The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems*

Anisoara Nica[1], Amy J. Lee[1] and Elke A. Rundensteiner[2]

[1] Department of EECS, University of Michigan
Ann Arbor, MI 48109-2122
[2] Department of Computer Science
Worcester Polytechnic Institute, Worcester, MA 01609-2280

Abstract. Current view technology supports only *static views* in the sense that views become undefined and hence obsolete as soon as the underlying information sources (ISs) undergo capability changes. We propose to address this new view evolution problem - which we call *view synchronization* - by a novel solution approach that allows affected view definitions to be dynamically evolved to keep them in synch with evolving ISs. We present in this paper a general strategy for the view synchronization process that guided by constraints imposed by the view evolution preferences embedded in the view definition achieves view preservation (i.e., view redefinition). We present the formal correctness, the CVS algorithm, as well as numerous examples to demonstrate the main concepts.

1 Introduction

Advanced applications such as web-based information services, digital libraries, and data mining typically operate in an information space populated with a large number of dynamic information sources (ISs) such as the WWW [14]. In order to provide easy access to information in such environments, relevant data is often retrieved from several sources, integrated as necessary, and then materialized at the user site as what's called a *view* (or data warehouse). The ISs in such environments are however dynamic, updating not only their content but also their capabilities, and joining or leaving the environment frequently.
Views in such environments thus introduce new challenges to the database community [14]. In our prior work [12, 4], we have identified view evolution caused by capability changes of one or several of the underlying ISs as a critical new problem faced by these applications. Current view technology is insufficient for supporting *flexible* view definitions. That is views are *static*, meaning views are assumed to be specified on top of a fixed environment and once the underlying ISs change their capabilities, the views defined upon them become undefined.
In our prior work, we have proposed a novel approach to solve this view inflexibility problem [12, 5, 8], called EVE (Evolveable View Environment). EVE

* This work was supported in part by the NSF NYI grant #IRI 94-57609. We would also like to thank our industrial sponsors, in particular IBM for the IBM Partnership Award and our collaborators at IBM Toronto for their support.

"preserves as much as possible" of the view instead of completely disabling it with each IS change. While the evolution of views is assumed to be implicitly triggered by capability changes of (autonomous) ISs in our work, previous work [3, 7] assumed that view redefinition was explicitly requested by the view developer at the view site, while the ISs remained unchanged. They [3, 7] thus focused on the maintenance of the materialized views after such view redefinition and not on the modification of the view definitions themselves as done in our work.
One key component of our EVE framework is E-SQL (SQL extended with view evolution preferences) that allows the view definer to control the view evolution process by indicating the criticality and dispensability of the different components of the view definition. A second key component of our EVE framework is a language called MISD for capturing descriptions of the content, capabilities as well as semantics interrelationships of all ISs in the system. Descriptions of ISs expressed in this language are maintained in a meta-knowledge base (MKB), thus making a wide range of resources available to the view synchronizer during the view evolution process.
Given a view defined in E-SQL and a MKB, we present in this paper a formal foundation for the concept of *legal rewritings* of a view affected by capability changes. This includes properties characterizing all MKB constraints must be obeyed, as well as that maximal preservation of the E-SQL evolution preferences must be achieved. Based on this formal foundation, we then propose a general strategy for solving the view synchronization problem. Our algorithm, called CVS (Complex View Synchronization), finds valid replacements for affected (deleted) components of the existing view definitions based on the semantic constraints captured in the MKB. For this, rather than just providing simple so-called 'one-step-away' view rewritings [4, 12], our solution succeeds in determining possibly complex view rewrites through multiple join constraints given in the MKB. To demonstrate our approach, we present algorithms for handling the most difficult capability change operator, namely, the delete-relation operator, in depth in this paper. The proposed strategy is shown to find a new valid definition of a view in many cases where current view technology (as well as our initial simple solution [4, 12]) would have simply disabled the view.
The remainder of the paper is structured as follows. In Sections 2 and 3 we present the IS description language and E-SQL, respectively. Section 4 describes the formal basis for correct view synchronization, while Section 5 introduces our CVS algorithm for synchronizing views based on this formal model. Sections 6 and 7 conclude the paper.

2 MISD: Model for Information Source Description

While individual ISs could be based on any data model, the schema exported by an IS is described by a set of relations $IS.R_1$, $IS.R_2$, ..., $IS.R_n$. A relation description contains three types of information specifying its data structure and content, its query capabilities as well as its relationships with exported relations from other ISs that semantically express the operations allowed between ISs.

The descriptions of the ISs are stored in the meta knowledge base (MKB) and are used in the process of view evolution [5].

Name	Syntax
Type Integrity Constraint	$\mathcal{TC}_{R.A_i} = (R(A_i) \subseteq Type_i(A_i))$
Order Integrity Constraint	$\mathcal{OC}_R = (R(A_1, \ldots, A_n) \subseteq \mathcal{C}(A_{i_1}, \ldots, A_{i_k}))$
Join Constraint	$\mathcal{JC}_{R_1,R_2} = (C_1 \ AND \ \cdots \ AND \ C_l)$
Partial/Complete Constraint	$\mathcal{PC}_{R_1,R_2} = (\pi_{\bar{A}_1}(\sigma_{C(\bar{B}_1)} R_1) \ \theta \ \pi_{\bar{A}_2}(\sigma_{C(\bar{B}_2)} R_2))$ $\theta \in \{\subset, \subseteq, \equiv, \supseteq, \supset\}$

Fig. 1. Semantic Constraints for IS Descriptions.

Example 1. We will use the following example in the rest of the paper. Consider a large travel agency which has a headquarter in Detroit, USA, and many branches all over the world. It helps its customers to arrange flights, car rentals, hotel reservations, tours, and purchasing insurances. A part of relevant IS descriptions is summarized in Fig. 2 in MISD format described below.

We introduce below MISD constraints that are used in the remainder of this paper. All MISD constraints are summarized in Fig. 1 [4, 8].
A relation R is described by specifying its information source and its set of attributes as $IS.R(A_1, \ldots, A_n)$. Each attribute A_i is given a name and a data type to specify its domain of values. This information is specified by using a *type integrity constraint* of of the format $R(A_1, \ldots, A_n) \subseteq Type_1(A_1), \ldots, Type_n(A_n)$. It says that an attribute A_i is of type $Type_i$, for $i = 1, \ldots, n$. If two attributes are exported with the same name, they are assumed to have the same type.
A *join constraint* is used to specify a meaningful way to combine information from two ISs. The join constraint is a conjunction of primitive clauses (not necessarily equijoin) of the form $\mathcal{JC}_{R_1,R_2} = (C_1 \text{ AND } \cdots \text{ AND } C_l)$ where $C_1, \ldots, C_l$ are primitive clauses over the the attributes of R_1 and R_2. The join constraint gives a default join condition that could be used to join R_1 and R_2, specifying that the join relation $J = R_1 \bowtie_{(C_1 \cdots \ AND \ \cdots C_l)} R_2$ is a meaningful way of combining the two relations. The MISD *attribute function-of constraint* relates two attributes by defining a function to transform one of them into another. This constraint is specified by $\mathcal{F}_{R_1.A,R_2.B} = (\ R_1.A = f(R_2.B)\ )$ where f is a function. $\mathcal{F}_{R_1.A,R_2.B}$ specifies that if there exists a meaningful way of combining the two relations R_1 and R_2 (e.g., using join constraints) then for any tuple t in that join relation we have $t[R_1.A] = f(t[R_2.B])$.

Example 2. For our running example, some join constraints and function-of constraints are given in Fig. 2 (underlined names are the relations for which the join constraints are defined).

IS #	Descriptions
IS 1	**Customer(Name, Addr, Phone, Age)**
IS 2	**Tour(TourID, TourName, Type, NoDays)**
IS 3	**Participant(Participant, TourID, StartDate, Loc)**
IS 4	**FlightRes(PName, Airline, FlightNo, Source, Dest, Date)**
IS 5	**Accident−Ins(Holder, Type, Amount, Birthday)**
IS 6	**Hotels(City, Address, PhoneNumber)**
IS 7	**RentACar(Company, City, PhoneNumber, Location)**

$\mathcal{JC}$	**Join Constraint**
JC1	**Customer.Name = FlightRes.PName**
JC2	**Customer.Name = Accident−Ins.Holder** AND **Customer.Age** $>$ 1
JC3	**Customer.Name = Participant.Participant**
JC4	**Participant.TourID = Tour.TourID**
JC5	**Hotels.Address = RentACar.Location**
JC6	**FlightRes.PName = Accident−Ins.Holder**

$\mathcal{F}$	**Function-of Constraints**
F1	**Customer.Name = FlightRes.PName**
F2	**Customer.Name = Accident−Ins.Holder**
F3	**Customer.Age** = (today - **Accident−Ins.Birthday**)/ 365
F4	**Customer.Name = Participant.Participant**
F5	**Participant.TourID = Tour.TourID**
F6	**Hotels.Address = RentACar.Location**
F7	**Hotels.City = RentACar.City**

Fig. 2. Content Descriptions, Join and Function-of Constraints for Ex. 1

3 Extending SQL for Flexible View Synchronization

In this section, we present E-SQL, which is an extension of SELECT-FROM-WHERE SQL augmented with specifications for how the view definition may be synchronized under IS capability changes. Evolution preferences, expressed as *evolution parameters*, allow the user to specify criteria based on which the view will be transparently evolved by the system under capability changes at the ISs. As indicated in Fig. 3, each component of the view definition (i.e., attribute, relation or condition) has attached two evolution parameters. One, the *dispensable parameter* (notation $\mathcal{X}D$, where $\mathcal{X}$ could be $\mathcal{A}$, $\mathcal{R}$ or $\mathcal{C}$) specifies if the component could be dropped (*true*) or must be present in any evolved view definition (*false*). Two, the *replaceable parameter* (notation $\mathcal{X}R$) specifies if the component could be replaced in the process of view evolution (*true*) or must be left unchanged as defined in the initial view (*false*). In Fig. 3, each type of evolution parameter used by E-SQL is represented by a row in that table, column one gives the parameter name and its abbreviation while column two lists the possible values each parameter can take (default values are underlined). The example below demonstrates the integrated usage of these evolution parameters (a detailed description of E-SQL can be found in [5]).

Evolution Parameter		Semantics
Attribute-	dispensable ($\mathcal{AD}$)	*true:* the attribute is dispensable *false:* the attribute is indispensable
	replaceable ($\mathcal{AR}$)	*true:* the attribute is replaceable *false:* the attribute is nonreplaceable
Condition-	dispensable ($\mathcal{CD}$)	*true:* the condition is dispensable *false:* the condition is indispensable
	replaceable ($\mathcal{CR}$)	*true:* the condition is replaceable *false:* the condition is nonreplaceable
Relation-	dispensable ($\mathcal{RD}$)	*true:* the relation is dispensable *false:* the relation is indispensable
	replaceable ($\mathcal{RR}$)	*true:* the relation is replaceable *false:* the relation is nonreplaceable
View-	extent ($\mathcal{VE}$)	$\underline{\equiv}$: the new extent is equal to the old extent $\supseteq$: the new extent is a superset of the old extent $\subseteq$: the new extent is a subset of the old extent $\approx$: the new extent could be anything

Fig. 3. View Evolution Parameters of E-SQL Language.

Example 3. Let's assume a web-based travel agency TRAV has a promotion for its customers who travel to Asia by air. TRAV will either going to send promotion letters to these customers or call them by phone. Therefore, it needs to find the customers' names, addresses, and phone numbers. Since an SQL view definition is static, we formulate this view in Eq. (1) using E-SQL, setting the view evolution parameters so that the view **Asia-Customer** may survive in a changing environment. Assume the company is willing to put off the phone marketing strategy, if the customer's phone number cannot be obtained, e.g., the information provider of the **Customer** relation decides to delete **Phone**. This preference is stated in the SELECT clause of Eq. (1) by the *attribute-dispensable parameter* $\mathcal{AD} = true$ for the attribute **Phone**. In addition, if the travel agent is willing to accept the customer information from other branches, we set the *relation-replaceable parameter* $\mathcal{RR}$ in the FROM clause to true for the relation **Customer**. Further, let's assume TRAV is willing to offer its promotion to *all* the customers who travel by air, if identifying who travels to Asia is impossible (i.e., the second WHERE condition cannot be verified). This preference can be explicitly specified by associating the *condition-dispensable parameter* $\mathcal{CC} = true$ with that condition in the WHERE clause.

```
CREATE VIEW Asia-Customer (VE =⊇)AS
SELECT      C.Name (AR = true), C.Addr (AR = true),
            C.Phone (AD = true, AR = false)
FROM        Customer C (RR = true), FlightRes F                  (1)
WHERE       (C.Name = F.PName)AND (F.Dest = 'Asia') (CD = true)
```

4 Formal Foundation for View Synchronization

We propose in [8] a three-step strategy for the view synchronization:
Step 1. Given a capability change *ch*, EVE system will first evolve the meta knowledge base MKB into MKB' by detecting and modifying the affected MISD descriptions found in the MKB.
Step 2. EVE detects all views affected either directly or indirectly (due to MKB evolution) by the capability change *ch*.
Step 3. Lastly, for affected yet potentially curable views we apply some view synchronization algorithm to find legal rewritings guided by constraints imposed by E-SQL evolution preferences from the view definition.
Due to limited space, the rest of the paper concentrates on the most difficult step of the view synchronization process, namely, the third one. In the remainder of this section, we introduce and formally define the concept of a *legal rewriting* for an affected view. In Section 5, we present an algorithm for view synchronization, referred to as Complex View Synchronization (or short, CVS) algorithm.
We assume SELECT-FROM-WHERE E-SQL views defined such that all *distinguished* attributes (i.e., the attributes used in the WHERE clause in an indispensable condition) are among the *preserved* attributes (i.e., the attributes in the SELECT clause). Plus, we assume that a relation appears at most once in the FROM clause of a view.

Definition 1. Let *ch* be a capability change, and MKB and MKB' be the state of the meta knowledge base containing the IS descriptions right before and right after the change *ch*, respectively. We say that a view V' is a **legal rewriting** of the view V under capability change *ch* if the following properties hold:
P1. The view V' is **no longer affected** by the change *ch*.
P2. The view V' can be **evaluated** in the new state of the information space (i.e., the view V' contains only elements defined in MKB').
P3. The **view extent parameter** $\mathcal{V}E_V$ of V (Fig. 3) is satisfied by the view V'. I.e., if $\bar{B}_V$ and $\bar{B}_{V'}$ are the attributes of interfaces of V and V', respectively, then

$$\pi_{\bar{B}_V \cap \bar{B}_{V'}}(V') \;\; \mathcal{V}E_V \;\; \pi_{\bar{B}_V \cap \bar{B}_{V'}}(V) \tag{2}$$

is satisfied for any state of the underlying information sources.
P4. All **evolution parameters** attached to the view elements such as attributes, relations or conditions of the view V are satisfied by the view V'. For example, any legal rewriting V' of the view V must have in the interface all indispensable attributes (i.e., the ones having $\mathcal{A}D = false$ in V) [3].

Example 4. Let an E-SQL view be defined as in Eq. (3) and the change *ch* is "delete attribute **Customer.Addr**".

[3] See [8] for a discussion of how evolution parameters are set for new components

$$\begin{array}{lll}
\text{CREATE VIEW} & \textbf{Asia-Customer (AName, AAddr, APh)} \ (\mathcal{VE} = \supseteq)\text{AS} & \\
\text{SELECT} & \textbf{C.Name, C.Addr}(\mathcal{AD} = \mathit{false}, \mathcal{AR} = \mathit{true}), \textbf{C.Phone} & \\
\text{FROM} & \textbf{Customer C, FlightRes F} & (3) \\
\text{WHERE} & \textbf{(C.Name = F.PName) AND (F.Dest = 'Asia')} &
\end{array}$$

$$\begin{array}{lll}
\text{CREATE VIEW} & \textbf{Asia-Customer' (AName, AAddr, APh)} \ (\mathcal{VE} = \supseteq)\text{AS} & \\
\text{SELECT} & \textbf{C.Name, } \underline{\textbf{P.PAddr } (\mathcal{AD} = \mathit{false}, \mathcal{AR} = \mathit{true})}, \textbf{C.Phone} & \\
\text{FROM} & \textbf{Customer C, FlightRes F, } \underline{\textbf{Person P}} & (4) \\
\text{WHERE} & \textbf{(C.Name = F.PName) AND (F.Dest = 'Asia')} & \\
 & \underline{\textbf{AND (P.Name = C.Name)}} &
\end{array}$$

We have to find a replacement for this attribute that could be obtained using constraints defined in MKB. Let's assume we have defined the following constraints in MKB:
(i) The relation **Person** is defined by **Person(Name, SSN, PAddr)**;
(ii) $\mathcal{JC}_{\textbf{Customer, Person}}$ = (**Customer.Name = Person.Name**);
(iii) $\mathcal{F}_{\textbf{Customer.Addr, Person.PAddr}}$ = (**Customer.Addr = Person.PAddr**);
(iv) $\mathcal{PC}_{\textbf{Customer, Person}} = (\pi_{\textbf{Name, PAddr}}(\textbf{Person}) \supseteq \pi_{\textbf{Name, Addr}}(\textbf{Customer}))$.
It is easily verifiable that the new view definition **Asia-Customer'** defined in Eq. (4) is a legal rewriting (new elements are underlined) conform to Def. 1.
We use $\mathcal{JC}_{\textbf{Customer, Person}}$ (defined in (ii)) to obtain the address from the relation **Person** by using in the WHERE clause the join relation (**Customer** $\bowtie_{\mathcal{JC}_{\textbf{Customer, Person}}}$ **Person**), and the function-of constraint defined in (iii). We can prove that the the extent parameter "$\mathcal{VE} = \supseteq$" is satisfied given the $\mathcal{PC}$ constraint from (iv). I.e., for any state of the relations **Customer**, **Person** and **FlightRes**, **Asia-Customer'** $\supseteq$ **Asia-Customer**.

5 View Synchronization: The CVS Algorithm

We now describe our solution for the third step of the view synchronization process given in Section 4, namely, the actual rewriting of an affected view definition. Four of the six capability change operations we consider can be handled in a straightforward manner. Namely, *add-relation*, *add-attribute*, *rename-relation* and *rename-attribute* capability changes do not cause any changes to existing (and hence valid) views.
However, the two remaining capability change operators, i.e., *delete-attribute* and *delete-relation*, cause existing views to become invalid and hence need to be addressed by the view synchronization algorithm. Below, we present the algorithm for handling the most difficult operator, namely, the *delete-relation* operator, in depth. The algorithm for the *delete-attribute* operator is a simplified version of it and is omitted in this paper due to space limitations.
We start by giving definitions of concepts needed to characterize valid replacements of view components (the assumptions made in Section 4 about view definitions are still valid here).

Example 5. To illustrate the steps of our approach for rewriting, we will use the view defined by Eq. (5) and the change operator "delete relation **Customer**" defined in the context of our Ex. 1. The view **Customer-Passengers-Asia** defines (*passenger*, *participant*) pairs of passengers flying to Asia and participants to a tour in Asia that fly and start the tour at the same day, respectively. Such a view could be used to see what participants of a tour are flying to "Asia" on the same day as the tour starts.

```
CREATE VIEW Customer-Passengers-Asia (𝒱E_V) AS
SELECT      C.Name (false,true), C.Age (true,true),
            P.Participant (true,true), P.TourID (true,true)
FROM        Customer C (true,true), FlightRes F (true,true),        (5)
            Participant P (true,true)
WHERE       (C.Name=F.PName)(false,true)AND(F.Dest='Asia')
            (P.StartDate = F.Date) AND (P.Loc = 'Asia')
```

Generally, a database schema can be represented as a hypergraph whose nodes are the attributes and whose hyperedges are the relations.We extent this representation for the MISD descriptions (relations, attributes, constraints) described in MKB by defining the hypergraph
$\mathcal{H}(MKB) = \{\ (\mathcal{A}(MKB))\ , (\mathcal{J}(MKB), \mathcal{S}(MKB), \mathcal{F}(MKB))\ \}$
whose components correspond to the set of attributes as hypernodes, and the set of join constraints, relations, and function-of constraints as hyperedges, respectively.

Example 6. Fig. 4 depicts the hypergraph for our travel agency example with:
$\mathcal{A}(MKB)$ = { **Name, Addr, Phone, Age, Tour.TourID, TourName, Tour.Type, NoDays, Participant, TourID, StartDate, Loc, PName, Airline, FlightNo, Source, Dest, Data, Holder, Type, Amount, Birthday, Hotels.City, Hotels.Address, Hotels.PhoneNumber, Company, City, PhoneNumber, Location** } (see Fig. 2);
$\mathcal{J}(MKB)$ = { **JC1, JC2, JC3, JC4, JC5, JC6** } (see Fig. 2);
$\mathcal{S}(MKB)$ = { **Customer, Tour, Participant, FlightRes, Accident-Ins, Hotels, RentACar** } (see Fig. 2);
$\mathcal{F}(MKB)$ = { **F1, F2, F3, F4, F5, F6, F7** } (see Fig. 2).

We say that a hypergraph is disconnected if one can partition its hyperedges into nonempty sets such that no hypernode appears in hyperedges of different sets. If such partition doesn't exist, then we say that the hypergraph is connected. Using these definitions, one can define connected sub-hypergraphs of a disconnected hypergraph as being its maximal connected components. For our problem, we are interested in finding the connected sub-hypergraph that contains a given relation R denoted by $\mathcal{H}_R(MKB)$. Note that because $\mathcal{JC}$-nodes are the only shared nodes between relation-edges in $\mathcal{H}(MKB)$ and because $\mathcal{H}_R(MKB)$ is a connected sub-hypergraph, we have: $\forall\ S_1,\ S_2 \in \mathcal{S}_R(MKB)$, there exists a sequence of join constraints $\mathcal{JC}_{S_1,R_1}, \ldots, \mathcal{JC}_{R_n,S_2}$ defined in MKB, with $R_1, \ldots, R_n \in \mathcal{S}_R(MKB)$ such that the following join relation can be defined $S_1 \bowtie_{\mathcal{JC}_{S_1,R_1}} R_1 \cdots \bowtie \cdots \bowtie_{\mathcal{JC}_{R_n,S_2}} S_2$.

Example 7. Fig. 4 depicts two connected sub-hypergraphs for the hypergraph $\mathcal{H}$(MKB) for Ex. 1. E.g., the connected sub-hypergraph $\mathcal{H}_{\mathbf{Customer}}(MKB)$ is the connected sub-hypergraph drawn on the top left of the Fig. 4.

Given a view definition referring to a relation R and an MKB, we want to determine which parts of the view need to be replaced when R is dropped. To find possible replacements, we look in the MKB for join constraints related to the relation R that are also used in the view definition. That is, the view could be seen as a join between a join relation defined using only join constraints from MKB and some other relations (the rest of the view definition). As we will show later, if R is to be dropped, our synchronization algorithm will try to substitute the affected part of the view definition with another join relation defined using join constraints from MKB. Def. 2 formally defines this relationship between a view definition and the (default) join constraints in MKB.

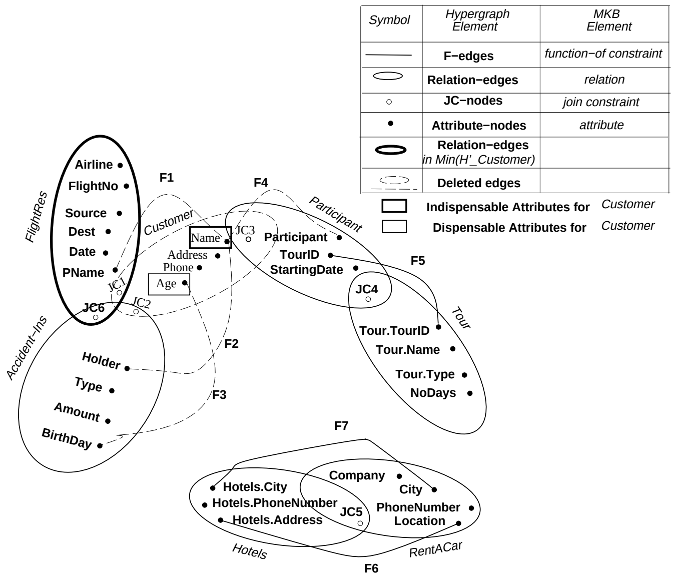

Fig. 4. The Hypergraphs $\mathcal{H}$(MKB) and $\mathcal{H}'$(MKB') for Ex. 1.

Definition 2. R-mapping of a view V into sub-hypergraph $\mathcal{H}_R(MKB)$. We define the R-mapping of V into $\mathcal{H}_R(MKB)$ by R-mapping(V, $\mathcal{H}_R(MKB)$) $= (Max(V_R), Min(\mathcal{H}_R))$ to be a pair of two subexpressions one constructed from the view V and the second one constructed from the connected sub-hypergraph $\mathcal{H}_R(MKB)$ such that the following must hold:
(I) The expression $Max(V_R)$ is of the form:

$$Max(V_R) \;=\; R_{v_1} \bowtie_{\mathcal{C}_{R_{v_1},R_{v_2}}} \cdots \bowtie_{\mathcal{C}_{R_{v_{l-1}},R_{v_l}}} R_{v_l} \tag{6}$$

such that relations $\{R_{v_1}, \ldots, R_{v_l}\} (\ni R)$ are from the FROM clause of V, and $\{\mathcal{C}_{R_{v_1},R_{v_2}}, \ldots, \mathcal{C}_{R_{v_{l-1}},R_{v_l}}\}$ are conjunctions of primitive clauses from the WHERE clause of V. A conjunction $\mathcal{C}_{R_{v_{s-1}},R_{v_s}}$ contains all the primitive clauses that use only attributes of relations $R_{v_{s-1}}$ and R_{v_s}.
(II) The expression $Min(\mathcal{H}_R)$ is of the form:

$$Min(\mathcal{H}_R) = R_{v_1} \bowtie_{\mathcal{J}C_{R_{v_1},R_{v_2}}} \cdots\cdots \bowtie_{\mathcal{J}C_{R_{v_{l-1}},R_{v_l}}} R_{v_l} \tag{7}$$

with $\{R_{v_1}, \ldots, R_{v_l}\} \subseteq \mathcal{S}_R(MKB)$, $\{\mathcal{J}C_{R_{v_1},R_{v_2}}, \ldots, \mathcal{J}C_{R_{v_{l-1}},R_{v_l}}\} \subseteq \mathcal{J}_R(MKB)$.
(III) The relation $Max(V_R)$ is *contained* in the relation $Min(\mathcal{H}_R)$:

$$Max(V_R) \subseteq Min(\mathcal{H}_R) \tag{8}$$

(IV) $Max(V_R)$ is maximal with the properties (I) and (III). I.e., there is no other relations from the FROM clause and primitive clauses from the WHERE clause of the view V that could be added to it and still be able to find a subexpression in $\mathcal{H}_R(MKB)$ such that (I) and (III) are satisfied.
(V) $Min(\mathcal{H}_R)$ is minimal with the properties (II) and (III). I.e., we cannot drop a relation or a join condition from it and still have (II) and (III) satisfied.

Def. 2 implies that there exists a conjunction of primitive clauses $\mathcal{C}_{Max/Min}$ such that

$$Max(V_R) = \sigma_{\mathcal{C}_{Max/Min}} (Min(\mathcal{H}_R)) \tag{9}$$

The goal of Def. 2 is to find the expressions $Max(V_R)$ and $Min(\mathcal{H}_R)$ such that the view V could be written as:

$$V = \pi_{\bar{B}_V}(\underbrace{(\sigma_{\mathcal{C}_{Max/Min}} (Min(\mathcal{H}_R)))}_{Max(V_R)} \bowtie_{\mathcal{C}_{Rest}} Rest) \tag{10}$$

where $\bar{B}_V$ is the view interface, $\mathcal{C}_{Rest}$ and $Rest$ are the rest of the primitive clauses and relations in V, respectively. $Rest$ is a join relation containing relations from the FROM clause that don't appear in $Min(\mathcal{H}_R)$.

Example 8. In Fig. 4, the minimal subexpression $Min(\mathcal{H}_{\textbf{Customer}})$ of $\mathcal{H}_{\textbf{Customer}}$(MKB) is marked by bold lines and corresponds to:

$$Min(\mathcal{H}_{\textbf{Customer}}) = \textbf{FlightRes} \bowtie_{\underbrace{\textbf{FlightRes.PName=Customer.Name}}_{\textbf{JC1}}} \textbf{Customer} \tag{11}$$

The maximal subexpression $Max(\textbf{Customer-Passenger-Asia}_{\textbf{Customer}})$ of the view defined by Eq. (5) and the relation **Customer** is:

$$Max(\textbf{Customer-Passenger-Asia}_{\textbf{Customer}}) = \tag{12}$$

$$= \textbf{FlightRes} \bowtie_{\underbrace{\begin{pmatrix}\textbf{(FlightRes.PName= Customer.Name)} \\ \text{AND } \textbf{(FlightRes.Dest='Asia')}\end{pmatrix}}_{\mathcal{C}_{\textbf{FlightRes, Customer}}}} \textbf{Customer}$$

$$= \sigma_{\underbrace{\textbf{FlightRes.Dest='Asia'}}_{\mathcal{C}_{Max/Min}}} (Min(\mathcal{H}_R))$$

The relation defined in Eq. (12) is contained in the relation defined by Eq. (11) and they are maximal and minimal, respectively, with this property (conform with Def. 2).

To find two expressions $Max(V_R)$ and $Min(\mathcal{H}_R)$ with the properties from Def.. 2 it is sufficient to have each join constraint $\mathcal{JC}_{S,S'}$ of expression $Min(\mathcal{H}_R)$ (Eq. (7)) implied by the corresponding join condition $\mathcal{C}_{S,S'}$ of expression $Max(V_R)$ (Eq. (6)), where $S, S' \in \{R_{v_1}, \ldots, R_{v_l}\}$. The algorithm for computing the R-mapping is straightforward and it is omitted here (see [8]).
Intuitively, we now have found the maximal part of the view definition that "relates" to our MKB (Def. 2). So now we can ask how this part (i.e., $Max(V_R)$) is affected by the relation R being dropped. And, further, we need to determine how we can find new join relations from the MKB that can replace affected view components in the view definition (i.e., $Max(V_R)$). The next definition identifies what are the most useful candidates for such replacement constructed using join constraints defined in MKB. At this point we don't worry about the relationship between the R-mapping and the potential candidates (e.g., subset, equivalent or superset). Our goal is to find all possible replacements for the relation $Max(V_R)$ (Eq. (10)). Only after that, when given the view-extent parameter $\mathcal{VE}_V$ (Section 3) and the $\mathcal{PC}$ constraints from MKB (Section 2), we want to choose the ones that satisfy the property P3 from Def. 1.

Definition 3. **R-replacement($V, \mathcal{H}_R(MKB)$).** For a view V and the MKB, we compute a set of expressions constructed from $\mathcal{H}_R(MKB)$ that don't contain R and could be used to meaningfully replace the maximal subexpression $Max(V_R)$ in V. Let MKB' be the meta knowledge base evolved from MKB when relation R is dropped; and $\mathcal{H}'_R(MKB')$ be the sub-hypergraph of $\mathcal{H}_R(MKB)$ obtained by erasing relation-edge R. We define R-replacement(V, $\mathcal{H}_R(MKB)$) $= \{Max(V_{1,R}), \ldots, Max(V_{l,R})\}$ to be a set of subexpressions constructed from $\mathcal{H}'_R(MKB')$ and $Max(V_R)$ such that $Max(V_{j,R})$ has the following properties:
(I) $Max(V_{j,R}) = \sigma_{\mathcal{C}'_{Max/Min}} \left(R_1 \bowtie_{\mathcal{JC}_{R_1,R_2}} \cdots \bowtie_{\mathcal{JC}_{R_{k-1},R_k}} R_k\right)$ with $R_1, \ldots, R_k$ and $\mathcal{JC}_{R_1,R_2}, \ldots, \mathcal{JC}_{R_{k-1},R_k}$ in $\mathcal{H}'_R(MKB')$.
(II) R doesn't appear in $Max(V_{j,R})$. I.e., R is not among $R_1, \ldots, R_k$.
(III) The expression $Min(\mathcal{H}_R)$ without R, $Min(\mathcal{H}'_R)$, could be mapped into $Max(V_{j,R})$. That is, if $Min(\mathcal{H}_R)$ is given by the Eq. (7) then: $\{R_{v_1}, \ldots, R_{v_l}\} \backslash \{R\} \subseteq \{R_1, \ldots, R_k\}$ and $\{\mathcal{JC}_{R_{v_1},R_{v_2}}, \ldots \mathcal{JC}_{R_{v_{l-1}},R_{v_l}}\} \backslash \{\mathcal{JC}_{S,S'} \mid S = R \, or \, S' = R\} \subseteq \{\mathcal{JC}_{R_1,R_2, \ldots \mathcal{JC}_{R_{k-1}}}, R_k\}$. I.e., the expression $Max(V_{j,R})$ must contain all the elements of the expression $Min(\mathcal{H}_R)$ unaffected by dropping relation R.
(IV) For any attribute $A \in R$ that is indispensable and replaceable in the view definition, the expression $Max(V_{j,R})$ contains a relation $S \in \{R_1, \ldots, R_k\}$ such that there exists a function-of constraint $\mathcal{F}_{R.A,S.B} = (R.A = f(S.B))$ in MKB. We call the relation S a **cover** for the attribute A and the attribute $f(S.B)$ a **replacement** for the attribute A in $Max(V_{j,R})$.
(V) The conjunction $\mathcal{C}'_{Max/Min}$ is obtained from conjunction $\mathcal{C}_{Max/Min}$ by substituting the attributes of R with their **replacements** (see (IV)) if any, or dropping primitive clauses that are dispensable and for which no replacement was

found for their attributes.

Erasing R from the connected sub-hypergraph $\mathcal{H}_R(MKB)$ could lead to a disconnected sub-hypergraph $\mathcal{H}'_R(MKB')$. If $\mathcal{H}'_R(MKB')$ is disconnected and the relations left in $Min(\mathcal{H}'_R)$ are in disconnected components then the set R-replacement$(V, \mathcal{H}_R(MKB))$ is empty. If relations left in $Min(\mathcal{H}'_R)$ are in a connected component of $\mathcal{H}'_R(MKB')$, the construction algorithm of the set $\{Max(V_{1,R}), \ldots, Max(V_{k,R})\}$ is following directly from Def. 3 (see [8]).

Example 9. In Fig. 4, the expression $Min(\mathcal{H}'_{\textbf{Customer}})$ defined by Eq. (11) is marked with bold lines: $Min(\mathcal{H}'_{\textbf{Customer}}) = (\textbf{FlightRes})$. We give now an example of R-replacements for the view defined by Eq. (5) and R = **Customer**. $\mathcal{H}'(MKB')$ is depicted in Fig. 4.
<u>Step 1.</u> In our example, using the hypergraph depicted in Fig. 4, we find:
$Cover$(**Customer.Name**) =
{ (**Accident−Ins**, **F2** = (**Customer.Name** = **Accident−Ins.Holder**)),
(**Participant**, **F4** = (**Customer.Name** = **Participant.Participant**)),
(**FlightRes**, **F1** = (**Customer.Name** = **FlightRes.PName**)) }.
<u>Step 2.</u> From Def. 3 (V), $\mathcal{C}'_{Max/Min}$ = (**FlightRes.Dest = 'Asia'**). Let's now construct the candidate expressions $Max(\textbf{Customer-Passenger-Asia}_{j,\textbf{Customer}})$ and define what is the replacement for the attribute **Customer.Name**.
(1) For the cover (**Accident−Ins**, (**Customer.Name** = **Accident−Ins.Holder**)) the expression below has all the properties from Def. 3. Similarly, we can construct $Max(\textbf{Customer-Passenger-Asia}_{2,\textbf{Customer}})$ from the third cover.

$$Max(\textbf{Customer-Passenger-Asia}_{1,\textbf{Customer}}) = \sigma_{\underbrace{(\textbf{FlightRes.Dest = 'Asia}}_{\mathcal{C}'_{Max/Min}}}$$

$$\left(\underbrace{\textbf{FlightRes}}_{Min(\mathcal{H}'_{\textbf{Customer}})} \bowtie_{\underbrace{(\textbf{FlightRes.PName = Accident-Ins.Holder})}_{\textbf{JC6}}} \underbrace{\textbf{Accident-Ins}}_{Cover(\textbf{Customer.Name})}\right.$$

(2) The cover (**Participant**,(**Customer.Name** = **Participant.Participant**)) cannot be used as replacement as there is no connected path in $\mathcal{H}'(MKB')$ (Fig. 4) that contains both the cover and the relation **FlightRes**.

Now we are ready to give the **Complex View Synchronization (CVS)** algorithm that has as input a view definition V, the MKB and a change "delete relation R", and returns all legal rewritings (see Def. 1) of the view V.
CVS(V, ch **=delete−relation** R**, MKB, MKB')**
INPUT: a SELECT-FROM-WHERE E-SQL view definition V;
change ch = *delete-relation* R;
MKB represented by the hypergraph $\mathcal{H}(MKB)$;
evolved MKB' represented by the hypergraph $\mathcal{H}'(MKB')$.
OUTPUT: A set of legal rewritings $V_1, \ldots V_l$ of V.
<u>Step 1.</u> Construct the sub-hypergraph $\mathcal{H}_R(MKB)$.
<u>Step 2.</u> Compute R-mapping$(V, \mathcal{H}_R(MKB)) = (Max(V_R), Min(\mathcal{H}_R))$ (Def. 2).

Step 3. Compute R-replacement$(V, \mathcal{H}'_R(MKB'))$ $=\{Max(V_{1,R}), \ldots, Max(V_{k,R})\}$ as defined in Def. 3. If R-replacement$(V,\ \mathcal{H}'_R(MKB') = \emptyset$ then the algorithm fails to find an evolved view definition for the view V.
Step 4. A synchronized view definition V' is found by replacing $Max(V_R)$ with $Max(V_{j,R})$ in Eq. (10); and then by substituting the attributes of R in V with the corresponding replacements found in $Max(V_{j,R})$. Because some more conditions are added in the WHERE clause (corresponding to the join conditions in $Max(V_{j,R})$), we have to check if there are no inconsistencies in the WHERE clause.
Step 5. Set the E-SQL evolution parameters for all V' obtained at Step 4.
Step 6. All the rewritings obtained by Step 4 have properties P1, P2, and P4 from Def. 1, Section 4. At this step, we have to check for which rewriting V' obtained in Step 4 the extent parameter $\mathcal{V}E_V$ of the view V is satisfied (property P3 from Def. 1) This problem is similar to the problem of answering queries using views which was extensively studied in the database community [6, 13]. However, our rewritings are not necessarily equivalent to the initial view, the relationship among them being imposed by the view-extent evolution parameter. We use the partial/complete information constraints defined in MKB' to compare the extents of the initial view V and the evolved view V'. This development is beyond the scope of current paper and it is part of our future work.

Example 10. For our view **Customer-Passenger-Asia** defined by Eq. (5), we now show how to apply Steps 4 and 5 from the algorithm CVS and find replacement under the change "delete relation **Customer**".
$Max(\textbf{Customer-Passenger-Asia}_{\textbf{Customer}}) =$
$$\textbf{FlightRes} \bowtie_{\left(\begin{array}{c}\textbf{(FlightRes.PName = Customer.Name) AND}\\ \textbf{(FlightRes.Dest = 'Asia')}\end{array}\right)} \textbf{Customer}$$
(Ex. 8, Eq. (12)) could be replaced, for example, with the following expression found at Step 3 of **CVS** (the second solution is similarly obtained from $Max(\textbf{Customer-Passenger-Asia}_{2,\textbf{Customer}})$):
(1) $Max(\textbf{Customer-Passenger-Asia}_{1,\textbf{Customer}}) =$
$\sigma_{\textbf{(FlightRes.Dest = 'Asia')}}(\ \textbf{FlightRes} \bowtie_{\textbf{(FlightRes.PName = Accident-Ins.Holder)}}$ $\textbf{Accident-Ins})$.

$$\begin{array}{ll}
\text{CREATE VIEW} & \textbf{Customer-Passengers-Asia}_1 \text{ AS} \\
\text{SELECT} & \underline{\textbf{A.Holder}\ (false, true)},\ \underline{\textbf{f(A.Birthday)}\ (true, true)}, \\
 & \underline{\textbf{P.Participant}\ (true, true),\ \textbf{P.TourID}\ (true, true)} \\
\text{FROM} & \underline{\textbf{Accident-Ins A}(true, true)},\ \textbf{FlightRes F}\ (true, true), \\
 & \underline{\textbf{Participant P}\ (true, true)} \\
\text{WHERE} & \underline{\textbf{(F.PName=A.Holder)}(false, true)}\text{AND}\textbf{(F.Dest='Asia')} \\
 & \overline{\textbf{(P.StartDate = F.Date)}\ \text{AND}\ \textbf{(P.Loc = 'Asia')}}
\end{array} \qquad (13)$$

For this particular case, we see that the attribute **Customer.Age** is also covered by the relation **Accident–Ins** with the function-of constraint **F3** = (**Customer.Age** = (*today* – **Accident–Ins.Birthday**)/365). In this case, we can replace the attribute **Customer.Age** in the view, too. A new rewriting of Eq. (5) using this substitution is given in Eq. (13).

6 Related Work

While no one has addressed the view synchronization problem itself before, there are several issues we address for EVE that relate to work done before in other contexts as outlined below.
Gupta et al. [3] and Mohania et al. [7] address the problem of materialized view maintenance after a view redefinition explicitly initiated by the user takes place. They study under which conditions this view maintenance can take place without requiring access to base relations, i.e., the self-maintainability issue.
The EVE system can be seen as an information integration system using view technology to gather and customize data across heterogeneous ISs [4, 12, 5, 8]. On this venue, related work that addresses the problem of information integration are among others the SIMS [1] and SoftBot [2] projects. In the SIMS project, the user interaction with the system is via queries posed against a unified schema. The SoftBot project has a very different approach to query processing as the system discovers the "link" among data sources. None of the two projects addresses the particular problem of evolution under IS changes.
Much research has been done on query reformulation using materialized views. For example, Levy et. al. [6, 13] consider the problem of replacing a query with a new query expression containing view definitions such that the new query is *equivalent* to the old one. To the best of our knowledge, there is no work done that has as purpose query reformulation without *equivalence* (e.g., the new query definition is a subset of the original view). We, on the other hand, have extended the notion of query reformulation by using E-SQL to specify constraints on query reformulation. Thus, when in compliance to those constraints, we allow the view redefinitions to be a subset or a superset of the original view.

7 Conclusion

To our knowledge, we are the first to study the problem of view synchronization caused by capability changes of participating ISs. In [12], we establish a taxonomy of view adaptation problems which distinguishes our new view synchronization problem, while in [4, 5] we lay the basis for the EVE solution framework. Formal criteria of correctness for view synchronization as well as actual algorithms for achieving view synchronization are the key contributions of this current work. To summarize, the main contributions of this paper are:
– We have formally presented the properties for *legal rewritings*.
– We have designed a solution approach for view synchronization that achieves view rewriting by exploiting chains of *multiple join constraints* given in the MKB.
– To demonstrate our solution approach, we have presented the *Complex View Synchronization (CVS)* algorithm for handling the most difficult capability change operator, namely, the "delete-relation" operator.
This work has opened a new problem domain important for a wide range of modern applications, and we thus expect that much future research will be conducted within the context of our proposed framework. Examples of work to be

done include the exploration of alternative view evolution preference models, MKB evolution and cost models for maximal view preservation.

References

1. Y. Arens, C. A. Knoblock, and W.-M. Shen. Query Reformulation for Dynamic Information Integration. *J. of Intelligent Information Systems*, 6:99–130, 1996.
2. O. Etzioni and D. Weld. A Softbot-Based Interface to the Internet. *Communication of ACM*, 1994.
3. A. Gupta, I.S. Mumick, and K.A. Ross. Adapting Materialized Views after Redefinition. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 1995.
4. A. J. Lee, A. Nica, and E. A. Rundensteiner. Keeping Virtual Information Resources Up and Running. In *Proc. of IBM Centre for Advanced Studies Conf. CASCON97, Best Paper Award*, pages 1–14, November 1997.
5. A. J. Lee, A. Nica, and E. A. Rundensteiner. The EVE Framework: View Evolution in an Evolving Environment. Technical Report WPI-CS-TR-97-4, Worcester Polytechnic Institute, Dept. of Computer Science, 1997.
6. Alon Y. Levy, Anand Rajaraman, and Jeffrey D. Ullman. Answering queries using limited external processors. In *Proc. of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–237, 1996.
7. M. Mohania and G. Dong. Algorithms for Adapting Materialized Views in Data Warehouses. *Int. Symposium on Cooperative Database Systems for Advanced Applications*, December 1996.
8. A. Nica, A.J . Lee, and E. A. Rundensteiner. View Synchronization with Complex Substitution Algorithms. Technical Report WPI-CS-TR-97-8, Worcester Polytechnic Institute, Dept. of Computer Science, 1997.
9. A. Nica and E. A. Rundensteiner. On Translating Loosely-Specified Queries into Executable Plans in Large-Scale Information Systems. In *Proc. of Second IFCIS Int. Conf. on Cooperative Information Systems CoopIS*, pages 213–222, 1997.
10. A. Nica and E. A. Rundensteiner. Loosely-Specified Query Processing in Large-Scale Information Systems. *Int. Journal of Cooperative Information Systems*, 1998.
11. Y. G. Ra and E. A. Rundensteiner. A transparent schema-evolution system based on object-oriented view technology. *IEEE Transactions on Knowledge and Data Engineering*, September 1997.
12. E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proc. of 4th Int. Workshop on Knowledge Representation Meets Databases (KRDB'97): Intelligent Access to Heterogeneous Information*, pages 13.1–13.11, Athens, Greece, August 1997.
13. D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy. Answering Queries with Aggregation Using Views. In *Proc. of Int. Conf. on Very Large Data Bases*, 1996.
14. J. Widom. Research Problems in Data Warehousing. In *Proc. of Int. Conf. on Information and Knowledge Management*, pages 25–30, November 1995.

This article was processed using the LaTeX macro package with LLNCS style