POLITECNICO DI MILANO

Dipartimento di
Elettronica e Informazione

Context schema evolution in context-aware data management

Elisa Quintarelli, Emanuele Rabosio, Letizia Tanca

Elisa Quintarelli, Emanuele Rabosio – 25th May 2011

Introduction

- The current technological scenario is characterized by an extremely large variety of information sources, providing the users with an enormous amount of data
- This constitutes an unprecedented opportunity, but at the same time risks to confuse and overwhelm them
- The problem is particularly relevant when users employ portable devices, with limited memory availability
- A possible solution to this problem is **context-based data tailoring**: the system allows a user to access only the view that is relevant for his/her context
- The context describes the situation in which the user is involved through a series of dimensions

Context schema evolution

- The context dimensions and their values (together constituting the **context schema**) useful for data tailoring depend on the application requirements
- Application requirements are intrinsically dynamic and thus can change
- The change of requirements can be due to various reasons:
  - Changes in business policies
  - Market developments
  - Technology developments
- The changes in the application requirements lead to **context schema evolution**

→ *facilitate the modification of the context schema making the contexts defined according to old schemas still utilizable*

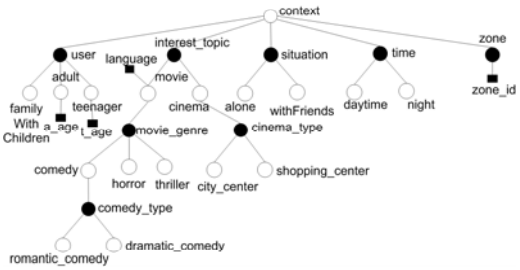
Context schema evolution: Examples

- Context perspectives in the movie domain:
  - The **kind of user**: e.g., adult, teenager or family with children
  - The **situation**: e.g., alone or with friends
  - The **time**: e.g., daytime or night
  - The **location**
- Possible evolution scenarios:
  - The company may change its business policy, removing the distinction between daytime and evening schedule
  - Marketing researches could reveal that teenagers and adults show the same behavior, thus making the distinction among them useless
  - Technological changes may make new kinds of devices available, suggesting to tailor data also on the basis of the device type

Context model: Context Dimension Tree (CDT)

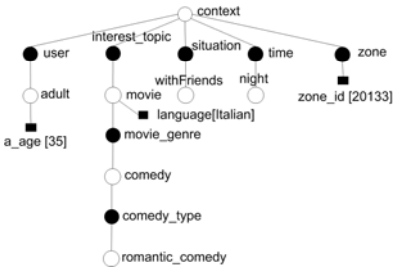
5

- Provides a tree-based representation of context schemas and instances
- A context instance is a subtree of the related schema



Context schema

A context instance



CDT formal definition
Context schema

6

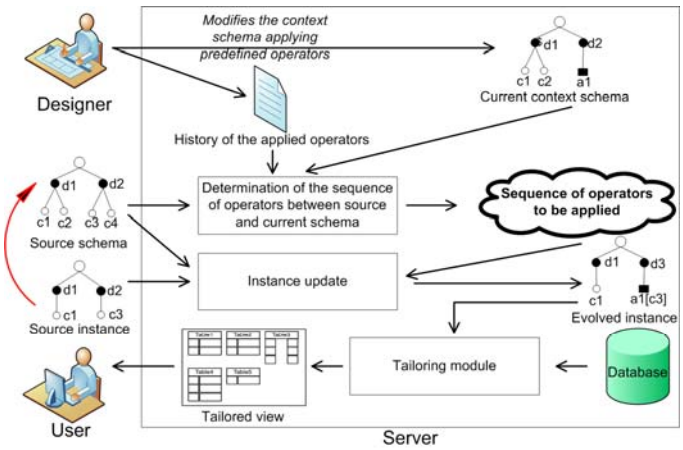
- A **context semischema** is a tuple *(N, E, r, Att, α, λ)*
  - *N*: set of node identifiers, partitioned in dimensions ($N^{\prime}$) and concepts (N^{c})
  - *E*: set of directed edges such that *(N, E)* is a tree
  - $r \in N$: root of the tree *(N, E)*
  - *Att*: set of attribute identifiers
  - *α: Att → N*: a function assigning a node identifier to each attribute
  - *λ: N U Att →* $\mathcal{V}$: a labeling functions, associating each node identifier and each attribute identifier with a value in a set of labels

- A **context schema** is a context semischema *(N, E, r, Att, α, λ)* such that:
  - *r* is a concept node
  - *λ(r) = "context"*

CDT formal definition
Context instance

- A **context semi-instance** is a pair (S, ρ):
  - $S = (N, E, r, \alpha, \lambda)$: a context semischema where each dimension node without attributes has exactly one child
  - $r: Att \rightarrow \mathcal{V}$: a function assigning a value to each attribute identifier
- A **context instance** is a context semi-instance (S, ρ) such that S is a context schema
- An instance $I = (S_I, \rho_I)$ is an **instance of $S = (N_S, E_S, r_S, \alpha_S, \lambda_S)$** if there exist two functions $h_N: N_I \rightarrow N_S$ and $h_A: Att_I \rightarrow Att_S$ such that:
  - $h_N(r_I) = r_S$
  - for all $(n_1, n_2) \in E_I, (h_N(n_1), h_N(n_2)) \in E_S$
  - for all $n \in N_I, att \in Att_I$, if $n = \alpha_S(a)$ then $h_N(n) = \alpha_S(h_A(a))$
  - for all $n \in N_I, \lambda_I(n) = \lambda_S(h_N(n))$
  - for all $a \in Att_I, \lambda_I(a) = \lambda_S(h_A(a))$
- *Observation*: both context schemas and context instances may be represented as **XML documents**, the latter containing a subset of the elements of the former

Framework for context evolution

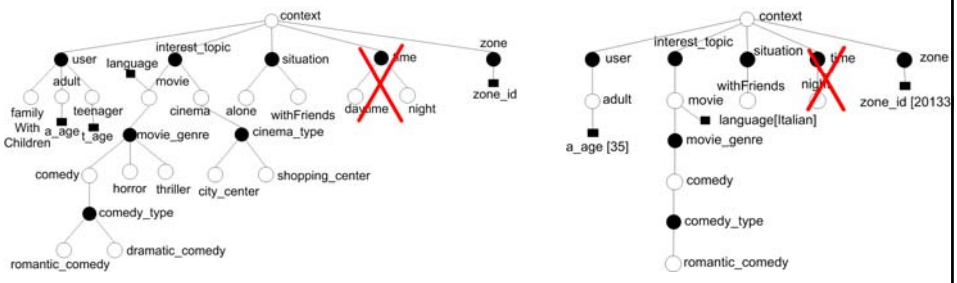


Evolution operators

- An **update operation** *op* is implemented by two operators:
  - A **schema evolution operator** SU_{op}
  - An **instance evolution operator** IU_{op}
- The schema operators, employed by the designer to modify the schema, are characterized by a set of **preconditions** imposing restrictions on the source schema on which they are applicable
- The instance operators adapt the instances to the new schema, trying to preserve as much information as possible
- Two categories of operators:
  - **Atomic operators**:
    - *Minimal*: each operator cannot be obtained as a combination of other ones
    - *Complete*: allow to evolve to any valid target context schema
  - **High-level operators**: compactly express common evolution needs

Atomic evolution operators
Delete

- SU_{Delete}
  - Eliminates the subtree rooted in a node *n* from the source schema
  - A dimension must have either an attribute or a concept child, therefore it is possible to remove either a dimension node (and its subtree) or a concept node with at least another sibling (and its subtree)
- IU_{Delete}
  - Eliminates from the instance the subtree rooted in the node corresponding to *n*, if such a node is present
  - If *n* is a concept, the node corresponding to its father must be removed too



Atomic evolution operators
Delete (2)

- Preconditions and semantics of Delete

Delete

$\mathsf{SU}_{Delete} : \mathcal{S} \times \mathcal{N} \rightarrow \mathcal{S}$

$\mathsf{SU}_{Delete}(S_S, n) = S_T$

- $N_T = N_S \setminus \widetilde{desc}(S_S, n)$
- $E_T = E_S \setminus \{(n_1, n_2) \in E_S : n_2 \in \widetilde{desc}(S_S, n)\}$
- $r_T = r_S$
- $Att_T = Att_S \setminus \{a_1 \in Att_S : \alpha_S(a_1) \in \widetilde{desc}(S_S, n)\}$
- $\alpha_T(a_1) = \alpha_S(a_1)$ if $a_1 \in Att_T$
- $\lambda_T(n_1) = \lambda_S(n_1)$ if $n_1 \in N_T \cup Att_T$

$\mathsf{IU}_{Delete} : \mathcal{S} \times \mathcal{S} \times \mathcal{I} \times \mathcal{N} \rightarrow \mathcal{I}$

$\mathsf{IU}_{Delete}(S_S, S_T, I_S, n) = I_T$

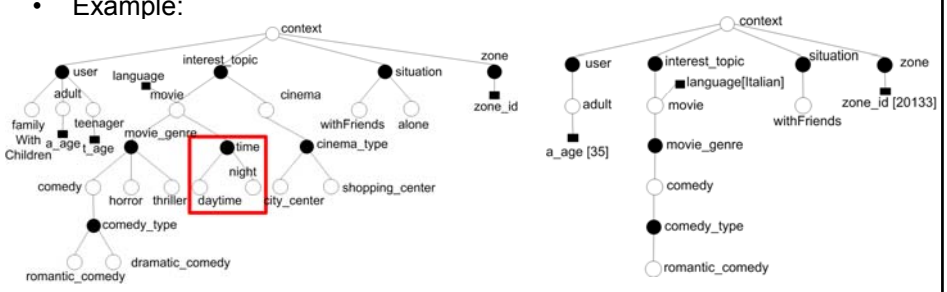
- $N_{IT} = \{n_1 \in N_{IS} : h_N(n_1) \in N_T \wedge (n_1 \in N^{\bullet}_{IS} \Rightarrow (\nexists n_2 \in children(S_{IS}, n_1))(h_N(n_2) = n))$
- $E_{IT} = \{(n_1, n_2) \in E_{IS} : (h_N(n_1), h_N(n_2)) \in E_T \wedge (n_2 \in N^{\bullet}_{IS} \Rightarrow (\nexists n_3 \in children(S_{IS}, n_2))(h_N(n_3) = n))\}$
- $r_{IT} = r_{IS}$
- $Att_{IT} = \{a_1 \in Att_{IS} : h_A(a_1) \in Att_T\}$
- $\alpha_{IT}(a_1) = \alpha_{IS}(a_1)$ if $n_1 \in Att_{IT}$
- $\lambda_{IT}(n_1) = \lambda_{IS}(n_1)$ if $n_1 \in N_{IT} \cup Att_{IT}$
- $\rho_{IT}(a_1) = \rho_{IS}(a_1)$ if $a_1 \in Att_{IT}$

Preconditions

1) $n \in N_S$, 2) $n \in N^{\circ}_S \Rightarrow siblings(S_S, n) \neq \emptyset$, 3) $n \neq r_S$

Atomic evolution operators
Insert

- *SU*$_{Insert}$
  - Inserts a semischema *R* as a child of a specified node *n*
  - The insertion must preserve the correct type alternation and must not introduce label conflicts
- *IU*$_{Insert}$
  - Since *SU*$_{Insert}$ does not alter the existing nodes and attributes, the instances are not affected
- Example:



Completeness

Theorem

Given two arbitrary context schemas S_1 and S_2, it is possible to find a finite sequence of operators belonging to *{SUInsert, SUDelete}* that transforms S_1 into S_2.

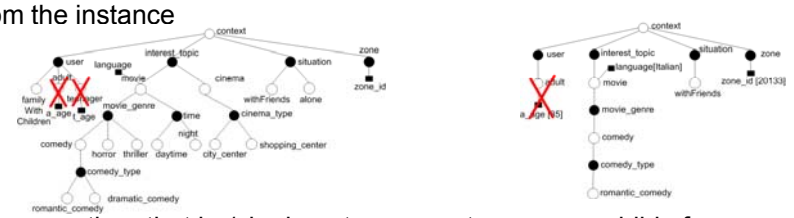
Proof

Let us consider two context schemas S_S and S_T.
Let $c_{S1}, \ldots, c_{Sn}$ be the children of r_S and $c_{T1}, \ldots, c_{Tm}$ the children of r_T.
Let $T_{T1}, \ldots, T_{Tm}$ be the subtrees rooted in $c_{T1}, \ldots, c_{Tm}$.
The following sequence of *Delete* and *Insert* operations builds S_T starting from S_S:

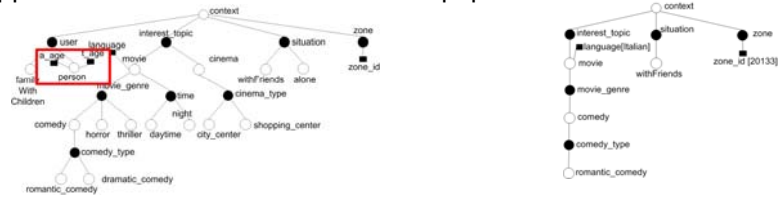
$S_0 = S_S$
for i: 1, …, n, $S_i = SU_{Delete}(S_{i-1}, c_{Si})$
S0 = Sn
for i: 1, …, m, $S_i = SU_{Insert}(S_{i-1}, T_{Ti}, r_{Si})$
$S_T = S_m$

Further atomic evolution operators
Replace

- Suppose that the designer delete from the schema the concepts *adult* and *teenager* – children of *user* –, triggering the removal of the *user* dimension from the instance



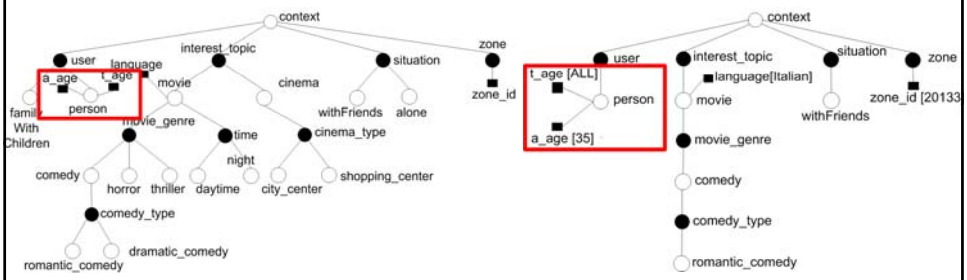
- Suppose then that he/she insert a concept person as child of user



- The instance remains unaltered, but this operation intuitively represents a "replace"

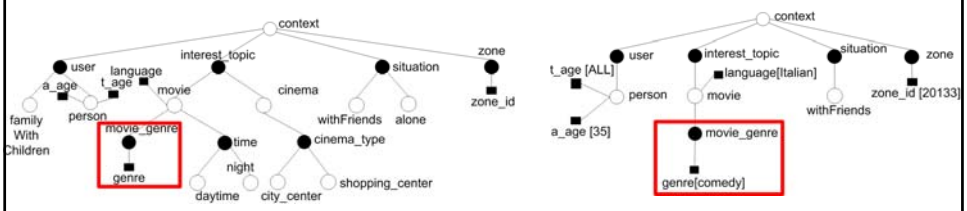
Further atomic operators Replace (2)

- $SU_{Replace}$
  - Substitutes a set of concept siblings with a unique node labeled ℓ
  - Label conflicts must not be introduced
- $IU_{Replace}$
  - If an instance contains one of the replaced nodes, it is substituted with a new node labeled ℓ
- Example:



Further atomic operators ReplaceSubtreesWithAttribute

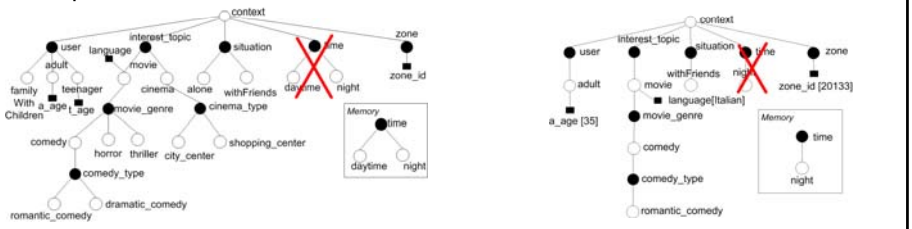
- $SU_{RepSubWithAttr}$
  - Replaces all the subtrees rooted in a dimension node *n* with an attribute
- $IU_{RepSubWithAttr}$
  - If an instance contains a node *k* corresponding to one of the children of *n*, its subtree is removed and replaced by the new attribute whose value will be the label of *k*
- Example:



Further atomic operators
Memory functionality

17

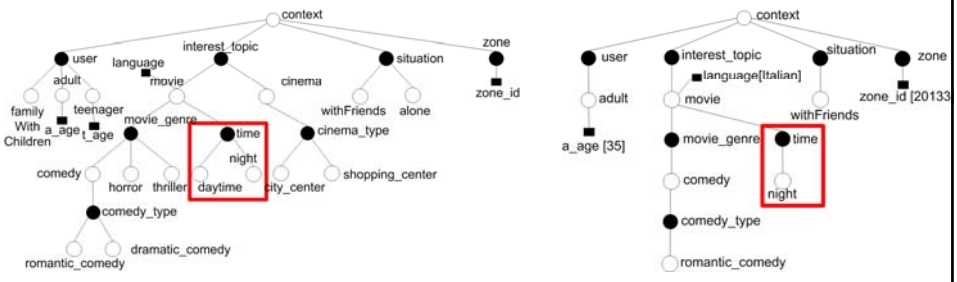
- Recall the previous examples:
  - The dimension *time* was eliminated from the schema causing the same deletion on the instance
  - The same dimension *time* was reinserted under the concept node *movie*
- This sequence of changes intuitively represents a *"move"*
- To recognize this fact, it is necessary to cache the subtrees deleted from the schema and instance
- The delete operation is enriched with two new operators:
  - **Schema cache operator**: defines the semischema to store
  - **Instance cache operator**: defines the semi-instance to store
- Example:



Further atomic operators
InsertFromMemory

18

- $SU_{InsertFM}$
  - Behaves exactly as SU_{Insert} does, but the semischema to be inserted is retrieved from the cache
  - It is allowed to modify attributes and labels, but not nodes and edges
- $IU_{InsertFM}$
  - If the new subtree belonged to the instance in the past, it is reintegrated
- Example:



Soundness of the evolution process

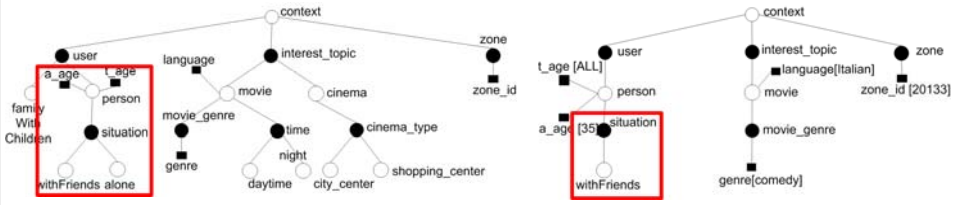
- Every update operation guarantees to produce a well-formed schema and well-formed instances, and to maintain the consistency between context instances and schemas

Theorem (Soundness of the schema evolution)
Let S_s be a context schema, SU_{op} a schema evolution operator. Then the application of SU_{op} to S_S gives as result a context schema S_T.

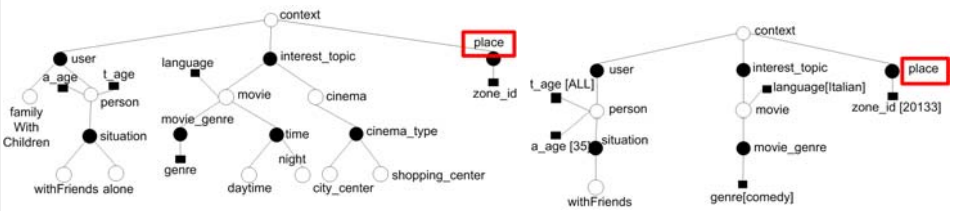
Theorem (Soundness of the instance evolution)
Let S_S be a context schema, Is an instance of S_S, SU_{op} a schema evolution operator, IU_{op} an instance evolution operator, S_T the results of the application of SU_{op} to S_S. The result I_T obtained applying IU_{op} to I_S is an instance of S_T.

High-level evolution operators

- **Move**: moves a subtree
  (*Delete the subtree + InsertFromMemory in the new position*)

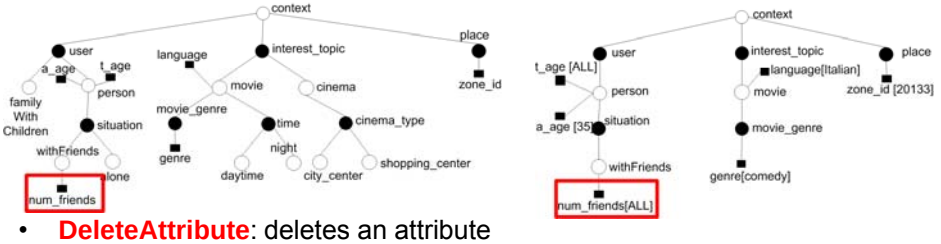


- **Rename**: renames a node or an attribute
  (Delete a subtree with the node/attribute + InsertFromMemory changing the label)

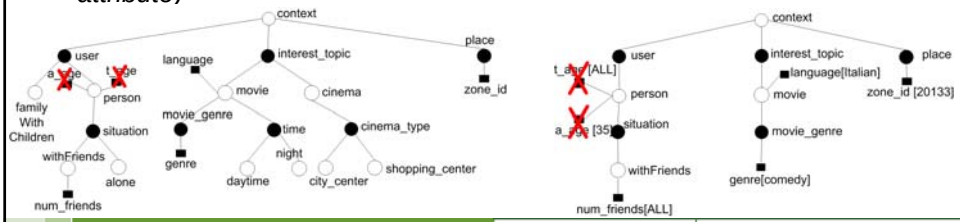


High-level evolution operators (2)

- **InsertAttribute**: inserts a new attribute, connected to a node *n*
  (Delete the subtree rooted in n + InsertFromMemory adding the attribute)



- **DeleteAttribute**: deletes an attribute
  (Delete a subtree with the attribute + InsertFromMemory deleting the attribute)



Conclusions and future work

- In this work we have investigated the problem of context schema evolution
- A sound and complete set of schema evolution operators has been introduced
- Currently, we are studying how the application of the proposed operators influences data tailoring
- Moreover, in the future we plan to explore techniques to optimize the sequence of operators applied to migrate from the initial instance to the instance of the target schema