# Update Support for Database Views Via Cooperation

Stephen J. Hegner[1] and Peggy Schmidt[2]

[1] Umeå University, Department of Computing Science
SE-901 87 Umeå, Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner
[2] Christian-Albrechts-University Kiel, Department of Computer Science
Olshausenstraße 40, D-24098 Kiel, Germany
pesc@is.informatik.uni-kiel.de
http://www.is.informatik.uni-kiel.de/~pesc

**Abstract.** Support for updates to views of database schemata is typically very limited; only those changes which can be represented entirely within the view, or changes which involve only generic changes outside of the view, are permitted. In this work, a different point of view towards the view-update problem is taken. If a proposed update cannot be performed within the view, then rather than rejecting it outright, the cooperation of other views is sought, so that in their combined environments the desired changes can be realized. This approach has not only the advantage that a wider range of updates are supported than is possible with more traditional approaches, but also that updates which require the combined access privileges of several users are supported.

**Keywords:** update, view.

## 1 Introduction

Support for updates to database views has long been recognized as a difficult problem. An update which is specified on a view provides only partial information on the change of state of the main schema; the complementary information necessary to define a complete *translation* of that update to the main schema must be determined in other ways. Over the years, a number of approaches have been developed for such translations. In the *constant-complement strategy*, first defined in [1] and later refined in [2] and [3], the fundamental idea is that the translation must leave unaltered all aspects of the main database which are not visible from the view; formally, the so-called *complementary view* is held constant. Theoretically, it is the cleanest approach, in that it defines precisely those translations which are free from so-called *update anomalies* [2, 1.1] which involve changes to the database which are not entirely visible within the view itself. Unfortunately, the family of anomaly-free updates is relatively limited, and for this reason the constant-complement strategy has

Y. Ioannidis, B. Novikov, and B. Rachev (Eds.): ADBIS 2007, LNCS 4690, pp. 98–113, 2007.

been viewed as inadequate by some investigators [4], and so numerous more liberal approaches have been forwarded, including both direct approaches [5] and those which relax some, but not all, of the constraints of the constant-complement strategy [6]. All of these more liberal approaches involve, in one way or another, updates to the main schema which are not visible within the view itself.

Even if one accepts that view update strategies which are more liberal than the constant-complement approach are necessary and appropriate, there is a significant further issue which must be taken into consideration — access rights. It is a fundamental design principle of modern database systems that users have access rights, and that all forms of access, both read and write, must respect the authorization of those rights. With the constant-complement update strategy, in which only those parts of the main schema which are visible in the view may be altered in an update, this issue poses no additional problems beyond those of specifying properly the access rights on each view. However, with a more liberal update approach, changes to the main schema may be mandated which are not visible within the view. This implies that the user of the view must have write access privileges beyond that view, which is often unrealistic. Thus, even if one is willing to accept some update anomalies, view update support beyond the constant-complement strategy is to a large extent unacceptable because of the serious problems surrounding access rights which it implies.

To address these concerns, a quite different approach to supporting view updates is proposed in this paper. When an update $u$ to view $\Gamma$ cannot be supported by the constant-complement strategy, the *cooperation* of other views is enlisted. If translation of $u$ implies that an update to the main schema must be made which is not visible within $\Gamma$, then these additional changes must be embodied in the cooperating views. If the user of $\Gamma$ who desires to effect $u$ does not have the necessary access privileges on the cooperating views, then the cooperation of suitable users of these views must also be enlisted, in order to effect the update "in unison". This, in turn, provides information on the workflow pattern which is necessary to realize the update.

For such a theory of cooperation to take form, it is necessary to be able to regard a database schema as a collection of interconnected views. The fundamental ideas of such representations of database schemata are found in *component-based modelling*, as forwarded by Thalheim [7] [8] [9]. Roughly speaking, a component is an encapsulated database schema, together with *channels* which allow it to be connected to other components. A database schema is then modelled as an interconnection of such components. The work of Thalheim is, in the first instance, oriented towards conceptual modelling and the design of database schemata using the *higher-order entity-relationship model (HERM)* [10]. In [11], the ideas of component-based modelling have been recast and formalized in a way which makes them more amenable to view-update problems. It is this latter work which is used, in large part, as the basis of this paper.

## 2 The Core Concepts by Example

To present the ideas underlying cooperative updates in a complete and unambiguous fashion, a certain amount of formalism is unavoidable. However, it is possible to illustrate many of the key ideas with a minimum of formalism; such an illustration, via a running example, is the goal of this section. First, the main ideas of database components will be illustrated via an example, with that same example then used as the basis for the illustration of a cooperative update.

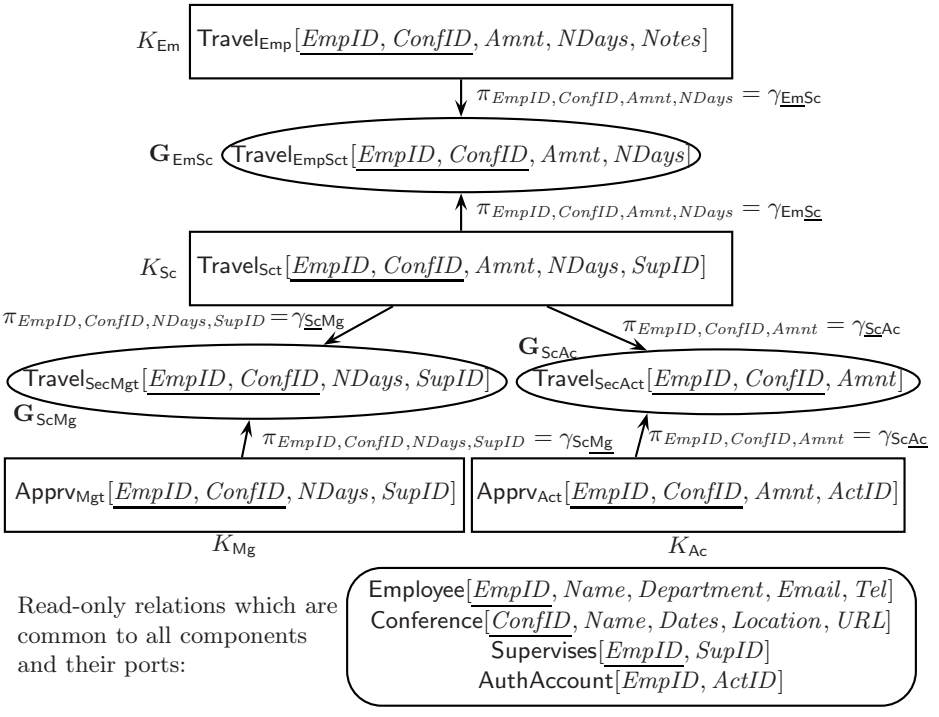


**Fig. 1.** Graphical depiction of the components of the running example

**Discussion 2.1 (An informal overview of database components).** For a much more thorough and systematic presentation of the ideas underlying the database components of this paper, the reader is referred to [11]. A *component* is an ordered pair $C = (\mathsf{Schema}(C), \mathsf{Ports}(C))$ in which $\mathsf{Schema}(C)$ is a database schema and $\mathsf{Ports}(C)$ is a finite set of nonzero views of $\mathsf{Schema}(C)$, called the *ports* of $C$. The running relational example is depicted in Fig. 1; there are four components, the *employee component* $K_{\mathsf{Em}}$, the *secretariat component* $K_{\mathsf{Sc}}$, the *management component* $K_{\mathsf{Mg}}$, and the *accounting component* $K_{\mathsf{Ac}}$. The relation unique to the schema of a given component is shown enclosed in a rectangle. In addition, there is a set of relations which are common to all components; these

are shown in a box with rounded corners at the bottom of the figure. For example, the schema of $K_{\mathsf{Em}}$ consists of the relations $\mathsf{Travel}_{\mathsf{Emp}}$, Employee, Conference, Supervises, and AuthAccount. The primary key of each relation is underlined; in addition, the following inclusion dependencies are assumed to hold:

$$\mathsf{Travel}_{\mathsf{Emp}}[\mathit{EmpID}] \subseteq \mathsf{Employee}[\mathit{EmpID}],$$
$$\mathsf{Travel}_{\mathsf{Sct}}[\mathit{EmpID}, \mathit{SupID}] \subseteq \mathsf{Supervises}[\mathit{EmpID}, \mathit{SupID}],$$
$$\mathsf{Apprv}_{\mathsf{Mgt}}[\mathit{EmpID}, \mathit{SupID}] \subseteq \mathsf{Supervises}[\mathit{EmpID}, \mathit{SupID}],$$
$$\mathsf{Apprv}_{\mathsf{Act}}[\mathit{EmpID}, \mathit{ActID}] \subseteq \mathsf{AuthAccount}[\mathit{EmpID}, \mathit{ActID}],$$
$$\mathsf{Supervises}[\mathit{EmpID}] \subseteq \mathsf{Employee}[\mathit{EmpID}],$$
$$\mathsf{Supervises}[\mathit{SupID}] \subseteq \mathsf{Employee}[\mathit{EmpID}],$$
$$\mathsf{AuthAccount}[\mathit{EmpID}] \subseteq \mathsf{Employee}[\mathit{EmpID}].$$

The ports of the components are also represented in Fig. 1. The schema of each view (qua port) $\Gamma_x$ is represented within an ellipse, labelled with the name of that schema. The associated view mapping $\gamma_x$ for each port is shown next to the arrow which runs from the component schema to the port schema. For example, the component $K_{\mathsf{Em}}$ has only one port, which is denoted by $\Gamma_{\underline{\mathsf{Em}}\mathsf{Sc}} = (\mathbf{G}_{\mathsf{Em}\mathsf{Sc}}, \gamma_{\underline{\mathsf{Em}}\mathsf{Sc}})$, with $\gamma_{\underline{\mathsf{Em}}\mathsf{Sc}}$ the projection $\pi_{\mathit{EmpID},\mathit{ConfID},\mathit{Amnt},\mathit{NDays}} : \mathsf{Schema}(K_{\mathsf{Em}}) \to \mathbf{G}_{\mathsf{Em}\mathsf{Sc}}$. In other words, $\mathsf{Ports}\langle K_{\mathsf{Em}}\rangle = \{\Gamma_{\underline{\mathsf{Em}}\mathsf{Sc}}\}$. The component $K_{\mathsf{Sc}}$, on the other hand, has three ports, $\Gamma_{\mathsf{Em}\underline{\mathsf{Sc}}} = (\mathbf{G}_{\mathsf{Em}\mathsf{Sc}}, \gamma_{\mathsf{Em}\underline{\mathsf{Sc}}})$, $\Gamma_{\underline{\mathsf{Sc}}\mathsf{Mg}} = (\mathbf{G}_{\mathsf{Sc}\mathsf{Mg}}, \gamma_{\underline{\mathsf{Sc}}\mathsf{Mg}})$, and $\Gamma_{\underline{\mathsf{Sc}}\mathsf{Ac}} = (\mathbf{G}_{\mathsf{Sc}\mathsf{Ac}}, \gamma_{\underline{\mathsf{Sc}}\mathsf{Ac}})$, with the port definitions as given in the figure. Similarly, the component $K_{\mathsf{Mg}}$ has one port, $\Gamma_{\mathsf{Sc}\underline{\mathsf{Mg}}} = (\mathbf{G}_{\mathsf{Sc}\mathsf{Mg}}, \gamma_{\mathsf{Sc}\underline{\mathsf{Mg}}})$, and the component $K_{\mathsf{Ac}}$ has one port, $\Gamma_{\mathsf{Sc}\underline{\mathsf{Ac}}} = (\mathbf{G}_{\mathsf{Sc}\mathsf{Ac}}, \gamma_{\mathsf{Sc}\underline{\mathsf{Ac}}})$. The underlying *interconnection family*, which describes which ports are connected to which, is $\{\{\Gamma_{\underline{\mathsf{Em}}\mathsf{Sc}}, \Gamma_{\mathsf{Em}\underline{\mathsf{Sc}}}\}, \{\Gamma_{\underline{\mathsf{Sc}}\mathsf{Mg}}, \Gamma_{\mathsf{Sc}\underline{\mathsf{Mg}}}\}, \{\Gamma_{\underline{\mathsf{Sc}}\mathsf{Ac}}, \Gamma_{\mathsf{Sc}\underline{\mathsf{Ac}}}\}\}$. Each member of this family is called a *star interconnection*. The names of ports may be arbitrary, although for convenience in this example a special naming convention has been used. The view $\Gamma_{\underline{xy}} = (\mathbf{G}_{xy}, \gamma_{\underline{xy}})$ is a port of the component $K_{\mathsf{x}}$, and is connected to the port $\Gamma_{x\underline{y}} = (\mathbf{G}_{xy}, \gamma_{x\underline{y}})$ of the component $K_{\mathsf{y}}$.

The ports of connected components must have *identical* (and not just isomorphic) schemata. It becomes clear why this is necessary when defining the state of a combined interconnection. Let $M = (M_{\mathsf{Em}}, M_{\mathsf{Sc}}, M_{\mathsf{Mg}}, M_{\mathsf{Ac}}) \in \mathsf{LDB}(\mathsf{Schema}(K_{\mathsf{Em}})) \times \mathsf{LDB}(\mathsf{Schema}(K_{\mathsf{Sc}})) \times \mathsf{LDB}(\mathsf{Schema}(K_{\mathsf{Mg}})) \times \mathsf{LDB}(\mathsf{Schema}(K_{\mathsf{Ac}}))$, with $\mathsf{LDB}(-)$ denoting the set of legal databases. For $M$ to be a legal state of the interconnected component, the local states must agree on all ports. More precisely, it is necessary that the following hold: $\gamma_{\underline{\mathsf{Em}}\mathsf{Sc}}(M_{\mathsf{Em}}) = \gamma_{\mathsf{Em}\underline{\mathsf{Sc}}}(M_{\mathsf{Sc}})$, $\gamma_{\underline{\mathsf{Sc}}\mathsf{Mg}}(M_{\mathsf{Sc}}) = \gamma_{\mathsf{Sc}\underline{\mathsf{Mg}}}(M_{\mathsf{Mg}})$, and $\gamma_{\underline{\mathsf{Sc}}\mathsf{Ac}}(M_{\mathsf{Sc}}) = \gamma_{\mathsf{Sc}\underline{\mathsf{Ac}}}(M_{\mathsf{Ac}})$.

Some difficulties can arise if the underlying hypergraph is cyclic [11, 3.2]; i.e., if there are cycles in the connection. While these can often be overcome, the details become significantly more complex. Therefore, in this paper, it will always be assumed that the interconnections are acyclic.

**Example 2.2 (An example of cooperative update).** Suppose that Lena is an employee, and that she wishes to travel to a conference. The successful approval of such a request is represented by the insertion of an appropriate tuple

in the relation $\mathsf{Travel}_{\mathsf{Emp}}$. She has insertion privileges for the relation $\mathsf{Travel}_{\mathsf{Emp}}$ for tuples with her *EmpID* (which is assumed to be Lena, for simplicity); however, these privileges are qualified by the additional requirement that the global state of the interconnected components be consistent. Thus, any insertion into $\mathsf{Travel}_{\mathsf{Emp}}$ must be matched by a corresponding tuple in $\mathsf{Travel}_{\mathsf{Sct}}$ — that is, a tuple whose projection onto $\mathsf{Travel}_{\mathsf{EmpSct}}$ matches that of the tuple inserted into $\mathsf{Travel}_{\mathsf{Emp}}$. This tuple must in turn be matched by corresponding tuples in $\mathsf{Apprv}_{\mathsf{Mgt}}$ and $\mathsf{Apprv}_{\mathsf{Act}}$. Thus, to accomplish this update, Lena requires the cooperation someone authorized to update the component $K_{\mathsf{Sc}}$, which in turn requires the cooperation of those authorized to update $K_{\mathsf{Mg}}$ and $K_{\mathsf{Ac}}$.

The process of cooperative update proceeds along a project-lift cycle. The update request of Lena is *projected* (see Definition 3.2) to the ports of $K_{\mathsf{Em}}$; the connected components (in this case just $K_{\mathsf{Sc}}$) then *lift* (see Definition 3.3) these projections to their schemata. The process then continues, with $K_{\mathsf{Sc}}$ projecting its proposed update to $K_{\mathsf{Mg}}$ and $K_{\mathsf{Sc}}$. These projections and liftings cannot modify the state of the database immediately, as they are only proposed updates until all parties have agreed. Rather, a more systematic process for managing the negotiation process is necessary. The process is controlled by a nondeterministic automaton, which maintains key information about the negotiation and determines precisely which actions may be carried out by components (and their associated actors) at a given time. It also manages the actual update of the database when a successful negotiation has been completed. This automaton is described formally in Definition 3.4. In this section, it will be described more informally, and consequently somewhat incompletely, by example.

The central data structures of this automaton are two sets of registers for managing proposed updates. For each component $C$ there is a *pending-update register* $\mathsf{PendingUpdate}(C)$ which records proposed updates which are initiated by that component, but not yet part of the permanent database. In addition, for each component $C$ and each port $\Gamma$ of $C$, there is a *port-status registers* $\mathsf{PortStatus}(C, \Gamma)$ which is used to record projections of updates received by neighboring components. In addition, for each component $C$, the register $\mathsf{CurrentState}(C)$ records the actual database state for that component. The automaton also has a Status, which indicates the phase of the update process in which the machine lies. In *Idle*, it is waiting for an initial update request from one of the components. In *Active*, it is processing such a request by communication and refinement; the bulk of the processing occurs in this state. The value *Accepted* indicates that all components have agreed on a suitable set of updates. In the *Final* phase, one of these updates is selected, again by propagating proposals amongst the components. Finally, the value of the variable Initiator identifies the component which initiated the update request. A state of this update automaton is given by a value for each of its state variables. These variables, together with their admissible and initial values, are shown in Table 1.

In Table 2, a sequence of twelve steps which constitutes a successful realization of a travel request from Lena is shown. An attempt has been made to represent all essential information, albeit in compact format. Notation not already described,

**Table 1.** The state variables of the update automaton

| Name | Range of values | Initial Value |
|---|---|---|
| Status | $\in \{$ *Idle*, *Active*, *Accepted*, *Final* $\}$ | *Idle* |
| Initiator | $\in X \cup \{$NULL$\}$ | NULL |
| For each component $C$: | | |
| CurrentState($C$) | $\in$ LDB(Schema$\langle C \rangle$) | $(M_C)_0$ |
| PendingUpdate($C$) | $\in$ RDUpdates(Schema$\langle C \rangle$) $\cup \{$NULL$\}$ | NULL |
| For each component $C$ and $\Gamma \in$ Ports$\langle C \rangle$: | | |
| PortStatus($C, \Gamma$) | $\in$ RDUpdates(Schema$\langle \Gamma \rangle$) $\cup \{$NULL$\}$ | NULL |

such as the step of the update automaton which is executed, will be described as the example proceeds. The formal descriptions of these steps may be found in Definition 3.4.

To understand how this request is processed, it is first necessary to expand upon the request itself, which involves alternatives. Suppose that Lena wishes to travel either to ADBIS or else to DEXA. For ADBIS, she requires a minimum of €800; for DEXA €1000. For ADBIS she needs to travel for at least five days; for DEXA only three. Although she is flexible, she also has preferences. A trip to ADBIS is to be preferred to a trip to DEXA, and within a given conference, more money and time is always to be preferred. To express these alternatives, a *ranked directional update* (see Definition 3.1) is employed. It is *directional* in the sense that it is either an insertion or a deletion (for technical reasons, only insertions and deletions are supported in the current framework), and it is *ranked* in the sense that there is a partial order which expresses preferences on the updates. The update request, call it $\mathbf{u}_0$, would then consist of all tuples of the form $\mathsf{Travel}_{\mathsf{Emp}}[\text{Lena}, \text{ADBIS}, e_A, d_A, \eta]$ together with those of the form $\mathsf{Travel}_{\mathsf{Emp}}[\text{Lena}, \text{DEXA}, e_D, d_D, \eta]$, with $800 \leq e_A \leq 2000$, $5 \leq d_A \leq 10$, $1000 \leq e_D \leq 2000$, $3 \leq d_D \leq 10$. Here $\eta$ denotes a null value for the *Notes* field. There is also a technical requirement that a set of ranked updates be finite; hence the upper bounds on the values for time and money. The set of all possible ranked directed updates on a schema $\mathbf{D}$ is denoted RDUpdates($\mathbf{D}$). At the end of a successful update negotiation, all parties will agree to support some subset of the elements of $\mathbf{u}_0$, with Lena then choosing one of them.

In processing this request, the automaton is initially in the state *Idle*, awaiting an update request from some component. This is illustrated in step 0 of Table 2. Lena initiates her travel request by executing InitiateUpdate($K_{\mathsf{Em}}, \mathbf{u}_0$), which communicates the projection $\mathsf{Proj}(\mathbf{u}_0,)$ of $\mathsf{Travel}_{\mathsf{Emp}}$ onto the common port relation $\mathsf{Travel}_{\mathsf{SecMgt}}$, as shown in step 1 of the table. Because this is only an update request, and not an actual update, it is placed in the appropriate port status register, in this case the status register for $\Gamma_{\mathsf{Em}\underline{\mathsf{Sc}}}$; the database state remains unchanged. A value for a state variable of the form PortStatus($C, \Gamma$) represents

**Table 2.** The state evolution of Example 2.2

| $K_{\mathsf{Em}}$ | | $K_{\mathsf{Sc}}$ | | | | $K_{\mathsf{Mg}}$ | | $K_{\mathsf{Ac}}$ | |
|---|---|---|---|---|---|---|---|---|---|
| Pending Update $K_{\mathsf{Em}}$ | Port Status $\Gamma_{\underline{\mathsf{Em}\mathsf{Sc}}}$ | Pending Update $K_{\mathsf{Sc}}$ | Port Status $\Gamma_{\underline{\mathsf{Em}\mathsf{Sc}}}$ | Port Status $\Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}}$ | Port Status $\Gamma_{\underline{\mathsf{Sc}\mathsf{Ac}}}$ | Pending Update $K_{\mathsf{Mg}}$ | Port Status $\Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}}$ | Pending Update $K_{\mathsf{Ac}}$ | Port Status $\Gamma_{\underline{\mathsf{Sc}\mathsf{Ac}}}$ |
| 0. Initial database state $= (M_{\mathsf{Em}}, M_{\mathsf{Sc}}, M_{\mathsf{Mg}}, M_{\mathsf{Ac}})$ Status: *Idle* Initiator: NULL | | | | | | | | | |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL |
| 1. InitiateUpdate$(K_{\mathsf{Em}}, \mathbf{u}_0)$ Status: *Active* Initiator: $K_{\mathsf{Em}}$ $\mathbf{u}'_0 = \mathsf{Proj}(\mathbf{u}_0, \Gamma_{\underline{\mathsf{Em}\mathsf{Sc}}})$ | | | | | | | | | |
| $\mathbf{u}_0$ | NULL | NULL | $\mathbf{u}'_0$ | NULL | NULL | NULL | NULL | NULL | NULL |
| 2. PromoteInitialUpdate$(K_{\mathsf{Sc}})$ Status: *Active* Initiator: $K_{\mathsf{Em}}$ $\mathbf{u}_1 \in \mathsf{NERestr}(\mathsf{MinLift}(\mathbf{u}'_0, \Gamma_{\underline{\mathsf{Em}\mathsf{Sc}}}, M))$ $\mathbf{u}'_1 = \mathsf{Proj}(\mathbf{u}_1, \Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}})$ $\mathbf{u}''_1 = \mathsf{Proj}(\mathbf{u}_1, \Gamma_{\underline{\mathsf{Sc}\mathsf{Ac}}})$ | | | | | | | | | |
| $\mathbf{u}_0$ | NULL | $\mathbf{u}_1$ | NULL | NULL | NULL | NULL | $\mathbf{u}'_1$ | NULL | $\mathbf{u}''_1$ |
| 3. PromoteInitialUpdate$(K_{\mathsf{Mg}})$ Status: *Active* Initiator: $K_{\mathsf{Em}}$ $\mathbf{u}_2 \in \mathsf{NERestr}(\mathsf{MinLift}(\mathbf{u}'_1, \Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}}, M))$ $\mathbf{u}'_2 = \mathsf{Proj}(\mathbf{u}_2, \Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}})$ | | | | | | | | | |
| $\mathbf{u}_0$ | NULL | $\mathbf{u}_1$ | NULL | $\mathbf{u}'_2$ | NULL | $\mathbf{u}_2$ | NULL | NULL | $\mathbf{u}''_1$ |
| 4. PromoteInitialUpdate$(K_{\mathsf{Ac}})$ Status: *Active* Initiator: $K_{\mathsf{Em}}$ $\mathbf{u}_3 \in \mathsf{NERestr}(\mathsf{MinLift}(\mathbf{u}''_1, \Gamma_{\underline{\mathsf{Sc}\mathsf{Ac}}}, M))$ $\mathbf{u}'_3 = \mathsf{Proj}(\mathbf{u}_3, \Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}})$ | | | | | | | | | |
| $\mathbf{u}_0$ | NULL | $\mathbf{u}_1$ | NULL | $\mathbf{u}'_2$ | $\mathbf{u}'_3$ | $\mathbf{u}_2$ | NULL | $\mathbf{u}_3$ | NULL |
| 5. RefineUpdate$(K_{\mathsf{Sc}})$ Status: *Active* Initiator: $K_{\mathsf{Em}}$ $\mathbf{u}_4 \in \mathsf{NERestr}(\mathsf{Refine}(\mathbf{u}_1, ((\Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}}, \mathbf{u}'_2), (\Gamma_{\underline{\mathsf{Sc}\mathsf{Ac}}}, \mathbf{u}'_3))))$ $\mathbf{u}'_4 = \mathsf{Proj}(\mathbf{u}_4, \Gamma_{\underline{\mathsf{Em}\mathsf{Sc}}})$ | | | | | | | | | |
| $\mathbf{u}_0$ | $\mathbf{u}'_4$ | $\mathbf{u}_4$ | NULL | NULL | NULL | $\mathbf{u}_2$ | NULL | $\mathbf{u}_3$ | NULL |
| 6. RefineUpdate$(K_{\mathsf{Em}})$ Status: *Active* Initiator: $K_{\mathsf{Em}}$ $\mathbf{u}_5 = \mathsf{Refine}(\mathbf{u}_1, ((\Gamma_{\underline{\mathsf{Em}\mathsf{Sc}}}, \mathbf{u}'_4)))$ | | | | | | | | | |
| $\mathbf{u}_5$ | NULL | $\mathbf{u}_4$ | NULL | NULL | NULL | $\mathbf{u}_2$ | NULL | $\mathbf{u}_3$ | NULL |
| 7. AcceptUpdate Status: *Accepted* Initiator: $K_{\mathsf{Em}}$ | | | | | | | | | |
| $\mathbf{u}_5$ | NULL | $\mathbf{u}_4$ | NULL | NULL | NULL | $\mathbf{u}_2$ | NULL | $\mathbf{u}_3$ | NULL |
| 8. SelectFinalUpdate$(K_{\mathsf{Em}}, u_5)$ Status: *Final* Initiator: $K_{\mathsf{Em}}$ $u_5 = (M_{\mathsf{Em}}, M'_{\mathsf{Em}}) \in \mathsf{Updates}(\mathbf{u}_5)$ $u'_5 = \mathsf{Proj}(u_5, \Gamma_{\underline{\mathsf{Em}\mathsf{Sc}}})$ | | | | | | | | | |
| $u_5$ | NULL | $\mathbf{u}_4$ | $u'_5$ | NULL | NULL | $\mathbf{u}_2$ | NULL | $\mathbf{u}_3$ | NULL |
| 9. RefineFinalUpdate$(K_{\mathsf{Sc}})$ Status: *Final* Initiator: $K_{\mathsf{Em}}$ $u_4 = (M_{\mathsf{Sc}}, M'_{\mathsf{Sc}}) \in \mathsf{NERestr}(\mathsf{Refine}(\mathbf{u}_4, (u'_5, \Gamma_{\underline{\mathsf{Em}\mathsf{Sc}}})))$ $u'_4 = \mathsf{Proj}(u_4, \Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}})$ $u''_4 = \mathsf{Proj}(u_4, \Gamma_{\underline{\mathsf{Sc}\mathsf{Ac}}})$ | | | | | | | | | |
| $u_5$ | NULL | $u_4$ | NULL | NULL | NULL | $\mathbf{u}_2$ | $u'_4$ | $\mathbf{u}_3$ | $u''_4$ |
| 10. RefineFinalUpdate$(K_{\mathsf{Mg}})$ Status: *Final* Initiator: $K_{\mathsf{Em}}$ $u_2 = (M_{\mathsf{Mg}}, M'_{\mathsf{Mg}}) \in \mathsf{NERestr}(\mathsf{Refine}(\mathbf{u}_2, (u'_4, \Gamma_{\underline{\mathsf{Sc}\mathsf{Mg}}})))$ | | | | | | | | | |
| $u_5$ | NULL | $u_4$ | NULL | NULL | NULL | $u_2$ | NULL | $\mathbf{u}_3$ | $u''_4$ |
| 11. RefineFinalUpdate$(K_{\mathsf{Ac}})$ Status: *Final* Initiator: $K_{\mathsf{Em}}$ $u_3 = (M_{\mathsf{Ac}}, M'_{\mathsf{Ac}}) \in \mathsf{NERestr}(\mathsf{Refine}(\mathbf{u}_3, (u''_4, \Gamma_{\underline{\mathsf{Sc}\mathsf{Ac}}})))$ | | | | | | | | | |
| $u_5$ | NULL | $u_4$ | NULL | NULL | NULL | $u_2$ | NULL | $u_3$ | NULL |
| 12. CommitUpdate Status: *Idle* Initiator: NULL New database state $= (M'_{\mathsf{Em}}, M'_{\mathsf{Sc}}, M'_{\mathsf{Mg}}, M'_{\mathsf{Ac}})$ | | | | | | | | | |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

an *unprocessed* update request to component $C$; that is, a request which has yet to be lifted to that component. Once the lifting takes place, $\mathsf{PortStatus}(C, \Gamma)$ is reset to NULL. Note further that the request is not placed in the status register for $\Gamma_{\underline{\mathsf{EmSc}}}$, since the initiating component already knows about it.

The secretariat component must agree to the selected update by proposing a corresponding update to the $\mathsf{Travel}_{\mathsf{Sct}}$ relation, so that the projections of $\mathsf{Travel}_{\mathsf{Emp}}$ and $\mathsf{Travel}_{\mathsf{Sct}}$ agree on $\mathsf{Travel}_{\mathsf{EmpSct}}$. Formally, this is accomplished via a *lifting* of $\mathbf{u}_0$; that is, a ranked directed update $\mathbf{u}_1$ on $\mathsf{Schema}(K_{\mathsf{Sc}})$ which projects to $\mathbf{u}_0$ under $\gamma_{\underline{\mathsf{EmSc}}}$. In principle, this could be any such lifting, but since the secretariat is assumed to be largely an administrative arm in this example, it is reasonable to assume that the lifting retains all possibilities requested by the employee. This lifting must in turn be passed along to the other components to which $K_{\mathsf{Sc}}$ is connected; namely the management component $K_{\mathsf{Mg}}$ and the accounting component $K_{\mathsf{Ac}}$. It is not passed back to $K_{\mathsf{Em}}$ at this point, since it is assumed that $\mathsf{Proj}(\mathbf{u}_1, \Gamma_{\underline{\mathsf{EmSc}}}) = \mathsf{Proj}(\mathbf{u}_0, \Gamma_{\underline{\mathsf{EmSc}}})$. A lifted update is passed back to the sending component only when the lifting alters the projection on to their interconnection. Thus, if the secretariat had made restrictions to the proposed update (by limiting the number of days, say), then the lifting would need to be passed back to $K_{\mathsf{Em}}$ as well. In any case, the lifting which is selected by the secretariat is passed along to $K_{\mathsf{Mg}}$ and $K_{\mathsf{Act}}$ as $\mathbf{u}'_1$ and $\mathbf{u}''_1$, respectively, as represented in step 2 of Table 2: PROMOTEINITIALUPDATE($K_{\mathsf{Sc}}$).

Management responds by lifting $\mathbf{u}'_1$ to an update $\mathbf{u}_2$ on the entire component $K_{\mathsf{Mg}}$, as illustrated in step 3 of Table 2: PROMOTEINITIALUPDATE($K_{\mathsf{Mg}}$). Suppose, for example, that $\mathbf{u}_2$ approves travel to ADBIS for seven days, but denies travel to DEXA completely. This decision must then be passed back to $K_{\mathsf{Sc}}$; this is represented as $\mathbf{u}'_2$ in the table. On the other hand, if $K_{\mathsf{Mg}}$ decides to allow all possible travel possibilities which are represented in $\mathbf{u}'_1$, the lifting is not passed back to $K_{\mathsf{Sc}}$. This point will be discussed in more detail later.

Similarly, $K_{\mathsf{Ac}}$ must respond to $\mathbf{u}''_1$, which is a ranked update regarding travel funds, but not the number of days. Accounting must decide upon an appropriate lifting $\mathbf{u}_3$. For example, in the lifting $\mathbf{u}_3$ of $\mathbf{u}''_1$, it may be decided that €1500 can be allocated for travel to DEXA, but only €900 for travel to ADBIS. In step 4 of Table 2, this component reports its lifting decision back to $K_{\mathsf{Sc}}$ via $\mathbf{u}'_3$.

In step 5, REFINEUPDATE($K_{\mathsf{Sc}}$), the decisions $\mathbf{u}'_2$ and $\mathbf{u}'_3$ of $K_{\mathsf{Mg}}$ and $K_{\mathsf{Ac}}$, which were reported to $K_{\mathsf{Sc}}$ in steps 3 and 4, are lifted to $\mathbf{u}_4$ and then reported back to $K_{\mathsf{Em}}$ via $\mathbf{u}'_4$. In this example, the employee Lena would discover that she may travel only to ADBIS and not to DEXA, with a maximum funding allocation of €900 and for at most seven days. She acknowledges this with REFINEUPDATE($K_{\mathsf{Em}}$) in step 6, producing $\mathbf{u}_5$. As she is the originator of the update request, it is highly unlikely that $\mathbf{u}_5$ would be anything but the maximal lifting of $\mathbf{u}_4$. As such, further update is passed back to $K_{\mathsf{Sc}}$ during this step.

The system next observes that no component has any pending updates in its port-status registers, and so marks the cooperative update process as successful via the ACCEPTUPDATE action in step 7, which includes a transition to *Accepted*

status. Unlike the previous steps, this action is taken entirely by the system, and it is the only possibility from the state reached after step 6.

The process is not yet complete, however, as Lena must select a particular update, and then that update must be lifted to the entire network of components. She may select any member of $\mathbf{u}_5$; however, in this case, there is a maximal entry in what remains of her initial ranked update: travel to ADBIS for seven days with €900. Given her indicated preferences, this would likely be her choice. A second round of confirmation via project-lift requires each of the other components to select a specific update to match her choice. These choices (for example, the *ActID* which pays for the trip) will be invisible to Lena. The details are contained in steps 8-11 of Table 2 via a SELECTFINALUPDATE($K_{\mathsf{Em}}$,) step, followed by three REFINEFINALUPDATE($C$) commands, one for each of the three other components, and are similar in nature to the previous negotiation. Keep in mind that boldface letters (e.g., $\mathbf{u}$) represent ranked updates, while italics letters (e.g.$u$) represent simple updates. After these, the final step is the system-initiated action COMMITUPDATE, in which the agreed-upon update is committed to the database.

There are a few further points worth mentioning. First of all, the order in which the steps were executed is not fixed. For example, steps 3 and 4 can clearly be interchanged with no difference in subsequent ones. However, even greater variation is possible. Step 5, REFINEUPDATE($K_{\mathsf{Sc}}$), could be performed before PROMOTEINITIALUPDATE($K_{\mathsf{Ac}}$) of step 4. In that case, upon completion of REFINEUPDATE($K_{\mathsf{Ac}}$), a second execution of REFINEUPDATE($K_{\mathsf{Sc}}$) would be necessary. The final result would nonetheless be the same. More generally, the final result is independent of the order in which decisions are made. See Observation 3.5 for a further discussion.

Suppose that in PROMOTEINITIALUPDATE($K_{\mathsf{Ac}}$) of step 4, all incoming updates in $\mathbf{u}_1''$ are supported by $\mathbf{u}_3$, i.e., $\mathsf{Proj}(\mathbf{u}_3, \Gamma_{\mathsf{Sc}\underline{\mathsf{Ac}}}) = \mathbf{u}_1''$. In that case, no revised request of the form $\mathbf{u}_3'$ is transmitted back to $K_{\mathsf{Sc}}$ via the port-status register. It is thus natural to ask how $K_{\mathsf{Sc}}$ "knows" that no such request will appear. The answer is that it does not matter. The entire process is nondeterministic, and $K_{\mathsf{Sc}}$ can execute REFINEUPDATE($K_{\mathsf{Sc}}$) based upon the input $\mathbf{u}_2'$ from $K_{\mathsf{Mg}}$ alone. If an update request $\mathbf{u}_3'$ comes from $K_{\mathsf{Ac}}$ later, a second REFINEUPDATE($K_{\mathsf{Sc}}$) based upon it would allow precisely the same final result as would the process described in Table 2. Again, see Observation 3.5 below.

It is also worthy of note that some decisions may lead to a rejection. For example, management might decide to allow travel only to ADBIS, while the accountant might find that there are funds for travel to DEXA but not ADBIS. In that case, the refinement step 5 would fail, and the only possible step to continue would be a rejection. Additionally, any component can decide to reject any proposed update on its ports at any time before the *Accepted* state is reached simply by executing a REJECTUPDATE($C$), even if there is a possible update. For example, a supervisor might decide to disallow a trip.

## 3 A Formal Model of Cooperative Updates

In this section, some of the technical details regarding updates and update families are elaborated, and then a more complete description of the behavior of the update automaton is given. In Definition 3.1–Definition 3.3 below, let $\mathbf{D}$ be a database schema, and let $\Gamma = (\mathbf{V}, \gamma)$ be a view of $\mathbf{D}$; that is, $\gamma : \mathbf{D} \to \mathbf{V}$ is a database morphism whose underlying mapping $\mathring{\gamma} : \mathsf{LDB}(\mathbf{D}) \to \mathsf{LDB}(\mathbf{V})$ is surjective. Consult [11] for details.

**Definition 3.1 (Updates and update families).** Following [2, Sec. 3], an *update* on $\mathbf{D}$ is a pair $u = (M_1, M_2) \in \mathsf{LDB}(\mathbf{D}) \times \mathsf{LDB}(\mathbf{D})$. This update is called an *insertion* if $M_1 \leq_{\mathbf{D}} M_2$, and a *deletion* if $M_2 \leq_{\mathbf{D}} M_1$. A *directional update* is one which is either an insertion or else a deletion. A *ranked update* on $\mathbf{D}$ is a triple $\mathbf{u} = (M, S, \leq_{\mathbf{u}})$ in which $M \in \mathsf{LDB}(\mathbf{D})$, $S$ is a finite subset of $\mathsf{LDB}(\mathbf{D})$, and $\leq_{\mathbf{u}}$ is a preorder (i.e., a reflexive and transitive relation [12, 1.2]) on $S$, called the *preference ordering*. The set of *updates of* $\mathbf{u}$ is $\mathsf{Updates}(\mathbf{u}) = \{(M, M') \mid M' \in S\}$. $\mathbf{u}$ is *deterministic* if $\mathsf{Updates}(\mathbf{u})$ contains exactly one pair, and *empty* if $\mathsf{Updates}(\mathbf{u}) = \emptyset$. In general, ranked updates are denoted by boldface letters (e.g., $\mathbf{u}$), while ordinary updates will be denoted by italic letters (e.g., $u$). The ranked update $\mathbf{u}$ is an *insertion* (resp. a *deletion*) if every $(M, M') \in \mathsf{Updates}(\mathbf{u})$ is an insertion (resp. a deletion), and $\mathbf{u}$ is a *ranked directional update* if it is either an insertion or else a deletion. Every (ordinary) update can be regarded as a ranked update in the obvious way; to $(M_1, M_2)$ corresponds the ranked update $(M_1, \{M_2\}, \leq_{\mathbf{D}|\{M_1, M_2\}})$. The set of all ranked directional updates on $\mathbf{D}$ is denoted $\mathsf{RDUpdates}(\mathbf{D})$. For $S' \subseteq S$, the *restriction* of $u$ to $S'$ is $\mathsf{Restr}(\mathbf{u}, S') = (M_1, S', \leq_{\mathbf{u}|S'})$ with $\leq_{\mathbf{u}|S'}$ the restriction of $\leq_{\mathbf{u}}$ to $S'$. The set of all *nonempty restrictions* of $u$ is $\mathsf{NERestr}(\mathbf{u}) = \{\mathsf{Restr}(\mathbf{u}, S') \mid (S' \subseteq S) \wedge (S' \neq \emptyset)\}$.

**Definition 3.2 (Projection of updates and update families).** For $u = (M_1, M_2)$ an update on $\mathbf{D}$, the *projection of* $u$ *to* $\Gamma$ is the update $(\mathring{\gamma}(M_1), \mathring{\gamma}(M_2))$ on $\mathbf{V}$. This update is often denoted $\gamma(u)$. Now let $\mathbf{u} = (M_1, S, \leq_{\mathbf{u}})$ be a ranked update on $\mathbf{D}$. The *projection of* $\mathbf{u}$ *to* $\Gamma$, denoted $\mathsf{Proj}(\mathbf{u}, \Gamma)$, is the ranked update $\gamma(\mathbf{u}) = (\mathring{\gamma}(M_1), \mathring{\gamma}(S), \leq_{\gamma(\mathbf{u})})$ in which $\mathring{\gamma}(S) = \{\gamma(M) \mid M \in S\}$ and $N \leq_{\gamma(\mathbf{u})} N'$ iff for every $M, M' \in S$ for which $\mathring{\gamma}(M) = N$ and $\mathring{\gamma}(M') = N'$, $M \leq_{\mathbf{u}} M'$. Observe that projection preserves the property of being an insertion (resp. deletion) in both the simple and ranked cases. Another important operation in component-based updating is refinement. Suppose that $\mathbf{u}$ is a proposed ranked update to a component, and that for each of its ports is given a ranked update which is a restriction of $\mathbf{u}$ onto that component. The refinement of $\mathbf{u}$ by those restrictions is the largest restriction of $\mathbf{u}$ which is compatible with all of the ranked updates on the ports. The formal definition is as follows. Let $\{\Gamma_i \mid 1 \leq i \leq n\}$ be a set of views of $\mathbf{D}$, $\mathbf{u'}_i = \mathsf{Restr}(\mathsf{Proj}(\mathbf{u}, \Gamma_i), S)$, and for $1 \leq i \leq n$, let $S_i \subseteq \{\mathring{\gamma}_i(M) \mid M \in S\}$, with $S' = \{M \in S \mid (\forall i \in \{1, 2, \ldots, n\})(\exists N_i \in S_i)(\mathring{\gamma}_i(M) = N_i)\}$. The *refinement* of $\mathbf{u}$ by $U = \{(\Gamma_i, \mathbf{u'}_i) \mid 1 \leq i \leq n\}$ is defined to be $\mathsf{Restr}(\mathbf{u}, S')$, and is denoted $\mathsf{Refine}(\mathbf{u}, U)$.

**Definition 3.3 (Liftings of updates and update families).** The operation which is inverse to projection is lifting, in which an update to a view is "lifted" to the main schema. In contrast to projection, lifting is inherently a nondeterministic operation. Let $u = (N_1, N_2)$ be an update on $\mathbf{V}$, and let $M_1 \in \mathsf{LDB}(\mathbf{D})$ with $\mathring{\gamma}(M_1) = N_1$ as well. A *lifting* of $u$ to $\mathbf{D}$ for $M_1$ is an update $u' = (M_1, M_2)$ on $\mathbf{D}$ with the property that $\mathring{\gamma}(M_2) = N_2$. If $u$ is an insertion (resp. deletion), the lifting $u'$ is *direction preserving* if $u'$ is also an insertion (resp. deletion). If $u$ is an insertion and $u'$ is direction preserving, $u'$ is *minimal* if for for any lifting $(M_1, M_2')$ of $u$ to $\mathbf{D}$ for $M_1$ with $M_2' \leq_u M_2$, it must be the case that $M_2' = M_2$, and $u'$ is *least* if for any lifting $(M_1, M_2')$ of $u$ to $\mathbf{D}$ for $M_1$, $M_2 \leq_u M_2'$. A corresponding definition holds for deletions, with "$\leq_u$" replaced by "$\geq_u$". These ideas extend in a straightforward manner to ranked updates. Let $\mathbf{u} = (N_1, S, \leq_{\mathbf{u}})$ be a ranked update on $\mathbf{V}$, and let $M_1 \in \mathsf{LDB}(\mathbf{D})$ with $\mathring{\gamma}(M_1) = N_1$. A *lifting* of $\mathbf{u}$ to $\mathbf{D}$ for $M_1$ is a ranked update $\mathbf{u'} = (M_1, S', \leq_{\mathbf{u}'})$ on $\mathbf{D}$ which satisfies the following three properties:

(lift–i) $(\forall M_2 \in S')(\exists N_2 \in S)(\mathring{\gamma}(M_2) = N_2)$.
(lift–ii) $(\forall N_2 \in S)(\exists M_2 \in S')(\mathring{\gamma}(M_2) = N_2)$.
(lift–iii) For $M_2, M_2' \in S'$, $M_2 \leq_{\mathbf{u}'} M_2'$ iff $\mathring{\gamma}(M_2) \leq_{\mathbf{u}} \mathring{\gamma}(M_2')$.

If $\mathbf{u}$ is a ranked directional update, then the lifting $\mathbf{u'}$ is *direction preserving* if it satisfies the obvious conditions —- if $\mathbf{u}$ is an insertion (resp. deletion), then so too is $\mathbf{u'}$. A direction-preserving ranked directional update $u$ is *minimal* (resp. *least*) if each member of $\mathsf{Updates}(\mathbf{u})$ is minimal (resp. least). The set of all minimal liftings of $\mathbf{u}$ for $M_1$ is denoted $\mathsf{MinLift}(\mathbf{u}, \Gamma, M_1)$.

**Definition 3.4 (The update automaton).** Details regarding the precise conditions under which the actions of the automaton may be applied are expanded here. For the most part, information which has already been presented in Example 2.2 and Table 1 will not be repeated.

The machine operates nondeterministically. There are eight classes of actions; any member of any of these classes may be selected as the next step, provided that its preconditions are satisfied. The listed actions are then executed in the order given. All of these operations, with the exception of AcceptUpdate and *Commit*, must be initiated by a user of the associated component. Operation is synchronous; that is, only one operation may be executed at a time. Subsequent operations must respect the state generated by the previous operation. Formally, a *computation* of this automaton is a sequence $D = \langle D_1, D_2, \ldots, D_n \rangle$ in which $D_1 = \text{InitiateUpdate}(C, \mathbf{u})$ for some $C \in X$ and $\mathbf{u} \in \mathsf{RDUpdates}(\mathsf{Schema}(C))$, and for $1 \leq i \leq n-1$, $D_{i+1}$ is a legal step to follow $D_i$ according to the rules spelled out below. The computation *defines a single negotiation* if $D_n =$ CommitUpdate, while $D_k \neq$ CommitUpdate for any $k < n$. The *length* of the computation is $n$. In the description which follows, $X$ is taken to be a finite set of components with $J$ an interconnection family for $X$.

InitiateUpdate$(C, \mathbf{u})$: This is the first step in the update process, and is initiated by a user of the component $C$ by proposing a ranked update $\mathbf{u}$ to its

current state. The appropriate projections of this update are propagated to all ports of components which are connected to $C$ and whose state is altered by the update.

Preconditions:
$P_{11}$: $\mathsf{Status} = \mathit{Idle}$
Actions:
$Q_{11}$: $\mathsf{Initiator} \leftarrow C$
$Q_{12}$: $\mathsf{Status} \leftarrow \mathit{Active}$
$Q_{13}$: $\mathsf{PendingUpdate}(C) \leftarrow \mathbf{u}$
$Q_{14}$: $(\forall \Gamma \in \mathsf{Ports}\langle C\rangle)(\forall \Gamma' \in \mathsf{AdjPorts}\langle C, J\rangle)((\mathsf{Proj}(\mathbf{u}, \Gamma) \neq \text{ identity })$
$\Rightarrow \mathsf{PortStatus}(\mathsf{SrcCpt}(\Gamma'), \Gamma') \leftarrow \mathsf{Proj}(\mathbf{u}, \Gamma))$.

PromoteInitialUpdate($C$): This step is relevant in the situation that a component has received an update request on one of its ports, but it has not yet proposed any corresponding update to its own state. A user of that component selects a lifting of this update request for its own state, and propagates its projections to its ports to all neighboring components.

Preconditions:
$P_{21}$: $\mathsf{Status} = \mathit{Active}$
$P_{22}$: $\mathsf{PendingUpdate}(C) = \mathsf{NULL}$
$P_{23}$: $(\exists \Gamma' \in \mathsf{Ports}\langle C\rangle)(\mathsf{PortStatus}(C, \Gamma') \neq \mathsf{NULL})$
/* Since the hypergraph of $J$ is acyclic, $\Gamma'$ must be unique. */
Actions:
$Q_{21}$: $\mathsf{PendingUpdate}(C) \leftarrow \mathsf{Choose}\ \mathbf{u} \in \mathsf{NERestr}(\mathsf{MinLift}(\mathbf{u}, \Gamma', \mathsf{Schema}\langle C\rangle)))$
where $(\Gamma' \in \mathsf{Ports}\langle C\rangle) \wedge (\mathsf{PortStatus}(C, \Gamma') \neq \mathsf{NULL})$
$Q_{22}$: $(\forall \Gamma \in \mathsf{Ports}\langle C\rangle)(\mathsf{PortStatus}(C, \Gamma) \leftarrow \mathsf{NULL})$
$Q_{23}$: $(\forall \Gamma \in \mathsf{Ports}\langle C\rangle)(\forall \Gamma'' \in \mathsf{AdjPorts}\langle C, J\rangle)((\mathsf{Proj}(\mathbf{u}, \Gamma) \neq \text{ identity })$
$\Rightarrow \mathsf{PortStatus}(\mathsf{SrcCpt}(\Gamma''), \Gamma'') \leftarrow \mathsf{Proj}(\mathbf{u}, \Gamma))$.

RefineUpdate($C$): In this step, a user of component $C$ further restricts its current proposal for an update, based upon additional ranked updates received at its ports. Upon successful completion, the new proposed update of the component is consistent with the proposed updates which were on its ports.

Preconditions:
$P_{31}$: $\mathsf{Status} = \mathit{Active}$
$P_{32}$: $\mathsf{PendingUpdate}(C) \neq \mathsf{NULL}$
$P_{33}$: $\mathsf{Refine}(\mathsf{PendingUpdate}(C),$
$\{(\Gamma, \mathsf{PortStatus}(C, \Gamma)) \mid \mathsf{PortStatus}(C, \Gamma) \neq \mathsf{NULL}\}) \neq \emptyset$.
Actions:
$Q_{31}$: $\mathsf{PendingUpdate}(C) \leftarrow \mathsf{Choose}\ \mathbf{u} \in \mathsf{NERestr}(\mathsf{Refine}(\mathsf{PendingUpdate}(C),$
$\{(\Gamma, \mathsf{PortStatus}(C, \Gamma)) \mid \mathsf{PortStatus}(C, \Gamma) \neq \mathsf{NULL}\}))$
$Q_{32}$: $(\forall \Gamma \in \mathsf{Ports}\langle C\rangle)(\mathsf{PortStatus}(C, \Gamma) \leftarrow \mathsf{NULL})$
$Q_{33}$: $(\forall \Gamma \in \mathsf{Ports}\langle C\rangle)(\forall \Gamma' \in \mathsf{AdjPorts}\langle C, J\rangle)((\mathsf{Proj}(\mathbf{u}, \Gamma) \neq \text{ identity })$
$\Rightarrow \mathsf{PortStatus}(\mathsf{SrcCpt}(\Gamma'), \Gamma') \leftarrow \mathsf{Proj}(\mathbf{u}, \Gamma))$.

RejectUpdate($C$): This step is a crude, all-purpose rejection step. Any component can reject the proposed update and terminate the entire process at any time for any reason. One reason might be that it cannot unify the update proposals on its ports, but it could also be that one of its users wishes to terminate the update for other reasons. Upon such termination, the update automaton is returned to its initial state; all proposed updates are discarded.

Preconditions:
$P_{41}$: $\mathsf{Status} = \mathit{Active}$
Actions:
$Q_{41}$: $\mathsf{Status} \leftarrow \mathit{Idle}$
$Q_{42}$: $\mathsf{Initiator} \leftarrow \mathsf{NULL}$
$Q_{43}$: $(\forall C \in X)(\mathsf{PendingUpdate}(C) \leftarrow \mathsf{NULL})$
$Q_{44}$: $(\forall C' \in X)(\forall \Gamma \in \mathsf{Ports}\langle C' \rangle)(\mathsf{PortStatus}(C', \Gamma) \leftarrow \mathsf{NULL})$

AcceptUpdate: This step is executed when all components which are involved in the update process agree that the update can be supported. This agreement is indicated by the fact that no port has a pending update; all such port updates have been integrated into component updates. It is initiated automatically; users cannot effect it directly.

Preconditions:
$P_{51}$: $\mathsf{Status} = \mathit{Active}$
$P_{52}$: $(\forall C \in X)(\forall \Gamma \in \mathsf{Ports}\langle C \rangle)(\mathsf{PortStatus}(C, \Gamma) = \mathsf{NULL})$
Actions:
$Q_{51}$: $\mathsf{Status} \leftarrow \mathit{Accepted}$

SelectFinalUpdate($C, u$): Upon acceptance, the components have agreed upon a ranked update. However, the database must be updated to a single new state; thus, a single update must be chosen from the ranked set. The purpose of this step is to initiate that selection process; it is always executed by a user of the component in which the update request initiated.

Preconditions:
$P_{61}$: $\mathsf{Status} = \mathit{Accepted}$
$P_{62}$: $\mathsf{Initiator} = C$
$P_{63}$: $\mathbf{u} \in \mathsf{NERestr}(\mathsf{PendingUpdate}(C)) \wedge \mathbf{u}$ is deterministic.
Actions:
$Q_{61}$: For each $\Gamma \in \mathsf{Ports}\langle C \rangle$, if $\mathsf{Proj}(\mathbf{u}, \Gamma)$ is not the identity update, then for all $\Gamma' \in \mathsf{AdjPorts}\langle C, J \rangle$, $\mathsf{PortStatus}(\mathsf{SrcCpt}(\Gamma'), \Gamma') \leftarrow \mathsf{Proj}(\mathbf{u}, \Gamma)$
$Q_{62}$: $\mathsf{Status} \leftarrow \mathit{Final}$

RefineFinalUpdate($C$): In this step, a user of component $C$ selects a final deterministic update which is consistent with that chosen by the initiator.

Preconditions:
$P_{71}$: $\mathsf{Status} = \mathit{Final}$

Actions:
$Q_{71}$: $\mathsf{PendingUpdate}(C) \leftarrow$ some *deterministic* restriction of $\mathsf{Refine}(\mathsf{PendingUpdate}(C), \{(\Gamma, \mathsf{PortStatus}(C, \Gamma)) \mid \mathsf{PortStatus}(C, \Gamma) \neq \mathsf{NULL}\})$
$Q_{72}$: $(\forall \Gamma \in \mathsf{Ports}\langle C\rangle)(\mathsf{PortStatus}(C, \Gamma) \leftarrow \mathsf{NULL})$

CommitUpdate: In this step, the update is committed to the database, and the update automaton is returned to its initial state (albeit with the new database state). This step is executed automatically when its preconditions are satisfied; users cannot initiate it.

Preconditions:
$P_{81}$: $\mathsf{Status} = \mathit{Final}$
$P_{82}$: $(\forall \Gamma \in \mathsf{Ports}\langle C\rangle)(\mathsf{PortStatus}(C, \Gamma) = \mathsf{NULL})$
Actions:
$Q_{81}$: $(\forall C \in X)(\mathsf{CurrentState}(C) \leftarrow \mathsf{PendingUpdate}(C))$
$Q_{81}$: $(\forall C \in X)(\mathsf{PendingUpdate}(C) \leftarrow \mathsf{NULL})$
$Q_{83}$: $\mathsf{Initiator} \leftarrow \mathsf{NULL}$
$Q_{84}$: $\mathsf{Status} \leftarrow \mathit{Idle}$

Because the model of negotiation which has been presented is very simple, it has some nice theoretical properties. Firstly, infinite computations are not possible; the machine will always halt. Second, although the machine is nondeterministic, it is not necessary to guess correctly to make things work. These ideas are formalized in the following observation.

**Observation 3.5 (Computations are well behaved).** *In the automaton model given in Definition 3.4, the following conditions hold.*

(a) *For any $C \in X$ and any $\mathbf{u} \in \mathsf{RDUpdates}(\mathsf{Schema}(C))$, there is a natural number $n$ (which may depend upon the current state of the database) such that every computation beginning with* InitiateUpdate$(C, \mathbf{u})$ *has length at most $n$.*
(b) *Let $D = \langle D_1, D_2, \ldots, D_{k-1}, D_k, \ldots, D_n\rangle$ be a computation of the update automaton which defines a single negotiation, and let $E = \langle E_1, E_2, \ldots, E_{k-1}, E_k\rangle$ be a computation of that same machine with $E_i = D_i$ for $1 \leq i \leq k-1$. Assume further that $E_k$ is not of the form* RejectUpdate$(C)$. *Then there is a choice of lifting associated with $E_k$ and a computation $E' = \langle E_1, E_2, \ldots, E_{k-1}, E_k, \ldots, E_{n'}\rangle$ which also defines a single negotiation, and with the further property that $D$ and $E'$ result in the same update on the database. (The values of $n$ and $n'$ need not be the same).*

Proof outline: Part (a) follows from the observation that only one decision for an initial ranked update may be made for each component. After that, the process only refines these initial decisions. Since a ranked update is finite by definition, there can only be a finite number of such refinements, and each such refinement must reduce the number of possibilities in some pending update. Part (b) follows by observing that one may simply choose for the lifting of $E_k$ the final ranked update for that component under the computation $D$. □

## 4 Conclusions and Further Directions

The basic idea of supporting view update by negotiating with other views (*qua* components) has been presented. A formal model of this process has been developed using a nondeterministic automaton as the underlying computational model. Although it has certain limitations, it does provide a formal model of update by component cooperation within that context, thus providing a firm basis for further development of these ideas, a few of which are identified below.

**Extension of the basic model:** The basic model of cooperative update is limited in several ways. For these reasons, the it should be viewed as a proof-of-concept effort rather than a comprehensive solution; further research will address the following issues. First, the current model of cooperation is monotonic. Once the actors have made initial proposals for the updates which they support, the process of identifying the solution update is solely one of refining those initial proposals. On the other hand, in realistic situations, it is often necessary for the parties to negotiate nonmonotonically, by retracting their initial proposals and then submitting new ones, or by modifying their existing proposals adding new alternatives rather than by just refining existing ones. A key extension for future work is to develop an extended model which supports such nonmonotonic negotiation. A second important direction for future work is the development of a computational formalism which embodies more specific modelling of communication between components. The current automaton-based rendering does not provide the necessary information to the actors to effect an efficient negotiation; the nondeterminism allows very long and inefficient although completely correct solutions. With a more detailed model of communication, much more efficient negotiation will be possible.

**Relationship to workflow:** The topic of *workflow* involves the systematic modelling of processes which require the coordinated interaction of several actors [13]; closely related ideas are known to have central importance in information systems [14]. Within the context of the development of interactive database systems and cooperative work, long-term transactions and workflow loops are central topics [15] [16] [17]. There is a natural connection between these ideas and those surrounding cooperative update which have been introduced in this paper. Indeed, underlying the process of negotiating a cooperative solution to the view-update problem, as illustrated, for example, in Table 2, is a natural workflow of update tasks, each defined by a step in the execution of the automaton.

An important future direction for this work is to develop the connection between workflow for database transactions and the models of cooperative view update which have been presented in this paper. Indeed, given a database schema defined by components and a requested update on one of these components, it should be possible to use the ideas developed in this paper to define and identify the workflow pattern which is required to effect that update. This, in effect, would provide a theory of *query-based workflow construction*.

**Acknowledgment.** Much of this research was carried out while the first author was a visitor at the Information Systems Engineering Group at the University

of Kiel. He is indebted to Bernhard Thalheim and the members of the group for the invitation and for the many discussions which led to this collaborative work.

## References

1. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Systems 6, 557–575 (1981)
2. Hegner, S.J.: An order-based theory of updates for database views. Ann. Math. Art. Intell. 40, 63–125 (2004)
3. Hegner, S.J.: The complexity of embedded axiomatization for a class of closed database views. Ann. Math. Art. Intell. 46, 38–97 (2006)
4. Langerak, R.: View updates in relational databases with an independent scheme. ACM Trans. Database Systems 15, 40–66 (1990)
5. Bentayeb, F., Laurent, D.: View updates translations in relational databases. In: Quirchmayr, G., Bench-Capon, T.J.M., Schweighofer, E. (eds.) DEXA 1998. LNCS, vol. 1460, pp. 322–331. Springer, Heidelberg (1998)
6. Gottlob, G., Paolini, P., Zicari, R.: Properties and update semantics of consistent views. ACM Trans. Database Systems 13, 486–524 (1988)
7. Thalheim, B.: Database component ware. In: Schewe, K.D., Zhou, X. (eds.) Database Technologies, Proceedings of the 14th Australasian Database Conference, ADC 2003, Adelaide, South Australia, February 2003, pp. 13–26. Australian Computer Society (2003)
8. Schmidt, P., Thalheim, B.: Component-based modeling of huge databases. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 113–128. Springer, Heidelberg (2004)
9. Thalheim, B.: Component development and construction for database design. Data Knowl. Eng. 54, 77–95 (2005)
10. Thalheim, B.: Entity-Relationship Modeling. Springer, Heidelberg (2000)
11. Hegner, S.J.: A model of database components and their interconnection based upon communicating views. In: Jakkola, H., Kiyoki, Y., Tokuda, T. (eds.) Information Modelling and Knowledge Systems XXIV. Frontiers in Artificial Intelligence and Applications, IOS Press, 2007 (in press)
12. Davey, B.A., Priestly, H.A.: Introduction to Lattices and Order, 2nd edn. Cambridge University Press, Cambridge (2002)
13. van der Aalst, W., van Hee, K.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge (2002)
14. Flores, F., Graves, M., Hartfield, B., Winograd, T.: Computer systems and the design of organizational interaction. ACM Trans. Inf. Syst. 6, 153–172 (1988)
15. Alonso, G., Agrawal, D., Abbadi, A.E., Kamath, M., Günthör, R., Mohan, C.: Advanced transaction models in workflow contexts. In: Su, S.Y.W. (ed.) Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, pp. 574–581. IEEE Computer Society, New Orleans, Louisiana (1996)
16. Rusinkiewicz, M., Sheth, A.P.: Specification and execution of transactional workflows. In: Kim, W. (ed.) Modern Database Systems, pp. 592–620. ACM Press, Addison-Wesley (1995)
17. Hidders, J., Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M., Verelst, J.: When are two workflows the same? In: Atkinson, M.D., Dehne, F.K.H.A. (eds.) CATS. CRPIT, vol. 41, pp. 3–11. Australian Computer Society (2005)