Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Viale Teresa Michel 11, 15121 Alessandria
http://www.di.unipmn.it

A temporal relational data model for proposals and evaluations of updates

Authors: Luca Anselma, Alessio Bottrighi, Stefania Montani, Paolo Terenziani (anselma@di.unito.it, {alessio.bottrighi,stefania,terenz}@mfn.unipmn.it)

TECHNICAL REPORT TR-INF-2009-09-07-UNIPMN
(September 2009)

The University of Piemonte Orientale Department of Computer Science Research Technical Reports are available via WWW at URL `http://www.di.unipmn.it/`. Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2009-06 *Performance analysis of partially symmetric SWNs: efficiency characterization through some case studies*, Baarir, S., Beccuti, M., Dutheillet, C., Franceschinis, G., Haddad, S., July 2009.

2009-05 *SAN models of communication scenarios inside the Electrical Power System*, Codetta-Raiteri, D., Nai, R., July 2009.

2009-04 *On-line Product Conguration using Fuzzy Retrieval and J2EE Technology*, Portinale, L., Galandrino, M., May 2009.

2009-03 *GSPN Semantics for Continuous Time Bayesian Networks with Immediate Nodes*, Portinale, L., Codetta-Raiteri, D., March 2009.

2009-02 *The TAAROA Project Specification*, Anglano, C., Canonico, M., Guazzone, M., Zola, M., February 2009.

2009-01 *Knowledge-Free Scheduling Algorithms for Multiple Bag-of-Task Applications on Desktop Grids*, Anglano, C., Canonico, M., February 2009.

2008-09 *Case-based management of exceptions to business processes: an approach exploiting prototypes*, Montani, S., December 2008.

2008-08 *The ShareGrid Portal: an easy way to submit jobs on computational Grids*, Anglano, C., Canonico, M., Guazzone, M., October 2008.

2008-07 *BuzzChecker: Exploiting the Web to Better Understand Society*, Furini, M., Montangero, S., July 2008.

2008-06 *Low-Memory Adaptive Prefix Coding*, Gagie, T., Nekrich, Y., July 2008.

2008-05 *Non deterministic Repairable Fault Trees for computing optimal repair strategy*, Beccuti, M., Codetta-Raiteri, D., Franceschinis, G., July 2008.

2008-04 *Reliability and QoS Analysis of the Italian GARR network*, Bobbio, A., Terruggia, R., June 2008.

2008-03 *Mean Field Methods in performance analysis*, Gribaudo, M., Telek, M., Bobbio, A., March 2008.

2008-02 *Move-to-Front, Distance Coding, and Inversion Frequencies Revisited*, Gagie, T., Manzini, G., March 2008.

2008-01 *Space-Conscious Data Indexing and Compression in a Streaming Model*, Ferragina, P., Gagie, T., Manzini, G., February 2008.

2007-05 *Scheduling Algorithms for Multiple Bag-of-Task Applications on Desktop Grids: a Knowledge-Free Approach*, Canonico, M., Anglano, C., December 2007.

2007-04 *Verifying the Conformance of Agents with Multiparty Protocols*, Giordano, L., Martelli, A., November 2007.

2007-03 *A fuzzy approach to similarity in Case-Based Reasoning suitable to SQL implementation*, Portinale, L., Montani, S., October 2007.

2007-02 *Space-conscious compression*, Gagie, T., Manzini, G., June 2007.

2007-01 *Markov Decision Petri Net and Markov Decision Well-formed Net Formalisms*, Beccuti, M., Franceschinis, G., Haddad, S., February 2007.

Abstract

The cooperative construction of data\knowledge bases has recently had a significant impulse (see Wikipedia [Wikipedia]). In cases in which data\knowledge quality and reliability are crucial, proposals of update\insertion\deletion need to be evaluated by experts. So far, no *theoretical framework* has been devised to model the *semantics* of update proposal\evaluation in the *relational context*. Since update proposal\evaluation deeply involves the notion of *time*, semantic approaches to *temporal* relational databases (specifically, BCDM [Jensen and Snodgrass 1996]) are the starting point of a theoretical framework definition. In this paper, we propose $BCDM^{PV}$, a semantic temporal relational model that extends BCDM to deal with multiple update\insertion\deletion proposals and with acceptances\rejections of proposals themselves. We define the related *data structures*, *manipulation operations* and temporal relational *algebra* and show that it is a *consistent extension* of and that it is *reducible* to BCDM. These properties ensure consistency with most relational temporal database frameworks, facilitating future implementations.

KEYWORDS: update proposal and evaluation, temporal relational databases, semantics of relational data and operations, data model, algebraic operators, Bitemporal Conceptual Data Model.

1 Introduction

In this paper we extend the temporal database data model and algebra in order to cope with proposals and evaluation of updates, as required in several emerging applications concerning cooperative modeling\update of shared data\knowledge.

1.1 Context: the "proposal vetting" phenomena

As widely discussed in Section 2, cooperative modeling\update of shared data\knowledge is nowadays a main paradigm in many areas, ranging from the collaborative definition of encyclopedias (e.g., Wikipedia) to the identification of specific goal-oriented workflows, protocols and guidelines. In some of such areas, the need for quality demands for some form of evaluation process: users propose updates to the reference version of data\knowledge, and a team of domain experts evaluates them, so that only approved proposals modify the reference data\knowledge, leading to a new *data\knowledge reference version* (for short, in the following we term such a process "*proposal vetting*"). The history of versions is usually maintained (and tagged with their *transaction time*, i.e., with the time when data are inserted\(logically)deleted [Snodgrass and Ahn 1986]), and, in many cases, data\knowledge are temporal, so that also their *valid time* (i.e., the time when data hold in the modeled mini-world [Snodgrass and Ahn 1986]) must be explicitly dealt with.

The relational model has a formal basis in set theory and logic, and relational databases are widely used also because the relational model is well understood in theory and practice. Despite the generality and the spread of the phenomena, in real world applications (e.g., Wikipedia [Wikipedia] and Citizendium [Citizendium]) update proposals and evaluations are coped with in an *ad-hoc* fashion (primarily at the application level) using relational databases. Clearly, this defeats the very purpose of the relational model: a high level of independence between the data and the application programs. Thus the need for a more comprehensive data model arises, where update proposals and evaluations are an intrinsic part of the model. We are not aware of any theoretical framework coping with the above phenomena in a purely relational database context. In this paper, we provide a *semantic* analysis of the proposal vetting phenomena extending the relational model

semantics. Our aim is to provide a clean and general (as opposed to ad-hoc) modeling of the phenomena, abstracting away from efficiency issues and implementation strategies, leading to both a deep understanding of the phenomena and a rigorous reference specification for future implementations.

Since transaction time (and, in several cases, valid time) need to be supported, temporal relational databases are the natural starting point of our work.

1.2 Context: temporal database models

Given the relevance and diffusion of time-related issues in real-world phenomena, there has been much work over the last three decades in incorporating time into data models, query languages, and database management system (DBMS) implementations. In particular, all such approaches demonstrate that time is a peculiar aspect that deeply affects data semantics, so that it cannot be coped with just through the addition of one (or more) additional attribute in relational tables, but it deserves a specific treatment (see, e.g., Chapter 1 of the TSQL2 book [Snodgrass 1995]). Given the pervasive character of time, great efforts in terms of research were made in order to provide once-and-for-all a general solution to the problem (as opposed to ad-hoc solutions to be independently built in each application coping with time). In this spirit, many extensions to the standard *relational* model were devised, and more than 2000 papers on temporal databases (TDBs) were published over the last two decades (cf., the cumulative bibliography in [Wu et al. 1998], the section about TDBs in the forthcoming Springer Encyclopedia of Databases [Liu and Ozsu 2008], which includes over 30 entries about TDBs, the entry "Temporal Database" in [Liu and Ozsu 2008], and the surveys in [McKenzie & Snodgrass 1991, Tansel et al. 1994, Özsoyoğlu & Snodgrass 1995, Jensen & Snodgrass 1999].

A core issue concerning most of such approaches is their compatibility and reducibility to the standard (non-temporal) relational model, in order to grant (i) that if time is disregarded, the extended temporal model behaves like the standard one and (ii) interoperability with pre-existent non-temporal data. In such a sense, most temporal (relational) database approaches can be seen as a consistent layer built upon standard (relational) databases, providing once-and-for-all users with the facilities (e.g., data structures, query languages) to cope with temporal data in an easy and correct way.

Despite such a common goal, the large variety of different approaches to TDBs in the literature is partly due to the fact that many diverse issues, including presentation and efficient implementation, have been taken into account. In order to identify the "essence" of modeling time in relational databases, and the common "core" between many of such approaches, the BCDM (Bitemporal Conceptual Data Model) *relational data model* and *algebra* have been identified [Jensen and Snodgrass 1996, Snodgrass 1995]. BCDM does not face issues such as data representation and storage optimization, aiming at a "*semantic*" approach, in the sense discussed in the below citation, quoted from [Snodgrass 1995]: "*It is our contention that focusing on* data presentation *(how temporal data is displayed to the user), on* data storage *with its requisite demands of regular structure, and on efficient* query evaluation*, has complicated the central task of capturing the time-varying semantics of data*. […] *We therefore advocate a separation of concerns. Time-varying semantics is obscured in the representational schemes by other considerations of presentation and implementation. We feel that the conceptual data model to be discussed shortly* [i.e., BCDM] *is the most appropriate basis for expressing this semantics*."

As BCDM, also our approach operates at the semantic level, in the sense discussed above (not to be confused with the "conceptual" -e.g., Entity-Relationship- level).

Although BCDM cleanly copes with both *valid* and\or *transaction* time (i.e., it supports bitemporal data), both its data model and operations are not expressive enough to support update proposals and evaluations (see the discussion in Subsection 3.1).

1.3 Goals, methodology and main results

Given the wide diffusion and increasing relevance of update proposal and evaluation phenomena (e.g., related to Citizendium-like activities: see the discussion in Section 2), as well as the deep impact it has on *data semantics*, we strongly believe that, once again, a *general* (non ad-hoc) solution is needed here.

For the sake of generality and clarity, we have chosen to operate at the semantic level, proposing $BCDM^{PV}$ (where "PV" stands for "proposal vetting"), an extension of BCDM unifying semantic model.

However, we still retain consistency with relational model and implementability as a fundamental goal for our approach. Just in the same way in which temporal database models can be regarded as a consistent upper layer upon standard (non-temporal) models (as granted by the consistency and reducibility properties) to cope with a new range of phenomena (namely, valid and transaction time), the goal of our work is to propose a new model which is a consistent *upper layer* upon temporal relational model to cope with proposals and evaluations of updates.

The main original contributions of our approach lie in extending BCDM to support proposals and evaluation of updates, and, specifically, we propose:

1. a new ***data model*** to cope with both "reference" (accepted) and proposed (to be evaluated) data; in particular, we support *alternative* data proposals (while in BCDM relations are defined as *conjunctive* sets of tuples only);
2. new ***manipulation operations*** to propose insertions, deletions and updates (for proposers) and to accept\reject such proposals (for evaluators);
3. new ***algebraic operations*** on the extended data model.

Our extensions have been devised in such a way that $BCDM^{PV}$ can be regarded as a consistent upper layer built upon BCDM, i.e., we have proved that:

i. $BCDM^{PV}$ data model is *reducible* to BCDM one (see Section 4);
ii. $BCDM^{PV}$ manipulation operations are a *"proposal vetting" consistent extension* to BCDM ones (see Section 5);
iii. $BCDM^{PV}$ algebraic operations are *reducible* to BCDM ones (see Section 6).

By proving properties i-iii, we grant that our approach can be added as a support for proposal vetting on top of any of the temporal relational database approaches grounded on the BCDM semantics. This fact enhances the generality of our work, as well as its implementability (see also Section 9). Concerning implementability, it is worth noting that Oracle™ Database, since version 10g, supports both transaction time and valid time consistently with BCDM [Oracle 2005]).

Additionally, uniqueness of (data) model is a major source of clarity for a semantic approach [Snodgrass 1995], and we proved that such a property holds for $BCDM^{PV}$, i.e.,

iv. Uniqueness of model holds for $BCDM^{PV}$ (see Section 4).

In summary, through the extensions 1-3 of BCDM and their properties i-iv we propose a *general* and *implementable* theoretical support to proposal vetting consistent with the relational model.

1.4 Phenomena out of the scope of this paper

To conclude, it is worth citing the phenomena that are outside the scope of the present paper:

- we do not propose any SQL-like extension to the query language, choosing, as in BCDM, to operate at the algebraic level only (this issue is left as a future work, see Section 9); moreover, as in BCDM – since we operate at the semantic level – we do not address issues such as query optimization, integrity constraint support, storage optimization, data indexing;
- since we operate at the semantic level, for the sake of brevity we only focus on proposals and evaluations concerning one tuple in isolation with the others; while this suffices at the semantic level, at the implementation level a high-level interface should also provide users with the possibility of defining "macro-objects" (whose description may usually involve tuples in different relations), and supporting global operations on them (in the form of a unique transaction). Moreover, since we operate at the relational level, we do not cope with issues typically considered within the object-oriented environment, such as the development of a high-level language providing user-friendly facilities to manipulate versioning of complex objects, part-of relations, clustering of object versions, change propagation and configuration [see, e.g., Dittrich and Lorie 1988, Katz 1990, Vines et al. 1988];
- we cope with (proposals\evaluations of) updates to data, not to schema. Therefore the treatment of schema versioning is outside our goals.

1.5 Structure of the paper

The rest of this paper is organized as follows: Section 2 settles the context of our approach, considering several possible application domains, and introduces a running example. Section 3 recalls the BCDM model and its properties. It also analyses the expressive limitations of BCDM as regards the treatment of update proposals and evaluations. Sections 4, 5 and 6 are the core of the paper: they introduce BCDM^{PV}. In particular, Section 4 goes into the technical details, by presenting the data structures. The reducibility and uniqueness of model properties for the new data model are also proven. Section 5 introduces the manipulation operations, and demonstrates that BCDM^{PV} is a "proposal vetting" consistent extension of BCDM. Section 6 describes the algebraic operations we provide, and demonstrates their reducibility to BCDM ones. Section 7 discusses related work, and Section 8 is devoted to the analysis of alternative approaches. Section 9 draws some conclusions and discusses future work directions. Appendix A presents an example application of our proposal vetting approach to a collaborative encyclopedia. Appendix B completes the set of the manipulation operations we have defined, and Appendix C completes the set of the algebraic operators we have introduced. Appendix D is devoted to the proofs. Finally, Appendix E deals with data model and operations in the case in which only transaction time has to be considered.

Please notice that appendices have been added for the sake of completeness and to help the reviewing work. However, we do not think that appendices should be part of the final version of the paper. In the same spirit, Sections 7 and 8 settle the context of our approach, describing a wide spectrum of (even loosely) related approaches and alternative approaches. Both sections might be shortened in the final version.

2 Applicative contexts

In this section, we first briefly analyze several areas in which cooperative modeling\update of data\knowledge is used, devoting more attention to some contexts in which proposal vetting on transaction or bitemporal (i.e., involving both

valid and transaction time) data is important. Then, in Subsection 2.2 we introduce a running example, considering the application domain of clinical guidelines.

2.1 Cooperative modeling\update of data\knowledge

Computer Supported Cooperative Work (CSCW) is a widely spread paradigm. The CSCW time\space matrix [Johansen 1988] has been introduced in order to classify the different possible modalities of computer-based interactions among cooperative users. Specifically, in this paper we focus on the class "different time \ different place" of interactions, meaning that users can interact asynchronously, being in different physical locations.

Such a type of interaction, finalized to the cooperative modeling\update of shared data\knowledge, is an important paradigm in Computer Science, and seems to become more and more important and spread due to the large-scale availability of the Internet. For instance, the construction of a free encyclopedia as Wikipedia, and of vocabularies like Wiktionary, to which hundreds of thousands of authors contribute, is a modern phenomenon with yet undiscovered social and cultural impacts. Among the others, the Wiki technology is also being used in order to build Citizendium, a free encyclopedia in which experts are called to approve the piece of data proposed by contributors, for the sake of reliability and quality. While the above phenomena are relatively recent, the cooperative paradigm has a long tradition in several other areas. For instance, in the field of software engineering, several versioning control tools such as, e.g., CVS [CVS] and in general Computer-Aided Software Engineering (CASE) tools, have been provided in order to support cooperative software development, starting from the 1970s. Analogously, cooperative software systems (such as, e.g., Bugzilla) have been provided in order to track software bugs. Many Content Management Systems, such as, e.g., Alfresco, support cooperative creation, update and publishing of documents.

Although there are relevant differences of scope, goals and methodologies among the above-mentioned types of approaches, some similarities, which are relevant to the goals of our work, can be captured. First of all, most of the above approaches deal with ***data versioning***: different versions of data have to be stored along time. In many cases, the whole history of data needs to be maintained: the set of data versions is stored, together with the time in which the version has been issued (i.e., the ***transaction time*** of the version). In several cases, it is useful to identify, among the ***alternative*** versions of data, a ***reference*** one. Depending on the approach, the reference version may be the latest issued version (i.e., in temporal database terms, the current version; consider, e.g., Wikipedia), or the latest approved version (consider, e.g., Citizendium, in which a team of experts is called to choose among alternatives).

A range of different policies are used in order to regulate the interactions between cooperative users in these contexts. At one extreme of the range, all users are *completely and equally trusted*: for instance, in Wikipedia, any user can introduce a new term, or update a previously introduced one, leading to a new reference version. Stricter policies are adopted by other approaches, especially in cases in which data quality and reliability is a crucial issue. For instance, in the current effort of building a database encyclopedia by Springer, only selected authors are invited to contribute, and their contributions are reviewed by other domain experts. In general, a ***two-level*** process in which insertions\updates are first ***proposed*** and then ***evaluated*** for approval by a team of experts is a widely adopted policy, whenever data\knowledge quality must be granted (consider also, e.g., the Citizendium encyclopedia, based on the Wiki technology).

In principle, all the above-mentioned applications can take advantage of relational DBMSs, exploiting their generality and efficiency. For instance, Wikipedia stores data into a relational database system (i.e., MySQL). However, the ***structured representation*** and ***structured query language*** features provided by the relational framework are fully

exploited especially in domains in which the data\knowledge to be captured is "structured" (as opposed to textual unstructured data). This is the case, for instance, of organizational knowledge like the one in *workflows, protocols, guidelines* and *plans*, which play an important role in many application contexts, providing useful tools to model, control, and\or optimize processes, granting for their efficiency, standardization and\or quality. In all such contexts, data\knowledge regards classes of *entities* (mostly modeling processes\activities) with a structured internal description, and their *interactions* (see, e.g., the example in Subsection 2.2, concerning clinical guidelines). In such cases, the ***valid time*** is usually an intrinsic part of the information to be captured, to describe the temporal extent of entities and\or the temporal constraints on their interactions. Modeling and keeping up-to-date *workflows, protocols, guidelines* and *plans* is usually a complex task, involving the cooperation of several contributors. Specifically, since workflows, protocols and guidelines are usually used as a normative reference for members of institutions adopting them, quality is an essential issue, so that the two-level (proposal vs. evaluation) policy is usually adopted.

2.2 Case study: proposal vetting about clinical guidelines

We introduce two real-world applications where proposal vetting has to be dealt with. The first application, described below, regards clinical guidelines and the second one regards a collaborative encyclopedia such as Citizendium. The application about clinical guidelines is the more complex and interesting, since it involves the treatment of both valid and transaction time. Therefore, we use such an application as a running example in this paper. On the other hand, the collaborative encyclopedia application is simpler (requiring the treatment of transaction time only), so that it is only sketched in Appendix A (it is worth stressing that the case in which only transaction time has to be considered is a trivial restriction of the general bitemporal case described in this paper; see Appendix E for an explicit treatment).

Our research group has a long-term cooperation with Azienda Ospedaliera San Giovanni Battista in Turin (henceforth called "Hospital" for short), one of the largest hospitals in Italy (more than 1000 beds), focusing on the development of a computer-based approach for the management of clinical guidelines.

Clinical guidelines are usually built incrementally, and need to be kept up-to-date, whenever new therapeutic and\or diagnostic procedures are discovered. In practice, different alternative proposals of insertion\update\deletion, issued by specialists, are periodically evaluated by a team of experts who are responsible for the final result. Accepted proposals lead to a new version of the guideline, which becomes the reference one for all the medical and paramedical personnel of the hospital. However, past versions of the guideline must be maintained, e.g., for legal purposes. A very similar approach is also being followed by the Hospital in the creation of a(n internal) clinical vocabulary, which will be resorted to in order to standardize the terminology used in the hospital information system.

As a running example, we consider the guideline for the management of suspected acute pulmonary embolism [BTS 2003; MacDonald et al. 2005] adopted by the Hospital.

The guideline indicates further diagnostic investigations (to confirm or discard the suspect of suspected acute pulmonary embolism) and, in case the suspect is confirmed, it dictates the proper set of therapeutic actions. The first action of the guideline is *pulmonary embolus detection*. In the initial version of the hospital guideline, such an action had to be executed through pulmonary ventilation perfusion scintigraphy, performed using isotope lung scanning (VQS). The estimated cost of such an operation is about 100 €, and image acquisition lasts about 15 minutes.

In Figure 1, we show a little part of the Entity-Relationship (ER) model of the guideline clinical actions (for the sake of brevity, many parts of the whole model are omitted). For simplicity, we use standard ER diagrams, augmented with the possibility of introducing transaction time (Ts and Te stand for the start and the end of the transaction time respectively)

and valid time (Vs and Ve stand for the start and the end of the valid time respectively). The use of four atomic-valued timestamp attributes to represent bitemporal chronons, exploited in this example, is derived from the TSQL2 representational approach [Snodgrass 1995]. A more accurate treatment of temporal aspects at the conceptual level could be obtained using, e.g., ST-USM [Khatri et al. 2004]; however, such a conceptual treatment is outside the goals of this paper.

Notice, in particular, that all the entities and the relationships have a transaction time, since we need to model the full history of the evolution of the guideline into the database. The valid time associated with the "CLINICAL_ACTION" entity models the time when the action is to be executed (starting and ending time), expressed as a temporal distance from the beginning of the execution of the guideline to which the action belongs[1].

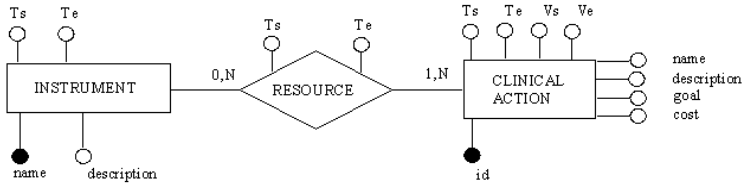

Figure 1: part of the guideline conceptual model.

In Figure 2 we show three relations modeling this part of the conceptual model, where we refer to the "pulmonary embolus detection" action. The transaction time start (2/20/2001) denotes the timestamp when the tuples were entered into the database (i.e., when the guideline was acquired). The value "UC" in the transaction-time end (Te) stands for "Until Changed", which is a special value used to denote the fact that the tuple is still present (not deleted) in the database (we import the use of UC from the BCDM model [Jensen and Snodgrass 1996], see Section 3). In the relation CLINICAL_ACTION, Vs is 0 to denote the fact that the action has to be executed as soon as the guideline about suspected acute pulmonary embolism is started.

[1] It is worth noticing that, in this example, we adopt a non-standard notation for the valid time, since we represent it as a displacement from the starting point of the guideline instead of a displacement from a standard reference point (such as, e.g., the birthday of Christ in the Gregorian calendar) as, e.g., in BCDM.

INSTRUMENT

Name	Description	Ts	Te
VQS	ventilation perfusion scintigraphy	2/20/2001	UC

RESOURCE

INSTR_name	ACT_name	Ts	Te
VQS	101	2/20/2001	UC

CLINICAL_ACTION

id	name	Description	goal	cost	Ts	Te	Vs	Ve
101	pulmonary embolus detection	detection by imaging techniques	diagnosis of pulmonary embolism	100€	2/20/2001	UC	0s	3000s

Figure 2: three relations modeling the part of the conceptual model in Figure 1.

In 2002, such a guideline was modified, to reflect the availability of a more sophisticated tool: computed tomographic pulmonary angiography (CTPA), which can detect pulmonary emboli. Although the estimated cost of the action increases to about 300 €, the use of CTPA has several advantages: the execution time is much shorter (about 15 seconds); CTPA is relatively more available than VQS. Moreover, it is advantageous in term of sensitivity, specificity, positive predictive value, negative predictive value and accuracy [MacDonald et al. 2005].

In the following example, we show how the guideline can be updated, through a session of cooperative work in which different proposers (P1, P2, P3, and P4) introduce different update proposals, and an evaluator (E1) incrementally accepts\rejects such proposals. While the previous part of the example is real (although simplified, for the sake of brevity), the working session we describe is a hypothetical one, and aims at presenting the different possible operations of proposers and evaluators, to be used as a running example in the rest of the paper. The working session is introduced as a sequence of steps. Since in this paper we do not aim at capturing the notion of transaction, we deal with each atomic operation (proposal, acceptance or rejection) as a separate step (step 1 to step 12 below). In other words, proposals\evaluations concerning more than one tuple at the same time are not dealt with by our current approach. Notice, however, that, even if we do not show such a possibility in this example, our definition of update (see Section 4.2.1) supports the update of the value of more than one attribute of the same tuple as a unique action (e.g. in the example operations at steps 4 and 5 could be done by P1 in a single proposal). The extension of our approach to cope with sets of operations performed as a unique transaction is left as a future work (see Section 9).

Step 1. Proposer P1 proposes to insert the new instrument Computed Tomographic Pulmonary Angiography (CTPA) into the INSTRUMENT relation;

Step 2. Evaluators E1 accepts the proposal made in step 1;

Step 3. Proposer P1 proposes to update the relation RESOURCES, to store the fact that now CTPA is the instrument to be used for pulmonary embolus detection within the guideline about suspected acute pulmonary embolism.

Step 4. Proposer P1 proposes to update relation CLINICAL_ACTION to modify the end of the valid time of action 101 (pulmonary embolus detection) from 3000 seconds to 15 seconds;

Step 5. Proposer P1 proposes to further update relation CLINICAL_ACTION to modify the cost of action 101 from 100 to 500;

Step 6. Evaluator E1 accepts the proposal at step 3;

Step 7. Evaluator E1 rejects the proposal at step 5;

Step 8. Proposer P2 proposes to delete the tuple with key 101 from CLINICAL_ACTION;

Step 9. Proposer P3 proposes to update the proposal issued by proposer P1 at step 4, by adding to such an update also the update (from 100 to 300) of the cost;

Step 10. Proposer P4 proposes to update the current evaluator version of the tuple 101 in CLINICAL_ACTION changing the cost from 100 to 400;

Step 11. Evaluator E1 queries the database to check all the active proposals concerning the tuple about pulmonary embolus detection in CLINICAL_ACTION;

Step 12. Evaluator E1 accepts the proposal at step 9.

The sequence of the different operations involving just the tuple about pulmonary embolus detection from CLINICAL_ACTION is graphically shown in Figure 3.

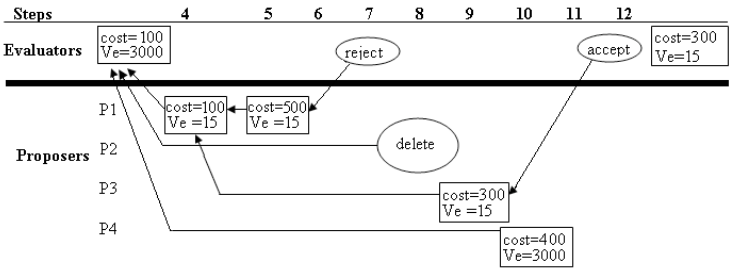

Figure 3: the sequence of operations involving the tuple about the "pulmonary embolus detection" action (steps from 4 to 12) in our example.

3 The BCDM model

BCDM (Bitemporal Conceptual Data Model [Jensen and Snodgrass 1996]) is a unifying data model, which has been developed in order to isolate the "core" notions underlying many temporal relational approaches, including the "consensus" TSQL2 one [Snodgrass 1995].

In BCDM, tuples are associated with *valid time* and *transaction time*. For both domains, a limited precision is assumed (the *chronon* is the basic time unit). Both time domains are totally ordered and isomorphic to the subsets of the domain of natural numbers. The domain of valid times D_{VT} is given as a set $D_{VT}=\{t_1,t_2,\ldots,t_k\}$ of chronons, and the domain

of transaction times as $D_{TT}=\{t'_1,t'_2,\ldots,t'_j\}\cup\{UC\}$ (where UC –Until Changed– is a distinguished value). In general, the schema of a bitemporal conceptual relation $R=(A_1,...,A_n|T)$ consists of an arbitrary number of non-timestamp attributes $A_1, \ldots,A_n$, encoding some fact, and of a timestamp attribute T, with domain $D_{TT}\times D_{VT}$. Thus, a tuple $x=(a_1,\ldots,a_n|t_b)$ in a bitemporal relation r(*R*) on the schema R, henceforth called a BCDM (bitemporal) tuple, consists of a number of attribute values associated with a set of bitemporal chronons $t_{bi}=(c_{ti}, c_{vi})$, with $c_{ti} \in D_{TT}$ and $c_{vi} \in D_{VT}$. The intended meaning of a bitemporal BCDM tuple is that the recorded fact is *true in the modeled reality* during each *valid-time* chronon in the set, and is *current in the relation* during each *transaction-time* chronon in the set. Valid-time, transaction-time and atemporal tuples are special cases, in which either the transaction time, or the valid time, or none of them are present.

Notation. Given a tuple x defined on the schema $R=(A_1,...,A_n, B_1,...,B_l |T)$, we denote with A the set of attributes $(A_1,...,A_n)$. Then x[A] denotes the values in x of the attributes in A, x[T] denotes the set of bitemporal chronons constituting the timestamp of x, $x[T_v]$ and $x[T_t]$ denote the valid and transaction time of a valid-time and transaction-time tuple respectively. ♦

The BCDM model explicitly requires that no two tuples with the same data part (i.e., *value-equivalent* tuples [Snodgrass 1995]) are allowed in the same relation. As a consequence, in BCDM the full time history of a fact is recorded in a single tuple. This choice enhances the semantic clarity of the model, and is essential in order to enforce the property of uniqueness of model introduced below (see property 3.1).

A special routine makes explicit the semantics of the special value UC: as time passes, at each clock tick for each bitemporal chronon (UC,c_v), a new bitemporal chronon (c_t,c_v) is added to the set of chronons, where c_t is the new transaction-time value.

Notation. A bitemporal BCDM tuple x is *current* if it is present at the current time ("now") in the database (i.e., it has not been updated or deleted yet). Formally, this means that the bitemporal chronons of x contain UC as a transaction time, i.e., *current(x)*: $\exists c_v : (UC,c_v)\in x[T]$. ♦

Insertion and *deletion* of tuples are directly defined in BCDM.

For instance, $insert^B(r,(a_1,\ldots,a_n),t_v)$ (where r is a BCDM relation, $(a_1,\ldots,a_n)$ are the non-timestamp values of the tuple to be inserted and t_v is a set of chronons denoting valid time) is defined by three cases:

(i) if $(a_1,\ldots,a_n)$ was never recorded in r, then it has to be added, with timestamp $\{UC\}\times t_v$;

(ii) if $(a_1,\ldots,a_n)$ was part of some previously current state, the tuple recording this is updated with the new valid-time information and is made current;

(iii) if $(a_1,\ldots,a_n)$ is already current in r, the insertion is rejected.

Transaction-timeslice, *valid-timeslice* and *algebraic* operators are defined on the bitemporal model. For instance, let us consider the algebraic operator of bitemporal *natural join* which is defined as follows. Define two relation schemata $R=(A_1,...,A_n, B_1,...,B_h |T)$ and $S=(A_1,...,A_n, C_1,...,C_k |T)$, and let *r* and *s* be instances of R and S respectively. Let A, B, and C stand for $A_1,...,A_n$, $B_1,...,B_h$, and $C_1,...,C_k$ respectively.

$r \bowtie^{B} s = \{z : \exists\, x \in r, \exists\, y \in s$

$(x[A]=y[A] \wedge x[T] \cap y[T] \neq \varnothing \wedge$

$z[A]=x[A] \wedge z[B]=x[B] \wedge z[C]=y[C] \wedge z[T]=x[T] \cap y[T]))\}$ ♦

In the BCDM model, the notion of *snapshot equivalence* [Snodgrass 1995] has been introduced. Informally, two relations are snapshot equivalent if and only if they are equal at each bitemporal chronon, i.e., if they have the same information content. It is a major source of semantic clarity that two relations have the same information content exactly when they are identical. In [Snodgrass 1995], it is proved that BCDM has such a property:

Property 3.1: Uniqueness of data model.

Identity and snapshot equivalence coincide for the BCDM conceptual model.♦

As a second important result, the authors have shown that the semantics of the representational relations in any of the five temporal relational data models considered in [Snodgrass 1995] is identical to that of the corresponding bitemporal conceptual relation in BCDM. This correspondence, known as *structural equivalence*, provides a way of converting relations between different representations, on the basis of the notion of snapshot equivalent conceptual relations. Thus, BCDM operates as a unifying link among different representation approaches, as stated as the main objective of the conceptual model BCDM itself. In particular, in [Snodgrass 1995] it has been proven that BCDM represents the semantic level underlying the TSQL2 approach as well as the approaches of Snodgrass, Jensen, Gadia (more precisely, Gadia-3), McKenzie, and Ben-Zvi.

Finally, *operational equivalence* has been shown as well: namely, extensions to the conventional relational algebra operators, defined in any of the representational models dealt with in [Snodgrass 1995], can be meaningfully mapped to analogous operators in the BCDM conceptual data model.

The following property also holds [Snodgrass 1995]:

Property 3.2: Reducibility.

BCDM algebraic operators reduce to the corresponding snapshot relational algebra operators: with identical arguments, they always return identical results. ♦

In summary, the BCDM model is, from the one side, a general unifying semantic model for several temporal database approaches, including TSQL2; from the other side, it can be seen as un upper “high-level” layer added upon the standard relational theory, since it is a “consistent extension” of snapshot relational model.

However, the treatment of proposal vetting imposes on data structures and manipulation and algebraic operations new requirements, which were outside the scope (and, consequently, of the expressiveness) of BCDM. Such requirements are briefly explored in the next subsection.

3.1 Limitations of BCDM considering proposal vetting

In short, BCDM is intended to model “standard” interactions with temporal data, in which all users directly operate manipulations and arise queries on the given database. As a consequence, these are the major limitations of BCDM, in case proposal vetting phenomena have to be coped with:

1.a The data model supports only one level of data, shared by all the users. In particular, relations (and databases) are interpreted as *conjunctions* of tuples;

2.a All the users can directly access data, through query and manipulation operations. In particular, manipulation operations directly affect the status of the database.

On the other hand, in the proposal vetting context:

1.b Two levels of data need to be supported: the "reference" (accepted) data, and the "proposed" (to-be-evaluated) data. In particular, proposals of update concerning the same tuple must be interpreted in a *disjunctive* ("exclusive or") way (since the acceptance of one of them implicitly involves the rejection of all its alternatives);

2.b Standard users must be distinguished from evaluators, and different manipulation operations must be provided for the two classes. In particular, standard users can only propose insertion\deletion\update of reference data, and the effect of such operations is delayed (since such operations affect reference data only if they are approved by evaluators). The "proposal" level of data is used to support such a delay.

Although, in principle, such limitations could be directly coped with in an ad-hoc way when developing specific applications, we strongly believe that the proposal vetting phenomenon, due to its generality and wide spread, and to the strong impact it has on relational data semantics, deserves a general theoretical approach (providing solid bases for future implementations).

In the rest of the paper we propose BCDM^{PV}, our extension to the BCDM data model's data structures, manipulation operations and algebraic operations, to cope with the "proposal vetting" environment.

4. Extending the data model

In this section, we introduce the data structures of BCDM^{PV}, which extends the BCDM data model to support two levels of data (evaluator data vs. proposer data) and to cope with proposals of insertion, deletion and update. Evaluator data are the "reference" (validated by evaluators) data for all the users of the database. In the "standard" relational (and BCDM) environment all manipulation operations are executed as soon as they are invoked by users. On the other hand, in the proposal vetting context the proposals of insertion, deletion and update need to be stored, waiting for evaluators to evaluate them. Only accepted proposals of operations have an impact on the reference (evaluator-approved) version of the data. Moreover, the history of all proposals is maintained.

To cope with the above issues, in our data model we need to distinguish between accepted data and proposals that still need to be validated by evaluators. To this end, we introduce a two-layered approach, in which:

(1) as anticipated in Section 1, we define two categories of users: a set of proposers, who issue proposals, and a set of evaluators, who can accept\reject them;

(2) we split the data in two levels. Namely, all validated data, accepted by evaluators, are stored in the evaluator data level. Current data in the evaluator data level constitute the reference (accepted) version of data. On the other hand, all the proposals, generated by any proposer, are stored at the proposer data level.

Definition 4.0.1: Proposers and Evaluators.

We term Proposers=$\{p_1,\ldots,p_y\}$ and Evaluators=$\{e_1,\ldots,e_z\}$ the set of proposers and evaluators respectively. ♦

Notice that our approach is independent of whether Proposers and Evaluators are disjoint sets or not (so that different policies can be implemented).

Definition 4.0.2: We define a database as a pair **<DB_Evaluators, DB_Proposers>**. DB_Evaluators is a set of relations $\{r_1(R_1),...,r_k(R_k)\}$ where r_i $(1 \le i \le k)$ is an instance of the schema R_i. DB_Proposers contains, for each relation $r_i \in$ DB_Evaluators, three separate sets:

- **pi(r_i)**, containing proposals of *insertion* into r_i,
- **pd(r_i)**, containing proposals of *deletion* of tuples in r_i, and
- **pu(r_i)**, containing the proposals of *update* (concerning tuples in r_i, pi(r_i), and pu(r_i)). ♦

Both in DB_Evaluators and in DB_Proposers we have to deal with the *valid time* of tuples and with their *transaction time*. In particular, the transaction time is crucial in order to associate manipulation actions (proposals and evaluations) to the time when they are executed. As in BCDM (see Section 3), we denote with T the bitemporal attribute.

Moreover, in DB_Proposers we keep track of the proposer who is the author of each proposal. We denote with P the attribute on the domain Proposers.

In Table 1 we report a summary of the data structures of our extended data model.

Evaluator level	relation *r∈DB_Evaluators* schema *(A_1,...,A_n|T)*
Proposer level	set *pi(r)∈DB_Proposers* of tuples schema *(A_1,…,A_n,P|T)* set *pd(r)∈DB_Proposers* of tuples schema *(A_1,…,A_n,P|T_t)* set *pu(r)∈DB_Proposers* of Proposal-tuples schema *<(A_1, …,A_n), (A_1, …,A_n,P|T)>*

Table 1: summary of the data structures of BCDM^{PV}.

Terminology (value equivalence, weak value equivalence). In the case of tuples in DB_Evaluators, we use the term *value equivalent* in the standard way, to denote tuples that have equal values for the atemporal attributes. In the case of tuples in DB_Proposers, whose schema also includes the proposer attribute P, we consistently say that two tuples x_1 and x_2 on the schema (A_1,…,A_n,P|T) are *value equivalent* if and only if $x_1[A_1,\ldots,A_n,P] = x_2[A_1,\ldots,A_n,P]$, while we say that x_1 and x_2 are *weakly value equivalent* if and only if $x_1[A_1,\ldots,A_n] = x_2[A_1,\ldots,A_n]$, regardless of $x_1[P]$ and $x_2[P]$. ♦

4.1 DB_Evaluators

Definition 4.1.1: DB_Evaluators.

DB_Evaluators is a BCDM database. We denote with R=(A_1,…,A_n|T) the schema of a relation r∈DB_Evaluators. As in BCDM, we do not admit value-equivalent tuples in the same relation r∈ DB_Evaluators (**Condition 4.1.2**). ♦

Here and in the following, T is an implicit bitemporal timestamp attribute, with domain $D_{TT} \times D_{VT}$ (as in the BCDM model).

As in BCDM, we consider also relations in which only the transaction time is present (see, e.g., relations INSTRUMENT and RESOURCE in Figure 2 and the example application to a collaborative encyclopedia in Appendix A). However, transaction-time relations can be easily coped with as a special case of bitemporal relations (the simplified version of some of the proposal vetting operators to the case in which only the transaction time is considered can be found in Appendix E).

4.2 DB_Proposers

In this section, first we quickly introduce the definitions concerning proposals of insertion and of deletion, whose representation does not require extensions to the BCDM data model. Then we move to one of the core contributions of our approach, namely the definition of proposals of update.

4.2.1 Proposals of insertion

Definition 4.2.1.1: pi(r).

Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, we define pi(r) as the set containing the tuples *t* which are proposed for insertion into *r*. The *schema* of pi(r) is $R'=(A_1,\ldots,A_n,P|T)$. As in BCDM, in pi(r) we do not admit value-equivalent tuples (**Condition 4.2.1.2**). Moreover, in pi(r) we do not admit current weakly-value-equivalent tuples with different proposers (**Condition 4.2.1.3**). ♦

Therefore, also the tuples in pi(r) are standard (bitemporal) tuples in the BCDM model.

Condition 4.2.1.3 is motivated by the sake of avoiding redundant situations: we allow different proposers to issue the very same proposal of insertion only at different times. Otherwise, in analogy with BCDM, if a different proposer wanted to issue a proposal weakly value equivalent to a current proposal, e.g., to change the valid time, we would force her\him to edit it as an update to the current proposal itself.

4.2.2 Proposals of deletion

Definition 4.2.2.1: pd(r).

Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, we define pd(r) as the set containing the tuples *t* which are proposed for deletion from *r*. The *schema* of pd(r) is $R'=(A_1,\ldots,A_n,P|T_t)$, where T_t represents the transaction time (defined over the domain D_{TT}). As in BCDM, in pd(r) we do not admit value-equivalent tuples (**Condition 4.2.2.2**). Moreover, in pd(r) we do not admit current weakly-value-equivalent tuples with different proposers (**Condition 4.2.2.3**). ♦

Notice that, in our approach, a tuple in pd(r) identifies the tuple in r to be deleted. Therefore, the valid time is not needed (in fact atemporal attributes univocally identify the evaluator tuples, since value-equivalent tuples are not admitted in the BCDM data model). Thus, tuples in pd(r) are standard transaction-time tuples in the BCDM model.

The motivation of Condition 4.2.2.3 is analogous to that of Condition 4.2.1.3 above.

4.2.3 Proposals of update

Given a relation r∈DB_Evaluators with schema R=(A_1,…,A_n|T), the set pu(r) contains the proposals of updates concerning tuples in r, or in pi(r), or in pu(r).

Notice that, in the proposal vetting context, a proposal of update cannot be simply implemented as a proposal of deletion followed by a proposal of insertion; in fact, in such a case, it would be possible for an evaluator to accept just one of the two operations, obtaining a very different result with respect to the acceptance of a proposal of update. Moreover, as we will see in Section 5, in the proposal vetting environment the acceptance of certain proposals of update may lead to an insertion (without deletion). Therefore, we need an ad hoc operation to define the semantics of the proposal of update atomically, as well as a proper data structure to store proposals waiting for evaluation.

In order to define the set pu(r) of proposals of updates, we first have to introduce our representation of a proposal of update.

In principle, each proposal of update could be modeled independently of the others. However, the underlying semantics is that all the proposals of modification concerning the same tuple must be interpreted as mutually exclusive alternatives, since the acceptance of one proposal implicitly involves the rejection of all the others. In other words, *unlike* the standard relational model and the BCDM model, the proposal vetting context involves coping with *mutually exclusive disjunctions* of pieces of information. We introduce a primitive semantic notion – the Proposal-tuple – to explicitly cope with such a new phenomenon. A Proposal-tuple groups together all the alternative proposals concerning a given tuple (thus resembling, e.g., the notion of *Design Object* in the approach in [Dittrich and Lorie 1988]). As we will see henceforth, defining such a grouping of disjunctive pieces of information as a primitive notion also provides several advantages to our approach, simplifying the definition of manipulation and algebraic operators.

Definition 4.2.3.1: Proposal-tuple.

A Proposal-tuple may either concern (i) a tuple in an evaluator level relation, or (ii) a tuple in a proposal of insertion[2].

In the case (i), given a relation schema R = (A_1, …,A_n|T) (for a relation in DB_Evaluators), let r∈DB_Evaluators be an instance of R and x∈r a tuple in r. A Proposal-tuple pt∈pu(r) concerning x can be defined as follows:

$$pt = \langle o, Alt(alt_1, alt_2, .., alt_m) \rangle$$

where o=x[A_1, …,A_n] and alt_i ($1 \le i \le m$) are BCDM tuples. alt_i in their turns are defined on the schema (A_1, …,A_n,P|T), where the attribute P is used to model the author of the proposal. o (called *origin*) is used in order to univocally identify the tuple x to be updated. and Alt(alt_1,alt_2,..,alt_m) is a disjunctive non-empty set of mutually exclusive tuples referring to the tuple x, representing the different proposals of update concerning x (in other words, $\langle o, Alt(alt_1, alt_2, .., alt_m) \rangle$ is the *notation* we adopt to denote the disjunction $\langle o, alt_1 \rangle$ *xor* … *xor* $\langle o, alt_m \rangle$).

In the case (ii) the schema of a set of proposals of insertion also includes the proposer attribute (i.e., it is of the form R'= (A_1, …,A_n,P|T)). Apart from that, the definition of a Proposal-tuple in such a case is exactly the same as above. In

2 Notice that, in our approach, proposals of update concerning a preceding update proposal *pt*∈pu(r_i) are directly referred to the origin of *pt* (which may be either a tuple in r or in pi(r)); see the explanations in Sections 5 and 8.

fact, the origin o is only used to uniquely identify the tuple to be modified. Since neither value-equivalent proposals of insertion, nor weakly-value-equivalent current proposals of insertions are allowed in our model (see Conditions 4.2.1.2 and 4.2.1.3), the proposer attribute P can be omitted from the origin of the Proposal-tuple even in the case it is a tuple of proposal of insertion.

As in BCDM, in a Proposal-tuple we do not admit value-equivalent alternatives (**Condition 4.2.3.2**). Moreover, in a Proposal-tuple we do not admit current weakly-value-equivalent alternatives, with different proposers, but with equal valid times (**Condition 4.2.3.3**). ♦

Differently from proposals of insertion and of deletion, we admit current weakly-value-equivalent alternative tuples in the same Proposal-tuple, since they have to be interpreted one as the update of the other, proposed by a different proposer. Obviously, in order to avoid redundancy, such an update must propose some change to the previous update proposal: at least as regards the valid time. That is why, through Condition 4.2.3.3, we allow different proposers to issue the very same proposal of insertion, but only with different valid times.

Terminology (schema of a Proposal-tuple). Given the Definition 4.2.3.1, we call the pair <(A_1, …,A_n), (A_1, …,A_n,PlT)> the *schema* of pt. ♦

Terminology (origin, alternatives of a Proposal-tuple). Given the Definition 4.2.3.1, we call x the *origin* of the Proposal-tuple and {alt_1,alt_2,..,alt_m} its *alternatives*. In the definition, o is used in order to uniquely identify x; in the following, we therefore call both x and o "*origin*". ♦

It is worth noticing that we devise our approach at an abstract (semantic) level. Accordingly, we conceive Proposal-tuples as a purely abstract semantic notion, which can be implemented in different ways (for instance in Section 9 we discuss some issues concerning a possible implementation of Proposal-tuples). Also notice that, in order to group the alternatives, we need to identify the origin they refer to. Working at the abstract level, we simply choose to identify the origin tuple through the atemporal part of the tuple itself. .

We now introduce the definition of the functions *origin* and *alternatives*.

Definition 4.2.3.4: origin(pt) and **alternatives(pt).**

Given a Proposal-tuple pt=<o,Alt(alt_1,alt_2,..,alt_n)>,

- origin(pt) = o, and
- alternatives(pt) = {alt_1,alt_2,..,alt_n}♦

Notice that, given a Proposal-tuple pt, *origin(pt)* denotes a standard (atemporal) tuple, while *alternatives(pt)* denotes a non-empty set of standard bitemporal BCDM tuples.

Example. Consider our running example. The Proposal-tuple representing all the alternative update proposals issued until step 10 concerning the evaluator tuple "(101, pulmonary embolus detection, detection by imaging techniques, diagnosis of pulmonary embolism,100)" is shown in Figure 4. For the sake of brevity, here and in the following, we omit the values of the "name" "description" and "goal" attributes. Times are represented in seconds, and we suppose that step

i is performed at time *i*. For the sake of compactness, when we describe the temporal part of the tuples, we do not use the notation of BCDM model, but we use the TSQL2 notation. For example (101, 100) represents the origin of the Proposal-tuple, being 101 the identifier of the action, and 100 its original cost. In the first alternative, the identifier of the action and the cost are unchanged; P1 represents the proposer of the alternative itself; 4 is the transaction time start, i.e., the time at which the alternative is issued by P1, and UC is the transaction time end, meaning that such an alternative is still current in the database. 0 is the valid-time start (since the action must begin as soon as the guideline begins), and 15 is the proposed valid-time end (i.e., P1 proposes to record the fact that the action will last just 15 seconds).

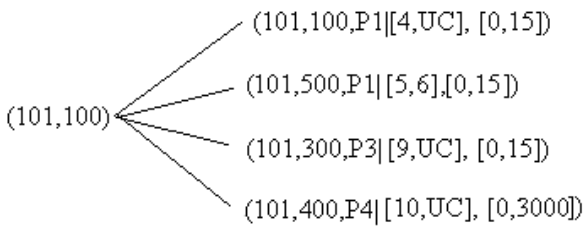

Figure 4: the Proposal-tuple that describes all the proposals of update concerning the tuple with ID=101 until step 10.

Having shown how we cope with update proposals, we can finally define the set pu(r) of update proposals concerning an evaluator relation r.

Definition 4.2.3.5: Proposal-tuple-set pu(r).

Given a relation r∈DB_Evaluators with schema R=(A_1,…,A_n|T), we define pu(r) (henceforth called *Proposal-tuple-set*) as the set containing the Proposal-tuples pt=<o,Alt(alt_1,alt_2,..,alt_m)> whose origin o identifies a tuple in r or in pi(r). The *schema* of pu(r) is <(A_1, …,A_n), (A_1, …,A_n,P|T)>. As in BCDM, Proposal-tuples having the same origin are not admitted in the same Proposal-tuple-set (**Condition 4.2.3.6**).♦

Condition 4.2.3.6 is motivated by consistency with BCDM (notice that origins only include atemporal attributes, therefore value-equivalent origins would be identical).

Additional operators can be introduced in order to select from a Proposal-tuple-set the set of all the origins and of all the alternatives. In particular in the case of the operator π^{Alt}, which selects the set of all the alternatives, if there are *m* value equivalent alternatives, referring to *k* different Proposal-tuples, the output is one tuple having the atemporal attributes of the value equivalent alternatives and one bitemporal attribute consisting of the union of the *m* bitemporal attributes. For the sake of brevity, only an informal definition is given.

Definition 4.2.3.7: Origin-projection and **Alternative-projection.**

Let *s* be a Proposal-tuple-set whose elements are defined over the schema R=<(A_1,...,A_n),(A_1,...,A_n,P|T)>, and let A stand for A_1,….,A_n. The *origin-projection* of *s* is defined as follows:

$\pi^{o}(s)= \{z : (z=origin(pt) : pt \in s)\}$.

The *alternative-projection* of s is defined as:

$\pi^{Alt}(s)= \{z : (z=$ alt

if alt is an alternative in a Proposal-tuple in s, and there is no other alternative value equivalent to alt in any Proposal-tuple in s;

$z[A,P] \leftarrow alt_1[A,P] \wedge z[T] \leftarrow alt_1[T] \cup ... \cup alt_m[T]$

if $alt_1,\ldots,alt_m$ are value-equivalent alternatives in different Proposal-tuples $pt_1,..,pt_k$ in s)}.♦

4.3 Properties of the data model

In this section, we analyze the properties of the data structures of $BCDM^{PV}$. Two properties are essential for the new data model:

(i) **uniqueness** of the model, and

(ii) **reducibility** to the BCDM model.

The uniqueness of the model (see property 4.3.6 and corollary 4.3.7) is a very important property, especially for a semantic model, since "it is a major source of semantic clarity that two instances have the same information content exactly when they are identical" [Snodgrass 1995, page 221]. We demonstrate that such a property also holds for Proposal-tuple-sets.

Commento [T1]: Gia' detto nella sezione 3, ma senza citazione. Uniformare o togliere?

In BCDM, the notion of *snapshot equivalence* has been used in order to formally characterize (bitemporal) relations having the same information content (see property 3.1). In this section, we extend such a notion, in order to apply it also to Proposal-tuple-sets. As a preliminary step, we introduce the notion of transaction- and valid-timeslice operators on BCDM tuples (which is a trivial adaptation of the analogous operators on BCDM relations in [Snodgrass 1995]).

Definition 4.3.1: Transaction-timeslice operator on BCDM tuples.

Let x be a tuple belonging to a relation r in the BCDM model, defined over the schema $(A_1,...,A_n|T)$, and let A stand for $A_1,...,A_n$. Let T_1 be a transaction time not exceeding the current time. The transaction-timeslice operator on BCDM tuples is defined as follows:

$$\rho^{e}_{T1}(x)= \{(a_1,..,a_n|T_v) : x[A]=(a_1,..,a_n) \wedge T_v =\{ c_v : (T_1,c_v) \in x[T]\} \wedge T_v \neq \varnothing\}.♦$$

Definition 4.3.2: Valid-timeslice operator on BCDM tuples.

Let x be a tuple belonging to a relation r in the BCDM model, defined over the schema $(A_1,...,A_n|T)$, and let A stand for $A_1,...,A_n$. Let T_2 be a valid time. The valid-timeslice operator on BCDM tuples is defined as follows:

$$\tau^{e}_{T2}(x)= \{(a_1,..,a_n|T_t) : x[A]= (a_1,..,a_n) \wedge T_t=\{c_t : (c_t,T_2) \in x[T]\} \wedge T_t \neq \varnothing\}.♦$$

The transaction-timeslice operator on transaction-time tuples ρ^{et}_{T1} and the valid-timeslice operator on valid-time tuples τ^{ev}_{T2} are straightforward special cases. We can now extend the above preliminary definitions to deal with Proposal-tuples and Proposal-tuple-sets.

Definition 4.3.3: Transaction-timeslice operator on Proposal-tuples and Proposal-tuple-sets.

Given a Proposal-tuple pt = <o, Alt(alt_1, alt_2,.., alt_k)> and a transaction time T_1 not exceeding the current time, we define the transaction-timeslice operator as follows:

$$\rho^{PT}_{T1}(pt)=<o, Alt(\rho^{e}_{T1}(alt_1), \rho^{e}_{T1}(alt_2) ,\dots , \rho^{e}_{T1}(alt_k))>,$$

where ρ^{e}_{T1} is the transaction-timeslice operator on BCDM tuples (see Definition 4.3.1).

The transaction-timeslice operator on a Proposal-tuple-set s is:

$$\rho^{PV}_{T1}(s)=\{\rho^{PT}_{T1}(pt) : pt \in s\}\blacklozenge$$

Alt($\rho^{e}_{T1}(alt_1)$, $\rho^{e}_{T1}(alt_2)$, …, $\rho^{e}_{T1}(alt_k)$) denotes a disjunctive set of mutually exclusive valid-time BCDM tuples, derived from the alternatives originally stored in pt, after the application of the transaction-timeslice operator on BCDM tuples. Observe that if all $\rho^{e}_{T1}(alt_i)$ ($1 \le i \le k$) provide an empty output, the transaction-timeslice operator on Proposal-tuples provides an empty output.

The definition of valid-timeslice operator is similar.

Definition 4.3.4: Valid-timeslice operator on Proposal-tuples and on Proposal-tuple-sets.

Given a Proposal-tuple pt = <o, Alt(alt_1,alt_2,..,alt_k)> and a valid time T_2, we define the valid-timeslice operator as follows:

$$\tau^{PT}_{T2}(pt)= <o,Alt(\tau^{e}_{T2}(alt_1), \tau^{e}_{T2}(alt_2) ,\dots, \tau^{e}_{T2}(alt_k))>,$$

where τ^{e}_{T2} is the valid-timeslice operator on BCDM tuples (see Definition 4.3.2).

The valid-timeslice operator on a Proposal-tuple-set s is:

$$\tau^{PV}_{T1}(s)=\{ \tau^{PT}_{T1}(pt) : pt \in s\}\blacklozenge$$

Al($\{\tau^{e}_{T2}(alt_1)$, $\tau^{e}_{T2}(alt_2)$, …, $\tau^{e}_{T2}(alt_k)$) denotes a disjunctive set of mutually exclusive transaction-time BCDM tuples, derived from the alternatives originally stored in pt, after the application of the valid-timeslice operator on BCDM tuples. Observe that if all $\tau^{e}_{T2}(alt_i)$ ($1 \le i \le k$) provide an empty output, the valid-timeslice operator on Proposal-tuples provides an empty output.

The transaction-timeslice operator on transaction-time Proposal-tuple-sets ρ^{PVt}_{T1} and the valid-timeslice operator on valid-time Proposal-tuple-sets τ^{PVv}_{T2} are straightforward special cases.

Definition 4.3.5: Snapshot equivalence on Proposal-tuple-sets.

Two Proposal-tuple-sets r and s are snapshot equivalent if for all the transaction times T_1 not exceeding the current time and for all the valid times T_2: $\tau^{PVv}_{T2}(\rho^{PV}_{T1}(r))= \tau^{PVv}_{T2}(\rho^{PV}_{T1}(s))$. ♦

Given the above definitions, we can prove that property 4.3.6 holds (see Appendix D for the proofs). It is worth stressing that Conditions 4.2.3.2 and 4.2.3.6 (in the definitions of Proposal-tuple and Proposal-tuple-set) are essential to obtain such a fundamental property, which "certifies" the semantic clarity of the data model we use.

Property 4.3.6: Uniqueness of model on Proposal-tuple-sets.

Two Proposal-tuple-sets defined over the same schema are snapshot equivalent if and only if they are identical. ♦

Corollary 4.3.7 trivially follows from property 4.3.6, and from the fact that in BCDM^{PV} the uniqueness property trivially holds for the evaluator relations (which are standard BCDM relations; uniqueness of model has been proved for the BCDM model, see Section 3), for the sets of proposals of insertion (which, again, are BCDM tuples) and for the sets of proposals of deletion (which are transaction-time tuples in the BCDM model).

Corollary 4.3.7: In our data model, identity and snapshot equivalence coincide, i.e., two databases over the same evaluator schema in our model are identical if and only if the corresponding evaluator relations and proposal sets are snapshot equivalent. ♦

Another important property is reducibility to the BCDM model. Intuitively, it guarantees that, in the same conditions of the BCDM model (i.e., in case only one level of users\data is taken into account), the BCDM^{PV} model is equivalent to the BCDM one. Reducibility trivially holds, since the pair <DB_Evaluators, DB_Proposers> trivially reduces to a BCDM database in case only one level of data (i.e., DB_Evaluators) is taken into account. This case models the "non-proposal-vetting" context in which users can directly operate insert\delete\update operations on the data.

Property 4.3.8: Reducibility of BCDM^{PV} data model.

The BCDM^{PV} data model reduces to the BCDM data model in case no proposals are proposed\evaluated.

Note. Property 4.3.8 grants that BCDM^{PV} data model is a consistent upper layer built upon BCDM data model to cope with proposal vetting (in turn, BCDM is a consistent upper layer built upon the standard relational model to cope with transaction and\or valid time).

5 Manipulation operations

In this section we define the manipulation operations of BCDM^{PV}. In particular, in our model we introduce two levels of operations: evaluator operations and proposer operations. As regards proposer operations, we define *proposal of insertion*, *proposal of deletion*, and *proposal of update* (which has to be coped with as a primitive operation, as discussed in Subsection 4.2.3). On the other hand, evaluators can either *accept* or *reject* proposals.

5.1 Proposer operations

As already discussed, in the proposal vetting context, given any relation $r \in$ DB_Evaluators, proposers can propose insertions, deletions and updates on *r*. Such operations have no direct effect on DB_Evaluators relations: they are stored into the sets *pi(r), pd(r),* and *pu(r),* waiting for an evaluators' evaluation. In the following we present the definition of proposal of update, which is the most complex operation; the other proposal operations can be found in Appendix B. The simplified versions of the operators where only transaction time is dealt with, can be found in Appendix E.

Given a relation r∈DB_Evaluators, a proposal of update can be used in order to modify (i) a tuple in r, or (ii) a tuple in pi(r), or (iii) an alternative of a Proposal-tuple in pu(r) (we thus allow chaining of update proposals, to support incremental update, i.e., further updating of an already existing proposal of update). As explained in the previous sections, proposal operations do not have a direct effect on DB_Evaluators relations, but are recorded in DB_Proposals waiting for an acceptance\rejection. Specifically, in all cases (i)-(iii) above, the result of a proposal of update is a

Proposal-tuple which, depending on the cases, may be a newly generated one or a modification of an already existing Proposal-tuple. The definition of *propose_update* is quite complex, since it has to cover the three cases, granting also that the operation is admissible (e.g., that it refers to existing tuples) and that it does not introduce incorrect Proposal-tuples (e.g., value-equivalent origins, or (weakly-)value-equivalent alternatives to the same origin, see Conditions 4.2.3.2, 4.2.3.3 and 4.2.3.6).

In the following, we provide a general definition of the *propose_update* operation, covering all the above issues.

Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, the arguments of a *propose_update* operation regarding r are: (i) *r* itself, (ii) the old tuple to be modified, and (iii) the new tuple (i.e., $(a"_1,\ldots,a"_n,p_{new}|t_{vt\_new})$). While the new tuple always has the schema $(A_1,\ldots,A_n,P|T_v)$ (where T_v denotes the valid time), we specify the old tuple in different ways, depending on the case (i)-(iii) we cope with. Specifically, if we cope with an update to an alternative of a Proposal-tuple in pu(r), the alternative is uniquely identified by a pair <origin,alternative> (i.e., $<(a_1,\ldots,a_n),(a'_1,\ldots,a'_n,p_{old})>$).[3] On the other hand, if we cope with an update to an evaluator tuple or to a proposal of insertion, the old tuple is uniquely identified by its atemporal values (i.e., $(a_1,\ldots,a_n)$). In order to shape this case within the above pattern, in such cases we assume that the old tuple is specified by the pair $<(a_1,\ldots,a_n),(a_1,\ldots,a_n)>$.

The propose_update operation first checks the applicability of the proposal operation, through the admissible_propose_update routine.

Definition 5.1.1: admissible_propose_update.
Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, let A stand for $(A_1,\ldots,A_n)$, let $<(A_1,\ldots,A_n),(A_1,\ldots,A_n,P|T)>$ be the schema of pu(r). We define admissible_propose_update, applied to an operation "propose_update(r,$<(a_1,...,a_n),(a'_1,...,a'_n,p_{old})>$, $(a"_1,...,a"_n,p_{new}|t_{vt\_new})$)", as follows:

admissible_propose_update(propose_update(r,$<(a_1,...,a_n),(a'_1,...,a'_n,p_{old})>$, $(a"_1,...,a"_n,p_{new}|t_{vt\_new})$)):

(1) $(\exists x\in r : (x[A]=(a_1,...,a_n)\wedge \text{current}(x)) \vee \exists x\in pi(r) : (x[A]=(a_1,...,a_n)\wedge\text{current}(x))) \wedge$

(2) $(\exists pt\in pu(r) : (\text{origin}(pt)=(a_1,...,a_n) \wedge \exists\, y \in \text{alternatives}(pt) :$
$(y[A] = (a_1',...,a_n') \wedge y[P]= p_{old} \wedge \text{current}(y)) \vee (a_1,...,a_n)=(a'_1,...,a'_n))) \wedge$

(3) $\forall pt \in pu(r)\ (\text{origin}(pt)=(a_1,...,a_n)) \Rightarrow (\neg\exists\, z \in \text{alternatives}(pt) : (z[A] = (a_1",...,a_n") \wedge \text{current}(z) \wedge \rho^{e}_{UC}(z)[T_v] = t_{vt\_new}))) \wedge$

(4) $\forall k\in r\ ((k[A]= (a_1",...,a_n") \wedge \text{current}(k)) \Rightarrow (a_1",...,a_n")=(a_1,...,a_n)) \wedge$

(5) $p_{new} \in$ Proposers♦

A proposal of update is admissible if a conjunction of five conditions (above tagged as (1)-(5)) holds:

---

3 Since in our model we may have value-equivalent alternatives which belong to different Proposal-tuples, a given alternative can be uniquely identified only if also the Proposal-tuple it belongs to is specified. Since, in our approach, each Proposal-tuple in a Proposal-tuple-set is uniquely identified by its origin, we specify a given alternative through the pair <origin, alternative>. Finally, notice that, since we disallow value-equivalent alternatives to the same origin (see Condition 4.2.3.2), the timestamp of the alternative is not needed to identify it.

(1) $(a_1,...,a_n)$ identifies a tuple x in the evaluator relation r or in the proposal of insertion set $pi(r)$ and such a tuple is current (see Section 3 for the definition of current(x));

(2) either (i) the input $<(a_1,...,a_n),(a'_1,...,a'_n, p_{old})>$ identifies a current alternative of a Proposal-tuple pt in $pu(r)$, or (ii) it identifies a tuple in r or (iii) in $pi(r)$ (given the convention on the input format we have discussed above, the condition $(a_1,...,a_n)=(a'_1,...,a'_n)$ holds exactly in cases (ii) and (iii));

(3) there is not any alternative z of a Proposal-tuple $pt \in pu(r)$ with origin $(a_1,...,a_n)$ which is weakly value equivalent to the input $(a_1'',\ldots,a_n'')$, is current, and has (at transaction time equal to UC) the same valid time of the input (i.e., t_{vt_new}; notice that $\rho^e_{UC}(z)\ [T_v]$ denotes the valid time of the alternative z when its transaction time is equal to UC). Intuitively, this condition avoids that value-equivalent alternatives are inserted in the same Proposal-tuple (see Condition 4.2.3.2), and also that weakly-value-equivalent alternatives with different proposers, but with the same valid time, are inserted in the same Proposal-tuple (see Condition 4.2.3.3 and its explanation);

(4) there is no current tuple $k \in r$ which is value equivalent to the new proposal $(a_1'',...,a_n'')$, except (possibly) the origin itself. In such a case, the proposal concerns an update to the valid time of the origin. This condition is used to disallow a new proposal $(a_1'',...,a_n'', p_{new}\ |t_{vt\_new})$ which is value equivalent to a current tuple t' in the evaluator level relation, which is not the origin of the Proposal-tuple to be modified. In fact, the acceptance of such a proposal of update would lead, in the evaluator relation, to the deletion of the old tuple, and to the insertion of the new proposal. However, since the new proposal would be value equivalent to t', an insertion of $(a_1'',\ldots\ ,a_n''|t_{vt\_new})$ is not possible (insertion of value-equivalent tuples is explicitly disallowed in BCDM [Snodgrass 1995]);

(5) the proposer p_{new} belongs to the set of proposers.

We can now define our operator to propose updates.

In order to simplify the formulae, we introduce a function to create Proposal-tuples.

Definition 5.1.2: create_pt.

Given a relation in DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, create_pt takes as an input an atemporal tuple o defined on the schema $(A_1,\ldots,A_n)$ and a set $\{alt_1,alt_2,..,alt_m\}$ of non-value-equivalent tuples on the schema $(A_1,\ldots,A_n,P|T)$, and gives as an output a Proposal-tuple having o as an origin and $\{alt_1,alt_2,..,alt_m\}$ as alternatives, i.e.,

create_pt $(o,\{alt_1,alt_2,..,alt_m\}) = <o,Alt(alt_1,alt_2,..,alt_m)>$ ♦

Notation. The manipulation operations are described using the following general pattern:

Operation(parameters)

if admissibility_conditions

then

begin

if conditions$_1$ **then** actions$_1$

….

else if conditions$_n$ **then** actions$_n$

end

where we intend that the parameters and the (logical) variables quantified in admissibility_conditions are *global* to the whole operation, while the (logical) variables in each condition $conditions_i$ ($1 \le i \le n$) are *local* to the corresponding action $actions_i$.

In simple cases, the begin-end part of the procedure may just reduce to a sequence of actions. ♦

Definition 5.1.3: propose_update.

Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, let A stand for $(A_1,\ldots,A_n)$, let $<(A_1,\ldots,A_n), (A_1,\ldots,A_n,P|T)>$ be the schema of pu(r), and let $(A_1,\ldots,A_n,P|T)$ be the schema of a new proposal of update. We define propose_update as follows:

propose_update(r, $<(a_1,\ldots,a_n), (a'_1,\ldots,a'_n, p_{old})>$,$(a''_1,\ldots,a''_n, p_{new}|t_{vt\_new})$):
if(admissible_propose_update(propose_update(r,$<(a_1,\ldots,a_n),(a'_1,\ldots,a'_n,p_{old})>$,$(a''_1,\ldots,a''_n,p_{new}|t_{vt\_new})$))) **then**
begin
(1) **if** $(\neg\ \exists\ pt \in pu(r) : origin(pt)=(a_1,\ldots,a_n))$
then $pu(r) \leftarrow pu(r) \cup \{create\_pt((a_1,\ldots,a_n),\{(a''_1,\ldots,a''_n, p_{new}|\{UC\}\times t_{vt\_new})\})\}$
(2) **else if** $(\exists\ pt \in pu(r) : (origin(pt)=(a_1,\ldots,a_n)\ \wedge\ \forall y \in alternatives(pt)\ (y[A]=(a_1'',\ldots,a_n'') \Rightarrow y[P]\neq p_{new})))$
then $pu(r) \leftarrow pu(r) - \{pt\} \cup \{create\_pt((a_1,\ldots,a_n),alternatives(pt) \cup \{(a''_1,\ldots,a''_n,p_{new}|\{UC\}\times t_{vt\_new})\})\}$
(3) **else if** $(\exists\ pt \in pu(r) : (origin(pt)=(a_1,\ldots,a_n)\ \wedge\ \exists\ y \in alternatives(pt) : (y[A] = (a_1'',\ldots,a_n'') \wedge y[P]= p_{new})))$
then $pu(r) \leftarrow pu(r) - \{pt\} \cup \{create\_pt\ ((a_1,\ldots,a_n), alternatives(pt) - y \cup$
$\{(a''_1,\ldots,a''_n,p_{new}|y[T]- uc\_ts(y[T]) \cup \{\{UC\}\times t_{vt\_new}\})\})\}$
end♦

First, the admissibility of the update proposal is checked. If admissibility holds, three different (mutually exclusive) cases must be considered (otherwise the operation has no effect, and an appropriate warning may be signaled):

(1) the input $(a_1,\ldots,a_n)$ does not identify any already existing Proposal-tuple $pt \in pu(r)$; in such a case a new Proposal-tuple is inserted into $pu(r)$, having as an origin the input origin $(a_1,\ldots,a_n)$ and having the input proposal as the (unique) alternative. Notice that the bitemporal timestamp of the alternative has UC as a transaction time, and the input t_{vt_new} as a valid time (i.e., it is obtained by performing the Cartesian product $\{UC\}\times t_{vt\_new}$);

(2) the input $(a_1,\ldots,a_n)$ identifies an already existing Proposal-tuple $pt \in pu(r)$, and the proposer p_{new} has not proposed any alternative proposal in pt which is value equivalent to the current proposal $(a_1'',\ldots,a_n'',p_{new})$. In such a case the new alternative $(a''_1,\ldots,a''_n,p_{new}|\{UC\}\times t_{vt\_new})$ is added to the already existing alternatives of pt;

(3) the input $(a_1,\ldots,a_n)$ identifies an already existing Proposal-tuple $pt \in pu(r)$, and the proposer p_{new} has already proposed an alternative of pt value equivalent to $(a_1'',\ldots,a_n'')$. In such a case the alternative is updated with the new temporal information: notice that, having satisfied condition (3) in Definition 5.1.1 implies that in this third case, we are just modifying the valid time. Adding $(a_1'',\ldots,a_n'')$ as a new alternative would not be possible, since value-equivalent alternatives are not admitted in the same Proposal-tuple (see Condition 4.2.3.2). Note that *uc_ts* is a function that gives as an output the set of all bitemporal chronons (UC, c_v) (i.e., all chronons having UC as their transaction time) from the timestamp of the tuple and it is defined as in BCDM [Snodgrass 1995]:

Definition 5.1.4: $uc\_ts(t_b) = \{(UC, c_v) : (UC, c_v) \in t_b\}$♦

The simplified versions of the operators where only transaction time is dealt with, can be found in the Appendix E.

Example. Referring to our running example, the proposal of update issued by proposer P3 at step 9 is coped with in our approach as follows:

propose_update(CLINICAL_ACTION,<(101,100),(101,100,P1)>, (101,300,P3|[0,15]))

In particular proposer P3 makes a proposal of update to the proposal issued by P1 at step 4, which refers to the tuple with ID=101 in the DB_Evaluators relation CLINICAL_ACTION (see Figure 2). Before step 9, the status of the Proposal-tuple concerning tuple 101 in pu(CLINICAL_ACTION) is shown in the left part of Figure 5.

The proposal of update is admissible, because (1) *(101,100)* identifies a current tuple that belongs to *CLINICAL_ACTION*; (2) *<(101,100), (101,100,P1)>* identifies a current alternative of a Proposal-tuple in *pu(CLINICAL_ACTION)*; (3) there is not any alternative in the Proposal-tuple with origin *(101,100)* which is weakly value equivalent to the input *(101,300),* is current, and has (at transaction time equal to UC) the same valid time; (4) there is no current tuple in *CLINICAL_ACTION* which is value equivalent to the new proposal *(101,300)*; (5) P3 belongs to the set of proposers. Action (2) in formula 5.1.3 is then performed, since the input origin *(101,100)* identifies an already existing Proposal-tuple in *pu(CLINICAL_ACTION)*, and *P3* has not proposed any alternative proposal in such a Proposal-tuple which is value equivalent to the current proposal *(101,300)*. The new alternative *(101,300,P3|[9,UC],[0,15])* is added to the already existing alternatives of *(101,100)* as shown on the right part of Figure 5.

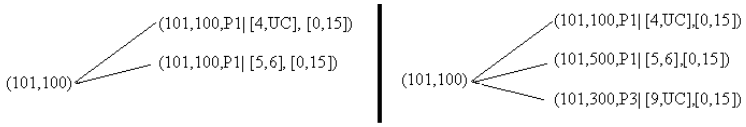

Figure 5: the Proposal-tuple concerning the tuple with ID=101 in CLINICAL_ACTION, before (left side) and after (right side) the execution of the propose_update at step 9.

It is worth noticing that, given a relation $r \in$ DB_Evaluators and a Proposal-tuple $pt=<o,Alt(alt_1,\ldots,alt_n)> \in pu(r)$, in the case a new tuple alt_{new} is proposed to update an alternative $<o,alt_i>$ ($alt_i \in \{alt_1,\ldots, alt_n\}$) of *pt* (as in the above example), the *propose_update* operation adds the new proposal as a new alternative $<o, alt_{new}>$ of *pt* (i.e., *pt* becomes $<o,Alt(alt_1,\ldots, alt_n, alt_{new})>$) instead of creating a new Proposal-tuple $pt'=<alternative,\{alt_{new}\}>$ having as an origin the alternative to be updated. In other words, although we cope with chains of proposals of update, we do not explicitly store the whole chaining of updates in Proposal-tuples, since we directly relate each proposal to the origin tuple to be modified (and not to the alternative proposal it directly modifies). This is a deliberate choice we made to simplify both the data model and the definition of the algebraic operations. We motivate such a choice in Section 8, where we discuss alternative approaches.

5. 2 Evaluator operations

Besides performing insertion and deletion operations (which are standard BCDM operations on the DB_Evaluators database), in our approach evaluators can take into account DB_Proposers proposals, rejecting or accepting them.

The rejection of a proposal has no effect on the DB_Evaluators database. Since we want to retain the whole database history, including the history of proposals, a rejected proposal is not physically deleted from DB_Proposers database; instead, its bitemporal timestamp is made not current by "*closing*" it: in other words, the bitemporal chronons with UC as a transaction time have to be removed from its timestamp (see for instance the transaction time [5,6] for the second alternative of the Proposal-tuple in Figure 4, representing the rejection of the alternative itself at time 7). As a specific policy, for the sake of space efficiency, *vacuuming* [Snodgrass 1995] can be used to prune "closed" tuples which are older than a given date from DB_Proposers. On the other hand, the acceptance of a proposal is used by evaluators to make a given current proposal effective, i.e., to execute it on the DB_Evaluators database. Notice that, besides causing a modification of the DB_Evaluators database, the acceptance of a proposal may also have some effects on the content of DB_Proposers, since proposals that are mutually exclusive alternatives of the accepted one need to be "closed".

As an example of an evaluator operation, in the following we describe our operation for accepting an update proposal. The other evaluator operations are simpler, and can be found in Appendix B.

An acceptance of a proposal of update is used by evaluators in order to update the DB_Evaluators data according to the given proposal. Only proposals that are current may be accepted by evaluators. As anticipated, to enforce the semantics that alternative proposals in a Proposal-tuple are mutually exclusive, the acceptance of an update proposal must "close" the alternatives proposals. Moreover, the accepted tuple as well as the deletion and\or insertion proposals concerning the tuple itself must be "closed". Such operations are performed through the *delete_alternatives* routine (see Definition B.10 in Appendix B).

We can now define the operation of acceptance of a proposal of update. The arguments of the *accept_update* operation are the DB_Evaluators relation r to be modified, the selected alternative $<(a_1,...,a_n),(a'_1,...,a'_{n,}\ p_{new})>$ of a Proposal-tuple in $pu(r)$, and the evaluator e.

As a first step, admissible_accept_update is invoked in order to check the acceptability of the operation.

Definition 5.2.1: admissible_accept_update.

Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, let A stand for $A_1,\ldots,A_n$, let $<(A_1, \ldots,A_n), (A_1, \ldots,A_n,P|T)>$ be the schema of pu(r); we define admissible_accept_update as follows:

admissible_accept_update(accept_update(r, $<(a_1,...,a_n), (a'_1,...,a'_{n,}\ p_{new})>$, e)):

(1) $\exists$ pt∈pu(r) : origin(pt)=$(a_1,...,a_n)$ ∧

(2) $\exists y \in$ alternatives(pt) : y[A] = $(a_1',...,a_n')$ ∧ y[P] = p_{new} ∧ current(y) ∧

(3) $\forall$ z∈r (z[A]=$(a_1',...,a_n')$ ∧ current(z)) ⇒ $(a_1',...,a_n')$ =$(a_1,...,a_n)$) ∧

(4) e ∈Evaluators♦

The operation is admissible if (the conjunction of) four conditions hold:

(1) the input origin $(a_1,...,a_n)$ identifies a Proposal-tuple $pt \in pu(r)$;

(2) $(a_1',\ldots,a_n', p_{new})$ identifies a current alternative of pt;

(3) there is no current tuple $z \in r$ which is value equivalent to the chosen alternative $(a_1',...,a_n')$, except (possibly) the origin itself (see the comments to Definition 5.1.1, part (4));

(4) *e* is an evaluator.

Concerning the admissibility conditions, notice that, since the data stored in the database could change, it is possible that a proposal is admissible when it is issued (i.e., it is consistent with the status of DB_Evaluators), but it is not admissible any more at acceptance time (e.g., because the tuple to be updated is no longer current in the database, e.g., due to a direct deletion operation performed by an evaluator). We choose to perform admissibility checks at acceptance time following a "lazy evaluation" policy in which every acceptance operation locally examines just the portion of the database it is concerned with (and the effect on the proposals in DB_Proposers of direct insertion and deletion operations performed by evaluators is not checked as soon as possible – but only in a "lazy" way, if some further operation regarding the modified data will be accepted\executed). Although our approach might be easily adapted to perform checks as soon as possible, a "lazy evaluation" strategy appears to us more efficient from a computational point of view.

We can finally define the accept_update operation.

Definition 5.2.2: accept_update.

Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, let A stand for $A_1,\ldots,A_n$, let $<(A_1, \ldots,A_n), (A_1, \ldots,A_n,P|T)>$ be the schema of pu(r); we define accept_update as follows:

accept_update(r, $<(a_1,...,a_n), (a'_1,...,a'_{n,}\, p_{new})>$, e)
if (admissible_accept_update(accept_update(r, $<(a_1,...,a_n), (a'_1,...,a'_{n,}\, p_{new})>$, e)))
then
begin
(1) **if** $(\neg\ \exists\ x \in r : x[A] = (a_1,...,a_n)\ \wedge\ current(x))\ \wedge\ (\exists\ y \in pi(r) : y[A] = (a_1,...,a_n)\ \wedge\ current(y))$
then $insert^B\ (r,(a'_1,...,a'_n), \rho^e_{uc}(y)[T_v])$; delete_alternatives(r, $(a_1,...,a_n)$)
(2) **else if** $(\exists\ x \in r : x[A]=(a_1,...,a_n) \wedge current(x) \wedge \exists\ pt \in pu(r) : origin(pt)=(a_1,...,a_n)\ \wedge$
$\exists y \in alternatives(pt) : y[A] = (a_1',...,a_n') \wedge y[P] = p_{new} \wedge current(y))$
then
$delete^B(r, (a_1,...,a_n))$; $insert^B\ (r,(a'_1,...,a'_{n,}), \rho^e_{uc}(y)[T_v])$;
delete_alternatives(r, $(a_1,...,a_n)$)
end◆

Accept_update first checks the admissibility of the operation. If it is admissible, two cases must be distinguished:

(1) the evaluator is accepting an update to a proposal of insertion. In such a case, the *accept_update* operation uses the BCDM $insert^B$ routine [Snodgrass 1995] to insert into *r* the tuple $(a'_1,...,a'_n|\rho^e_{uc}(y)[T_v]))$ where $\rho^e_{uc}(y)[T_v]$ denotes the valid time of the tuple *y* when the transaction time is UC. Moreover, *accept_update* also "closes" the Proposal-tuple *pt* and (possibly) proposals of insertion and deletion concerning $(a_1,...,a_n)$ (through the *delete_alternatives* routine). Notice that, in this case, no tuple needs to be deleted from the evaluator relation *r* (since the update concerns a new tuple proposed for insertion, and not a tuple in *r*);

(2) the evaluator is accepting an update to a tuple in *r*. In such a case, the *accept_update* operation first deletes (using the BCDM delete^B operation [Snodgrass 1995]) the tuple $(a_1,...,a_n)$ from *r*, and then performs the same operations as in the first case.

The simplified versions of the admissible_accept_update and accept_update operators where only transaction time is dealt with can be found in the Appendix E.

Example. Referring to our running example, at step 12 the proposal of update issued by proposer P3 at step 9 is accepted by E1 as follows:

accept_update(CLINICAL_ACTION,<(101,100),(101,300,P3)>, E1)

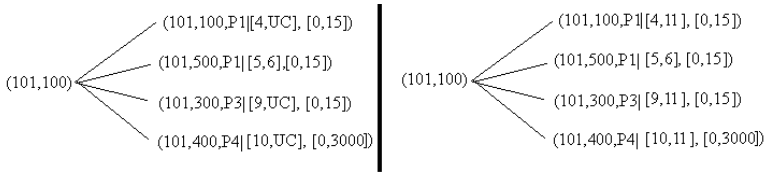

Figure 6: the Proposal-tuple concerning the tuple with ID=101 in CLINICAL_ACTION, before (left side) and after (right side) the execution of the accept_update at step 12.

The accept operation is admissible, because (1) the input origin *(101,100)* identifies a Proposal-tuple in *pu(CLINICAL_ACTION)*, (2) *(101,300,P3)* identifies a current alternative of such a Proposal-tuple, (3) there is no current tuple in *CLINICAL_ACTION* which is value equivalent to the chosen alternative *(101,300)* and (4) *E1* is an evaluator. The action (2) in formula 5.2.3 is performed because the evaluator is accepting an update to a tuple in *CLINICAL_ACTION*. Thus, the *accept_update* routine first deletes (using the BCDM delete^B operation [Snodgrass 1995]) the tuple *(101,100)* from *CLINICAL_ACTION*, and then uses the BCDM insert^B routine [Snodgrass 1995] to insert into *CLINICAL_ACTION* the tuple *(101,300|[12,UC],[0,15])*. Moreover, *accept_update* also executes the *delete_alternatives* routine on *(101,100)* in order to "close" such a Proposal-tuple. The resulting Proposal-tuple in pu(CLINICAL_ACTION) is shown on the right side of Figure 6.

Figure 7 reports the updated content of the CLINICAL_ACTION relation after the acceptance at time step 12.

CLINICAL_ACTION

id	**name**	**description**	**goal**	**cost**	**Ts**	**Te**	**Vs**	**Ve**
101	pulmonary embolus detection	detection by imaging techniques	diagnosis of pulmonary embolism	100€	2/20/2001	11	0	3000s
101	pulmonary embolus detection	detection by imaging techniques	diagnosis of pulmonary embolism	300€	12	UC	0	15s

Figure 7: the updated content of the CLINICAL_ACTION relation after the acceptance operation at step 12.

5.3 Properties of manipulation operations

We now analyze the properties of our manipulation operations. The fundamental property is the fact that we have defined our manipulation operations in such a way that they are a *consistent extension* of the ones in the BCDM model, which guarantees that all the operations performed in BCDM can also be performed in our extended model, with the same syntax and the same results.

The goal of our approach is to be as general as possible, given the proposal vetting typology of cooperative work on relational data. Since we aim at providing a general semantic framework, coping with specific domain\task-dependent policies is outside the goal of our work, and we see these policies as refinements of our abstract framework, to be devised at the implementation level (for instance, whether the evaluation is performed by a specific evaluator alone, or through a voting mechanism between evaluators, basically does not affect our framework). However, in order to analyse the properties of our operators, two basic cases must be distinguished:

(Policy 1) We let evaluators directly insert and delete relations in DB_Evaluators;

(Policy 2) Direct insertion, deletion and modify operations on DB_Evaluators relations are not allowed to evaluators.

Under Policy 1, property 5.3.1 holds, since BCDM insert^B and delete^B operations can be performed by evaluators.

Property 5.3.1: Consistent extension of BCDM.

Under Policy 1 our manipulation language is a consistent extension of the BCDM model (supposing that all users are evaluators).♦

On the other hand, if Policy 2 is considered, in our model there is no direct way of inserting or deleting tuples in DB_Evaluators. In fact, changes to relations in DB_Evaluators can only be performed in two steps: first, a proposal must be issued (by proposers), and secondly, it has to be accepted by evaluators. We can therefore prove a less strict property. In fact, under Policy 2 our approach is a "proposal vetting" consistent extension of BCDM, in the sense that, in case we assume that the set of users in the BCDM model is equal to both the sets Evaluators and Proposers in our approach (i.e., each BCDM user is both an evaluator and a proposer in our approach), and each BCDM user operation Op^B is actually implemented (in our model) as an atomic pair of operations <propose_Op; accept_Op>, then each operation in the BCDM model can also be performed in our approach, leading to the same results, as concerns the data in DB_Evaluators only.

Property 5.3.2: "Proposal vetting" consistent extension of BCDM.

Under Policy 2, considering that all users are both evaluators and proposers, our model in which each manipulation operation Op^B is executed as an atomic pair of operations <propose_Op; accept_Op> is a *"proposal vetting" consistent extension* of the BCDM model (considering only data in DB_Evaluators). ♦

This means that, even if Policy 2 is enforced, users can obtain in our approach the same results that they could obtain in BCDM, by performing pairs of <propose_Op; accept_Op> operations instead of standard ones.

6 Relational Algebra

In this section, we define algebraic operators on the new data model.
At the evaluator level, since the relations are BCDM relations, algebraic operators are exactly the same as in the standard BCDM model. The same holds at the proposer level for the proposals of insertion, where standard bitemporal BCDM tuples are used, and for proposals of deletion, where transaction-time BCDM tuples are used. On the other hand, the treatment of proposals of update demands for the definition of new algebraic operators operating on Proposal-tuple-sets (as well as for the definition of operators allowing one to combine a Proposal-tuple-set with a set of BCDM tuples).

By passing, notice that, given a relation r∈DB_Evaluators, and its Proposal-tuple-set pu(r), BCDM algebraic operators can also be applied on the *origin-projection* (i.e., $\pi^{o}(pu(r))$) and on the *alternative-projection* (i.e., $\pi^{Alt}(pu(r))$) of pu(r), which are sets of atemporal and bitemporal BCDM tuples respectively (see Definition 4.2.3.7).

Algebraic operators on Proposal-tuple-sets are very useful, since they support the possibility of querying data, navigating and aggregating them. Therefore they can help evaluators in taking their acceptance\rejection decisions, as well as proposers in proposing updates to data. For example, selection can be used in order to focus the attention only on a subset of proposals that satisfy some conditions. Moreover, as highlighted in our running example, the data concerning the same guideline action can be stored into more than one relation. Therefore, join operations can be useful for proposers and evaluators, e.g., in order to have a global view of all the data concerning the same action.

Notice that, in our approach, algebraic operators are used only to query data. On the other hand, the tuples in the relations resulting from the application of algebraic operators cannot be directly accepted\rejected by evaluators. In fact, such operations would be meaningless, since accepted proposals must conform the schema of data at the evaluators' level.

We present here the natural join operator. The other basic operators (union, difference, selection and projection) can be found in Appendix C. Other operators (e.g., cartesian product) can be defined on the basis of such operators.

Notation.

Here and in Appendix C we use the following notation in order to define the algebraic operators on Proposal-tuple-sets:

Operation = {z :: <origin(z),alternatives(z)> :
if conditions
then
assignments_to_origin;
alternatives(z) = {alt :
    if alternatives_conditions
    then assignments_to_alt;
    ….
}

…..
}

In other words, we characterize the output Proposal-tuple-set as a set of Proposal-tuples z of the general form <origin(z),alternatives(z)>, that can be defined by mutually exclusive cases. Each case concerning the origin is characterized by a set of conditions *conditions* and leads to a set of assignments (*assignments_to_origin*) defining the output origin. In turn, the corresponding alternatives are defined by mutually exclusive cases.

In such formulae, we assume the standard "nesting" policy for the scope of the variables in the conditions (so that, e.g., assignments_to_alt can use only the variables introduced by the corresponding *alternatives_conditions* and by the *conditions*).

Notice that, in some simple cases (such as the Definition 6.1 below), conditions are absent (so that assignments are directly used, instead of "if conditions then assignments" patterns). ♦

Definition 6.1: natural join $\bowtie^{PV}$.

Given the Proposal-tuple-sets $s_1=pu(r_1)$ and $s_2=pu(r_2)$ corresponding to relations $r_1 \in$ DB_Evaluators and $r_2 \in$ DB_Evaluators with schema $R_1=(A_1,\ldots,A_n,B_1,\ldots,B_m \mid T)$ and $R_2=(A_1,\ldots,A_n,C_1,\ldots,C_k \mid T)$ respectively, let the schema of s_1 and s_2 be $<(A_1,\ldots,A_n,B_1,\ldots,B_m)$, $(A_1,\ldots,A_n,B_1,\ldots,B_m,P \mid T)>$ and $<(A_1,\ldots,A_n,C_1,\ldots,C_k)$, $(A_1,\ldots,A_n,C_1,\ldots,C_k,P \mid T)>$ respectively. Natural join $\bowtie^{PV}$ (where "PV" stands for proposal vetting) provides as an output a Proposal-tuple-set defined over the schema $<(A_1,\ldots,A_n,B_1,\ldots,B_m,C_1,\ldots,C_k)$, $(A_1,\ldots,A_n,B_1,\ldots,B_m,C_1,\ldots,C_k,P \mid T)>$. $\bowtie^{PV}$ is defined as follows (let A stand for $A_1,\ldots,A_n$, B for $B_1,\ldots,B_m$ and C for $C_1,\ldots,C_k$):

$r \bowtie^{PV} s = \{ z :: <origin(z),alternatives(z)> :$

if $\exists\, pt_1 \in s_1, \exists\, pt_2 \in s_2 : origin(pt_1)[A] = origin(pt_2)[A]$

$\wedge\ \exists alt_1 \in alternatives(pt_1), \exists\, alt_2 \in alternatives(pt_2) :$

$alt_1[A,P] = alt_2[A,P] \wedge\ alt_1[T] \cap alt_2[T] \neq \varnothing$

then

$origin(z)[A] \leftarrow origin(pt_1)[A]; origin(z)[B] \leftarrow origin(pt_1)[B];$

$origin(z)[C] \leftarrow origin(pt_2)[C];$

$alternatives(z) = \{alt :$

$alt[A,P] \leftarrow alt_1[A,P]; alt[B] \leftarrow alt_1[B]; alt[C] \leftarrow alt_2[C];$

$alt[T] \leftarrow alt_1[T] \cap alt_2[T]\}$

$\}$♦

The result of proposal-vetting natural join is a set of Proposal-tuples built as follows. Two proposal-tuples pt_1 and pt_2 with origins value equivalent on the common attributes $A_1,\ldots,A_n$ are merged into one Proposal-tuple having as an origin the standard (atemporal) natural join of the origins. The alternatives of the new tuple are built by performing the standard natural join on the atemporal attributes, and the intersection of bitemporal timestamps. Only alternatives whose temporal intersection is not empty are recorded as an output. Notice that the proposer attribute P can be renamed in order to avoid joining Proposal-tuples on the attribute itself (i.e., to merge also alternatives with different proposers).

We have defined our algebraic operators on Proposal-tuple-sets in such a way that they have the property of reducibility [Snodgrass 1995] with respect to BCDM algebraic operators.

Reducibility is a fundamental property of our operators, since it guarantees that the semantics of BCDM operators is preserved in our more complex counterpart. To define reducibility, we define the *convert* operator, that maps a Proposal-tuple to a set of standard BCDM tuples into a BCDM relation, in such a way that each tuple has as a schema the union of the schemata of the origin and of the alternatives (of course, renaming is used to retain common attributes).

Definition 6.2: convert.

Given a Proposal-tuple-set *s* defined over the schema $<(A_1, \ldots,A_n), (A_1, \ldots,A_n,P|T)>$, the result of convert(s) is a BCDM relation defined over the schema $(A_1, \ldots,A_n, A'_1, \ldots,A'_n,P|T)$ (where attributes $A'_1, \ldots,A'_n$ are a renaming of $A_1, \ldots,A_n$ respectively) defined as follows:

$$\text{convert}(s)=\{(a_1,\ldots,a_n,a'_1,\ldots,a'_n, p\,|T) : \exists\, pt\in s : (a_1,\ldots,a_n)=\text{origin}(pt) \wedge (a'_1,\ldots,a'_n,p\,|T) \in \text{alternatives}(pt)\}\blacklozenge$$

Note. It is worth stressing that, although the convert operation produces as an output relations which "syntactically" conform relations in the BCDM model, this does not mean that the BCDM model can directly capture (the semantics of) Proposal-tuple-sets. As a matter of fact, BCDM relations only support conjunctions of tuples. On the other hand, due to the disjunctive semantics of Proposal-tuples (as it clearly emerges from the definition of the manipulation operations in Section 5), tuples in *convert(s)* must be interpreted as a conjunction (one for each Proposal-tuple in *s*) of disjunctions (one for each alternative in the same Proposal-tuple). In Section 9 we will briefly address implementation issues, sketching how such a disjunctive interpretation can be supported.

Property 6.3: Reducibility.

$BCDM^{PV}$ algebraic operators on Proposal-tuple-sets are reducible to BCDM algebraic operators, i.e., for each algebraic unary operator Op^{PV} in our model, and indicating with Op^{B} the corresponding BCDM operator, for each Proposal-tuple-set *s*, the following holds (the analogous holds for binary operators):

$$\text{convert}(Op^{PV}(s)) = Op^{B}(\text{convert}(s)) \blacklozenge$$

Property 6.3 can be trivially extended to consider also our algebraic operators on relations in $r \in$ DB_Evaluators and on the corresponding *pi(r)* and *pd(r)* sets (where *pi(r)* and *pd(r)* can be trivially interpreted as BCDM relations).

Corollary 6.4: Reducibility to BCDM.

The algebraic operators in our approach are reducible to BCDM algebraic operators. ♦

Finally, given property 3.2 of BCDM (i.e., BCDM algebraic operators reduce to relational algebra operators), also corollary 6.5 trivially holds.

Corollary 6.5: Reducibility to relational algebra.

The $BCDM^{PV}$ algebraic operators are reducible to relational algebra operators. ♦

In summary, we have that:

(1) as regards evaluator data only, we adopt the BCDM model (see Section 4.1);

(2) as regards manipulation operations, we provide a consistent extension of the BCDM's ones (see properties 5.3.1 and 5.3.2);

(3) as regards algebraic operators, we provide an extension reducible to the BCDM's ones (see corollary 6.4).

We therefore provide an "upper layer" that is super-imposed to BCDM in order to cope with proposal vetting needs (see also the discussion in Section 9), adding to BCDM the expressiveness to cope with different levels of users, and with alternative proposals (while BCDM and SQL do not provide any support to cope with disjunctions of tuples)

Notice also that, as we recalled in Section 3, BCDM is *structurally equivalent* to a set of representation models [Snodgrass 1995], including TSQL2. In turn, TSQL2 is *upward compatible* with SQL (i.e., all the data structures and legal queries in SQL are contained in TSQL2 syntax; moreover, all statements expressible in SQL evaluate the same results in both models) [Bohlen et al. 1995]. We operate at a semantic level (e.g., we do not provide any query language, so that our approach is not directly comparable with SQL). Nevertheless, given the discussion above, we may say that our $BCDM^{PV}$ model is "consistent" with SQL, in the sense that it provides a consistent extension of its underlying semantics (see also corollary 6.5) to cope with time (as in BCDM) and proposal vetting.

7 Related works

In this paper, we have discussed our extensions to the BCDM model, to cope with proposal vetting in a relational environment, in which both valid time and transaction time need to be supported.

We are not aware of any other approach in the literature coping with such an issue. However, we think that it might be important to compare our approach with other approaches that share at least some of our goals. In particular, in the area of database versioning, many approaches have been devised to face changes to a database, due to the proposals of different versions, evolving in time.

Our approach is related to the database approaches aiming at managing versioning at the data level (i.e., *extension versioning*, using the terminology in [Ozsoyoglu and Snodgrass 1995]). However, some main general differences with respect to such approaches are that:

- they do not extend the relational model because:
  - they operate on object-oriented databases (or implement object-oriented concepts on relational databases), or
  - they operate mostly at the language level (and not at a semantic level, as in our approach), providing high-level constructs to manipulate object versions; as a consequence, they do not provide an algebra to query versions, or
  - they extend conceptual design models such as ER;
- they do not allow an easy transition from existing relational data (see properties 5.3.1 and 5.3.2);
- they do not cope also with valid and transaction time; in fact most approaches in the literature deal either with time or with alternative versions.

Two more specific differences concern the fact that, in such approaches, there is not a direct management of:

- two different levels of data (evaluator vs. proposer data levels) and two different sets of manipulation operations (accept\reject operations vs. insert\delete\update proposal operations);
- mutually exclusive alternatives (i.e., when an alternative is accepted, the related alternatives must be disallowed).

Despite these relevant differences, in the following, we discuss in more detail some specific approaches. Please note that most of these approaches are meant for specific application areas, i.e., CAD and CASE domains; thus, they support features that are not accounted for in our approach, such as composite objects, types\classes of design objects, dynamic configurations and change propagation.

The seminal approach by McLeod et al. [McLeod et al. 1983] is inscribed in the framework of conceptual design data models (however, it describes how to implement it in the relational model). It introduces the concepts of object composition and object version.

The approach by Batory and Kim [Batory and Kim 1985] (later extended in [Chou and Kim 1986]) deals with VLSI CAD and is an ER model. They introduce object-oriented-like inheritance and they deal with versions as objects with the same interface but with different implementations. They describe a mechanism of version timestamps (e.g., "change-notification timestamp", "change approval timestamp") for dealing with change notification. In [Chou and Kim 1986] the approach has been revised to deal also with different levels (types) of versions (i.e., *transient*, *working* and *released*), which loosely resemble our evaluator\proposer levels.

Dittrich and Lorie in [Dittrich and Lorie 1988] introduced an influential approach in CAD domain. They describe the design of a mechanism to support versioning proposing an implementation based on relational databases. Their model comprises *design objects* with some associated versions. In this sense, the design object resembles the Proposal-tuples in our approach. Moreover, Dittrich and Lorie describe a mechanism for identifying the current (reference) version. However, they do not discriminate between accepted and rejected versions, and do not enable users to explicitly propose a deletion of a design object.

[Vines et al. 1988] and [SUN 1988] are two works meant for the CASE domain and for software configuration management. In the first work - unlike the other works in this section and similarly to our approach - a version is identified with timestamps, rather than version identifiers. In the second one, similarly to our approach, versions are organized in levels (i.e., *development*, *integration* and *release*).

[Katz 1990] reviews the various version data models and proposes a comprehensive approach inspired to the ER model.

None of the works presented so far support transaction and valid time. More recently, several *object-oriented* approaches have been proposed for dealing both with versions and with valid and transaction times, such as [Sciore 1994], [Gançarski 1999] and [Moro et al. 2001]. The work of [Rodríguez et al. 1999] supports only versioning and transaction time. In general, a main difference between object-oriented approaches and relational approaches has been pointed-out by Sciore [Sciore 1991, page 425]: "*The relational model has a limited modeling capacity, and so researchers in historical relations have all being forced to extend the relational model in some way. On the other hand, object-oriented models are able to encapsulate the notion of time in classes. Thus there is no need to develop a new historical object-oriented model; what we need is a methodology for using these classes in our existing model*".

Specifically, in [Sciore 1991, 1994], Sciore has proposed an approach coping with transaction and valid time, and alternative versions, while issues such as physical strategies to store versions, change notification and schema evolution are explicitly outside the scope of that approach. Transaction and valid time, and alternative versions are dealt with using the notion of *annotated variables* (roughly speaking, annotated variables in [Sciore, 1991, 1994] are variables whose intension can be addressed by time and\or version indexes), and introducing proper *methods* to access and manipulate them. Separate annotations and methods are defined to cope with the different aspects (valid time, transaction time and

alternative versions), and examples are proposed about their interaction. On the other hand, as in BCDM (and, in general, in all relational approaches to bitemporal data), in our approach valid and transaction time are coped with explicitly and in an integrated way; following such a guideline, through the definition of Proposal-tuple, we give an explicit data model for data in which both bitemporal aspects and alternative version are taken into account. We have also defined new manipulation operations to cope with such a new data model, proving that it is a consistent extension of the BCDM model.

The approach in [Gançarski 1999] is based on the notion of Database Version model (DBV) [Cellary and Jomier 1990]. DBV is a model for dealing with versions in object-oriented databases: a database is a set of database versions, the objects can be versioned and a version of an object can be shared by more database versions, so that version stamps associate object versions with database versions. [Gançarski 1999] implements bitemporal databases using database versions expressed with DBV model. Branching alternatives are expressed using alternative identifiers managed at the application level. The approach has been implemented on top of O_2 object-oriented DBMS. In [Gançarski 1999] the main goal is that of providing minimal extensions to DBV in order to cope with transaction and valid time, but neither the underlying semantics of the interplay of time and alternative versions, nor the formal properties of the extension being built are taken into account.

More recently, another object-oriented data model has been proposed for dealing both with valid and transaction time and with versions, the Temporal Versions Model (TVM) [Moro et al. 2001, 2002, Machado et al. 2006]. TVM, implemented on an object-relational database, supports four version statuses (*working*, *stable*, *consolidated* and *deactivated*), where *working* and *stable* statuses loosely resemble our proposer level, *consolidated* status loosely resembles our evaluator level, and *deactivated* status loosely resembles our rejected proposal status. Time can be associated with objects, attributes and associations, and both transaction and valid times are supported. Although TVM is an object-oriented approach, it is implemented on top of a relational DBMS (i.e., DB2), and supports an SQL-based query language (TVQL). A mapping from examples of TVQL queries to SQL queries is provided [Moro et al. 2002]. However, although in [Machado et al. 2006] an operational semantics of (an extension of) TVQL is given, the treatment of the temporal aspects is not explicitly stated. Last, but not least, no property of being a consistent extension of any previous model is provided.

At least in the recent years [ECDM2006; ECDM2004; ECDM2002], most approaches supporting changes in databases focused on schema evolution and schema versioning. Schema evolution is "*the ability of the database schema to evolve by incorporating changes to its structure without loss of existing data and without significantly affecting the day-to-day operations of the database*" [Shankaranarayanan and Ram 2003], while schema versioning requires "*the maintenance of more than one schema*" [Grandi and Mandreoli 2003] in order "*to support different users\team concurrently working on parallel schema versions*" [ibidem].

Roddick in [Roddick 1995] surveyed the main issues involved with schema versioning and evolution. When changes to the schema are performed, two main problems have to be dealt with: maintaining the consistency of the schema and handling the consistency of data with regard to the modified schema. A comprehensive survey of schema evolution works is [Shankaranarayanan and Ram 2003]. Several approaches have been proposed regarding the various data models: for the relational model (see, e.g., [De Castro et al. 1997]), for the object-oriented model (e.g., [Grandi et al. 2000; Grandi and Mandreoli 2003]) and for conceptual models such as ERM (e.g., [Liu et al. 1993]).

Such approaches seem to us only loosely related with our one, since we operate at the level of data (tuples), not at the level of schema; specifically, our approach aims at managing the change to data values, not the change to the schema. On

the other hand, in the schema versioning approaches, data change is usually not managed as a "primitive" notion, but as a (possibly automatically managed) process induced by changes to the schema.

8 Alternative approaches

As mentioned above, comparisons to related works in the database literature are relatively limited. In fact, to the best of our knowledge, our approach is the only one facing proposal vetting in a temporal relational environment, and operating at the level of data model and algebra (i.e., at the semantic level). On the other hand, we think it is worth mentioning some alternative strategies we have explored to achieve our goals, motivating why we discarded them in favor of the approach we presented in this paper.

(1) We could use a set of temporal relational database versions (each one consisting of a set of BCDM relations) to model the different possible versions of the database, in response to the proposers' operations. In other words, in this solution, instead of maintaining a different level of data to store the proposals of proposers, and executing them only if\when accepted, one could directly execute each proposal, thus leading to a new version of the database, and model acceptance of operations by taking the corresponding version as the reference one. For instance, supposing for the sake of brevity that the database only consists of a relation r, we would initially have a unique version {r} which is the reference one for the evaluator. After a proposal of insertion of a new tuple x_1 in r, we would have two versions {r,r_1} (where r_1 denotes the result of inserting x_1 in r), after a successive proposal of insertion of x_2 we would have four versions {r,r_1,r_2,r_{12}} (where r_{12} is obtained by inserting both x_1 and x_2 in r) and so on. Proposals of update and of deletion could be managed similarly. Evaluating the different proposals therefore could be modeled as the selection of one of the versions by the evaluator. Since each version can be modeled as a standard BCDM database, at a first glance this approach seems very appealing, supporting an easy definition of manipulation and algebraic operations. However, it has several drawbacks. The first three drawbacks concern technical issues related to the expressiveness of the solution, while the fourth one concerns its (spatial) complexity and its implementability:

   a. this approach does not allow evaluators to accept\reject any single proposal in isolation from the others. As a matter of fact, in this approach, the selection of one of the proposal versions as the reference evaluator version involves a *global* evaluation of all the proposal operations, which is usually an impractical and not user-friendly task in practice. In fact, each version V has been obtained by executing a subset O' of the set O of proposed operations. Thus, selecting V corresponds to accepting all operations in O' and rejecting all the others (i.e., O – O');

   b. in this approach acceptance of a proposal does not automatically disallow all its "competing" alternatives (actually, in our approach the acceptance of a proposal involves an implicit rejection of all its alternative proposals). The versions keep no track of whether proposals concern the update of the same tuple. Therefore, there is no direct way of retrieving the mutually exclusive alternatives of an accepted proposal, in order to discard them. Of course, one could store in some dedicated "system table" the information about which tuple is modifying which other, and the connection between update operations and corresponding versions. Given such a piece of information, suitable algorithms could retrieve mutually exclusive alternatives of each proposals. However, the implementation of such a mechanism is not trivial and the notion of what are the alternatives of a tuple is not explicitly managed, so that it is not visible for the user (specifically, we think it is very important, for evaluators,

to have a "localized" picture of what are the alternative proposals of a given piece of information). On the other hand, one of the cue points of our approach is that we give to alternative mutually exclusive proposals the dignity of a "first class" explicit entity (called Proposal-tuple; in this way, for instance, this problem can be easily faced). Notice that an analogous choice has been made by several object-oriented data versioning approaches (see, e.g., the "design objects" in [Dittrich and Lorie 1988]);

c. if we want to retain the history of the database (including all the proposals), we have no way to distinguish between an active and a rejected proposal of deletion;

d. there is a combinatorial explosion of the number of versions (exponential in the number of proposal operations). This would not be a major drawback for an abstract (semantic) approach "per se". However, it is a major drawback if the semantics is regarded as the basis for implementation;

(2) we could use a different temporal relational semantic model, in which standard BCDM is extended with a new dimension, the "version dimension", orthogonal to valid time and to transaction time, to cope with disjunctive (alternative) proposals. In such a way, each tuple could be indexed with the identifier of the version it belongs to. This solution is, at a first glance, very elegant and appealing. In particular, algebraic operators can be easily and clearly defined, and reducibility to the BCDM model can be easily obtained through the introduction of a "version slicing" operator, selecting all and only the proposals concerning a chosen version. There is a spectrum of alternative ways to implement such an approach, mostly depending on the extent to which version identifiers can be managed by proposers. The two extremes of the spectrum are sketched below:

a. versions (and version identifiers) could be automatically managed by the system (e.g., using surrogates). In such a solution, in order to let evaluators to accept\reject any single proposal in isolation from the others , the system would have to generate a new version whenever a new proposal is issued. In such a case, the approach is conceptually close to the approach at point (1) discussed above, having the same limitations we discussed about it;

b. versions (and version identifiers) could be directly managed by proposers. In such a way, we could avoid the combinatorial explosion of the number of versions which affects the above solution. However, this solution is not feasible from the practical point of view, since it involves demanding to the proposers the burden to identify a whole global version to include the proposed update (in other words, the proposers should be aware of all other proposals concerning the whole database, and state explicitly which subset of them must be considered, in addition with the proposer's proposals, as a complete candidate new version of the database. This choice is not practically feasible in most practical contexts);

(3) we could augment the BCDM model by distinguishing between two different types of transaction times (in addition to the valid time, which is orthogonal to them): the time of proposal and the time of acceptance. Initially, this approach seemed to us quite interesting, leading to a "three dimensional" treatment of time which has also a very nice "visual" counterpart. For instance, algebraic operations could be built considering the "visual" perspective (e.g., difference could be modeled, in the case of value-equivalent tuples, by the "spatial" difference between the three-dimensional objects representing their times). However, this very same "visual" perspective convinced us that such an approach was problematic, since certain unexpected and undesired behaviors arose in the definition of algebraic operations (for instance, using the "standard" intersection semantics, the natural join between accepted tuples having intersecting valid and acceptance time would be empty in case their proposal times did not intersect). We recognized that the reason of such a problem was the fact that, actually, proposal time and acceptance time cannot be dealt with as orthogonal temporal dimensions since, indeed, they are not orthogonal at all. Moreover, in principle, this approach

provides the possibility to deal with evaluator tuples and proposals of insertion in the same relation. In fact, the absence\presence of acceptance time could actually distinguish between proposals of insertion and evaluator tuples. However, we realized that this would not apply in the event that proposals of deletion and proposals of updates were to be considered. Nevertheless, if proposals of update have to be contained in dedicated relations, there is no need to have acceptance time for them (in fact, being proposals, their acceptance time would be null; if accepted, their acceptance time would be redundant with respect to the acceptance time of the corresponding evaluator tuple). Finally, in such an approach, there is no explicit notion of mutually exclusive alternative proposals; therefore, disallowing alternative proposals after an acceptance operations would be technically quite complex. Thus, we abandoned such an approach, moving towards the notion of Proposal-tuple and towards the distinction between evaluator relations and sets of (insertion\deletion\update) proposals which characterize our current approach;

(4) finally, a slightly different approach would be to modify the approach in this paper with a different definition of Proposal-tuple, in which (in the case of updates of updates) the dependency between updates (i.e., the relationship between each update proposal and the tuple to be modified) is explicitly stored in a tree. For instance, considering the example in Figure 4 (see Section 4), the "extended" Proposal-tuple coping with the results of steps 4,5,7,8,9,10 could be the one shown in Figure 8.

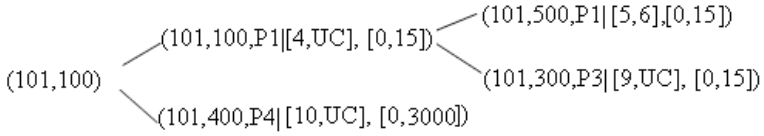

Figure 8: an alternative "tree-like" representation of Proposal-tuples (compare to Figure 4).

With respect to our notion of Proposal-tuple, the "extended" representation is somehow more expressive, since it explicitly captures the notion of "what update depends on what other update" (compare Figure 8 with Figure 4 in Subsection 4.2.1). However,

(1) on the one side, it seems to us that there is no significant expressive gain, and

(2) on the other side, there is a very relevant additional complexity, and a loss of "desirable" properties.

As regards issue (1), notice that the acceptance of an update proposal in the representation in Figure 8 would lead to the very same result, at the evaluator data level, than the corresponding acceptance in the simpler representation we proposed in this paper.

As regards issue (2), we have proved that, using the above tree representation, the property of uniqueness of model (see property 4.3.6) does no longer hold. In other words, one can have different "extended" Proposal-tuples that are snapshot equivalent. This seems to us a relevant drawback, since, in such a context, snapshot equivalence can no more be considered as a synonym of "structural" equivalence. Moreover, the definition of the algebraic operators operating on "extended" Proposal-tuples is far from being clear and intuitive, and we could not provide any definition based on strongly motivating design criteria (such as, e.g., reducibility to BCDM algebraic operators in the approach in this paper).

9 Discussion and future work

In this paper, we have proposed a *semantic* (in the sense discussed in the quotation from [Snodgrass 1995] reported in the introductory section) framework supporting proposal vetting (i.e., proposal and evaluation of update) about data in a relational environment, in which transaction time and (possibly) valid time have to be managed.

The need for this kind of approaches is emerging in many different areas, including workflow, protocol and guideline management, and shared vocabulary development. As a matter of fact, some tools are being developed to support this kind of need in a relational context (consider, e.g., Citizendium [Citizendium], where – however - just transaction time is dealt with). Anyway, to the best of our knowledge, no *theoretical* framework has been designed to define the underlying *semantics* of these tools. In this paper, we have proposed a semantic approach overcoming such a limitation. In particular, our approach guarantees that a set of properties (see Sections 5 and 6) are satisfied by any implementation compliant with it.

For the sake of generality, our approach has been defined at the semantic level, defining a new data model, manipulation operations and algebra. We have based our approach on the "unifying" BCDM semantic model, extending it to support cooperative sessions of work, in which proposals are issued and evaluated. Specifically, the most relevant extensions to the BCDM model are:

(1) the treatment of mutually exclusive alternatives of relational tuples. This phenomenon has been faced with the introduction of the basic notion of Proposal-tuple, which is the core of our approach. Notice that, while the notion of alternative versions of data has been already explored by some database data versioning approaches (based on the object-oriented paradigm – see, e.g., [Sciore 1994]) this notion is, to the best of our knowledge, new in the relational environment, in which relations are usually interpreted as sets (i.e., *conjunctions*) of tuples. The extension to BCDM to cope with *alternative* (and mutually exclusive) tuples has involved substantial changes to BCDM itself at the level of (i) data model, (ii) manipulation language, (iii) algebra;

(2) the treatment of two levels of data (the evaluator and the proposal levels) and of users (evaluators and proposers), each one with its manipulation operations.

Our extensions have been devised in such a way that $BCDM^{PV}$ can be regarded as an *upper layer* built upon BCDM, i.e., we have proved that (i) $BCDM^{PV}$ *data model* and (ii) *algebra* are ***reducible*** to the BCDM one, and that (iii) $BCDM^{PV}$ *manipulation operations* are a ***consistent extension*** to BCDM ones.

By proving properties i-iii, we grant that our approach can be added as a support for update proposal and evaluation on top of any of the temporal relational database approaches grounded on the BCDM semantics. This fact enhances the ***generality*** of our work, as well as its ***implementability***.

As a matter of fact, the above properties make quite easy to devise an implementation strategy for our framework, using a three-layered approach, as graphically shown in Figure 9.

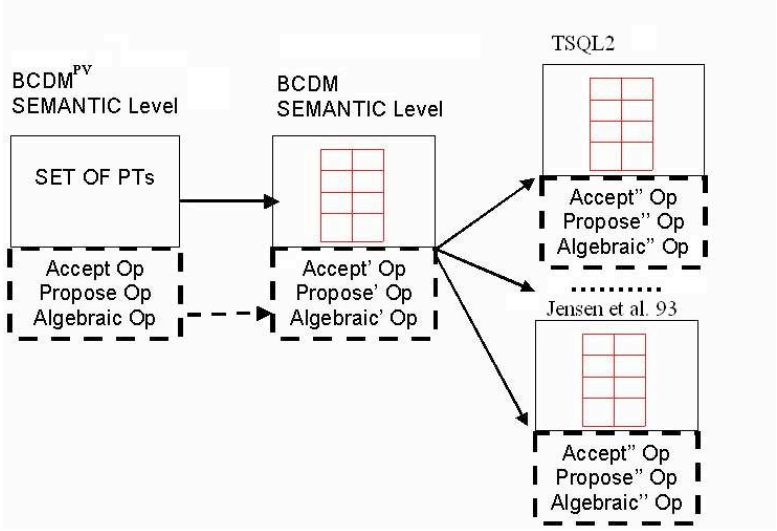

Figure 9: a three layer approach as a strategy to implement our proposal vetting semantic model in practice.

As a first step in this direction, as shown in Figure 9, by applying the convert operator (see Section 6), a set of Proposal-tuples can be converted into a set of tuples in the BCDM model. Observe that these tuples still need to be interpreted as "disjunctions" (while in a standard BCDM relation they would be interpreted as conjunctions): it is the role of the underlying operations (both the manipulation and the algebraic ones, properly converted to the BCDM model as well) to provide the correct interpretation. In other words, to use the object-oriented programming terminology, we could say that the manipulation and algebraic operations, working at the BCDM semantic level, act as methods operating on an object, which is somehow similar, but not identical to a BCDM relation, since its tuples may be in disjunction. By means of these methods, users can correctly manipulate or query the data, having preserved the correct semantics.

Going further, as a second step, such a "BCDM semantic level object", composed by the set of tuples in disjunction, and by its methods, can be easily implemented by relying on an extension of any of the temporal database models supported by the BCDM semantics, such as TSQL2 [Snodgrass 1995], or [Jensen et al. 1993], or on a recent extension of Oracle™ (which, since version 10g, through the Workspace Manager facility, deals with both valid and transaction time, and is consistent with the BCDM semantic level [Snodgrass 2008]). Therefore our approach can be seen as a further layer (as it clearly appears by reading Figure 9 from right to left) which can be added on top of most current approaches to temporal relational databases, to extend them to cope with proposal vetting. We are currently starting to develop a prototypical implementation of our approach, on top of TimeDB [TimeDB] (a prototype implementing – an extension of – TSQL2 on top of Oracle 10g). Moreover, it is worth stressing once again that, since TSQL2 is upward compatible with (i.e., it is a "consistent extension" of) SQL, our approach may also be conceived as a further layer on top of SQL.

Considering the specific application to clinical guidelines discussed in Subsection 2.2, we think that a further *high-level* layer should be added on top of the core framework discussed in this paper, along the lines shown in Figure 9. The data model and algebraic level, at which our approach operates, is suitable for clearly defining the specifications of the work, and for providing the semantic basis of the implementation. Nevertheless, it is not directly usable by proposers and evaluators, to whom we cannot ask to manipulate quite unfamiliar constructs such as Proposal-tuples. A higher-level interface, exploitable to execute SQL-like queries, is thus required. In particular, considering also applications such as the one described in Section 2.2., the high-level interface should also provide users with the possibility of defining "macro-

objects" (whose description may usually involve tuples in different relations), and supporting global operations on them (in the form of a unique transaction). For instance, in the case of the clinical guidelines domain, tuples from several relations are usually needed to model a single clinical action. On the other hand, for the sake of user friendliness, the interface should support the view of such an action as a unitary object for (medical) end-users. The definition of a TSQL2-like query language, and of the notion of "macro-objects", will be object of our future work.

From a more theoretical viewpoint, in the future we also plan to extend our model in order to deal with more than two levels of users. For example, in CASE approaches such as [SUN 1988] three (and more) levels are supported: the level of developers, the level of integrators of modules and the level of supervisors for releasing a final version. In principle, we believe that such extensions should be quite easily supported by our approach, by treating each intermediate level *i* as an evaluator level with respect to level *i+1* and as a proposer level with respect to level *i–1*.

Moreover, we plan to extend our approach by defining more restricted (task\domain-dependent) policies.

Acknowledgements

The authors are very indebted to R.T. Snodgrass and C. Dyreson for many useful comments and suggestions about a preliminary version of the work discussed in this paper. They are also very grateful to F. Grandi, S. Ram, and J. Roddick for their suggestions concerning references to related works. The contribution of G. Molino and M. Torchio, two physicians of Azienda Ospedaliera San Giovanni Battista in Turin, has been essential in order to acquire the guideline we discussed as applicative example in this paper.

The work reported in this paper has been partially supported by a grant by Koiné Sistemi, the software house which is in charge of the development and management of information systems of Azienda Ospedaliera San Giovanni Battista, Turin, Italy.

References

[Bohlen et al. 1995] M.H. Bohlen, C.S. Jensen and R.T. Snodgrass, Evaluating the Completeness of TSQL2. In Clifford, J. and Tuzhilin, A. (eds.), Proceedings of the International Workshop on Temporal Databases, 153-172, Zurich, 1995.

[Batory and Kim 1985] D.S. Batory, W. Kim, Modeling Concepts for VLSI CAD Objects. ACM Trans. Database Syst. 10(3), 322-346, 1985.

[BTS 2003] British Thoracic Society guidelines for the management of suspected acute pulmonary embolism, Thorax 2003, 58, 470-483, 2003.

[Cellary and Jomier 1990] W. Cellary and G. Jomier, Consistency of Versions in Object Oriented Databases. In Proc. 16th VLDB, pages 432-441, 1990.

[Chou and Kim 1986] H.T. Chou, W. Kim, A Unifying Framework for Version Control in a CAD Environment. VLDB 1986, 336-344, 1986.

[Citizendium] http://www.citizendium.org/, Citizendium, a citizens' compendium of everything (URL last accessed on 05/07/2008).

[Conradi and Westfechtel 1998] R Conradi, B Westfechtel, Version Models for Software Configuration Management, ACM Computing Surveys, 30(2), 232:282, 1998.

[CVS] http://www.nongnu.org/cvs/, CVS - Concurrent Versions System (URL last accessed on on 05/07/2008)

[De Castro et al. 1997] C. De Castro, F. Grandi and M.R. Scalas, Schema versioning for multi-temporal relational databases. Information Systems 22(5), 249-290, 1997.

[Dittrich and Lorie 1988] K.R. Dittrich, R.A. Lorie, Version Support for Engineering Database Systems. IEEE Trans. Software Eng. 14(4): 429-437, 1988.

[Doucet et al. 1996] A. Doucet, S. Gancarski, G. Jomier, and S. Monties. Using database versions to implement temporal integrity constraints. In Workshop "Constraints and Databases", Boston (USA), August 1996.

[ECDM 2002] Second International Workshop on Evolution and Change in Data Management, in M. Genero, F. Grandi, W.J. van den Heuvel, J. Krogstie, K. Lyytinen, H.C. Mayr, J. Nelson, A. Olivé, M. Piattini, G. Poels, J.F. Roddick, K. Siau, M. Yoshikawa, E.S.K. Yu (Eds.): Advanced Conceptual Modeling Techniques, ER 2002 Workshops: ECDM, MobIMod, IWCMQ, and eCOMO, Tampere, Finland, October 7-11, 2002, Revised Papers. Lecture Notes in Computer Science 2784 Springer 2003.

[ECDM 2004] Third International Workshop on Evolution and Change in Data Management, in S. Wang, D. Yang, K. Tanaka, F. Grandi, S. Zhou, E.E. Mangina, T. Wang Ling, I.Y. Song, J. Guan, H.C. Mayr (Eds.): Conceptual Modeling for Advanced Application Domains, ER 2004 Workshops CoMoGIS, COMWIM, ECDM, CoMoA, DGOV, and ECOMO, Shanghai, China, November 8-12, 2004, Proceedings. Lecture Notes in Computer Science 3289 Springer 2004.

[ECDM 2006] 4th International Workshop on Evolution and Change in Data Management, in J.F. Roddick, V.R. Benjamins, S. Si-Said Cherfi, R.H.L. Chiang, C. Claramunt, R. Elmasri, F. Grandi, H. Han, M. Hepp, M.D. Lytras, V.B. Misic, G. Poels, I.Y. Song, J. Trujillo, C. Vangenot (Eds.): Advances in Conceptual Modeling - Theory and Practice, ER 2006 Workshops BP-UML, CoMoGIS, COSS, ECDM, OIS, QoIS, SemWAT, Tucson, AZ, USA, November 6-9, 2006, Proceedings. Lecture Notes in Computer Science 4231 Springer 2006.

[Franconi et al. 2000] E. Franconi, F. Grandi, F. Mandreoli, A Semantic Approach for Schema Evolution and Versioning in Object-Oriented Databases, Computational Logic, 2000

[Gançarski 1999] S. Gançarski, Database Versions to Represent Bitemporal Databases. In Proceedings of the 10th international Conference on Database and Expert Systems Applications (August 30 - September 03, 1999). T. J. Bench-Capon, G. Soda, and A. M. Tjoa, Eds. Lecture Notes In Computer Science, vol. 1677. Springer-Verlag, London, 832-841, 1999.

[Grandi and Mandreoli 2003] F. Grandi, F. Mandreoli, A formal model for temporal schema versioning in object-oriented databases. Data Knowl. Eng. 46(2), 123-167, 2003.

[Grandi et al. 2000] F. Grandi, F. Mandreoli and M.R. Scalas, A Generalized modeling framework for schema versioning support. Proceedings of the Australian Database Conference (ADC 2000),Canberra, Australia, Maria Orlowska (Ed.), 33-40, 2000.

[Jensen et al. 1993] C.S. Jensen, L. Mark, N. Roussopoulos and T.Sellis, Using differential techniques to efficiently support transaction time, The VLDB Jornal, 2(1),75-111, 1993.

[Jensen and Snodgrass 1996] C. S. Jensen and R. T. Snodgrass, Semantics of Time-Varying Information, Information Systems, 21(4), 311–352, 1996.

[Jensen and Snodgrass 1999] C. S. Jensen and R. T. Snodgrass. Temporal Data Management. IEEE Transactions on Knowledge and Data Engineering, 11(1): 36--45 (1999).

[Jensen and Snodgrass 2008] C. S. Jensen and R. T. Snodgrass, "Temporal Database", in Encyclopedia of Database Systems, L. Liu and T. Ozsu (editors), Springer, 2008 (in press).

[Johansen 1988] R. Johansen. GroupWare: Computer Support for Business Teams. The Free Press, New York, NY, USA, 1988.

[Katz 1990] R.H. Katz, Towards a Unified Framework for Version Modeling in Engineering Databases. ACM Comput. Surv., 22(4), 375-408, 1990.

[Khatri et al. 2004] V. Khatri, S. Ram and R.T. Snodgrass, Augmenting a Conceptual Model with Geospatiotemporal Annotations, IEEE Transactions on Knowledge and Data Engineering 16(11), 1324-1338, November 2004.

[Liu et al. 1993] C.T. Liu, S.K. Chang and P.K. Chrysanthis, An entity-relationship approach to schema evolution. Proceedings of the International Conference on Computing and Information. Abou-Rabia, 0., Chang, C. K., and Koczkodaj, W. W. (Ed.), 575-578, 1993.

[Liu and Ozsu 2008] L. Liu and T. Ozsu (editors), Encyclopedia of Database Systems, Springer, 2008 (in press).

[MacDonald et al. 2005] W.B.G. Macdonald, A.P. Patrikeos, R.I. Thompson, B.D. Adler, and A.A. van der Schaaf, Diagnosis of pulmonary embolism: Ventilation perfision scintigraphy versus helical computed tomography pulmonary angiography, Australiasian Radiology 49, 32-38, 2005.

[Machado et al. 2006] R. Machado, Á. F. Moreira, R. de Matos Galante & M. M. Moro, Type-safe Versioned Object Query Language; Journal Of Universal Computer Science JUcs, volume 12, issue 7, september 2006, Pages: 938-957, ISSN: 0948-695X.

[McKenzie & Snodgrass 1991] L.E. McKenzie and R.T. Snodgrass, "Evaluation of relational algebras incorporating the time dimension in databases", ACM Computing Surveys 23(4), 501-543, 1991.

[McLeod et al. 1983] D. McLeod, K. Narayanaswamy, K. V. Bapa Rao: An Approach to Information Management for CAD/VLSI Applications. Engineering Design Applications 1983, 39-50, 1983.

[Moro et al. 2002] M.M. Moro, N. Edelweiss, A.P. Zaupa and C.S. Santos, TVQL - Temporal Versioned Query Language. In Proceedings of the 13th international Conference on Database and Expert Systems Applications (September 02 - 06, 2002). A. Hameurlain, R. Cicchetti, and R. Traunmüller, Eds. Lecture Notes In Computer Science, vol. 2453. Springer-Verlag, London, 618-627, 2002.

[Moro et al. 2001] M. M. Moro, S.M. Saggiorato, N. Edelweiss and C.S. Santos, Adding Time to an Object-Oriented Versions Model. In Proceedings of the 12th international Conference on Database and Expert Systems Applications (September 03 - 05, 2001). H. C. Mayr, J. Lazanský, G. Quirchmayr, and P. Vogel, Eds. Lecture Notes In Computer Science, vol. 2113. Springer-Verlag, London, 805-814., 2001.

[Oracle 2005] Oracle Database 10g Workspace Manager Overview. An Oracle White Paper, May 2005 http://www.oracle.com/technology/products/database/workspace_manager/pdf/twp_AppDev_Workspace_Manager_10gR2.pdf (URL last accessed on 05/07/2008)

[Özsoyoğlu and Snodgrass 1995] G. Özsoyoğlu and R. T. Snodgrass, Temporal and Real-Time Databases: A Survey, IEEE Transactions on Knowledge and Data Engineering, 7(4), 513-532, 1995.

[Roddick 1995] J.F. Roddick, A survey of schema versioning issues for database systems, Information and Software Technology, 37(7),383-393, 1995.

[Roddick 1996] J.F. Roddick, A model for schema versioning in temporal database systems" Aust. Comput. Sc. Commun., 18(1), 446-452. 1996.

[Rodríguez et al. 1999] L. Rodríguez, H. Ogata and Y. Yano, TVOO: A Temporal Versioned Object-Oriented data model, Information Sciences, Elsevier, 114, 281-300, 1999.

[Sciore 1991] E. Sciore, Using Annotations to Support Multiple Kinds of Versioning in an Object-Oriented Database System. ACM Trans. Database Syst. 16(3), 417-438, 1991.

[Sciore 1994] E. Sciore, Versioning and Configuration Management in an Object-Oriented Data Model VLDB J. 3(1), 77-106, 1994.

[Shankaranarayanan and Ram 2003] G. Shankaranarayanan and S. Ram (2003). Research issues in database schema evolution - the road not taken. Technical Report 2003-15. University of Arizona.

[Snodgrass 1995] R. T. Snodgrass (Ed.), The TSQL2 Temporal Query Language. Kluwer 1995.

[Snodgrass 2008] http://www.cs.arizona.edu/~rts/sql3.html (URL last accessed on 05/07/2008)

[Snodgrass and Ahn 1986] R.T. Snodgrass and I. Ahn. Temporal Databases. IEEE Computer, pages 35-42, Sept. 1986.

[SUN 1988] SUN MICROSYSTEMS, Introduction to the NSE. SUN Part No. 800-2362-1300 (Mar. 7), 1988.

[Tansel et al. 1994] A. Tansel, J. Cliffor, S. Gadia, S. Jajodia, A. Segev and R.T. Snodgrass (eds), Temporal databases: theory, design and implementation, Benjamin/Cummings, 1994.

[TimeDB] http://www.timeconsult.com/Software/Software.html (URL last accessed on 05/07/2008)

[Vines et al. 1988] P. Vines, D. Vines And T. King. Configuration and change control in GAIA. (Sept.). ACM, New York, 1988.

[Wikipedia] http://www.wikipedia.org, Wikipedia, the free encyclopedia URL last accessed on 05/07/2008).

[Wu et al. 1998] Yu Wu, Sushil Jajodia, and X. Sean Wang. Temporal Database Bibliography Update. In O. Etzion, S. Jajodia and S. Sripada (eds.), Temporal Databases: Research and Practice LNCS 1399, pages 338-366. Springer-Verlag, 1998.

Appendix A Case study: collaborative encyclopedias

In order to illustrate the usefulness of a proposal vetting approach, we introduce a further example considering an emerging phenomenon, namely the cooperative creation of an encyclopedia. Specifically, we consider a simple session of work in which a proposal is further refined, and finally accepted by an evaluator. For instance, such a session of work could be a session with Citizendium [Citizendium]. Citizendium – as Wikipedia – is a collaborative encyclopedia where an entry can be proposed and modified by multiple authors. It improves Wikipedia in the sense that it stresses reliability, so each entry must be approved by an editor[4].

Typically, while the history of proposals is maintained (so that transaction time should be supported), valid times are not accounted for in encyclopedias, so that we limit ourselves to showing transaction times.

Let us consider a simple example of collaborative editing of the entry "Prime number" in Citizendium:

Time 10. Proposer P1 writes (inserts) a definition of the entry;

Time 15. Proposer P2 refines (updates) the definition;

Time 20. Evaluator E1 accepts the proposal issued at time 15.

In Figure 10 we depict a graphical representation of the example, in Figure 11 we show how the work session is supported in our approach, and in Figure 12 we represent the content of the data structures in our model.

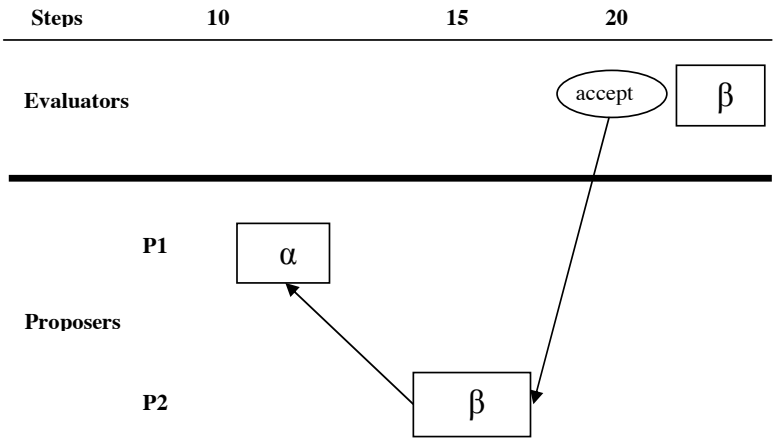

Figure 10: High-level description of a session of work adding a new entry in an encyclopedia. α stands for "*A 'prime number' is a whole number (i.e., one having no fractional or decimal part) that cannot be evenly divided by any numbers but 1 and itself.*"; β stands for "*A 'prime number' is a number that can be evenly divided by exactly two positive whole numbers, namely 1 and itself.*".

[4] Citizendium actually relies on a relational non-temporal DBMS, and it copes with versioning issues at the application level, in an ad-hoc way. In this paper we provide a principled data model underlying the behavior of applications supporting collaborative encyclopedias such as Citizendium and so on.

propose_insert(PAGE, (“Prime number”, α, P1))

propose_update(PAGE, <(“Prime number”, α), (“Prime number”, α, P1)>, (“Prime number”, β, P2))

accept_update(PAGE, <(“Prime number”, α), (“Prime number”, β, P2)>, E1)

Figure 11: Operations for implementing the work session depicted in Figure 10.

Time 15

Evaluator level	**PAGE** ∅
Proposer level	**pi(PAGE)** (Prime number, α, P1 | [10, UC]) **pu(PAGE)** (Prime number, α) ——— (Prime number, β, P2 | [15, UC])

Time 20

Evaluator level	**PAGE** (Prime number, β | [20, UC])
Proposer level	**pi(PAGE)** (Prime number, α, P1 | [10, 19]) **pu(PAGE)** (Prime number, α) ——— (Prime number, β, P2 | [15, 19])

Figure 12: Content of data structures in our model for the work session of collaborative editing at time 15 (upper table) and at time 20 (lower table).

Appendix B. Manipulation operations

In this Appendix, we describe the basic manipulation operations we provide for proposers and evaluators. Direct insertion and deletion operations for evaluators (admitted in Policy 1 – see Section 5) are not described, since they are the same as in BCDM.

Given a relation $r \in$DB_Evaluators, defined on the schema $R=(A_1,\ldots,A_n|T)$, proposals of insertion can be used (by proposers) to propose the insertion of a currently unrecorded tuple *t* into *r*. The result of such an operation is, if admissibility conditions hold, the insertion of *t* into the set *pi(r)*.

The arguments of the *propose_insert* operation are the relation r, and the valid-time tuple $(a_1,\ldots,a_n,p_{new}|t_{vt\_new})$ to be inserted (where t_{vt_new} is a valid time).

The admissibility of a proposal of insertion is defined as follows (where A stands for $A_1,\ldots,A_n$):

Definition B.1: admissible_propose_insert(propose_insert(r, $(a_1,\ldots,a_n,p_{new}| t_{vt\_new})$):

(1) $\neg$ ($\exists$ x $\in$ r : x[A] = $(a_1,\ldots,a_n)$ $\wedge$ current(x))

(2) $\wedge \neg$ ($\exists$ y $\in$ pi(r) : y[A] = $(a_1,\ldots,a_n)$ $\wedge$ current(y))

(3) $\wedge$ $p_{new} \in$ Proposers♦

A proposal of insertion is admissible if the conjunction of three conditions holds:

(1) $(a_1,\ldots,a_n)$ does not identify a current tuple $x \in r$;

(2) $(a_1,\ldots,a_n)$ does not identify a current tuple $y \in pi(r)$;

(3) p_{new} is a proposer.

Notice that conditions in (1) and (2) are used to avoid that one proposes an insertion of a new tuple which is a current one in *r*, or which is (weakly) value equivalent to a current one in *pi(r)* (see Conditions 4.2.1.2 and 4.2.1.3). Accordingly with the BCDM policy, in this case we enforce that such an operation can only be performed as an *update* to (the temporal part of) the existing tuple.

We can now define our propose_insert operator (where A stands for $A_1,\ldots,A_n$):

Definition B.2: propose_insert(r, $(a_1,\ldots,a_n,p_{new}|t_{vt\_new})$):

if admissible_propose_insert(propose_insert(r, $(a_1,\ldots,a_n,p_{new}|t_{vt\_new})$)

then

begin

(1) **if** ($\neg$ $\exists$z $\in$pi(r) : (z[A]=$(a_1,\ldots,a_n)$ $\wedge$ z[P]=p_{new}))

then pi(r) $\leftarrow$ pi(r) $\cup$ {$(a_1,\ldots,a_n,p_{new}|\{UC\}\times t_{vt\_new})$}

(2) **else if** ($\exists$ z $\in$pi(r) : (z[A]=$(a_1,\ldots,a_n)$ $\wedge$ z[P]=p_{new}))

then pi(r) $\leftarrow$ pi(r) – {z} $\cup$ {$(a_1,\ldots,a_n,p_{new}|$ z[T] $\cup$ {{UC}$\times t_{vt\_new}$ })}

end♦

Two cases can be distinguished:

(1) p_{new} has not previously made any insertion proposal value equivalent to $(a_1,\ldots,a_n)$. In such a case, a new tuple is inserted in *pi(r);*

(2) p_{new} has previously made an insertion proposal value equivalent to $(a_1,\ldots,a_n)$. In such a case, the time of the old proposal of insertion is properly updated.

Notice that the conjunction of the admissibility conditions in admissible_propose_insert with the condition (2) implies the fact that only a value-equivalent proposal which is *not* current can be found in case (2) of propose_insert. This means that the old proposal has been already evaluated. In such a case, we still admit the possibility that the old proposer (or another one) makes the same proposal again. Finally, notice that the action in case (2) (i.e., the update of the timestamp of the old proposal) enforces the constraint that value-equivalent tuples with the same proposer cannot coexist in the same set of proposals of insertion (see Condition 4.2.1.2).

Given a relation $r \in$DB_Evaluators, defined on the schema $R=(A_1,\ldots,A_n|T)$, proposals of deletion can be used (by proposers) to propose the deletion of a current tuple *t* from *r*. The result of such an operation is, if admissibility conditions hold, the insertion of *t* into the set *pd(r)*.

The arguments of the *propose_delete* operation are the relation *r*, and the tuple $(a_1,\ldots,a_n,p_{new})$ to be deleted.

The admissibility of a proposal of deletion is defined as follows (where A stands for $A_1,\ldots,A_n$):

Definition B.3 admissible_propose_delete(propose_delete(r, $(a_1,\ldots,a_n,p_{new})$) :

(1) $\exists\, x \in r : (x[A] = (a_1,\ldots,a_n) \wedge \text{current}(x))$

(2) $\wedge \neg \exists\, y \in pd(r) : (y[A] = (a_1,\ldots,a_n) \wedge \text{current}(y))$

(3) $\wedge\, p_{new} \in$ Proposers♦

A proposal of deletion is admissible if a conjunction of three conditions holds:

(1) $(a_1,\ldots,a_n)$ identifies a current tuple $x \in r$;

(2) $(a_1,\ldots,a_n)$ does not identify a current tuple $y \in pd(r)$;

(3) p_{new} is a proposer.

Notice that condition (2) is used to avoid that there is already a *current* proposal of deletion of $(a_1,\ldots,a_n)$ (i.e., that the given proposal is redundant), see Conditions 4.2.2.2 and 4.2.2.3.

We can now define our propose_delete operator (where A stands for $A_1,\ldots,A_n$):

Definition B.4: propose_delete(r, $(a_1,\ldots,a_n,p_{new})$):

if admissible_propose_delete(propose_delete(r, $(a_1,\ldots,a_n,p_{new})$)

then

begin

(1) **if** $(\neg\, \exists z \in pd(r) : (z[A]=(a_1,\ldots,a_n) \wedge z[P]=p_{new}))$

then $pd(r) \leftarrow pd(r) \cup \{(a_1,\ldots,a_n,p_{new} | \{UC\})\}$

(2) **else if** $(\exists\, z \in pd(r) : (z[A]=(a_1,\ldots,a_n) \wedge z[P]=p_{new}))$

then $pd(r) \leftarrow pd(r) - \{z\} \cup \{(a_1,\ldots,a_n,p_{new} | z[T_t] \cup \{UC\})\}$

end♦

Two cases can be distinguished:

(1) p_{new} has not previously made any proposal of deletion of $(a_1,\ldots,a_n)$. In such a case, a new tuple is inserted in *pd(r);*

(2) p_{new} has previously made a proposal of deletion of $(a_1,\ldots,a_n)$. In such a case, the time of the old proposal of deletion is properly updated.

Notice that the conjunction of the admissibility conditions in admissible_propose_delete with the conditions in (2) implies the fact that only a value-equivalent proposal which is *not* current can be found in case (2) of propose_delete. This means that the old proposal has been already evaluated. In such a case, we still admit the possibility that the old proposer (or another one) makes the same proposal again. Finally, notice that the action in case (2) (i.e., the update of the timestamp of the old proposal) enforces the constraint that value-equivalent tuples with the same proposer cannot coexist in the same set of proposals of deletion (see Condition 4.2.2.2).

An acceptance of a proposal of insertion is used by evaluators in order to insert the proposed tuple into the proper DB_Evaluators relation. As a side effect, the acceptance of the insertion proposal must "close" all proposals concerning the same tuple.

Given a relation $r \in$ DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, the arguments of the *accept_insert* routine are *r* itself and the new tuple $(a_1,\ldots,a_n, p_{new})$ to be inserted. Accept_insert is defined as follows (where A stands for $A_1,\ldots,A_n$):

Definition B.5: accept_insert(r, $(a_1,\ldots,a_n, p_{new})$):

(1) **if** $(\exists\ y \in pi(r) : (y[A]=(a_1,\ldots,a_n) \wedge y[P]=p_{new} \wedge current(y)) \wedge$

(2) $(\neg\ \exists\ x \in r : (x[A] = (a_1,\ldots,a_n) \wedge current(x))))$

then

begin

insert^B(r,$(a_1,\ldots,a_n)$, $(\rho^e_{uc}(y))[T_v]$); delete_alternatives(r, $(a_1,\ldots,a_n)$)

end♦

A proposal of insertion can be accepted if the conjunction of two conditions holds:

(1) $(a_1,\ldots,a_n, p_{new})$ is a current proposal in *pi(r);*

(2) *r* does not contain any current tuple value equivalent to $(a_1,\ldots,a_n)$.

Notice that condition (2) is used to avoid to accept an insertion in *r* of a new tuple which is value equivalent to an already existing current one (see Condition 4.1.2). Accordingly with the BCDM policy, we enforce that such an operation can only be performed as an *update* to (the temporal part of) the existing tuple.

If conditions (1) and (2) hold, the BCDM insert^B routine is invoked in order to insert the proposal in *r*, with valid time $(\rho^e_{uc}(y))[T_v]$. Notice also that the BCDM insert^B routine is defined in such a way that no value-equivalent tuples are generated in *r* (specifically, in the case *r* already contains a tuple *x* value equivalent to $(a_1,\ldots,a_n)$ which is not current, the

BCDM insert^B routine properly modifies *x*'s bitemporal timestamp, without introducing any new tuple in *r*). delete_alternatives "closes" all proposals concerning $(a_1,\ldots,a_n)$ (see Definition B.10 in the following).

An acceptance of a proposal of deletion is used by evaluators in order to delete the given tuple from the proper DB_Evaluators relation. As a side effect, the acceptance of a deletion proposal must "close" all proposals concerning the same tuple.

Given a relation $r \in$ DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, the arguments of the *accept_delete* routine are *r* itself and the proposal of deletion of $(a_1,\ldots,a_n, p_{new})$. Accept_delete is defined as follows (where A stands for $A_1,\ldots,A_n$):

Definition B.6: accept_delete(r, $(a_1,\ldots,a_n, p_{new})$):

(1) **if** ($\exists$ y $\in$ pd(r) : (y[A]=$(a_1,\ldots,a_n)$ $\wedge$ y[P]=p_{new} $\wedge$ current(y))

(2) $\wedge$ ($\exists$ x $\in$ r : (x[A] = $(a_1,\ldots,a_n)$ $\wedge$ current(x))))

then

begin

delete^B(r, $(a_1,\ldots,a_n)$); delete_alternatives(r,$(a_1,\ldots,a_n)$)

end♦

A proposal of insertion can be accepted if the conjunction of two conditions holds:

(1) $(a_1,\ldots,a_n, p_{new})$ is a current proposal in *pd(r);*

(2) *r* contains a current tuple equal to $(a_1,\ldots,a_n)$.

If conditions (1) and (2) hold, the BCDM delete^B routine is invoked. Notice that such a routine does not "physically" delete the tuple from *r*, but simply "closes" its transaction time. delete_alternatives (see Definition B.10) "closes" all proposals concerning $(a_1,\ldots,a_n)$.

Rejection operations are used by evaluators to give a negative evaluation to a given proposal. Since we are interested in maintaining the whole history of proposals, rejections do not cause a "physical" deletion of the proposals. In our approach, rejected proposals are simply "closed" (which means that they cannot be executed henceforth, since only *current* proposals can be accepted by evaluators).

Three different rejection operators are provided, operating on proposals of update, insertion and deletion respectively.

Evaluators can reject a proposal of update through the reject_update operation. Given a relation $r \in$ DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, the arguments of the *reject_update* operation are *r* itself and the proposal of update $<(a_1,\ldots,a_n), (a'_1,\ldots,a'_n, p_{new})>$ to be rejected. *reject_update* is defined as follows (where A stands for $A_1,\ldots,A_n$):

Definition B.7: reject_update(r, $<(a_1,\ldots,a_n), (a'_1,\ldots,a'_n, p_{new})>$)

(1) **if** ($\exists$ pt $\in$ pu(r) : (origin(pt)[A]=$(a_1,\ldots,a_n)$

(2) $\wedge \exists$ y $\in$ alternatives(pt) : (y[A] = $(a_1',\ldots,a_n', p_{new})$ $\wedge$ current(y))))

then

begin

pu(r)$\leftarrow$pu(r)$-$ {pt} $\cup$ {create_pt($(a_1,\ldots,a_n)$,alternatives(pt) $-$ {y})

$\cup$ {$(a'_1,\ldots,a'_n, p_{new}|y[T]$-uc_ts(y[T]))}}

end♦

The rejection of an update proposal can be performed if the conjunction of two conditions holds:

(1) $(a_1,\ldots,a_n)$ identifies the origin of a Proposal-tuple $pt \in pu(r)$;

(2) $(a'_1,\ldots,a'_n, p_{new})$ identifies a current alternative in pt.

In such a case, *reject_update* "closes" such a proposal (i.e., it removes the bitemporal chronons with transaction time equal to UC from its timestamp). Notice that the DB_Evaluators relation r is *unaffected* by such a reject operation, which only operates on $pu(r)$.

Evaluators can reject a proposal of insertion through the *reject_insert* operation. Given a relation $r \in$DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, the arguments of the *reject_insert* operation are r itself and the proposal of insertion $(a_1,\ldots,a_n, p_{new})$ to be rejected. *reject_insert* is defined as follows (where A stands for $A_1,\ldots,A_n$):

Definition B.8: reject_insert(r, $(a_1,\ldots,a_n, p_{new})$):

(1) **if** $(\exists\, y \in pi(r) : (y[A] = (a_1,\ldots,a_n, p_{new}) \wedge \text{current}(y)))$

then

begin

delete_alternatives(r, $(a_1,\ldots,a_n, p_{new})$)

end♦

The rejection of an insertion proposal can be performed if $(a_1,\ldots,a_n, p_{new})$ identifies a current proposal of insertion $y \in pi(r)$. In such a case, *reject_insert* "closes" such a proposal, as well as other proposals (if any) concerning $(a_1,\ldots,a_n, p_{new})$, through the delete_alternatives routine. Notice that the DB_Evaluators relation r is *unaffected* by the such a reject operation.

Evaluators can reject a proposal of deletion through the *reject_delete* operation. Given a relation $r \in$DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, the arguments of the *reject_delete* operation are r itself and the proposal of deletion $(a_1,\ldots,a_n, p_{new})$ to be rejected. *reject_delete* is defined as follows (where A stands for $A_1,\ldots,A_n$):

Definition B.9: reject_delete(r, $(a_1,\ldots,a_n, p_{new})$)

(1) **if** $(\exists\, y \in pd(r) : (y[A]=(a_1,\ldots,a_n, p_{new}) \wedge \text{current}(y)))$

then

begin

pd(r)$\leftarrow$pd(r)$-$ $\{y\} \cup \{(a_1,\ldots,a_n,\text{uly}[T]\text{-uc\_ts}(y[T]))\}$

end♦

The rejection of a deletion proposal can be performed if $(a_1,\ldots,a_n, p_{new})$ identifies a current proposal of deletion $y \in pd(r)$. In such a case, *reject_delete* "closes" such a proposal. Once again, notice that the DB_Evaluators relation r is *unaffected* by the such a reject operation.

Finally, notice that several of the above operations adopt the *delete_alternatives* routine. Such a routine takes as an input the origin $(a_1,\ldots,a_n)$ of the Proposal-tuple that has to be "closed", and its corresponding DB_Evaluators relation r. It

"closes" (through an application of the uc_ts function - see Definition 5.1.4) all the alternatives of the Proposal-tuple in *pu(r)* having $(a_1,...,a_n)$ as an origin (step 2). It also "closes" the current proposals of deletion in *pd(r)* (step 3) and of insertion in *pi(r)* regarding $(a_1,...,a_n)$, if they exist (step 4).

Definition B.10: delete_alternatives.

Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, let A stand for $A_1,\ldots,A_n$. We define delete_alternatives as follows:

delete_alternatives(r, $(a_1,...,a_n)$):

(1) pt ← find($(a_1,...,a_n)$,pu(r))
{alt_1, … alt_n}← alternatives(pt)

(2) pu(r) ← pu(r) – {pt} ∪ {create_pt(origin(pt),
{(alt_1[A,P]|alt_1[T]-uc_ts(alt_1[T])),...,(alt_n[A,P]|alt_n[T]- uc_ts(alt_n[T]))})}

(3) **if** (∃ y : (y∈ pd(r)∧ y[A]= $(a_1,...,a_n)$)
then pd(r) ← pd(r) – {y} ∪ {(y[A,P] |y[T]-uc_ts(y[T]))}

(4) **else if** (∃ z : (z∈ pi(r) ∧ z[A]=$(a_1,...,a_n)$)
then pi(r) ← pi(r) – {z} ∪ {(z[A,P] |z[T]-uc_ts(z[T]))}♦

The function *find* is used to get the (unique) Proposal-tuple whose origin is $(a_1,...,a_n)$ from pu(r). Notice that, in the above definition, *(t[A,P]| t[T]-uc_ts(t[T]))* represents the tuple having as atemporal part *t[A,P]*, and as bitemporal chronons the result of deleting from *t[T]* the chronons having UC as transaction time.

Appendix C. Algebraic operators

In the following, we define the basic algebraic operators operating on bitemporal Proposal-tuple-sets. Algebraic operators on proposals of insertion, on proposals of deletion, and on DB_Evaluators relations, are not reported, since they are the same as in BCDM. Also operators working on transaction-time or valid-time Proposal-tuple-sets are not explicitly reported, since they are straightforward special cases of the ones listed below. For the sake of brevity, also "mixed" operators, operating on a Proposal-tuple-set and on a set of BCDM tuples are not described.

Please remember that operations have been defined using the notation introduced in Section 6.

To define our union operator ($\cup^{PV}$) on Proposal-tuple-sets, let $r_1 \in$DB_Evaluators and $r_2 \in$DB_Evaluators be two relations with the same schema R=($A_1,\ldots,A_n$\|T), let A stand for $A_1,\ldots,A_n$, and let $s_1=pu(r_1)$ and $s_2=pu(r_2)$ be two Proposal-tuple-sets.

Definition C.1 $s_1 \cup^{PV} s_2$ = {z :: <origin(z),alternatives(z)> :

(1) **if** ($\exists pt_1 \in s_1$, $\exists pt_2 \in s_2$: (origin(pt_1)= origin(pt_2)))

then

origin(z) ←origin(pt_1);

alternatives(z)={alt :

if ($\exists alt_1 \in$alternatives(pt_1), $\exists alt_2 \in$alternatives(pt_2) : (alt_1[A,P]=alt_2[A,P]))

then alt[A,P]←alt_1[A,P], alt[T]←alt_1[T]∪alt_2[T] ;

else if ($\exists alt_1 \in$alternatives(pt_1) : (¬ $\exists alt_2 \in$alternatives(pt_2) :

(alt_1[A,P]=alt_2[A,P])))

then alt←alt_1;

else if ($\exists alt_2 \in$alternatives(pt_2) : (¬ $\exists alt_1 \in$alternatives(pt_1) :

(alt_1[A,P]=alt_2[A,P])))

then alt←alt_2}

(2) **else if** ($\exists pt_1 \in s_1$: (¬$\exists pt_2 \in s_2$: (origin(pt_1)=origin(pt_2))))

then origin(z)←origin(pt_1);

alternatives(z)←alternatives(pt_1)

(3) **else if** ($\exists pt_2 \in s_2$: (¬$\exists pt_1 \in s_1$: (origin(pt_1)= origin(pt_2))))

then origin(z)←origin(pt_2);

alternatives(z)←alternatives(pt_2)

}♦

The result of our union operator ($\cup^{PV}$) on two Proposal-tuple-sets s_1 and s_2 as above is a new Proposal-tuple-set that contains the proposals which belong to s_1 or to s_2. Three cases can be distinguished:

(1) if s_1 and s_2 contain two Proposal-tuples pt_1 and pt_2 with the same origin, a unique Proposal-tuple *<origin(z),alternatives(z)>* with such an origin must be given as an output. The alternatives of such a new Proposal-tuple are the set of all *alt∈alternatives(z)* obtained as follows: (i) any alternative alt_1 in pt_1 for which there is no value-equivalent alternative in pt_2 is included into *alternatives(z)*; (ii) any alternative alt_2 in pt_2 for

which there is no value-equivalent alternative in pt_1 is included into *alternatives(z)*; (iii) in the case of two value-equivalent alternatives $alt_1 \in alternatives(pt_1)$ and $alt_2 \in alternatives(pt_2)$, a unique alternative is added into *alternatives(z)*, having as an atemporal part and as a proposer the common atemporal part and proposer, and as a bitemporal timestamp the union of the bitemporals of alt_1 and alt_2;

(2) if s_1 contains a Proposal-tuple pt_1 such that there is not any Proposal-tuple in s_2 having the same origin, then pt_1 is part of the result;

(3) if s_2 contains a Proposal-tuple pt_2 such that there is not any Proposal-tuple in s_1 having the same origin, then pt_2 is part of the result.

Notice that case (1) above has been treated in such a way that (i) no Proposal-tuples with the same origin are provided in the result (see Condition 4.2.3.6), and (ii) in no one of the Proposal-tuples in the result there may be two (or more) value-equivalent alternatives (see Condition 4.2.3.2). This grants that the output of our union operator is still a Proposal-tuple-set.

To define our difference operator ($-^{PV}$) on Proposal-tuple-sets, let $r_1 \in$ DB_Evaluators and $r_2 \in$ DB_Evaluators be two relations with the same schema $R=(A_1,\ldots,A_n|T)$, let A stand for $A_1,\ldots,A_n$, and let $s_1=pu(r_1)$ and $s_2=pu(r_2)$ be two Proposal-tuple-sets.

Definition C.2. $s_1 -^{PV} s_2 = \{ z :: \langle origin(z), alternatives(z) \rangle :$

(1) **if** $(\exists pt_1 \in s_1 , \exists pt_2 \in s_2 : (origin(pt_1) = origin(pt_2)))$

then

origin(z)←origin(pt_1);

alternatives(z) = {alt :

if $(\exists alt_1 \in alternatives(pt_1) : (\neg \exists alt_2 \in alternatives(pt_2) :$

$(alt_1[A,P]=alt_2[A,P])))$

then alt←alt_1;

else if $(\exists alt_1 \in alternatives(pt_1)\ \exists alt_2 \in alternatives(pt_2) : (alt_1[A,P]=alt_2[A,P]))$

then alt[A,P]←alt_1[A,P]; alt[T]←alt_1[T]- alt_2[T];

}

(2) **else if** $(\exists pt_1 \in s_1 : (\neg \exists pt_2 \in s_2 : (origin(pt_1)=origin(pt_2))))$

then origin(z)←origin(pt_1);

alternatives(z)←alternatives(pt_1)

}♦

The result of our difference operator ($-^{PV}$) on two Proposal-tuple-sets s_1 and s_2 as above is a new Proposal-tuple-set that contains the proposals which belong to s_1 and not to s_2. Two cases can be distinguished:

(1) if s_1 and s_2 contain two Proposal-tuples pt_1 and pt_2 with the same origin, a unique Proposal-tuple *<origin(z),alternatives(z)>* with such an origin must be given as an output. The alternatives of such a new Proposal-tuple are the set of all $alt \in alternatives(z)$ obtained as follows: (i) any alternative alt_1 in pt_1 for which there is no value-equivalent alternative in pt_2 included into *alternatives(z)*; (ii) in the case of two value-

equivalent alternatives $alt_1 \in alternatives(pt_1)$ and $alt_2 \in alternatives(pt_2)$, a unique alternative is added into *alternatives(z)*, having as an atemporal part and as a proposer the common atemporal part and proposer, and as a bitemporal timestamp the difference of bitemporal timestamps of alt_1 and alt_2 (notice that such an alternative is included in the result only in case $alt_1[T] - alt_2[T] \neq \emptyset$; Proposal-tuples with no alternatives are excluded by definition);

(2) if s_1 contains a Proposal-tuple pt_1 such that there is not any Proposal-tuple in s_2 having the same origin, then pt_1 is part of the result.

Notice that case (1) above has been treated in such a way that (i) no Proposal-tuples with the same origin are provided in the result (see Condition 4.2.3.6), and (ii) in none of the Proposal-tuples in the result there may be two (or more) value-equivalent alternatives (see Condition 4.2.3.2). This grants that the output of our difference operator is still a Proposal-tuple-set.

The definition of our selection operator on Proposal-tuple-sets requires a brief preliminary discussion. Given a relation r∈DB_Evaluators with schema $R=(A_1,\ldots,A_n|T)$, the schema of the corresponding Proposal-tuple-set is $<(A_1,\ldots,A_n),(A_1,\ldots,A_n,P|T)>$. Since atemporal attributes appear twice in the schema, we need a way of specifying, for each condition referring to an attribute A_i, if it concerns the values of A_i in the origin or in the alternatives part of the Proposal-tuple[5]. In the following, we assume that selection conditions can be split into two components (each of which may be empty): P_o (conditions concerning the values of the attributes in the origin) and P_a (conditions concerning the values of the attributes in the alternatives).

To define our selection operator ($\sigma^{PV}_{Po,Pa}$) on Proposal-tuple-sets, let $r \in$ DB_Evaluators be a relation with schema $R=(A_1,\ldots,A_n|T)$, and let $s=pu(r)$ a Proposal-tuple-set.

Definition C.3 $\sigma^{PV}_{Po,Pa}(s) = \{z :: <origin(z),alternatives(z)> :$

if $\exists$ pt∈s : (P_o(origin(pt) ∧ ∃alt∈alternatives(pt) : P_a(alt))

then origin(z)←origin(pt);

alternatives(z)={alt : alt ∈alternatives(pt) ∧ P_a(alt)}}♦

The result of our selection operator ($\sigma^{PV}_{Po,Pa}$) on a Proposal-tuple-set *s* as above is a new Proposal-tuple-set that contains the Proposal-tuples <origin(z),alternatives(z)> obtained by selecting only the alternatives *a* that satisfy the condition P_a from the Proposal-tuples *pt* whose origin satisfies the condition P_o. Notice that either P_o or P_a may be empty (so that no selection condition is imposed on the origin/alternative part of the Proposal-tuples), and Proposal-tuples whose origin satisfies P_o but such that none of their alternatives satisfies P_a are not part of the result.

To define our projection operator (π^{PV}_{D}) on Proposal-tuple-sets, let $r \in$ DB_Evaluators be a relation with schema $R=(A_1,\ldots,A_n|T)$, let $D \subseteq (A_1,\ldots,A_n)$ a subset of the atemporal attributes, and let $s=pu(r)$ be a Proposal-tuple-set.

5 This can be easily done, e.g., by prefixing a keyword (e.g., *origin* vs *alternatives*) to the attribute (e.g., the condition origin.A_i=10 regards attribute A_i in the origin).

Definition C.4 $\pi^{PV}_{D}(s) = \{z :: \langle origin(z), alternatives(z)\rangle :$

(1) **if** $\exists!\ pt_1,..,pt_k \in s\ (k \geq 1) : (origin(pt_1)[D]=\ldots= origin(pt_k)[D])$

then $origin(z) \leftarrow origin(pt_1)[D]$;

$alternatives(z)=\{alt :$

(2) **else if** $\exists!\ alt_1,..,alt_m \in alternatives(pt_1) \cup \ldots \cup alternatives(pt_k)\ (m \geq 1):$

$(alt_1[D]=\ldots=alt_m[D])$

then $alt[D,P] \leftarrow alt_1[D,P];\ alt[T] \leftarrow alt_1[T] \cup \ldots \cup alt_m[T])\}$

$\}\blacklozenge$

Given the above definition, the result of our projection operator $\pi^{PV}_{D}(s)$ is a Proposal-tuple-set with schema <(D),(D,P|T)>, whose Proposal-tuples only contain the values for the attributes in *D* (plus the proposer and the temporal attributes). In our data model we do admit neither (i) Proposal-tuples with the same origin in the same Proposal-tuple-set, nor (ii) value-equivalent alternatives within the same Proposal-tuple. Therefore, in the definition of our projection operator, we had to pay attention to all the cases when, considering only the *D* part of the data, such cases could arise.

The condition (1) is used to partition the set of Proposal-tuples in *s* into maximal subsets $\{pt_1,..,pt_k\}$ containing *all and only* the Proposal-tuples which, considering only the values of the attributes in *D* (e.g., $origin(pt_1)[D]$ denotes the values of the attributes in *D* for the *origin* of the Proposal-tuple pt_1), have the same origin[6]. The output Proposal-tuple-set contains exactly one Proposal-tuple $\langle origin(z), alternatives(z)\rangle$ for each one of such subsets, having as origin the "common" origin. For the sake of simplicity, let us focus on just one of these maximal subsets, say $\{pt_1,..,pt_k\}$. In a first approximation, the set *alternatives(z)* for $\{pt_1,..,pt_k\}$ should contain all the alternatives in $\{alternatives(pt_1) \cup \ldots \cup alternatives(pt_k)\}$, properly "restricted" to the *D* attributes. However, once again, we need to avoid value equivalences. Therefore, condition (2) is used to partition into maximal "value-equivalent" subsets the alternatives concerning Proposal-tuples in the same maximal subset. Finally, notice that, in the case of alternatives "to be merged", the union of bitemporal chronons is performed.

6 Notice that *k* may also assume value 1, for certain values of $origin(pt)[D]$.

Appendix D. Proofs

Property 4.3.6: Uniqueness of model on Proposal-tuple-sets.

Two Proposal-tuple-sets defined over the same schema are snapshot equivalent if and only if they are identical. ♦

Proof: If two Proposal-tuple-sets are identical, since Proposal-tuple-sets cannot include duplicates, every Proposal-tuple in the first Proposal-tuple-set is identical to exactly one Proposal-tuple in the second Proposal-tuple-set. Let us denote two corresponding Proposal-tuples as pt = <o, Alt(alt_1,…,alt_n)> and pt' = <o', Alt(alt'_1,…,alt'_n)>. Since pt=pt', we have that o=o' and for each i,=1,..,n, alt_i=alt'_i. Since o=o', and origins are made only by atemporal attributes, the two origins are necessarily snapshot equivalent. As regards the alternatives, since alt_i=alt'_i, the two alternatives in each pair of snapshots produced by the application of the timeslice operators 4.3.1 and 4.3.2 on them are identical. The timeslice operators 4.3.3 and 4.3.4 on alternatives simply provide the set of all these alternative snapshots. The two sets are obviously identical. Since all corresponding Proposal-tuple pairs are snapshot equivalent, the two Proposal-tuple-sets are snapshot equivalent.

In the other direction, if two Proposal-tuple-sets are snapshot equivalent, for every Proposal-tuple pt = <o, Alt(alt_1,…,alt_n)> in the first set there is a set of Proposal-tuples {pt_1',..,pt_k'} in the second set, with pt_i'= <o_i', Alt(alt'_1,…,alt_i'$_m$)>, such that (1) for all the transaction times T_1 not exceeding the current time and for all the valid times T_2 (henceforth, $\forall T_1 \forall T_2$) $\tau^{PVv}_{T2}(\rho^{PV}_{T1}\{pt\}) = \tau^{PVv}_{T2}(\rho^{PV}_{T1}\{pt_1',\ldots,pt_k'\})$.

We first aim at showing that

(A) $pt_1' = \ldots = pt_k'$.

By Definitions 4.3.3 and 4.3.4

(2) $\tau^{PVv}_{T2}(\rho^{PV}_{T1}\{pt\}) = \tau^{PTv}_{T2}(\rho^{PT}_{T1}(pt))$ and

(3) $\tau^{PVv}_{T2}(\rho^{PV}_{T1}\{pt_1',\ldots,pt_k'\}) = \{\tau^{PTv}_{T2}(\rho^{PT}_{T1}(pt_1')),\ldots,\tau^{PTv}_{T2}(\rho^{PT}_{T1}(pt_k'))\}$.

Therefore, substituting (2) and (3) in (1) we have

(4) $\forall T_1 \forall T_2\, \tau^{PTv}_{T2}(\rho^{PT}_{T1}(pt)) = \{\tau^{PTv}_{T2}(\rho^{PT}_{T1}(pt_1')),\ldots,\tau^{PTv}_{T2}(\rho^{PT}_{T1}(pt_k'))\}$ where, by Definitions 4.3.3 and 4.3.4,

(5) $\tau^{PTv}_{T2}(\rho^{PT}_{T1}(pt)) = <o, Alt(\tau^{ev}_{T2}(\rho^{e}_{T1}(alt_1)),\ldots,\tau^{ev}_{T2}(\rho^{e}_{T1}(alt_n)))>$ and,

(6) for every 1<=i<=k

$\tau^{PTv}_{T2}(\rho^{PT}_{T1}(pt_i')) = <o_i', Alt(\tau^{ev}_{T2}(\rho^{e}_{T1}(alt_i'_1)),\ldots,\tau^{ev}_{T2}(\rho^{e}_{T1}(alt_i'_m)))>$.

Let us consider just origin components. To have the equality in (4), given (5) and (6), we must have {o}={o_1',…,o_k'}, so that o= o_1'=…= o_k'. Henceforth we denote with o' the o_i' (since o_1'=…= o_k').

Note that origins uniquely identify Proposal-tuples (see Definition 4.2.3.1), thus

(7) pt_1'=..=pt_k' which proves (A).

Henceforth we denote with pt' the pt_i' (since pt_1'=..=pt_k').

Secondly, we now need to show that:

(B) pt = <o, Alt(alt_1,…,alt_n)> = pt' = <o', Alt(alt'_1,…,alt'_m)> where

(8) o=o'

From (6), having proved (A), we can easily deduce that

(9) $\forall T_1 \forall T_2\ \tau^{PTv}{}_{T2}(\rho^{PT}{}_{T1}(pt')) = \langle o', Alt(\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt'_1)),\ldots,\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt'_m)))\rangle$.

By substituting (5) and (9) in (4), we thus have

(10) $\forall T_1 \forall T_2 \langle o, Alt(\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt_1)),\ldots,\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt_n)))\rangle =$
$\langle o', Alt(\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt'_1)),\ldots,\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt'_m)))\rangle$
Given (8), this implies
(11) $\forall T_1 \forall T_2 \{\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt_1)),\ldots,\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt_n))\} =$
$\{\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt'_1)),\ldots,\tau^{ev}{}_{T2}(\rho^{e}{}_{T1}(alt'_m))\}$
By Definitions 4.3.1 and 4.3.2, (11) corresponds to (12) below
(12) $\forall T_1 \forall T_2\ \tau^{Bv}{}_{T2}(\rho^{B}{}_{T1}(\{alt_1,\ldots,alt_n\})) = \tau^{Bv}{}_{T2}(\rho^{B}{}_{T1}(\{alt'_1,\ldots,alt'_m\}))$
where $\rho^{B}{}_{T1}$ and $\tau^{Bv}{}_{T2}$ are the BCDM's transaction- and valid-timeslice operators [Snodgrass 1995].
Notice that, by hypothesis, $\{alt_1,\ldots,alt_n\}$ are alternatives of the same Proposal-tuple. Thus, from Condition 4.2.3.2, $alt_1,\ldots,alt_n$ are bitemporal not-value-equivalent tuples, so that $\{alt_1,\ldots,alt_n\}$ is a set of BCDM tuples (the same holds for $\{alt'_1,\ldots,alt'_m\}$).
Therefore, (12) states that the two sets of tuples $\{alt_1,\ldots,alt_n\}$ and $\{alt'_1,\ldots,alt'_m\}$ are snapshot equivalent in the BCDM approach.
Given property 3.1, snapshot equivalence of BCDM sets of tuples (i.e., of BCDM relations) implies that the sets are identical, so that,
(13) $\{alt_1,\ldots,alt_n\} = \{alt'_1,\ldots,alt'_m\}$
From (13) and (8), we can conclude pt=pt', which proves (B).

In conclusion, every Proposal-tuple pt in the first Proposal-tuple-set has exactly one exact match pt' in the second Proposal-tuple-set.
By the symmetrical argument, each Proposal-tuple pt' in the second set has an exact match pt in the first set, and the two Proposal-tuple-sets are consequently identical. ♦

Corollary 4.3.7 In our data model, identity and snapshot equivalence coincide, i.e., two databases over the same evaluator schema in our model are identical if and only if the corresponding evaluator relations and proposal sets are snapshot equivalent. ♦

Proof: The proof follows:

1. from the similar property in the BCDM model [Snodgrass 1995], as regards relations $r_i \in$ DB_Evaluators, $pi(r_i)$ and the $pd(r_i)$;
2. from property 4.3.6 for the sets $pu(r_i)$.♦

Property 5.3.1: Consistent extension of BCDM.

Under Policy 1 our manipulation language is a consistent extension of the BCDM model.♦

Proof. Under Policy 1, evaluators can perform BCDM operations of insertion and deletion of tuples in relations in BD_Evaluators. Therefore, our approach is trivially a consistent extension of the BCDM one, supposing that all BCDM users are included in the Evaluators set (and that only DB_Evaluators data are taken into account). ♦

Property 5.3.2: "Proposal vetting" consistent extension of BCDM.

Under Policy 2, considering that all users are both evaluators and proposers, our model in which each manipulation operation Op^B is executed as an atomic pair of operations <propose_Op;accept_Op> is a *"proposal vetting" consistent extension* of the BCDM model (considering only data in DB_Evaluators). ♦

The operations Op defined in BCDM [Snodgrass 1995] are the $insert^B$ and the $delete^B$ operations. Regarding the update operation, in [Snodgrass 1995] there is not a definition of an update operation; in fact Snodgrass in [Snodgrass 1995] defines only the modify operation as a sequence of $delete^B$ and $insert^B$; moreover, such a modify operation is also very limited, since it is used only in order to change the valid time of a bitemporal tuple. That is why we prove the Property in two steps. First we consider the operations of $insert^B$ and $delete^B$ only. Secondly we generalize the BCDM modify operation introducing an $update^B(r, (a_1,\ldots,a_n), (a''_1,\ldots,a''_n), t_{vt\_new})$ operation that allows one to modify also the atemporal values in the BCDM model, and we prove the property also with respect to the $update^B$ operation.

In this proof, we suppose that the operations are applied on "equivalent" relations in the two models, in the sense that, since BCDM does not support proposals, also in our approach there are no current proposals, i.e., all previous proposals have been accepted or rejected.

More formally, for any relation $r \in$ DB_Evaluators, we suppose that the following conditions hold:

Assumptions D.1.

a. $\forall t \in pi(r)\ \neg current(t)$, and

b. $\forall t \in pd(r)\ UC \notin t[T_t]$, and

c. $\forall pt \in pu(r)\ \forall alt_i \in alternatives(pt)\ \neg current(alt_i)$;

d. moreover, we assume that all users are both proposers and evaluators, so that both the proposal-level and evaluator-level operations are allowed.

Proof.

insert.

Let us consider the case of the sequence of operations <propose_insert; accept_insert> in our approach. We prove that it is equivalent (as regards the DB_Evaluators relation) to an $insert^B$ operation in the BCDM model, i.e.:

$$<propose\_insert(r, (a_1,\ldots,a_n,p_{new}|t_{vt\_new})\ );\ accept\_insert(r, (a_1,\ldots,a_n,p_{new}))>$$

is equivalent to

$$insert^B(r, (a_1,\ldots,a_n), t_{vt\_new})$$

For the sake of commodity, we recall here the definition of $insert^B$ from [Snodgrass 1995] (adapting the notation to our one):

$insert^B(r, (a_1,\ldots,a_n), t_{vt\_new})$:

(1) **if** $(\neg\exists t_b: (a_1,\ldots,a_n \mid t_b) \in r)$

then $r \leftarrow r \cup \{(a_1,\ldots,a_n \mid \{UC\} \times t_{vt\_new})$

(2) **else if** ($\exists t_b$: $((a_1,\ldots,a_n \mid t_b) \in r \wedge \neg current(r))$)

then $r \leftarrow r - \{(a_1,\ldots,a_n \mid t_b)\} \cup \{(a_1,\ldots,a_n \mid t_b \cup \{\{UC\} \times t_{vt\_new}\})\}$

Let us consider Definition B.1 (admissible_propose_insert). Under the assumptions D.1, it is possible to write the Definition B.1 in a simpler form. We report in Definition B.1' the definition of admissible_propose_insert striking through the parts trivially true:

Definition B.1': admissible_propose_insert(propose_insert(r, $(a_1,\ldots,a_n,p_{new} \mid t_{vt\_new})$):

(1) $\neg (\exists x \in r : x[A] = (a_1,\ldots,a_n) \wedge current(x))$

(2) ~~$\wedge \neg (\exists y \in pi(r) : y[A] = (a_1,\ldots,a_n) \wedge current(y))$~~

(3) ~~$\wedge p_{new} \in Proposers$~~ ♦

In fact, a propose_insert is admissible if (1), (2) and (3) are true; i.e.:

(1) $(a_1,\ldots,a_n)$ does not identify a current tuple $x \in r$

(2) since we assume that all alternatives in pi(r) are not current, row (2) is trivially true (see Assumptions D.1a-D.1c);

(3) since we assume that all users are both proposers and evaluators, row (3) is true (see Assumption D.1d).

We have two cases:

a) there exists in r a current tuple value equivalent to the tuple to be inserted (i.e., $\exists x \in r : x[A] = (a_1,\ldots,a_n) \wedge current(x)$);

b) there not exists in r a current tuple value equivalent to the tuple to be inserted.

In case a), the propose_insert(r, $(a_1,\ldots,a_n,p_{new} \mid t_{vt\_new})$) operation is not admissible; in fact, condition (1) in Definition B.1' is false. Also the subsequent accept_insert operation has no effect because there are no current proposals to be accepted.

In this case (and only in this case), also the $insert^B$ operation does not have any effect, because both conditions in rows (1) and (2) of $insert^B$ are false.

In case b), the propose_insert(r, $(a_1,\ldots,a_n,p_{new} \mid t_{vt\_new})$) operation is admissible, because condition (1) in Definition B.1' is true. Then, after the propose_insert operation is executed, we have in pi(r) a tuple $(a_1,\ldots,a_n,p_{new} \mid t_b)$ such that $(UC, t_{vt\_new}) \in t_b$.

Now let us consider the accept_insert operation (Definition B.5). Condition (1) is true because of the propose_insert operation (see above) and also condition (2) is true since, in case b), the conditions of admissible_propose_insert are satisfied. Then, an $insert^B(r,(a_1,\ldots,a_n), \rho^e_{uc}(y)[T_v])$ operation is executed, which is equivalent to an $insert^B(r, (a_1,\ldots,a_n), t_{vt\_new})$ operation since y is the tuple inserted by the propose_insert operation and $\rho^e_{uc}(y)[T_v] = t_{vt\_new}$.

Moreover, the delete_alternatives(r,$(a_1,\ldots,a_n, p_{new})$) operation "closes" in pi(r) the tuple inserted by the propose_insert operation, in such a way that also for the subsequent operations we can assume that there are no current proposals.

delete.

The case of deletion is analogous. Let us consider the case of the sequence of operations <propose_delete; accept_delete> in our approach. We prove that it is equivalent (with regards to the DB_Evaluators relation) to a delete^B operation in the BCDM model, i.e.:

<propose_delete(r, $(a_1,\ldots,a_n,p_{new})$); accept_delete(r, $(a_1,\ldots,a_{n,}\, p_{new})$)>

is equivalent to

delete^B(r, $(a_1,\ldots,a_n)$)

For the sake of commodity, we recall here the definition of delete^B from [Snodgrass 1995] (adapting the notation to ours):

delete^B(r, $(a_1,\ldots,a_n)$)

if $(\exists t_b: (a_1,\ldots,a_n \mid t_b) \in r)$

then $r \leftarrow r - \{(a_1,\ldots,a_n \mid t_b)\} \cup \{(a_1,\ldots,a_n \mid t_b - uc\_ts(t_b))\}$

Let us consider Definition B.3 (admissible_propose_delete). Under the Assumptions D.1, it is possible to write it in a simpler form. We report in Definition B.3' the definition of admissible_propose_delete striking through the parts trivially true:

Definition B.3' admissible_propose_delete(propose_delete(r, $(a_1,\ldots,a_n,p_{new})$)

(1) $\exists\, x \in r : (x[A] = (a_1,\ldots,a_n) \ \wedge\ current(x))$

(2) ~~$\wedge\ \neg\ \exists\, y \in pd(r) : (y[A] = (a_1,\ldots,a_n) \ \wedge\ current(y))$~~

(3) ~~$\wedge\ p_{new} \in$ Proposers~~♦

In fact, a propose_delete is admissible if (1), (2) and (3) are true, i.e.,:

(1) $(a_1,\ldots,a_n)$ does identify a current tuple $x \in r$;

(2) row (2) is trivially true because we assume that all alternatives in pd(r) are not current (see Assumptions D.1a-D.1c);

(3) row (3) is true because we assume that all users are both proposers and evaluators (see Assumption D.1d).

We have two cases:

a) there is not in r a current tuple value equivalent to the tuple to be deleted (i.e., $\neg\exists\, x \in r : (x[A] = (a_1,\ldots,a_n) \ \wedge$ current(x)));

b) there exists in r a current tuple value equivalent to the tuple to be deleted.

Let us consider case a). The propose_delete(r, $(a_1,\ldots,a_n,p_{new})$) operation is not admissible, in fact, condition (1) in Definition B.3' is false. Also the subsequent accept_delete operation has no effect because there are no current proposals to be accepted.

In this case (and only in this case), also the delete^B operation does not have any effect, because either

(i) the condition of delete^B is false (i.e., there is not any tuple in r value equivalent to the one to be deleted), or

(ii) the assignment in delete^B has no effect (i.e., there is a tuple in r value equivalent to the one to be deleted but it is not current, so that uc_ts(t_b)=∅).

Let us consider case b). The propose_delete(r, (a_1,…,a_n,p_{new}) operation is admissible, because condition (1) in Definition B.3' is true. Then, after the operation is executed, we have in pd(r) a tuple (a_1,…,a_n,p_{new}| t_t) such that UC ∈ t_t.

Now let us consider the accept_delete operation (Definition B.6). The first conjunct in row (1) is true because of the propose_delete operation (see above) and also condition (2) is true because, in case b), the conditions of admissible_propose_delete are satisfied. Then, a delete^B(r, (a_1,…,a_n)) operation is executed.

The delete_alternatives(r,(a_1,…,a_n, p_{new})) operation "closes" in pd(r) the tuple inserted by the propose_delete operation, in such a way that also for the subsequent operations we can assume that there are no current proposals.

update.

As stated above, in this part of the proof we generalize the BCDM modify operation and we define the update^B(r, (a_1,…,a_n,), (a"$_1$,…,a"$_n$), t_{vt_new}) operation that allows one to modify also the atemporal values in the BCDM model.

update^B(r, (a_1,…,a_n,), (a"$_1$,…,a"$_n$), t_{vt_new}) is a sequence of delete^B and insert^B executed as an atomic operation, i.e., if the delete^B or the insert^B operation fail, the update^B operation has no effect.

update^B(r, (a_1,…,a_n,), (a"$_1$,…,a"$_n$), t_{vt_new}):
<delete^B(r, (a_1,…,a_n,)); insert^B(r, (a"$_1$,…,a"$_n$), t_{vt_new})>

Since there is no current proposal in DB_Proposers sets (see Assumptions D.1), the possible tuples to be updated are only the tuples belonging to DB_Evaluators relations, i.e., an update cannot update a proposal. Therefore, the sequence of proposal and acceptance of update in our approach has the following parameters:

propose_update(r, <(a_1,…,a_n), (a_1,…,a_n)>,(a"$_1$,…,a"$_n$, p_{new}|t_{vt_new}))
accept_update(r, <(a_1,…,a_n), (a"$_1$,…,a"$_n$, p_{new}|t_{vt_new})>, e)

In particular, notice that, since we update a DB_Evaluators tuple, in the second parameter of propose_update the origin and the alternative coincide (see discussion in Section 5.1).

We prove that a sequence <propose_update; accept_update> in our approach is equivalent (with regards to the DB_Evaluators relation) to an update operation in BCDM, i.e.:

<propose_update(r, <(a_1,…,a_n),(a_1,…,a_n)>,(a"$_1$,…,a"$_n$,p_{new}|t_{vt_new})); accept_update(r, <(a_1,…,a_n), (a"$_1$,…,a"$_n$, p_{new})>, e)>
is equivalent to

update^B(r, (a_1,…,a_n,), (a"$_1$,…,a"$_n$), t_{vt_new})

Let us consider Definition 5.1.1 (admissible_propose_update). For the sake of convenience, we report the definition here:

Definition 5.1.1: admissible_propose_update.

admissible_propose_update(propose_update(r,<(a_1,...,a_n),(a'_1,...,a'_n,p_{old})>, (a''_1,...,a''_n,p_{new}|t_{vt_new}))):

(1) ($\exists$x$\in$r: (x[A]=(a_1,...,a_n)$\wedge$ current(x)) $\vee$ $\exists$x$\in$pi(r): (x[A]=(a_1,...,a_n)$\wedge$current(x))) $\wedge$

(2) ($\exists$pt$\in$pu(r) : (origin(pt)=(a_1,...,a_n) $\wedge$ $\exists$ y $\in$ alternatives(pt) :

(y[A] = (a_1',...,a_n') $\wedge$ y[P]= p_{old} $\wedge$ current(y)) $\vee$ (a_1,...,a_n)=(a'_1,...,a'_n))) $\wedge$

(3) $\forall$pt $\in$ pu(r) (origin(pt)=(a_1,...,a_n)) $\Rightarrow$ ($\neg\exists$ z $\in$ alternatives(pt):

(z[A] = (a_1'',...,a_n'') $\wedge$ current(z) $\wedge$ ρ^{e}_{UC} (z) [T_v] = t_{vt_new}))) $\wedge$

(4) $\forall$k$\in$ r ((k[A]= (a_1'',...,a_n'') $\wedge$ current(k)) $\Rightarrow$ (a_1'',...,a_n'')=(a_1,...,a_n)) $\wedge$

(5) p_{new} $\in$ Proposers♦

Under Assumptions D.1, it is possible to write the definition in a simpler form, in fact:

(1) the second disjunct of condition (1) (i.e., $\exists$x$\in$pi(r) : (x[A]=(a_1,...,a_n) $\wedge$ current(x))) is trivially false for Assumption D.1a;

(2) the first disjunct of condition (2) (i.e., $\exists$pt$\in$pu(r) : (origin(pt)=(a_1,...,a_n) $\wedge$ $\exists$ y $\in$ alternatives(pt) : (y[A] = (a_1',...,a_n') $\wedge$ y[P]= p_{old} $\wedge$ current(y)) is trivially false for Assumption D.1c; while the second disjunct of condition (2) (i.e., (a_1,...,a_n)=(a'_1,...,a'_n)) is trivially true because we are considering only updates of tuples in a DB_Evaluators relation;

(3) condition (3) is trivially true for Assumption D.1c;

(5) condition (5) is trivially true for Assumption D.1d.

Taking into account the above simplifications, we report the Definition 5.1.1 in Definition 5.1.1'':

Definition 5.1.1'': admissible_propose_update.

admissible_propose_update(propose_update(r,<(a_1,...,a_n),(a'_1,...,a'_n,p_{old})>, (a''_1,...,a''_n,p_{new}|t_{vt_new}))):

(1) ($\exists$x$\in$r : (x[A]=(a_1,...,a_n) $\wedge$ current(x)) $\wedge$

(4) $\forall$k$\in$ r ((k[A]= (a_1'',...,a_n'') $\wedge$ current(k)) $\Rightarrow$ (a_1'',...,a_n'')=(a_1,...,a_n)) ♦

Specifically, under Assumptions D.1, a proposal of update is admissible if:

(1) *(a_1,…,a_n)* identifies a tuple *x* in the evaluator relation *r* and such a tuple is current , and

(4) there is no current tuple *k*$\in$*r* which is value equivalent to the new proposal *(a_1'',…,a_n''),* except (possibly) the origin itself.

There are three cases:

a) there is not a current tuple x in the evaluator relation r such that x[A]=*(a_1,…,a_n);*

b) there exists a current tuple *k*$\in$*r* value equivalent to the new proposal *(a_1'',…,a_n'')* and *(a_1'',…,a_n'')* $\neq$ *(a_1,…,a_n);*

c) otherwise

In both cases a) and b) operation propose_update(r, <(a_1,…,a_n), (a_1,…,a_n)>,(a''_1,…,a''_n, p_{new}|t_{vt_new})) is not admissible (and the propose_update operation has no effect).

In case a) (and only in case a)), also the delete^B operation in the sequence <delete^B(r, (a_1,…,a_n)); insert^B(r, (a''_1,…,a''_n), t_{vt_new})> has no effect (see the proof for propose_delete operation).

In case b) (and only in case b), also the insert^B operation in the sequence <delete^B(r, (a_1,…,a_n,)); insert^B(r, (a"$_1$,…,a"$_n$), t_{vt_new})> has no effect (see the proof for propose_insert operation).

Therefore, in both cases, the update^B operation has no effect.

Now let us consider case c). In this case, the propose_update(r, <(a_1,…,a_n,), (a_1,…,a_n)>,(a"$_1$,…,a"$_n$, p_{new}|t_{vt_new})) operation is admissible. Then, after the operation is executed, we have in pu(r) a Proposal-tuple pt with origin(pt)=(a_1,…,a_n) and with an alternative (a"$_1$,…,a"$_n$, p_{new}|t_b) such that the valid time of the tuple at transaction time UC is t_{vt_new}, i.e. ρ^e_{uc}(a"$_1$,…,a"$_n$, p_{new}|t_b)[T_v] = t_{vt_new}.

Now let us consider the admissible_accept_update function (Definition 5.2.1) and the accept_update operation (Definition 5.2.2).

The accept_update(r, <(a_1,...,a_n), (a"$_1$,...,a"$_n$, p_{new})>, e) operation is admissible because (see Definition 5.2.2): conjuncts (1) and (2) are true because of the result of the propose_update operation; conjunct (3) is true since the case in which condition (3) does not hold has been already analyzed (case b) above); conjunct (4) is true because of Assumption D.1d.

In the accept_update operation (Definition 5.2.2), case (1) is not executed because we assume that there is no current proposal in pi(r) (see Assumption D.1a). Case (2) is executed since, in case c) there is a current tuple x in r such that x[A]= a_1,…,a_n and, as a result of the propose_update operation, there is a current Proposal-tuple with origin (a_1,…,a_n) and with an alternative (a"$_1$,…,a"$_n$, p_{new}|t_b) such that the valid time of the tuple at transaction time UC is t_{vt_new}, i.e. ρ^e_{uc}(a"$_1$,…,a"$_n$, p_{new}|t_b)[T_v] = t_{vt_new}. In this case, the operation performs a sequence of operations <delete^B(r, (a_1,…,a_n)); insert^B(r, (a"$_1$,...,a"$_n$,), ρ^e_{uc}(a"$_1$,…,a"$_n$, p_{new}|t_b) [T_v]))>, which is equivalent to <delete^B(r, (a_1,…,a_n)); insert^B(r, (a"$_1$,...,a"$_n$,), t_{vt_new})>.

Finally, the delete_alternatives(r,(a_1,…,a_n, p_{new})) operation "closes" in pu(r) the tuple inserted by the propose_update operation, in such a way that also for the subsequent operations we can assume that there are no current proposals. ♦

Property 6.3: Reducibility of Proposal-tuple-sets.

Our algebraic operators on Proposal-tuple-sets are reducible to BCDM algebraic operators, i.e., for each algebraic unary operator Op^{PV} in our model, and indicating with Op^B the corresponding BCDM operator, for each Proposal-tuple-set *s*, the following holds (the analogous holds for binary operators):

$$\text{convert}(Op^{PV}(s)) = Op^B(\text{convert}(s)) \blacklozenge$$

Proof:

For the sake of brevity, we prove the property considering the natural join operator. The proofs for the other operators are similar.

Let r and s be Proposal-tuple-sets, with schemata <(A^{orig},B^{orig}), (A,B|T)> and <(A^{orig},C^{orig}), (A,C|T)> respectively, where A^{orig}, B^{orig}, C^{orig}, A, B and C are sets of atemporal attributes, and A and A^{orig} denote the join attribute(s), then

$$\text{convert}(r \bowtie^{PV} s) = \text{convert}(r) \bowtie^B \text{convert}(s),$$

where $\bowtie^{PV}$ is the natural join operator of our approach (see Definition 6.1), $\bowtie^B$ is the natural join operator of BCDM ([Snodgrass 1995], see Section 3) and convert is the operator introduced in Definition 6.2.

We show the equivalence by proving the two inclusions separately, i.e., we prove that the left-hand side of the formula (henceforth lhs) implies the right-hand side (henceforth rhs) and that the rhs implies the lhs.

($x'' \in lhs \Rightarrow x'' \in rhs$)

Let $x'' \in lhs$. Then, by the definition of convert, there exists a Proposal-tuple $x' \in r \bowtie^{PV} s$ such that $origin(x')=x''[A^{orig},B^{orig},C^{orig}]$ and there exists an alternative $alt' \in alternatives(x')$ such that $alt'[A,B,C]=x''[A,B,C]$ and $alt'[T]=x''[T]$.

By the definition of $\bowtie^{PV}$, there exist Proposal-tuples $x_1 \in r$ and $x_2 \in s$ such that $origin(x_1)[A^{orig}]= origin(x_2)[A^{orig}]=origin(x')[A^{orig}]$, $origin(x_1)[B^{orig}]= origin(x')[B^{orig}]$, $origin(x_2)[C^{orig}]= origin(x')[C^{orig}]$ and there exist alternatives $alt_1 \in alternatives(x_1)$ and $alt_2 \in alternatives(x_2)$ such that $alt_1[A]=alt_2[A]=alt'[A]$, $alt_1[B]=alt'[B]$, $alt_2[C]=alt'[C]$ and $alt_1[T] \cap alt_2[T]=alt'[T]$.

Then, by the definition of convert, there exists a tuple $x_1' \in convert(r)$ such that $x_1'[A^{orig},B^{orig}]=origin(x_1)[A^{orig},B^{orig}]=origin(x')[A^{orig},B^{orig}]$, $x_1'[A,B]=alt_1[A,B]=alt'[A,B]$, $x_1'[T]=alt_1[T]$ and there exists a tuple $x_2' \in convert(s)$ such that $x_2'[A^{orig},C^{orig}]=origin(x_2)[A^{orig},C^{orig}]=origin(x')[A^{orig},C^{orig}]$, $x_2'[A,C]= alt_2[A,C]=alt'[A,C]$, $x_2'[T]=alt_2[T]$.

Then, by the definition of $\bowtie^{B}$, since $x_1'[A^{orig}]=x_2'[A^{orig}]$ and $x_1'[A]=x_2'[A]$, there exists $x_{12}'' \in rhs$ such that $x_{12}''[A^{orig},B^{orig}]=x_1'[A^{orig},B^{orig}]$, $x_{12}''[A^{orig},C^{orig}]=x_2'[A^{orig},C^{orig}]$, $x_{12}''[A,B]=x_1'[A,B]$, $x_{12}''[A,C]=x_2'[A,C]$ and $x_{12}''[T]=x_1'[T] \cap x_2'[T]$.

By construction, $x_{12}''=x''$.

($x'' \in rhs \Rightarrow x'' \in lhs$)

Now assume $x'' \in rhs$. Then, by definition of $\bowtie^{B}$, there exist tuples $x_1' \in convert(r)$ and $x_2' \in convert(s)$ such that $x_1'[A^{orig}]=x_2'[A^{orig}]$ and $x_1'[A]=x_2'[A]$ and $x_1'[A^{orig},B^{orig}]=x''[A^{orig},B^{orig}]$, $x_2'[A^{orig},C^{orig}]=x''[A^{orig},C^{orig}]$, $x_1'[A,B]=x''[A,B]$, $x_2'[A,C]=x''[A,C]$ and $x_1'[T] \cap x_2'[T]=x''[T]$.

By the definition of convert, there exists a Proposal-tuple $x_1 \in r$ such that $origin(x_1)[A^{orig},B^{orig}]=x_1'[A^{orig},B^{orig}]$ with an alternative $alt_1 \in alternatives(x_1)$ such that $alt_1[A,B]=x_1'[A,B]$ and $alt_1[T]=x_1'[T]$, and there exists a Proposal-tuple $x_2 \in s$ such that $origin(x_2)[A^{orig},C^{orig}]= x_2'[A^{orig},C^{orig}]$ with an alternative $alt_2 \in alternatives(x_2)$ such that $alt_2[A,C]= x_2'[A,C]$ and $alt_2[T]= x_2'[T]$.

Then by definition of $\bowtie^{PV}$, since $x_1[A^{orig}]=x_2[A^{orig}]$ and $x_1[A^{o}]=x_2[A]$, there must exist a Proposal-tuple $x' \in r \bowtie^{c} s$ such that $origin(x')[A^{orig},B^{orig}]=x_1[A^{orig},B^{orig}]$, $origin(x')[A^{orig},C^{orig}]= x_2[A^{orig},C^{orig}]$ and there must be an alternative $alt' \in alternatives(x')$ such that $x_1[A,B]=x_2[A,C]$, $alt'[A,B]=x_1[A,B]$, $alt'[A,C]=x_2[A,C]$ and $alt'[T]=alt_1[T] \cap alt_2[T]$.

Then, by definition of convert, there exists a tuple $x_{12}'' \in lhs$ such that $x_{12}''[A^{orig},B^{orig},C^{orig}]=origin(x')[A^{orig},B^{orig},C^{orig}]$, $x_{12}''[A,B,C]=alt'[A,B,C]$ and $x_{12}''[T]=alt'[T]$.

By construction, $x_{12}''=x''$. ♦

Corollary 6.4: Reducibility.

The algebraic operators in our approach are reducible to BCDM algebraic operators. ♦

Proof: It is trivial, considering Property 6.3 for pu(r) and the reducibility property proved in [Snodgrass 1995] for BCDM relations (in fact, pi(r) and pd(r) sets can be interpreted as BCDM relations). ♦

Appendix E. Focusing on transaction-time relations

In this Appendix, we explicitly deal with the case in which only transaction-time relations (i.e., relations which have no valid time in their schema) are to be taken into account. This is the case, for instance, of the example in Appendix A. Although this is a quite trivial restriction of the general approach described in the paper, we mention it here, for the sake of completeness.

Also in case the database contains only transaction-time relations, the general Definition 4.0.2 applies. Specifically, a database is still a pair <DB_Evaluators, DB_Proposers>. Of course, the schema of the relations in DB_Evaluators is now $R=(A_1,\ldots,A_n| T_t)$, where T_t is an implicit timestamp attribute with domain D_{TT}. Analogously, for each relation r having schema $R=(A_1,\ldots,A_n| T_t)$ belonging to DB_Evaluators, the schema of the set pi(r) is now $R'=(A_1,\ldots,A_n,P|T_t)$, and the schema of pu(r) is now $<(A_1, \ldots,A_n), (A_1, \ldots,A_n,P|T_t)>$ (pd(r) is unchanged, since also in the original model it did not contain the valid time).

As concerns manipulation operations, in this Appendix, we describe only the new version of propose_update (called *propose_update'*), because the other adaptations are analogous.

Notice that, when only transaction time is used, the definition of the function current must be adapted, i.e., given a (transaction time) tuple x, **current'(x)**: $(UC \in x[T_t])$.

The propose_update' operation first checks the applicability of the proposal, through the admissible_propose_update' routine.

Definition E.1: admissible_propose_update'.

Given a relation $r \in$ DB_Evaluators with schema $R=(A_1,\ldots,A_n|T_t)$, let A stand for $(A_1,\ldots,A_n)$, let $<(A_1, \ldots,A_n), (A_1, \ldots,A_n,P|T_t)>$ be the schema of pu(r). We define admissible_propose_update' as follows:

admissible_propose_update'(propose_update'(r,<$(a_1,\ldots,a_n)$,$(a'_1,\ldots,a'_n,p_{old})$>,$(a''_1,\ldots,a''_n,p_{new})$)):

(1) $(\exists x \in r: (x[A]=(a_1,\ldots,a_n) \wedge$ current'(x)) $\vee \exists x \in pi(r): (x[A]=(a_1,\ldots,a_n) \wedge$current'(x))) $\wedge$

(2) $(\exists pt \in pu(r) : (origin(pt)=(a_1,\ldots,a_n) \wedge \exists y \in$ alternatives(pt) :

$(y[A] = (a_1',\ldots,a_n') \wedge y[P]= p_{old} \wedge$ current'(y))

$\vee (a_1,\ldots,a_n)=(a'_1,\ldots,a'_n))) \wedge$

(3) $(\forall pt \in pu(r)$ $(origin(pt)=(a_1,\ldots,a_n)) \Rightarrow (\neg \exists z \in$ alternatives(pt):

$(z[A] = (a_1'',\ldots,a_n'') \wedge$ current'(z)))) $\wedge$

(4) $\forall k \in r$ $(k[A]= (a_1'',\ldots,a_n'') \Rightarrow \neg$ (current'(k))) $\wedge$

(5) $p_{new} \in$ Proposers♦

Conditions (1), (2), and (5) are the same as in Definition 5.1.1.

In the condition (3) in Definition 5.1.1 there is another conjunct (i.e., $\rho^e_{UC}(z)[T_v] = t_{vt\_new}$) that is used to allow the case in which a new proposal $(a_1'',\ldots,a_n'')$ is weakly value equivalent to an already existing alternative *z* in the Proposal-tuple *pt* with origin $(a_1,\ldots,a_n)$, but differs from *z* due to its valid time (at transaction time equal to UC).

Also condition (4) is more restrictive than condition (4) in Definition 5.1.1. In this version, we do not admit that the new proposal $(a_1'',...,a_n'')$ is value equivalent to any current tuple $k \in r$ while in Definition 5.1.1 we admit new proposals value equivalent to the origin in case they have a different valid time.

The new version of propose_update is similar to the one in Definition 5.1.3; the only difference consisting in the action part of the operation. As a matter of fact, in all the cases in the definition, the new tuple has to be inserted, having UC in its transaction time. However, in the case in which no valid time has to be considered, this simply amounts to adding UC to the (transaction) time of the tuple (while in the general case in Definition 5.1.3 the bitemporal chronons holding at UC must be introduced).

Definition E.2: propose_update'.

Given a relation $r \in DB\_Evaluators$ with schema $R=(A_1,\ldots,A_n|T_t)$, let A stand for $(A_1,\ldots,A_n)$, let $<(A_1, \ldots,A_n), (A_1, \ldots,A_n,P|T_t)>$ be the schema of pu(r), and let $(A_1, \ldots,A_n,P|T_t)$ be the schema of a new proposal of update. We define propose_update' as follows:

if(admissible_propose_update'(propose_update'(r,<$(a_1,...,a_n)$,$(a'_1,...,a'_n,p_{old})$>,$(a''_1,...,a''_n,p_{new})$))) **then**

begin

(1) **if** $(\neg \exists\, pt \in pu(r) : origin(pt)=(a_1,...,a_n))$

then $pu(r) \leftarrow pu(r) \cup \{create\_pt((a_1,...,a_n),\{(a''_1,...,a''_n, p_{new}|\{UC\})\})\}$

(2) **else if** $(\exists\, pt \in pu(r) : (origin(pt)=(a_1,...,a_n) \wedge$

$(\forall y \in alternatives(pt)\ (y[A]=(a_1'',...,a_n'') \Rightarrow y[P] \neq p_{new})))$

then $pu(r) \leftarrow pu(r) - \{pt\} \cup \{create\_pt((a_1,...,a_n), alternatives(pt) \cup \{(a''_1,...,a''_n,p_{new}|\{UC\})\})\}$

(3) **else if** $(\exists\, pt \in pu(r) : (origin(pt)=(a_1,...,a_n) \wedge$

$\exists\, y \in alternatives(pt) : (y[A] = (a_1'',...,a_n'') \wedge y[P]= p_{new})))$

then $pu(r) \leftarrow pu(r) - \{pt\} \cup \{create\_pt((a_1,...,a_n), alternatives(pt) - y \cup \{(a''_1,...,a''_n,p_{new}|y[T_t] \cup \{UC\})\})\}$

end♦