A Simple Model of Negotiation for Cooperative Updates on Database Schema Components

Stephen J. HEGNER
Umeå University, Department of Computing Science
SE-901 87 Umeå, Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

Abstract. Modern applications involving information systems often require the cooperation of several distinct users, and many models of such cooperation have arisen over the years. One way to model such situations is via a cooperative update on a database; that is, an update for which no single user has the necessary access rights, so that several users, each with distinct rights, must cooperate to achieve the desired goal. However, cooperative update mandates new ways of modelling and extending certain fundamentals of database systems. In this paper, such extensions are explored, using database schema components as the underlying model. The main contribution is an effective three-stage process for inter-component negotiation.

Keywords. database, component

Introduction

The idea of modelling large software systems as the interconnection of simpler components, or *componentware* [3], has long been a central topic of investigation. In recent work, Thalheim has forwarded the idea that a similar approach, that of *database componentware*, is a fruitful direction for the modelling of large database systems [23]. Database componentware is a true software-component approach, in that it embodies the principle of *co-design* [24] [10] — that applications should be integrated into the design of information systems. Indeed, the formal model [25] is closely related to that of the software components of Broy [5] [6]. While this approach has obvious merits, it does involve one substantial compromise; namely, the classical notion of *conceptual data independence* [17, p. 33] is sacrificed, since the applications are integral to the design. As new applications become necessary, or as existing applications must be modified, a change to the entire design may become necessary. It is therefore appropriate to ask whether a component-based approach to modelling database systems which preserves conceptual data independence, and thus mirrors more closely the traditional notions of a database schema, is feasible. In [12], the foundations for such a framework were presented. The core idea is that of a *schema component*, consisting of database schema and a collection

of its views, called *ports*. Interconnections are formed by connecting ports; that is, by requiring the states of connected ports to match. Such an interconnection defines a composite database schema. The idea is closely related to lossless and dependency-preserving decomposition, but it is really a theory of composition — the main schema is constructed from components rather than decomposed into constituents. The structure necessary to connect components together is part of the definition of the components themselves.

The ultimate value of any concept lies in its applicability. In [15], initial ideas surrounding the use of schema components as the underlying framework for the support of cooperative update were presented. The model developed was a proof-of-concept effort, and many simplifying assumptions were made. Furthermore, the focus was upon a formal computational model rather than upon an illustration of how the technique may be used to model situations requiring cooperative update. The goal of this paper is to complement and extend [15]. The main contribution the presentation of a simple yet effective negotiation process. Any approach to cooperative update must support negotiation while still providing for reasonable convergence. While the process described in [15] is guaranteed to converge, the number of steps which are possible can be very large [15, 3.5(a)]. In this paper, a much more efficient negotiation process is developed in which each component executes at most three negotiating steps. This process is illustrated via an extended and annotated example, rather than via a completely formal model.

There are a number of other aspects of cooperative update which were not even mentioned, much less addressed, in [15]. In this paper, several of the most important are discussed briefly, and illustrated relative to the running example. One of the most important is relative authority. Even in cooperative situations, there will typically be a hierarchy of authority, so that some players will be obligated in certain ways to accommodate the proposals of others. Others include models of behavior when actors are presented with choices for supporting an update request, and models for ensuring the cooperation does not lead to corruption.

There has been considerable research on the general topic of cooperative work in general and cooperative transactions in particular [16] [22] [28]. There has also been some very recent work on synchronizing updates to repositories [18]. Relative to these, the focus of this paper is upon how an update which is proposed by a single agent (the initiator) to a single schema component may be realized via suitable updates to other components. It does not address more general situations in which a group of agents must begin from scratch to produce a desired final result, although such situations could conceivably be modelled within the context of schema components also.

1. Fundamentals of Schema Components and Cooperative Update

The work of this paper is based upon the formal foundations of schema components and cooperative update, as presented in [12] and [15], respectively. While a complete understanding of the formalisms of those papers is not absolutely necessary for this paper, it is nevertheless useful for the reader to be familiar with the basic concepts and notation. The purpose of this section is to summarize the material from those two references which is central to the rest of this paper. The reader may wish to skim this section rather briefly, referring back to it as the need arises. In any case, the reader is referred to those papers for details and a more systematic presentation. The ideas are presented in terms of the

classical relational model, although they may easily be generalized to any data model admitting the notions of state and of view.

1.1. Schema Components

Let $\mathbf{E}_0$ be the relational schema with the single relation symbol $R[ABCDE]$, constrained by the functional dependencies (FDs) $\mathcal{F} = \{B \to C, C \to DE\}$. The notation $\mathsf{LDB}(\mathbf{E}_0)$ is used to represent the set of all *legal databases* of $\mathbf{E}_0$; that is, the set of all relations on $ABCDE$ which satisfy the FDs in $\mathcal{F}$, while $\mathsf{DB}(\mathbf{E}_0)$ denotes the set of all databases on $\mathbf{E}_0$ which may or may not satisfy the constraints of $\mathcal{F}$.

Consider the decomposition of this schema into its four projections in $\{AB, BC, CD, CE\}$. Using classical relational database theory, it is easy to establish that this decomposition is *lossless*, in the sense that the original database may be reconstructed by joining together the projections, and *dependency preserving* in the sense that the elements of $\mathcal{F}$ may be recovered from the dependencies which are implied on the projections. Together, these two properties imply that there is a natural bijective correspondence between $\mathsf{LDB}(\mathbf{E}_0)$ and the *decomposed databases*. More precisely, if $N = \langle N_{AB}, N_{BC}, N_{CD}, N_{CE} \rangle$ is a quadruple of databases, with N_{AB} a relation on AB which satisfies all of the dependencies in (the closure of) $\mathcal{F}$ which embed into AB, and likewise for N_{BC}, N_{CD}, and N_{CE} on their respective projections, then there is an $M \in \mathsf{LDB}(\mathbf{E}_0)$ which decomposes into N.

To proceed further, a more comprehensive notation is essential. Define $\Pi^{\mathbf{E}_0}_{BC} = (\mathbf{E}^{BC}_0, \pi^{\mathbf{E}_0}_{BC})$ to be the view which is the projection of R onto BC. Here $\mathbf{E}^{BC}_0$ is the relational schema with the single relation symbol R_{BC}, constrained by $\mathcal{F}_{AB} = \{B \to C\}$, and $\pi^{\mathbf{E}_0}_{BC} : \mathbf{E}_0 \to \mathbf{E}^{BC}_0$ is the projection of R onto R_{BC}. The views $\Pi^{\mathbf{E}_0}_{AB}$, $\Pi^{\mathbf{E}_0}_{CD}$, and $\Pi^{\mathbf{E}_0}_{CE}$ are defined in a completely analogous fashion, with analogous notation, as the projections onto the given sets of attributes.

Modelling using components embraces explicitly two related notions which are only implicit in the above view-based approach. First, the model is totally distributed, in the sense that no reference to a main schema is necessary. Second, because of this lack of an explicit main schema, the means by which the components are interconnected must be made explicit. These ideas are now examined in more detail in the light of the above example.

The component corresponding to $\Pi^{\mathbf{E}_0}_{AB}$ consists of the schema $\mathbf{E}^{AB}_0$ together with the view $\Pi^{\mathbf{E}^{AB}_0}_{B}$ of $\mathbf{E}^{AB}_0$ which projects AB onto B. Write $K_{AB} = (\mathbf{E}^{AB}_0, \{\Pi^{\mathbf{E}^{AB}_0}_{B}\})$. The view $\Pi^{\mathbf{E}^{AB}_0}_{B}$ is called a *port* of K_{AB} because it is used to connect to other components. A component may have more than one port. Indeed, $K_{BC} = (\mathbf{E}^{BC}_0, \{\Pi^{\mathbf{E}^{BC}_0}_{B}, \Pi^{\mathbf{E}^{BC}_0}_{C}\})$ has two ports. The components $K_{CD} = (\mathbf{E}^{BC}_0, \{\Pi^{\mathbf{E}^{CD}_0}_{C}\})$ and $K_{CE} = (\mathbf{E}^{BC}_0, \{\Pi^{\mathbf{E}^{CE}_0}_{C}\})$, each with a single port, are defined similarly. For each of these components, the first entry is the schema and the second its set of ports. It is convenient to have a graphical notation for the representation of interconnected components. Figure 1 illustrates this notation for the example just given. The components are represented as rectangles, with the ports depicted as circles. When two ports are connected, they are shown as a single circle.

The *interconnection family* for Figure 1 specifies how the components are interconnected, and gives the sets of ports which are connected together. In this case, it is $J_0 = \{\{\Pi^{\mathbf{E}^{AB}_0}_{B}, \Pi^{\mathbf{E}^{BC}_0}_{B}\}, \{\Pi^{\mathbf{E}^{BC}_0}_{C}, \Pi^{\mathbf{E}^{CD}_0}_{C}, \Pi^{\mathbf{E}^{CE}_0}_{C}\}\}$. A single member of an interconnection fam-

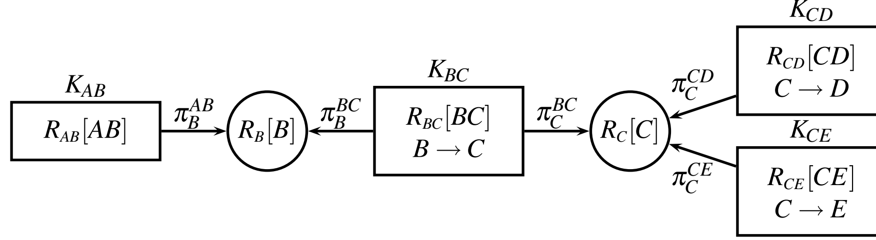

Figure 1. An interconnection of components

ily is called a *star interconnection*. Thus, J_0 consists of two star interconnections. For this notation to be unambiguous, the set of components must be *name normalized*, in that globally, over all components, no two ports have the same name. Since this is just a naming convention, it can always be met through suitable renaming. Note, on the other hand, for two ports to be members of the same star interconnection, they must have identical schemata. For example, even though $\Pi_B^{\mathbf{E}_0^{AB}}$ and $\Pi_B^{\mathbf{E}_0^{BC}}$ are distinct ports, from distinct components, they have identical (and not just isomorphic) schemata. This condition is essential because the semantic condition on such an interconnection is that the states of all such view schemata must be identical. When the port schema (defined by R_B in this case) is from a view of a main schema ($\Pi_B^{\mathbf{E}_0}$ in this case), this happens automatically, but in the case of component interconnection without reference to a main schema, it must be enforced explicitly. Note further that the graphical notation of Figure 1 embodies this idea implicitly, since each common port schema is represented by a single circle.

1.2. Cooperative Update

For convenience, assume that the current state of the main schema is $M = \{R(\mathrm{a}_1,\mathrm{b}_1,\mathrm{c}_1,\mathrm{d}_1,\mathrm{e}_1), R(\mathrm{a}_2,\mathrm{b}_2,\mathrm{c}_2,\mathrm{d}_2,\mathrm{e}_2)\}$. The state of $\Pi_{AB}^{\mathbf{E}_0}$ is then $M_{AB} = \{R_{AB}(\mathrm{a}_1,\mathrm{b}_1), R_{AB}(\mathrm{a}_2,\mathrm{b}_2)\}$, with the states M_{BC}, M_{CD}, and M_{CE} of $\Pi_{BC}^{\mathbf{E}_0}$, $\Pi_{CD}^{\mathbf{E}_0}$, and $\Pi_{CE}^{\mathbf{E}_0}$ obtained similarly. Suppose that a given user $\mathfrak{a}_{AB}$ has access to the database only through view $\Pi_{AB}^{\mathbf{E}_0}$, and wishes to insert $R_{AB}(\mathrm{a}_3,\mathrm{b}_2)$. This update can be realized entirely within $\Pi_{AB}^{\mathbf{E}_0}$. By inserting $R(\mathrm{a}_3,\mathrm{b}_2,\mathrm{c}_2,\mathrm{d}_2,\mathrm{e}_2)$ into M, the desired update to $\Pi_{AB}^{\mathbf{E}_0}$ is achieved without altering the state of any of the other three views. Indeed, this is an instance of update via the classical *constant-complement strategy* [2]. The mutual view $\Pi_B^{\mathbf{E}_0}$, the projection onto B, is called the *meet* of $\Pi_{AB}^{\mathbf{E}_0}$ and $\Pi_{BC}^{\mathbf{E}_0}$, and is precisely that which must be held constant under the constant complement strategy [11].

Now suppose that instead that user $\mathfrak{a}_{AB}$ wishes to insert $R_{AB}(\mathrm{a}_3,\mathrm{b}_3)$. This update cannot be realized by a change to the state of $\Pi_{AB}^{\mathbf{E}_0}$ which holds the states of the other three views constant. Indeed, it is necessary to insert a tuple of the form $R_{BC}(\mathrm{b}_3,\mathrm{c}_?)$ into the state of $\Pi_{BC}^{\mathbf{E}_0}$. Since user $\mathfrak{a}_{AB}$ does not have write access to view $\Pi_{BC}^{\mathbf{E}_0}$, the cooperation of another user who has such write access, say $\mathfrak{a}_{BC}$, is necessary. If that user chooses to insert, say, $R_{BC}(\mathrm{b}_3,\mathrm{c}_2)$, then the process terminates without any need for cooperation from $\Pi_{CD}^{\mathbf{E}_0}$ or $\Pi_{CE}^{\mathbf{E}_0}$. However, if user $\mathfrak{a}_{BC}$ chooses to cooperate by inserting, say, $R_{BC}(\mathrm{b}_3,\mathrm{c}_3)$, then the cooperation of additional users, one for $\Pi_{CD}^{\mathbf{E}_0}$ and one for $\Pi_{CE}^{\mathbf{E}_0}$ is necessary. Finally, if these additional users choose to insert $R_{CD}(\mathrm{c}_3,\mathrm{d}_3)$ and $R_{CE}(\mathrm{c}_3,\mathrm{e}_3)$, respectively, then

the tuple $R(\mathrm{a}_3, \mathrm{b}_3, \mathrm{c}_3, \mathrm{d}_3, \mathrm{e}_3)$ may be inserted into the state M of $\mathbf{E}_0$ to achieve the desired result. Note that no single user, of a single view, could effect this update; by its very nature it requires the cooperation of distinct views, likely controlled by distinct users.

2. Three-Stage Negotiation for Cooperative Update

In this section, a three-stage negotiation process for cooperative update on an interconnection of schema components is developed. Rather than presenting a completely formal model, the main ideas are developed in detail in the context of a simple business process, the approval of a travel request. This example is superficially similar to that found in [15]; however, not only the example process but also the underlying schema differs substantially, because the points which require emphasis are quite different.

2.1. The Schemata and Components of the Example

Figures 2 and 3, together with Table 1, provide the basic definitions for the example, which is presented in the relational model. In Figure 2, the *immutable* relations of the model; that is, the ones which may not be updated (at least for the purposes of servicing a business process) are shown. Keys are marked with an underline, while set-valued attributes (*i.e.*, *multisets* in the terminology of SQL:2003 [8]) are marked with a wavy underscore. Thus, each employee has an employee ID, a home department defined by the ID of that department, and a set of assigned projects. Similarly, each department has a supervisor, each account has an account manager, and each project has a supervisor and a set of accounts (for travel funds). These relations are shared by all components.

Employee [EmpID, DeptID, ProjIDs]   *Department* [DeptID, SupID]

Project [ProjID, SupID, ProjAccts]   *Account* [AcctID, AMgrID]

Figure 2. The immutable relations of the running example

Figure 3, which employs the symbolic notation which was introduced in [12] and is summarized in Section 1, shows the basic schema components and ports. The upper line in each rectangle (*e.g.*, Accounting) gives the name of the associated component, while the lower line (*e.g.*, $R_{\mathsf{Actg}}\ S_{\mathsf{Bank}}$) identifies the *mutable* relations which define the schema of that component; that is, the relations which may be modified in the course of servicing a travel request. Shown within each circle is the relation defining the schema of the associated port.

Information on the attributes of the individual relations of the components, aside from the port relations, is given in Table 1. For each attribute name, a checkmark in the column of a relation indicates that the attribute is included in that relation, and an underline of a checkmark indicates that the given attribute is a key. Thus, for example, R_{Actg} may be expressed more completely in standard relational notation as R_{Actg}[TripID, EmpID, ProjID, TotalCost, AcctID, ApprvAcct]. Since TripID is a key for every relation of the form R_{xxx} (*i.e.*, every relation except S_{Bank}), those relations may be joined together to form one large relation R on the set of all attributes

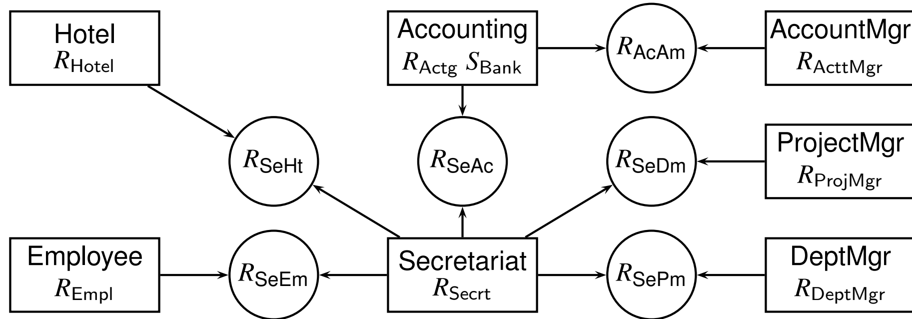

Figure 3. The components of the running example and their relations

Travel	R_{Empl}	R_{Secrt}	R_{Hotel}	R_{Actg}	R_{ActtMgr}	R_{ProjMgr}	R_{DeptMgr}	S_{Bank}
TripID	$\underline{\checkmark}$	$\underline{\checkmark}$	$\underline{\checkmark}$	$\underline{\checkmark}$	$\underline{\checkmark}$	$\underline{\checkmark}$	$\underline{\checkmark}$	
EmpID	✓	✓		✓	✓	✓	✓	
ProjID	✓	✓		✓	✓	✓	✓	
Purpose	✓	✓				✓	✓	
StartDate	✓	✓	✓			✓	✓	
EndDate	✓	✓	✓			✓	✓	
Location	✓	✓	✓			✓	✓	
HotelCost	✓	✓	✓					
TotalCost	✓	✓		✓	✓			
AcctID		✓		✓	✓			$\underline{\checkmark}$
ApprvProj		✓				✓		
ApprvSup		✓					✓	
ApprvAcct		✓		✓	✓			
HotelName	✓	✓	✓					
Balance								✓

Table 1. The mutable relations of the running example

shown in Table 1, save for the last one, Balance, which is used only in S_{Bank}. Then S_{Bank} may be joined with R, since AcctID is a key for it, and thus a universal relation *Travel* on all of the attributes may be obtained, with each of the component relations a projection of *Travel*. Each relation associated with a port is also a projection of *Travel*; the attributes of a port schema are given by the intersection of the attributes associated with the connecting components. For example, the attributes of R_{SeAc} are $\{\mathsf{TripID}, \mathsf{EmpID}, \mathsf{ProjID}, \mathsf{TotalCost}, \mathsf{AcctID}, \mathsf{ApprvAcct}\}$.

The semantics of the attributes of Table 1 are self explanatory, for the most part. Each trip is taken by a single employee and is associated with a single project. It has a purpose, a start date, and end date, and a location. There is a total cost for the entire trip, as well as the cost of just the hotel. The costs are charged to a single account. A trip must receive three distinct approvals, one by the project supervisor, one by the department supervisor, and one by account manager for the account to which the charges are made.

Finally, the relation S_{Bank} recaptures that each account has a balance, which is reduced accordingly when a trip is charged to that account.

The component interconnection of Figure 3 illustrates a *spoke-and-hub* topology, in that there is a central vertex (in this case Secretariat) which embodies most, but not all, of the mutable information. This is not an essential feature of the schema-component model, but it is a very useful architecture for many applications, such as the travel-request example considered here. Also, in Figure 3, each port schema connects only two components, but this is not a general requirement either, as the example of Section 1 illustrates.

2.2. The Representation of a Simple Update Request

In principle, a travel request may be initiated as an update to any of the components. Indeed, this is one of advantages of the using schema components to model business processes — the actual control flow need not be specified; rather, only the constraints on that flow imposed by the model need be respected. One of the most common cases is that that an employee, say Annie for the sake of concreteness, initiates a request for her own travel. Annie has write access only to the component Employee, and indeed, only to tuples of R_{Empl} which are associated with her EmpID. Suppose that she is working on the French project and wishes to travel to one of Nantes or Nice from April 1 to April 5. To express this request as an update, she obtains a new TripID from a server and proposes an insertion of a single tuple into R_{Empl} satisfying the following expression.

$$\begin{aligned}\mathbf{u}_{\mathsf{Empl:0}} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\ & \mathsf{Purpose} = \text{“meet with project partners”},\\ & \mathsf{StartDate} = 01.04.10,\ \mathsf{EndDate} = 05.04.10,\\ & (\langle \mathsf{Location} = \text{Nantes}\rangle \vee \langle \mathsf{Location} = \text{Nice}\rangle),\\ & 1000 \leq \mathsf{TotalCost} \leq 1500,\ \mathsf{HotelCost} \leq 1500,\ \mathsf{HotelName} = *\rangle.\end{aligned}$$

The plus sign indicates that the update is an insertion; that is, the tuple(s) indicated by the expression are to be inserted. It actually represents many possibilities, and so is termed a *nondeterminstic* update request, and the expression $\mathbf{u}_{\mathsf{Empl:0}}$ identifies an *update family*. Each possible update inserts only one tuple, but the values of the TotalCost, HotelCost, and HotelName fields are not fixed. No values for HotelCost and HotelName are excluded. Since Annie does not know Nantes, she has used the $*$ wildcard to indicate that she expresses no preference for a hotel, and allows a cost up to and including the total amount for the trip. Similarly, any value for TotalCost between 1000 and 1500 Euros inclusive is a possibility. In effect, an update family may be thought of as a set of ordinary, deterministic updates. In this case, there is one deterministic update in $\mathbf{u}_{\mathsf{Empl:0}}$ for each quadruple $(\text{Loc}, \text{TC}, \text{HC}, \text{HN})$ in which $\text{Loc} \in \{\text{Nantes}, \text{Nice}\}$, $1000 \leq \text{TC} \leq 1500$, $0 \leq \text{HC} \leq 1500$, and HN is the name of a hotel in the appropriate city.

It is assumed that all such update families are checked for integrity with the given constraints. For example, the relation *Employee* must reflect that Annie is a member of the French project.

2.3. The Three Stages of the Negotiation Process

Annie has the authority to update R_{Empl} only insofar as that update does not affect the other components. However, any of these proposed updates would affect the state of

R_{SeEm} as well. Thus, the cooperation of neighboring components, in this case the Secretariat component, must be obtained in order to obtain a *completion* of her initial request. The component Secretariat will then need to cooperate with other components. The process by which all components come to agreement on a completion of the initial update request $\mathbf{u}_{\mathsf{Empl:0}}$ is called *negotiation*.

In [15], a negotiation process is described in which any component can make a decision at any time. While such a model is very attractive theoretically and is well suited for the formal model presented there, convergence may be very slow. Here, a simple negotiation process is described in which each component goes through three distinct stages, although different components may be in different stages at different times. For a given component, each stage requires the execution of one well-specified task. Once these tasks are completed, the negotiation process is complete. In particular, negotiation cannot continue indefinitely in a back-and-forth fashion.

The description given below assumes that the interconnection are *acyclic* [12, Sec. 3], in the sense that there are no cycles in the graph which represents the interconnection of the components. The example interconnection of Figure 3 is acyclic. It also requires a few simple definitions. For components K and K', a *simple* path from K to K' goes through no component more than once. For example, in Figure 3, $\langle \mathsf{Employee}, \mathsf{Secretariat}, \mathsf{DeptMgr} \rangle$ is a simple path from Employee to DeptMgr, while $\langle \mathsf{Employee}, \mathsf{Secretariat}, \mathsf{ProjectMgr}, \mathsf{Secretariat}, \mathsf{DeptMgr} \rangle$ is a path which is not simple. For an acyclic graph, there is at most one simple path between any two components. Let Γ be a port of K'. Call Γ *inner* relative to K if it occurs on the simple path from K to K', and *outer* otherwise. For example, the port of Accounting defined by R_{SeAc} is inner with respect to Employee, while the port defined by R_{AcAm} is outer. Call a component K' *extremal* with respect to another component K if there is a simple path $K = K_0, K_1, \ldots, K_n = K'$ from K to K' and this path cannot be extended beyond K' while keeping it simple. Relative to Employee, the components Hotel, AccountMgr, ProjectMgr, and DeptMgr are extremal, while the others are not.

The three stages of the negotiation process are described as follows.

Stage 1 — Outward propagation: During Stage 1, the initial update request is radiated from the initiating component outwards to the other components. Each user of a given component, as it receives information about the initial update request, makes a decision regarding the way in which it is willing to support that request. It is only during this stage that such decisions may be made. In the later stages, each user must respect the decisions which were made in Stage 1. Since the underlying graph is assumed to be acyclic, each component receives information about the proposed update from at most one of its neighbors. Thus, there is no need to integrate information from different sources during this step.

The component which initiates the update request enters Stage 1 immediately. It then *projects* this request onto its ports; each neighboring component then *lifts* state on the port to an update request on its own schema. These neighboring components enter Stage 1 as soon as they have performed this lifting. The process then continues, with each component which are newly in Stage 1 projecting its lifting onto their inner ports relative to the initiating component. It ends when the liftings have been propagated to the extremal components.

Stage 2 — Propagate inward and merge: During Stage 2, the liftings which were chosen during Stage 1 are radiated back inwards towards the initiating component. In each component, the information from its neighbors which are connected to its outer ports is *merged* into a single update family. Since an extremal component has no outer ports, it enters Stage 2 as soon as it has decided upon a lifting for the update request. After that decision has been made, it is transmitted it back to the component from which the initial update request was received during Stage 1 by projecting it onto the appropriate port. Components which are not extremal enter Stage 2 when they have received a return update request from each neighbor which is connected to an external port, and then have merged the possibilities of these into a single update family. This merged update family is then transmitted back towards the initiating component via the inner ports of the current component. This merger may be empty, in which case it is impossible to realize the initial update request. However, even if it is empty, it is transmitted back.

Stage 3 — Choose final state and commit: Once the initiator of the update request has received and merged all of the incoming requests, it has reached Stage 2, and that marks the end of Stage 2 for all components, since all components have now merged the information from their more outward neighbors. The final step is for the initiating component to select one of the possibilities which it has computed in its merge as the actual update to its schema. (If this set of possibilities is empty, the update request fails.) Once it has chosen a possibility, it transmits this decision outward, just as in Stage 1. Each component must make a decision as to which of the possibilities in the update family determined in Stage 2 will be the actual update. This decision process is called Stage 3. Once all of these decisions are made, the update can be committed to the database.

There is one detail which was not elaborated in the above description. It is possible that some components will not need to be involved in the negotiation process, because none of the possible liftings will change their states. These components are simply ignored in the process.

2.4. The Negotiation Process via Example

The three-stage process described above is now illustrated on the running example, using the update family $\mathbf{u}_{\mathsf{Empl:0}}$ defined in 2.2.

In the first step, the update to the component Employee is projected onto the view R_{SeEm}; in this case R_{SeEm} and R_{Empl} have the same attributes and so this projection is the identity. At this point, Employee has completed Stage 1. Next, this projection must be lifted to an update family on the schema of the component Secretariat, which must include values for every attribute of R_{Secrt}; that is, every attribute listed in Table 1 save for Balance. Without further restrictions, a user of the Secretariat component (a human secretary, say) could choose any subset of the set of possible liftings to propagate forward, including the empty set, which would abort the proposed update. This liberal model is in fact used in [15]. In a real modelling situation, the set of liftings which are allowed must be regulated in some way; this topic is discussed further in 3.3. For now, assume that the rôle of the Secretariat carries no decision-making authority; thus, it must allow all possible liftings which do not involve extraneous *riders*, such as additional

travel for someone else. See 3.2 for an elaboration of this notion. The lifting will then have a representation of the following form.

$$\begin{aligned}\mathbf{u}_{\mathsf{Secrt:0}} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\ & \mathsf{Purpose} = \text{“meet with project partners”},\\ & \mathsf{StartDate} = 01.04.10,\ \mathsf{EndDate} = 05.04.10,\\ & (\langle \mathsf{Location} = \text{Nantes}\rangle \vee \langle \mathsf{Location} = \text{Nice}\rangle),\\ & \mathsf{HotelName} = *,\ \mathsf{HotelCost} \le 1500,\ 1000 \le \mathsf{TotalCost} \le 1500,\\ & \mathsf{ApprvProj} = \text{Carl},\ \mathsf{ApprvSup} = \text{Barbara},\\ & (\ \ \langle \mathsf{AcctID} = \text{A1},\ \mathsf{ApprvAcct} = \text{AM1}\rangle\\ & \ \ \vee\langle \mathsf{AcctID} = \text{A2},\ \mathsf{ApprvAcct} = \text{AM2}\rangle\\ & \ \ \vee\langle \mathsf{AcctID} = \text{A3},\ \mathsf{ApprvAcct} = \text{AM3}\rangle\\ & \ \ \vee\langle \mathsf{AcctID} = \text{A4},\ \mathsf{ApprvAcct} = \text{AM4}\rangle)\rangle\end{aligned}$$

The IDs for the project supervisor and department manager have been filled in, since these are single valued and given in the immutable tables *Project* and *Department*. Similarly, the identities of the four accounts which are associated with the French project, together with their managers, are obtained from the table *Account*. No decision on the part of the secretariat is required to determine these values. To complete the process for Stage 1 for component Secretariat, $\mathbf{u}_{\mathsf{Secrt:0}}$ is projected onto each outer port. At this point, Stage 1 for component Secretariat is complete.

Consider first the communication with the component Hotel, which is assumed to be autonomous (with no decision-making authority) and simply returns a list of available hotel rooms for the given time interval. Suppose that the following lifting is obtained.

$$\begin{aligned}\mathbf{u}_{\mathsf{Hotel:0}} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{StartDate} = 01.04.10,\ \mathsf{EndDate} = 05.04.10,\\ & \mathsf{Location} = \text{Nantes},\\ & (\ \ \langle \mathsf{HotelCost} = 1600,\ \mathsf{HotelName} = \text{TrèsCher}\rangle\\ & \ \ \vee\langle \mathsf{HotelCost} = 1200,\ \mathsf{HotelName} = \text{AssezCher}\rangle\\ & \ \ \vee\langle \mathsf{HotelCost} = 400,\ \ \mathsf{HotelName} = \text{PasCher}\rangle\\ & \ \ \vee\langle \mathsf{HotelCost} = 200,\ \ \mathsf{HotelName} = \text{Simple}\rangle)\rangle\end{aligned}$$

Thus, there are no hotels available in Nice for the request period of time, but there are four from which to choose in Nantes (although one turns out to be too expensive). Hotel is an extremal component, so upon placing this lifting on the port defined by R_{SeHt}, both Stage 1 and Stage 2 for that component are complete. This result is held by Secretariat until the other responses are received and it can complete its processing for Stage 2.

Next, consider the projection onto the outer port defined by R_{SeAc}, connected to component Accounting. Only the values for TripID, EmpID, ProjID, and TotalCost, as well as the alternatives for AcctID and ApprvAcct, are included. The lifting to the component Accounting must add information on the relation S_{Bank}, as shown below.

$$\begin{aligned}\mathbf{u}_{\mathsf{Actg:0}} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\ & (\ \ \langle \mathsf{AcctID} = \text{A1},\ 1000 \le \mathsf{TotalCost} \le 1500,\ \mathsf{ApprvAcct} = \text{AM1}\rangle\\ & \ \ \vee\langle \mathsf{AcctID} = \text{A2},\ \mathsf{TotalCost} = 1000,\ \mathsf{ApprvAcct} = \text{AM2}\rangle\\ & \ \ \vee\langle \mathsf{AcctID} = \text{A3},\ 1000 \le \mathsf{TotalCost} \le 1100,\ \mathsf{ApprvAcct} = \text{AM3}\rangle)\\ & \qquad\qquad \cup \pm\langle \mathsf{Balance} \leftarrow \mathsf{Balance} - \mathsf{TotalCost}\rangle\rangle\end{aligned}$$

The account A4 has been excluded because the balance was insufficient to fund the trip. (Assume that it was 900 Euros, say.) Similarly, the amounts allowed for accounts A2 and A3 are below those of the initial request, since these accounts cannot fund the entire 1500 Euros. This process of reducing the allowed liftings is called *trimming*. A decision to exclude other accounts, such as A2, might also be made; whether or not this would be allowed would depend upon the authority of the user of this component (see 3.3). However, in this example, all applicable accounts with sufficient balance have been included. Also, in this model, the entire cost of the trip must be paid from one account; the cost of a single trip may not be shared amongst accounts. In contrast to the update families which have been obtained thus far, this one is not a pure insertion. In order to pay for the trip, funds must be removed from the paying account. Thus, the update, which is tagged with a "+" indicating an insertion, also has a sub-update which is tagged with a "±", indicating a modification. Standard imperative programming notation has been used to express this.

To complete Stage 1 for Accounting, this update family is passed to component AccountMgr via the port with schema R_{AcAm}. Here there is not a single user which must construct a lifting; rather, each account manager must make a decision, and these decisions subsequently combined into a single lifting. However, no negotiation amongst these managers is required; the individual decisions are independent of one another. Suppose that two of the account managers agree to funding, each at a different level, but a third (AM2 for account A2) does not, so that the lifting in AccountMgr is given by the following expression.

$$\begin{aligned}\mathbf{u}_{\mathsf{Actg}:0} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\ & (\ \langle \mathsf{AcctID} = \text{A1},\ 1000 \leq \mathsf{TotalCost} \leq 1500,\ \mathsf{ApprvAcct} = \text{AM1}\rangle\\ & \ \vee\langle \mathsf{AcctID} = \text{A3},\ 1100 \leq \mathsf{TotalCost} \leq 1100,\ \mathsf{ApprvAcct} = \text{AM3}\rangle)\rangle\end{aligned}$$

Since AccountMgr is an extremal component, this lifting is transmitted back to component Accounting, thus completing not only Stage 1 but also Stage 2 for AccountMgr. This information requires that component Accounting trim its initial proposal to remove the possibility of using account A2. The following is computed as the final lifting in Accounting.

$$\begin{aligned}\mathbf{u}_{\mathsf{Actg}:1} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\ & (\ \langle \mathsf{AcctID} = \text{A1},\ 1000 \leq \mathsf{TotalCost} \leq 1500,\ \mathsf{ApprvAcct} = \text{AM1}\rangle\\ & \ \vee\langle \mathsf{AcctID} = \text{A3},\ 1000 \leq \mathsf{TotalCost} \leq 1100,\ \mathsf{ApprvAcct} = \text{AM3}\rangle)\\ & \qquad\qquad \cup \pm\langle \mathsf{Balance} \leftarrow \mathsf{Balance} - \mathsf{TotalCost}\rangle\rangle\end{aligned}$$

Component Accounting now projects this result back to its inner port defined by R_{SeAc}, thus completing its Stage 2.

The component Secretariat is still in Stage 1, and must communicate the initial update request to the other two manager components, ProjectMgr and DeptMgr. The project manager and department manager make only approve/disapprove decisions; no other parameters are involved. They are presented only with the proposed values for TripID, EmpID, ProjID, Purpose, StartDate, EndDate, and Location. They indicate approval by placing their IDs in the respective approval fields: ApprvProj or ApprvSup. For example, the update expression which is passed to the component ProjectMgr is

$$
\begin{aligned}
\mathbf{u}_{\mathsf{SePm:0}} := \langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\
& \mathsf{Purpose} = \text{“meet with project partners”},\\
& \mathsf{StartDate} = 01.04.10,\ \mathsf{EndDate} = 05.04.10,\\
& (\langle \mathsf{Location} = \text{Nantes}\rangle \vee \langle \mathsf{Location} = \text{Nice}\rangle),\\
& \mathsf{ApprvProj} = \text{Carl}\ \rangle
\end{aligned}
$$

Observe in particular that the location is given as either Nantes or else Nice. Even though there are no hotels available in Nice, for this simple model, the communication of component Secretariat with Hotel, Accounting, ProjectMgr, and DeptMgr occurs in parallel. Thus, it is not necessarily known that there are no hotels available in Nice when this update request is sent to ProjectMgr. Furthermore, even if Secretariat had received the reply from Hotel before initiating communication with ProjectMgr, it may not have the authority to pass this information along to that component. See 3.1 and 3.3 for a further discussion of this type of situation.

Returning to the communication with ProjectMgr, it indicates approval by returning this same expression, and indicates rejection by returning the empty expression. In either case, since it is an extremal component, returning the decision completes Stages 1 and 2 for it. An analogous expression applies for communication with the component DeptMgr. In the decision flow of this example, assume that both return positive decisions.

At this point the Secretariat component has received all of the responses, and is in a position to complete its Stage 2. To do this, it *merges* all of these responses to find a greatest common expression; that is, the largest update family which respects each of the update families which was reflected back to it. The expression which is obtained is the following.

$$
\begin{aligned}
\mathbf{u}_{\mathsf{Secrt:1}} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\
& \mathsf{Purpose} = \text{“meet with project partners”},\\
& \mathsf{StartDate} = 01.04.10,\ \mathsf{EndDate} = 05.04.10,\ \mathsf{Location} = \text{Nantes},\\
& \mathsf{ApprvSup} = \text{Barbara},\ \mathsf{ApprvProj} = \text{Carl},\\
& (\ \ \langle 1200 \le \mathsf{TotalCost} \le 1300,\ \mathsf{AcctID} = \text{A1},\ \mathsf{ApprvAcct} = \text{AM1},\\
& \qquad \mathsf{HotelCost} = 1200,\ \mathsf{HotelName} = \text{AssezCher}\rangle\\
& \ \ \vee\langle 1000 \le \mathsf{TotalCost} \le 1300,\ \mathsf{AcctID} = \text{A1},\ \mathsf{ApprvAcct} = \text{AM1},\\
& \qquad (\ \ \langle \mathsf{HotelCost} = 400,\ \mathsf{HotelName} = \text{PasCher}\rangle\\
& \qquad\ \ \vee\langle \mathsf{HotelCost} = 200,\ \mathsf{HotelName} = \text{Simple}\rangle)\rangle\\
& \ \ \vee\langle 1000 \le \mathsf{TotalCost} = 1100,\ \mathsf{AcctID} = \text{A3},\ \mathsf{ApprvAcct} = \text{AM3},\\
& \qquad (\ \ \langle \mathsf{HotelCost} = 400,\ \mathsf{HotelName} = \text{PasCher}\rangle\\
& \qquad\ \ \vee\langle \mathsf{HotelCost} = 200,\ \mathsf{HotelName} = \text{Simple}\rangle)\rangle)\rangle
\end{aligned}
$$

To complete Stage 2 for Secretariat, this expression is projected back to component Employee as the following. Note that details about approval and about which account can fund the trip are not included; such information is not part of the view for Employee.

$$\begin{aligned}\mathbf{u}_{\mathsf{Empl:1}} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\ & \mathsf{Purpose} = \text{“meet with project partners”},\\ & \mathsf{StartDate} = 01.04.10,\ \mathsf{EndDate} = 05.04.10,\ \mathsf{Location} = \text{Nantes},\\ & (\ \ \langle 1200 \leq \mathsf{TotalCost} \leq 1300,\\ & \qquad \mathsf{HotelCost} = 1200,\ \mathsf{HotelName} = \text{AssezCher}\rangle\\ & \ \ \vee\langle 1000 \leq \mathsf{TotalCost} \leq 1300,\\ & \qquad (\ \ \langle \mathsf{HotelCost} = 400,\ \mathsf{HotelName} = \text{PasCher}\rangle\\ & \qquad\ \ \vee\langle \mathsf{HotelCost} = 200,\ \mathsf{HotelName} = \text{Simple}\rangle)\rangle)\rangle\end{aligned}$$

This completes Stage 2 for Employee. Now, for Stage 3, Annie must choose one of the possibilities. If she decides to take as much travel funds as possible, namely 1300 Euros, she will have only 100 Euros left for the hotel. So, she chooses the hotel PasCher for 400 Euros. Because she is a very responsible person, and because the hotel is so inexpensive, she decides to take only 1100 Euros in total expenses, since 700 is more than enough to cover the other expenses. Her final, deterministic update request is thus the following.

$$\begin{aligned}\mathbf{u}_{\mathsf{Empl:2}} := +\langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French},\\ & \mathsf{Purpose} = \text{“meet with project partners”},\\ & \mathsf{StartDate} = 01.04.10,\ \mathsf{EndDate} = 05.04.10,\ \mathsf{Location} = \text{Nantes},\\ & \mathsf{TotalCost} = 1100,\ \mathsf{HotelCost} = 400,\ \mathsf{HotelName} = \text{PasCher}\rangle\end{aligned}$$

To complete Stage 3 for all components, this decision must be propagated to the other components, and then committed to the database. This is not quite trivial, because even though Annie has made a decision, there is still a choice to be made in another component. In this example, since she chose to take only 1100 Euros, either account A1 or account A3 may be charged. It is within the domain of the administrator who has update rights on the Accounting component to make this decision. In any case, the process of propagating the decision to the other components is again a simple project-lift process, which will not be elaborated further here. Once these decisions are made, the update may be committed to the database, completing Stage 3.

2.5. Analysis of the Three-Stage Negotiation Process

The process presented here is a very simple one. Basically, there are only two points at which an actor may make a decision. The first is during Stage 1, when the set of alternatives which the actor will accept is asserted. In effect, the actor agrees to support each of these alternatives for the life of the negotiation process. This stands in sharp contrast to the model forwarded in [15], in which an actor may at any time decide to withdraw alternatives which it previously agreed to support. Similarly, in Stage 3, an actor must decide which of the alternatives to support in the final update, but this is also a single decision which may not be modified once it is made. Stage 2 does not involve any decisions at all. Rather, its purpose is to merge the decisions made in Stage 1, and may be carried out in an entirely automated fashion, without any input at all from the actors. Again, this is in contrast to the approach of [15], in which the actors may examine the results of merging the previous results and make new decisions as to which alternatives to support and which to reject. The upshot is that the total number of steps required in the negotiation process is effectively independent of the number of alternatives considered.

In contrast, the process described in [15] will in the worst case require a number of steps proportional to the total number of alternatives possible for satisfying the update request. Of course, this reduction comes at the expense of some flexibility in the process itself, but for many applications it should be more than adequate.

The dominant cost for this approach is governed not by the number of decisions but rather by the resources required to specify and manage nondeterministic update specifications. This is indeed an important issue which requires further work. It may be addressed both by exploring efficient methods for representing such specifications, as discussed in Section 4.2, and by controlling the number of such alternatives and the ways in which they are propagated, as discussed further in Sections 3.1 and 3.2. However, the point is that with the approach to negotiation presented here, the evolution of that process itself is not the bottleneck.

3. Further Modelling Issues for Cooperative Update

In describing the update and negotiation process via the running example of Section 2, some issues were glossed over in the interest of not clouding the main ideas with details. In this section, some of these more important details are elaborated. On the other hand, issues which are not addressed at all in this paper, such as concurrency control, are discussed in 4.2.

3.1. Context Sensitivity of the Lifting Strategy

In the lifting $\mathbf{u}_{actg:0}$ in the example of Section 2, employee Annie made a request to travel either to Nantes or else to Nice for the French project, and department manager Barbara approved this request. However, suppose that Barbara had instead rejected this request, but would have approved a reduced request which includes only the possibility to travel to Nantes, but not to Nice. In other words, she would reject the request to travel to Nantes were it accompanied by an alternative to travel to Nice, but not if Nantes were given as the sole possibility for the destination. In this case, it is said that her decision is *context sensitive*. Although context-sensitive lifting behavior might seem less than completely rational, it must be acknowledged that human actors may sometimes exhibit such characteristics in their decision making.

This work is not primarily about modelling human decision makers. However, context sensitivity in lifting behavior does have important implications. Suppose that, for efficiency purposes, the component Secretariat were allowed to check hotel availability before forwarding travel requests on to the managers. In that case, since no hotel is available in Nice for the requested time period, the department manager would not see that Annie had requested also to travel to that city, since that information would be filtered out before being transmitted to DeptMgr. Thus, Barbara would see only the request to travel to Nantes, and so would approve it. In this case, whether or not the travel request is approved depends upon the order in which impossibilities are filtered out. On the other hand, if Barbara exhibited a *context-free* decision behavior; that is, if whether she would approve the trip to Nantes were independent of any other requests which Annie had made, allowing the Secretariat to check hotel availability before forwarding the request on to the managers would not affect the final outcome.

It is important to emphasize that this notion of context sensitivity relates to alternatives in the update family, and not upon conjunctive combinations. For example, if the request of Annie contained two alternatives, one to travel just to Nantes, and a second to travel both to Nantes and to Nice, then to approve the travel to Nantes, but not the combined travel to both Nantes and Nice would be perfectly context free. Context sensitivity has only to do with rejecting a given alternative on the grounds of the presence of other alternatives.

3.2. Admissibility for the Lifting Strategy

In Stage 1 of the negotiation process, the liftings should be *minimal* in the sense that they do not make any changes which are not essential to the update request. Within the limited framework of the running example, it is difficult to illustrate liftings which are not minimal. However, suppose that the component DeptMgr contains an additional relation $S_{\mathsf{Budget}}(\underline{\mathsf{DeptID}}, \mathsf{Amount})$ which represents the department budget, and this component is connected to an additional component UpperMgt representing upper management, as illustrated in Figure 4.

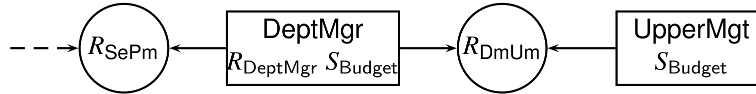

Figure 4. Additional component for rider update

Now, suppose that in approving the travel for the trip of Annie, the department manager also adds an increase of 100000 Euros to the department budget to the lifting, so that it becomes

$$\begin{aligned}\mathbf{u}_{\mathsf{DeptMgr:0'}} := \langle & \mathsf{TripID} = 12345,\ \mathsf{EmpID} = \text{Annie},\ \mathsf{ProjID} = \text{French}, \\ & \mathsf{Purpose} = \text{“meet with project partners”}, \\ & \mathsf{StartDate} = 01.04.10,\ \mathsf{EndDate} = 05.04.10, \\ & (\langle \mathsf{Location} = \text{Nantes}\rangle \vee \langle \mathsf{Location} = \text{Nice}\rangle), \\ & \mathsf{ApprvProj} = \text{Carl}\rangle \\ & \cup \langle \mathsf{DeptID} = \text{CDpt},;\ \mathsf{Amount} \leftarrow \mathsf{Amount} + 100000\rangle\end{aligned}$$

Here Carl has added a *rider* to the update request; to be approved, an additional update which is irrelevant to the original request must be realized as well. This lifting is not minimal because the rider could be removed without compromising support for the original update request.

It may not always be possible to characterize minimality of a lifting in terms of inserting and deleting the minimal number of tuples. There might be a situation, such as a funds transfer, in which the amount should be minimal. However, the principle remains clear.

3.3. The Model of Authority

A suitable framework for describing and managing access rights in the context of cooperative update requires certain special features beyond those of conventional database systems, since traditional access rights do not take into account any form of cooperation.

One suitable model builds upon the widely-used notion of *rôle-based access control*, which was introduced in [1] using the terminology *named protection domain* or *NPD*, and which is elaborated more fully in articles such as [20]. The key idea is that rights are assigned not to individual users, but to *rôles*. Each user may have one or more rôles, and each rôle may have one or more users as members. For example, Barbara may have the rôle of manager of the French project, but she may also be an ordinary employee when making a travel request for herself.

In addition to the usual privileges hierarchy, in which $A \leq B$ means that B has all privileges which A has, there is a *authority hierarchy*, in which $A \leq B$ means that A must support fully the requests of B. A possible authority hierarchy for the example of Section 2 might be the following, in which the ordering is represented from left to right.

The employee Annie might make the travel request from the component Employee in the rôle of Scientist, in which case someone (or something — a program perhaps) in the rôle of *Secretary* using the component Secretariat and someone/something in the rôle of TravelAgent using the component Hotel would need to respect the update request of Annie, but those assuming the rôles of Accountant or of Manager (in the components with corresponding names) would have the right to trim her request as they see fit. This is only a sketch of how the model of authority works; the details will appear in a forthcoming paper.

3.4. The Model of Representation and Computation

The representation of update families, and the computations involved in lifting and merging them, are illustrated via example in Section 2, with the basic ideas hopefully clear. It is nevertheless appropriate to provide a bit more information as to what is allowed. First of all, update families are generally taken to be finite; that is, they represent only a finite number of alternatives. This means that, at least in theory, the liftings of Stage 1 of the negotiation process can be computed on a case-by-case basis. Consider the initial update request $\mathbf{u}_{\mathsf{Empl:0}}$ of 2.2. While the ranges on values for TotalCost and HotelCost are finite, the ranges for HotelName is specified by a wildcard and thus appear to be unconstrained. However, it is assumed that there are only a finite number of hotels, so this range may be taken to be finite.

A second, computational issue arises in the context of computing merges in Stage 2 of the negotiation process. Here the set of liftings which agree with the update requests on each of several ports must be computed. In the most general case, this is an unsolvable problem. There is nevertheless a very natural case in which such problems do not arise. If the port views are defined by basic SPJ (select-project-join) queries, and if the schema

has the *finite-extension property* [13, Def. 28]; that is, if the classical chase procedure [9] always terminates with a finite structure, then the merger can be computed as the result of the chase. Of course, there will be one such chase for each set of alternatives in the respective update families, but the total number of such alternatives is finite. In [19], many cases which guarantee such termination, and thus the semantic-extension property, are identified. Included in these is the classical situation of schemata constrained by functional dependencies and unary inclusion dependencies (which include in particular foreign-key dependencies), provided that the latter have the property of being acyclic [7].

The bottom line is that, from a theoretical standpoint, there are no problems with representation and computation. However, further work is needed to identify suitable cases which are both useful and efficiently solvable. See 4.2 for a further discussion.

4. Conclusions and Further Directions

4.1. Conclusions

A straightforward but useful model of negotiation for cooperative update on database schemata defined by components has been presented. In contrast to the approach given in [15], the method presented here involves only three simple stages for each component and thus terminates rapidly. The key idea is that decisions are made only during the first stage; thereafter the operations involve only merging those decisions and then selecting one of them as the final result. Other aspects of the modelling process, such as the representation of update requests, have been illustrated via a detailed example. This has illustrated that, at least for some examples, such representation is a viable alternative to more traditional, task-based representations. Nevertheless, there are many issues which remain to be solved before the ideas can be put into practice.

4.2. Further Directions

Relationship to workflow and business-process modelling formalisms The kinds of applications which can be modelled effectively via cooperative update overlap in substantial part those which are typically modelled using workflow [26] and/or business-process modelling languages [4]. Furthermore, some database transaction models, such as the ConTract model [27], [21], are oriented towards modelling these sorts of processes. Relative to all of these, the cooperative update approach developed here is constraint based, in that it does not specify any flow of control explicitly; rather, it places constraints on what that flow may be. The identification of workflow and business-process representations for those flows of control which are representable by cooperative update, as well as a way to translate between the various representations, is an important direction which warrants further investigation.

An appropriate model of concurrency control Update requests to databases, whether cooperative or not, typically overlap, thus requiring some form of concurrency control. However, traditional approaches are generally inadequate for cooperative update. Since they typically involve at least some human interaction, cooperative update processes are by their very nature long running, and so locking large parts of the database in order to avoid unwanted interaction of distinct transactions is not a feasible solu-

tion. On the other hand, cooperative transactions typically involve changes to only a very small part of the overall database. Work is currently underway on a non-locking approach which uses information contained in the initial update request to identify tight bounds on the part of the database which must be protected during a cooperative transaction [14].

A distributed model of control and communication The operation of a database system constructed from schema components, particularly in the context of cooperative updates, involves the passing of messages (*i.e.*, projections and liftings) from component to component. Thus, a unified model of control and communication which is distributed amongst the components is essential to an effective realization of systems with this architecture. Future work will look at the properties and realization of such models.

An efficient representation for nondeterministic update families This issue has already been discussed briefly in 3.4. Work is currently underway in two areas. The first is to identify economical and computationally flexible representations for nondeterministic update families. The second is to identify ways of computing merges of such nondeterministic update families using only one, or at least relatively few, instances of the chase procedure.

More complex models of negotiation The model of negotiation which has been developed and presented in this paper is a very simple one. Although it is useful in modelling many business processes, there is clearly also a need for more complex negotiation processes, particularly ones with a back-and-forth nature in which parties compromise to reach a decision. Future work will look at such general notions of negotiation.

Acknowledgments

For three to four months each year from 2005-2008, the author was a guest researcher at the Information Systems Engineering Group at Christian-Albrechts-Universität zu Kiel, and many of the ideas in this paper were developed during that time. He is particularly indebted to Bernhard Thalheim for suggesting the idea that his ideas of database components and the author's work on views and view updates could have a fruitful intersection, as well as for inviting him to work with his group on this problem. He is furthermore indebted to Peggy Schmidt, for countless discussions and also for fruitful collaboration on the ideas of schema components. She furthermore read initial drafts of this paper and made several insightful comments.

References

[1] R. W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *Proc. 1990 IEEE Symposium on Research in Security and Privacy*, pages 116–132. IEEE Computer Society Press, 1990.

[2] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Systems*, 6:557–575, 1981.

[3] G. Beneken, U. Hammerschall, M. Broy, M. V. Cengarle, J. Jürjens, B. Rumpe, and M. Schoenmakers. Componentware - State of the Art 2003. In *Proceedings of the CUE Workshop Venedig*, 2003.

[4] Business process modeling notation v1.1. http://www.omg.org/spec/BPMN/1.1/PDF, 2008.
[5] M. Broy. A logical basis for modular software and systems engineering. In B. Rovan, editor, *SOFSEM*, volume 1521 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 1998.
[6] M. Broy. Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones. *Innovations Syst. Softw. Eng.*, 3(1):75–102, 2007.
[7] S. S. Cosmadakis and P. C. Kanellakis. Functional and inclusion dependencies. *Advances in Computing Research*, 3:163–184, 1986.
[8] A. Eisenberg, J. Melton, K. G. Kulkarni, J.-E. Michels, and F. Zemke. SQL:2003 has been published. *SIGMOD Record*, 33(1):119–126, 2004.
[9] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. *Theoret. Comput. Sci.*, 336:89–124, 2005.
[10] G. Fiedler, H. Jaakkola, T. Mäkinen, B. Thalheim, and T. Varkoi. Co-design of Web information systems supported by SPICE. In Y. Kiyoki, T. Tokuda, H. Jaakkola, X. Chen, and N. Yoshida, editors, *Information Modelling and Knowledge Bases XX, 18th European-Japanese Conference on Information Modelling and Knowledge Bases (EJC 2008), Tsukuba, Japan, June 2-6, 2008*, volume 190 of *Frontiers in Artificial Intelligence and Applications*, pages 123–138. IOS Press, 2008.
[11] S. J. Hegner. An order-based theory of updates for closed database views. *Ann. Math. Art. Intell.*, 40:63–125, 2004.
[12] S. J. Hegner. A model of database components and their interconnection based upon communicating views. In H. Jakkola, Y. Kiyoki, and T. Tokuda, editors, *Information Modelling and Knowledge Systems XIX*, Frontiers in Artificial Intelligence and Applications, pages 79–100. IOS Press, 2008.
[13] S. J. Hegner. Internal representation of database views. Submitted for publication, 2010.
[14] S. J. Hegner. A model of independence and overlap for transactions on database schemata. Submitted for publication, 2010.
[15] S. J. Hegner and P. Schmidt. Update support for database views via cooperation. In Y. Ioannis, B. Novikov, and B. Rachev, editors, *Advances in Databases and Information Systems, 11th East European Conference, ADBIS 2007, Varna, Bulgaria, September 29 - October 3, 2007, Proceedings*, volume 4690 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 2007.
[16] G. E. Kaiser. Cooperative transactions for multiuser environments. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pages 409–433. ACM Press and Addison-Wesley, 1995.
[17] M. Kifer, A. Bernstein, and P. M. Lewis. *Database Systems: An Application-Oriented Approach*. Addison-Wesley, second edition, 2006.
[18] L. Kot and C. Koch. Cooperative update exchange in the Youtopia system. *Proc. VLDB Endow.*, 2(1):193–204, 2009.
[19] M. Meier, M. Schmidt, and G. Lausen. On chase termination beyond stratification. *CoRR*, abs/0906.4228, 2009.
[20] S. L. Osborn and Y. Guo. Modeling users in role-based access control. In *ACM Workshop on Role-Based Access Control*, pages 31–37, 2000.
[21] A. Reuter and F. Schwenkreis. ConTracts – a low-level mechanism for building general-purpose workflow management-systems. *IEEE Data Eng. Bull.*, 18(1):4–10, 1995.
[22] M. C. Sampaio and S. Turc. Cooperative transactions: A data-driven approach. In *29th Annual Hawaii International Conference on System Sciences (HICSS-29), January 3-6, 1996, Maui, Hawaii*, pages 41–50. IEEE Computer Society, 1996.
[23] B. Thalheim. Database component ware. In K.-D. Schewe and X. Zhou, editors, *Database Technologies 2003, Proceedings of the 14th Australasian Database Conference, ADC 2003, Adelaide, South Australia, February 2003*, volume 17 of *CRPIT*, pages 13–26. Australian Computer Society, 2003.
[24] B. Thalheim. Co-design of structuring, functionality, distribution, and interactivity for information systems. In S. Hartmann and J. F. Roddick, editors, *APCCM*, volume 31 of *CRPIT*, pages 3–12. Australian Computer Society, 2004.
[25] B. Thalheim. Component development and construction for database design. *Data Knowl. Eng.*, 54(1):77–95, 2005.
[26] W. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
[27] H. Wächter and A. Reuter. The ConTract model. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann, 1992.

[28] W. Wieczerzycki. Multiuser transactions for collaborative database applications. In G. Quirchmayr, E. Schweighofer, and T. J. M. Bench-Capon, editors, *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings*, volume 1460 of *Lecture Notes in Computer Science*, pages 145–154. Springer, 1998.