VerSE: Towards Hypertext Versioning Styles

Anja Haake & David Hicks
GMD – German National Research Center for Information Technology
IPSI – Integrated Publication and Information Systems Institute
Dolivostr. 15, D - 64392 Darmstadt, Germany
E-mail: {ahaake, hicks}@darmstadt.gmd.de

ABSTRACT
Much of the previous work on version support for hypertext has focused primarily on the development of functionality for specific hypertext systems and/or a specific hypertext application domain. Although these models address crucial version support problems in specific hypertext application domains, they cannot be easily adapted and then integrated into other hypertext applications.

Hypertext version support environments have been introduced to help alleviate these problems. They are designed to meet the version support needs of a wide range of hypertext applications. However, so far few high level versioning facilities have been constructed in these environments, creating a gap between the facilities provided directly within the environment and the versioning needs of some applications.

The intent of this research is to bridge this gap. It turned out that task-based versioning styles are easy to use by both hypertext application developers and hypertext application users. As shown in previous work, task-based versioning helps to alleviate cognitive overhead and disorientation problems for users. In addition, it requires little investment from the point of view of application development, since task-based versioning does not necessarily require an application to incorporate an extra notion for individual versions. This paper presents a set of task-based hypertext versioning styles that are offered in the VerSE flexible version support environment and shows the direction towards the design of additional versioning styles.

KEYWORDS
version support / control, version support environment, versioning styles / policies, task-based versioning

Hypertext '96, Washington DC USA
© 1996 ACM 0-89791-778-2/96/03..$3.50

INTRODUCTION
As research efforts into the versioning problem in hypertext have progressed, many researchers have focused on the development of version support functionality for specific hypertext systems or specific hypertext application domains [5, 7, 24]. Several of the version models resulting from such efforts address crucial version support issues such as the cognitive overhead problem encountered during version creation and the disorientation that can occur during version selection [7, 24]. However, these models are tuned to support specific application domain requirements and cannot be easily adapted and integrated into other hypertext applications.

Other researchers, attempting to widen the applicability of their results, have concentrated on the development of general hypertext version support environments [14, 30]. The versioning functionality offered in a general hypertext version support environment can support the needs of a wide range of hypertext applications. However, to maximize their generality, the versioning services directly offered in such an environment are typically low level. This has created a gap between the version facilities offered within the environment and the needs of applications. The introduction of an intermediate layer of versioning functionality between the hyperbase and application layers has been suggested as a mechanism to bridge this gap in general hypertext version support environments (Figure 1) [15]. While this approach appears promising, so far few intermediate layer versioning facilities have been constructed.

Aiming at bridging this gap, we have designed and implemented the VerSE flexible version support environment. VerSE offers a set of flexible low and high level version support facilities. To meet the needs of a wide variety of applications, an underlying design principle for the VerSE versioning facilities was to be capable of being flexibly combined to create versioning styles or policies. Another important goal was to maximize the cost/benefit ratio for the integration of the resultant versioning styles into applications. Therefore, the VerSE versioning concepts were designed to be used in a variety of ways to create versioning styles that meet specific versioning needs and are simple to integrate into applications. The different versioning styles are maintained by VerSE and thus help to bridge the gap between the

versioning facilities typically provided by hypertext version support environments and the needs of applications.

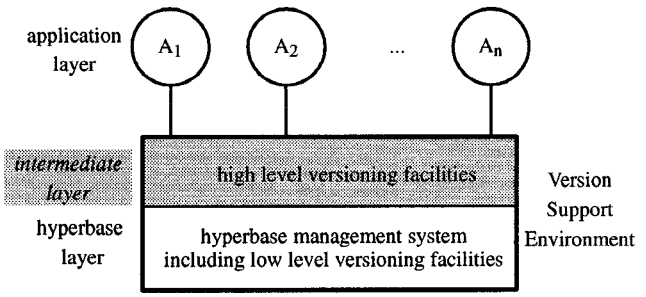

Figure 1: General hypertext version support environment

To emphasize the applicability of our results to other hypertext version support environments, we describe the VerSE versioning concepts and versioning styles from a general perspective. After discussing the current status of version support mechanisms for hypertext, the general hypertext version support concepts are introduced. This is followed by a discussion of how the concepts have been used to develop several hypertext versioning styles. Next our implementation of the general version support facilities and the versioning styles in the VerSE version support environment is described. The research is then compared to related work, followed by a brief conclusion and discussion of future research.

RELATED WORK

Versioning is a key problem in many applications, especially hypertext ones. Since described by Halasz [12, 13], the version control problem in hypertext has started to receive increasing attention. Version control was identified as a key topic for hypertext databases during the NSF Workshop on Hyperbase Systems in 1992 [18] and at the Hypertext'93 Workshop on Hyperbase Systems in 1993 [17]. In September of 1994 the first workshop on versioning in hypertext systems was held in connection with ECHT'94 [3].

The spectrum of version support proposals for hypertext ranges from version models implemented in a monolithic hypertext system [5, 24, 16, 19, 22, 29] to work concentrating on general version support environments like HB3 [15] and Hyperform [30]. While the former consider versioning issues in the context of a specific application area, the latter are intended to support a variety of hypertext applications. Between the extremes along this spectrum version models have emerged offered in the form of a version server [1], often for specific hypertext application domains [7, 8]. Still others have focused on the needs of distributed hypertext systems [20, 21].

Abstracting from the system architecture, it is possible to classify version models into the state-based and task-based categories. State-based version models maintain the versions of an individual hypertext object in a data structure often called a version set [7, 15, 24, 26, 29, 30]. When a versioned object is referenced in a state-based versioning environment, the reference must explicitly identify a specific version of the object.

A problem frequently encountered with the basic state-based versioning approach is that it is not directly possible to track a coordinated set of changes involving several components of a hypertext network. Additional layers of functionality are required to provide this capability. For example, a bug fix may involve several software files or modules. In the basic state-based approach, it is not possible to easily and directly identify the specific objects (and versions of those objects) that represent such a coordinated change. The composition of versions of complex objects, such as a hypertext composite node, requires the identification of specific versions of its constituent objects, and can thus be error prone. In particular, basic state-based versioning can aggravate the cognitive overhead and disorientation problems encountered during version creation and version selection in hypertext systems [7, 24].

While state-based versioning focuses on maintaining individual versions of individual objects, task-based versioning focuses on tracking versions of complex systems as a whole. The central idea of the task-based approach is to provide system support for maintaining the relationships between versions of objects that have been changed in a coordinated manner during the performance of a task, and to assure that those versions can be later identified as having implemented the set of coordinated changes. In this way, unintended combinations of component versions can be ruled out a priori and version selection can be done via change information. Thus, task-based versioning in particular eases the version creation and version selection problems. Additionally, task-based versioning concepts are often easier to integrate into applications. Since task-based versioning inherently provides a default version creation mechanism along with a mechanism to track coordinated changes made to a hypertext network, application code does not have to be extended with additional versioning creation and identification operations in order to use this functionality.

The Personal Information Environment (PIE), a software development environment for Smalltalk, was one of the first task-based version models [6]. The PIE model has been applied directly to provide version support for a hypertext system that offers atomic nodes and binary links [25]. The version model of the HyperPro system [24] represents a step towards task-based version support. It allows each composite node to have its own selection criteria that indicates which versions of objects will be selected for the generic links the composite contains. In the Palimpsest version model, a task-based approach supports cooperative editing applications [2]. RHYTHM uses a task-based approach to provide version support in a distributed hypertext system [20, 21]. CoVer [7] integrates a fully state-based version support model including versions of links and composites with a work flow oriented task concept influenced by PIE's layers and contexts.

These task-based version models are all integrated into monolithic hypertext systems and are not accessible for use

by different hypertext applications. To a certain extent, CoVer is an exception since it offers its version support functionality in the form of a version server. However, the version model is very much influenced by the requirements of hypertext publishing applications.

An underlying design principle during the course of this research was for the resultant versioning concepts to be capable of being flexibly combined to create versioning styles to meet the versioning needs of a wide variety of applications. Another important goal was to maximize the cost/benefit ratio for the integration of version support into applications. While task-based versioning offers benefits that help to ease user interaction with versioning services and maximize the cost/benefit ratio for integration into applications, general version support architectures provide a flexible environment in which versioning services can be adapted to meet the needs of specific application areas. The versioning concepts developed during the course of this research are based on a combination of both approaches. Existing task-based version models are built upon to develop easy to use concepts that can be easily integrated into applications. Additionally, the concepts are sufficiently general to be appropriate for integration into a general hypertext version support environment.

A GENERALIZATION OF TASK-BASED VERSIONING CONCEPTS

Many design decisions are required when integrating task-based versioning concepts into a hypertext version support environment. The choices made influence both the range of applicability of the versioning functionality and the cost/benefit ratio experienced by application programmers as they integrate versioning functionality into their applications. The design decisions made to address the following three issues are particularly important.

Issue 1 – The first issue concerns how many versions of a particular versioned object are to be accessible at a time (exist within the scope of a single task). If only one version of each versioned object is allowed, interaction with the versioning services is simplified for both application developers and users. The task concept can be used directly for version identification since only one version of each object will be present within the scope of any single task. However, this prevents the possibility of presenting and working with alternative versions within the same task. For example, the PIE system limits the number of valid versions to one at any particular time. In contrast, the CoVer system allows multiple versions or alternatives to exist within a task.

Issue 2 – The second issue is how to define the set of versioned objects that are subject to change within the scope of a task. This can be a fixed set that is determined by some default mechanism or an adjustable set that is defined by the application program. When the set is fixed, application interaction is simplified since no extra functionality is required to define the set of versioned objects to be included within the scope of a task. For example, in the PIE system, a layer represents an entire Smalltalk image and includes all objects in the programming environment. In contrast, the CoVer system allows versioned objects to be explicitly included in or excluded from a task. Additionally, CoVer limits the default versions automatically included in newly created tasks to those of predecessor / successor and super / sub tasks.

Issue 3 – Task organization and composition is the third issue. In some systems, completed tasks can be flexibly combined to construct new system configurations. This capability is often found in systems that support the automatic derivation of a system configuration, like PIE. However, this approach requires the application developer or user to define consistency and compatibility constraints among different tasks. In contrast, the organization of tasks (allowable composition) in other systems is inherently defined in the versioning concept so that no extra effort for task organization is required. This approach is especially useful when the primary emphasis of version support is to keep the history of changes to a system and provide meaningful cues for version identification. An example of this approach is the creation of change tasks within the context of hierarchically structured application tasks in CoVer.

As mentioned earlier, two important underlying goals of this research were to maximize both the applicability of versioning functionality and the cost / benefit ratio for its integration. These goals influenced the mechanisms selected to address each of the issues. Specifically, we developed the following general task-based versioning concepts.

Two levels are identified within the design: the *change control level* and the *task description level.* The change control level is where actual versioning related activities are performed within the environment. It is appropriate for integration into the hyperbase level of a general hypertext version support environment (Figure 1). The change control level uses a general *change task* to log the changes that are made to a system as development proceeds. The simplest type of change task supports only a single version of a versioned object within the scope of a task (issue 1) and by default applies to an entire network of hypertext objects managed by a hyperbase management system (issue 2).

The task description level provides a mechanism to organize the change tasks at the change control level. To reduce the effort required to maintain the organization of tasks, we chose to incorporate mechanisms for specifying application work flow and history documentation policies into the versioning concepts (issue 3). Various work flow patterns for different applications can be specified at the task description level. The *application task* is the primary mechanism used for organizing change tasks and expressing work flow patterns. Application tasks allow change tasks to be organized into groups that correspond to the various subtasks that are performed during the course of completing the larger overall task. As application tasks are performed, a history of the evolving system is maintained at the change control level. The functionality of the task description level is appropriate

for integration into the intermediate layer of a general hypertext version support environment (Figure 1).

Both levels are described in more detail in the following two sections.

Change Control Level

The change task is the basic unit of versioning defined within the environment. It is the mechanism used to actually preserve intermediate states of an evolving hypertext network as development proceeds. The change task is intended to represent a snapshot image of an entire hypertext network. Several different types of change tasks can be defined at the change control level. We have identified two basic types, the *linear change task* and the *parallel change task*, that are useful in supporting the objectives of our versioning styles.

To simplify application interaction with and integration of the versioning styles, both linear and parallel change tasks contain only a single version of each versioned hypertext object (issue 1). This frees the application from having to cope with alternative versions or even be aware of the existence of multiple versions of objects. As new versions of objects are created they are automatically associated with and placed into the context of the appropriate (currently active) change task. In this way, versions of objects can later be identified by simply specifying a task. Since each task contains only a single version of any particular versioned object, the task specification alone is sufficient to uniquely identify the appropriate version of the object. No extensions are required to applications forcing them to be aware of multiple versions of objects.

To provide a mechanism for defining the set of versioned objects included within the scope of a task, the *root task* concept is defined. The root task collects the set of objects created as the application proceeds thus defining the application data space. When created, a change task includes all of the objects maintained by the root task , thus providing a default set of objects for the task (issue 2). In addition, the root task provides an entry point to the application's task structure.

Linear change task – The linear change task is the simplest type of change task defined at the change control level. It is intended to be used by versioning styles supporting applications that require basic versioning functionality. A linear evolution history is supported for linear change tasks. Every linear change task, except the initial one of the application data space, has exactly one predecessor linear change task. Every linear change task may have at most one successor linear change task. A linear change task always inherits the previous state of the application data space from its predecessor linear change task as the starting point for further changes. Each linear change task stores only the updates or differences between its own version of the application data space and that of its predecessor linear change task.

Two states are defined for linear change tasks: active and frozen. When created, a linear change task is placed into the active state. When a successor linear change task is defined, the predecessor linear change task automatically becomes immutable to preserve the state of the system with respect to that task. Such a linear change task is said to be frozen. An explicit freeze operation is also provided to allow a linear change task to be frozen independent of creating a successor linear change task. If necessary, a successor linear change task can later be created for a frozen linear change task so that development can continue.

Parallel change task – The parallel change task is intended to support versioning styles used by applications with more elaborate versioning and parallel development needs. A directed acyclic graph (DAG) evolution history is supported for parallel change tasks. Each parallel change task may have several successor parallel change tasks and, except for the initial parallel change task for an application data space, several predecessor parallel change tasks. As with linear change tasks, parallel change tasks can either be in the active or frozen states. As successor parallel change tasks are created, the predecessor parallel change tasks are frozen to preserve the state of the system with respect to those tasks. Alternatively, parallel change tasks can be frozen with an explicit freeze operation and, when necessary, successor parallel change tasks can later be created for frozen parallel change tasks to allow development to continue.

When a parallel change task is created that has multiple predecessor parallel change tasks, a merge operation is required to produce the initial state of the system for the newly created parallel change task. To support different application types that may have different cooperation needs, flexible merge concepts influenced by the ideas of [23] have been adapted and integrated as a basic mechanism to support the merging of parallel change tasks. Further details of this adaptation and integration can be found in [10] and [11].

The primary reason to support linear change tasks as well as parallel change tasks is to ensure that a simpler form of interaction with applications is possible. The interaction required for creating a linear history or a sequence of snapshots of the application data space is minimal. A simple command is sufficient to define one linear change task after the other. Although no parallel development is supported with this type of interaction, no merge operations are required, ensuring simpler interaction with applications using this functionality.

Additional types of change tasks are possible. For example, if an application is capable of interacting explicitly with multiple versions of an individual object, change tasks supporting alternatives may be introduced at the change control level. Applications could then merge or work with the alternative versions (issue 1). Further support for structuring workspaces is also possible. For example, if applications are able to indicate which objects should be included in or excluded from the context of a task, a change task supporting these capabilities could be defined (issue 2). A change task

supporting both of these features would correspond to the original change task concept of the CoVer system. However, due to space limitations, only the two types of change tasks described above are considered in this paper.

Task Description Level

The general application task is defined at the task description level to support version control at different levels of detail and abstraction. Application tasks are primarily used to describe structured work patterns and organize the versions of an application data space that are produced at the change control level (issue 3). Application tasks keep track of the development history in the context of different work flow models. They can be used to group change tasks into higher level units according to the higher level workflow specification.

To support a variety of different application work flow and history documentation policies, the application task concept at the task description level is a general one. Application dependent work policies can be defined by refining the general application task concept. Through the definition of relationships among application tasks, it is possible to express the details of a particular work process or job. The degree of expression allowed between application tasks at the task description level can vary among versioning styles, and determines the potential elaborateness of the work patterns that can be specified at this level.

Similar to change tasks, operations are required within the environment for application tasks. These operations support the specification of the work patterns required to perform a job, the subsequent traversal through an application task hierarchy as the job is performed, and the creation of change tasks to preserve the state of the hypertext network at significant points during the course of the job. Additionally, like change tasks, application tasks can have a notion of state indicating their current status or level of activity in the development process.

It is possible to define a variety of versioning styles with the three general concepts (1) linear change tasks, (2) parallel change tasks, and (3) application tasks. The legal combinations of application and change tasks, the allowable relationships among application tasks, and the status set of application tasks may be refined to create new versioning styles. Additionally, each particular versioning style can have its own set of operations and required behaviors for those operations from which applications may select. We will introduce and discuss meaningful versioning styles based on these concepts in the next section.

TASK-BASED VERSIONING STYLES

The versioning concepts described in the previous section can be combined in a variety of ways to develop useful versioning styles. Building upon the basic change task concept, both linear and parallel versioning styles can be created. We have designed and prototyped three versioning styles, two linear and one parallel. Each of the styles is intended to support a different class of versioning needs. They are all based directly on refinements of the general change task and application task concepts. As each versioning style is described, the VerSE graphical version browser is used to illustrate its functionality. Note, applications could also utilize other tools to display and manipulate task structures in other ways, or apply versioning style functionality to directly manipulate and/or display these structures themselves.

Linear Versioning Styles

Linear versioning styles are intended to support applications that require only basic version control functionality. As implied by the name, linear versioning styles support a linear evolution history for an evolving system. We have designed two linear versioning styles. Their primary difference is in their work process specification capabilities.

The first linear versioning style is intended to support applications that do not require work process specification functionality. In this style, the root task provides an initial linear change task for an application. As the application progresses, freeze commands can be issued (by the application or the application user; cf. Discussion section) to request that the current state of the hypertext network be preserved. A new linear change task will then be created in which subsequent updates and changes to objects can be made. The root task thus maintains a series of linear change tasks for the application. Since workflow specification is not supported in the linear versioning style, only a root task is required at the task description level.

Figure 2 illustrates a screen image of the graphical task browser displaying an instance of this linear versioning style. As indicated in the figure, the overall task being performed is that of writing a paper. A number of linear change tasks have been created representing snapshot images of the hypertext network as the task proceeds. Each linear change task is labeled with the date and time of its creation and has a link identifying its predecessor linear change task. Linear change tasks are linked to the root task by "child" links.

The hierarchical linear versioning style expands upon the capabilities of the linear style to provide support for work process specification. It is intended to support those applications that require some work process specification capabilities, but do not require support for parallel object evolution patterns.

The change control level of the hierarchical linear versioning style is similar to that of the linear style (cf. Figure 3). A simple linear version thread is supported to capture the evolution of the hypertext network. The task description level of the hierarchical linear versioning case is structured into a tree of application tasks that allows the specification of work patterns (by the application or the application user; cf. Discussion section). The tree is rooted by a single root task that represents the overall job to be performed (writing a paper, cf. Figure 3).

Application tasks can be defined recursively to represent a decomposition of the work processes associated with the

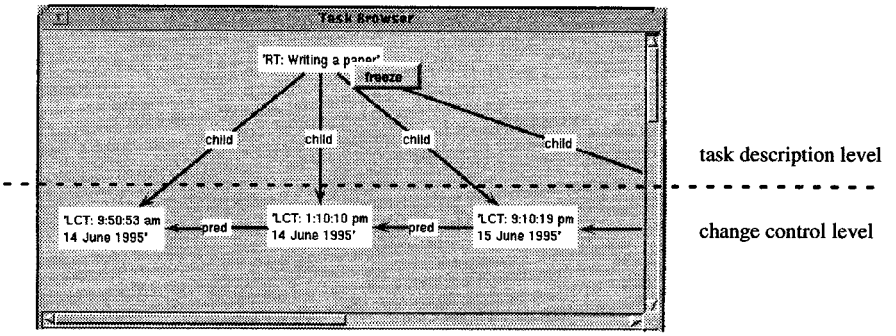

Figure 2: Task Browser displaying the first linear versioning style. The dashed line superimposed over the screen image separates the root task (RT) from linear change tasks (LCTs).

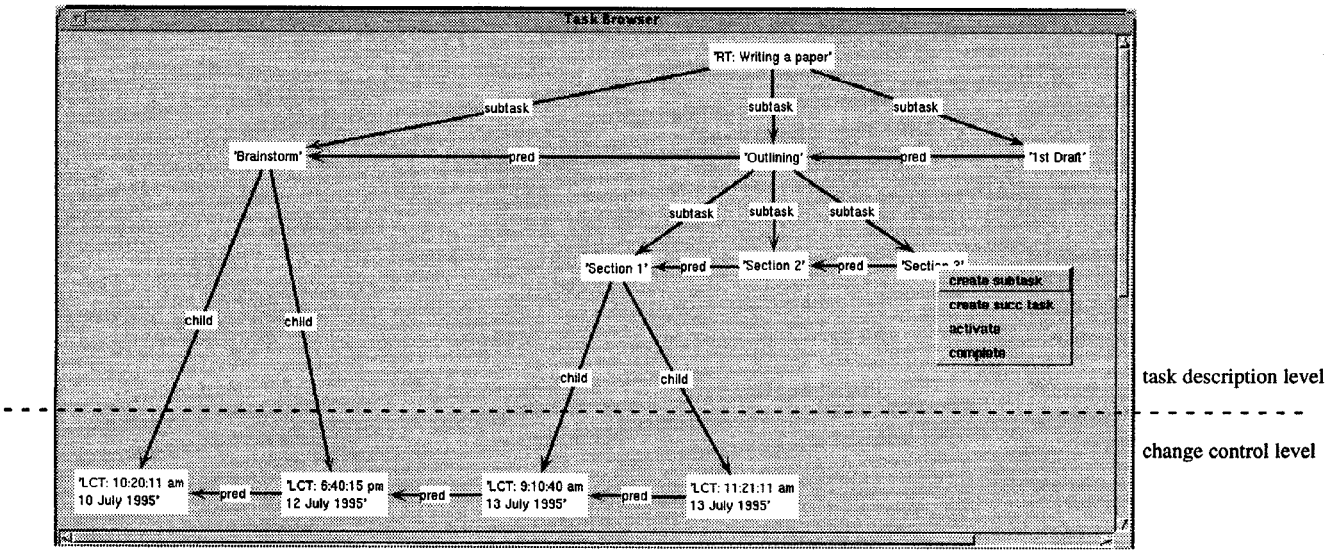

Figure 3: Task Browser displaying hierarchical linear versioning style. The dashed line superimposed over the screen image separates the root task (RT) and application tasks from linear change tasks (LCTs).

overall job. The application tasks linked by a "subtask" link to an application task in this tree represent the component tasks that are required to perform the task. Application tasks can be defined to any required depth allowing work patterns to be specified to any desired level of granularity. Parallel work activities are not supported in the hierarchical linear versioning style. As indicated by the predecessor links in the figure, there is a left to right ordering of sibling application tasks in the application task tree, corresponding to the temporal sequence in which tasks should be performed to accomplish the overall job. For example, in Figure 3, the author is currently outlining a paper based on the results of an initial brainstorming session. Note, the task hierarchy shown in Figure 3 represents the writing style of a specific author. Other organizations are, of course, possible.

Three operations are defined for application tasks in the hierarchical linear versioning style: create, activate, and complete. Three corresponding states are defined for application tasks: scheduled, active, and complete. The "create" command defines a new application task, either as a successor task or subtask of an existing task, and is used to construct the application task tree that specifies the work process. When created, an application task is given an initial status of scheduled, as are the application tasks labeled "1st Draft", "Section 2", and "Section 3" in Figure 3.

As the overall job is performed, the "activate" and "complete" commands are used to traverse the application task tree and create linear change tasks to preserve various states of the system. The "activate" command changes the state of an application task from scheduled to active. As indicated in the figure, linear change tasks in the hierarchical linear versioning style are associated with the application task that was active when they were created. In this example, the author has completed the brainstorming phase of the paper and is currently working on an outline for the text. When the task represented by an application task is finished, the "com-

plete" operation is used to change the status of the task to complete and freeze its last linear change task.

Parallel Versioning Styles

The parallel change task concept offers the ability to support numerous different work flow models for monitoring and coordinating parallel activities. For example, the parallel change task could be used directly like the linear change task (cf. Figure 2) to support a basic parallel versioning style. However, it is likely that a more structured organization mechanism will be required when parallel work activities are allowed. Here we describe a particular parallel versioning style designed to support applications that require extensive work process specification and version control capabilities. Other parallel versioning styles could be developed based on the parallel change task such as one like that proposed in [8].

The task description level of the parallel versioning style introduced here (cf. Figure 4) is similar to that of the hierarchical linear style. A tree of application tasks, rooted by a single root task, is used to specify work patterns. These work patterns may be either defined by the application or the application user (cf. Discussion section). The application tasks linked by a "subtask" link to another application task represent the component tasks required to perform the task. Three states are defined for application tasks: scheduled, active, and complete. Additionally, the same three corresponding operations are defined for application tasks in the parallel style: create, activate, and complete, and they each have similar functions.

To provide support for parallel activities, successor/predecessor relationships can be created between sibling application tasks. These relationships enable the specification of temporal constraints among the component tasks of an application task. Unlike the hierarchical linear case, there is no implicit left to right ordering of tasks in the application task tree. Sibling tasks that are not related by successor/predecessor relationships, either directly or transitively, can be performed in parallel. For example, Figure 4 illustrates the task browser displaying an instance of the parallel versioning style being applied to write this paper. Note, some task labels in Figure 4, in particular those at the change control level, have been shortened to allow more of the task structure to be shown.

The application task hierarchy in this example has been structured according to the sections of the paper. There are no predecessor/successor relationships defined between the application tasks representing the "Introduction", "Related Work", "Task Concept Generalization", and "Versioning Styles" sections, so work on these sections can proceed in parallel. Two subtasks have been defined for the "Task Concept Generalization" application task. A predecessor/successor relationship has been established indicating that the "Change Control Level" section (labeled "C.C. Level" in

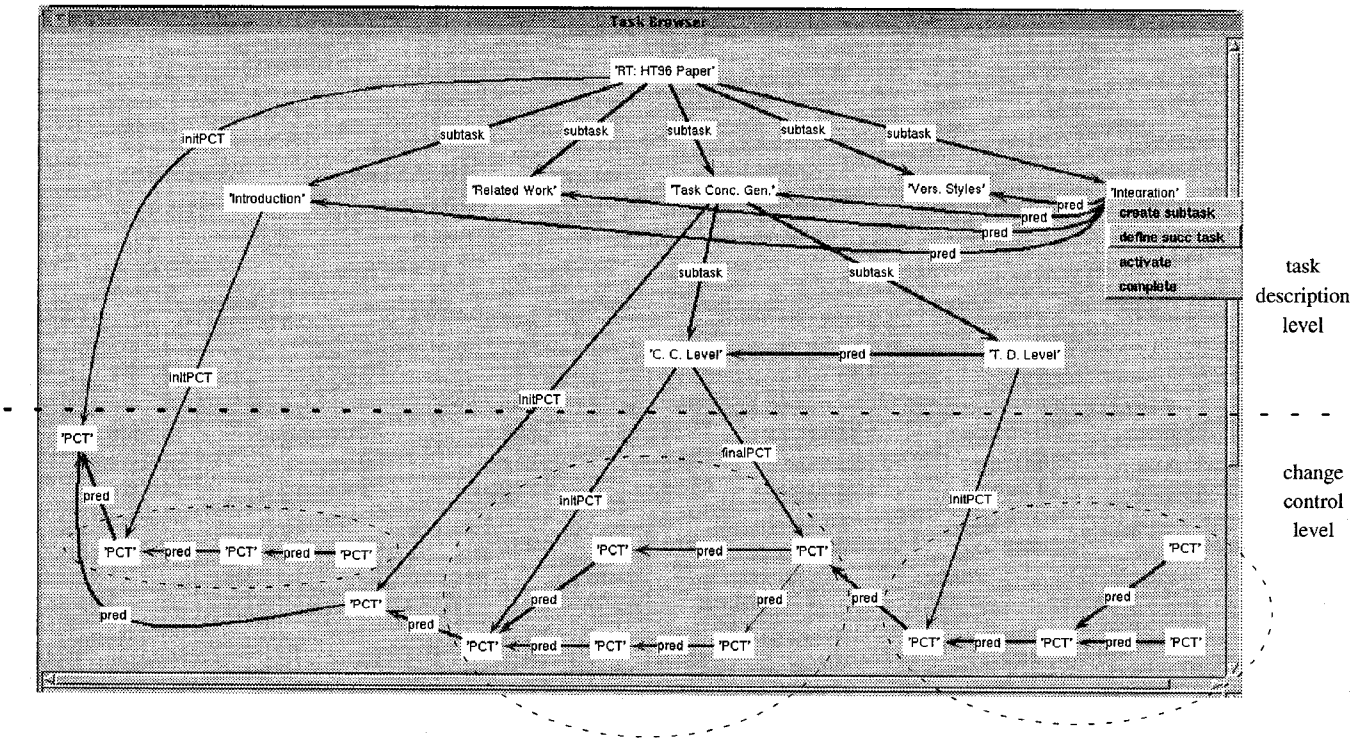

Figure 4: Task Browser displaying a parallel versioning style. The dashed line superimposed over the screen image separates the root task (RT) and application tasks from parallel change tasks (PCTs). The dashed ellipsis indicate groupings of PCTs according to their parent application task.

Figure 4) must be complete before work on the "Task Description Level" section (labeled "T.D. Level" in Figure 4) can begin. As indicated in this example, the support for parallelism in this versioning style enables a more flexible description of a work process.

Parallel change tasks at the change control level are used to preserve the state of the hypertext network at significant points during the performance of the task represented by an application task. Each application task from the task description level has an associated set of parallel change tasks at the change control level (indicated by dashed ellipsis in Figure 4). They correspond to the snapshots taken to preserve significant states of the system during the performance of the application task. The parallel change tasks contained within an application task's set of parallel change tasks are structured into a DAG, allowing parallel development activities to be supported within the context of a task, e.g. two authors working asynchronously on the same subsection of a paper. Additionally, the parallelism allowed for application tasks structures the sets of parallel change tasks themselves into a DAG. As illustrated in the example above, the DAG of DAGs supported for change tasks at the version control level of the parallel versioning style provides a very flexible support environment for parallel development activities.

Additional conditions have been defined for the parallel versioning style to support its more elaborate task specification capabilities. All application tasks from the task specification level have at least two associated parallel change tasks at the change control level. One, the initial parallel change task (related to its application task with an "initPCT" link in Figure 4), corresponds to the initial state of the system as performance of the task began. The other, the final parallel change task (related to its application task with a "finalPCT" link in Figure 4), represents the state of the system upon completion of the task. These conditions provide for two types of consistency within application tasks. The first ensures that all parallel lines of work within the scope of an application task (or recursively, a subtask) can be traced back to a single common state of the hypertext network. The second ensures that the result of each application task will be a single consistent version of the hypertext network, possibly one resulting from the merge of two or more parallel lines of development. For example, the application task labeled "C.C. Level" involved parallel development activities that started from a common (the initial) parallel change task. As the task was completed the most recent parallel change tasks along the parallel development paths were merged to produce the final parallel change task for the "C.C. Level" application task. This task served as the value for the initial parallel change task of the successor application task labeled "T.D. Level".

In the parallel versioning style, the leaf node application tasks of the application task tree represent the basic activities required to complete an overall task. They are the context in which work activities are actually performed. For example, a leaf node application task might correspond to the process of writing a section or subsection of a paper, such as the application tasks labeled "C.C. Level" and "T.D. Level" in Figure 4. Interior node application tasks, such as the application task labeled "Task Conc. Gen." in Figure 4, serve a structural purpose, allowing the more basic activities represented by leaf node application tasks to be organized into logical groups reflecting the activities required to complete the overall task. To support this arrangement, only leaf node application tasks in the application task hierarchy are allowed to have more than 2 parallel change tasks. All other (interior node) tasks have exactly 2 parallel change tasks, representing their initial and final states.

As the process of performing the job described at the task description level progresses, the "activate" and "complete" operations are used to traverse the application task hierarchy. To be eligible for activation, an application task's parent application task must be active, and all of its predecessor application tasks must be complete. For example, in Figure 4, the application task labeled "Integration" will not be eligible for activation until work on the individual sections of the paper has been completed. As an application task is activated, an initial parallel change task is created for it. The value assigned to an initial parallel change task is derived from predecessor application tasks. If it has only one predecessor (such as the application task labeled "T.D. Level" in Figure 4) it receives the value of the final parallel change task of that single application task. If the application task has multiple predecessors (such as the task labeled "Integration" in Figure 4) a merge operation is required to derive a value for the initial parallel change task. For an application task to be eligible for completion, all of its component tasks must be complete. As an application task is completed, its final parallel change task is created. If there are multiple threads of development (parallel change tasks) within the scope of an application task as it is completed, such as the application task labeled "C.C. Level" in Figure 4, a merge operation is required to produce its final parallel change task. Further details on merge support for the parallel versioning style can be found in [10].

Discussion

By design, linear versioning styles do not support elaborate object evolution patterns. It is the simplicity of these styles that makes them appropriate for an important class of applications – those with basic versioning needs, such as the ability to investigate the state of the system at a previous point in time. For example, in using DOLPHIN, a hypertext based meeting room support tool, it is often useful to examine the state of a group discussion at an earlier point in time [28].

Basic versioning styles are designed to be easily integrated into applications. For example, to incorporate the first linear versioning style, an application simply needs to be capable of indicating when a snapshot or freeze operation should be performed. In most applications this could be achieved with a user interface "freeze" button involving few actual modifications to the application. Alternatively, the freeze function could be invoked from outside the application. For example, it could be invoked from a button or menu option offered by the task browser (cf. Figure 2). The task browser could also

be used to initiate an application with a previous state of the application data space, supporting the version investigation process. In both of these cases, no modifications would be required to the application.

Similarly, the task definition, version creation, and version selection functionality of the second linear and the parallel versioning styles could be invoked from the task browser (Figures 3 and 4). Using this approach, it is up to the application user to specify the work processes manually. Alternatively, applications may use the functionality provided by a versioning style directly to establish and monitor application work processes without user intervention. For example, in the SEPIA cooperative authoring system a state of the hypertext network could be automatically frozen and a new one created when a new member joins a cooperative group [9].

In parallel versioning, states of the hypertext network must occasionally be merged. If an automatic merge procedure or interactive merge tool is available, it could be invoked automatically by specific versioning style functionality. For example, the process of completing an application task with parallel lines of development could automatically initiate a merge operation to produce a final parallel change task for the application task.

VerSE: A FLEXIBLE VERSION SUPPORT ENVIRONMENT

We have implemented the VerSE flexible version support environment offering linear and parallel change tasks and various types of application tasks. VerSE is implemented in the Smalltalk Frame Kit (SFK) [4], a frame-system based on Smalltalk. SFK has been used to implement the SEPIA [27] and DOLPHIN [28] cooperative hypertext systems and was also the basis for the CoVer version server that has been used to integrate version support into SEPIA [9].

The most recent implementation of the SFK system supports an application data space concept. For each instantiation of an application, SFK persistently maintains a separate data space. Consequently, we applied the task concepts specified in this paper to SFK application data spaces. Each change task is directly and statically linked to a specific application data space. It represents a certain state of the application data space. Initially, the application data space is empty and the initial change task does not contain any changes. By creating and updating objects/frames in a certain application data space, the respective change task gets updated. A task-based implementation strategy is used to implement version functionality for frames at the property level [8]. A detailed description of the implementation can be found in [10].

The implementation of root tasks and application tasks is straight forward. For each versioning style defined (specializations or types of root and application tasks) a special object class has been implemented in SFK. When an application data space is created, the application or user can choose one of the version support styles offered, for example one of the styles presented above, and the application data space is initialized accordingly. A general task browser provided by VerSE can be used to both examine and define the task structure. Figures 2 – 4 illustrate the implementation of different versioning styles in the VerSE environment.

In addition to the task browser, a general tool for merging versioned frames, the List-Merger, is provided by VerSE. [11] discusses how this tool can be used for merging hypertext networks.

COMPARISON TO RELATED WORK

The research presented in this paper introduces a set of versioning styles and discusses directions toward additional versioning styles. The versioning styles developed fit into general version support environments proposed for hypertext. Change tasks are incorporated at the hyperbase layer and application tasks at the intermediate layer (cf. Figure 1). In the HB3 version control framework, change tasks correspond to hyperbase level functionality and application tasks represent metadata level functionality [15].

The set of versioning styles has been influenced by the PIE [6] and CoVer [7, 8] systems. Change tasks correspond to PIE layers and application tasks to PIE contexts. Change tasks and application tasks can also be considered a generalization of the CoVer change and application task concepts, and the root task a generalization of the CoVer top level task. Whereas PIE layers can be flexibly combined into new contexts, the change tasks and application tasks of the versioning styles presented here are arranged according to their temporal history like in the CoVer system. While the intent in the PIE system was to support the flexible configuration of software systems, our motivation was to support the versioning process in the context of different work flow patterns to provide an easy to integrate mechanism for monitoring changes. Change tasks are also similar to the session concept in RHYTHM [20, 21] and the delta mechanism in the Palimpsest system [2]. However, to our knowledge neither RHYTHM or Palimpsest provide a higher level aggregation mechanism for sessions or deltas.

Our approach is especially well suited to support versioning for hypertext because it provides a mechanism to easily cope with references in a versioned hypertext network – there is only a single version present at any time. Additionally, it keeps minimal changes without copying the whole hypertext network.

CONCLUSION AND FUTURE WORK

In this paper general versioning concepts for hypertext have been introduced. Task-based versioning styles based on these concepts have been developed that can be integrated into general hypertext version support environments. The definition of a versioning style consists of a selection of change tasks for the change control level and a selection of application tasks for the task description level. The primary criteria for developing versioning styles was the cost/benefit question considered from an application developer point of view. Each of the styles introduced in detail in this paper are based on the assumption that versioning is applied to an entire hypertext network and that at most one version per versioned hypertext object is present in each version of the ap-

plication data space. These assumptions make the versioning styles very easy to use by hypertext applications. Many additional styles are possible if applications are willing to extend and enhance more application functionality to incorporate version support facilities. These versioning styles will be described in a future paper.

We have implemented the three task-based versioning styles illustrated in Figures 2 – 4 in the VerSE version support environment. Additional versioning support styles are currently under development. For testing purposes the linear versioning styles have been used to provide version support facilities for DOLPHIN, a hypertext based meeting room support system. We plan to test the applicability of the different versioning styles as they are integrated into various hypertext applications. With this research, we hope to develop a set of recommended versioning styles for hypertext.

Additionally, as mentioned earlier, the ability to merge versions of a hypertext network into a single consistent state is necessary to support parallel versioning capabilities. We are currently investigating the process of merging hypertext networks. We have developed a list-based tool for merging hypertext networks and are currently designing graph-based interfaces to provide extended support for the merge process [11]. As more is learned about the process of merging hypertext networks, we hope to eventually define (semi-) automatic hypertext merging policies.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge Lothar Rostek and Dietrich Fischer for providing the Smalltalk Frame Kit (SFK) and Andreas Kühn and Karl Skibinski for assistance in the implementation of VerSE.

REFERENCES

1. Campbell, B., and Goodman, J. 1988. HAM: A general-purpose hypertext abstract machine. CACM 31, 7 (July), 856–861.

2. Durand, D. 1993. Cooperative Editing Without Synchronization. In [17].

3. Durand, D., Haake, A., Hicks, D., and Vitali, F. (Eds.). 1995. Proceedings of the Workshop on Versioning in Hypertext Systems held in connection with ECHT'94. Available as Arbeitspapiere der GMD 894, GMD-IPSI, Darmstadt, Germany.

4. Fischer, D., and Rostek, L. 1995. SFK: A Smalltalk Frame Kit – Concepts and Use. GMD-IPSI, Darmstadt, Germany, March 1995.

5. Garg, P. and Scacchi, W. 1987. On Designing Intelligent Hypertext Systems for Information Management in Software Engineering. In Proceedings of the Fifth ACM Conference on Hypertext (Hypertext'93), (Seattle, Washington, November).

6. Goldstein, I., and Bobrow, D. 1984. A layered approach to software design. In Interactive Programming Environment, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. McGraw-Hill, NY, pp. 387– 413.

7. Haake, A. 1992. CoVer: A contextual version server for hypertext applications. In Proceedings of the European Conference on Hypertext (ECHT'92) (Milan, Italy, Nov.), pp. 43–52.

8. Haake, A. 1994. Under CoVer: The implementation of a contextual version server for hypertext applications. In Proceedings of the European Conference on Hypertext (ECHT'94) (Edinburgh, UK, Sept.), pp. 81–93.

9. Haake, A., and Haake, J. 1993. Take CoVer: Exploiting Version Support in Cooperative Systems. In Proceedings of INTERCHI '93 – Human Factors in Computing Systems (Amsterdam, The Netherlands, April), pp. 406–413.

10. Haake, A. and Hicks, D.L. 1995. VerSE: a flexible version support environment. Arbeitspapier der GMD, 1995. GMD-IPSI, Darmstadt, Germany.

11. Haake, A., Haake, J., and Hicks, D.L. 1995. On Merging Hypertext Networks. Position paper for the Workshop on Version Control in CSCW held in connection with ECSCW'95.

12. Halasz, F. 1988. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. CACM 31, 7 (July), 836–852.

13. Halasz, F. 1991. Hypertext '91 Keynote Address. Third ACM Conference on Hypertext (Hypertext'91), (San Antonio, Texas, December).

14. Hicks, D. L., Leggett, J.J., Nürnberg, P.J., Schnase, J.L. 1995. A Hypermedia Version Control Framework. Submitted to ACM Transactions on Information Systems.

15. Hicks, D. L. 1993. A version control architecture for advanced hypermedia environments. Dissertation. Department of Computer Science, Texas A&M University, College Station, TX.

16. Kydd, S., Dyke, A., and Jenkins, D. 1995. Hypermedia Version Support for the Online Design Journal. In [3].

17. Leggett, J. J. 1993. Report of the Workshop on Hyperbase Systems held in conjunction with Hypertext'93. Available as Department of Computer Science Technical Report No. TAMU–HRL 93–009, Texas A&M University, College Station, TX.

18. Leggett, J. J., Schnase, J. L., Smith, J. B., and Fox, E. A. 1993. Final report of the NSF workshop on hyperbase systems. Available as Department of Computer Science Technical Report No. TAMU–HRL 93–002, Texas A&M University, College Station, TX.

19. Leung, R. 1995. Versioning on Legal Applications Using Hypertext. In [3].

20. Maioli, C., Sola, S., and Vitali, F. 1993. External anchors as a means of avoiding bottlenecks in the wide-area distribution of hypertext data. In [17].

21. Maioli, C., Sola, S., and Vitali, F. 1993. Wide-area distribution issues in hypertext systems. In Proceedings of SIGDOC'93, (Ontario, Canada, October)

22. Möller, H. 1995. Versioning Structured Technical Documentation. In [3].

23. Munson J., and Dewan, P. 1994. A Flexible Object Merging Framework. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'94), (Chapel Hill, North Carolina, Oct.), pp. 231–241.

24. Østerbye, K. 1992. Structural and cognitive problems in providing version control for hypertext. In Proceedings of the European Conference on Hypertext (ECHT'92) (Milan, Italy, Nov.), pp. 33–42.

25. Prevelakis, V. 1990. Versioning issues for hypertext systems. In Object Management, D. Tsichritzis, Ed. Centre Universitaire d'Informatique, Université de Genève, Switzerland, pp. 89–105.

26. Soares, L., Rodriguez, N., and Casanova, M. 1995. Nested Composite Nodes and Version Control in Hypermedia Systems. In [3].

27. Streitz, N., Haake, J., Hannemann, J., Lemke, A., Schuler, W., Schütt, H, and Thüring, M. 1992. SEPIA: A Cooperative Hypermedia Authoring Environment. In Proceedings of the European Conference on Hypertext (ECHT'92) (Milan, Italy, Nov.), pp. 11–22.

28. Streitz, N., Geissler, J., Haake, J., and Hol, J. 1994. DOLPHIN: Integrated meeting support across Liveboards, local and remote desktop environments. In Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'94), (Chapel Hill, North Carolina, Oct.), pp. 345–358.

29. Whitehead, E., Anderson, K., and Taylor, R. 1995. A Proposal for Versioning Support for the Chimera System. In [3].

30. Wiil, U., and Leggett, J. 1992. Hyperform: Using Extensibility to Develop Dynamic, Open, and Distributed Hypertext Systems. In Proceedings of the European Conference on Hypertext (ECHT'92) (Milan, Italy, Nov.), pp. 251–261.