WPI-CS-TR-98-4 February 1998

Using Complex Substitution Strategies for View Synchronization

by

Anisoara Nica
Elke A. Rundensteiner

Using Complex Substitution Strategies for View Synchronization [*]

Anisoara Nica[†] and Elke A. Rundensteiner[‡]

(†) Department of EECS
University of Michigan, Ann Arbor
Ann Arbor, MI 48109-2122
anica@eecs.umich.edu
(313) 764-1571

(‡)Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
rundenst@cs.wpi.edu
(508) 831-5815

March 6, 1998

Abstract

Abstract

Large-scale information systems typically contain autonomous information sources (ISs) that dynamically modify their content, interfaces as well as their query services regardless of the data warehouses (views) that are built on top of them. Current view technology fails to provide adaptation techniques for such changes giving support to only *static views* in the sense that views become undefined when ISs undergo capability changes. We propose to address this new view evolution problem - which we call *view synchronization* - by allowing view definitions to be dynamically evolved when they become undefined. The foundations of our approach to view synchronization include: the Evolvable-SQL view definition language (E-SQL), the model for information source description (MISD), and the concept of legal view rewritings. In this paper, we now introduce the concept of the strongest synch-equivalent view definition that explicitly defines the evolution semantics associated with an E-SQL view definition. Plus, we propose a strategy and proofs of correctness for transforming any user-specified E-SQL view definition into the strongest E-SQL query. We also present the Complex View Synchronization (CVS) algorithm that fully exploits the constraints defined in MISD by allowing relation substitution to be done by a sequence of joins among candidate relations. Examples illustrating this multi-step approach are given throughout the paper.

Keywords: Self-adapting views, view synchronization and preservation, data warehouse, large-space information space, information descriptions, evolving information sources.

1 Introduction

Advanced applications such as web-based information services, data warehousing, digital libraries, and data mining typically operate in an information space populated with a large number of dynamic information sources

[*]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.

(ISs) such as the WWW [Wid95]. The ISs in such environments are usually distributed, have distinct schemas, support different query languages, update not only their content but also their capabilities[1], and even join or leave the environment frequently. 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*.

Views in such environments introduce new challenges to the database community [Wid95]. In our prior work [RLN97, LNR97a], 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. The problem is that current view technology is insufficient for supporting *flexible* view definitions. That is, under current view technology, views are *static*, meaning views are assumed to be specified on top of a fixed environment and once the external 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 [RLN97, LNR97b, NLR97].

Namely, we have designed a framework for view adaptation in these evolving environments, called *EVE* (Evolvable View Environment), which supports to "preserve 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 that dealt with view redefinition (e.g., by Gupta et al. [GJM96] and Mohania et al. [MD96]) typically assumed that the view redefinition was explicitly requested by the view developer at the view site, while the underlying information sources remained unchanged. Furthermore, previous work Gupta et al. [GJM96], Mohania et al. [MD96], etc., has 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 the view definition language E-SQL (essentially 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. For example, a view definer could indicate that the attribute **Name** is indispensable to the view, whereas the attribute **Address** is desirable yet can be omitted from the original view definition, if keeping it becomes impossible, without jeopardizing the utility of the view.

A second key component of our *EVE* framework is a language for capturing descriptions of the content, capabilities as well as interrelationships of all ISs in the system. Descriptions of ISs expressed in this language are maintained in a meta-knowledge base (MKB) available to the view synchronizer during the view evolution process. In order to keep our approach general we only consider basic types of constraints in our model that are likely to be applicable to a wide range of information sources ranging from more structured DBMSs to more unstructured web resources. For this reason, constraints such as keys and functional dependencies that are not captured by models of most ISs are not relied upon in our approach. Instead we focus on the more difficult problem of how to perform query rewritings even when given only minimal amounts of meta-knowledge.

[1] Capabilities here refer to information such as their schema, their query interface, as well as other services offered by the information source.

Given a view defined in E-SQL and a MKB, we present in this paper a formal foundation for what are legal rewritings of the view affected by capability changes. This includes both evolution semantics associated with E-SQL evolution parameters as well as properties that new components used to replace the obsolete ones must have. The E-SQL view definition language allows any combination of the evolution parameters to be set for the view components in order to simplify the specifications task for the users. Because of the relationships between components, e.g., an attribute in the SELECT clause is coming from a relation in the FROM clause, the synchronization process cannot take full advantage of the apparent flexibility of all combinations of evolution parameters. We introduce the synch-equivalence concept to express the real evolution flexibility of an E-SQL view definition. We also present a strategy for finding the strongest synch-equivalent E-SQL definition for a given view specification, and prove that our transformation rules are correct. Finding the synch-equivalent definitions is essential for helping the view definer to understand the semantics associated with a view definition and makes the implicit E-SQL evolution semantics explicit.

Based on this formal foundation, we then propose a strategy for solving the view synchronization problem. Our view synchronization algorithm finds valid replacements for affected (deleted) components of the existing view definitions based on the semantic constraints captured in the MKB. Rather than just providing simple so-called 'one-step-away' view rewriting [LNR97b, LNR97a], these replacements may correspond to possibly complex pieces of information from several ISs. For this, our solution succeeds in determining view rewritings 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 would have simply disabled the view, and where our proposed one-step view synchronization (SVS) [LNR97b] would have failed to locate a suitable solution.

The remainder of the paper is structured as follows. In Section 2, we present our *EVE* solution approach. In Section 3, we present the IS description model MISD. The extended view definition language E-SQL designed to model evolution preferences is presented in Section 4. Section 5 describes the formal basis for correct view synchronization, while Section 6 introduces the concept of the strongest synch-equivalent view definition and provides algorithms of how to compute it. In Section 7 we introduce the CVS algorithm for synchronizing views based on the proposed formal model. Section 8 lists related work, and Section 9 presents our conclusions.

2 Background

2.1 *EVE*: The Evolveable View Environment

We give a brief architectural overview of the Evolveable View Environment (*EVE*) we have designed for view synchronization in order to set the context for the remainder of this paper (Figure 1). The *EVE* environment is populated by a possibly large number of heterogeneous ISs. These ISs are autonomous in the sense that they are free to change their capabilities dynamically at any time without regard to views defined upon them. Similar to other large-scale systems [NR98a], an IS is integrated into the *EVE* framework via a wrapper (e.g., [PGMU96]), called the Information Source Interface (ISI), that translates between IS's native language and model to a common model understood by the *EVE* system. While the *EVE* framework is generally applicable, we restrict ourselves in this paper to *EVE*'s common data model being a relational one. The ISI is assumed to be intelligent enough to extract not only raw data, but also metadata about the IS, such as changes at the schema level of the IS, performance data, or relationships with other ISs.

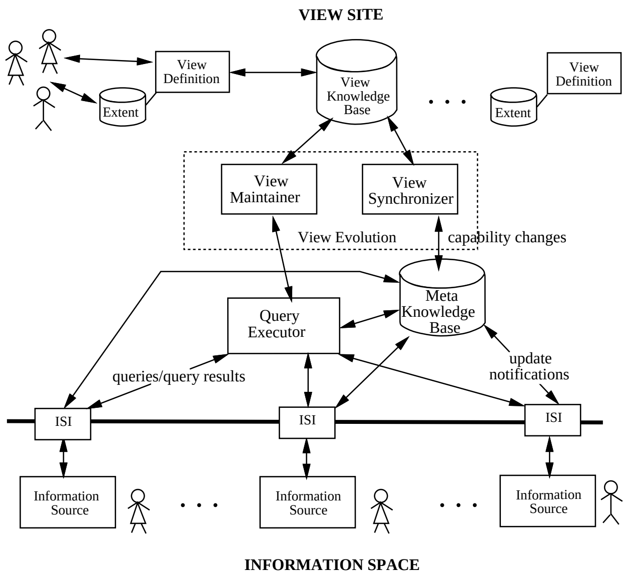

Figure 1: The *EVE* Framework: View Synchronization in an Evolving Environment.

When an IS joins the *EVE* framework, it advertises to the meta knowledge base (MKB) its capabilities (e.g., the type of queries or interfaces supported), data model (e.g., the semantic mappings from its concepts to the concepts already in the MKB), and data content (see Section 3) ([NR97])[2]. The IS descriptions collected in the MKB are critical for identifying alternate view definitions when synchronizing a view (see Section 7). The knowledge about the views, such as the E-SQL view definitions and the locations of the views, is stored in the view knowledge base (VKB).

Our *view evolution* module consists of two tools: the *view synchronizer* (the topic of this paper) that evolves the specifications of views themselves and the *view maintainer* (not further focussed upon in this paper) that is responsible for bringing the view extents up-to-date after view synchronization in the case when the views are materialized.

After *EVE* learns about a capability change of an IS, the *view synchronizer* identifies all views in the VKB that are potentially affected by a capability change of an IS. It then explores alternate techniques for query rewriting of the view definition VD of an affected view with the goal of meeting all view preservation constraints in the

[2]The information providers have strong economic incentives to provide the meta knowledge of their individual ISs as well as of the relationships with other ISs, since populating the MKB not only advertises their resources to potential view users, but it also increases the utilization of their data set in cases where users of other competitive ISs are in need of alternate sources (especially, if they offer the same information at a better price).

view definition (expressed in E-SQL). For this, it extracts appropriate information from the MKB about other ISs so to use this as replacement of the affected view components. The *view synchronization* is concerned with synchronizing the affected view definition so that it remains both valid and consistent with the view evolution parameters as well as with all IS constraints kept in the MKB.

After *view synchronization* is complete, the *view maintainer* works with the *query executor* to get appropriate information from one or more ISs in order to fix the extents of all evolved views. In this context, the *query executor* translates query requests into several smaller query fragments to be sent to individual ISs and then assembles the results submitted from the ISs into one coherent response.

2.2 Running Example: The Travel Agency Service

To demonstrate our solution approach, we use a travel agency service provider as running example. Below we describe the relevant base information (expressed using relations in our system), while additional relations and views are defined on top of these relations later in the paper, as needed.

IS 1: Personal Customer Information
Content Description: Customer(Name, Address, PhoneNo, Age)
IS 2: Vacation Tour Information
Content Description: Tour(TourID, TourName, Type, NoDays)
IS 3: Vacation Tour Participants Information
Content Description: Participant(Participant, TourID, StartDate, Location)
IS 4: Flight Reservation Information
Content Description: FlightRes(PName, Airline, FlightNo, Source, Dest, Date)
IS 5: Insurance Information
Content Description: Accident–Ins(Holder, Type, Amount, Birthday)
IS 6: Hotel Information
Content Description: Hotels(City, Address, PhoneNumber)
IS 7: Renting Companies Information
Content Description: RentACar(Company, City, PhoneNumber, Location)

Figure 2: Information Sources Content Descriptions

Example 1 *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. Therefore, the travel agency needs to access many disparate information sources, including domestic as well as international sites. Since the connections to external information sites, such as the overseas branches, are very expensive and have low availability, the travel agency materializes the query results (views) at its headquarter or other US branches (at the view site). A part of relevant ISs is summarized in the table in Figure 2.*

Assume the headquarter maintains complete sets of information of the customers, tours, and tour participants in the following formats: ***Customer(Name, Address, PhoneNo, Age)****,* ***Tour(TourID, TourName, Type, NoDays)*** *- where* ***Type*** $= \{luxurious, economy, super\text{-}valued\}$*, and* ***Participant(Participant, TourID, StartDate, Location)*** *that states which customer joins which tour starting on what day. We further assume the local branches keep partial sets of information of its local customers, the tours offered locally, and the participation information of its local customers. The flight reservation information* ***FlightRes(PName, Airline, FlightNo, Source, Dest, Date)*** *is managed by each individual airline company. Insurance information*

Accident–Ins(Holder, Type, Amount, Birthday) *is kept by each individual insurance company.*

3 MISD: Model for Information Source Description

While individual ISs could be based on any data model, we assume that the information source interface (ISI) agent of an IS describes the schema exported by the IS as a set of relations $IS.R_1$, $IS.R_2, \ldots, IS.R_n$ that reside at the IS. 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) (see Figure 1) and are used in the process of view evolution, when alternative rewritings must be found for the affected views after a capability change at an IS[3]. Below we introduce the MISD model as used by the remainder of this paper. All MISD constraints are summarized in Figure 3 [LNR97a, LNR97b].

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	$\mathcal{JC}_{R_1,R_2} = (C_1 \text{ AND } \cdots \text{ AND } C_l)$
Partial/Complete Constraint	$\mathcal{PC}_{R_1,R_2} = (\pi_{\bar{A}_1}(\sigma_{\mathcal{C}(\bar{B}_1)} R_1)\ \theta\ \pi_{\bar{A}_2}(\sigma_{\mathcal{C}(\bar{B}_2)} R_2))$ $\theta \in \{\subset, \subseteq, \equiv, \supseteq, \supset\}$

Figure 3: Semantic Constraints for IS Descriptions.

3.1 Data Content Description

The model used to describe the basic units of information available in each of the ISs is the relational model. A base relation is an n-ary relation, with $n \geq 2$. A relation name is not required to be unique in the MKB, but the pair (IS name, relation name) is. A relation R is described by specifying its information source and its set of attributes as follows:

$$IS.R(A_1, \ldots, A_n). \tag{1}$$

3.2 Type Integrity Constraints

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 with the format $Type_i(A_i)$. A *type constraint* for a relation $R(A_1, \ldots, A_n)$ is specified as:

$$R(A_1, \ldots, A_n) \subseteq Type_1(A_1), \ldots, Type_n(A_n) \tag{2}$$

which says that an attribute A_i is of type $Type_i$, for $i = 1, \ldots, n$. For simplicity, we assume that the attribute types are primitive. If two attributes are exported with the same name, they are assumed to have the same type (which must be reflected by the type constraints for their relations).

[3]We assume in this paper that information sources support at least SPJ (SQL) queries with conjunctions of primitive clauses in the WHERE clause, and thus we are not discussing how query capabilities are described in our system [NR98a].

3.3 Order Integrity Constraints

The *order constraints* specify data constraints that are satisfied by any tuple of a relation at any time. For a relation $R(A_1, \ldots, A_n)$, an order constraint is defined by:

$$R(A_1, \ldots, A_n) \subseteq \mathcal{C}(A_{i_1}, \ldots, A_{i_k}) \tag{3}$$

where $(A_{i_1}, \ldots, A_{i_k})$ is a subset of the attributes of R and $\mathcal{C}(A_{i_1}, \ldots, A_{i_k})$ is a conjunction of primitive clauses defined over the attributes. A primitive clause has one of the following forms: $< attribute - name > \theta < attribute - name >$ or $< attribute - name > \theta < value >$ with $\theta \in \{<, \leq, =, \geq, >\}$. Expression (3) specifies that for any state of the database R, and for any tuple $t \in R$, $\mathcal{C}(t[A_{i_1}], \ldots, t[A_{i_k}])$[4] is satisfied.

Example 2 *An insurance relation* ***Expensive-Insurance****, containing all expensive accidental insurances that cover more than* $\$1,000,000$*, can be expressed by the following order constraint:*

$$\textbf{\textit{Expensive-Insurance(Holder, Type, Amount, Birthday)}} \subseteq (\textbf{\textit{Amount}} > 1,000,000).$$

3.4 Join Constraints

In order to evolve views defined over a set of relations exported from different ISs, the IS description also contains a specification to characterize the IS content in terms of its relationship with other sources in the environment. A join constraint is used to specify a meaningful way to combine information from two ISs, i.e., a join condition between two relations is used to capture our knowledge of their interrelationship. The join condition is a conjunction of primitive clauses (not necessarily equijoin clauses). Formally, a join constraint is of the form:

$$\mathcal{JC}_{R_1,R_2} = (C_1 \text{ AND } \cdots \text{ AND } C_l) \tag{4}$$

where $C_1, \ldots, C_l$ are primitive clauses over the the attributes of R_1 and R_2. Expression (4) gives a default, legal 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.

Example 3 *For our running Example 1, some of the join constraints are given in the table of Figure 4 (the underlined names are the relations for which the join constraints are defined).*

For example, for the relations ***Customer(Name, Address, PhoneNo, Age)*** *and* ***Person(SSN, Name, PermanentAddress)****, the join constraint* $\mathcal{JC}_{\textbf{\textit{Person, Customer}}}$ = ***(Person.Name=Customer.Name*** AND ***Person.PermanentAddress=Customer.Address)*** *states that the two relations* ***Customer*** *and* ***Person*** *can be meaningfully joined on the attributes* ***Name*** *and* ***Address****.*

[4]The expression $t[A]$ refers to the value of the attribute A in the tuple t.

$\mathcal{JC}$	**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**

Figure 4: Join Constraints for Example 1

3.5 Partial/Complete Information Constraints

The *partial/complete* ($\mathcal{PC}$) information constraints make it possible to describe that a fragment of a relation is part of or equal to a fragment of another relation for all extents of the two relations. The $\mathcal{PC}$ constraints are used to decide if an evolved view is equivalent, subset of, or superset of the initial view. For two relations R_1 and R_2, the $\mathcal{PC}$ information constraint is given by:

$$\mathcal{PC}_{R_1,R_2} = \quad (\pi_{A_{i_1},\ldots,A_{i_k}}(\sigma_{\mathcal{C}(A_{j_1},\ldots,A_{j_l})} R_1) \;\; \theta \;\; \pi_{A_{n_1},\ldots,A_{n_k}}(\sigma_{\mathcal{C}(A_{m_1},\ldots,A_{m_t})} R_2)) \tag{5}$$

where θ is $\{\subseteq, \supseteq, \equiv\}$ for the partial ($\subseteq$ and $\supseteq$) or complete ($\equiv$) information constraint, respectively; $A_{i_1},\ldots,A_{i_k},A_{j_1},\ldots,A_{j_l}$ are attributes of R_1; and $A_{n_1},\ldots,A_{n_k},A_{m_1},\ldots,A_{m_t}$ are attributes of R_2. The sets $A_{i_1},\ldots,A_{i_k}$ and $A_{n_1},\ldots,A_{n_k}$ are such that for $s = \overline{1,k}$ the attributes A_{i_s} and A_{n_s} have the same type.

Example 4 *To give an example, Eq. (6) states that the relations* ***Person*** *and* ***Customer*** *have the same data for the attributes* ***Name*** *and* ***Address*** *for customers age* 1 *or older. (This means that the relation* ***Person*** *contains* ***SSN*** *only for persons older than* 1 *year.)*

$$\mathcal{PC}_{\textit{Person, Customer}} = \quad (\pi_{\textit{Name, PermanentAddress}}(\boldsymbol{Person}) \quad \equiv \pi_{\textit{Name, Address}}(\sigma_{\textit{Age}>1}\boldsymbol{Customer})) \tag{6}$$

Using the $\mathcal{PC}$ information constraints and type constraints we can, for example, define that two relations are equivalent: (1) they have attributes of the same types (expressed by type constraints); and (2) their extents are the same (expressed by a $\mathcal{PC}$ constraint).

3.6 Attribute Function-Of Constraints

The **attribute function-of constraint** relates two attributes by defining a function to transform one of them into another. This constraint is specified by:

$$\mathcal{F}_{R_1.A,R_2.B} = (\; R_1.A = f(R_2.B) \;) \tag{7}$$

where f is a function[5]. The function-of constraint $\mathcal{F}_{R_1.A,R_2.B}$ 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 the join relation J, we have $t[R_1.A] = f(t[R_2.B])$.

Example 5 *For our running Example 1, function-of constraints are given in the table of Figure 5.*

[5]Note that the inverse of f is not required to exist, and hence if an inverse is available it must be explicitly listed as $R_2.B = f^{-1}(R_1.A)$.

$\mathcal{F}$	**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**

Figure 5: Function-of Constraints for Example 1.

4 Extending SQL for Flexible View Synchronization

In this section, we present the *EVE* view definition language (E-SQL), which is an extension of 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. In this paper, we assume SELECT-FROM-WHERE views defined as in Equation (8) with a conjunction of primitive clauses in the WHERE clause.

$$\begin{array}{ll} \text{CREATE VIEW} & V\ (B_1, \ldots, B_m)\ \ (\mathcal{VE} = \delta_V)\ \text{AS} \\ \text{SELECT} & R_1.A_{1,1}(\mathcal{AD} = \mathcal{AD}_{1,1}, \mathcal{AR} = \mathcal{AR}_{1,1}), \ldots, R_1.A_{1,i_1}(\mathcal{AD} = \mathcal{AD}_{1,i_1}, \mathcal{AR} = \mathcal{AR}_{1,i_1}), \ldots \\ & R_n.A_{n,1}(\mathcal{AD} = \mathcal{AD}_{n,1}, \mathcal{AR} = \mathcal{AR}_{n,1}), \ldots, R_n.A_{n,i_n}(\mathcal{AD} = \mathcal{AD}_{n,i_n}, \mathcal{AR} = \mathcal{AR}_{n,i_n}) \\ \text{FROM} & R_1(\mathcal{RD} = \mathcal{RD}_1, \mathcal{RR} = \mathcal{RR}_1), \ldots, R_n(\mathcal{RD} = \mathcal{RD}_n, \mathcal{RR} = \mathcal{RR}_n) \\ \text{WHERE} & C_1(\mathcal{CD} = \mathcal{CD}_1, \mathcal{CR} = \mathcal{CR}_1)\ \textbf{AND}\ \ldots\ \textbf{AND}\ C_k(\mathcal{CD} = \mathcal{CD}_k, \mathcal{CR} = \mathcal{CR}_k) \end{array} \quad (8)$$

Figure 6: Syntax of E-SQL View Definition.

View Evolution Parameters			
Evolution Parameter		Semantics	Default Value
Attribute-	dispensable ($\mathcal{AD}$)	*true:* the attribute is dispensable *false:* the attribute is indispensable	false
	replaceable ($\mathcal{AR}$)	*true:* the attribute is replaceable *false:* the attribute is nonreplaceable	false
Condition-	dispensable ($\mathcal{CD}$)	*true:* the condition is dispensable *false:* the condition is indispensable	false
	replaceable ($\mathcal{CR}$)	*true:* the condition is replaceable *false:* the condition is nonreplaceable	false
Relation-	dispensable ($\mathcal{RD}$)	*true:* the relation is dispensable *false:* the relation is indispensable	false
	replaceable ($\mathcal{RR}$)	*true:* the relation is replaceable *false:* the relation is nonreplaceable	false
View-	extent ($\mathcal{VE}$)	$\equiv$: 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 $\approx$: no restrictions on the new extent	$\equiv$

Figure 7: View Evolution Parameters of E-SQL Language.

As indicated in Figure 7, each component of the view definition (i.e., attribute, relation or condition) has attached two evolution parameters. One, the *dispensable parameter* (notation $\mathcal{XD}$, where $\mathcal{X}$ could be $\mathcal{A}$, $\mathcal{R}$ or $\mathcal{C}$ for attribute, relation or condition component, respectively) specifies if the component could be dropped (value *true*) or must be present in any evolved view definition (value *false*). Two, the *replaceable parameter* (notation $\mathcal{XR}$, where again $\mathcal{X}$ could be $\mathcal{A}$, $\mathcal{R}$ or $\mathcal{C}$ for attribute, relation or condition component, respectively) specifies if the component could be replaced in the process of view evolution (value *true*) or must be left unchanged as defined in the initial view (value *false*).

In Figure 7, each type of evolution parameter used by E-SQL is represented by a row in that table. Figure 7 has three columns: column one gives the parameter name and the abbreviation for each parameter, column two the possible values each parameter can take on plus the associated semantics, and column three the default value. When the parameter setting is omitted from the view definition, then the default value is assumed. This means that a conventional SQL query (without explicitly specified evolution preferences) has well-defined evolution semantics in our system, i.e., anything the user specified in the original view definition must be preserved exactly as originally defined in order for the view to be well-defined. Our extended view definition semantics are thus well-grounded and compatible with current view technology.

The general format of the extended view definition language is given in Eq. (8) in Figure 6. The view interface $\bar{B}_V = (B_1, \ldots, B_m)$ corresponds to the local names given to attributes preserved in the view V, the set $\{A_{j,1}, \ldots, A_{j,i_j}\}$ is a subset of the attributes of the relation R_j for all $j = \overline{1,n}$; any C_i, $i = \overline{1,k}$, is a primitive clause defined over the attributes of relations in the FROM clause. All parameters $\mathcal{VE}, \mathcal{AD}, \mathcal{AR}, \mathcal{RD}, \mathcal{RR}, \mathcal{CD}$, and $\mathcal{CR}$ are defined as described in Figure 7.

Next, we use one example to demonstrate the usage of and interactions among proposed evolution parameters, while an extensive justification for the design of this language plus many more examples can be found in [NLR97].

Example 6 *In our Example 1, let's assume that the travel agency has a promotion for the customers who travel to Asia. Therefore, the travel agency needs to find the customers' names, addresses, and phone numbers. The travel agency is either going to send promotion letters to these customers or call them by phone. The query for getting the necessary information can be specified as follows:*

```
CREATE VIEW   Asia-Customer AS
SELECT        Name, Address, PhoneNo
FROM          Customer C, FlightRes F                    (9)
WHERE         (C.Name = F.PName)
              AND (F.Dest = 'Asia')
```

Eq. (9) is a static SQL query. Next, we incorporate view evolution parameters into Eq. (9) that indicate restrictions and preferences on how the view ***Asia-Customer*** *may be evolved when the environment changes.*

Assume the travel agency is willing to accept the query results with the customer's names and addresses only. That is, the company is okay to put off the phone marketing strategy, if the customer's phone number attribute ***PhoneNo*** *is deleted from the relation* ***Customer*** *for some reason and a suitable substitute cannot be found. The* *user can state this preference in the SELECT clause (Eq. (9)) by using the attribute dispensable parameter $\mathcal{AD}$.*

$$\text{SELECT} \quad \textbf{Name}\ (\mathcal{AD} = false), \textbf{Address}\ (\mathcal{AD} = false), \textbf{PhoneNo}\ (\mathcal{AD} = true)$$

The user may want to guide the system as to whether it is acceptable for an attribute to be obtained from other sources besides the original relation. For example, if the user only accepts the customer name and address to come from the relation ***Customer****, but agrees to have the phone number come from other source(s), then the user can augment the SELECT clause (Eq. (9)) with the attribute replaceable parameter.*

$$\text{SELECT} \quad \textbf{Name}\ (\mathcal{AR} = false), \textbf{Address}\ (\mathcal{AR} = false), \textbf{PhoneNo}\ \ (\mathcal{AR} = true)$$

Further, let's assume the person who defines the ***Asia-Customer*** *view is willing to accept a view without the second (local) condition specified, as long as the equijoin condition is kept*[6]*. As a consequence (if the local condition is dropped), the promotion invitation letters are sent to all customers traveling by air. The user can specify her preference by adding the condition-dispensable parameter to the conditions in the WHERE clause of Eq. (9).*

$$\text{WHERE}\ (\textbf{C.Name=F.PName})(\mathcal{CD} = false)\ \text{AND}(\textbf{F.Dest='Asia'})(\mathcal{CD} = true)$$

If the user requires the redefined view extent to be either equivalent to or larger than the original view extent, the user sets the view-extent parameter ($\mathcal{VE}$) to $\supseteq$. This means any substitution of a relation, condition, or attribute should make the new view extent at least as large as the original view extent for the view synchronization process to be valid. For example, if originally the ***Asia-Customer*** *view returns the customers who travel to Japan, Korea, or Hong Kong, then the view is still valid if in addition to these customers it also returns the customers who travel to Thailand and Malaysia.*

$$\text{CREATE VIEW} \quad \textbf{Asia-Customer}\ \text{AS}\ \ (\mathcal{VE}\ = \supseteq)$$

Putting together all view evolution parameters proposed above with the initial view definition from Eq. (9), we get Eq. (10).

$$\begin{array}{ll} \text{CREATE VIEW} & \textbf{Asia-Customer}\ (\mathcal{VE} = \supseteq)\ \text{AS} \\ \text{SELECT} & \textbf{Name, Address, PhoneNo}\ \ (\mathcal{AD} = true,\ \mathcal{AR} = true) \\ \text{FROM} & \textbf{Customer C}\ (\mathcal{RR} = true), \textbf{FlightRes F} \\ \text{WHERE} & (\textbf{C.Name = F.PName})\ \text{AND}\ (\textbf{F.Dest = 'Asia'})\ \ (\mathcal{CD} = true) \end{array} \tag{10}$$

In Eq. (10), wherever no view evolution parameter values are specified for a view component, then the default values are assumed as indicated in Figure 7. To name a few, **Name** *and* **Address** *attributes in the select clause are indispensable and nonreplaceable, and the relation* **FlightRes** *is indispensable and nonreplaceable.*

[6]Note that in general dropping a local condition is more acceptable than dropping a join condition, since dropping a join condition may change the view definition dramatically. For example, removing the only join condition between two relations, that returns some subset of tuples, ends up with a Cartesian product of these two relations, which then returns all pairwise combinations of tuples from both relations as the view result.

5 Formal Foundation for View Synchronization

In this section we give a formal definition of what is considered to be a *legal view rewriting* for a view which became obsolete due to a capability change of an underlying information source.

5.1 Notations

First we introduce some basic definitions that are used to introduce the concept of legal view rewritings. For two relations R and R' with different attribute sets (i.e., denoted by $Attr(R)$ and $Attr(R')$) such that $Attr(R) \cap Attr(R') \neq \emptyset$, we compare the extents of the two relations by comparing the projections on their common attributes. Definitions 1 and 2 introduce this concept of equivalence with respect to common subset of attributes, which we call π-*equivalence*, while Table 8 summarizes the other set operations defined on this common-subset-of-attributes notion.

Definition 1 ***Common-Subset-of-Attributes of*** R ***with respect to*** R'. *Let* R *and* R' *be two relations.* $R^{\pi(R')}$ *denotes the projection of relation* R *on the common attributes of* R *and* R'. *That is,* $R^{\pi(R')} = \pi_{Attr(R)\cap Attr(R')}R$.

Definition 2 π***-Equivalence.*** *We say that a relation* R *is* π*-equivalent with relation* R' *denoted by* $R \equiv_\pi R'$*, iff*

(I) $\forall t \in R, \exists\, t' \in R'$ *s.t.* $t[Attr(R) \cap Attr(R')] = t'[Attr(R) \cap Attr(R')]$. *That is,* $R^{\pi(R')} \subseteq R'^{\pi(R)}$.

(II) $\forall t' \in R', \exists\, t \in R$ *s.t.* $t'[Attr(R) \cap Attr(R')] = t[Attr(R) \cap Attr(R')]$. *That is,* $R'^{\pi(R)} \subseteq R^{\pi(R')}$.

Name	Set Operator	Semantics
π-equivalent	$R =_\pi R'$	$\forall\, t' \in R', \exists\, t \in R$ s.t. $t'[Attr(R) \cap Attr(R')] = t[Attr(R) \cap Attr(R')]$ and $\forall\, t' \in R', \exists\, t \in R$ s.t. $t'[Attr(R) \cap Attr(R')] = t[Attr(R) \cap Attr(R')]$
π-subset	$R' \subseteq_\pi R$	$\forall\, t' \in R', \exists\, t \in R$ s.t. $t'[Attr(R) \cap Attr(R')] = t[Attr(R) \cap Attr(R')]$
π-superset	$R' \supseteq_\pi R$	$\forall\, t \in R, \exists\, t' \in R'$ s.t. $t[Attr(R) \cap Attr(R')] = t'[Attr(R) \cap Attr(R')]$
π-intersection	$R \cap_\pi R'$	$\{z \mid \exists\, t \in R \text{ and } \exists\, t' \in R',\ s.t., t[Attr(R) \cap Attr(R')] = t'[Attr(R) \cap Attr(R')];$ $z = t[Attr(R) \cap Attr(R')]\}$
π-difference	$R \setminus_\pi R'$	$\{z \mid \exists\, t \in R \text{ and } \nexists\, t' \in R',\ s.t. t[Attr(R) \cap Attr(R')] = t'[Attr(R) \cap Attr(R')];$ $z = t[Attr(R) \cap Attr(R')]\}$

Figure 8: Set Operators on the Common Subset of Attributes.

5.2 *Legal Rewriting*: The Semantics for View Synchronization

The capability changes "*delete−attribute R.A*" and "*delete−relation R*" are said to affect views that refer to that particular attribute or relation in their definitions. As we will see in this paper, the algorithms for view synchronization could also change other components of the original view definition besides the deleted components, i.e., other indirectly affected components. For example, replacing an attribute $R.A$ after the capability change "*delete−attribute R.A*" could result in substituting the entire relation R by some other relation S (even though the relation R without the attribute A is still available). Thus, we distinguish between two types of components of an affected view: the ones that are directly related to the deleted component (we called them *affected components*) and those that are not affected by a capability change (*unaffected components*).

Namely, for the capability change "*delete−attribute R.A*" with $R.A$ in the SELECT or WHERE clause of the view V, we have:

- The attribute $R.A$ in the SELECT clause of V (if $R.A \in Attr(V)$) is a ***directly affected*** **component**;
- The relation R in the FROM clause of V is an ***indirectly affected*** **component**;
- All attributes of relation R other than $R.A$ that appear in the SELECT clause of V are ***indirectly affected*** **components**;
- All conditions using the attribute $R.A$ in the WHERE clause of V are ***directly affected*** **components**;
- All conditions using attributes of relation R (other than $R.A$) in the WHERE clause of V are ***indirectly affected*** **components**.

And for the capability change "*delete−relation R*" with R in the FROM clause of the view V, we have:

- The relation R in the FROM clause of V is a ***directly affected*** **component**;
- All attributes of relation R that appear in the SELECT clause of V are ***directly affected*** **components**;
- All conditions using attributes of relation R in the WHERE clause of V are ***directly affected*** **components**.

The semantics of the evolution parameters impose that all indispensable components of a view must be preserved in any synchronized view definition (either exactly as they are in the original view definition or possibly replaced if they are replaceable). For a delete capability change, all *directly affected* components *must* be replaced or dropped in order for the view definition to satisfy the evolution parameters. Thus, the following theorem establishes a necessary (but not sufficient) condition for a view definition to be **evolvable**.

Theorem 1 *If the view V is evolvable under a capability change ch then all the directly affected components are not both indispensable and nonreplaceable, i.e., their evolution parameters are not ($\mathcal{XD} = false, \mathcal{XR} = false$).*

Proof. The proof is immediate and hence omitted here.

Definition 3 *Let ch be a capability change and the view V be evolvable under the capability change ch (Theorem 1). Let MKB and MKB' be the state of the meta knowledge base containing the IS descriptions right before and right after the change ch, respectively*[7]*. 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, i.e., V' has no directly affected components.*
P2. The view V' can be ***evaluated*** *in the new state of the information space (i.e., V' contains only elements*

[7]We assume that the meta knowledge base MKB is evolved into MKB' to reflect the change *ch* using the approach described in our technical report ([NLR97]). This evolution of the MKB is relatively straightforward, and hence a description of it is omitted here.

defined in MKB').

P3. The ***view extent parameter*** $\mathcal{VE} = \delta$ *(*$\delta \in \{\equiv, \supseteq, \subseteq\}$*) of* V *(Fig. 7) is satisfied by the view* V'*. I.e.,*

$$V' \delta_\pi V \tag{11}$$

is true for any state of the underlying information sources (δ_π *is the* π*-set operator corresponding to* δ *as defined in Table 8).*

P4. All ***evolution parameters*** *attached to the view components such as attributes, relations or conditions of the view* V *are satisfied by the view* V'*.*

The property P4 from Definition 3 states that all any legal rewriting V' of the view V must preserve all indispensable components. For example, all indispensable attributes from the SELECT clause (i.e., the ones having $\mathcal{AD} = false$ in V) of the view V must appear in the SELECT clause of the view V' as well. Definition 4 specifies how the evolution parameters are set for the new view definition V'.

Definition 4 ***Evolution Parameter Assignment.*** *When a view component* X' *is used to replace an affected view component* X*, the evolution parameters associated with* X' *are set by the following rules:*

- *Rule 1. If* X' *is used to replace exactly one view component* X *of the original view* V*, the new evolution parameters are set to be the same as those of the original component* X*. If a view component is replaced by more than one new view component, we say that each of the new view components replaces exactly one view component and this Rule 1 applies for each of the new view components.*

- *Rule 2. If a new view component* X' *is used to replace more than one view component of the original view* $X_1(\mathcal{XD} = val_{1,1}, \mathcal{XR} = val_{1,2})$, $\ldots$, $X_k(\mathcal{XD} = val_{k,1}, \mathcal{XR} = val_{k,2})$ *where* $\mathcal{XD}$ *and* $\mathcal{XR}$ *are view dispensable and replaceable evolution parameters, respectively, and* $val_{i,j} \in \{true, false\}$ *their values. We set the evolution parameters of components* X' *as:* $(\mathcal{XD} = val_{1,1}$ ***AND*** $\cdots$ ***AND*** $val_{k,1}$, $\mathcal{XR} = val_{1,2}$ ***AND*** $\cdots$ ***AND*** $val_{k,2})$.

Definition 3 gives the general semantics that the evolution parameters in an E-SQL view definition impose on the view evolution process. Conforming with Definitions 3 and 4, a view V evolved in V' with some unaffected components dropped would be considered legal as long as *all* evolution parameters of V are satisfied and all evolution parameters of V' are appropriately set.

Example 7 *The view given in Equation (12)*[8] *could be legally evolved under the capability change delete−attribute R.A in different ways: (1) dropping the directly affected component R.A and leaving anything else unchanged (Equation (13)); and (2) dropping the directly affected component R.A and, as well, dropping the unaffected component R.C (Equation (14)).*

[8] Default values for evolution parameters are not shown in the definitions.

```
CREATE  VIEW   $V(\mathcal{VE} =\subseteq)$ AS
SELECT         $R.A(\mathcal{AD} = true, \mathcal{AR} = true)$,
               $R.B(\mathcal{AD} = true)$
               $R.C(\mathcal{AD} = true)$                          (12)
FROM           $R(\mathcal{RR} = true)$
WHERE          $R.B > 200$
```

```
CREATE  VIEW   $V_1(\mathcal{VE} =\subseteq)$ AS
SELECT         $R.B(\mathcal{AD} = true)$
               $R.C(\mathcal{AD} = true)$
FROM           $R(\mathcal{RR} = true)$                  (13)
WHERE          $R.B > 200$
```

```
CREATE  VIEW   $V_2(\mathcal{VE} =\subseteq)$ AS
SELECT         $R.B(\mathcal{AD} = true)$
FROM           $R(\mathcal{RR} = true)$                  (14)
WHERE          $R.B > 200$
```

Because view synchronization algorithms primarily are concerned with affected components, we make a distinction between legal view rewritings that are obtained by modifying only affected components (leaving all unaffected components intact), called *base rewritings*, and those that have unaffected components modified as well (e.g., replaced or dropped), called *derived rewritings.* Namely, in base rewritings all "linked" affected components are either all *replaced*, all *dropped* or all *unchanged.* For example, if we find a replacement for an attribute $R.A$, then a rewriting is called a base rewriting if **all** occurrences of $R.A$ in replaceable components are replaced (even though the evolution parameters would have allowed for example to drop a condition referring to $R.A$). If a relation S replaces a relation R (i.e., relation S has replacements for some attributes of R) then a base rewriting must have preserved all attributes that could be replaced from S. Formally, we give the following definitions for base and derived rewritings.

Definition 5 ***Base View Rewriting.*** *Let a view V' be a rewriting of the view V after a "delete−relation" or "delete−attribute" capability change. We say that V' is a* ***base view rewriting*** *if*

B0. V' is a legal rewriting of V (by Definition 3).

B1. All unaffected view components of V are still in V'.

B2. If an (directly or indirectly) affected attribute $R.A$ in V has a replacement attribute $S.B$ in V', then any replaceable component referring to $R.A$ (i.e., an attribute in the SELECT *clause or a condition in the* WHERE *clause of V) must appear in V' with $R.A$ replaced by $S.B$.*

B3. If an indirectly affected relation R is still in the FROM *clause of the view V', then all its indirectly affected attributes that are referred to in V and are not replaced in V' must appear in V' as well.*

Definition 6 ***Derived View Rewriting.*** *Let a view V' be a base rewriting of the view V by Definition 5. Then a view V'' is a* ***derived rewriting*** *of the view V' iff:*

D0. V'' is a legal rewriting of V (by Definition 3).

D1. V'' is obtained from V' by dropping dispensable components (affected or unaffected components of V), i.e., components with dispensable evolution parameters set to true ($\mathcal{XD} = true$).

In the following we give examples of base and derived rewritings for different replacement strategies and capability changes.

Example 8 *Let V be defined by the Equation (15). Given the capability change "delete-attribute R.A", we give some examples of base (Equations (16),(17)) and derived (Equations (18),(19),(20)) rewritings of the view V.*

$$\begin{array}{ll} \textit{CREATE VIEW} & V(\mathcal{VE} = \supseteq) \ \textit{AS} \\ \textit{SELECT} & R.A(\mathcal{AD} = true, \mathcal{AR} = true), \\ & R.B(\mathcal{AD} = false, \mathcal{AR} = false), \\ & R.C(\mathcal{AD} = true, \mathcal{AR} = false) \\ \textit{FROM} & R(\mathcal{RD} = false, \mathcal{RR} = true) \\ \textit{WHERE} & (R.B \ \theta \ value)(\mathcal{CD} = true, \mathcal{CR} = false) \end{array} \tag{15}$$

Legal Base Rewritings of the view V.

The affected component R.A is dropped. Since R.A is no longer in R and it is dispensable in V, dropping it from the view gives a base rewriting (one can verify that all the above conditions B0, B1 and B3 are satisfied). Therefore, we get a legal base rewriting V' (V' is π-equivalent to V, i.e., $V' \equiv_\pi V$):

$$\begin{array}{ll} \textit{CREATE VIEW} & V'(\mathcal{VE} = \supseteq) \ \textit{AS} \\ \textit{SELECT} & R.B(\mathcal{AD} = false, \mathcal{AR} = false), \\ & R.C(\mathcal{AD} = true, \mathcal{AR} = false) \\ \textit{FROM} & R(\mathcal{RD} = false, \mathcal{RR} = true) \\ \textit{WHERE} & (R.B \ \theta \ value)(\mathcal{CD} = true, \mathcal{CR} = false) \end{array} \tag{16}$$

The affected component R.A is replaced with $S.A'$ using a join relation between R and S. If $\pi_{S.A',S.B'}S = \pi_{R.A,R.B}R$, then the view V'' defined in Equation (17) is a legal based rewriting (e.g., $V'' \equiv_\pi V$), i.e., all the properties from Definition 5 are satisfied[9]*.*

$$\begin{array}{ll} \textit{CREATE VIEW} & V''(\mathcal{VE} = \supseteq) \ \textit{AS} \\ \textit{SELECT} & \underline{S.A'(\mathcal{AD} = true, \mathcal{AR} = true)}, \\ & R.B(\mathcal{AD} = false, \mathcal{AR} = false), \\ & R.C(\mathcal{AD} = true, \mathcal{AR} = false) \\ \textit{FROM} & \underline{S(\mathcal{RD} = true, \mathcal{RR} = true)}, R(\mathcal{RD} = false, \mathcal{RR} = true) \\ \textit{WHERE} & \underline{(S.B' = R.B)(\mathcal{CD} = true, \mathcal{CR} = true)} \ \textit{AND} \\ & (R.B \ \theta \ value)(\mathcal{CD} = true, \mathcal{CR} = false) \end{array} \tag{17}$$

[9] View synchronization algorithms for "*delete−attribute*" are given in our previous work in [LNR97a, LNR97b].

Legal Derived Rewritings of the view V.

The base rewriting V' defined in Equation (16) has three derived rewritings: one in which the attribute $R.C$ is dropped from the SELECT *clause (its dispensable evolution parameter is $\mathcal{AD} = true$) (Equation (18)); one in which the condition referring the attribute $R.B$ is dropped (its dispensable evolution parameter is $\mathcal{CD} = true$) (Equation (19)); and one in which both of these two components are dropped (Equation (20)).*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V_1'(\mathcal{VE} = \supseteq) \textsf{ AS} \\ \textsf{SELECT} & R.B(\mathcal{AD} = false, \mathcal{AR} = false), \\ \textsf{FROM} & R(\mathcal{RD} = false, \mathcal{RR} = true) \\ \textsf{WHERE} & (R.B\ \theta\ value)(\mathcal{CD} = true, \mathcal{CR} = false) \end{array} \tag{18}$$

$$\begin{array}{ll} \textsf{CREATE VIEW} & V_2'(\mathcal{VE} = \supseteq) \textsf{ AS} \\ \textsf{SELECT} & R.B(\mathcal{AD} = false, \mathcal{AR} = false), \\ & R.C(\mathcal{AD} = true, \mathcal{AR} = false) \\ \textsf{FROM} & R(\mathcal{RD} = false, \mathcal{RR} = true) \end{array} \tag{19}$$

$$\begin{array}{ll} \textsf{CREATE VIEW} & V'(\mathcal{VE} = \supseteq) \textsf{ AS} \\ \textsf{SELECT} & R.B(\mathcal{AD} = false, \mathcal{AR} = false), \\ \textsf{FROM} & R(\mathcal{RD} = false, \mathcal{RR} = true) \end{array} \tag{20}$$

All three rewritings are derived rewritings, i.e., the properties from Definition 6 are satisfied. Note that they don't satisfy conditions imposed to qualify as base rewritings (by Definition 5). For example, the derived rewriting V_1' defined in Equation (18) has the attribute $R.C$ available in the relation R from the FROM *clause, but it gets dropped from the* SELECT *clause thus violating condition B3 from Definition 5.*

5.3 Correctness Criteria for View Synchronization

We assume SELECT-FROM-WHERE views defined by an E-SQL view definition as shown in Equation (8) with a conjunction of primitive clauses in the WHERE clause. We assume that a view V is defined such that all attributes used in the WHERE clause in an indispensable condition are among the preserved attributes (i.e., the attributes in the SELECT clause) (inheriting the evolution parameters from the condition they come from). In the following we use the term *view element* to refer to a pair composed of a *view component* such as an attribute, relation or condition used in the view definition together with the set of *evolution parameters* attached to it. If an attribute, relation or condition has no evolution parameters associated with it in the view definition, we assume that the view element corresponding to it has the default parameters as defined in Section 4.

Definition 7 *Let ch be a capability change, and let MKB and MKB' be the state of the meta knowledge base containing the IS descriptions right before and right after the change ch, respectively*[10]*. Then a view V' is a* ***minimal legal rewriting*** *of the view V under capability change ch if the following properties hold:*

[10] We assume that the meta knowledge base MKB is evolved into MKB' to reflect the change *ch* using the approach described in our technical report ([NLR97]). This evolution of the MKB is relatively straightforward, and hence a description of it is omitted here.

M1. V' is a base legal rewriting (Definition 3, Definition 5).

M2. The definition of the view V' is **consistent** *with the constraints of the evolved MKB'. That is, any new element in V' (e.g., new condition in the* WHERE *clause) is backed up by the existence of a supporting constraint in MKB'. Put differently, new elements appear in the view V' only if they are required to replace existing elements that, after a delete-relation or delete-attribute change, must be replaced by other elements.*

Thus, we have the following cases:

<u>*Case 1.*</u> *A new element $f(S.A')(\mathcal{AD} = \mathcal{AD}_A, \mathcal{AR} = true)$ could appear in the* SELECT *clause of V' if it replaces exactly one affected element (Section 5.2) $R.A(\mathcal{AD} = \mathcal{AD}_A, \mathcal{AR} = true)$ from the* SELECT *clause of V*[11]*. The attributes $f(S.A')$ and $R.A$ must have the same type and the following condition must hold: $\exists R_1, R_2, \dots, R_n$ such that $R_1 = R$ and $R_n = S$, $\exists \{\mathcal{JC}_{R_i,R_{i+1}} \mid \mathcal{JC}_{R_i,R_{i+1}}$ a join constraint in MKB, $i = \overline{1, n-1}\}$ and it exists a function-of constraint $\mathcal{F}_{R.A,S.A'} = (R.A = f(S.A'))$ in MKB. That is, for any tuple t in the join relation:*

$$R_1 \bowtie_{\mathcal{JC}_{R_1,R_2}} R_2 \bowtie_{\mathcal{JC}_{R_2,R_3}} \cdots \bowtie_{\mathcal{JC}_{R_{n-1},R_n}} R_n \tag{21}$$

we have $t[R.A] = f(t[S.A'])$.

All relations from the expression (21) (except relation R if R is to be dropped) must appear in the FROM *clause of the view V' and all primitive clauses (except the ones involving attributes of R if R is to be dropped) from the set of join constraints used in expression (21) must appear in the* WHERE *clause of the view V'. Moreover, all occurrences of the attribute $R.A$ in the conditions of the* WHERE *clause of V must be replaced by $f(S.A')$ in V'.*

<u>*Case 2.*</u> *A new element $C(\mathcal{CD} = \mathcal{CD}_C, \mathcal{CR} = \mathcal{CR}_C)$*[12] *could appear in the* WHERE *clause of V' if it is a primitive clause in one of the sequences of join constraints used to replace an attribute $R.A$ as in <u>Case 1</u>.*

<u>*Case 3.*</u> *A new element $R(\mathcal{RD} = \mathcal{RD}_R, \mathcal{RR} = \mathcal{RR}_R)$ could appear in the* FROM *clause of V' if it appears in one of the sequences of joins used to replace an attribute $R.A$ as in <u>Case 1</u>.*

M3. The definition of the view V' is **minimal** *with respect to the set of new relations in the* FROM *clause and the set of new conditions in the* WHERE *clause. That is, if we drop a newly added relation from the* FROM *clause or a newly added condition from the* WHERE *clause of V', the modified definition doesn't satisfy properties M1 and M2 any longer. M3 imposes that the new view definition cannot have new extraneous elements added that are not needed in the view, such as an extra attribute in the* SELECT *clause or an extra relation in the* FROM *clause that serves no purpose*[13]*.*

[11] In Section 5.2 we established what are the affected components for a given capability change. Note that the attribute $R.A$ must be replaceable, i.e., its attribute-replaceable parameter $\mathcal{AR}$ must be set to *true*.

[12] The evolution parameters are set as defined in Definition 4.

[13] We impose the minimality property only for avoiding the introduction of new elements into V. Conforming to this definition, two view rewritings V' of V one containing all attributes from V that are marked as dispensable and the other one with all dispensable

Example 9 *Let a view V be defined in Equation (22).*

$$\begin{array}{ll} \textsf{CREATE VIEW} & \textbf{Asia-Customer (AsiaName, AsiaAddress, AsiaPhone)}\ (\mathcal{VE} = \supseteq)\ \textsf{AS} \\ \textsf{SELECT} & \textbf{C.Name, C.Address}(\mathcal{AD} = false, \mathcal{AR} = true), \textbf{C.PhoneNo} \\ \textsf{FROM} & \textbf{Customer C, FlightRes F} \\ \textsf{WHERE} & \textbf{(C.Name = F.PName)}\ \textsf{AND}\ \textbf{(F.Dest = 'Asia')} \end{array} \tag{22}$$

*And let's assume that change ch is "delete attribute **Address** from the relation **Customer**". We have to find a replacement for this attribute that could be obtained from a chain of join constraints defined in MKB'. Let's assume we have the following constraints in MKB:*

*(i) The relation **Person** is defined by **Person(Name, SSN, PAddress)**;*

(ii) $\mathcal{JC}_{\textit{Customer, Person}} = (\textbf{Customer.Name} = \textbf{Person.Name})$ *;*

(iii) $\mathcal{F}_{\textit{Customer.Address, Person.PAddress}} = (\textbf{Customer.Address} = \textbf{Person.PAddress})$ *;*

(iv) $\mathcal{PC}_{\textit{Customer, Person}} = (\pi_{\textit{Name, PAddress}}(\textbf{Person}) \supseteq \pi_{\textit{Name, Address}}(\textbf{Customer}))$.

*It is easily verifiable that the new view definition **Asia-Customer'** defined in Equation (23) is a legal rewriting (new elements are underlined) of Equation (22):*

$$\begin{array}{ll} \textsf{CREATE VIEW} & \textbf{Asia-Customer' (AsiaName, AsiaAddress, AsiaPhone)}\ (\mathcal{VE} = \supseteq)\ \textsf{AS} \\ \textsf{SELECT} & \textbf{C.Name,}\ \underline{\textbf{P.PAddress}\ (\mathcal{AD} = false, \mathcal{AR} = true)}, \textbf{C.PhoneNo} \\ \textsf{FROM} & \textbf{Customer C, FlightRes F,}\ \underline{\textbf{Person P}} \\ \textsf{WHERE} & \textbf{(C.Name = F.PName)}\ \textsf{AND}\ \textbf{(F.Dest = 'Asia')} \\ & \underline{\textsf{AND}\ \textbf{(P.Name = C.Name)}} \end{array} \tag{23}$$

This legal rewriting uses the join constraint $\mathcal{JC}_{\textit{Customer, Person}}$ *(defined in (ii)) to obtain the address from the relation **Person** by using the join relation* $\left(\textbf{Customer} \bowtie_{\mathcal{JC}_{\textit{Customer, Person}}} \textbf{Person}\right)$ *in the WHERE clause, and the function-of constraint defined in (iii). Then the evolved view definition given in Equation (23) has all the properties M1 to M3 from Definition 3. Thus it is a legal rewriting. Note, that we can prove that for the evolved view defined by Equation (23), 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**, we have **Asia-Customer'** $\supseteq$ **Asia-Customer**.*

Example 10 *Queries that violate at least one of the properties from Definition 3 are given below:*

*(A) The evolution parameters from the initial query V are not satisfied by attribute **Address** being dropped. Thus property M1 is violated (i.e., the view obtained is not a legal rewriting (Definition 3)):*

$$\begin{array}{ll} \textsf{CREATE VIEW} & \textbf{Asia-Customer}(\mathcal{VE} = \supseteq)\ \textsf{AS} \\ \textsf{SELECT} & \textbf{C.Name, C.PhoneNo} \\ \textsf{FROM} & \textbf{Customer C, FlightRes F} \\ \textsf{WHERE} & \textbf{(C.Name = F.PName)}\ \textsf{AND}\ \textbf{(F.Dest = 'Asia')} \end{array} \tag{24}$$

attributes dropped are both legal. On this venue, we incorporated into *EVE* a cost model to differentiate between the "quality" of these two alternative yet legal rewritings [LKNR98].

*(B) The expression used to obtain a replacement for the attribute **Customer.Address** is not merged into the new definition by failing to add the join condition in the WHERE clause, thus violating M2:*

$$\begin{array}{ll} \textsf{CREATE VIEW} & \textbf{\textit{Asia-Customer}}(\mathcal{VE} = \supseteq) \ \textsf{AS} \\ \textsf{SELECT} & \textbf{\textit{C.Name}}, \ \underline{\textbf{\textit{P.PAddress}} \ (\mathcal{AD} = false, \mathcal{AR} = true)}, \ \textbf{\textit{C.PhoneNo}} \\ \textsf{FROM} & \textbf{\textit{Customer C, FlightRes F}}, \ \underline{\textbf{\textit{Person P}}} \\ \textsf{WHERE} & (\textbf{\textit{C.Name}} = \textbf{\textit{F.PName}}) \ \textsf{AND} \ (\textbf{\textit{F.Dest}} = \textbf{\textit{'Asia'}}) \end{array} \qquad (25)$$

*This is an example when failing to add the join constraint between the relations **Customer** and **Person** in the WHERE clause results in the new view including meaningless tuples (coming from the Cartesian product between the new relation **Person** and the rest of the view).*

6 Finding Strongest E-SQL View Definition: Resolution of Dependencies between E-SQL Evolution Parameter Settings

Our view definition language E-SQL allows any combination of the evolution parameters to be set for the view components. Because of the close relationship between components, e.g., an attribute in the SELECT clause is coming from a relation in the FROM clause, the synchronization cannot take advantage of the apparent full flexibility of all possible evolution parameter combinations. For example, consider a view V having an attribute $R.A(\mathcal{AD} = false, \mathcal{AR} = false)$ in the SELECT clause and the relation $R(\mathcal{RD} = true, \mathcal{RR} = true)$ in the FROM clause. Even though the parameters associated with the relation apparently permit it, the view could never be synchronized by dropping or replacing the relation R. Both of these actions would not satisfy the evolution parameters for the attribute $R.A$ (even though they are allowed by the evolution parameters of the relation R). Thus, we can safely assume that the view definition V' having the component $R(\mathcal{RD} = false, \mathcal{RR} = false)$ in the FROM clause is "synch-equivalent" to the original view. Synch-equivalence concept is defined to mean that any synchronization algorithm that fails for V' would have failed for V as well; and any algorithm that succeeds for V' would have succeeded for V as well.

In the following we give some criteria of how to find a "synch-equivalent" definition of a view V that expresses the real evolution flexibility of the initial view definition. Finding the synch-equivalent definitions is essential for helping the view definer to understand the semantics associated with a view definition[14] (it makes the implicit evolution semantics explicit).

Definition 1 *We say that two views V_1 and V_2 are **synch-equivalent** if they differ only in the evolution parameters of their components and for any state of the MKB and a change ch, any synchronization algorithm finds the same set of base rewritings (Definition 5) for the two views.*

[14]The process of defining a view should be an interactive process: the user defines an E-SQL view, EVE finds the synch-equivalent definition, the user decides if this is what he meant or not with the process continuing until an agreement is reached.

Definition 2 *Over the set of all evolution parameters of a component C of a view V, $\{(\mathcal{XD} = false, \mathcal{XR} = false), (\mathcal{XD} = true, \mathcal{XR} = false), (\mathcal{XD} = false, \mathcal{XR} = true), (\mathcal{XD} = true, \mathcal{XR} = true)\}$ (with $\mathcal{X}$ one of $\mathcal{A}$, $\mathcal{R}$ or $\mathcal{C}$ for attribute, relation or condition component, respectively) we introduce a partial order* ***"stronger than"****, denoted by $\leq$. We say that one value is "stronger than" another value, i.e., $(\mathcal{XD} = d_1, \mathcal{XR} = r_1) \leq (\mathcal{XD} = d_2, \mathcal{XR} = r_2)$ if and only if for a view definition V_1 with the evolution parameters for the component C $(\mathcal{XD} = d_1, \mathcal{XR} = r_1)$ and a view definition V_2 identical to V_1 but having the evolution parameters for the component C $(\mathcal{XD} = d_2, \mathcal{XR} = r_2)$, any base rewriting (Definition 5) found by a synchronization algorithm for the view V_1 is also found by the same algorithm for the view V_2, for any state of the MKB and capability change ch.*

Note that the relation "stronger than" is indeed a partial order: it is reflexive (any pair of evolution parameters is "stronger than" itself by Definition 2), transitive and antisymmetric. From Definition 2, it is easy to see that "stronger than" inequalities from Equations (26) and (27) are true and they define a partial order over the set of all evolution parameter values for a component C.

$$(\mathcal{XD} = false, \mathcal{XR} = false) \leq (\mathcal{XD} = true, \mathcal{XR} = false) \leq (\mathcal{XD} = true, \mathcal{XR} = true) \tag{26}$$

$$(\mathcal{XD} = false, \mathcal{XR} = false) \leq (\mathcal{XD} = false, \mathcal{XR} = true) \leq (\mathcal{XD} = true, \mathcal{XR} = true). \tag{27}$$

In Equation (26), the evolution parameters $(\mathcal{XD} = false, \mathcal{XR} = false)$ is stronger than the evolution parameter $(\mathcal{XD} = true, \mathcal{XR} = false)$ given that all the rewritings of a view V_1 with a component $C(\mathcal{XD} = false, \mathcal{XR} = false)$ must contain this component unchanged, while the rewritings for a view V_2 differing from V_1 only in the component $C(\mathcal{XD} = true, \mathcal{XR} = false)$ could preserve C or drop it. Thus, any view synchronization algorithm applied to the view V_2 finds all the rewritings corresponding to the view V_1 plus possibly some new rewritings when C is dropped.

Definition 3 *We say that the view V_1 is* ***"stronger than"*** *the view V_2, denoted by $V_1 \leq V_2$ if (1) they differ only in the evolution parameters of their components and (2) for any component $C(\mathcal{XD} = d_1, \mathcal{XR} = r_1)$ of the view V_1 and its corresponding component $C(\mathcal{XD} = d_2, \mathcal{XR} = r_2)$ in V_2, we have $(\mathcal{XD} = d_1, \mathcal{XR} = r_1) \leq (\mathcal{XD} = d_2, \mathcal{XR} = r_2)$.*

Our goal is to find the "strongest" view definition that is synch-equivalent to the initial view. This then represents exactly the evolution semantics any view synchronization algorithm attaches to the original view definition (even if not stated explicitly). In other words, an E-SQL view definition is at all times only evolved according to the evolution parameters of its strongest view definition.

Definition 4 *We say that V' is the* ***strongest synch-equivalent*** *definition of the view V iff (1) V' is synch-equivalent to V, (2) $V' \leq V$ and (3) there exists no other view V'' such that $V'' \leq V'$ and V'' is synch-equivalent to V.*

In the Tables 9, 10, 11 and 12[15] we give the transformation rules for changing a view definition into a synch-equivalent one. Any transformation applied to a view V gives a "stronger" view definition and at the same time a synch-equivalent one.

$R.A(\mathcal{AD},\mathcal{AR})$ \ $(\mathcal{C} = R.A\ \theta\ const)(\mathcal{CD},\mathcal{CR})$	$R.A(false,false)$	$R.A(true,false)$	$R.A(false,true)$	$R.A(true,true)$
$\mathcal{C}(false,false)$	-	$R.A(false,false)$	$R.A(false,false)$	$R.A(false,false)$
$\mathcal{C}(true,false)$	$\mathcal{C}(false,false)$	-	-	-
$\mathcal{C}(false,true)$	$\mathcal{C}(false,false)$	-	-	$R.A(false,true)$
$\mathcal{C}(true,true)$	$\mathcal{C}(false,false)$	-	$\mathcal{C}(false,true)$	-

Figure 9: Transformation rules for a condition ($\mathcal{C} = R.A\ \theta\ const$) in WHERE clause and its attribute $R.A$ in SELECT clause.

$R.A(\mathcal{AD},\mathcal{AR})$ \ $R(\mathcal{RD},\mathcal{RR})$	$R.A(false,false)$	$R.A(true,false)$	$R.A(false,true)$	$R.A(true,true)$
$R(false,false)$	-	-	-	-
$R(true,false)$	$R(false,false)$	-	-	-
$R(false,true)$	$R(false,false)$	-	-	-
$R(true,true)$	$R(false,false)$	-	-	-

Figure 10: Transformation rules for a relation R in FROM clause and an attribute $R.A$ in SELECT clause.

$\mathcal{C}(\mathcal{CD},\mathcal{CR})$ \ $R(\mathcal{RD},\mathcal{RR})$	$\mathcal{C}(false,false)$	$\mathcal{C}(true,false)$	$\mathcal{C}(false,true)$	$\mathcal{C}(true,true)$
$R(false,false)$	-	-	-	-
$R(true,false)$	$R(false,false)$	-	-	-
$R(false,true)$	$R(false,false)$	-	-	-
$R(true,true)$	$R(false,false)$	-	-	-

Figure 11: Transformation rules for a relation R in FROM clause and a condition $\mathcal{C}$ in WHERE clause using an attribute $R.A$.

Example 11 *To illustrate some of the transformation rules, let's consider the view V defined by Equation 28.*

$$\begin{array}{ll} \textsf{CREATE VIEW} & V\ \textsf{AS} \\ \textsf{SELECT} & X.A(\mathcal{AD} = true, \mathcal{AR} = true), Y.A(\mathcal{AD} = true, \mathcal{AR} = true), \ldots \\ \textsf{FROM} & X, Y, \ldots \\ \textsf{WHERE} & (X.A > Y.A(\mathcal{CD} = false, \mathcal{CR} = true)), \ldots \end{array} \tag{28}$$

This case corresponds to column three, row ten in the Table 12. If the attribute $X.A$ is deleted, the affected

[15]Note that in all the definitions related to the strongest synch-equivalent definition associated to a view, we are talking about base rewriting concept defined in Definition 5.

$\mathcal{C}(\mathcal{CD},\mathcal{CR})$ \ $(X.A(\mathcal{AD},\mathcal{AR}),Y.A(\mathcal{AD},\mathcal{AR})$ $(X,Y)\in\{(R,S),(S,R)\}$	$\mathcal{C}(false,false)$	$\mathcal{C}(true,false)$	$\mathcal{C}(false,true)$	$\mathcal{C}(true,true)$
$X.A(false,false)$ $Y.A(false,false)$	-	$\mathcal{C}(false,false)$	$\mathcal{C}(false,false)$	$\mathcal{C}(false,false)$
$X.A(false,false)$ $Y.A(true,false)$	$Y.A(false,false)$	-	-	-
$X.A(false,false)$ $Y.A(false,true)$	$Y.A(false,false)$	-	-	$\mathcal{C}(false,true)$
$X.A(false,false)$ $Y.A(true,true)$	$Y.A(false,false)$	-	$Y.A(false,true)$	-
$X.A(true,false)$ $Y.A(true,false)$	$X.A(Y.A)(false,false)$	-	-	-
$X.A(true,false)$ $Y.A(false,true)$	$X.A(Y.A)(false,false)$	-	-	-
$X.A(true,false)$ $Y.A(true,true)$	$X.A(Y.A)(false,false)$	-	$Y.A(false,true)$	-
$X.A(false,true)$ $Y.A(false,true)$	$X.A(Y.A)(false,false)$	-	-	$\mathcal{C}(false,true)$
$X.A(false,true)$ $Y.A(true,true)$	$X.A(Y.A)(false,false)$	-	$Y.A(false,true)$	-
$X.A(true,true)$ $Y.A(true,true)$	$X.A(Y.A)(false,false)$	-	$X.A(false,true)$ $Y.A(false,true)$	-

Figure 12: Transformation rules for the attributes $R.A$ and $S.B$ in SELECT clause and a condition $\mathcal{C}$ in WHERE clause using both attributes.

condition in the WHERE clause $C = (X.A > Y.A(\mathcal{AD} = false, \mathcal{AR} = true))$ must be replaced given its evolution parameters. Thus a replacement for the deleted attribute $X.A$ must be found and used in any legal base rewriting V'. Then by Definition 5, all occurrences of the attribute $X.A$ must be substituted by this replacement in V', including the component $X.A(\mathcal{AD} = true, \mathcal{AR} = true)$ in the SELECT clause. Thus, regardless how the replacements are found, any legal base rewriting must have the component $X.A(\mathcal{AD} = true, \mathcal{AR} = true)$ or a replacement of it in the SELECT clause. This behavior never takes advantage of the evolution parameter "$AD = true$", and it is consistent to having "$AD = false$" instead. This explains why the transformation rule from column three, row ten in the Table 12 changes the evolution parameters for the component $X.A(\mathcal{AD} = true, \mathcal{AR} = true)$ to $X.A(\underline{\mathcal{AD} = false}, \mathcal{AR} = true)$.

To find the strongest synch-equivalent view for a given view definition, we apply the transformation rules for all pairs of related components, until no more modifications could be done.

Theorem 2 *By applying the above transformation rules to an E-SQL view V until no more rules can be applied, the strongest synch-equivalent view definition is obtained.*

Theorem 3 *The strongest synch-equivalent view definition V' of a view V can be obtained by a sequence of transformations defined in the Tables 9, 10, 11 and 12.*

Due to space limitation, we omit here the proofs for the Theorems 2 and 3. We just make the observation that by applying any transformation rule from Tables 9, 10, 11 and 12 to a view definition V, a synch-equivalent view definition V_{st} is obtained. This process of applying the transformation rules ends after a finite number of transformations (i.e., no more transformations could be applied after a finite number of transformations) and the result doesn't depend on the order of the transformations.

Definition 5 *We call two evolution parameter values $\mathcal{P}_1$ and $\mathcal{P}_2$ of a component C incompatible if for any view V having the component C, there don't exist two synch-equivalent definitions V_1 and V_2 stronger than V with the evolution parameters of the component C set to $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively.*

For example, evolution parameters $(\mathcal{XD} = false, \mathcal{XR} = true)$ and $(\mathcal{XD} = true, \mathcal{XR} = false)$ are incompatible: there are no transformation rules that could be applied to an original view definition for a component C such that the two evolution parameters are obtained in stronger synch-equivalent definitions. From our transformation rules, we can conclude that the only incompatible evolution parameters are $(\mathcal{XD} = false, \mathcal{XR} = true)$ and $(\mathcal{XD} = true, \mathcal{XR} = false)$; and $(\mathcal{XD} = true, \mathcal{XR} = true)$ and $(\mathcal{XD} = true, \mathcal{XR} = false)$.

Given the above transformation rules and Theorems 2 and 3, we can now prove that there is only one minimal synch-equivalent definition for any E-SQL view.

Theorem 4 *The strongest synch-equivalent definition is unique for an E-SQL view V.*

Proof. Let's assume that there are two view definitions V_{st1} and V_{st2} qualified to be the strongest synch-equivalent for the same view V such that neither $V_{st1} \leq V_{st2}$ nor $V_{st2} \leq V_{st1}$ hold. Two strongest view definitions of the same view V cannot be compared because a component has in the two views incompatible parameters; or because not all parameters in one of the view are "stronger than" the corresponding parameters in the other view (as required by Definition 3.) Formally, we can have two cases: (I) there must exist a component C in V such that the evolution parameters $\mathcal{P}_{st1}$ of C in V_{st1} is incompatible with the evolution parameters $\mathcal{P}_2$ of C in V_{st2}; or (II) there exist two components C_1 and C_2 in V with the evolution parameters are set to $P_{1,1}$ and $P_{2,1}$ for C_1 in V_{st1} and V_{st2}, respectively; and the evolution parameters are set to $P_{1,2}$ and $P_{2,2}$ for C_2 in V_{st1} and V_{st2}, respectively.

In case (I), the only values of evolution parameters that are incompatible are $(\mathcal{XD} = false, \mathcal{XR} = true)$ and $(\mathcal{XD} = true, \mathcal{XR} = false)$; and $(\mathcal{XD} = true, \mathcal{XR} = true)$ and $(\mathcal{XD} = true, \mathcal{XR} = false)$. Let's assume that $\mathcal{P}_{st1} = (\mathcal{XD} = false, \mathcal{XR} = true)$ and $\mathcal{P}_2 = (\mathcal{XD} = true, \mathcal{XR} = false)$. The only way the component C of V could be transformed to have $\mathcal{P}_1$ and $\mathcal{P}_2$ in minimal synch-equivalent definitions, is for it to have the evolution parameter $(\mathcal{XD} = true, \mathcal{XR} = true)$ in the original view. From Theorem 3 we have that the views V_{st1} and V_{st2} are obtained by applying only the transformations rules from Tables 9, 10, 11 and 12 which don't contain any transformation that would change an evolution parameter in $\mathcal{P}_2 = (\mathcal{XD} = true, \mathcal{XR} = false)$. Thus, this case cannot occur while applying the transformation rules.

Case (II) could be proven never to happen using the transformation rules given in Tables 9, 10, 11 and 12. The discussion is omitted here for space reasons.

Q.E.D.

7 View Synchronization: The CVS Algorithm

7.1 Three Steps of the View Synchronization Process

We propose a three-step strategy for the view synchronization process:

Step 1. Given a capability change *ch*, our system will first evolve the meta knowledge base (MKB) itself by detecting and modifying the affected MISD descriptions found in the MKB. Figure 13 summarizes the types of capability changes *EVE* can handle as well as if they must be evolved (denoted by $\star$) under a capability change.

capability \ constraint change	type integrity constraint	order integrity constraint	attribute function-of constraint	join constraint	partial\complete info constraint
delete-attribute	$\star$	$\star$	$\star$	$\star$	$\star$
add-attribute	-	-	-	-	-
change-attribute-name	-	-	-	-	-
delete-relation	$\star$	$\star$	$\star$	$\star$	$\star$
add-relation	-	-	-	-	-
change-relation-name	-	-	-	-	-

Figure 13: MKB Evolution under Capability Changes.

Step 2. Given a capability change *ch* of an underlying IS, *EVE* detects if the definition of the view V (in VKB) is affected by the change *ch* when the state of the knowledge base is MKB. Namely, *EVE* will return one of of the following three results:

- *The view definition cannot be synchronized* - the view V is affected by the change *ch* but cannot be evolved under any circumstances. This would occur if there exists at least one view component (i.e., attribute, relation or condition) affected by the change that is indispensable and nonreplaceable (i.e, the attached dispensable and replaceable parameters are both *false*). This necessary condition is given by the Theorem 1, i.e., if the view doesn't satisfy it then the view cannot be evolved.

- *The view definition can be synchronized* - the view V is affected by the change *ch* but it may be possible to find a legal rewriting. Note that even if the view is *evolvable* (Theorem 1) then the *EVE* system may still fail to evolve the view if the affected components cannot be found in some other ISs.

- *The view definition is not affected by the change ch* - In this case, Step 3 below is skipped, and the view synchronization process terminates.

Step 3. Lastly, for affected yet potentially evolvable views (those corresponding to Step2, second case above) we apply some view synchronization algorithm to find legal rewritings for view definitions guided by constraints imposed by the view evolution preferences as well as by the knowledge captured in the evolved MKB'.

Due to limited space, the rest of the paper concentrates on the most difficult step of the view synchronization process, namely, the third one. We present a synchronization process referred to as Complex View Synchronization (or short, CVS) as solution approach for this third step.

7.2 Basics of the CVS Algorithm

We now describe our solution for the third step of the view synchronization process given in Section 7.1, namely, the actual rewriting of an affected view definition. As already indicated in Figure 13 of Section 7.1, four of the six capability change operations we consider can be handled in a straightforward manner. Namely, the *add-relation* and *add-attribute* capability changes do not cause any changes to existing (and hence valid) views, and we assume that our current system will not further optimize existing views based on this new knowledge. The two rename capability change operators, *rename-relation* and *rename-attribute*, are caught by the name mapping service in the MKB and hence also do not require any synchronization of 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 the *delete-relation* algorithm given below, and is omitted in this paper due to space limitations. Our **Complex View Synchronization (CVS)** algorithm could be easily adapted for the *delete-attribute* operator.

We make the following assumptions in order to keep the discussion simple, however, our described solution can be easily extended to handle less restrictive cases. One, given a "*delete-relation* R from IS" request, we assume that an affected view query V uses R only once in the FROM clause. Our **CVS** algorithm could be easily adapted for a more general case when the relation R appears more than once in the FROM clause. Two, we assume that any join constraint in MKB is augmented with the order constraints defined for the relations involved in that join constraint. The computation of this integrated representation of constraints from the MKB is straightforward, and is omitted here for space reasons [NLR97]. We now start by giving some definitions of concepts needed to characterize valid replacements of view components.

Example 12 *To illustrate the steps of our approach for rewriting, we will use the view defined by query 29 below and the change operator "delete relation* ***Customer****" both over the schema of our running example (Example 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.*

$$
\begin{array}{ll}
\textsf{CREATE VIEW} & \textbf{\textit{Customer-Passengers-Asia}}\ (\mathcal{VE}_V)\ \textsf{AS} \\
\textsf{SELECT} & \textbf{\textit{C.Name}}\ (\mathcal{AD} = false, \mathcal{AR} = true),\ \textbf{\textit{C.Age}}\ (\mathcal{AD} = true, \mathcal{AR} = true), \\
 & \textbf{\textit{P.Participant}}\ (\mathcal{AD} = true, \mathcal{AR} = true),\ \textbf{\textit{P.TourID}}\ (\mathcal{AD} = true, \mathcal{AR} = true) \\
\textsf{FROM} & \textbf{\textit{Customer C}}\ (\mathcal{RD} = true, \mathcal{RR} = true),\ \textbf{\textit{FlightRes F}}\ (\mathcal{RD} = true, \mathcal{RR} = true), \\
 & \textbf{\textit{Participant P}}\ (\mathcal{RD} = true, \mathcal{RR} = true) \\
\textsf{WHERE} & (\textbf{\textit{C.Name}} = \textbf{\textit{F.PName}})\ (\mathcal{CD} = false, \mathcal{CR} = true)\ \textsc{and}\ (\textbf{\textit{F.Dest}} = \textbf{\textit{'Asia'}}) \\
 & \textsc{and}(\textbf{\textit{P.StartDate}} = \textbf{\textit{F.Date}})\ \textsc{and}\ (\textbf{\textit{P.Location}} = \textbf{\textit{'Asia'}})
\end{array}
\tag{29}
$$

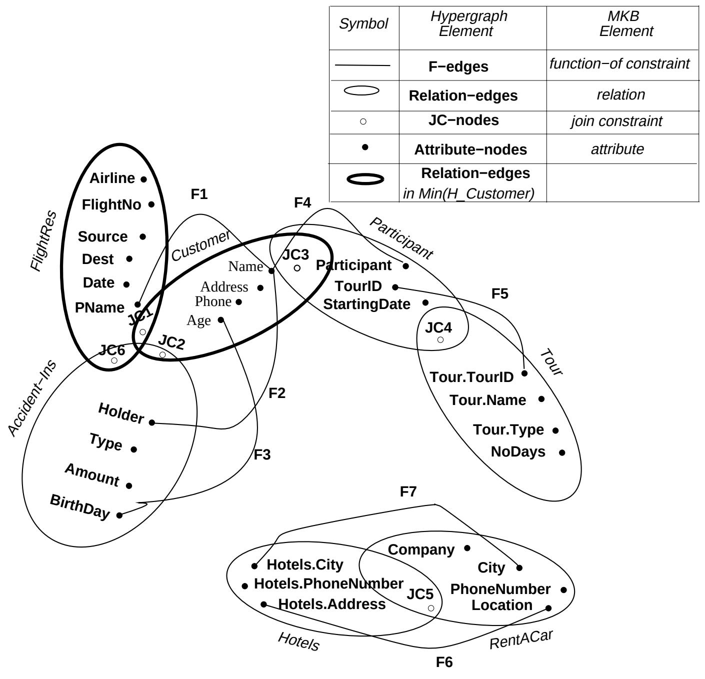

Figure 14: The Hypergraph $\mathcal{H}$(MKB) for Example 1 and View (29).

7.3 Mapping the MKB to a Hypergraph Representation

Generally, a database schema can be represented as a hypergraph whose nodes are the attributes and whose hyperedges are the relations [Ull89, GL94]. Inspired by this representation, we represent the set of attributes and the set of relations described in MKB by a hypergraph that is extended with extra nodes corresponding to the join constraints in the MKB and extra edges corresponding to the function-of constraints. Below we give the formal definition of a hypergraph representation of a MKB and in Figure 14 we give an example of its graphical representation.

Definition 8 ***Meta Knowledge Base Hypergraph*** $\mathcal{H}(MKB)$. *Given a meta knowledge base MKB, we define the hypergraph associated with the MKB as a four-tuple*
$\mathcal{H}(MKB) = \{\mathcal{A}(MKB), \mathcal{J}(MKB), \mathcal{S}(MKB), \mathcal{F}(MKB)\}$, *where:*

- $(\mathcal{A}(MKB) \cup \mathcal{J}(MKB)$ *) is the set of nodes of the hypergraph where:*
  (1) $\mathcal{A}(MKB)$ *are the attribute-nodes corresponding to the attributes defined in MKB; and*
  (2) $\mathcal{J}(MKB)$ *are the* $\mathcal{JC}$*-nodes corresponding to the join constraints defined in MKB.*

- $(\mathcal{S}(MKB) \cup \mathcal{F}(MKB))$ *is the set of edges for the hypergraph where:*
  (1) $\mathcal{S}(MKB)$ *are the relation-edges corresponding to the set of relations defined in MKB. For R a relation in MKB, the relation-edge R contains the set of attribute-nodes and* $\mathcal{JC}$*-nodes corresponding to the attributes of relation R and to the join constraints defined for relation R, respectively.*

(2) $\mathcal{F}(MKB)$ are the $\mathcal{F}$-edges corresponding to the set of function-of constraints defined in MKB. For $\mathcal{F}$ a function-of constraint in MKB, the $\mathcal{F}$-edge representing $\mathcal{F}$ contains two attribute-nodes corresponding to the attributes used in $\mathcal{F}$.

Example 13 *Figure 14 depicts the hypergraph for the MKB of our travel agency example (Example 2) where:*
$\mathcal{A}(MKB) = \{$ ***Name, Address, Phone, Age, Tour.TourID, TourName, Tour.Type, NoDays, Participant, TourID, StartDate, Participant.Location, PName, Airline, FlightNo, Source, Dest, Data, Holder, Type, Amount, Birthday, Hotels.City, Hotels.Address, Hotels.PhoneNumber, Company, City, PhoneNumber, RentACar.Location*** $\}$ *(see Fig. 2);*
$\mathcal{J}(MKB) = \{$ ***JC1, JC2, JC3, JC4, JC5, JC6*** $\}$ *(see Fig. 4);*
$\mathcal{S}(MKB) = \{$ ***Customer, Tour, Participant, FlightRes, Accident-Ins, Hotels, RentACar*** $\}$ *(see Fig. 2);*
$\mathcal{F}(MKB) = \{$ ***F1, F2, F3, F4, F5, F6, F7*** $\}$ *(see Fig. 5);*

The notations used for the graphical representation in Figure 14 and their connections with the elements of $\mathcal{H}(MKB)$ are given in the table below.

$\mathcal{H}(MKB)$ element	MKB Concept	graphical symbol
attribute-node A	*attribute A*	*filled circle labeled with the attribute name **A***
$\mathcal{JC}$-node $\mathcal{JC}_i$	*join constraint $\mathcal{JC}_i$*	*empty circle labeled with the join constraint name **JCi***
relation-edge R	*relation R*	*a circle with attribute-nodes and $\mathcal{JC}$-nodes inside; labeled with relation name **R***
$\mathcal{F}$-edge $\mathcal{F}_j$	*function-of constraint $\mathcal{F}_j$*	*a line connecting two attribute-nodes; labeled with function-of constraint name **Fj***

Note that two relation-edges can share a $\mathcal{JC}$-node if and only if it corresponds to a join constraint defined in MKB for the two relations.

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.

Definition 9 ***Connected Sub-Hypergraph $\mathcal{H}_R$(MKB).*** *For a relation R, we define the sub-hypergraph $\mathcal{H}_R$(MKB) of the hypergraph $\mathcal{H}$(MKB) as being the connected sub-hypergraph that contains the relation-edge R plus all other hypernodes and hyperedges connected to it. Namely, $\mathcal{H}_R$(MKB) is defined by:*

$$\mathcal{H}_R(MKB) = \{\mathcal{A}_R(MKB), \mathcal{J}_R(MKB), \mathcal{S}_R(MKB), \mathcal{F}_R(MKB)\} \tag{30}$$

where (1) $\mathcal{A}_R(MKB)$, $\mathcal{J}_R(MKB)$, $\mathcal{S}_R(MKB)$ and $\mathcal{F}_R(MKB)$ are subsets of $\mathcal{A}(MKB)$, $\mathcal{J}(MKB)$, $\mathcal{S}(MKB)$ and $\mathcal{F}(MKB)$, respectively; (2) R is a relation-edge in $\mathcal{S}_R(MKB)$; and (3) $\mathcal{H}_R(MKB)$ is a connected sub-hypergraph of $\mathcal{H}(MKB)$.

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 14 *Figure 14 depicts two connected sub-hypergraphs for the hypergraph $\mathcal{H}$(MKB) for Example 1. For $R =$ **Customer**, the connected sub-hypergraph $\mathcal{H}_{\textbf{Customer}}(MKB)$ is the connected sub-hypergraph drawn on the top left of the Figure 14.*

7.4 R-mapping from a View into MKB Hypergraph

Given a view definition referring to a relation R and an MKB, we want to determine which parts of the affected view definition need to be replaced when R is dropped. To find possible replacements, we must 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. Definition 10 formally defines this relationship between a view definition and the (default) join constraints in MKB related to the relation R that could potentially be exploited to locate replacements for R.

Definition 10 ***R-mapping of a view query V into sub-hypergraph $\mathcal{H}_R(MKB)$.*** *For a view query V defined as in Eq. (8) and a relation R from the FROM clause of the view query V, 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 query 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:

$$Max(V_R) \ = \ R_{v_1} \bowtie_{\mathcal{C}_{R_{v_1},R_{v_2}}} \cdots \bowtie_{\mathcal{C}_{R_{v_{l-1}},R_{v_l}}} R_{v_l} \tag{31}$$

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} (both local and join conditions).
(II) The expression $Min(\mathcal{H}_R)$ is of the from:

$$Min(\mathcal{H}_R) = R_{v_1} \bowtie_{\mathcal{JC}_{R_{v_1},R_{v_2}}} \cdots\cdots \bowtie_{\mathcal{JC}_{R_{v_{l-1}},R_{v_l}}} R_{v_l} \tag{32}$$

where relations $\{R_{v_1}, \ldots, R_{v_l}\} \subseteq \mathcal{S}_R(MKB)$, and $\{\mathcal{JC}_{R_{v_1},R_{v_2}}, \ldots, \mathcal{JC}_{R_{v_{l-1}},R_{v_l}}\} \subseteq \mathcal{J}_R(MKB)$.

(III) The relation defined by $Max(V_R)$ is contained in the relation defined by $Min(\mathcal{H}_R)$:

$$\underbrace{\left(R_{v_1} \bowtie_{\mathcal{C}_{R_{v_1},R_{v_2}}} \cdots \bowtie \cdots \bowtie_{\mathcal{C}_{R_{v_{l-1}},R_{v_l}}} R_{v_l}\right)}_{Max(V_R)} \subseteq \underbrace{\left(R_{v_1} \bowtie_{\mathcal{JC}_{R_{v_1},R_{v_2}}} \cdots \bowtie \cdots \bowtie_{\mathcal{JC}_{R_{v_{l-1}},R_{v_l}}} R_{v_l}\right)}_{Min(\mathcal{H}_R)} \tag{33}$$

(IV) The expression $Max(V_R)$ 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 (II) and (III) are satisfied.*

(V) The expression $Min(\mathcal{H}_R)$ 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.

Note that Equation 33 implies that there exists a conjunction of primitive clauses $\mathcal{C}_{Max/Min}$ such that

$$Max(V_R) = \sigma_{\mathcal{C}_{Max/Min}}(Min(\mathcal{H}_R)) \tag{34}$$

The goal of Definition 10 is to find the expressions $Max(V_R)$ and $Min(\mathcal{H}_R)$ such that the view V could be written as:

$$V = \pi_{\bar{B}_V}\left(\underbrace{(\sigma_{\mathcal{C}_{Max/Min}}(Min(\mathcal{H}_R)))}_{Max(V_R)} \bowtie_{\mathcal{C}_{Rest}} Rest\right) \tag{35}$$

where $\bar{B}_V$ is the view interface (see Eq. (8)), $\mathcal{C}_{Rest}$ 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 15 *In Figure 14, the minimal subexpression $Min(\mathcal{H}_{\textbf{Customer}})$ of $\mathcal{H}_{\textbf{Customer}}(MKB)$ is marked by bold lines and corresponds to:*

$$Min(\mathcal{H}_{\textbf{Customer}}) = \textbf{\textit{FlightRes}} \bowtie_{\underbrace{(\textbf{\textit{FlightRes.PName=Customer.Name}})}_{\textbf{\textit{JC1}}}} \textbf{\textit{Customer}} \tag{36}$$

*The maximal subexpression $Max(\textbf{\textit{Customer-Passenger-Asia}}_{\textbf{Customer}})$ of the view defined by Eq. (29) and the relation **Customer** is:*

$$\begin{aligned} Max(\textbf{\textit{Customer-Passenger-Asia}}_{\textbf{Customer}}) &= \textbf{\textit{FlightRes}} \bowtie_{\underbrace{\left(\begin{array}{c}\textbf{\textit{(FlightRes.PName = Customer.Name)}} \\ \textbf{\textit{AND (FlightRes.Dest = 'Asia')}}\end{array}\right)}_{\mathcal{C}_{\textbf{\textit{FlightRes, Customer}}}}} \textbf{\textit{Customer}} \\ &= \underbrace{\sigma_{\textbf{\textit{FlightRes.Dest = 'Asia'}}}}_{\mathcal{C}_{Max/Min}}(Min(\mathcal{H}_R)) \end{aligned} \tag{37}$$

The relation defined by Eq.(37) is contained in the relation defined by Eq. (36) and they are maximal and minimal, respectively, with this property. I.e., the relation defined by Eq. (37) is the maximal subexpression in the view defined in Eq. (29) having the properties (I) to (III): if we add a new relation from the query (29) to it, we cannot find any longer a subexpression of $\mathcal{H}(MKB)$ so that these properties hold. The same we can say for the subexpression defined by Eq. (36) to be minimal with the properties (I) to (III).

Observation. To find two expressions $Max(V_R)$ and $Min(\mathcal{H}_R)$ with the property (III), it is sufficient ([Ull89]) to have that each join constraint $\mathcal{JC}_{S,S'}$ of expression $Min(\mathcal{H}_R)$ (Eq. 32) is implied by the corresponding join condition $\mathcal{C}_{S,S'}$ of expression $Max(V_R)$ (Eq. 31), where $S, S' \in \{R_{v_1}, \ldots, R_{v_l}\}$.

Computing R-mapping(V, $\mathcal{H}_R(MKB)$) = $(Max(V_R), Min(\mathcal{H}_R))$.
To find $Max(V_R)$ and $Min(\mathcal{H}_R)$ having the above property, we start by selecting all relations S that join with R in V with a join condition $\mathcal{C}_{R,S}$ such that $\exists \mathcal{JC}_{R,S}$ in MKB, and $\mathcal{C}_{R,S}$ implies $\mathcal{JC}_{R,S}$. Then for the relations found by this first step, we recursively find others that are joined with them in V with join conditions that imply the corresponding join constraints in MKB, until we cannot find any new relation to add. $Min(\mathcal{H}_R)$ is the join of the relations found using the join constraints from MKB. $Max(V_R)$ is the join of the relations found using the join conditions from the WHERE clause of V.

7.5 R-replacements Associated With R-mappings

Intuitively, we now have found the maximal part of the view definition that "relates" to our MKB (Definition 10). 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 that we can construct using join constraints defined in MKB. Note that 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 $Max(V_R)$ (Equation 35). Only after that, when given the view-extent parameter $\mathcal{VE}_V$ (Section 4) and the $\mathcal{PC}$ constraints from MKB (Section 3), we want to choose the ones that satisfy the properties of legal rewritings (see Definition 3).

Definition 11 ***R-replacement(V,$\mathcal{H}_R(MKB)$).*** *For a given query 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 (see Section 7.1) 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 a subexpression $Max(V_{j,R})$ 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 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. (32) then: $\{R_{v_1}, \ldots, R_{v_l}\} \setminus \{R\} \subseteq \{R_1, \ldots, R_k\}$ and $(\{\mathcal{JC}_{R_{v_1},R_{v_2}}, \ldots \mathcal{JC}_{R_{v_{l-1}},R_{v_l}}\} \setminus \{\mathcal{JC}_{S,S'} \mid S = R \text{ or } S' = R\}) \subseteq \{\mathcal{JC}_{R_1,R_2}, \ldots \mathcal{JC}_{R_{k-1},R_k}\}$. 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.

Example 16 *In Figure 15, the expression $Min(\mathcal{H'}_{Customer})$ defined by Eq. (36) is marked with bold lines: $Min(\mathcal{H'}_{Customer}) = ($**FlightRes**$)$.*

If relations left in $Min(\mathcal{H'}_R)$ are in a connected component of $\mathcal{H'}_R(MKB')$, we construct the set $\{Max(V_{1,R}), \ldots, Max(V_{k,R})\}$ as in the following algorithm.
Computing R-replacement$(V, \mathcal{H}_R(MKB))$.
Step 1. First, we find all the relations that could qualify as **covers** for the indispensable and replaceable attributes of V as required by property (IV) of Definition 11. That is, for any indispensable attribute A of R from the SELECT clause, we find $Cover(A)$, a set of pairs (relation, function-of) from $\mathcal{H'}_R(MKB')$ such that $\forall(S, (R.A = f(S.B))) \in Cover(A)$, S is in $\mathcal{H'}_R(MKB')$ and there exists a function-of constraint $\mathcal{F}_{R.A,S.B}$ in MKB such that $\mathcal{F}_{R.A,S.B} = (R.A = f(S.B))$.
If there exists an indispensable attribute A of R such that $Cover(A) = \emptyset$, then the set R-replacement$(V, \mathcal{H}_R(MKB))$ is empty. In other words, no replacement can be found.
Step 2. Using the covers found at Step 1, we construct the expressions $Max(V_{j,R})$ as required by properties (I) and (III) of Definition 11. In $\mathcal{H'}_R(MKB')$, an expression $Max(V_{j,R})$ corresponds to a connected "path" that must contain all the join constraints (i.e., $\mathcal{JC}$-nodes) and relations (i.e., relation-edges) left in $Min(\mathcal{H'}_R)$ plus for each indispensable attribute A of R, one relation (i.e., the relation-edge) from the $Cover(A)$. The expression $Max(V_{j,R})$ is obtained by adding to this connected path, the conjunction $\mathcal{C'}_{Max/Min}$ defined by property (V) of Definition 11.

Example 17 *We give here an example of how R-replacements are constructed for the view defined by Eq. (29) and $R =$* ***Customer***. *$\mathcal{H'}(MKB')$ is depicted in Figure 15.*

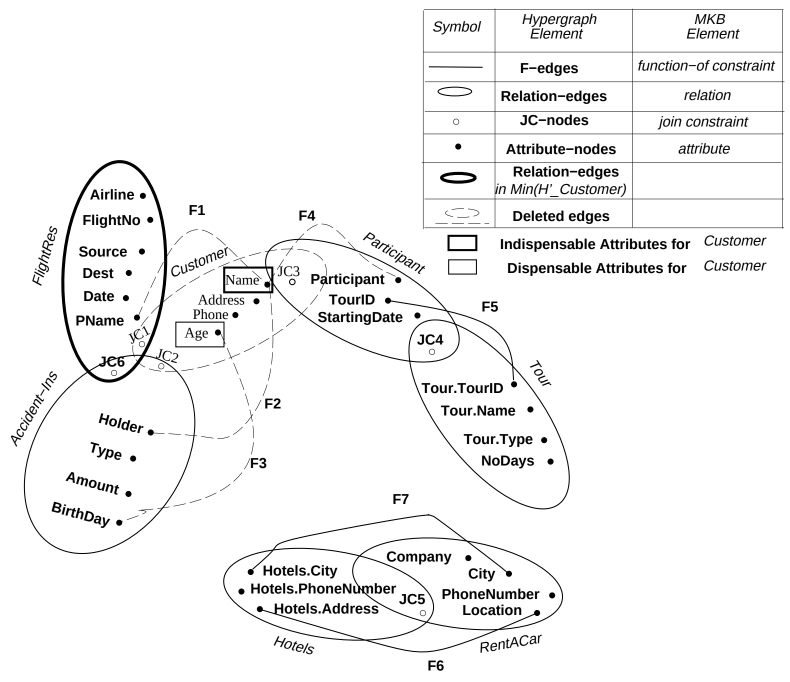

Figure 15: The Hypergraph $\mathcal{H}'$(MKB') for Example 1, View (29) and "delete-relation **Customer**" operator.

*Step 1. In our example, the only indispensable attribute of relation **Customer** is **Customer.Name**. Using the hypergraph depicted in Figure 15, 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**)) }.*
*Step 2. Following property (V) of Definition 11, we have that $\mathcal{C}'_{Max/Min}$ = (**FlightRes.Dest = 'Asia'**).*
*Let us now construct the candidate expressions Max(**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**)) ∈ Cover(**Customer.Name**), we find the following expression that has all the properties from Definition 11.*

$$Max(\textbf{\textit{Customer-Passenger-Asia}}_{1,\textit{Customer}}) \quad =(38)$$

$$\underbrace{\sigma_{(\textit{FlightRes.Dest} \ = \ \textit{'Asia'})}}_{\mathcal{C}'_{Max/Min}} \left(\underbrace{\textbf{\textit{FlightRes}}}_{Min(\mathcal{H}'_{\textit{Customer}})} \underbrace{\bowtie_{(\textit{FlightRes.PName} \ = \ \textit{Accident–Ins.Holder})}}_{\textit{JC6}} \underbrace{\textbf{\textit{Accident–Ins}}}_{in \ Cover(\textit{Customer.Name})} \right)$$

*(2) For the cover (**Participant**, (**Customer.Name** = **Participant.Participant**)) ∈ Cover(**Customer.Name**) we*

*see that there is no connected path in $\mathcal{H}'(MKB')$ (Figure 15) that contains this cover and the relation **FlightRes** (from $Min(\mathcal{H}'_{\textbf{Customer}})$). Thus we cannot generate any replacement for $Max(\textbf{Customer-Passenger-Asia}_{\textbf{Customer}})$ using this cover.*

*(3) For the cover (**FlightRes**, (**Customer.Name** = **FlightRes.PName**)) $\in$ Cover(**Customer.Name**), we find the following expression that has all the properties from Definition 11.*

$$Max(\textbf{Customer-Passenger-Asia}_{2,\textbf{Customer}}) \quad = \tag{39}$$

$$\underbrace{\sigma_{(\textbf{FlightRes.Dest} \ = \ \textbf{'Asia'})}}_{c'_{Max/Min}} \left(\underbrace{\textbf{FlightRes}}_{Min(\mathcal{H}'_{\textbf{Customer}}), in \ \ Cover(\textbf{Customer.Name})} \right)$$

7.6 Putting it all Together: The CVS Algorithm

Now we are ready to give the **Complex View Synchronization (CVS)** algorithm that has as input a view query V, the MKB and a change "delete relation R", and returns all legal rewritings (see Definition 3) of the view V.

Complex View Synchronization (CVS) Algorithm :

CVS(V, ch =delete−relation R, MKB, MKB')

INPUT:

view definition V defined as in Equation (8);

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.

Step 1. Construct the sub-hypergraph $\mathcal{H}_R(MKB)$ as defined in Definition 9.

Step 2. Compute R-mapping(V, $\mathcal{H}_R(MKB)$) = $(Max(V_R), Min(\mathcal{H}_R))$ as defined in Definition 10.

Step 3. Compute R-replacement(V, $\mathcal{H}'_R(MKB')$) = $\{Max(V_{1,R}), \ldots, Max(V_{k,R})\}$ as defined in Definition 11. 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. An evolved query V' is found by replacing $Max(V_R)$ with $Max(V_{j,R})$ in Equation 35; 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. Example 18 below gives some examples of evolved view definitions generated by Step 4 for the view defined by the Equation (29).

Step 5. Set the E-SQL evolution parameters for all V' obtained at Step 4 as defined in Section 5, Definition 4.

Step 6. All the rewritings obtained by Step 4 have properties M2, and M3 from Definition 7, Section 5.3. At this step, we have to check for which rewriting V' obtained in Step 4 the extent parameter $\mathcal{VE}_V$ of the query V is satisfied in order to see if the property P3 from Definition 3 is satisfied and hence property M1 (Definition 7) is satisfied. This issue is similar to the problem of answering queries using views which was extensively studied in the database community [CKP95, LSK95]. However, in our problem domain, we have an added issue of the availability of the set of partial/complete information constraints defined in MKB' that could be used to compare the extent of the initial view V and the extent of the evolved view V'. This development is beyond the scope of current paper but our current work is starting to address this problem [NR98b].

Example 18 *For our view* ***Customer-Passenger-Asia*** *defined by Equation (29), we now show how to apply Steps 4 and 5 from the CVS algorithm and find replacements under the change "delete relation* ***Customer****".*

The expression $Max(\textbf{\textit{Customer-Passenger-Asia}}_{\textit{Customer}}) =$

$$\left(\textbf{\textit{FlightRes}} \bowtie_{\left(\begin{array}{c}\textit{(FlightRes.PName = Customer.Name) AND}\\ \textit{(FlightRes.Dest = 'Asia')}\end{array}\right)} \textbf{\textit{Customer}}\right)$$

(Example 15, Eq. (37)) could be replaced by one of the following expressions found at Step 3 of the ***CVS*** *algorithm:*

(1) $Max(\textbf{\textit{Customer-Passenger-Asia}}_{1,\textit{Customer}}) =$

$\sigma_{\textit{(FlightRes.Dest = 'Asia')}}(\textbf{\textit{FlightRes}} \bowtie_{\textit{(FlightRes.PName = Accident-Ins.Holder)}} \textbf{\textit{Accident-Ins}})$.

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 Equation (29) using this substitution as well is given by Equation (40). There are no contradictions in the WHERE clause after the replacements are done.*

```
CREATE VIEW   Customer-Passengers-Asia_1 AS
SELECT        AI.Holder (AD = false, AR = true), f(AI.Birthday) (AD = true, AR = true),
              P.Participant (AD = true, AR = true), P.TourID (AD = true, AR = true)
FROM          Accident–Ins AI (RD = true, RR = true), FlightRes F (RD = true, RR = true),
              Participant P (RD = true, RR = true)
WHERE         (F.PName = AI.Holder) (CD = false, CR = true) AND (F.Dest = 'Asia')
              AND(P.StartDate = F.Date) AND (P.Location = 'Asia')
```
(40)

(2) $Max(\textbf{\textit{Customer-Passenger-Asia}}_{2,\textit{Customer}}) = \sigma_{\textit{(FlightRes.Dest = 'Asia')}}\textbf{\textit{FlightRes}}$.

A new rewriting of the query (29) is given by the query (41). There are no contradictions in the WHERE clause *after the replacement is done.*

$$\begin{array}{ll}
\textsf{CREATE VIEW} & \textbf{\textit{Customer-Passengers-Asia}}_2 \ \textsf{AS} \\
\textsf{SELECT} & \underline{\textbf{\textit{F.PName}} \ (\mathcal{AD} = false, \mathcal{AR} = true),} \\
 & \underline{\textbf{\textit{P.Participant}} \ (\mathcal{AD} = true, \mathcal{AR} = true)}, \ \textbf{\textit{P.TourID}} \ (\mathcal{AD} = true, \mathcal{AR} = true) \\
\textsf{FROM} & \textbf{\textit{FlightRes F}} \ (\mathcal{RD} = true, \mathcal{RR} = true), \\
 & \textbf{\textit{Participant P}} \ (\mathcal{RD} = true, \mathcal{RR} = true) \\
\textsf{WHERE} & (\textbf{\textit{F.Dest}} = \textbf{\textit{'Asia'}}) \\
 & \textsc{and} (\textbf{\textit{P.StartDate}} = \textbf{\textit{F.Date}}) \ \textsc{and} \ (\textbf{\textit{P.Location}} = \textbf{\textit{'Asia'}})
\end{array} \tag{41}$$

Step 6 in the CVS algorithm requires to verify that the view-extent evolution parameter is satisfied by rewritings found in the previous steps of the algorithm. Unlike the approach proposed for query rewriting using materialized views [LRU96, SDJL96, CKP95, LMS95] our proposed techniques address new issues: (1) finding view rewritings that are not necessarily *equivalent* to the original view definition ($\mathcal{VE} \in \{\subseteq, \equiv, \supseteq\}$), (2) using semantic containment information expressed using $\mathcal{PC}$-constraints for proving that candidate rewritings satisfy the view-extent evolution parameter, and (3) preserving at least indispensable attributes from the SELECT clause if preserving all is not possible. Due to the space limitation we don't present here our work done in this direction [NR98b], insead we simply give an example for when the view-extent parameter could be shown to be satisfied by a $\mathcal{PC}$-constraint.

Example 19 *Let's assume that the view* ***Customer-Passengers-Asia*** *has the view-extent evolution parameter* $\mathcal{VE}$ *set to "$\subseteq$". And the* $\mathcal{PC}$*-constraint shown in Equation (42) is defined in MKB between the relation* ***Customer*** *and the relation* ***FlightRes*** *(i.e., the* ***Customer*** *relation has all the passenger names from* ***FlightRes*** *relation). Then we can prove that the view* ***Customer-Passengers-Asia***$_2$ *defined in Equation (41) is satisfying the view-extent evolution parameter. I.e.,* ***Customer-Passengers-Asia***$_2$ $\subseteq_\pi$ ***Customer-Passengers-Asia*** *for any states of the relations involved in the views (as required by Definition 3, Section 5).*

$$\mathcal{PC}_{Customer,FlightRes} = (\pi_{Name}(\textbf{\textit{Customer}}) \supseteq \pi_{PName}(\textbf{\textit{FlightRes}})) \tag{42}$$

Indeed, let $t' \in$ ***Customer-Passengers-Asia***$_2$ *be a tuple in the new view. Then it must exist the tuples* $t_F \in$ ***FlightRes****, and* $t_P \in$ ***Participant*** *that generate the tuple* t'*. But from the* $\mathcal{PC}$*-constraint defined in Equation (42) there must exist a tuple* $t_C \in$ ***Customer*** *having the same value for the attribute* ***Name*** *as the tuple* t_F *has for the attribute* ***PName****. One can easily see that the tuples* t_C*,* t_F *and* t_P *generate a tuple* $t \in$ ***Customer-Passengers-Asia*** *such that* $t =_\pi t'$*.*

8 Related Work

To our knowledge, we are the first to study the problem of view synchronization caused by capability changes of participating information sources. In [RLN97], we establish a taxonomy of view adaptation problems that identifies alternate dimensions of the problem space, and hence serves as a framework for characterizing and hence distinguishing our view synchronization problem from other (previously studied) view adaptation problems. In

[LNR97a, LNR97b], we then lay the basis for the solutions presented in this current paper by introducing the overall *EVE* solution framework, in particular the idea of associating evolution preferences with view specifications. However, formal criteria of correctness for view synchronization as well as actual algorithms for achieving view synchronization with complex substitutions for "delete-relation" capability change are the key contributions of this current work. Moreover, we introduce in this paper the concept of the strongest synch-equivalent view definition that makes the implicit semantics of the view evolution parameters explicit and give the transformation rules for finding it for a E-SQL view definition.

Gupta et al. [GJM96] and Mohania et al. [MD96] address the problem of how most efficiently to maintain a materialized view 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. Their algorithms could potentially be applied in the context of our overall framework, once *EVE* has determined an acceptable view redefinition. Their results are thus complimentary to our work.

In the work of Levy et al. [LSK95], a global information system is designed using the world-view approach where the external information sources are described relative to the unified world-view relations. The language used here to describe external relations relative to the world-view schema parallels our MKB description language, except the fact that we don't have an apriori defined schema. Further, we introduce the concept of a join constraint in our model that allows expressing default conditions among external relations that should be used by the system to attempt to integrate information instead of evaluating (blindly) all possible Cartesian combinations based on value matches (full disjunction) [NR98a, NR97]. The problem of view evolution as posed by our work, i.e., that the world view itself may evolve, is not discussed in [LSK95].

Papakonstantinou et al. [PGMW95, PGMU96] are pursuing the goal of information gathering across multiple sources. Their proposed language OEM assumes queries that explicitly list the source identifiers of the database from which the data is to be taken. Like our MISD model, their data model allows information sources to describe their capabilities, but they don't assume that these capabilities could be changed and thus they do not address the view synchronization problem.

EVE system can be seen as an information integration system using view technology to gather and customize data across heterogeneous information sources. On this venue, related work that addresses the problem of information integration are among others the SIMS [AKS96] and SoftBot [EW94] projects. In the SIMS project, a unified schema is apriori defined and the user interaction with the system is via queries posed against the unified schema. Although addressing different issues, SIMS's process of translating a user query into subqueries targeting external relations raises some of the same problems as finding the right substitution for an affected view component in *EVE*. The SoftBot project has a very different approach to query processing as they assume that the system has to discover the "link" among data sources that are described by action schemas. While related to our view synchronization algorithms CVS, the SoftBot planning process also has to discover connections among information sources when very different source description languages are used. None of the two projects address the particular problem of evolution under capability changes of participating external information sources.

Research on query reformulation using materialized views Levy et al. [LRU96, LMS95, SDJL96] considers the problem of replacing an original query with a new expression containing materialized view definitions such that the new query is *equivalent* to the old one. To the best of our knowledge, there is no work done in this context of query reformulation using views with the goal of generating queries without *equivalence* (e.g., the new reformulated query could be a subset of the original query). This approach to query reformulation [LMS95] has some similarities with our view synchronization process, but again it is set in a different environment and has different goals. Namely, we have extended the idea of query reformulation by using a well-defined query language E-SQL to specify constraints on query reformulation, thus, when in compliance with those constraints, we allow the view redefinitions to be for example a subset or a superset of the original view. And, if not possible to preseve all view attributes (from the SELECT clause), our view redefinition semantic allows to specify evolution preferences that add flexibility and in the same time let the view definer control the view evolution process.

In the University of Michigan Digital Library project [NR98a], we have proposed the Dynamic Information Integration Model (DIIM) to allow information sources to dynamically participate in an information integration system. The DIIM query language allows loosely specified queries that the DIIM system refines into executable, well-defined queries based on the capability descriptions each information source exports when joining the DIIM system. For this, the notion of *connected relations* is introduced as a natural extension of the concept of full disjunction [GL94]. In the default case when only natural joins are defined in the IS descriptions in the MKB it then can be shown that the semantics of these two concepts (connected rules and full disjunction) are equivalent [NR98a]. AI planning techniques are used in DIIM for query refinement. In *EVE*, instead, we now assume that precise (SQL) queries are used to define views (instead of loosely-specified ones), and thus query refinement in the sense of DIIM is not needed.

9 Conclusion

Our work is the first to study the problem of view evolution in a dynamic environment [RLN97, LNR97a, LNR97b, NLR98, LKNR98, NR98b]. In our *EVE* system, views survive even when the underlying ISs upon which they are defined change their capabilities. One key component of our solution approach is the design of a view specification language based on SQL , called E-SQL, that incorporates user preferences for view evolution. In order to find alternative replacements for components of a view affected by IS capability changes, *EVE* maintains descriptions of the capabilities of ISs as well as interrelationships between ISs in a meta-knowledge base (MKB).

Equipped with E-SQL and the MKB, we propose in this paper strategies for the view synchronization process. First we introduce a formal definition of what is a *legal rewriting* for an affected view definition. Then, we define the concept of synch-equivalence that expresses the real evolution flexibility of an E-SQL view definition. We also propose a general strategy for finding the strongest synch-equivalent E-SQL definition for a given view specification, and provide proofs of the uniqueness of one such strongest definition.

In this paper, we next present the Complex View Synchronization (CVS) algorithm that fully exploits the constraints defined in MISD by allowing relation substitution to be done by a sequence of joins among candidate relations. It can be shown that CVS meets all preservation constraints in the view definition, while finding appropriate information from other ISs as replacement for affected components. CVS finds a new valid rewriting of a view in many cases where current view technology would have simply disabled the view, and where our previous one-step view synchronization (SVS) [LNR97b] would have failed to locate a suitable solution. Examples to illustrate the main ideas are given throughout the paper. In particular, we treat in depth view evolution caused by the "delete-relation" capability change. To summarize, the main contributions of this paper are:

- We have presented a formal description of when a view rewriting generated as response to an IS capability chance is considered to be *legal*.
- We introduce in the concept of *synch-equivalence* between E-SQL specifications to express the real evolution flexibility of an E-SQL view definition.
- We give a general strategy for finding the strongest synch-equivalent E-SQL definition for a given view specification, and prove this transformation process to always generate the unique strongest solution.
- 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 *EVE* work to be done include the exploration of alternate view evolution preference models, MKB evolution as well as cost models for maximal view preservation [LKNR98].

References

[AKS96] Y. Arens, C. A. Knoblock, and W.-M. Shen. Query Reformulation for Dynamic Information Integration. *Journal of Intelligent Information Systems*, 6 (2/3):99–130, 1996.

[CKP95] S. Chaudhuri, R. Krishnamurthy, and S. Potamianos. Optimizing Query with Materialized Views. In *Proceedings of IEEE International Conference on Data Engineering*, 1995.

[EW94] O. Etzioni and D. Weld. A Softbot-Based Interface to the Internet. *Communication of ACM*, 1994.

[GJM96] A. Gupta, H.V. Jagadish, and I.S. Mumick. Data Integration using Self-Maintainable Views. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, 1996.

[GL94] C. Galindo-Legaria. Outerjoins as disjunctions . *Proceedings of SIGMOD*, 1994.

[LKNR98] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Data Warehousing Evolution: Trade-offs between Quality and Cost. Technical Report WPI-CS-TR-98-2, Worcester Polytechnic Institute, Dept. of Computer Science, 1998.

[LMS95] A.Y. Levy, A.O. Mendelzon, and Y. Sagiv. Answering Queries Using Views. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 95–104, May 1995.

[LNR97a] A. J. Lee, A. Nica, and E. A. Rundensteiner. Keeping Virtual Information Resources Up and Running. In *Proceedings of IBM Centre for Advanced Studies Conference CASCON97, Best Paper Award*, pages 1–14, November 1997.

[LNR97b] 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.

[LRU96] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external processors. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–237, Montreal, Canada, 3–5 June 1996.

[LSK95] A. Y. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *Journal of Intelligent Information Systems. Special Issue on Networked Information Discovery and Retrieval*, 1995.

[MD96] M. Mohania and G. Dong. Algorithms for Adapting Materialized Views in Data Warehouses. *International Symposium on Cooperative Database Systems for Advanced Applications*, December 1996.

[NLR97] 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.

[NLR98] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. *To appear in Proceedings of International Conference on Extending Database Technology (EDBT'98)*, Valencia, Spain, March 1998.

[NR97] A. Nica and E. A. Rundensteiner. On Translating Loosely-Specified Queries into Executable Plans in Large-Scale Information Systems. In *Proceedings of Second IFCIS International Conference on Cooperative Information Systems CoopIS'97*, pages 213–222, June 1997.

[NR98a] A. Nica and E. A. Rundensteiner. Loosely-Specified Query Processing in Large-Scale Information Systems. *To appear in International Journal of Cooperative Information Systems*, 1998.

[NR98b] A. Nica and E. A. Rundensteiner. Using Containment Information for View Evolution in Dynamic Distributed Environments. Technical Report WPI-CS-TR-98-3, Worcester Polytechnic Institute, Dept. of Computer Science, 1998.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A Mediation System Based on Declarative Specifications. In *Proceedings of IEEE International Conference on Data Engineering*, 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[RLN97] E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proceedings 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.

[SDJL96] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy. Answering Queries with Aggregation Using Views. In *International Conference on Very Large Data Bases*, pages 318–329, 1996.

[Ull89] J.D. Ullman. *Principle of Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[Wid95] J. Widom. Research Problems in Data Warehousing. In *Proceedings of International Conference on Information and Knowledge Management*, pages 25–30, November 1995.