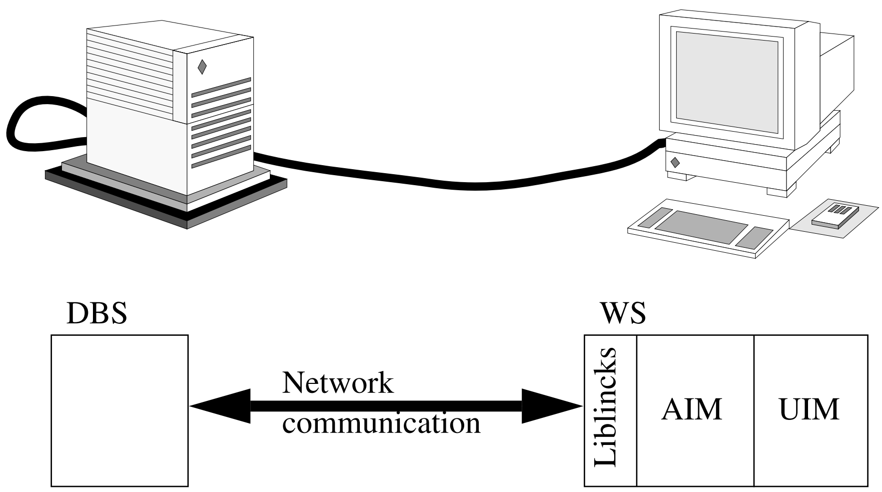


FIGURE 6 The LINCKS system

LIBLINCKS, AIM and UIM are grouped together and is referred to as the workspace. There is one workspace for each user (figure 6). The changes needed to incorporate command history in the system are mainly in the LIBLINCKS layer, but some minor changes are also needed in the AIM layer.

For more information about LINCKS see [Pad86] and [Pad88]. For more information about LIBLINCKS see [Man88].

CHAPTER 3 The command history

3.1 What is a command

In order to discuss command histories, we need first to be clear about what a command is, i.e. what it is that is kept track of with the command history. We therefore take the position that a command is an instantiated function that has been executed successfully. The information to store about a command is the function name and how it was instantiated, i.e. its parameters.

As mentioned in the previous chapter, LINCKS always operates on local copies of objects and changes are stored in the common database only with an explicit store operation. We therefore identify two different kinds of commands. The first are called *workspace commands* and the second *database commands*. Workspace commands operate on the local copies of objects while database commands operate on objects in the actual database; essentially store and get operations.

3.2 Command histories in general

We also need to answer ‘what do we mean with a command history?’, and more specifically ‘how do we want to use the command history?’.

The most basic question to answer with command history is ‘was command A executed before/after or in parallel with command B’. For this, we must have a temporal ordering relation between the commands. The ordering is partial, since different users can execute commands in parallel (and actually update the command history in parallel).

In figures, we present the command history as a graph where nodes are commands and arcs indicate temporal succession, see figure 7.

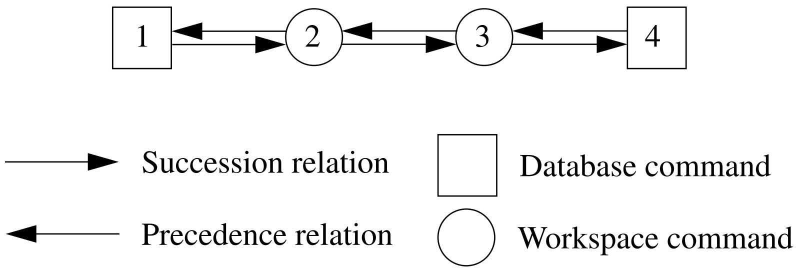


FIGURE 7 An example of a command history graph.

3.3 The relations

It is worth pointing out that the temporal order relationship between commands may be understood as a succession relation in the forward direction and as a precedence relation in the backward direction. Although there is no important difference between these views at the modelling level, it makes a significant difference which view is chosen for implementation through object linking. A forward direction linking more easily supports queries regarding what happened after a given command whereas the backward direction linking supports queries on what happened before a given command.

3.4 The session history

Commands are naturally grouped together into sessions so that each command represents one point in time during the session it occurs in. If a session is an object (in the sense of the LINCKS database model), then each command occurring during the session can be seen as an object version. In this way the database command history has a natural partitioning into session objects whose versions are the commands, and the building of a command history can utilise the existing temporal version management. See figure 10, chapter 4.

3.5 Multi user command history

Commands of different sessions, typically simultaneous sessions of different users, are also related temporally. This provides an additional dimension for temporal queries, and gives answer to questions like: ‘Did user α execute command X before/after or in parallel with users β’s execution of command Y’ or ‘Who changed document X, who changed document Y and which was done first’.

The links between commands of different sessions are always links between database commands. The workspace commands are, as a consequence of the workspace-database separation, invisible for other workspaces. Therefore it makes little sense to allow temporal links between workspace commands of different sessions.

So we will have two time structures in the command history. The first is the user session history that is stored as versions in a session history object. The second is a partially ordered time relation between the database commands, i.e. the commands that applied to the database, see figure 8.

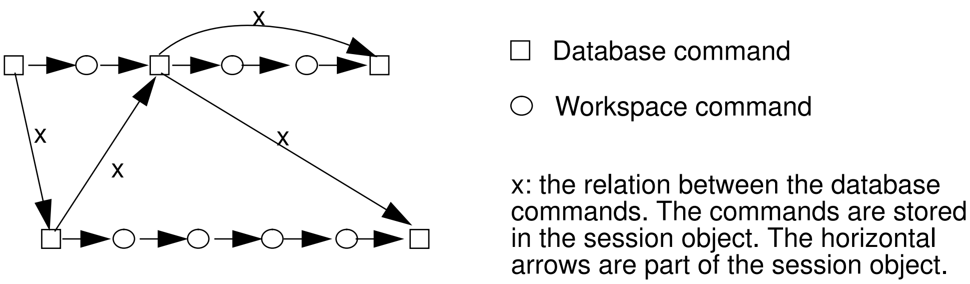


FIGURE 8 The two time structures.

3.6 Command history in LINCKS

In this implementation, the commands that are stored in the command history, are the editing commands of the LIBLINCKS interface. Also, two more abstract commands are logged: the composite store and expand; for operations on composite objects. See appendix C for a detailed list of commands logged in the command history.

The composite commands will work as a sort of delimiter. For instance, the composite store groups together the detailed store commands in the sequence that stores all newly changed parts of a composite object.

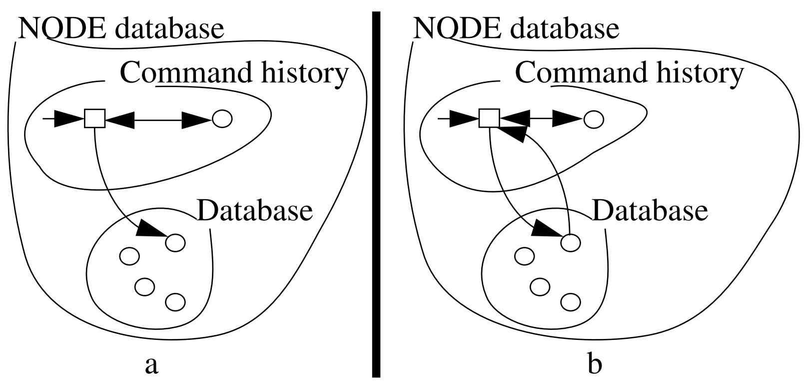


FIGURE 9 The integration command history - database

The commands in the command history are linked to the affected objects, and especially the object store command has a link to the object version stored, see figure 9a. For this command, it is also useful to have the inverse link, i.e. a link from the object version to the store command, see figure 9b. This allows for an easy way to compare the creation times of atomic objects, as well as of composite objects. We only store this backlink from the created version to the command history when the command is a store command.

The command history is itself contained in ordinary objects in the database, and thus requires some extra precautions in order to protect it from unauthorized editing.

CHAPTER 4 Representation of the command history

The command history is contained in a two level representation, where the commands issued within a user session are kept as distinct versions of a session object. Database commands are further ordered temporally by linking together the concerned versions. The temporal ordering between database commands will typically relate commands in different user sessions, thus providing a connection between different user session.

The fine-grained command history as contained within the session objects has a direct management support in the existing temporal history of the database. Session level connection management involves three specific problems:

1. to avoid unnecessary linking
2. to allow *parallel command history update* as well as containing *parallel commands* in the history, and
3. to choose linking direction relative to temporal order, i.e. forward or backward in time, or maybe both.

A direct linking of two commands is considered unnecessary if they already are connected indirectly through a chain of links. The chain of links already provides, by means of transitivity, the information of temporal order, and therefore a direct linking is redundant.

Parallel update of the command history occurs whenever two, or more users store database commands in the history at the same time, since they both have to update the command history. Parallel update of the command history will result in parallel database commands in the history. If there are two commands in the history such that there are no chain of links from one of them to the other, then they are considered as parallel commands.

Linking direction is an issue for reasons of complexity. At first I investigated a forward linking approach, which required considerable computation for managing parallel updates. Later, I turned to a backward linking approach, and could reduce code size by an estimate of 30%. The backward linking approach is also conceptually simpler.

4.1 The session object

The session object is a sequence of command nodes that represents a user session from login to logout. The command nodes hold information about how the commands were instantiated and links to the affected objects. See appendix B for a complete list of command node attributes and links.

By storing a session as versions of the same object, we use the existing temporal history to represent the succession relation and the edit history to represent the precedence relation. See figure 10 for an example.

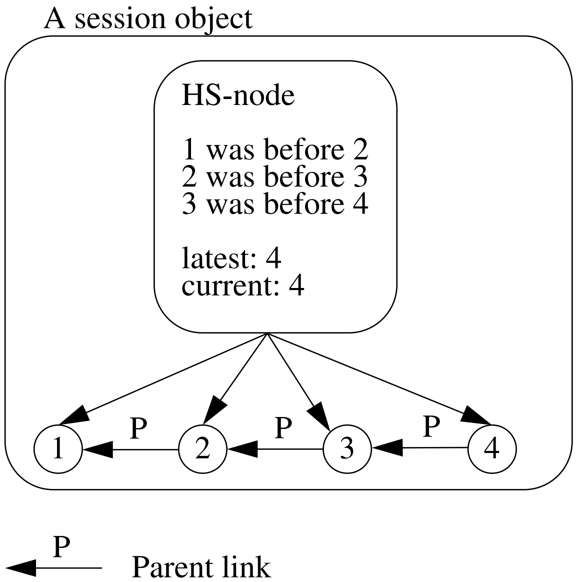


FIGURE 10 How the example history in figure 7 is represented.

It is important to note that this way of using the versioning mechanism of LINCKS does not utilise the contents inheritance along the edit history. Normally, as an object version is created sub-sequent to an existing version, the new version first inherits the contents of the old version, and thereafter the new version may be edited (until explicitly stored into the database). This inheritance is not useful for the command history, but rather an obstacle, since every new session object version must be emptied after creation and before the appropriate command information can be filled in.

For instance assume that a command A is recently stored in the command history with the particular information that the attribute ‘Group’ has the value ‘Chapters’. Assume further that the sub-sequent command B does not concern the attribute ‘Group’ at all. As the node to represent B is created as successor to A, it will, by virtue of the inheritance, automatically contain that attribute ‘Group’ has value ‘Chapters’. This information is however not relevant for B, and it must therefore be removed in forming the representation of the command.

We have used the session object despite the fact that it cause problems with the inheritance, because it gives a natural grouping of the sessions and we get the temporal relation as part of the object history.

4.2 Removing of redundant arcs

Between session objects, we set up links to handle queries like 'did user α know that user β had executed command Y when he executed X?'. These links are always between database commands, and are never connected to any workspace command.

One way to incorporate those links in the command history is to add arcs between the different single user command histories according to the following rule.

Let X_1,X_2 to X_n be the different logged in users. Each of them then has one latest executed database command stored in his own session history, l_1, l_2 to l_n. When user X_i executes a new database command, and stores that command in the command history, we add one arc from each of the commands l_1- l_n to the new command node l_i. See figure 11. We let a session be illustrated by a horizontal sequence of commands; circles for workspace commands and boxes for database commands.

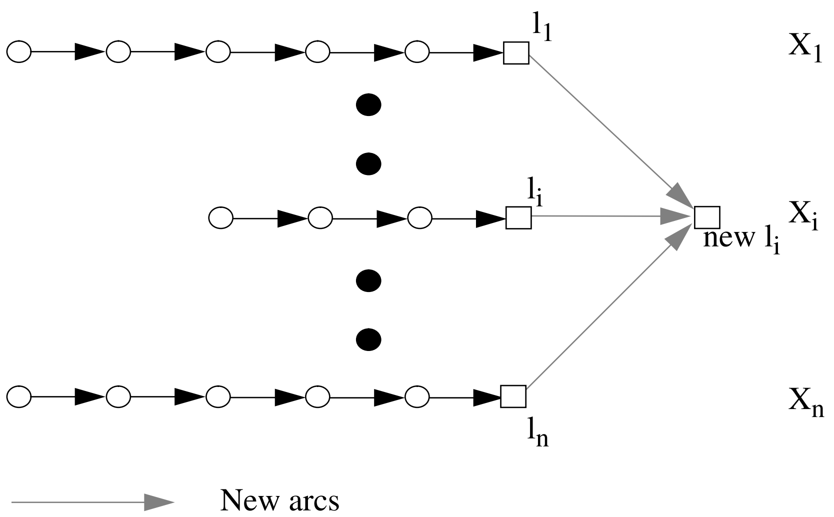


FIGURE 11 First attempt to a multi user command history.

Clearly this provides all information needed, but if carried out simple mindedly, it has the disadvantage that it may result in redundant arcs. For instance, assume that users A, B and C all have executed some commands, and then B executes a database command followed by C executing a database command. This gives a situation as showed in figure 12a-c.

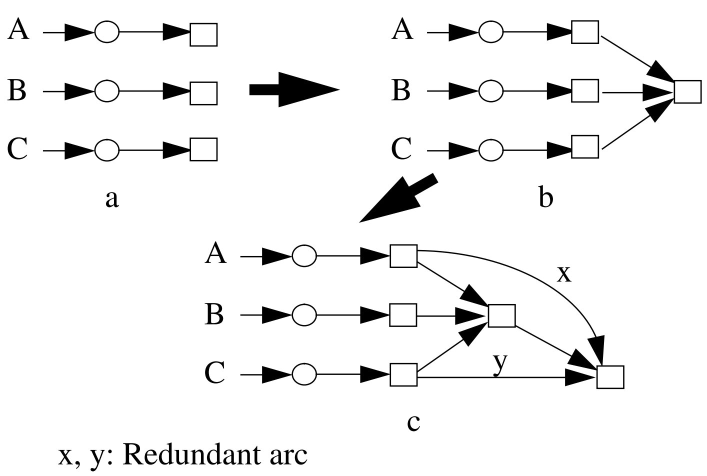


FIGURE 12 The problem of redundant arcs.

The arc x in figure 12c is unnecessary to store, because it is a transitive arc and the same command node can be reached by following two arcs. The arc y is however useful to store since it is a part of user C's session. If we removed the y arc, we would break up the session, which would be unfortunate.

The present implementation handles arc redundancy as follows. We maintain a set 'Front' that contains all the latest database commands that do not have any other database commands executed after them. When a new database command executes, it is connected to the commands in 'Front', which thereafter is reset to contain the one command that was just executed, see figure 13.

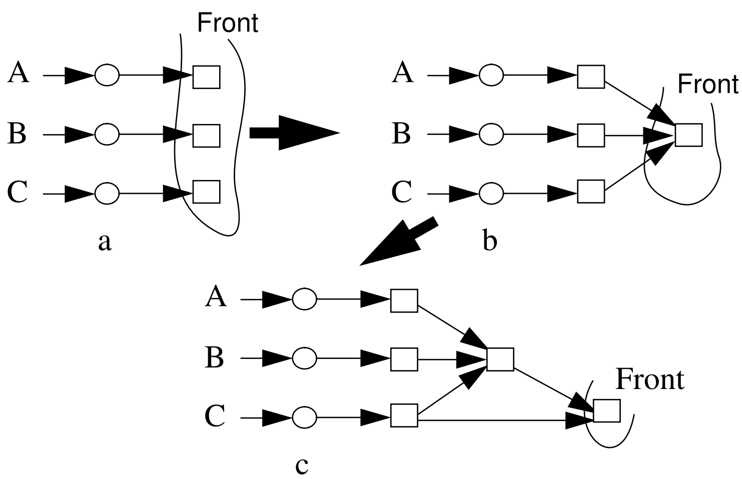


FIGURE 13 The solution to the problem with redundant arcs.

The only control structure needed is the set of front commands, i.e. the database commands that have no database commands after them. To that end we use a control object, called *ch object* (Command History object), which is also the handle to the command history as a whole. Each version of the ch object has one link 'Front' to exactly one front command node. The set of latest ch object versions, through their 'Front' links, forms the set of front commands in the command history, see figure 14.

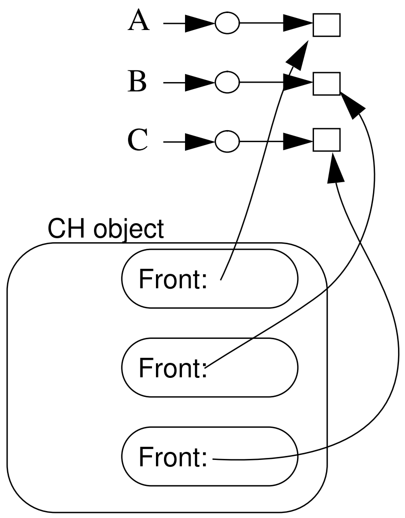


FIGURE 14 The Ch object for the command history in figure 13a.

4.3 Parallel updating of history

We need to ensure that we manage the situation when two, or more, users try to update the command history at the same time. Assume for instance, that after the situation showed in figure 13b, both user A and C execute some database commands in parallel. Then we want to get the command history shown in figure 15.

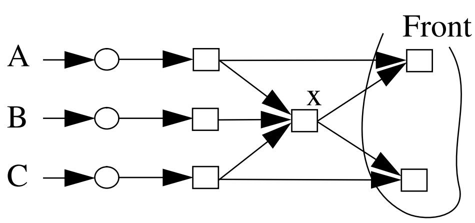


FIGURE 15 Parallel commands.

This is however not a problem, since it is solved by using the LINCKS feature that the editing is done on a local copy and then stored back to the database with an explicit store command. Both users get the same initial ch object; actually the same set of latest versions of the ch object. They each add their new database command in their session object and add links from the commands in the set ‘Front’ to the new command. The ‘Front’ set will in this case be only the command x. The set of latest version of the ch object will, by each user, then be reset to contain only one version, the one with a front link to the new command. The two new versions of the ch object are then stored in the database, which result in two versions in the set of latest versions.

4.4 Linking

When it comes to linking the database commands together there are two approaches, forward and backward linking. A forward direction linking more easily supports queries regarding what happened after a given command whereas the backward direction linking supports queries on what happened before a given command. My initial attempt used a solution very similar to the way the version history in LINCKS is implemented to represent the forward linking, see appendix A. However this approach had a number of potential problems such as redundant storage of information, very big nodes, centralized storage and it did not handle the backward links. My second attempt used the backward approach to handle the command history, this is also the solution used in the implementation.

The backward links can be stored as proper links in the object versions attached to their root ends. This could not be done with the forward links, since we do not know which commands will follow the command that had just been executed (at least it can not be done in an easy way in the LINCKS system). The distributed storage of the backward link has a direct advantage over the centralized storage of the forward links in that the information of the preceding database commands is stored in the corresponding command node and not somewhere else. Another advantage is that the size of the nodes that have to be retrieved to calculate the preceding database commands is small.

Since we most often are concerned only with the most recent end of the command history, the distributed storage gives a gain in information transfer time between database and workspace.

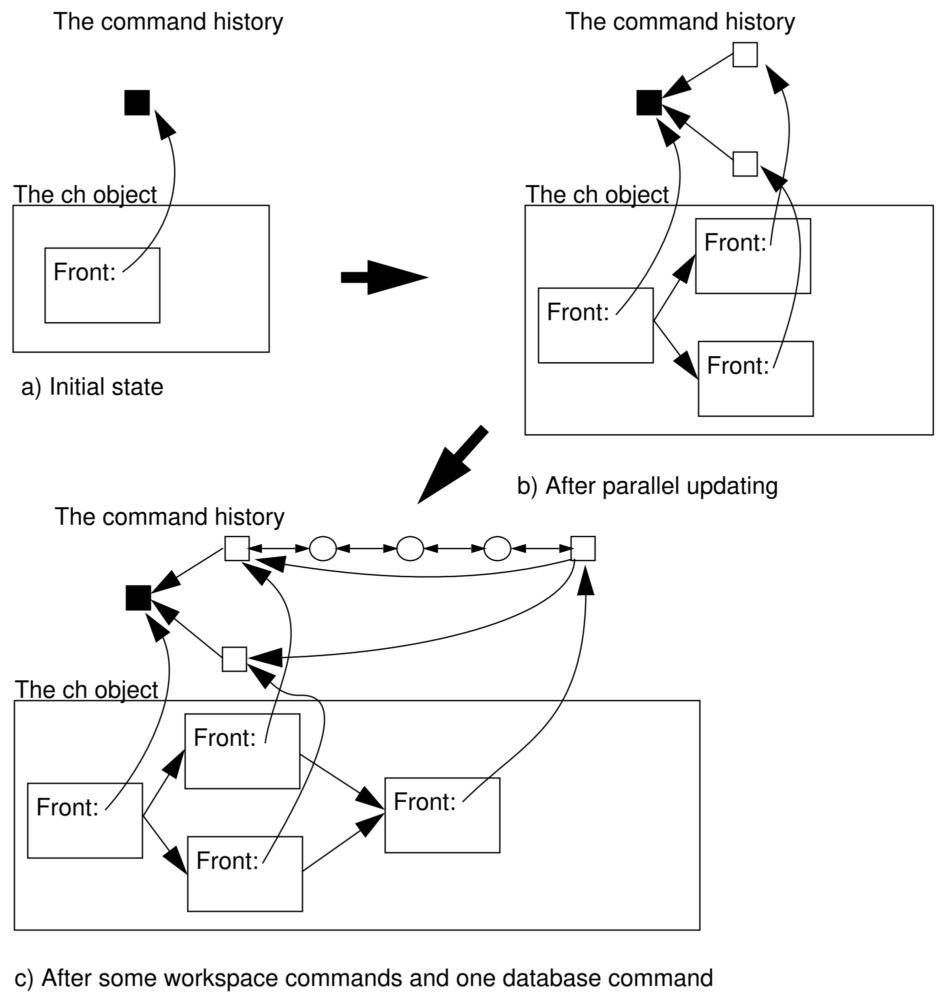
The backward linking also has the important advantage that it will contain a representation of the forward links by the way the command history is constructed. This is explained in more detail later.

For the backward linking approach, the update of the command history is done according to the following rule.

- For each command (workspace and database): add a new version to the session history object and fill in the appropriate values.

- If it is a workspace command, then store the command node in the database and end the update.
- Otherwise add a new link group CH:Previous command to the command node. In this link group we store links to the previous database commands, i.e. the ones that were front commands when we started to update the command history. These are found by traversing the latest of the ch object and retrieving the ‘Front’ link in each of the latest versions.
- Next we create a new version of the *ch object* and store a link ‘Front’ that points to the new command node. The latest version of ch object is set to be this new ch object version.
- Finally, store the new ch object version.

It is however important to note that the succession relation is indirectly available in the temporal development of the ch object. We illustrate this in figure 16, which corresponds to the following chain of events. At first, there are no commands in the command history. This is represented as a special null-command node (the null-command node is used because we want to have exactly one start node in the command history, it also simplifies the update, since the initial case is included in the normal update case). Then one user executes a database command (e.g. a login). To update the command history, he retrieves the then current ch object (figure 16a). Before he stores the updated ch object another user executes a database command and therefore retrieves the same (still current) version of the ch object. After this, the first user stores his new ch object and the second user finish his updating and stores his new ch object in the database. We have then the situation showed in figure 16b, where the ch object has parallel versions. The parallel versions of the ch object corresponds directly to the parallel occurrence of database commands. Continuing the example, the first user executes a sequence of workspace commands, and then another database command, which results in a command history as in figure 16c.



The arcs in the command history are stored in the nodes of the command history. The arcs from the versions of the ch object to the command history are the front links. The arcs in the ch object, between the versions, are stored in the history structure node of the ch object.

FIGURE 16 An example of how the command history evolves.

As we see in figure 16, the temporal history of the ch object (available in the VSNET group of the history structure node for the ch object) always reflects the history of database commands. To be able to use it for traversing the command history in succession order, we add a back link for each Front link, as illustrated in figure 17. The back link has not been implemented, since the management of the temporal history is currently being changed. When the back link is added it might be called ‘CH:succ command’ to reflect that it contains (indirectly) links to succeeding database commands.

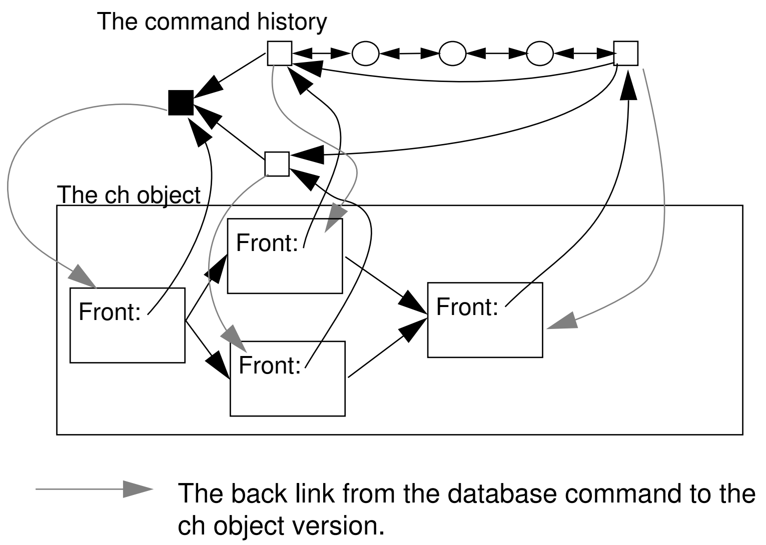


FIGURE 17 Connections command history - ch object

In order to find the database commands that succeed a database command X we follow the back link from X to the corresponding ch object version. We then find succeeding versions from the ch objects history structure node and from them follow each front link to find the actual commands.

4.5 Pros and cons of the representations

The two different approaches to represent the command history, i.e. backward versus forward linking, give some different advantages. To make the decision easier I have summoned them here:

- The implementation of the backward representation gives about 30% less code than the forward representation. The code that is removed is concerned with the merge of parallel versions of the ch object. This gives simpler, faster update and more reliable code.
- In the forward representation we ended up with redundant storage of the command history. This was avoided in the backward case.

In both the forward and the backward representation we end up with very big history structure nodes. This is more of a system design matter, but it may cause problems in the future with long response times.

It is important, for the decision on which representation to use, to note that the forward representation can be improved to avoid the redundant storage in the ch object version. This is done by simply removing the CHNET:after link group. By this we get the same structure of the command history as with the backward representation, see figure 16 and 17, except that we have not stored the precedence relation.

All things considered, the decision of which approach to take is quite simple: since the (improved) forward representation is included in the backward representation, we use the backward representation.

CHAPTER 5 Using the command history

The planned uses of the command history presented in the introduction were:

1. Find special version.
2. Composite object temporal history.
3. Do a reset of the database/Rollback.

These are just some of the planned uses of the command history since more extensive use of the command history is outside the scope of this project. We believe that a central use of the command history will be to determine the temporal order between two commands. It is also possible to use the command history to determine the temporal order between creation time of both atomic and composite objects. To determine the temporal order between two commands we provide a function that answers this query.

The query function determines the temporal relation between two commands, α and β, in the command history. The relation can be that α is *before* β, α is *after* β or α and β are *unrelated*. This function is intended to be used by other functions that query the command history.

The algorithm takes two commands α and β and uses the backward links between the database commands (and the forward links in the session containing β) to try to find a path from α back to β. If no such a path is found an attempt is made to find a path from β back to α, in which case α is before β. If neither case yields a path the commands are unrelated.

In searching for a backward path from α back to β the algorithm first follows backlinks between the database commands until it reaches a command in the same session as β. The within session forward links are then followed, searching for β. If β is found we conclude that this was not a path from α back to β. If the end of the session is found without encountering β we conclude that we have a path from α to β and terminate the search. See figure 18.

The within session forward links are used instead of the within session backward links because the forward links are stored in the history structure node of the session while the backward links are distributed over the commands. Using the forward links requires only one node to be retrieved while use of the backward links demands multiple retrieve operations.

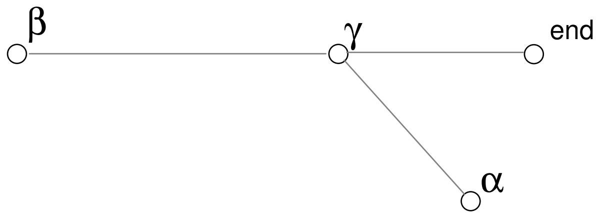


FIGURE 18 The search function

Each search branch is terminated when the null-command node (the start of the command history) or a command that is in the same session as β but before β is reached, in both these cases further computation can not reach a command that is in the same session as β and after β since that would imply a circular history.

To determine if two commands are in the same session we compare the x part of the label (label equals a pair x,y).

The search to find the γ command can essentially be done in two ways, breadth-first or depth-first. We use the breadth-first approach because we believe that the commands that we want to relate are relatively close in the command history and if so, the breadth-first-search is the better approach.

Here is a short example of how the query function works. Assume we have the command history shown in figure 19 and we want to determine the order between the command α and the command β.

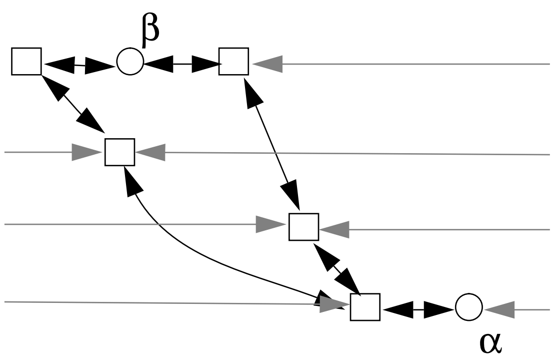


FIGURE 19 An example of how the query function works.

The first we have to do since α is a workspace command is to follow the temporal history structure of the session object that contains α backwards until we find a database command, note that we will always find a database command since the first command always is a LOGIN. We now have a set of reached commands at the distance one from α that contain one command, see figure 20. The workspace commands between α and the preceding database command are not counted.

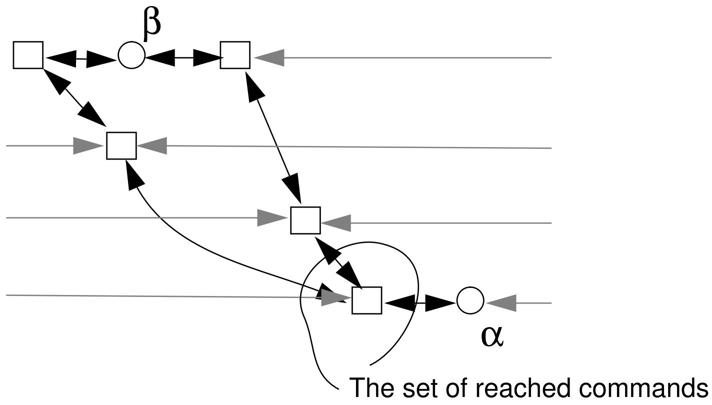


FIGURE 20 The search after one level.

Since that command is not in the same session as β we follow the backward links to the previous database commands. Each time we follow one of those links we test if the new database command is in the same session as β and if it is we also test if it is after β. If we find a command with both these properties we end the search and conclude that α is after β.

After 2 steps we get the situation viewed in figure 21 where none of the commands reached so fare are in the same session as β. We only keep the front commands of the breadth-first search, so the previous set of reached commands are deleted.

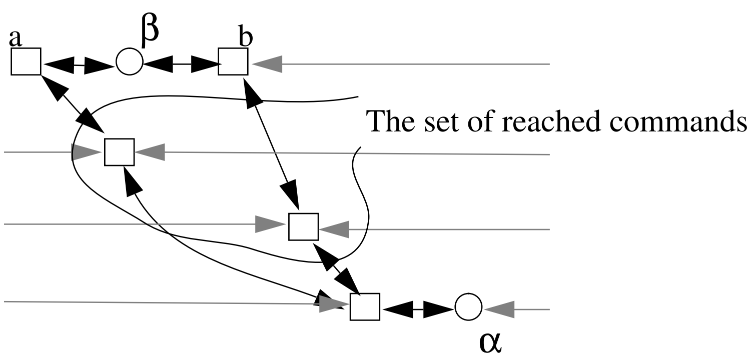


FIGURE 21 The search after two levels.

Thus we continue the search one step further. Now we find a command (a) that is in the same session as β but on following the within session forward links we discover that it is not after β, so we can not stop the search. However the next command (b) is both in the same session as β and after β, so we stop the search and conclude that α is after β.

CHAPTER 6 Design of the functions

The functions that handle the command history; both updating and querying, are placed in the new module comhist.c. This module is located in the LIBLINCKS layer. The comhist module interacts very closely with the LIBLINCKS interface functions. The module containing the interface functions, usfuns.c, is also changed. There are further some minor changes in some AIM-modules, to handle the initialization of the command history and to log the composite commands. For an overview of the file dependencies see figure 22.

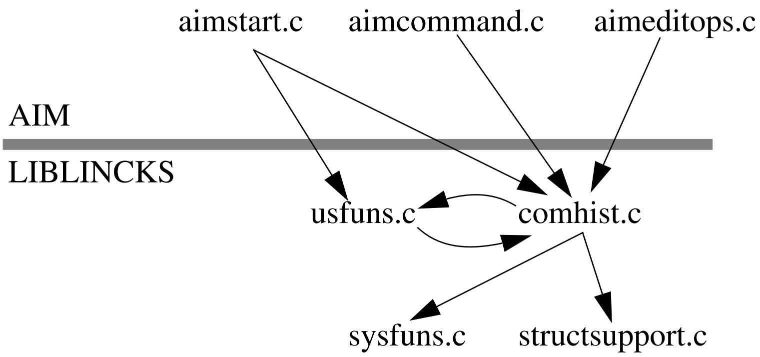


FIGURE 22 File dependencies

6.1 The database objects

The command history is represented as a set of objects in the Node database. For each user, or in the future for each user process, there is one session object. The different operations that the user executes are stored as versions of that session object. In this way we automatically get both the forward linking and the backward linking in the session object from the original history management. To store a command is just a matter of storing a new version of a session object.

The session objects are also called the user process objects. The is chosen to stress the point that there will, in a future system, be one object for each process and not only one for each user. In the system of today, there is only one process for each user.

Here is a description of how a user process object looks. The words in capital letters are the group tags and the following word the field tag. The parameters are the set of all the relevant parameters used in these functions which are represented in the command history. Some of the parameters are stored as attributes and some as bound links to affected objects.

A user process node has the following attributes:

- CALL:Function — Which function was executed.
- CALL:Type — Workspace or database command.
- CALL:application — Which application that executed the command.
- PARAMETER:from — The parameter from.
- PARAMETER:to — The parameter to.
- PARAMETER:group — The parameter group.
- PARAMETER:field — The parameter field.
- PARAMETER:pos — The parameter position.

And the following links:

- PARAMETER: lbl — The parameter lbl.
- PARAMETER:from — The parameter from.
- PARAMETER:to — The parameter to.
- CH:prev command — A set of bound links that points to the preceding database commands. Is only stored if it is a database command.

To construct the command history we use a special object, called the ch object, where each version has one link to one front command in the command history. Each of these versions represents one database command update of the command history.

The ch object has this link:

- CHNET:front — A link to the latest executed database operation. A single link.

The forward linking is not stored explicitly because it is indirectly created in the history structure node of the ch object.

6.2 Protection

Because the command history is stored as ordinary objects in the database, it is accessible to the user and could therefore be edited by him/her which is a problem. The command history should not be directly viewable, since the paradigm is that you can edit everything you see. This is done by adding a link in the system root history structure node. Users can not access any history structure node. That link, called CH:History, is an unbound link to the ch object.

On the workspace side, there is a test that the user is not trying to make a change in the command history. This is done by inspecting (in the beginning of the SO_STOREOBJ function) the x part of the label parameters which should not be equal to the user process or the ch object (we recall that a label is essentially a pair of node numbers (x,y) where x is the node number of an objects history structure node and y is the number of a version node).

A weak point in the protection is that there is no effective way to test if a user tries to edit a user process object of another user, since a user has control only of his own user process object. However as there is no way to obtain another user’s process object this is not considered a problem.

6.3 Global variables

We define a few global variables so as to make the command history management more time efficient. These are all static variables in the comhist module. The first is a label called user_process, which is the object where in the different commands will be stored. We have chosen to have this as a global variable because there is otherwise no way to find the right user process node amongst all other user processes. The label of the user process is needed as the command label in the call from SO_STOREOBJ to CH_STOREOBJ that is used to create the SYSTEM:command link from the created object to the creating command object in the command history.

The second global variable is a label for the ch object. The ch object can be found from the system root, but to avoid having to do that calculation every time we want to update the command history we keep it as a global variable.

There is also a flag, which indicates whether the user has done any updating of the command history before.

Finally we have, for future use, added a string variable called application that could be set by the function CH_set_application. This variable is currently not used, but in the future it might be interesting to differ between different applications in the command history.

6.4 Interface functions to the command history

There are six interface functions to the command history.

CH_init initializes the command history, should be called only when initializing the entire database. It could possibly be used also to restart the command history, i.e. start a new clean command history.

CH_update updates the command history with a new command.

CH_set_application sets the application variable, which is then stored with every command to indicate what application executed the command.

CH_relation determines the temporal order between two commands.

CH_get_user_process_object returns the label for the next user process object version, it is currently only used by SO_STOREOBJ to create the link from the created object to the creating command.

CH_bad_lbl is a boolean function that tests if the label is equal to the ch object or the user process object. Is currently only used by SO_STOREOBJ to protect the command history.

For a detailed definition of the interface functions see appendix E.

6.5 Which commands to log

Some of the LIBLINCKS interface functions and two composite commands are commands that are logged and stored in the command history. It is of course easy to change which commands are logged, for instance to log some other composite commands. To change a function to be logged by the command history it is usually just to add a call to the CH_update function with some appropriate parameters.

A rule of thumb for which to log and not to log is that commands that change the contents of a node (the edit commands) should be logged, the others should usually not be logged. Commands that retrieve different parts of objects from the database are stored as a common db_get command, since there is no point in logging ‘read’ commands at lower granularity than the atomic object. Edit commands are logged at the granularity of individual attributes, links, since it is interesting to know what part of the object was changed. The composite command works as delimiter in the command history, for instance the composite_store command represents the storage of a composite object and groups the store commands of the parts together.

The commands which are considered as database operations are those which get or put an object in the database, for instance SO_STOREOBJ and composite expand. Workspace operations are those which change the contents of an object in the workspace. Since the object is retrieved from the database if the user executes an edit operation and the object is not in the workspace, most of the commands can be both database and workspace command depending on if the object was in the workspace when the execution started or not.

The changes done to the composite functions (expand and store) are a call to the CH_update function just before the return. The changes to the other functions are a call to CH_update just before they return SUCCESS, but no call if the function returns FAIL.

The change to log the composite expand command is done in the aimeditops.c module. The change to log the composite store command is done in the aimcommand.c module. All the other will be changes in usfuns.c. For a list of what commands to log see appendix C and D. The change is essentially to add a call to the CH_update function, in some of the functions some additional changes have to be done to make the command history work properly.

6.6 Algorithms

To store a command in the command history there has only to be a call to the function CH_update. This function then incorporates the command in the command history. The command history is stored in the database as ordinary objects. To test if the command is to be considered as a workspace command we use the LIBLINCKS function INWORKSPACE, it is placed in the beginning of the command function and tests if the affected object is in the workspace or not. If it is a workspace command the update is just a matter of storing a new version of the user process object, since one user’s different commands are stored in the same object. If it is a database command we also have to connect the command to the previous front database commands (the connections between the sessions), this is done in the ADD_DB_COM function.

CH_update takes the following parameters:

func:char	The name of the function that was logged.
type:int	0 if the function is to be considered as a database command (DB_command) or 1 for a workspace command. This definition corresponds to the INWORKSPACE function.
from_attr:char	The function parameter.
to_attr:char	The function parameter.
group:char	The function parameter.
field:char	The function parameter.
pos:int	The function parameter.
lbl:label	The function parameter.

from_lbl:label	The function parameter.
to_lbl:label	The function parameter.

The CH_update function has the following algorithm:

CH_update(func, type, from_attr, to_attr, group, field, pos, lbl, from_lbl, to_lbl)

If ‘the user is not logged in’ then create a user process node.

Update the user process node according to the function parameters.

If type =DB_command then {

ADD_DB_COM()

Set CH:Front in ch object to point to the current version of the user process node.

SO_STOREOBJ(ch object)}

SO_STOREOBJ(user process)

If not logout-function then

create a buffer node in the user process node.

RETURN SUCCESS.

END_CH_update

The essential part of ch_update is the update needed for a database command. That part is placed in the ADD_DB_COM function.

ADD_DB_COM has the following algorithm:

ADD_DB_COM()

X:= GET_LATEST(ch object)

For all members y in X do {

Get the front link in CHNET:front in y.

Store the link front in CH:Prev_command in the user process node.

Remove y from the set of latest in the ch object.

}

RETURN SUCCESS

END_ADD_DB_COM

Here is a short example of how CH_update works. The commands that have been executed to get to the initial state are that two users (A and B) have executed two login commands, see figure 23a. The next that happen is that user B executes a workspace command. CH_update then fills in the function parameter in the user process object and since it was not a database command it stores the user process object and does a dummy edit of the user process object. The dummy edit is used to create a new instance of the user process object, i.e. the command node that will be filled with the next command. We now have the situation shown in figure 23b. Next, user A executes a database command, the update function (as always) fills in the user process object. This time it was an database command so the ADD_DB_COM function is called to construct connections to the previous database commands. ADD_DB_COM retrieves the latest versions of the ch object and copies the Front link in each of the latest versions to the user process object, see figure 23c. It also removes all the latest instance of the ch object from the set of latest instances. The CH_update then creates a new latest instance of the ch object that have a front link to the new command node. Next the new instance of the ch object and the user process object is stored and the dummy edit part is executed, see figure 23d.

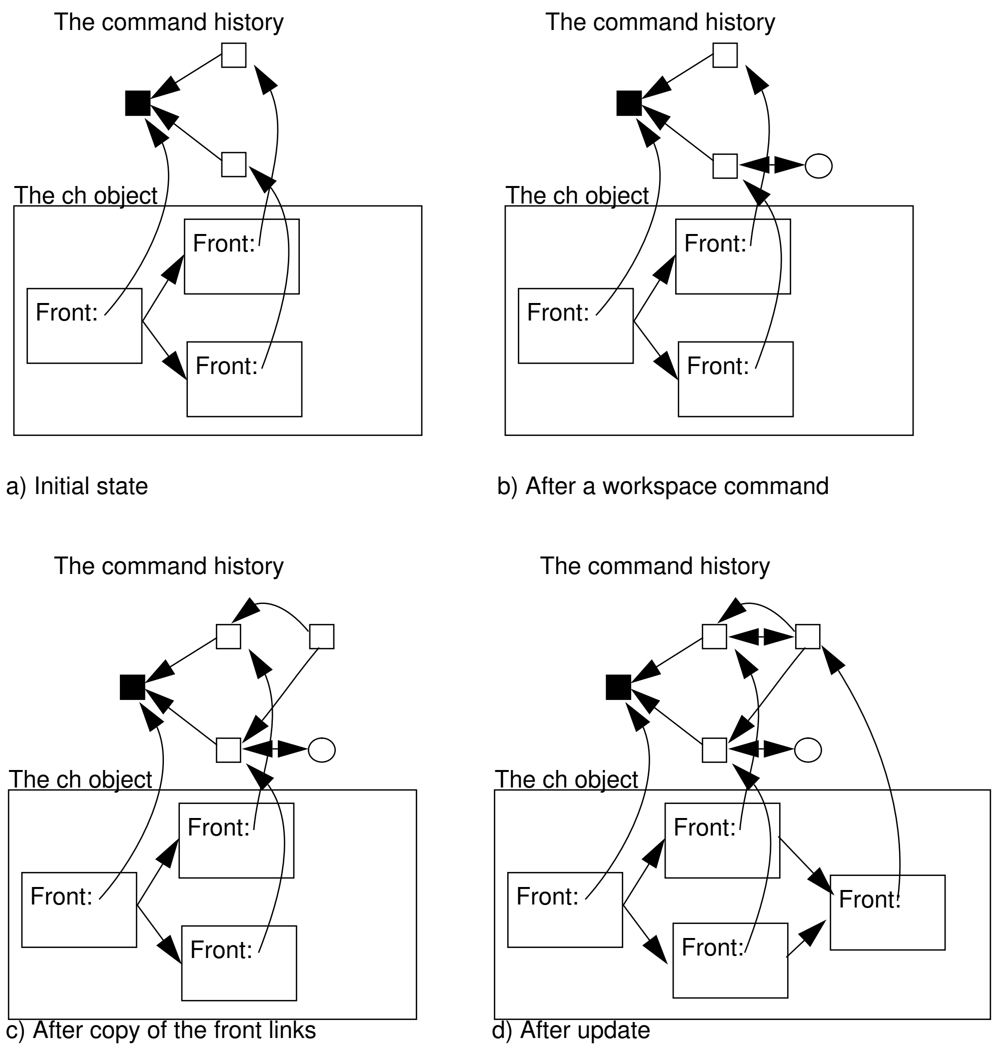


FIGURE 23 An example of how CH_update works.

6.7 The query relation function

This function tests how two commands (X and Y) are related in time. There are three kinds of answers: X before Y, X after Y or X and Y are unrelated. The function returns 1 if X is before Y, 0 if they are unrelated and -1 if X is after Y. The function has the following algorithm:

If X and Y are versions of the same user process: Follow the history forward from X; if you reach Y then return ‘X before Y’, and if you reach the end return ‘X after Y’.

If X and Y are versions of different userproc’s: This part is divided in two cases:

i) Follow the history backward from X until a database operation is reached. Follow the connection link (the links that are stored in CH:Prev_command) until a command Z in the same user process as Y and after Y is reached, then return ‘X after Y’ else go to test ii.

ii) Follow the history backward from Y until a database operation is reached. Follow the connection links until an operation Z in the same user process as X and after X is reached, then return ‘X before Y’ else return ‘unrelated’.

To determine the temporal order between two commands, X and Y, in the same session object we traverse the temporal history of that object. We start from X and traverse the temporal history in the forward direction, if we reach Y we conclude ‘X is before Y’ and if we reach the end we conclude ‘X is after Y’.

The CH:Prev_command links are traversed with a breadth-first-search algorithm, where a branch is terminated by reaching the null-command node or a database command that is in the target session object but before the target command.

6.8 Initialization

The final command history function which is reachable externally is:

CH_init	Initzialize the command history. Should be called once only when initializing the database as a whole.

Algorithm:

CH_init()

Create and store the ‘no commands executed’-node (Null-node).

Create the ch object.

Set CHNET:Front in ch object to point to the NULL-node.

Store the ch object.

Set the unbound link CH:History in the system roots history structure node to point to the ch object.

Store the system root.

END CH_init

For structure charts of the functions see appendix F.

6.9 Changes in existing code

The changes done to existing code are of four different types. The four types of changes are:

1. Composite commands and LIBLINCKS interface functions not used by the command history update, see section 6.9.1 and 6.9.2.
2. The LIBLINCKS interface functions that are used by the command history update, see section 6.9.3. These functions have to be treated specially to avoid infinite cycles. Otherwise the update would make calls to the interface functions which results in a new update, which makes new calls to the interface functions and so on.
3. The LIBLINCKS interface function SO_STOREOBJ, see section 6.9.4.
4. The common DB_GET command, see section 6.9.5

The changes in the LIBLINCKS interface commands are all in the usfuns.c module, the changes for in composite command are in the aimcommand.c and the aim editops.c module.

6.9.1 Composite commands

When the user executes a store he usually stores a composite object, i.e. several atomic objects are stored at the same (conceptual) time. Similarly when he executes an expand object operation, he usually expands a composite object. Both operations are logged in the command history.

The changes for these are in the AIM layer, in two different functions: the function closecmd in the aimcommand.c module (for the composite store command), and the function expand in the aimeditops.c module (for the composite expand command).

The changes are to add a call to the CH_update function just before exiting the functions.

6.9.2 Functions not used by CH_update

Functions not used in building the command history have been changed in the following way:

Change the line ‘return SUCCESS’ to ‘return CH_update(func, type, from_attr, to_attr, group, field, pos, lbl, from_lbl, to_lbl)’. This makes the interface function to return success only if both it and the command history update were successful. The functions that are changed in this way are:

CAGN_CHANGEATTRGROUPNAME	CAN_CHANGEATTRNAME
CLGN_CHANGELINKGROUPNAM	CLN_CHANGELINKNAME
RAGT_REMOVEATTRGROUPTAG	RAGV_REMOVEATTRGROUPVAL
RAT_REMOVEATTRTAG	RAV_REMOVEATTRVAL
SA_SETATTR	RLGT_REMOVELINKGROUPTAG
RLGV_REMOVELINKGROUPVAL	RLT_REMOVELINKTAG
RLV_REMOVELINKVAL	ULA_UNLINKAT
UL_UNLINK	SI_SETIMAGE
CPO_COPYOBJ	LOGIN
LOGOUT	CO_CREATEOBJ

6.9.3 Functions used by CH_update

Another kind of change in usfuns.c are the changes to functions that are themselves used in building the command history and that also are logged.We made these changes by renaming the existing functions by adding a CH_ prefix to the names. Then new functions were written with the old names, making calls to the CH_ functions and if they return successfully, additional calls to CH_update are made. See figure 24.

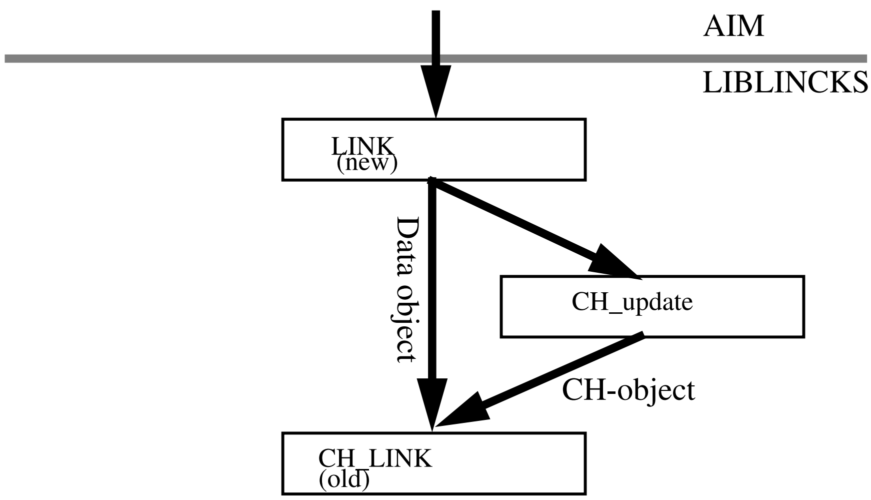


FIGURE 24 An sketch of how LINK will interact with other functions

The CH_ functions are legally called only from the corresponding LIBLINCKS interface function and from CH_update and the new SO_STOREOBJ.

The algorithm in pseudo code for the new LINK function is as follows:

```
LINK(from, group, field,to,pos)
        If (Errcode = CH_LINK(from, group, field, to, pos))
            then return Errcode
            else return CH_update('LINK',etc)

END_LINK
```

The functions that are changed in this way are:

RLV_REMOVELINKVAL	LINK
SA_SETATTR	SO_STOREOBJ
CO_CREATEOBJ	RLT_REMOVELINKTAG
RAT_REMOVEATTRTAG	RFL_REMOVEFROMLATEST
GL_GETLATEST	

Other LIBLINCKS function that are used in building the command history, but not logged, are:

RO_RELEASEOBJ
INWORKSPACE

If these functions are to be logged in a future system similar wrappers have to be made.

6.9.4 SO_STOREOBJ

One interface function that is handled differently than other functions which are both logged and used by CH_update is SO_STOREOBJ. It is not changed in the same way as the other functions, because it adds a link SYSTEM:command from the created object to the creating command in the command history.

There is a circular dependency between the creating command in the command history and the created object in the database. A new version in the command history should not be created before the stored object has been appropriately stored. That storing, however, can not be executed before we know the label of the command version.

We have solved this by using a sort of create ahead functionality. We create a new, not filled in version of the user process (a buffer node) just before we exit CH_update. By using this buffer node we have a reference to the node where the next command will be stored, although the command itself has not yet been stored in the command history. When a user executes a SO_STOREOBJ command we add a link (called SYSTEM:Command) to the object. That link points to the empty buffer node. The label for the buffer node is local to the com_hist.c module, so an interface function (CH_get_user_process_node) that returns the label of the user process object is used to make it possible for the SO_STOREOBJ function to retain the label of the buffer node.

SO_STOREOBJ has the following algorithm:

```
SO_STOREOBJ(lbl)
        If CH_bad_lbl(lbl) then return ERR_LABEL
        CH_get_user_process_node(com_lbl)
        CH_RLV_REMOVELINKVAL(lbl, ‘SYSTEM’, ‘Command’)
        CH_LINK(lbl, ‘SYSTEM’,’Command’, com_lbl,1)
        if (Errcode = CH_SO_STOREOBJ(lbl))
            then return Errcode
            else return CH_update(‘SO_STOREOBJ’, etc.)
END_SO_STOREOBJ
```

6.9.5 DB_GET commands

There are a number of LIBLINCKS interface functions that have in common that they only retrieve some part of an object. Consider for instance, GA_GETATTR which gets an attribute of an object. If the object is not in the workspace, it is automatically loaded from the database. That load operation is interesting to keep track of, since we want to be able to reconstruct what a user had in his workspace at a certain time point. However the access of the part of the object already in the workspace is irrelevant. The functions are changed by changing the line ‘return SUCCESS’ to returning CH_update(“DB_GET operation”, DB_COMMAND, NULL, NULL, NULL, NULL, NULL, lbl, NULL, NULL) if the object was not in the workspace.

The following functions are stored as a common DB_GET command when they result in a load operation from the database:

GAN_GETATTRNAMES	GAGN_GETATTRGROUPNAMES
GA_GETATTR	GI_GETIMAGE
GLGN_GETLINKGROUPNAMES	GLI_GETLINKITEM
GLN_GETLINKNAMES	GLV_GETLINKVAL

CHAPTER 7 Conclusion

7.1 Summary

We have developed and implemented a partially ordered command history for the multi-user LINCKS system. We have examined two approaches of how to represent the temporal order in the command history (forward vs backward linking). The backward linking was chosen, because it results in a faster update of the command history and also contains the forward links indirectly.

We have defined two types of commands: workspace commands and database commands. Workspace commands only affect objects in the workspace and database commands store or retrieve objects from the database.

The different user sessions (LOGIN to LOGOUT) are grouped together in session objects. Temporal ordering between sessions exists as a temporal ordering between database commands of different sessions. This ordering is represented through backward links in the commands.

The command history is stored as ordinary objects in the NODE database and has been protected from unauthorized editing by making the command history invisible for the ordinary user. It can only be reached by applications that are able to examine the history structure node of objects. This functionality is not provided by LINCKS to external applications.

A query function has been implemented. The function takes two commands (A and B) and determines the temporal order between them. The answer can be that ‘A was before B’, ‘A was after B’ or that ‘A and B are unrelated’.

7.2 Improvements

There are two improvements that I have thought of. The first is to add a link from each database command in the command history to the corresponding version in the ch object. This would give us a direct way to process the history in the forward direction. As mentioned, we need both forward and backward linking to be able to answer command history questions effectively.

The code that has to be added is probably fairly simple. It has not been added to the current implementation, because the management of the object's temporal history is currently being changed.

The second improvement that I suggest is to replace the query function implemented today with a more general search function. The temporal query function is then implemented using the more general search function.

I suggest the following design of the more general search function. The function is a mapping function over commands. The function could be defined as:

CH_search(Start_set, Fn, Search_type, Direction, Arc_type, Result)

Start_set	is the set of commands to start the search from.
Fn	is applied to the command, should place the result in the result structure. Returns *stop* or *continue* search.
Search_type	is either breath-first or depth-first search.
Direction	is the direction to search in, either forward, backward or both.
Arc_type	is the type of arcs that shall be followed, could be either work-space, database or both types of arcs.
Result	is a structure to return the search result in. Is defined by the user function.

Two useful sets of commands to use as Start_set in the calls to the CH_search function is the beginning and the end of the command history. To calculate these sets I suggest that two functions are added, the first returns the set of the null-command node and the second returns the set of front commands in the command history. These functions are intended to make it easier to use the search function for searches that starts in the beginning or the end of the command history.

Based on the search function above, it is easy to implement the various particular query functions. One should of course remember that the backward search is faster that the forward search, since there are fewer links to follow in the backward direction.

Another problem is that the history structure node of the ch object will be constantly growing. This indicates that it might be interesting to divide the history structure nodes in to smaller parts. This is however a general system problem, since other objects can also get very big history structure nodes, and is therefore not considered in this work.

7.3 The work

When I started the final project, my plan was to spend one month on learning the system and design, one month on implementation, and one month on writing the report. As it turned out, each part took about one and a half months.

Some of the extended time was caused by mistakes made in the design phase, while others were caused by difficulties detected during the implementation. It was also harder than I thought to write a report in english.

One thing that I would change if I could do the work again is that I would test my ideas and solutions much more with the members of the IISLAB group to get fast feedback, ideas of improvements and to avoid bad solutions. This would probably have helped me to avoid the mistakes in the design phase.

Appendix A

In this appendix I describe my first attempt to represent the command history, it was based on storing forward links. I tried a solution very similar to the way the version history in LINCKS is implemented, and in particular the way the history graph is represented. Since the users' own command histories are stored as different versions of the session objects, the only arcs that has to be stored elsewhere are the connections between different user command histories.

Obviously forward links can not (in an easy way) be stored together with the root end versions, since when the root end version is stored, the arrow end version does not yet exist. Therefore we must use a different method that stores the links separate from the linked versions.

We use a link group called CHNET, which can be seen as a dynamic two dimensional array where each array entry represents all the incoming arcs to one command. Each element in the array entry is a bound link to a command. The positions in an array entry have different meanings: every position except the last holds links to commands that are the root ends for arcs, and the last position only holds a link to the command that all arcs is pointing to. See figure 1.

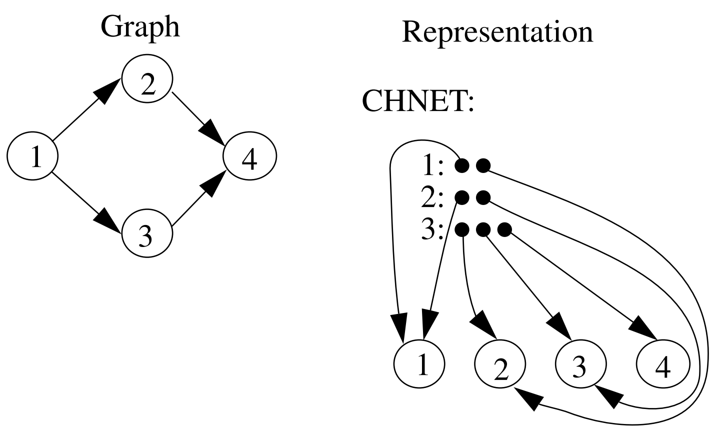


FIGURE 1 Graph representation in LINCKS.

We use the special object, the ch object, to store the array representation of the graph. In that object we also store a pointer to the front command, i.e. the command that was executed last (command 4 in figure 1). See figure 2 for a overview of where the different parts of the command history are stored.

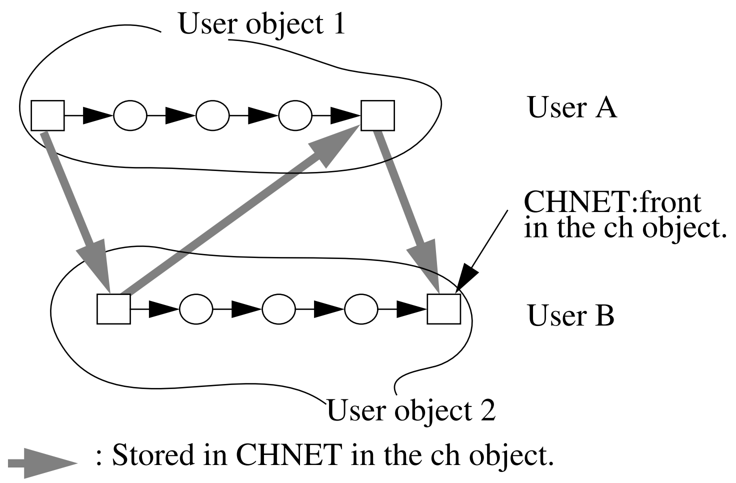


FIGURE 2 A figure on where the different parts of the command history are stored.

The ch object contains all forward links between database commands. This is a centralized representation. Each database command update of the history must retrieve the ch object and add the new links to the CHNET link group. If two or more users update the history in parallel, we get parallel versions of the ch object. This is a problem, since the next time someone wants to update the history, there are two versions of the command history that has to be merged together to give a single history version to use as basis for the update.

New commands are stored in the command history according to the following rule.

- For both workspace and database command: add a new version to the session history object and fill in the appropriate values.
- If it is a workspace command, then store the command node in the database and end the update.
- Otherwise, if there are parallel versions of the ch object, merge their CHNET link groups together to an intermediate version of the command history. It is newer actually stored in the command history, but is just used internally to construct the next ch object version.
- Next, add a new entry to the CHNET link group that contains the representation for the incoming arcs to the new database command.
- Finally, store the new ch object version.

The main problem with the forward linking is in the merge of the parallel versions of the ch object.

The simplest kind of ch object update occurs when there is only one version of it. First we update the session object by writing the new command node. This is done in the same way as for backward linking, see step A and B in figure 3. A ch object with only one latest version is updated by adding a new entry in CHNET. The first position in that entry points to the same object that CHNET:front points to, and the second position point to the new command object. Thereafter CHNET:front is set to point to the new command, see step C and D in figure 3.

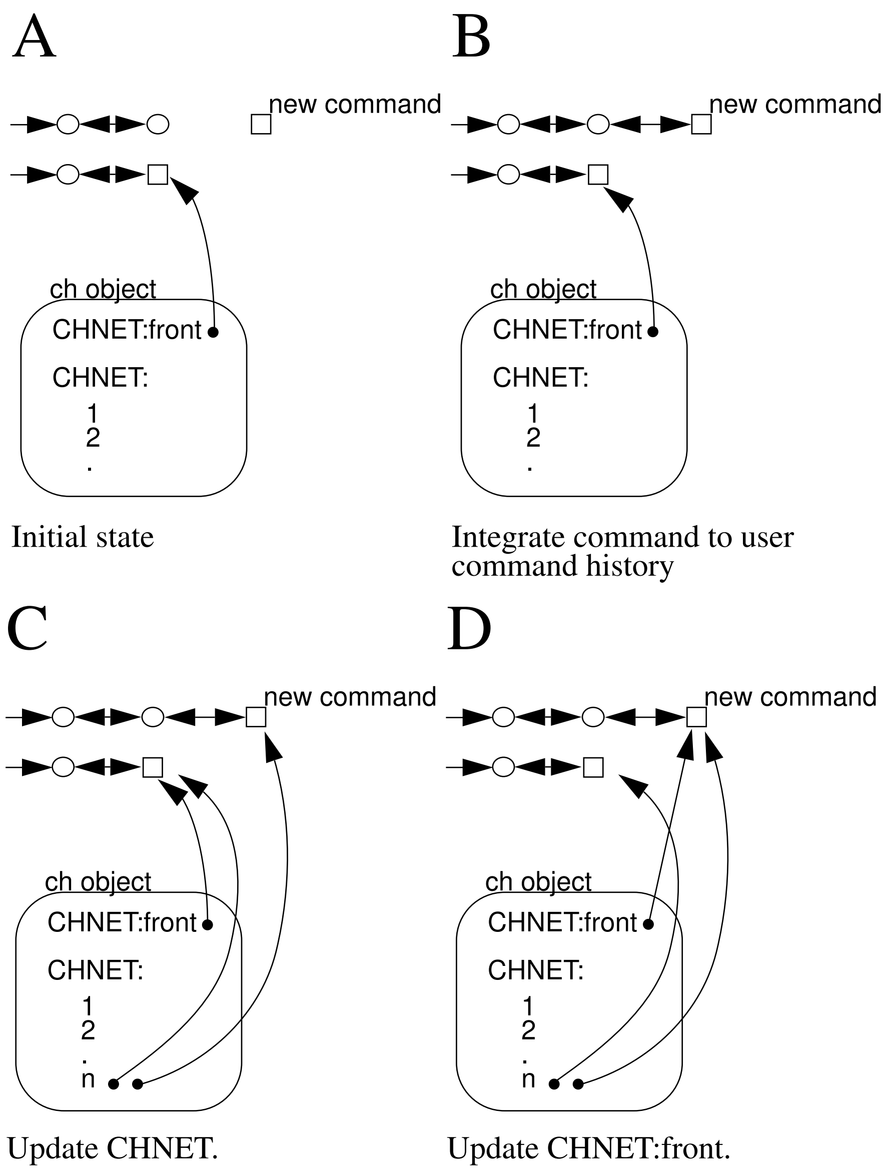


FIGURE 3 Updating the command history.

We now turn to the problem of how to update the history when two commands are executed in parallel, i.e.when there are more than one latest version of the ch object. Each of the different versions, X_1 to X_n, will have some information (arcs) that are unique and do not exist in any of the other versions, so what has to be done first is to construct an intermediate version that in essence is the union of X_1 to X_n, and which then can be used to add the new arcs to. The problem was how to make this union operation effectively.

For an example of how parallel versions of the ch object, and therefore of the command history, can occur see figure 4. Note that the graphs inside the nodes are the whole command histories and that the parts stored in the ch objects are only the non-horizontal arcs. The rest (both nodes and arcs) are stored in the different user session objects. The ch object versions A and B are parallel.

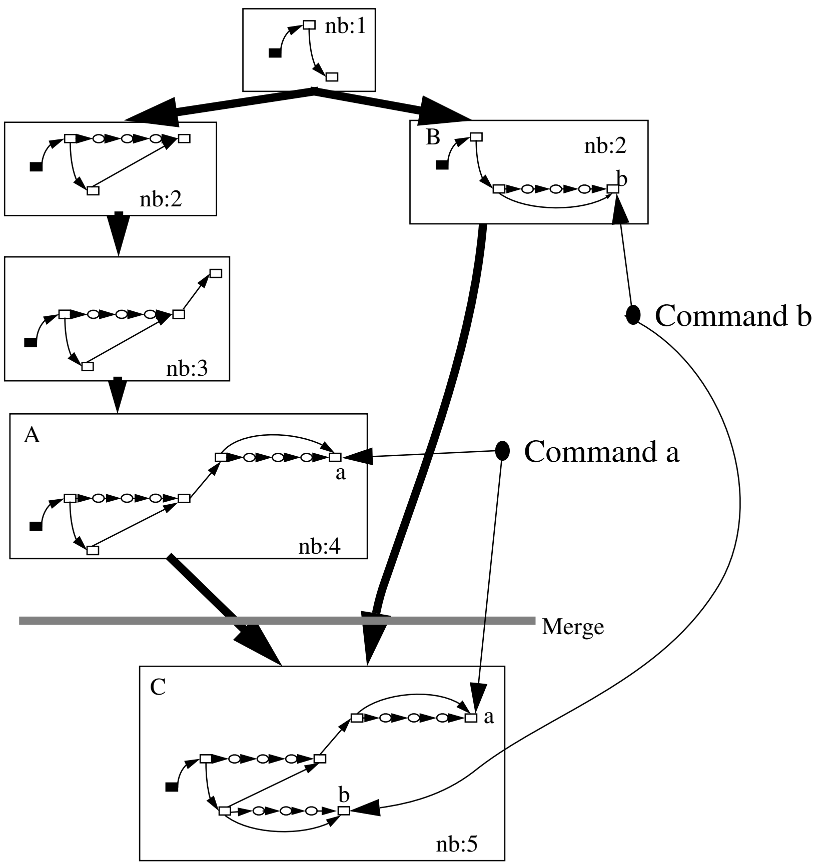


FIGURE 4 An example of how merge will work.

The operations that have taken place to get the situation in figure 4 are the following.

At the start, there is only one latest version of the ch object. One user executes some workspace commands and then a data base command (b). Before that user finishes the command history update, two other user execute some commands and finish with the data base command (a). Now there are two parallel versions of the command history, i.e. the ch object. To be able to both query the history and add new commands, we have to merge the parallel versions of the ch object together to one single version C.

To merge parallel versions of the ch object have I chosen the following algorithm.

The merge algorithm:

Let every ch object version have an attribute called number (nb) that is defined as follows. For the first version nb is zero. For any other version, it is the maximal nb amongst the parents plus one. The parents to a ch object version are the versions that were merged to get the new version. Nb is a measure of the longest path from the first version to any specific version, and it is also a measure of how much information a ch object holds.

Let X_1 - X_n be the parallel versions of the ch object.

Select one version as template, called $X_{template}$, according to this rule:

- Select the one with the highest nb.
- If several select the one with the most parents. This is the same as the one with the most incoming arcs to its front command.
- If there are several with the same nb and the same amount of parents, just pick one.

We have then picked the version that has the most information (biggest CHNET link group) to use as template. By this, we can minimize the amount of information that has to be copied to the template. Now we have to determine how much information (arcs) has to be copied from the parallel versions to the template. The most obvious solution is to use the following copy method:

Let X_p denote the version to copy from, and $X_{template}$ the version to copy to. The function front(X) returns the front command in the ch object X. Parent(x) returns the set of database commands that has arcs pointing to x.

1. let S = {front(X_p)}
2. if S empty end.
3. take a command $x \in S$
4. Let S = S -{x}
5. if there are no arcs pointing to x in $X_{template}$, then copy the entry in X_p that holds the arcs that points to x, else goto 2
6. Let S = S + Parent(x), goto 2

It is easy to see that this algorithm will do the work. For each command stored in the parallel version, we test to see if it is already incorporated in the template. If not, we add it and test its predecessor commands. This corresponds to a breadth-first-traversal of the parallel version history.

The particular disadvantage with this algorithm is that if the command under consideration is not in the template, we have to search through the whole template command history. That will take time and is even more annoying considering that the size of the command history is constantly growing.

One improvement that can be done, is to determine more exactly the measure on how many and which arcs shall be copied from the parallel version to the template. Looking at how the versions of the ch object correspond to each other, we observe two important properties. The first is that each version holds all information of its parent versions. The second is that the only arcs that have been added to one version, compared to its parents, are the incoming arcs to the front command. We can show that if the parents to X_p are all ancestors to $X_{template}$, then it is enough to copy the incoming arcs to the front command.

To simplify the proof, I have chosen to view the ch object versions as sets of arcs, and define functions that corresponds to the attributes and links of the implemented ch node. The notions we are concerned with are then denoted and interpreted as follows.

Merge:

If x and y are the versions that shall be merged, $x \cup y$ is the merged version of x and y.

The parent relation:

A version x is a parent to a version y, written parent(x,y), iff x was used in the merge to construct y.

The parent_set:

parent_set(x) = {y | parent(y,x)}

The ancestor relation:

A version x is an ancestor to a version y, written ancestor(x,y), iff there exists a sequence x_1 to x_n of versions such that each x_i (up to n-1) is a parent to x_{i+1}, and x_1=x and x_n=y.

The ancestor_set:

ancestor_set(x) = {y | ancestor(y,x)}

The front function:

Front is a function that for a given ch object version returns the latest executed command in that ch object version, i.e. it returns the value of the front link in a ch object version.

The incoming_arc function:

Incoming_arc is a function that for a given ch object version x and a command node y returns the subset of arcs that point to y. This is in reality done by a search in the CHNET link group.

The nb_count function:

Takes a ch object version and returns an integer according to the following:

nb_count($\emptyset$) = 0
nb_count(x) = max(nb(parent(x)) + 1

This is implemented as an attribute nb.

When an update of the command history occurs, it is divided in to two steps. The first step, merge, constructs one command history version of the parallel versions (if any). The last step, link in, connects the new command in the merged ch node. This is done by adding one arc from each front command and setting the new command to be the only front command.

Before the theorem we make some useful observations on how the ch versions correspond to each other.

Observation 1

$$\left(\bigcup_{parent(x,y)} x \right) \neq y$$

A command history update consists of both a merge and an addition of one new command. Therefore each node differs from the merge of parents. This leads directly to the following.

Observation 2

$$\left(\bigcup_{parent(x,y)} x \right) \cup Incoming\_arcs(front(y)) = y$$

Observation 3

$\forall x \forall y(ancestor(x,y) \rightarrow x \subset y)$

The history only grows.

Observation 4

$\forall x \forall y(\text{parent}(x,y) \rightarrow (\text{nb}(x) < \text{nb}(y)))$

Follows immediately from the definition of nb.

The template is selected, amongst the parallel versions according to the following rule:

- Select the version with the highest nb.
- If this selects several versions then pick those with the most parents. (This is the same as those with the highest number of incoming arcs to the front command).
- If there are several versions with the same nb and the same number of parents, then just pick one at random.

Now we are ready to prove theorem 1.

Theorem 1

It is sufficient to copy the incoming arcs to front(X_p) in X_p to $X_{template}$ to obtain a merged version of the two.

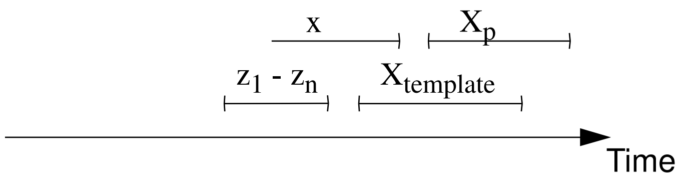
❑

Proof

By observation 1 and 2, we know that the only arcs in X_p that are not contained in any of its parents, are the incoming_arcs to front(X_p). It is therefore sufficient to show that all arcs that are in any of the parents to X_p are already in $X_{template}$, i.e we must show that the following holds:

Parent_set(X_p) $\subseteq$ ancestor_set($X_{template}$)

Assume the contrary. i.e. Parent_set(X_p) $\not\subset$ ancestor_set($X_{template}$). Then there is a node x in the parent set of X_p that is not in the ancestor set of $X_{template}$. This means in particular that the processing for x is not completed at the time when the processing of $X_{template}$ is begun, by completion we mean visible in the database, see figure 5.



z_1 -z_n are the parents of $X_{template}$

FIGURE 5 Time diagram.

Therefore, when X_p is formed, all parents of $X_{template}$ are completed, which implies that every parent of $X_{template}$ is an ancestor to X_p. From the definition of nb, we then obtain that nb($X_{template}$) $\leq$ nb(X_p), which by the first step in the template formation, implies nb($X_{template}$) = nb(X_p), and therefore also every parent of $X_{template}$ is a parent of X_p. By the second step in the template formation procedure, we then obtain that the case must be that $X_{template}$ and X_p has the same parents, which contradicts the initial assumption of a node x that is a parent of X_p but not a ancestor of $X_{template}$. Since the assumption leads to a contradiction, we must conclude the opposite, namely that every parent of X_p is an ancestor of $X_{template}$.

❑

So the copy algorithm is transformed to just:

- Copy the arcs, in X_p, that points to front(X_p) to $X_{template}$.

Further, since the incoming arcs to one node are stored together in one entry in the CHNET array, it is fairly easy to find the arcs that shall be copied. It is only a matter of finding the entry that holds the incoming arcs to the front command of X_p.

By using this latter algorithm, we copy the minimal amount of information from each version, since the copied information is the part that is unique for that version.

After a merge operation there will always be more than one front command; it will be exactly one front command for each parallel version. The set of front commands is the same as the set ‘Front’ introduced in chapter 3. So what to do to make a new version is just to add arcs from the front set to the new database command and reset ‘Front’ to hold only the new command.

There are two major disadvantages with the forward linking approach. First, the size of the ch object versions is constantly growing. Each version holds the information of all its ancestor plus some more, and thus the same information is stored in several places. Second, the number of ch object versions will also be very high; approximately one version for each database command.This leads to that the ch object history structure node becomes very large.

Appendix B

The two types of object in the command history.

The user history object.

Attributes

• Function	The name of the command.
• Type	Database or workspace command.
• Application	What application executed the command.
• From	Function parameter.
• To	Function parameter.
• Group	Function parameter.
• Field	Function parameter.
• Position	Function parameter.

Links

• Lbl	Function parameter.
• From	Function parameter.
• To	Function parameter.

The attributes func and type always have a value, the others depend on what command it is.

The command history object, ch object.

The forward representation.

Attributes

• Nb	A measure of how much information the version holds.

Links

• Front	A pointer to the latest executed command.
• After	A link group that contains the connections between the user histories.

The backward representation.

Links

• Front	A pointer to the latest executed command.

Appendix C

These are the commands that will be logged by the command history.

Database operations.

LOGIN()
LOGOUT()
SO_STOREOBJ(lbl)
COMPOSITE_ STORE()
COMPOSITE_EXPAND()

Workspace operations.

CO_CREATEOBJ(lbl)

Workspace operations if in workspace, else database operations:

CAGN_CHANGEATTRGROUPNAME(lbl,from,to)
CAN_CHANGEATTRNAME(lbl,group,from,to)
RAGT_REMOVEATTRGROUPTAG(lbl,group)
RAGV_REMOVEATTRGROUPVAL(lbl,group)
RAT_REMOVEATTRTAG(lbl,group,field)
RAV_REMOVEATTRVAL(lbl,group,field)
SA_SETATTR(lbl,group,val)
LINK(from,group,field,to,pos)
RLGT_REMOVELINKGROUPTAG(lbl,group)
RLGV_REMOVELINKGROUPVAL(lbl,group)
RLT_REMOVELINKTAG(lbl,group,field)
RLV_REMOVELINKVAL(lbl,group,field)
ULA_UNLINKAT(from,group,field,pos)
UL_UNLINK(from,group,field,to)
SI_SETIMAGE(lbl,val)
CPO_COPYOBJ(from,to)
CLGN_CHANGELINKGROUPNAMES(lbl,from,to)
CLN_CHANGELINKNAME(lbl,group,from,to)
RFL_REMOVEFROMLATEST(lbl)

Here is some get functions that will be stored as a common db_get operation. It will only be stored if it really was a database operation, i.e. if the object was not already in the workspace when the function was called.

GAN_GETATTRNAMES(lbl,group,mapfn,extra)
GAGN_GETATTRGROUPNAMES(lbl,mapfn,extra)
GA_GETATTR(lbl,group,field,val)
GI_GETIMAGE(lbl,val)
GLGN_GETLINKGROUPNAMES(lbl,mapfn,extra)
GLI_GETLINKITEM(lbl,group,name,pos,item)
GLN_GETLINKNAMES(lbl,group,mapfn,extra)
GLV_GETLINKVAL(lbl,group,name,mapfn,extra)
GL_GETLATEST(lbl,mapfn,extra)

Liblincks functions that will not be logged.

GSR_GETSYSTEMROOT(lbl)
GC_GETCURRENT(lbl)
SC_SETCURRENT(lbl)
UBL_UNBINDLABEL(lbl)
GVS_GETVERSIONS(lbl,vmapfn,pmapfn,extra)
INWORKSPACE(lbl)
RO_RELEASEOBJ(lbl)
EDITED(lbl)
BOUND(lbl)

Appendix D

Here is a list of what parameters are stored in the command nodes, for different commands.

Function	Func	Type	Appl	From attr	To attr	Group	Field	Pos	lbl	From	To
LOGIN	X	DB	?								
LOGOUT	X	DB	?								
SO_STOREOBJ	X	DB	?						X		
COMPOSITE_ STORE	X	DB	?								
COMPOSITE_EXPAND	X	DB	?								
CO_CREATEOBJ	X	WS	?						X		
CAGN_CHANGEATTRGROUPNAME	X	DW	?	X	X				X		
CAN_CHANGEATTRNAME	X	DW	?	X	X	X			X		
RAGT_REMOVEATTRGROUPTAG	X	DW	?			X			X		
RAGV_REMOVEATTRGROUPVAL	X	DW	?			X					
RAT_REMOVEATTRTAG	X	DW	?			X	X				
RAV_REMOVEATTRVAL	X	DW	?			X	X		X		
SA_SETATTR	X	DW	?			X	X		X		
LINK	X	DW	?			X	X	X		X	X
RLGT_REMOVELINKGROUPTAG	X	DW	?			X			X		
RLGV_REMOVELINKGROUPVAL	X	DW	?			X			X		
RLT_REMOVELINKTAG	X	DW	?			X	X		X		
RLV_REMOVELINKVAL	X	DW	?			X	X		X		
ULA_UNLINKAT	X	DW	?			X	X	X	X		
UL_UNLINK	X	DW	?			X	X	X			X
SI_SETIMAGE	X	DW	?						X		
CPO_COPYOBJ	X	DW	?							X	X
DB_GET	X	DB	?						X		
CLGN_CHANGELINKGROUPNAME	x	DW	?	x	x	x			x		
CLN_CHANGELINKNAME	x	DW	?	x	x				x		
RFL_REMOVEFROMLATEST	x	DW	?						x		

X	Has always a value.
?	Might have a value.
DB	Database command.
WS	Workspace command.
DW	Either a database or a workspace command.

Appendix E

Here is a short manual on the interface functions to the command history.

errcode CH_update(func, type, from_attr, to_attr, group, field, pos, lbl, from_lbl, to_lbl)

```
label *lbl, *from_lbl, *to_lbl;
char *func, *from_attr, *to_attr, *group, *field;
int   type, pos;
```

Updates the command history with the command named func. Type is the type of command, 0 for a database command and 1 for a workspace command. The rest of the parameters are the parameters to the function call.

errcode CH_relation(lbl_a, lbl_b, result)

```
label *lbl_a, *lbl_b;
int   *result;
```

Examines the relation between the commands lbl_a and lbl_b. It sets result to:

Before	if lbl_a was before lbl_b.
Unrelated	if lbl_a and lbl_b was unrelated.
After	if lbl_a was after lbl_b.

Before, unrelated and after are implemented as constants with the integer values 1 and 0 respectively -1.

void CH_set_application(app)d

```
char *app;
```

Sets the application of the system. Will be stored in each command node. If it isn’t explicitly set it will stored as ‘Unknown’.

errcode CH_get_user_process_object(lbl)

label *lbl;

Returns, in lbl, the label to the node where the next command will be stored. Is only used by SO_STOREOBJ to create the link from the created object to the creating command in the command history.

errcode CH_init()

Initializes the command history. It creates the ch node and a 'no command has been executed'-node (the null-command node). It then makes a link from the ch node's first version to the 'no command has been executed'-node, called front. Last it creates an unbound link in the system root's history structure node that points to the ch object.

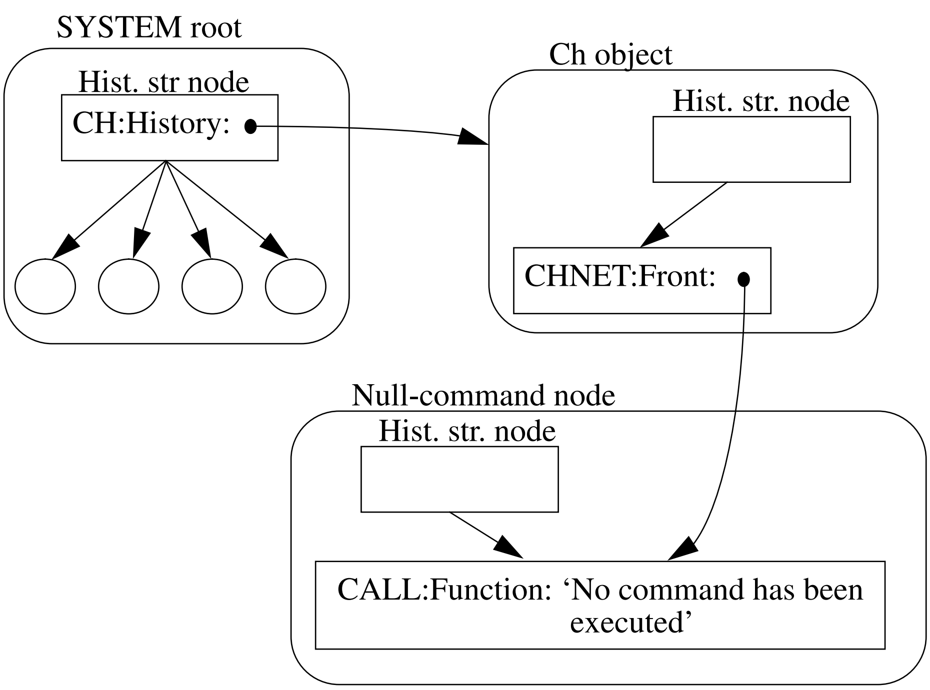


FIGURE 1. Initialized state of the command history.

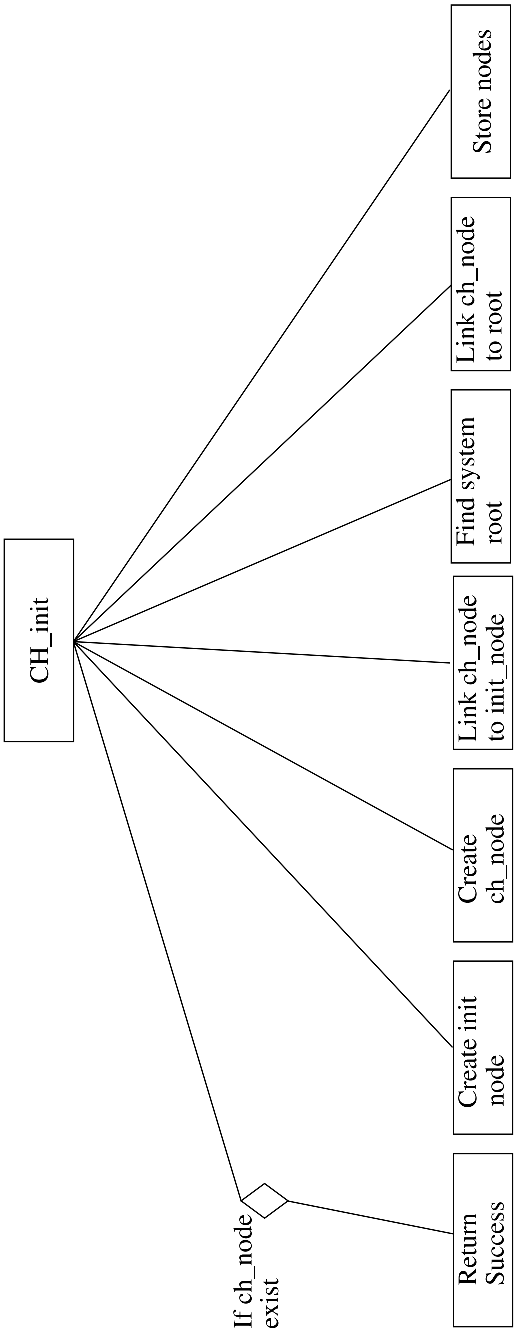
errcode CH_lbl_ok(lbl)

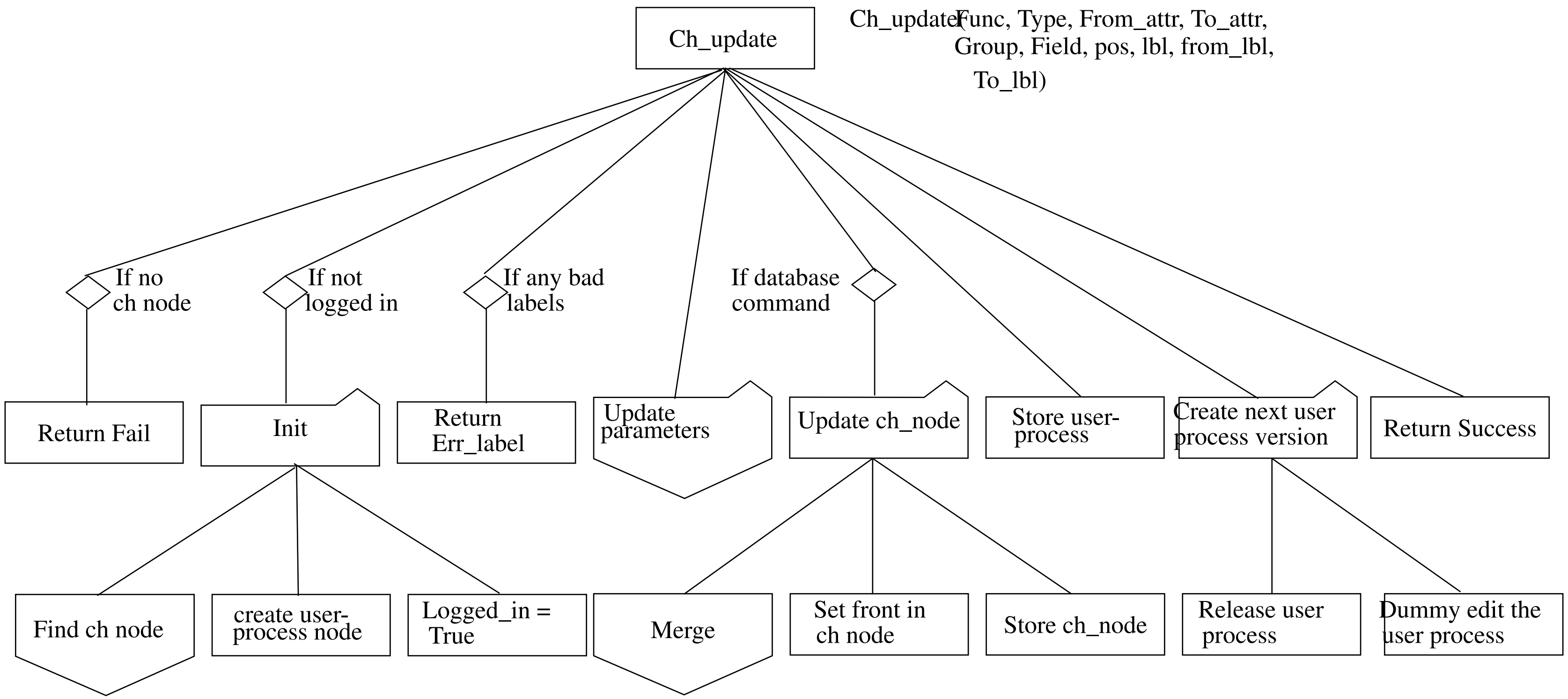
label *lbl;

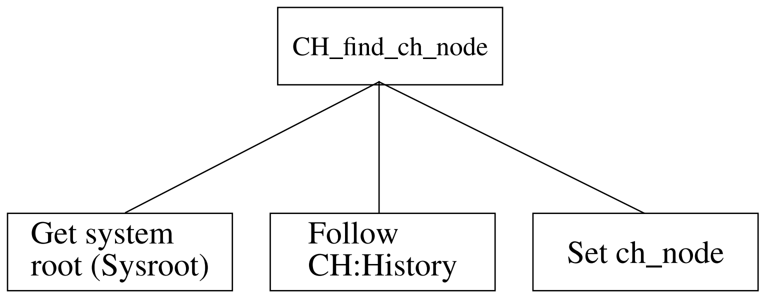
Test if the lbl is equal to the user process node or the ch node. Returns ERR_LABEL if the lbl is not ok.

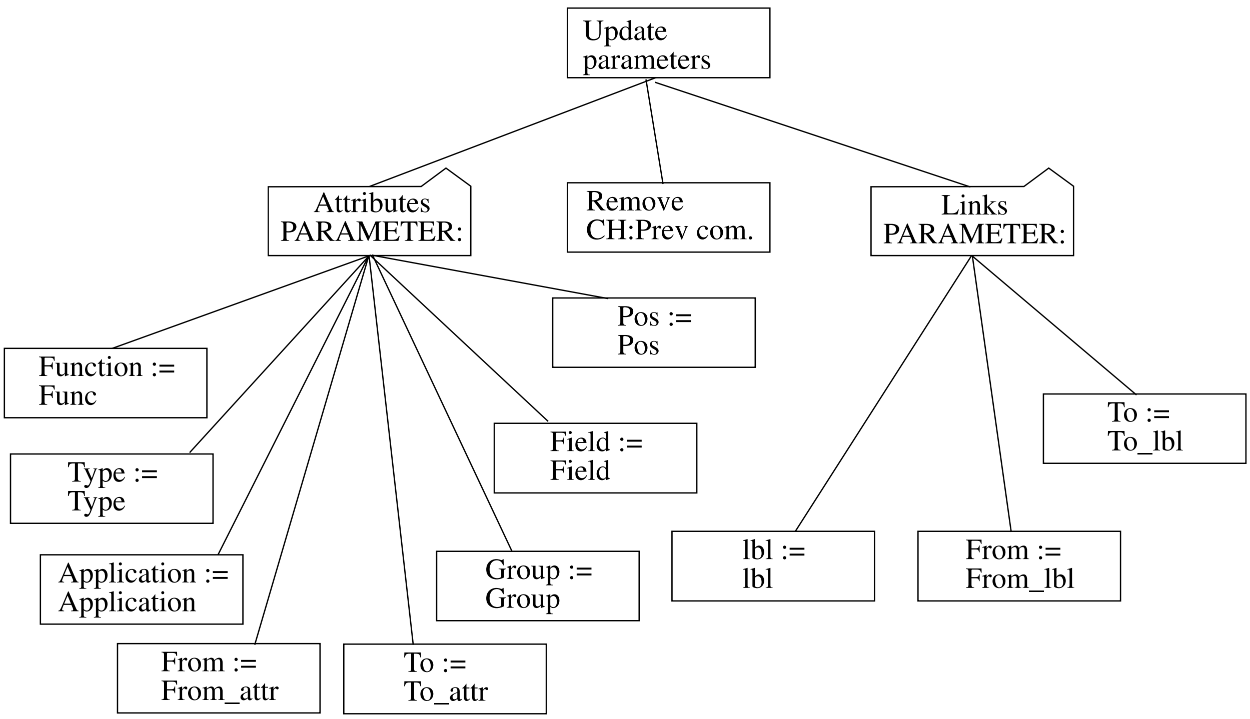
Appendix F

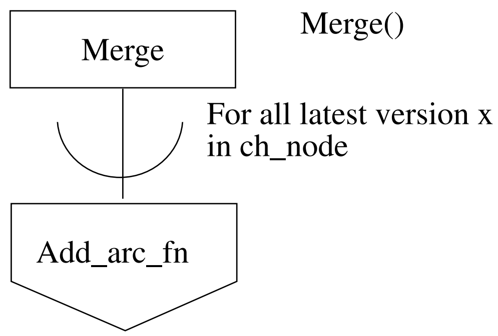
Here I present the structure charts of the central functions of the command history handler.

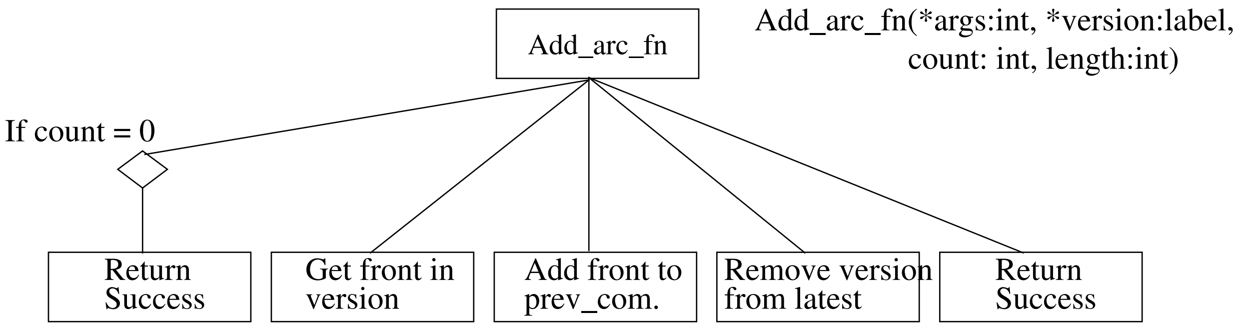


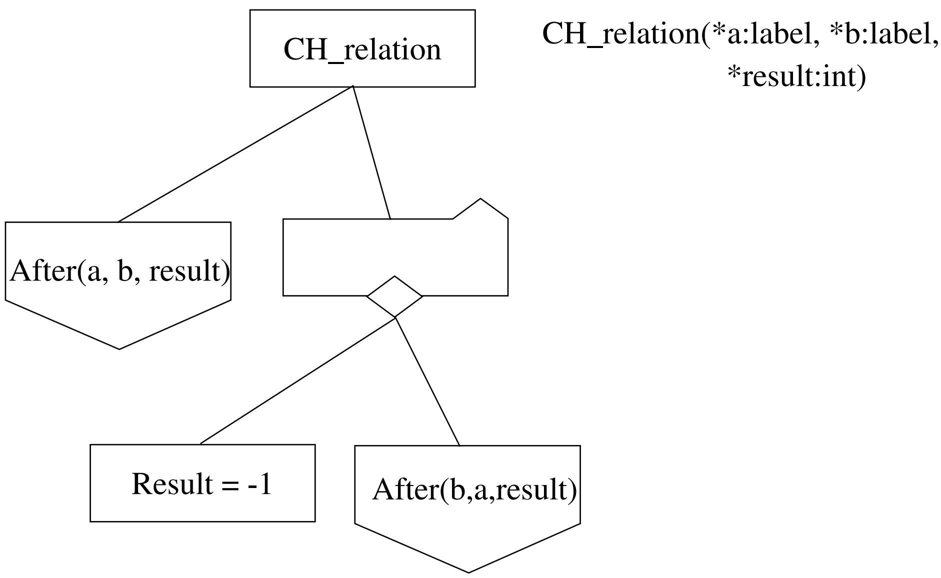

Ch_update
Ch_update(Func, Type, From_attr, To_attr, Group, Field, pos, lbl, from_lbl, To_lbl)
If no ch node
If not logged in
If any bad labels
If database command
Return Fail
Init
Return Err_label
Update parameters
Update ch_node
Store user-process
Create next user process version
Return Success
Find ch node
create user-process node
Logged_in = True
Merge
Set front in ch node
Store ch_node
Release user process
Dummy edit the user process

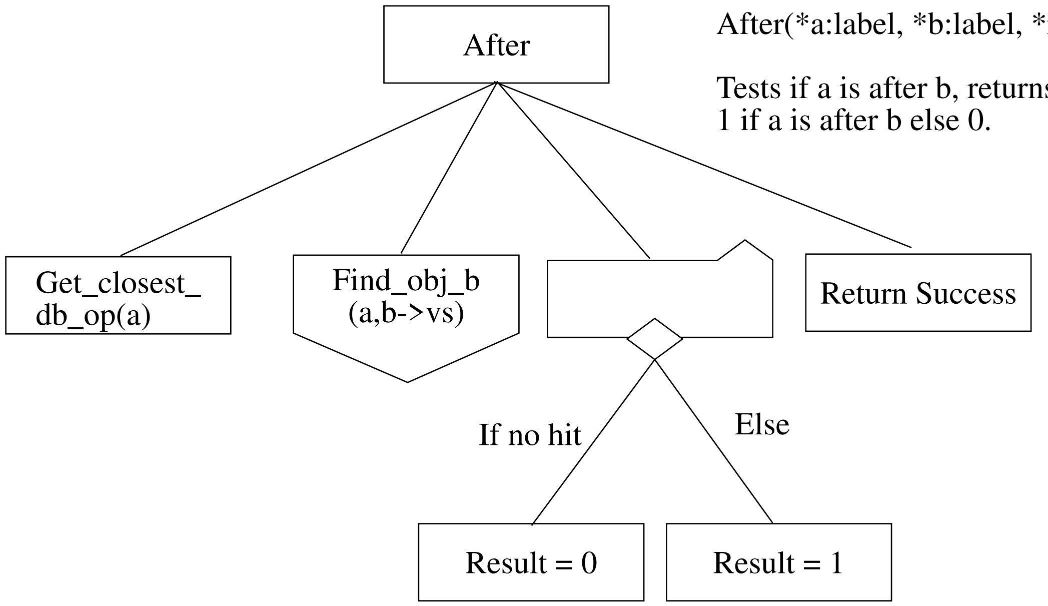

CH_find_ch_node
Get system root (Sysroot)
Follow CH:History
Set ch_node


Update parameters
Attributes PARAMETER:
Remove CH:Prev com.
Links PARAMETER:
Function := Func
Type := Type
Application := Application
From := From_attr
To := To_attr
Group := Group
Field := Field
Pos := Pos
lbl := lbl
From := From_lbl
To := To_lbl


Merge()
Merge
For all latest version x
in ch_node
Add_arc_fn

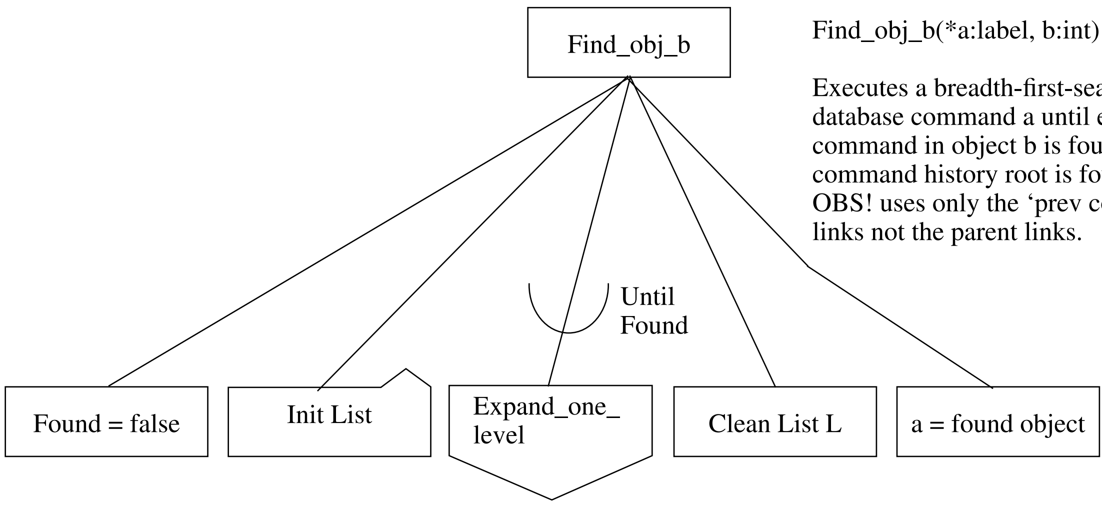

Add_arc_fn(*args:int, *version:label,
count: int, length:int)
Add_arc_fn
If count = 0
Return
Success
Get front in
version
Add front to
prev_com.
Remove version
from latest
Return
Success


CH_relation(*a:label, *b:label,
*result:int)
CH_relation
After(a, b, result)
Result = -1
After(b,a,result)



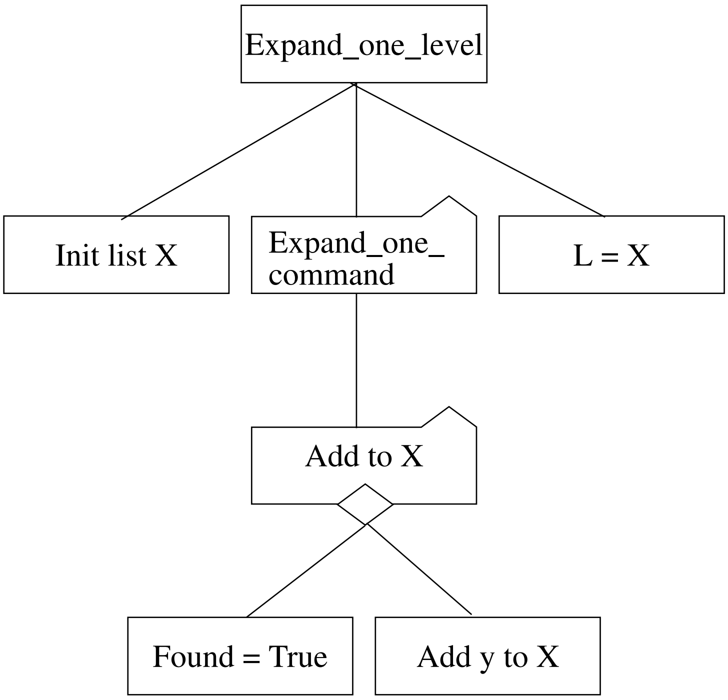
After(*a:label, *b:label, *result:int)

Tests if a is after b, returns
1 if a is after b else 0.



Find_obj_b(*a:label, b:int)

Executes a breadth-first-search from the database command a until either one command in object b is found or the command history root is found.
OBS! uses only the ‘prev command’ links not the parent links.


Expand_one_level
Init list X
Expand_one_
command
L = X
Add to X
Found = True
Add y to X

References

[Lam92] Lambrix, P., ‘Versioning of Strongly Connected Composite Objects’, Research Report, LiTH-IDA-R-91-14, Linköping University, 1992.

[Man88] ‘Application Interface to LINCKS on Sun; Programmers Manual’, LINCKS Project Group, Department of Computer Science, Linköping University, 1988.

[Pad86] Padgham, L., ‘ LINCKS: An Imperative Object Oriented System’. in *Proceedings of the 20th Hawaii International Conference on System Sciences,*vol 1, pg. 516.

[Pad88] Padgham, L., ‘NODE: a database for use by intelligent systems*’,* in *Proceedings of the International Symposium on Methodologies for Intelligent Systems* - ISMIS 88, pp 190-199, Torino,Italy, 1988.

[Sno86] Snodgrass, R., ‘Temporal Databases’, in *Computer*, September 1986, pp 35-42, 1986