Supporting Time-variant Artifacts in Groupware Applications

Eberhard Grummt[1,2] and Alexander Lorz[1]

[1] Dresden University of Technology
[2] SAP Research, CEC Dresden

Abstract. Asynchronous groupware strives to provide a "shared memory" for distributed workers. However, current systems fail to keep track of changes in work and organizational structures, leading to old information being discarded instead of being archived. Especially in so called "Virtual Organizations", where such changes happen often, being able to "go back in time" is desirable. We present a generic relational data model including operations capable of storing and querying time-variant data. The applicability of this model is discussed based on experiences with a prototypical application enabling visualization and interaction with respective information.

1 Introduction

Groupware systems supporting collaboration, coordination, and communication have established themselves as valuable tools for people working on joint projects. Especially in Virtual Organizations (VOs) spanning several physically distributed teams such support is crucial. The ultimate goal of asynchronous groupware is to provide a "shared memory" for all co-workers so every participant stays aware of appointments, tasks, documents etc. However, a widely accepted quality of VOs is their frequent reconfiguration by dynamically mapping satisfiers to requirements [7], which means that all these things as well as the teams themselves are changing constantly. Reflecting these changes is a requirement that is not sufficiently addressed by most open source groupware products. Their majority can store only one specific view of the modeled real world. This view is a "snapshot" considered to be "current", whereas former views are discarded by overwriting or deleting the respective data. For example, after a task has been removed, it is not possible to tell if it has ever existed, let alone who was responsible for it. To remedy these shortcomings, concepts in at least two areas have to be developed. Firstly, sound and easy to use generic data models describing objects and their relations with respect to temporal changes are essential. Secondly, suitable metaphors and user interfaces are required to seamlessly integrate time-variance into human-computer interaction.

This paper illustrates the need for supporting time-variance in asynchronous groupware tools as well as arising technical issues (Section 3). Based on requirements described in Section 4, we present a temporal data model (Section 5). In Section 6, we describe our experimental prototype, including our approach to introducing time-variance at the user interface level. We close with a discussion of the lessons learned and an outlook towards future work.

Grummt E. and Lorz A. (2007).
Supporting Time-variant Artifacts in Groupware Applications.
In *Proceedings of the 4th International Workshop on Computer Supported Activity Coordination*, pages 35-44
DOI: 10.5220/0002417400350044

2 Related Work

Frank [2] gives an overview of concepts for modeling temporal phenomena. Jensen and Dyreson [4] provide a glossary of terms related to time-variance and temporal databases. Knolmayer and Myrach [6] discuss how temporal data can be represented in business software. Most approaches to modeling time-variant data are extensions of the Entity-Relationship-Model. Gregersen and Jensen [3] survey more than ten of these, including *ERT*, *TEER*, and *TERM* [5]. A discussion of the meaning of "now" in databases is conducted by Clifford et al. [1]. Even though current commercial databases do not feature comprehensive support for temporal structures, a lot of theoretical background has already been explored. Snodgrass et al. [9] introduce the language *TSQL2*, an extension of the SQL standard supporting complex time-related queries. Wang and Zaniolo [10] propose an XML-based language for describing and querying temporal data.
However, existing approaches to incorporating time-variance in actual database systems suffer performance issues and high complexity, leading to a lack of reference implementations.

The kind of groupware we focus on is characterized by providing asynchronous access to shared tools and information such as a calendars and tasks. Other types of groupware include Wikis and Version Control Systems. These do feature a notion of "time" by preserving past states of the system. However, they are rather focused on processing *information rich* data (such as arbitrarily structured text) than on highly structured (*semantically rich*) data that contains relations and has specific demands regarding (temporal) integrity. An approach to enhancing the desktop metaphor by a notion of time is presented by Rekimoto [8]. The time-variant desktop called *TimeScape* can be set to any desired time, where objects can be created, pasted, modified, or removed. Objects are not deleted but archived, graphically this is expressed by letting icons of "removed" objects fade away as time passes by. The backup tool "Time Machine" of the announced *Mac OS X Leopard* reflects some of these ideas.

3 The Need for Time-variant Groupware in Virtual Organizations

None of the web based groupware systems we evaluated is capable of sufficiently documenting the changes over time regarding team membership, task assignment and other highly structured data. Yet often, it is desirable to "go back in time" to see what data was considered "current" at a chosen point in time. When dealing with potentially large objects such as media files, the costs of not deleting data that users discard need to be considered. In groupware systems processing mostly short and highly structured textual or numeric data this is not a limiting aspect. However, security and privacy concerns arise when potentially confidential information cannot be deleted permanently.

When talking about "time" with regard to groupware or database systems, it is important to distinguish between several notions (see [4]). *Valid Time* specifies when certain propositions in the modeled real world are considered true. In IT systems, these are usually specified by the user. In contrast, *Transaction Time* represents the time when an element was available inside the IT system. *User-defined Time* refers to temporal

attributes not interpreted by the system. A *Temporal Database* supports at least one of the first two conceptions of time. Databases supporting Valid Times are called *Historic Databases*, whereas databases supporting Transaction Times are called *Rollback Databases*. *Bitemporal Databases* support both aspects. These distinctions can be applied to groupware systems, too. We argue that in such systems, Valid Time is more important than Transaction Time, because we are more interested in facts about the real world than in functional aspects of the technical system. However, it needs to be considered that forcing users to enter Valid Times puts additional effort on the involved interactions.

Regarding group awareness, most systems only support the current time and place of co-workers, but not the respective history. Yet, information about "who worked on what, when and where" is potentially useful.

4 Requirements Analysis

The main requirement of the conceived time model is the ability to seamlessly document changes of the modeled real-world artifacts. Thus, we focus on Valid Time. We need to consider that certain points in time in the past or in the future may be unknown to the user. The reasons can be different: either the user does not have specific plans for the future yet, or he does not have access to all necessary facts from the past.

Since members of a VO can be distributed over different time zones, these need to be supported either explicitly or by using normalized time values. The time and data models need to support persistent storage mechanisms in a way that saving, loading, deleting and searching of data sets can be conducted with sufficient performance. The precision of the time information is to be derived from practical requirements of VOs.

The data model has to be as versatile as possible, to meet not only the requirements of a groupware system for VOs but also other potential applications. It is meant to serve as a solid, low-level basis for semantically rich (highly structured) models. Hierarchies and non-hierarchical relations are to be supported. The data model needs to support the basic operations create, read, update and delete with respect to well-defined temporal consistency constraints. To ensure ease of use, manageability, and reasonable implementation effort, the model needs to be easy to comprehend and should rely on as few operations as possible.

5 A Relational Data Model for Time-variant Structures

In this section, we describe our time and data models including temporal consistency constraints and a set of operations.

5.1 Time Model

Our time model is based on intervals describing the temporal validity of specific information about the real world (*validity intervals*). Intervals are defined by their start and end points (t_{start} und t_{end}) on a discrete time axis with a resolution of one second.

We consider this resolution sufficient for typical applications in VOs, however, the approaches presented below can equally well be applied to discrete time models of higher resolutions. The intervals are closed, i.e. their t_{start} and t_{end} values are part of them.

The format of time values is taken from *XML Schema* and is structured like this: `YYYY-MM-DDThh:mm:ss`, with `T` being a separating character. All time values are relative to a reference time zone (*UTC*), so the internal operations for comparison and checking of consistency constraints can be implemented with less effort. This only affects the internal storage of data, not necessarily the user interface built on top of this data.

Besides concrete time values, two special values are part of the model. *Until changed* (`uc`) represents the fact that information is valid until something different is specified. Consequently, `uc` can only be used as a value for t_{end}. Its counterpart is *not known* (`nk`), describing that the start time of a validity interval is unknown. Thus, `nk` can only be assigned to t_{start}.

5.2 Data model

`Objects` are considered unique entities composed of time-variant attributes. They can be put in time-variant relationships with other `Objects`. `Objects` are composed of `Versions`, each of which represents attributes and relations that are constant over a particular time interval. That means our data model is *temporally ungrouped*, however a *temporally grouped* view, e.g. using XML, can be created easily. Every `Object` contains at least one `Version` and itself has a validity interval, which can be derived from its `Versions`. Using a particular `Version`, holistic statements about the `Object` it belongs to can be made.

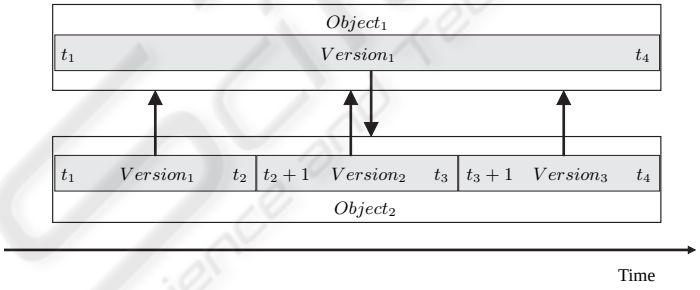

Fig. 1. Two related `Objects` and their `Versions`.

Relations are symmetric links between two `Objects`. They are treated as attributes of `Versions` and thus are not modeled as separate entities. Relations themselves have no attributes and inherit their validity intervals from the respective `Versions`. Because `Objects` are actually a kind of a container for `Versions`, relations point to `Objects` rather than to specific `Versions` of the linked `Object` (see Fig. 1). The required symmetry of the relation needs to be ensured by proper setup of the involved `Versions`.

5.3 Consistency Constraints

For `Objects` and their `Versions`, we introduce the following consistency constraints regarding their validity intervals:

- **C1: No changes during the validity interval.** The attributes and relations stored in a `Version` are constant over its whole validity interval.
- **C2: No redundancy through similar `Versions`.** Adjacent `Versions` differ in at least one attribute or relation.
- **C3: No gaps.** There are no gaps between the validity intervals of an `Object`'s `Versions`.
- **C4: No overlapping.** For any given point in time, each `Object` has exactly one or no `Version`.
- **C5: Temporal consistency of relations.** All `Objects` that are put in relation are valid throughout the whole validity interval of the relation.

From C3 follows that, for a given `Object`, the union of all the `Version`'s validity intervals is equal to the `Object`'s validity interval. It also follows that the validity interval of an `Object` can be determined by combining the oldest `Version`'s t_{start} and the newest `Version`'s t_{end} into an interval. From C4 follows that, for a given `Object`, the intersection of all the `Version`'s validity intervals is empty. This also implies that for only one `Version` per `Object` t_{start}=`nk` or t_{end}=`uc` may be true (one single `Version` with t_{start}=`nk` and t_{end}=`uc` is in particular valid). From C5 and C1 follows that at the beginning and the end time of a relation, all involved `Objects` need to have "version borders". While gaps in an `Object`'s validity interval are not permitted (C3), relations can have temporal breaks. For example, a user can be part of a group, then leave it and later rejoin it, leading to a logical "is member of"-relation with a gap.

5.4 Definition of Operations

In this section, we present four basic and two derived operations on our data model. They allow integration of new information with already known facts, automatically enforcing our constraints by adjusting validity intervals and creating, merging, or deleting `Versions` as needed. This simplifies usage of functionality for both the application programmer and the end user.

(1) Add Version. This operation adds a `Version` to an `Object`. It can be distinguished between *adding with overwriting* and *adding without overwriting*. The latter is equivalent to checking the consistency constraints and adding only if no constraints are violated (exception: overlapping with a right open `Version`, see Fig. 2A). The general case, that is, "adding with overwriting", only requires the validity interval of the input information (`Version`) to touch or overlap the validity interval of the `Object` that the new `Version` is to be assigned to (C3). The other consistency constraints can be satisfied automatically by overwriting, splitting (C1) or merging (C2) existing `Versions`.

(2) Remove Version. When removing a `Version`, two general cases can be distinguished. Firstly, the `Version` to be removed can be the temporally first or last `Version` of an `Object`. In this case, it can simply be removed from the `Object`. However, further steps might be necessary to meet C5. Secondly, a `Version` can be an "inner `Version`", having a temporal predecessor and successor. In this case, simply removing it would leave a gap and thus violate C3. This is compensated by extending the adjacent `Versions` by half the length of the removed `Version`. Another possibility would be to extend only one of the adjacent `Versions`, letting the user decide which one to use. However, we opted for the "interpolation approach" to keep required user interactions at a minimum.

(3) Change Version. Solely changing a `Version`'s attributes corresponds to adding a new `Version` with the desired attributes at exactly the same place, thus overwriting the old `Version`. Practically, the attributes can be changed directly.
Changing the validity interval of a specific `Version` is more complicated because all other `Versions` of the respective `Object` could be affected. The length of a `Version`'s validity interval can either remain unchanged, be compressed or stretched. The case that the length remains unchanged and t_{start} and t_{end} are shifted by the same amount into one direction can be described by using compression on one side and stretching on the other.

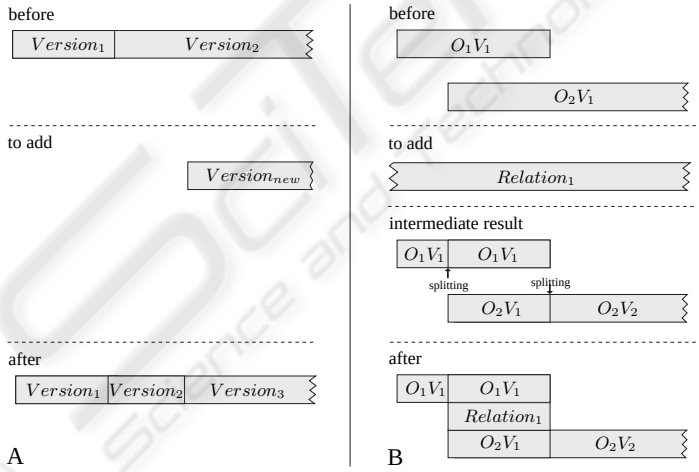

Fig. 2. Adding a `Version` (A) and Adding a relation (B).

If a `Version` is to be compressed, it is shortened and inserted into the `Object`. Then, copies of the formerly adjacent `Versions` are stretched to touch the newly inserted `Version` and inserted, thus overwriting the remainders of the formerly adjacent `Versions`. Stretching is done by making a copy of the respective `Version`, adjust-

ing the validity interval and adding it. This can lead to several (not only the adjacent) `Versions` to be partially or completely overwritten.

(4) Add Relation. Handling relations needs special care because C5 is not only affected by directly manipulating relations, but also by manipulating the `Objects` that are linked by a specific relation. The simplest case is that a relation is added between two `Objects` consisting of only one `Version` each and sharing a common validity interval. Here, the relation can be established without any further steps. When a relation is to be established between two `Objects` that only partially overlap in time, the maximum possible interval for the relation is determined by intersecting the validity intervals of the two `Objects` and the desired relation. Then, the `Versions` at t_{start} and t_{end} of this interval are split to ensure C1 after the relation is added. Finally, the relation itself can be applied (see Fig. 2B). Splitting `Versions` at t_{start} and t_{end} of the relation's validity interval can result in up to two new `Versions`.

(5) Remove Relation. Relations can be removed completely by deleting the respective references in all affected `Versions`. Afterwards, some `Versions` might need to be merged to ensure compliance with C2. If only a part of a relation is to be removed, then this part is subtracted from the original relation. Again, splitting or merging at the new t_{start} and t_{end} values of the resulting relations' intervals may be necessary.

(6) Change Relation. Changing a relation can be expressed as adding (overwriting) or removing parts of a relation.

Effects on relations of changes to Objects. When `Versions` are modified, the relations between the respective `Objects` might need to be updated. Deleting, compressing or stretching of `Versions` might become necessary, but also splitting or merging. Just like in relational databases, the referential integrity needs to be assured when a `Version` referencing an `Object` is deleted. Overwriting `Versions` containing relations is a separate problem. The easy way would be to remove the relations in the new `Version`'s validity interval. However, the information about the `Object`'s relations does not need to be sacrificed. Because adding a `Version` can not shorten an `Object`'s validity interval, C5 can not be violated. However, it needs to be ensured that after adding the `Version`, C1 is still valid. Therefore, the new `Version` needs to be split at all borders of the overwritten `Versions` where changes regarding relations occur.

6 Experimental Prototype

Using the Java programming language and JSP scripts, we implemented a groupware prototype demonstrating the developed concepts. Temporal data can be inserted, modified and deleted using a web-based GUI. Data is persisted by means of an XML-based language we specified using XML Schema, proving our model's applicability in XML

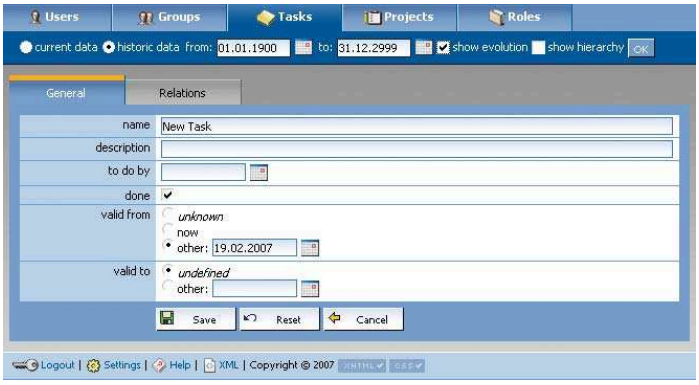

Fig. 3. Our GUI showing a form for entering time-variant information.

databases. As an example scenario, we used a typical data model for groupwork, consisting of hierarchical groups, projects, and tasks along with users that can be associated directly or by role. Each of these entities is derived from a special data type providing functionality for the operations discussed in Section 5.4. Working with these versioned objects and treating relations as attributes of versions proved to be convenient once the implementation taking care of our consistency constraints was available.

The user interfaces employs a set of widgets called “time control bar” enabling the selection of arbitrary time intervals. This selection acts as a filter for the data that can be interacted with in a main area. Information from the selected interval can be presented as lists, detail views, forms, and interactive timelines. Time variance is reflected in all of these elements:

Lists can be rendered either to display the current information or to show data from an arbitrary, user-defined time interval. In the latter case, the newest `Version` overlapping the chosen interval is used to represent the respective `Object`. Using the checkbox “show evolution”, earlier `Versions` of each `Object` can be displayed in separate, indented rows. Using colors and icons, the temporal role of each `Version` is further illustrated.

Detail views display all the `Versions` of one `Object`, highlighting attributes and relations that have changed from one `Version` to its successor. Bars are used to illustrate the validity intervals of each `Version`.

Interactive timelines display multiple `Object`’s and their `Version`’s validity intervals and, on mouse-over, their attributes and relations.

Forms provide common widgets to enable user input. In addition, each form lets users specify the information’s validity interval. By default, this interval is set to (*now*, `uc`). Relations can be specified using a separate tab which provides “select multiple” lists with `Objects` that can potentially be put in relation with the input data. Users found it intuitive to enter data using the enhanced forms. Detail views were also easily grasped. However, lists were sometimes found confusing as it was not clear which `Version` was actually used for display.

7 Lessons Learned and Future work

Our concepts of relations and `Objects` consisting of `Versions` have proved to be technically well suited for storage and retrieval of time-variant data. The advantage of using `Versions` is that *holistic* statements (as opposed to statements about single attributes) about some entity over certain intervals can be made.

Adding support for time-variant artifacts doesn't come for free, but the experience with applying our model to the prototype scenario has shown that it can be done with reasonable effort. The model provides lightweight metaphors and a small but sufficient set of predefined operations promising a steep learning curve. Application developers often implement relations as attributes of the objects involved. Expressing temporal changes of relations between objects as changes in their respective attributes instead of using additional time-variant entities supports this point of view. The ability to express temporal changes can be incorporated into an existing application model without changing its basic structure. Business logic can be implemented with a few rules in mind that map desired changes in the data model to the six operations of our temporal model. The price for simplicity is increased redundancy by saving unchanged attributes multiple times. This should be coped with by a more efficient versioning layer for objects that stores version differences instead of full object copies. Another solution would be a temporally grouped representation (that is, every attribute holds its own history). Such a representation can be serialized using XML, rather complicated table structures in common RDBMS, or nested tables as supported by *SQL:2003*.

From our point of view, finding sufficient user interface metaphors is crucial for implementing "temporal awareness" into groupware. Users are accustomed to software that only has concepts for "now". If a person is removed from a group, the interaction either happens because the person just left the group or the person was wrongly assigned to it. In conventional groupware there is no need to care about the difference between these operations because the result is the same – the person is currently not in the group. If we start to support time variance we have to make the user aware of a distinction he is not used to express explicitly. This could prove to be much harder than implementing an efficient data model.

Virtual Organizations can employ software similar to our prototype to keep track of all their internal changes. However in real world settings, one would need to consider concurrency and transactional issues. Enhanced access control would be necessary to prevent unauthorized users from overwriting information that refers to the past.

We also strive for more intuitive ways to input information, not forcing users to think in terms of versions. Our goal is to empower users to intuitively enter facts like "person X changed her e-mail address to w@xyz.com in 2006". We want them to be able to specify single facts (as opposed to filling in a complete form) and also want to support additional unspecific, "fuzzy" time values. Next, we want to examine possibilities how this can be supported at the user interface level. In addition, we want to explore how time-variant information can be visualized and manipulated graphically. We already developed concepts for interactive "time bars", that were however not included in our prototype due to technical limitations. These concepts need to be refined, evaluated and extended to support large amounts of data and longer time intervals, for example using perspective distortion and *Overview and Detail* techniques.

References

1. James Clifford, Curtis Dyreson, Tomas Isakowitz, Christian S. Jensen, and Richard Thomas Snodgrass. On the semantics of “now” in databases. *ACM Trans. Database Syst.*, 22(2):171–214, 1997.
2. Andrew Frank. *Spatial and Temporal Reasoning in Geographic Information Systems*, chapter Different types of “times” in GIS. Oxford University Press, 1995.
3. Heidi Gregersen and Christian S. Jensen. Temporal entity-relationship models - a survey. *Knowledge and Data Engineering*, 11(3):464–497, 1999.
4. C. S. Jensen and C. Dyreson. The consensus glossary of temporal database concepts february 1998 version. 1998.
5. M. R. Klopprogge. Term: An approach to include the time dimension in the entity-relationship model. In *Proceedings of the Second International Conference on the Entity Relationship Approach*, pages 477–512, Washington, DC, 1981.
6. G. Knolmayer and T. Myrach. Zur abbildung zeitbezogener daten in betrieblichen informationssystemen. *Wirtschaftsinformatik*, 38:63–74, 1996.
7. Abbe Mowshowitz. Virtual organization. *Communications of the ACM*, 40:30–37, 1997.
8. Jun Rekimoto. Timescape: a time machine for the desktop environment. In *CHI ’99: CHI ’99 extended abstracts on Human factors in computing systems*, pages 180–181, New York, NY, USA, 1999. ACM Press.
9. Richard T. Snodgrass, Ilsoo Ahn, Gad Ariav, Don S. Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Kafer, Nick Kline, Krishna G. Kulkarni, T. Y. Cliff Leung, Nikos A. Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo, and Suryanarayana M. Sripada. Tsql2 language specification. *SIGMOD Record*, 23(1):65–86, 1994.
10. Fusheng Wang and Carlo Zaniolo. An xml-based approach to publishing and querying the history of databases. *World Wide Web*, 8(3):233–259, 2005.