Context-Driven Information Base Update

Panos Constantopoulos and Yannis Tzitzikas

Department of Computer Science, University of Crete
and
Institute of Computer Science,
Foundation for Research and Technology - Hellas
email : $\{panos \mid tzitzik\}@ics.forth.gr$

Abstract

A major issue in information management is information update. In this paper we address this problem within the framework of a fairly general information model incorporating notions of context as a way to control and drive the update operations. We introduce three kinds of contexts: role,task and focus context. *Role context* relates users with update tasks, *task context* restricts the scope of updates and *focus context* is used to guide the user at run-time. In particular, we focus on issues regarding the expressive power of contexts, the consistency of context declarations with the contents of the information base, flexibility and brevity of context declaration and management and utilization of contexts. An implementation is proposed as part of a specific system, the Semantic Index System.

1 Introduction

It might be common place to claim that building and managing very large information bases is an activity of ever growing importance. By *information bases* we refer collectively to the various kinds of databases (relational, object-oriented, deductive, etc.), knowledge bases, hypermedia bases, etc. There seems to be a clear need emerging for heterogeneous information bases, obtained from disparate sources, to be managed in an integrated way and to be accessed by a range of applications and tools. Another clear trend appears to be the management of metadata. This has been particularly established for information bases on engineered artifacts, in which case *repository systems* have been proposed as a framework for metadata management [5]. However, metadata management is essentially a ubiquitous problem : legal databases, historical documentation, medical databases, educational and research information bases are but a few examples where metadata can be used to advantage besides engineering databases.

Two necessary ingredients of a general approach to information base management are [5, 21]: (i) a common information model and (ii) contexts, as meaningful decompositions of information. In this paper we address the problem of updating an information base. This is done within the framework of a fairly general information model, enabling uniform treatment of data and metadata and using notions of *context* to control and drive the update operations.

The information base management system must at least support a set of *primitive update operations* and an *update process*. The latter involves update transactions which are sequences of primitive operations. These sequences can be (i) predetermined, automatically executable or requiring some user input, or (ii) they may be dynamically generated. Dynamic update sequences are mostly the case in creative and judgemental applications, e.g. CAD. It is this class of applications that also call for metadata update and evolution, which we are primarily concerned with.

We introduce three kinds of *context* for driving the update process : *role* context, *task* context and *focus* context. Users assume roles, to which update tasks are assigned. The role context relates a user with his/her roles and the corresponding tasks. The task context restricts the scope of the updates to a subset of the information base, defined according to various criteria (filtering, authorization). The focus context finally guides the user to execute the primitive update operations needed by the current task.

A metamodeling approach is taken for defining the update contexts. This yields significant benefits in terms of maintaining context consistency with the information base contents. Positive and negative declarations (exceptions) are supported, as well as configurable composite declaration types and reuse of declarations, thus offering flexibility and efficiency.

Section 2 reviews the framework used for information representation and management. Section 3 introduces the primitive update operations. Section 4 discusses the issues of information update which motivate our work. Section 5 presents the update contexts and their use. Section 6 discusses implementation aspects. Section 7 reviews related work, while section 8 draws some conclusions.

2 Information Representation and Management

The information representation framework we adopt is the language Telos [20], an object-oriented knowledge representation language that supports a number of structuring mechanisms as well as an assertional and temporal reasoning sublanguage. In particular we confine ourselves to a version of the structural part of the Telos language, which we have implemented into a system for the management of very large collections of highly interrelated information objects with evolving structures, called the Semantic Index System (hereafter SIS) [10, 9]. This system is especially well suited for use as a repository system, providing metadata management and the kernel of an integrated environment of a dynamic collection of tools [5].

2.1 The SIS Data Model

The SIS data model complies with the structural part of the Telos language, which offers mechanisms analogous to those supported by semantic networks and semantic data models.

An information base consists of structured objects built from two kinds of primitive units: individuals (I) and attributes (A) . Individuals represent entities while attributes represent binary relationships from individuals or attributes to individuals. Individuals and attributes can be concrete or abstract, and they are commonly referred to as objects (O). They have unique system (internal) identifiers and they can be named.

Objects are organized along three dimensions : attribution, classification and generalization [2, 7, 16, 20]. A distinctive feature of Telos and, consequently, of the SIS data model, is the uniform treatment of individuals and attributes. This allows attributes to be organized in classification and generalization hierarchies and to have attributes of their own, which provides great expressive power and flexibility.

Multiple classification is allowed, supporting the separate representation of multiple modeling aspects. An open-ended classification hierarchy is possible. Atomic objects, that is, objects which cannot have instances are called tokens. These are instances of classes which are instances of meta-classes and so on. Every object must be declared as an instance of one system class. System classes (O_{sys}) are special classes which partition the information base according to two criteria: (i) instantiation level and (ii) object type (individual,attribute).

Classes within a given instantiation level are also organized in terms of generalization (or isA) relationships. These can be multiple and give rise to hierarchies that are directed acyclic graphs. They induce strict inheritance of attributes, in the sense that inherited attributes cannot be overridden but only restricted by the definition of the subclass.

Figure 1 gives a graphical example of a SIS-Telos information base.

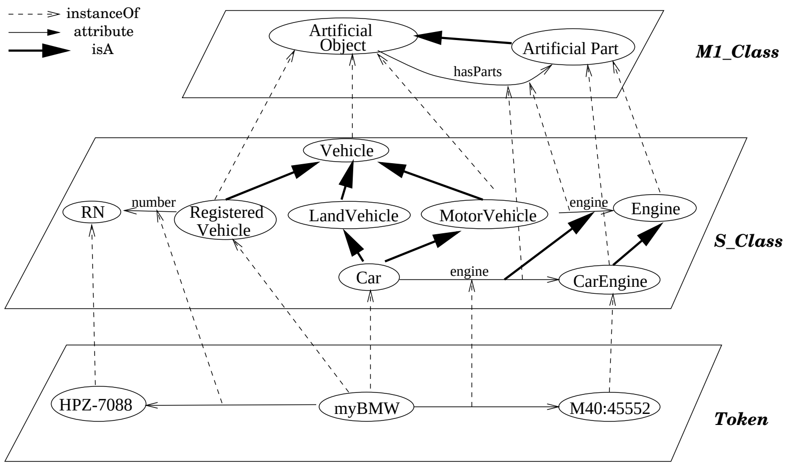


Figure 1: A SIS-Telos information base

2.2 The Semantic Index System

The Semantic Index System (SIS) is a tool for describing and documenting large evolving varieties of highly interrelated data, concepts and complex relationships, as opposed to large homogeneous populations in fixed formats (handled by traditional DBMS). As such, it is suited for the representation of scientific knowledge and engineering designs or constructs. These kinds of data are also characterized by relative stability, i.e. they undergo fewer updates than, say, administrative, financial or observational data, which give rise to continuously changing sets of uniform items.

The SIS persistent storage mechanism is based on the SIS-Telos data model described in section 2.1 and supports transactions and concurrency control.

The user interface supports menu-guided and form-based query formulation with graphical and textual presentation of the answer sets. It also supports graphical browsing and navigation in a hypertext-like manner. A hypertext annotation mechanism is also provided. Menu titles, menu layout and domain-specific queries are user-configurable. Thus, the user interface can be customized to the application without changing the executable code.

A form-based interactive data entry facility is provided. It allows for entering data and schema information in a uniform manner. By employing the schema information, it automatically adapts itself to the structure of the various classes and subclasses. Furthermore, it is customizable to application-specific tasks, such as classification of items, addition of descriptive elements, etc.

An API for communication with other tools is provided.

So far, SIS has been used as the kernel for various applications, such as the Software Static Analysis and Class Management System [9], the CLIO Cultural Documentation System [8], and prototype systems for hypermedia presentations [11], thesaurous management and mechanical fault documentation and diagnosis.

3 Primitive Update Operations

Each field of the SIS storage structures is subject to one or more *generic update actions* (add, delete, change, create, destroy). For each field and its related generic update action(s), we assign an identifier. These identifiers constitute the set of *elementary action identifiers*, EA, and are listed in table 1 together with their meaning.

EA	Storage Field	Meaning
REN	Name	object renaming
DEL	Internal ID	object deletion
$AddAF$	AttrsFrom	addition of an attribute
$DelAF$	AttrsFrom	deletion of an attribute
$AddAT$	AttrsTo	addition of an attribute reference
$DelAT$	AttrsTo	deletion of an attribute reference
$AddIn$	Instances	addition of an instantiation link
$DelIn,$	Instances	deletion of an instantiation link
$AddClass$	Classes	addition of a classification link
$DelClass$	Classes	deletion of a classification link
$AddSub$	Subclasses	addition of a generalization link
$DelSub$	Subclasses	deletion of a generalization link
$AddSup$	Superclasses	addition of a specialization link
$DelSup$	Superclasses	deletion of a specialization link
$CrObj$	systemclass.insts	creation of a new object
$DelObj$	systemclass.insts	deletion of an object

Table 1: The set of elementary action identifiers

We call *primitive update operations* (PUO) the finest update operations that leave the base consistent (according to our data model). These are listed in table 2, while examples of their use are drawn in figure 2.

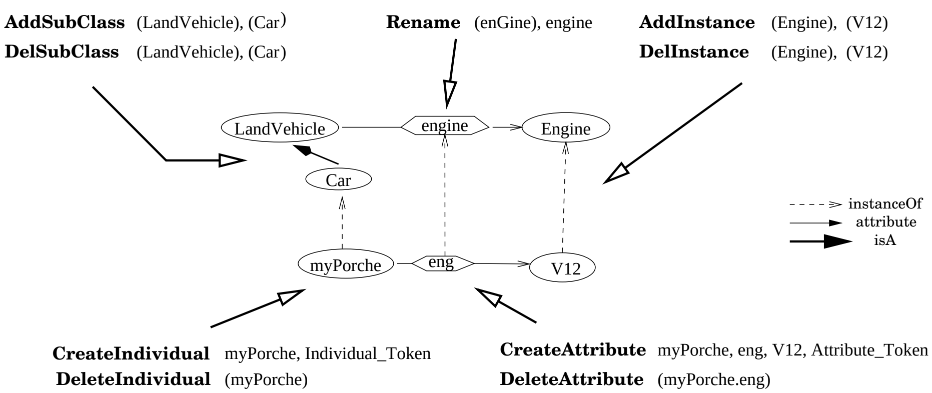


Figure 2: Examples of primitive update operations
A parenthesized logical name stands for the internal identifier of the object with that logical name.

Each primitive update operation causes one or more elementary update actions on the objects it takes as arguments. This mapping is shown in table 3. If U is the set of all primitive update operations of a base, table 3 essentially defines the function $ea : U \rightarrow 2^{O \times EA}$ which, for each $u \in U$, returns the pairs (object, elementary action identifier) of the corresponding elementary actions.

4 Issues of Information Update

Information bases for creative and judgemental applications present in particular the following requirements of the update operations : (i) *schema evolution* at run-time in order to capture new designs, concepts, aspects or changes thereof ; (ii) *update control* as a way to secure

Prim. Update Op.	Arguments	Description
CreateIndividual	SysClass,name	Creates an individual object
CreateAttribute	from, name, to, SysClass	Creates an attribute object
DeleteIndividual	o	Deletes an individual object
DeleteAttribute	a	Deletes an attribute object
Rename	o,*name'*	Renames object o as *name'*
AddInstance	a,b	Makes b an instance of a
AddSubClass	a,b	Makes b a subclass of a
DeleteInstance	a,b	Deletes b from the instances of a
DeleteSubClass	a,b	Deletes b from the subclasses of a

Table 2: The primitive update operations
SysClass represents a system class identifier, *name* a logical name, *o*, *from* are object identifiers, *to* is either an object identifier or a primitive value (int,float,char$*$) and *a*, *b* are object identifiers which are both either individuals or attributes.

Primitive Update Operation	Elementary Actions
CreateIndividual n, S	$(S, CrObj)$
CreateAttribute *from* , *to* , n, S	(*from* , $AddAF$), (*to* , $AddAT$), $(S, CrObj)$
DeleteIndividual *o*	(*o* , DEL), ($sys($ *o* $)$, $DelObj$)
DeleteAttribute a	$(a, DEL), (from(a), DelAF),$ $(to(a), DelAT), (sys(a), DelObj)$
Rename *o* , $newname$	(*o* , REN)
AddInstance *a* , *b*	(*a* , $AddIn$), (*b* , $AddClass$)
DeleteInstance *a* , *b*	(*a* , $DelIn$), (*b* , $DelClass$)
AddSubClass *a* , *b*	(*a* , $AddSub$), (*b* , $AddSup$)
DeleteSubClass *a* , *b*	(*a* , $DelSub$), (*b* , $DelSup$)

Table 3: Analysis of primitive update operations to elementary actions
The function $sys(o)$ returns the system class of object o, while the functions $from(a)$ and $to(a)$ return the starting and the ending point of an attribute a respectively.

data and facilitate interactive updates (filtering), taking account of the application domain, user and task ; (iii) *update process driving*, for user guidance ; (iv) *combination of browsing and update* at run-time. Thus, it should be possible to answer efficiently questions of the form: ”does this primitive operation belong to an update task of the current user ?” (update control), or ”which primitive update operations are related to the current object, current user and current task?” (guidance).

Schema evolution at run-time is supported directly by the SIS data model and system. Therefore we focus on issues concerning update control and update process driving. Update control capabilities depend mainly on the way that constraints are represented. Below we discuss some existing approaches and present our proposal through an example.

Consider a base with the schema shown in figure 3 and assume that `Manos` is the database administrator and `Charoula` is an ordinary user. We want to allow `Charoula` to update all the instances of the base, except for the attributes of class `born` . We will refer to this part of the base as `Charoula` 's context.

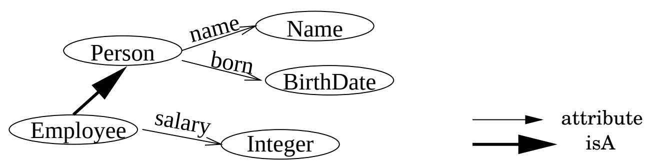


Figure 3: Example schema

(I) One approach is to define a virtual class by a query [14, 4] and attach it to a real class. The definition could be :

> **DEFINE VIEW** `EmployeeSecr` **AS**
> **SELECT** [`Person. name, Employee. salary`]
> **FROM** `Employee` [1]

Only the definition, i.e. the query string, is stored by the DBMS. This approach introduces some problems : If `Manos` decides to delete or rename the attribute `salary` as `Salary` , then `EmployeeSecr` loses its integrity, hence it needs redeclaration. Moreover, `Manos` does not get any warning about this. Additionally, a virtual class cannot be used for the definition of other virtual classes and in order to reference a virtual class through an attribute, we have to define a virtual class of the referencing type, too . In general, the management of virtual classes is neither flexible, nor uniform with the management of real classes.

Continuing with the example, in order to prevent `Charoula` from updating a specific person, say `Anna` , we have to redeclare `EmployeeSecr` as follows :

> **DEFINE VIEW** `EmployeeSecr` **AS**
> **SELECT** [`Person. name, Employee. salary`]
> **FROM** `Employee`
> **WHERE** [`Person. name <> "Anna"`]

Finally this approach requires `Manos` to learn and use a query language.

(II) A second approach is to define a view class by a query, which is now embodied in the schema of the base as an ordinary class [23, 1, 19]. This results in schema reorganization. For example, the following declaration will make the schema look as in figure 4.

[1]Virtual classes of only one class are supported.

DEFINE VIEW `EmployeeSecr` **AS**
SELECT [`Person. name, Employee. salary`]
FROM `Employee`

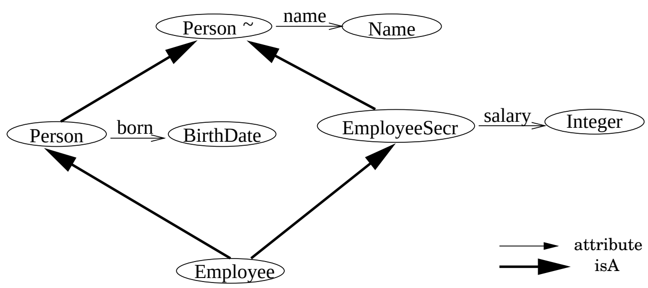


Figure 4: View `EmployeeSecr` is embodied in the schema as a class

In particular, the projection of a class is represented as a superclass of that class containing only the attributes to be projected. Now, `Manos` can rename/delete attributes without causing any problem to `Charoula` 's context. Unfortunately, other problems appear : the schema is extended (fragmented) very much (imagine defining many different contexts) and no symmetrical operation is proposed for contracting the schema (UNDO is not supported). In addition naming problems appear : how to name the new automatically constructed classes (eg: Person~)?. Moreover, `Manos` will be confused when trying to add a new person (which class to choose), or delete one (in which class to search).

Here, the protection of `Anna` from `Charoula` ‘s updates requires the definition of a new view class :

DEFINE VIEW `EmployeeSecr2` **AS**
SELECT *
FROM `EmployeeSecr,`
WHERE [`name<> "Anna"`]

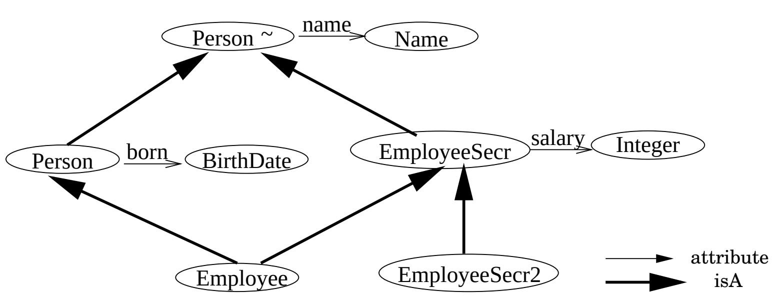


Figure 5: View `EmployeeSecr2` further extends the schema

resulting in the schema shown in figure 5. A new view class was created (`EmployeeSecr2`) which became subclass of the class `EmployeeSecr` . Although this is a simple case, if we want more than one selection views, the problem of predicate subsumption arises, which is undecidable in general.

(III) A third approach is to use an authorization control mechanism. In order to define the desired context, a number of grant/revoke commands must be executed. Usually such mechanisms store ownerships in the actual data. This, apart from large storage requirements, sometimes makes the context hard to define: eg. the population of `Employee` may be owned by many different users, therefore `Manos` will have to execute a number of granting commands.

Furthermore, these mechanisms usually organize hierarchically operation types, authorization objects and users [22, 25, 28]. This is not what we need because we may want to define a context containing only a part of the schema but not the corresponding data (instances). This is desired in cases that we want to define a context to act as a filter and not as an authorization constraint. Since a user can be assigned to more than one role contexts or update tasks, we can assign him/her a context permitting updates on the corresponding data, too.

(IV) Here we propose a different approach : we use a metamodel to represent update contexts. Context definition requires a set of declarations (metadata) which are relationships between objects (belonging to the application domain) and metadata types. The metamodel, the corresponding metadata and the relationships between data and metadata (declarations) are represented in the information base itself. We support a set of metadata types which allow fine grain, dynamic and flexible context definitions and can be used in declarations concerning individuals, attributes, tokens or classes.

The key point is that declarations are represented as Telos attributes belonging to a special class of attributes, *contextDeclaration*, defined for this purpose [2]. This results in improved metadata management (e.g. efficient context utilization during browsing) and data-metadata consistency. Storing metadata (declarations) as data also results in implementation benefits (usage of the existing mechanisms to represent/store/query/visualize metadata) and ease of use (`Manos` is not required to learn a query language). Furthermore, update context definitions leave the information model unchanged.

Returning to our example, the declaration of `Charoula` 's context looks like figure 6. Declaration `a1` defines that `Charoula` can update instances of the specialization hierarchy, while declaration `a2` represents the constraint concerning `born` attributes and `a3` represents the constraint concerning `Anna` . Thus, `Charoula` 's context maintains its integrity when `Manos` renames any attribute class (including `born`). Besides, `Manos` cannot delete attribute `born` , because that would violate a structural constraint of Telos (he should delete link `a2` first). Furthermore, if we filter out the attributes belonging to the special attribute class for declarations, we will get the original information model.

5 Update Contexts

The *role*, *task* and *focus* contexts are presented in detail in this section. The basic concepts underlying the update metamodel are shown in figure 7.

Users are classified in one or more groups. User groups are assigned Update Tasks (for short, Tasks) which are defined via Update Declarations (for short, Declarations) and are related to some Usage Info (user preferences, guidance comments and starting points).

5.1 Role Context

The users of the information base are represented in the information base itself as objects and are organized in groups. User groups correspond to *roles* and are related with update tasks. A user may belong to more than one group. We define as *role context* of a particular user, the set of roles that are assigned to that user. User groups are organized in a specialization

[2]SIS allows the definition of attribute classes which can be used (instantiated) from any object of the base.

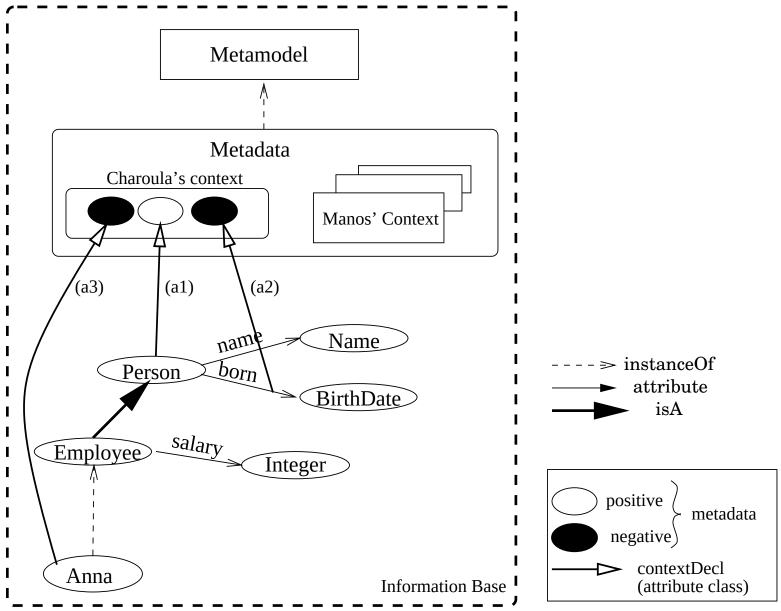


Figure 6: Context-driven update

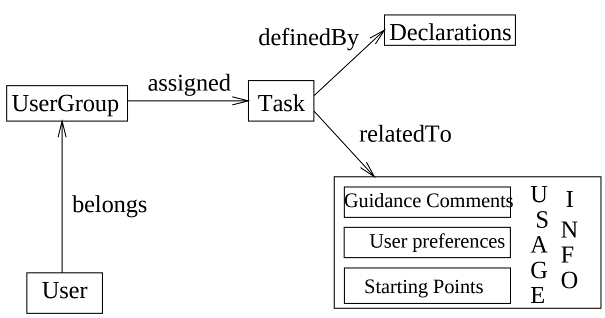


Figure 7: Basic concepts underlying the update metamodel

hierarchy, so we may have "subgroups". A subgroup inherits all the tasks assigned to its supergroups. An example is shown in figure 8.

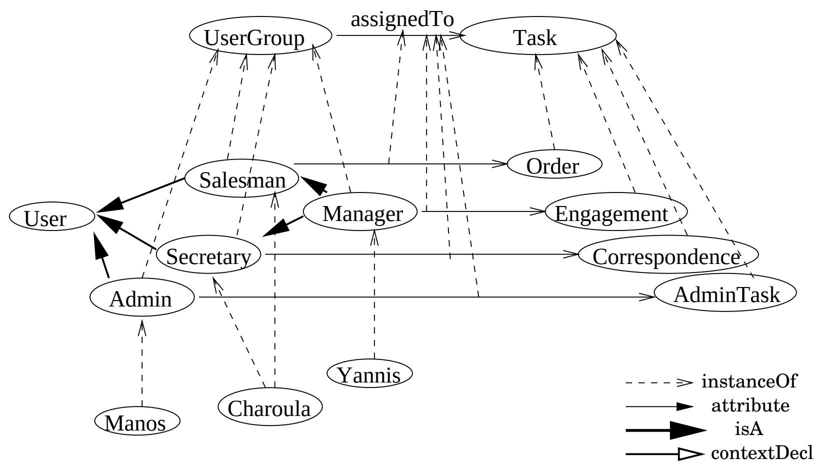


Figure 8: User context example

These declarations permit `Charoula`, the secretary, to work also as a salesman (in the afternoon), and `Yannis` can also replace his secretary or salesmen (on weekends).

5.2 Task Context

Update tasks are special objects stored in the base itself. These objects constitute the class *Tasks*.

Definition.

Each task is defined as a set of primitive update operations. This definition permits fine grain tasks which update data and/or schema. To define tasks we have to specify (via declarations) a function *contents* which returns the primitive update operations of each task :

$$contents : Tasks \rightarrow 2^U$$

To define a task t, we assign (via declarations which are presented next) elementary update action identifiers (EA) to objects. Let $Pred(t)$ be such an assignment ($Pred(t) \subset O \times EA$). This is construed as permissions regarding the execution of the participating pairs of objects and elementary actions. Obviously, $Pred(t)$ implicitly defines primitive update operations, since (as explained in section 3) each primitive update operation causes the execution of one or more elementary actions. Therefore, a primitive update operation u, is included in a task t, if the corresponding elementary update action identifiers ($ea(u)$) are members of $Pred(t)$. Thus, we define :

$$contents(t) = \{u \in U \mid ea(u) \subseteq Pred(t)\} \quad (1)$$

The multitude of elementary action identifiers is a measure of the level of detail of the possible task definitions. We could have used only two identifiers : one for schema update and one for data update.

Declarations.

Tasks are defined through declarations. We call $Decl$ the set of all possible declarations in a base. Each set of declarations is interpreted (mapped) to a subset of $O \times EA$ which defines primitive update operations.

Let I be the interpretation function $I : 2^{Decl} \rightarrow 2^{O \times EA}$ and $decl$ the function that for each $t \in Tasks$ returns the related declarations $decls : Tasks \rightarrow 2^{Decl}$. Therefore we can rewrite equation 1 as :

$$contents(t) = \{u \in U \mid ea(u) \subseteq I(decls(t))\}$$

$Decl$ consists of binary relationships between *objects* and *Declaration Types* (for short *Types*) :

$$Decl = \{(o, t) \mid o \in O, t \in Types\}$$

Types are 3-tuples, consisting of an EA, a *target identifier* and a *state identifier* :

$$Types = \{(id, targ, st) \mid id \in EA, targ \in Target, st \in State\}$$

State is the set {POS, NEG} and its members are used to distinguish positive (state=POS) from negative (state=NEG) types, hence positive from negative declarations. Using combinations of positive and negative declarations offers flexibility as illustrated in figure 9.

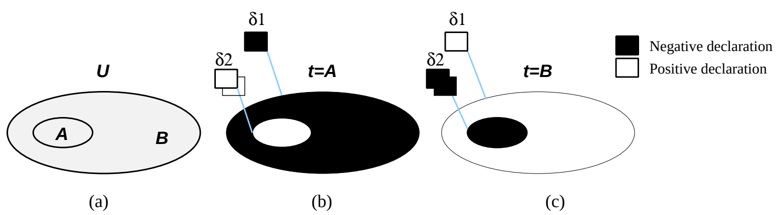


Figure 9: Combining positive and negative declarations
The sets A and B constitute a partition of U (a). The definition of a task containing only operations on set A (b) requires a negative declaration δ1 concerning the whole set U and some positive declarations δ2 concerning set A (they are exceptions to the declaration δ1). The opposite case is shown in (c).

Target is the set $\{onObj, onAttr, onInst\}$ and its members are used in order to characterize the declarations with respect to the declaration object. *Declaration object* is the object in reference to which the declaration is made. A declaration with target $onObj$ concerns the declaration object itself (see fig. 10(a)), with $onAttr$ it concerns all the attributes of the declaration object including inherited ones (see fig. 10(b)), and with $onInsts$ it concerns the instances of the declaration object (see fig. 10(c)).

Now, we present the *interpretation function*, $I : 2^{Decl} \rightarrow 2^{O \times EA}$, which determines the semantics of the declarations. Function I is the projection of a function I' which is a composition of other functions :

$$I' = I_{sys} \circ I_{sys_{onAttr}} \circ I_{onAttr} \circ I_{onInst} \circ I_{isa}$$

I_x are functions from 2^{Decl} to 2^{Decl}. If $A \subseteq Decl$ then

$$I'(A) = I_{sys}(I_{sys_{onAttr}}(I_{onAttr}(I_{onInst}(I_{isa}(A)))))$$

The composition order of functions I_x determines (implicitly) the prevalence among opposite declarations. For example, if $A \subseteq Decl$, then $I_x(A)$ does not contain any

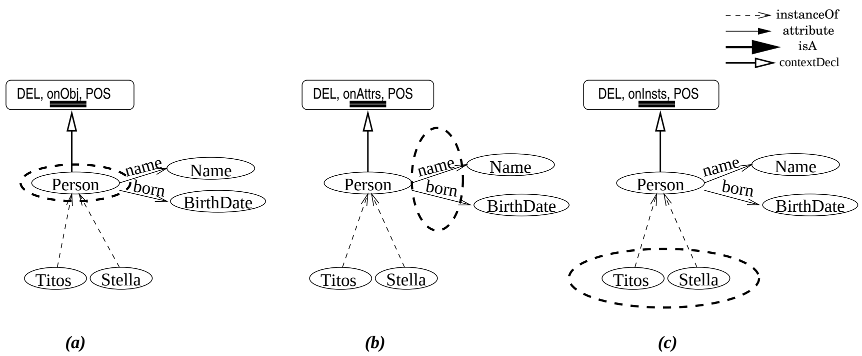


Figure 10: Interpretation of *Target* identifiers

declaration opposite to a declaration belonging to A. Actually, these functions combine positive and negative declarations, introduce inheritance rules and interpret target identifiers, in order to define finally a subset of $O \times EA$.

The declaration of a task, t, must be complete. Completeness is expressed by the following axiom :

$$\forall o \in O,\ id \in EA\ \exists d \in I'(decls(t)) : d = (o, (id, onObj, st))\ ,\ st\ \in State$$

Practically this axiom is satisfied by making a declaration for each $id \in EA$ and assign it to the system class *Object*. As this will be explained while describing the function I_{sys}, such a declaration concerns all the objects of the base, thus makes a task declaration complete.

Function I is a projection of I' :

$$if\ \ A \subseteq Decl\ then\ I(A) = \{(o, id) \mid\ (o,\ (id, onObj, \text{POS}))\ \in\ I'(A)\}$$

The description of functions I_x, follows :

• I_{isa}

If $A \subseteq Decl$, then $I_{isa}(A)$ includes besides A , declarations inherited through generalization (isA) relationships. The inheritance rules of Telos (and SIS) are modified in order to support negative, in addition to positive, information. The rules of $I_{isa}(A)$, illustrated in figure 11, are :

- A declaration is inherited to all subclasses (see nodes *h,i,j*).
- An explicit declaration is stronger than an inherited one (see nodes *a,c*).
- If an object inherits two opposite declarations (see *node j*), then the declaration of the more special class (*h* is a subclass of *g*) prevails. This rule is called *inferential distance ordering* [29, 24] and is used by systems which support positive and negative attributes.
- If an object inherits two opposite declarations and the previous rule does no apply (see node *e*), then the negative declaration prevails.

• I_{onInst}

Each declaration with target $onInst$ is analyzed (at run-time) in a set of declarations with target $onObj$, which concern the instances of the declaration object. If $A \subseteq Decl$, then $I_{onInst}(A)$ contains the declarations of A with target $\neq$ $onInst$, plus declarations which are deduced from the declarations of A with target $=$ $onInst$ and are not already in A. In case of conflicts, for example an object classified to two or more classes on which opposite declarations have been made, the negative one prevails.

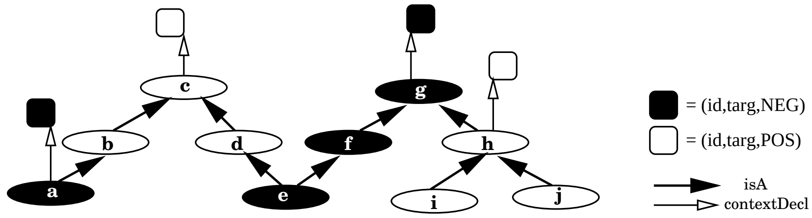


Figure 11: Inheritance of declarations
Black and white rectangles indicate two declarations with the same $id, target$, but with opposite $states$ (negative and positive, respectively). Classes have been colored according to the declaration they inherit.

- I_{onAttr}

Each declaration with target $onAttr$ is analyzed in a set of declarations with target $onObj$, which concern the attributes (including inherited ones) of the declaration object. If $A \subseteq Decl$, then $I_{onAttr}(A)$ contains the declarations of A except those with declaration object $\in (O - O_{sys})$ and target = $onAttr$, plus declarations which are deduced from the excepted declarations and are not already in A.

- $I_{sys_{onAttr}}$

Each declaration on a system class (O_{sys}) with target $onAttr$ is analyzed in a set of declarations with target $onObj$ which concern the attributes (including the inherited ones) of objects which belong to that system class. If $A \subseteq Decl$, then $I_{sys_{onAttr}}(A)$ contains the declarations of A except those on system classes with target = $onAttr$, plus declarations which are deduced from the excepted declarations and are not already in A.

These declarations offer coarse grain specifications which facilitate the definition of tasks. For example, in order to forbid the deletion of any attribute of a metaclass (regardless of the instantiation level of the attribute) we only have to make the declaration $(Individual\_M1Class, (DEL, onAttr, NEG))$.

- I_{sys}

Each declaration on a system class (O_{sys}) is analyzed in a set of declarations concerning the objects which belong to that system class. If $A \subseteq Decl$, then $I_{sys}(A)$ contains the declarations of A plus the ones deduced from declarations in system classes in A , which are not already in A.

These are also coarse grain declarations which contribute to task definition brevity. For example, to define a task which permits changes (e.g. deletion) of tokens only, we simply have to make two declarations :
$(Object, (DEL, onObj, NEG))$ and $(Token, (DEL, onObj, POS))$. The second declaration is an exception (concerning the Token level) of the first (which concerns all the objects of the base) [3].

Some indicative examples of the declarations usage are presented in figures 12, 13 and 14.

Composite declaration types: Since our types offer fine grain definitions, often more than one declarations are made in reference to the same object. In order to reduce the effort of the user, we support composite declaration types. A composite declaration type consists of a number of types with different pairs of elementary update action identifiers and target identifiers , and can be created by the user according to need. The combination of declarations with composite and simple types make the task declaration process efficient and effective. Some frequently used composite types are those which define: (i) specialization hierarchies with invariant

[3]This holds since SIS system classes are organized in a specialization hierarchy : e.g. *Token* isA *Object*.

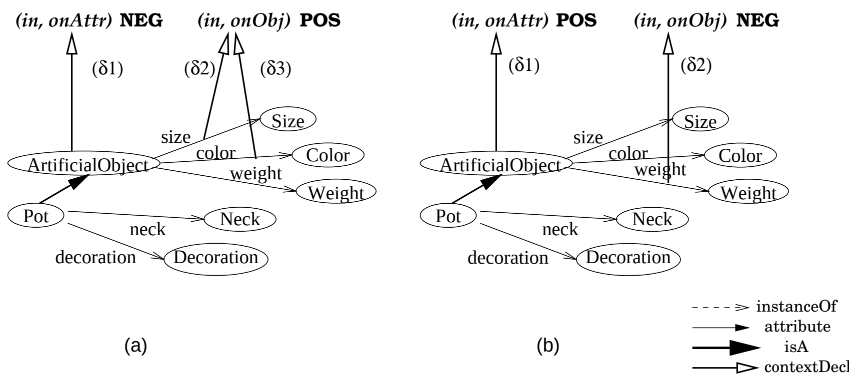


Figure 12: Declaring class projections

Identifier *in* stands for the identifiers $AddIn$ and $DelIn$. Declaration δ1, in (a), forbids the instantiation of all attributes classes of `ArtificialObject` . Due to inheritance, the same holds for the attributes of `Pot` . Declarations δ2,δ3 define the instantiation of classes `size` and `color` respectively. The latter prevail so the example defines the instantiation of a projection of the whole hierarchy, consisting of the attributes `size` and `color` . Figure (b) shows the opposite case. Here, declaration δ2 forbids the instantiation of class `weight` . Example (b) defines different projections from (a) because in (b) any new attribute, instead of (a), will be a member of the updatable projection.

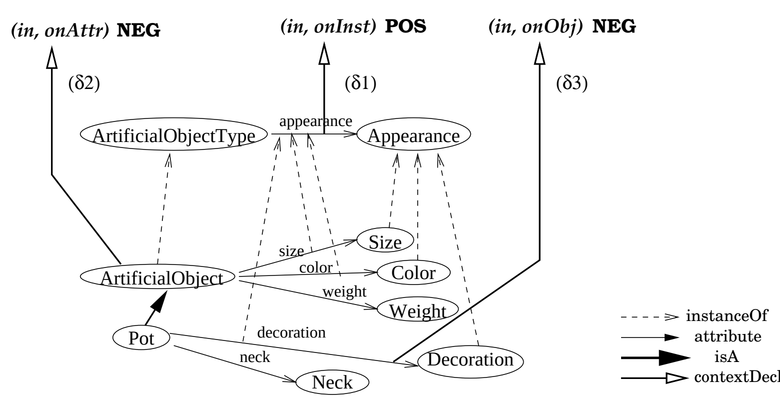


Figure 13: Declaring class projections via metalevel

Instantiation levels are exploited to provide efficient and dynamic definitions of updatable projections. Declaration δ1 concern the attributes `size`, `color` and `decoration`. This declaration prevails the negative declaration δ2, so any new attribute classified in the attribute metaclass `appearance` will be a member of the projection. Finally, declaration δ3 forbids the instantiation of class `decoration` (it prevails declaration δ1).

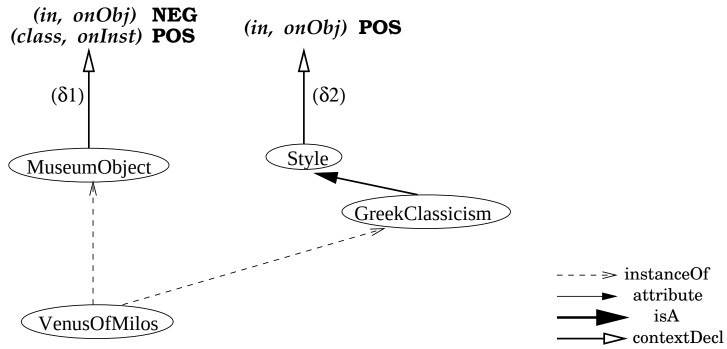


Figure 14: Declaration examples

Identifier *class* stands for the identifiers $AddClass$ and $DelClass$. Declaration δ1 forbids the instantiation of `MuseumObject` but also permits the classification of its instances. This means that for the instances of `MuseumObject` the deletion of the classification links to class `MuseumObject` is forbidden, but the classification to other classes, e.g. the subclasses of `Style` (see declaration δ2), is possible.

instances, (ii) specialization hierarchies used to classify objects, (iii) evolvable specialization hierarchies , (iv) specialization hierarchies whose population is used only as attribute values. An example is shown in figure 15.

Context Synthesis: Each task is related to a class whose extent are the declarations of that task. By organizing these classes in a specialization (isA) hierarchy we can reuse sets of declarations, obtaining fast task declarations.

Representation.

Each declaration relates an object with a *Type*, according to the model shown in figure 16. Although this is not the unique way to model the ternary relation $(object, task, type)$, the key point is that it is represented by links which are stored bidirectionally. This permits efficient deductions from any point : from *Object* (updatability checking), from *Task* (declaration analysis) or from *Type* (usage examples). The representation of declaration types is shown in figure 17. Finally figure 18 presents the whole metamodel.

5.3 Focus Context

The interactive user interface (UI) of an information management system should include concurrent browsing, presentation, modification and updating. Commonly, the development of such a UI is done using a toolkit, a graphical editor and a lot of programming effort. In order to reduce that cost, tools that map object structures to widget structures resulting in automatic UI construction, have been proposed and implemented (e.g. O_2Look [6]). In order to refine and customize (including information update restriction) these ready-made mappings, special tools have been developed (e.g. $ToonMaker$ [6]). Similarly, object display definition systems ([13]) and data-oriented UIMS have been proposed [15, 17].

We believe that, in addition, tools should take into account the role/task contexts in order both to control and guide/facilitate (customize) the interaction. Because the interaction is normally held on a per object basis, in this section we introduce the *Focus Context* as a

Controlled_Attribute_Values			
EA	*onObj*	*onAttr*	*onInst*
$AddIn, DelIn$	NEG	NEG	
$AddAT, DelAT$	POS		POS
$AddAF, DelAF$	NEG	NEG	NEG
$AddSub, DelSub$	NEG	NEG	NEG
$AddSup, DelSup$	NEG	NEG	NEG
$AddClass, DelClass$	NEG	NEG	NEG
REN	NEG	NEG	NEG
DEL	NEG	NEG	NEG

Figure 15: A composite declaration type

This type marks specialization hierarchies whose extent should remain invariant and are used as attribute values

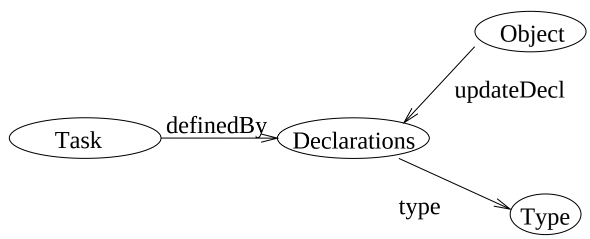


Figure 16: Declaration representation

concept which facilitates the interactive updates concerning one object, the focal point (or, simply, *focus*). The interactive mechanism we propose, which exploits the focus context, is described in section 6.

The focus context is implemented by a set of procedures which, assuming the *user* has selected a *task* and is focusing on one *object* , determine the set of candidate next primitive update operation. In order to compute that set, they take into account (i) the focus, (ii) the data model (and the corresponding modeling guidelines) and (iii) the current task [4]. Actually, they compute the set of all primitive update operations which (i) take the focus argument, (ii) are semantically correct [5] and (iii) belong to the contents of the current task.

[4]or more generally the role context

[5]Although the SIS executes only semantically correct updates, focus context procedures precompute the set of

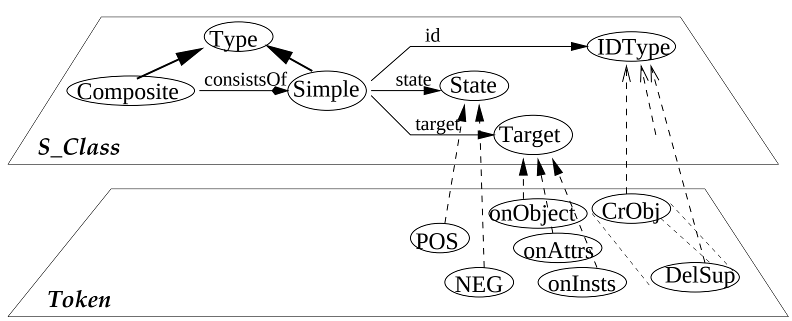


Figure 17: Representation of declaration types

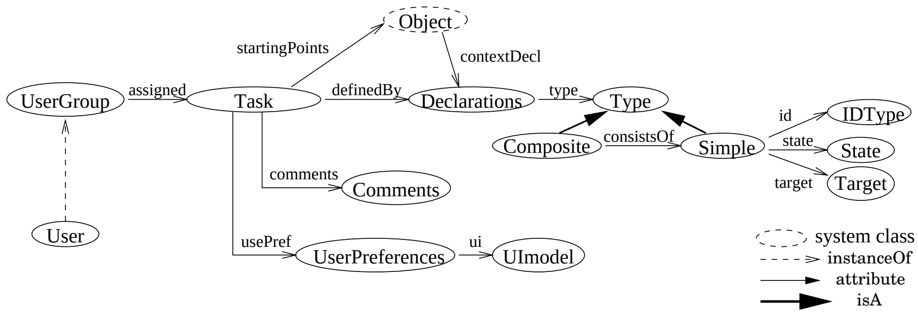


Figure 18: The task context metamodel

Assume that t is the current task, f is the focus, U is the set of all syntactically correct PUO, and $U_{sem}(f)$ is the set of semantically correct PUO which take f as argument. The set of candidate next update operations is :

$$CN = contents(t) \cap U_{sem}(f)$$

The data model of SIS-Telos and the representation of role/task contexts allows the efficient computation of the above set. When f is an attribute, CN is small, but when it is an individual, there are cases where it is too big to be practically useful. In such cases some common modeling guidelines based on the data model are used in order to restrict this set. Alternatively, the user can provide additional information for filtering this set.

An interesting extension of the focus context concerns the facilitation of creating composite objects or objects which satisfy some desired and predefined conditions (predicates). To face this need, we propose a high-level update operation, makecopy. *Makecopy* is able to produce copies of the focus (which can be a composite object), which users can subsequently differentiate [6]. This is a quite natural manner of evolution. It relies on detecting analogies between the intended new object (a mental conception) and an existing object (the focus). Parts of the structure and the contents of the existing object are preserved (reused) while the different elements are introduced, thus creating the new object. The rising problems concern the range of copying (an object may be connected with many others), and the copying process (is the value-object of a copied attribute the same as the original value, a new one automatically generated, or a user supplied value ?). To address the first problem we believe that *makecopy* should take into account the context of the current task and objects which can act as copy templates. To address the latter, makecopy should take into account the cardinality/dependency constraints of relationships (in order to face the problem concerning the attribute values) and could exploit the context-based naming mechanism (to name the automatically generated objects) which have been proposed for SIS-Telos by Theodorakis [27, 26]. An example is shown in figure 19.

Makecopy can also be used in order to facilitate the creation of objects which satisfy a desired condition. This can be achieved by copying an object which satisfies that condition.

We are currently working on all these issues concerning the focus context.

6 Implementation Aspects

Choosing a metamodel to represent role and task contexts, results in a number of implementation benefits: usage of existing mechanisms to represent, store, query and present the

all semantically correct updates in order to prevent the user from making wrong, hence rejected, requests.

[6] There are occasions where *makecopy* asks the user to make some decisions in order to drive the execution.

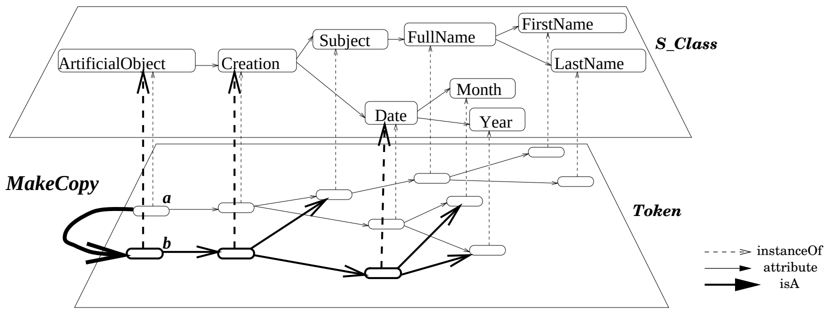
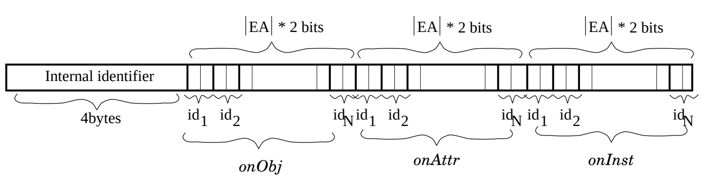


Figure 19: An example of *makecopy*

corresponding data.

As contexts are utilized at run-time, the speed of the corresponding deductions (regarding task contexts) is crucial [7]. The representation of declarations allows efficient deductions since SIS-Telos links are stored bidirectionally and SIS-Telos if very fast in link traversals. The interpretation algorithm comprises steps of determining explicit and inherited declarations. In terms of complexity the latter are the more significant. The average complexity of determining inherited declarations can be shown to depend on the average depth of generalization hierarchies and the average number of classes of an object. These average numbers are in practice bounded by small numbers, less than 10. Therefore, the average complexity of the interpretation algorithm is practically constant, independent of the size of the information base.

In order to optimize performance we propose the usage of a *cache* in order to reuse deductions (fig. 20 describes the cache elements). As the consistency of its contents is indispensable we must store deductions whose invalidation can be checked efficiently. Therefore we propose a cache to store the declarations inherited from superclasses. This reduces the cost of interpretation I_{isa}.



bits	meaning
00	positive
01	negative
10	none
11	invalid

Figure 20: Cache element structure

The interactive mechanism we propose which exploits role/task/focus contexts is described schematically in figure 21. A prototype which implements role/task contexts, enriched with some extra operations for context administration and supervision (user role tree, declarations tree, examples of type usage, composite type analysis), not shown in fig 21, has been implemented using the customizable user interface of SIS.

7 Related Work

Related work is found in many research areas including database views and authorization mechanisms. Below we draw comparisons to our work with regard to certain aspects, namely

[7]Speed requirements would be greater if trying to implement read contexts, since they would affect the query speed.

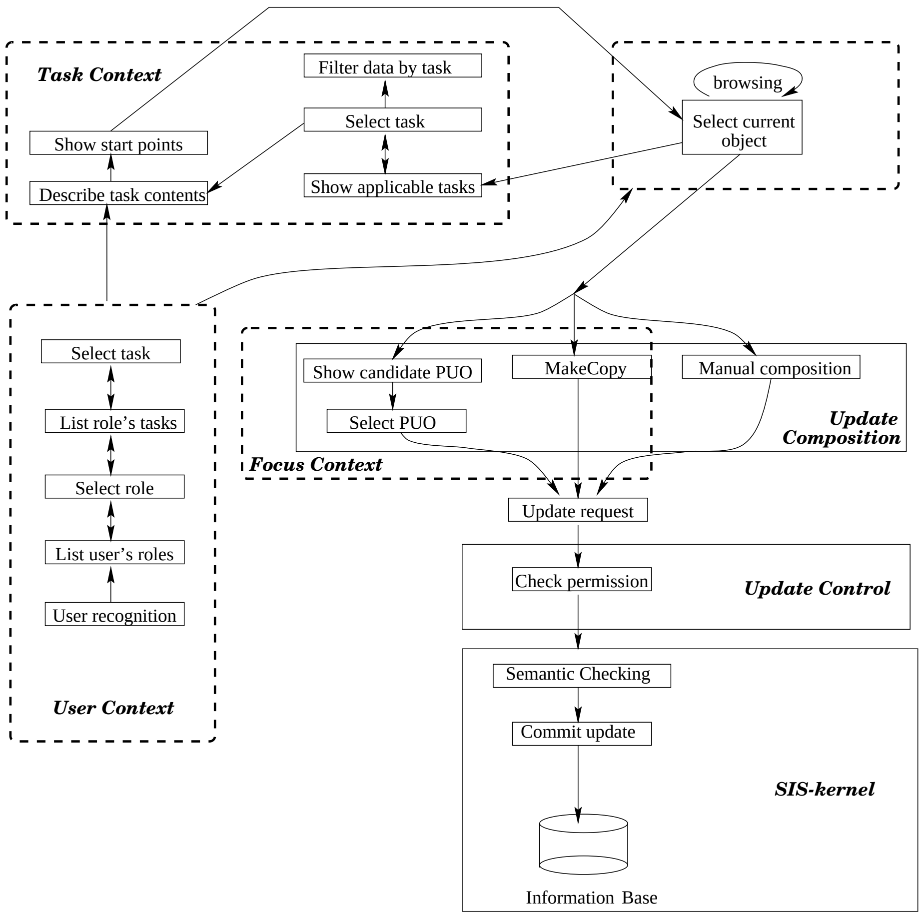


Figure 21: An interactive mechanism for context-driven update

expressive power, flexibility, representation, and utilization.

Expressive power

View mechanisms commonly define update contexts which concern only the data of the base [14, 23, 4]. They support dynamic definitions through the use of a query language.

We provide a uniform way to define tasks concerning data and/or schema. We support dynamic definitions, through (i) special dynamic declaration types, (ii) inheritance and (iii) instantiation levels (e.g. we exploit the attribute metaclasses to provide very efficient declaration of projection classes). We also support fine grain definitions and role/task context declaration reusability since one role or task context can be exploited for the declaration of other contexts.

Flexibility

To define a context, positive declarations are commonly used [25, 14]. Using combinations of positive and negative declarations offers flexibility and brevity in task declarations [22]. Combining positive and negative declarations may result in ambiguities which require time consuming search to detect [22].

We support combinations of positive and negative declarations which do not need checking since all ambiguities that can appear, are resolved. In order to speed up task declaration we support user-configurable composite declaration types which can be combined with simple types. This suites our needs better than organizing hierarchically the operation identifiers, as used by authorization mechanisms [22, 25]. Moreover, declarations do not pose any extra cognitive requirement to the administrator, since they are constructed using the existing structuring mechanisms.

Representation

Representing views as stored query strings results in contexts which are not easily supervised, maintained and utilized [14, 4]. On the other hand, embodying views in the schema results in schema fragmentation and ambiguities [23].

We use a metamodel to model context definitions (metadata). Work concerning the usage of metamodels can be found in [12, 18, 3]. Metadata are stored as attribute relationships between objects and metadata types. This enhances the concept of contexts because their declarations are subject to semantic checking, remain consistent with the base and can be utilized efficiently. In addition, they do not affect the information model.

Utilization

We relate each context with usage information (starting points, user preferences, comments) as proposed in [5]. In addition, we propose the focus context which exploits the role/task contexts and facilitates the interactive updates. It includes a high level update operation, *makecopy*, which helps creating composite objects. In [4], views which define virtual paths are proposed in order to face this process. Our proposal is simpler and requires less effort. Makecopy also facilitates the creation of objects which are supposed to satisfy predefined conditions (predicates). Some existing systems can specify such sets of objects (which satisfy a predicate) with the notion of selection view, but regarding the addition of new objects to these sets, they deal only with the dilemma of acceptance or rejection of the request for creating an object which does not satisfy the corresponding predicate (some permit it [23], some do not [14, 4]). They do not provide any user support.

8 Conclusion

In this paper we address the problem of information base update by introducing three kinds of contexts: *role*,*task* and *focus* context. *Role context* is used to relate users with update tasks, *task context* restricts the scope of updates to a subset of the information base, according to various criteria posed by actual tasks (filtering) or authorization constraints, and finally, *focus* *context* is used to guide the user at run-time, by exploiting the focus and the corresponding roles and tasks of the user.

We focus on issues regarding the enhancement of the notion of context in information bases. In particular, we offer a uniform way of defining tasks which concern the data or the schema, which also permits fine grain and dynamic definitions. We represent contexts in a way that ensures the consistency of its declarations with the contents of the information base, thus minimizing the maintenance cost. We believe that flexibility and brevity of context declarations are important qualities, therefore we support combinations of positive and negative declarations, as well as coarse grain declarations and declaration reuse. Our metamodel in conjunction with the adopted information representation framework offers efficient utilization of update contexts. Moreover, we make suggestions regarding further optimization. At last, we describe briefly the general interactive mechanism which exploits contexts, including a high level update operation, *makecopy*, which help users in creating objects which satisfy desired predicates and can be composite.

We are currently working on extensions concerning the expressive power of our metamodel, such as declarations inherited by attributes and the flexibility of task declarations (improved context synthesis). There are also open issues concerning the efficiency of task utilization, e.g. the implementation of declaration types in the kernel of our repository system, testing of cache, and focus context. In addition, for better context utilization, special user interface operations must be implemented.

Aknowledgements

We wish to thank the members of the Information Systems and Software Technology Group at ICS-FORTH for their explicit and implicit contribution to this work, in particular Dr. Martin Doerr, George Georgiannakis, Polivios Klimathianakis, Manos Theodorakis and Athena Trapsioti.

References

[1] Rakesh Agrawal and Linda G. DeMichiel. ''Type Derivation Using the Projection Operation''. *Information Systems*, 19(1):55--68, 1994.

[2] G. Attardi and M. Simi. ''Completeness and Consistency of OMEGA, A Logic for Knowledge Representation''. In *Proceedings of International Joint Conference on Artificial Intelligence*, Vancouver, 1991.

[3] Z. Bellahsene. ''An Active Meta-Model for Knowledge Evolution in an Object-oriented Database''. Technical report. LIRMM UMR CNRS/Montpellier II, 1992.

[4] Z. Bellahsene. ''The Point of View Notion for Defining and Updating Views in an Object-oriented Database''. Technical report. LIRMM UMR CNRS/Montpellier II, 1992.

[5] Philip A. Bernstein and Umeshwar Dayal. ''An Overview of Repository Technology''. In *Proceedings of the 20th VLDB Conference*, pages 705--713, Santiago, Chile, 1994.

[6] P. Borras, J.C. Mamou, D. Plateau, B. Poyet, and D. Tallot. ''Building User Interfaces for Database Applications : The O2 experience''. *SIGMOD RECORD*, 21(1):32--38, March 1992.

[7] M. Brodie, J. Mylopoulos, and J. Schmidt, editors. *``On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases and Programming Languages''*. Springer-Verlag, 1984.

[8] Panos Constantopoulos. ``Cultural Documentation: The CLIO System''. Technical Report 115, Institute of Computer Science Foundation for Research and Technology Hellas, January 1994.

[9] Panos Constantopoulos and Martin Doerr. ``Component Classification in the Software Information Base''. in O.Nierstrasz and D.Tsichritzis, eds.,Object-Oriented Software Composition, Prentice-Hall,1995.

[10] Panos Constantopoulos and Martin Doerr. ``The Semantic Index System : A brief presentation''. Institute of Computer Science Foundation for Research and Technology Hellas, May 1994. (http://www.ics.forth.gr/proj/isst/Systems/sis/).

[11] Panos Constantopoulos, Manos Theodorakis, and Yannis Tzitzikas. ``Developing Hypermedia Over an Information Repository ''. to appear in the 2nd WorkShop on Open Hypermedia Systems at Hypertext'96, Washington, DC, USA,March 16-20, 1996.

[12] Oscar Diaz and Norman W. Paton. ``Extending ODBMS Using Metaclasses''. *IEEE Software*, pages 40--47, May 1994.

[13] Belinda B. Flynn and David Maier. ``Supporting Display Generation for Complex Database Objects''. *SIGMOD RECORD*, 21(1):18--24, March 1992.

[14] Andreas Geppert, Stefan Scherrer, and Klaus R. Dittrich. ``Derived Types and Subschemas : Towards Better Support for Logical Data Independence in Object-Oriented Data Models''. Technical Report 93.27, Institut für Informatik, Universität Zürich, June 1993.

[15] S. Hudson and R. King. ``Semantic Feedback in the Higgens UIMS''. *IEEE Transactions of Sortware Engineering*, August 1988.

[16] Richard Hull and Roger King. ``Semantic Database Modeling''. *ACM Computing Surveys*, 19(3):202--260, September 1987.

[17] Michel Kuntz. ``The Gist of GIUKU. Graphical Interactive Intelligent Utilities for Knowledgeable Users of Data Base Systems''. *SIGMOD RECORD*, 21(1):58--64, March 1992.

[18] Leo Mark and Nick Roussopoulos. ``Metadata Management''. *IEEE Computer*, pages 26--36, December 1986.

[19] M. Missikoff and M. Scholl. ``An Algorithm for Insertion into a Lattice: Application to Type Classification''. In *Proceedings 3nd International Conference on Foundations of Data Organisation and Algorithms - FODO*, pages 64--82, Paris, June 1989. Springer-Verlag.

[20] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. ``Telos : Representing Knowledge about Information Systems''. *ACM Transactions on Information Systems*, 8(4), October 1990.

[21] John Mylopoulos and Renate Motschnig-Pitric. ``Partitioning Information Bases with Contexts''. In *Proceedings of Conference on Cooperative Information Systems, CoopIS-95*, pages 44--54, Vienna, Austria, May 1995.

[22] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrell Woelk. ‘‘A Model of Authorization for Next-Generation Database Systems”. *ACM Transactions on Object Oriented Database Systems-TODS*, 16(1), March 1991.

[23] Marc H. Scholl, Cristian Laasch, and Markus Tresch. ‘‘Updatable Views in Object-Oriented Databases”. In *Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases*, pages 189--207, 1991.

[24] Lokendra Shastri. ‘‘Default Reasoning in Semantic Networks: A Formalization of Recognition and Inheritance ”. *Artificial Intelligence*, 39:283--355, 1989.

[25] Gerhard Steinke. *‘‘Task-Based Security for Knowledge Base Systems”*. PhD thesis, University of Passau, July 1992.

[26] Manos Theodorakis. ‘‘Context-based Naming in Semantic Networks”. to appear in the 3rd Doctoral Consortium on Advanced Information Systems Engineering, Heraklion, Crete, Greece, May 20-21, 1996.

[27] Manos Theodorakis. ‘‘Name Scope in Semantic Data Models”. Master’s thesis, Department of Computer Science - University of Crete, September 1995. (in Greek).

[28] M.B. Thuraisingham. ‘‘Mandatory Security in Object-Oriented Database Systems”. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications - OOPSLA*, pages 203--210, October 1989.

[29] David Touretzky, John Horty, and Richmond Thomason. ‘‘A Clash of Intuitions : The Current State of Nonmonotonic Multiple Inheritance Systems”. In *Proceedings of the 10th IJCAI, Milan, Italy*, pages 476--482, 1989.