DBAI-TR-2012-76*
UMAP: A Universal Layer for Schema Mapping Languages

Florin Chertes and Ingo Feinerer

Technische Universität Wien, Vienna, Austria
Institut für Informationssysteme
FlorinChertes@acm.org Ingo.Feinerer@tuwien.ac.at

Abstract. Schema mappings are fundamental notions in data exchange and integration for relating source and target schemas. Visual mapping languages provide graphical means to visually describe such transformations. There is a plethora of tools and languages available however all use different notions and visualizations and are hardly extensible.
In this paper we propose a new universal layer for schema mapping languages which provides a unified abstraction and middleware for high-level visual mapping languages. We use only standardized Uml and Ocl artifacts which allow for easy code generation in a number of target languages (e.g. Xml, Java, or C++ code) and for a modular extension mechanism to support complex schema mappings (like second-order dependencies). We illustrate our layer by translating key elements of Clip, a recent visual mapping language more expressive than the well-known Ibm Clio mapping tool.

1 Introduction

Schema mappings are central notions both in data exchange and data integration. They provide a precise formalism for modeling and describing the process of transforming source instances to target instances of a database in an information exchange scenario. The most common formalism for expressing schema mappings are logical formulae, typically in first-order or second-order logic [6]. The use of logics allows for exact definitions of the syntax and semantics of schema mappings contributing to the success of data exchange in theoretical research during the last decade [8].

Similarly, schema mappings have been of tremendous importance in industrial data exchange applications, e.g., in the well-known Ibm Clio mapping tool [12, 7]. However, in an industrial context visual languages for modeling schema mappings have gained increasing importance over the last years. Visual languages hide logical formalisms behind graphical notations and allow users without deep technical and mathematical background to perform data exchange.

* Version 2

One of the most influential approaches along this line is Clip [13], a visual language for explicit schema mappings. Clip defines a set of custom language elements modeling source-to-target and hierarchical schema mappings. Clip and similar visual language formalism have their merits for high-level modeling by providing appropriate visual elements for the most common mappings.

Nonetheless, we observe a number of drawbacks. First, there is no unified formalism nor standard for the actual elements of such a visual mapping language: supported elements depend on the concrete schema mappings supported by the tool, and each visual language uses different visualizations for its elements. Second, when automatically generating code from schema mappings, various tools (IBM Clio, Altova MapForce, Stylus Studio, etc.) differ significantly in the number of target languages and the concrete implementation of the rules. Finally, there is a lack of easy extension mechanisms that allow the user to model additional types of schema mappings, e.g., for second-order dependencies, or mappings in the non-relational case. Consequently, these challenging tasks need to be addressed to foster the applicability of visual languages for schema mapping design in industry.

In this paper we propose a new unifying layer for visual schema mapping languages based on standardized UML class diagrams [10] and OCL constraints [11]. This layer is intended as a middleware underlying high-level visual languages like Clip or schema mapping toolkits like Clio but can also be used directly to visually design, model, and maintain schema mappings. By using only standardized and well-understood artifacts (basic features of UML class diagrams and selected OCL constraints) from the UML modeling language we obtain a precise syntax and semantics for our layer which can be translated back to logics [3, 2]. Most existing UML toolkits support the generation of code from class diagrams which we use for implementing our schema mappings in various target languages. Finally, our approach is modular and allows easy extension of new schema mappings and targets for code generation.

Our main contributions are:

- A theoretical presentation of our layer underlined by a translation of the core Clip language elements to our UML-based formalism, demonstrating the translation of source-to-target mappings to UML class diagrams augmented with OCL-constraints. These constraints correspond to Select-Project-Join constructs of SQL and can thus be efficiently implemented.
- The handling of more complex transformations like joins with grouping in the context of nested schema mappings for tree-like data structures (e.g., necessary for XML data sources) in our proposed formalism. These transformations are characterized by more involved restructuring operations to map the source schema to the target schema.
- A proposal on modeling advanced schema mappings like second-order dependencies in our framework, emphasizing its modular extension mechanism.

The paper is structured as follows. Section 2 defines basic notions and preliminaries. Section 3 presents the UMAP layer and a translation of basic core features of Clip to UMAP with a special focus on source-to-target dependencies.

Section 4 covers nested mappings with groups and joins whereas Section 5 deals with advanced topics like second-order dependencies and target dependencies. Section 6 concludes.

2 Preliminaries

Schemas and instances. A *schema* $\mathbf{R} = \{R_1, \ldots, R_n\}$ is a set of relation symbols R_i each of a fixed arity. An *instance* I over a schema $\mathbf{R}$ consists of a relation for each relation symbol in $\mathbf{R}$, s.t. both have the same arity. For a relation symbol R, we write R^I to denote the relation of R in I. We only consider finite instances here.

Schema mappings. Let $\mathbf{S} = \{S_1, \ldots, S_n\}$ and $\mathbf{T} = \{T_1, \ldots, T_m\}$ be schemas with no relation symbols in common. A *schema mapping* is given by a triple $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ where $\mathbf{S}$ is the source schema, $\mathbf{T}$ is the target schema, and Σ is a set of logical formulae called dependencies expressing the relationship between $\mathbf{S}$ and $\mathbf{T}$.

Instances over $\mathbf{S}$ (resp. $\mathbf{T}$) are called *source* (resp. *target*) *instances*. We write $\langle \mathbf{S}, \mathbf{T} \rangle$ to denote the schema $\{S_1, \ldots, S_n, T_1, \ldots, T_m\}$. If I is a source instance and J a target instance, then $\langle I, J \rangle$ is an instance of the schema $\langle \mathbf{S}, \mathbf{T} \rangle$.

Given a (ground) source instance I, a target instance J is called a *solution for I under $\mathcal{M}$* if $\langle I, J \rangle \models \Sigma$. The set of all solutions for I under $\mathcal{M}$ is denoted by $Sol(I, \mathcal{M})$.

Dependencies. *Source-to-target tuple generating dependencies (s-t tgd)* are logical formulae of the form $\forall \boldsymbol{x}(\phi(\boldsymbol{x}) \rightarrow \exists \boldsymbol{y} \psi(\boldsymbol{x}, \boldsymbol{y}))$. We write $\boldsymbol{x}$ for a tuple $(x_1, \ldots, x_n)$. However, by slight abuse of notation, we also refer to the set $\{x_1, \ldots, x_n\}$ as $\boldsymbol{x}$. Hence, we may use expressions like $x_i \in \boldsymbol{x}$ or $\boldsymbol{x} \subseteq X$, etc.

Equality-generating dependencies (egds) are of the form $\forall \boldsymbol{x}\,(\phi(\boldsymbol{x}\,) \rightarrow x_i = x_j)$ with $x_i, x_j \in \boldsymbol{x}$.

A *second-order tuple generating dependency (*SO *tgd)* is a logical formula of the form $\exists \boldsymbol{f}((\forall \boldsymbol{x}_1(\phi_1 \rightarrow \psi_1)) \wedge \cdots \wedge (\forall \boldsymbol{x}_n(\phi_n \rightarrow \psi_n)))$ where (1) each member of $\boldsymbol{f}$ is a function symbol, (2) each ϕ_i is a conjunction of atomic formulas of the form $S(y_1, \ldots, y_k)$ (with $S \in \mathbf{S}$ and $y_j \in \boldsymbol{x}_i$), and equalities of the form $t = t'$ (with t and t' terms based on $\boldsymbol{x}_i$ and $\boldsymbol{f}$), (3) each ψ_i is a conjunction of atomic formulas of the form $T(t_1, \ldots, t_\ell)$ (with $T \in \mathbf{T}$ where $t_1, \ldots, t_\ell$ are terms based on $\boldsymbol{x}_i$ and $\boldsymbol{f}$), and (4) each variable in $\boldsymbol{x}_i$ appears in some atomic formula of ϕ_i.

Clip. Clip is a mapping language for relational and XML schemas. Schema elements are visually connected from source to target by lines interpreted as mappings. Both structural mappings and simple value mappings are supported. The combination of value and structural mappings in Clip yields expressive language elements extending those from Clio [12, 7], one of the most prominent schema mapping tools developed by IBM Almaden Research Center and the University of Toronto, and gives users fine-grained control over generated transformations. Mappings are compiled into queries that transform the source instances into target instances. The main Clip language elements [13, Fig. 2] are as follows.

- *Value nodes* represent attributes and text. For example *value: int*, *value: String* or *@pid: int* in Fig. 3.
- *Single elements* consist of a *value node* and have a name. Examples are *pname*, *ename* and *sal* in Fig. 3.
- *Multiple elements* represent sets of elements. E.g. *Proj[0..*]* from Fig. 3.
- *Value mappings* are thin arrows with open ends. They are used to map values from source to target. E.g. in Fig. 3 the arrow connecting *ename* with *name*.
- *Builders or object mappings* are thick arrows with closed ends connecting elements. Examples are the four bold arrows in Fig. 3.
- *Build nodes* have at least one incoming *builder* and at most one outgoing *builder* and express a filtering condition in terms of the variables in the *builders* or enforce a hierarchy of *builders* if connected by *context arcs*.
- *Grouping nodes* are a special kind of *build nodes* used for grouping on attributes. Fig. 9 gives an example.
- *Context propagation trees* are trees with *build nodes* and *context arcs*. E.g. the arrow connecting the two *build nodes* in Fig. 3.

3 UMAP Layer and Translation of Clip Core Features

The UMAP layer abstracts source and target schemas as UML class diagrams. We assume both source and target as XML schemas (since relational schemas can be converted into XML schemas). The schemas represent trees consisting of nodes, attributes and sets of nodes. Individual schema elements, i.e. nodes, are modeled as classes, and attributes in the XML schema become attributes in the corresponding UML class. Sets of elements are modeled as generic container classes encapsulating the underlying class. The actual mappings between source and target schemas are done by associations augmented by association classes. We use OCL to specify constraints (post conditions and invariants) on the associations, association classes, and methods in association classes, to ensure the desired semantics of the mapping.

It follows a detailed presentation of the UMAP layer exemplified by a translation of the core features of the Clip language. We map each Clip artifact to a UML/OCL artifact of the UMAP layer. The Clip *value nodes* and *single elements* are translated by class attributes grouped semantically in a class. The Clip *multiple elements* are modeled in UML as generic container classes (sets of elements). The *value mappings* are modeled in UML with the help of an association class linking source to target. A class function named *map* implements the mapping. The definition of the mappings is achieved through OCL expressions which include also the filtering conditions. The *builders* or *object mappings* are modeled in UML also with the help of an association class linking source to target. A class function named *build* implements the iterator defined by OCL expressions. The associations between association classes model the hierarchy of *builders*. The *context propagation tree* is achieved with the help of the hierarchies of association classes and associations between them. Each iterator modeled by a class function named *build* from one level of the hierarchy triggers a class

function named *map* from a lower level in this hierarchy which maps source to target values. The class function named *map* from that level triggers one or more class functions named *build* from a lower level of the induced hierarchy of functions. As a general characteristic of the translation from Clip to Uml the translations of the Clip *value mappings* and *builders* are associations classes using functions named *map* or *build*. Successive alternations of these two functions correspond to the Clip feature of a *context propagation tree*. Joins and *grouping nodes* are modeled with the help of the Ocl expressions defining class functions and attributes.

A simple mapping. A simple Clip mapping is presented in Fig. 1 (adapted

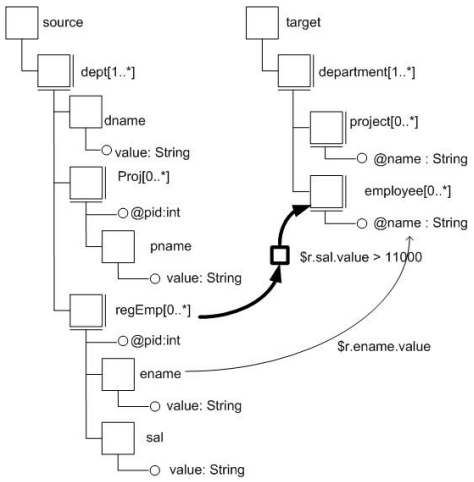

Fig. 1. A simple Clip mapping (adapted from [13, Fig. 3])

from [13, Fig. 3]): an *employee* is created for each *regEmp* whose salary is greater than *11000*. For each employee the name is also copied from source to target. In [13] is explained that this mapping is expressible in Clio. The authors mention further that this mapping is under specified: there is no indication how to map the *dept* from the source to *department* in the target. Using the notion of *universal solution* [6] the authors explain that there are at least two such solutions: an universal solution with a generic *department* for each mapped *employee* or an universal solution with a single generic *department* for all mapped *employees*. By adopting the principle of *minimum-cadinality*, the authors prefer the later solution. For clarity we present the mapping of the source set *regEmp[0..*]* to the target set *employee[0..*]* without the source *dept* and the target *department* to which they belong. Thus we represent only the essential part of Fig. 3 from [13] using a class diagram in Fig. 2.

The Uml structure. The Uml class diagram presents the structure and the Ocl expressions define the operations used to map the source to the target.

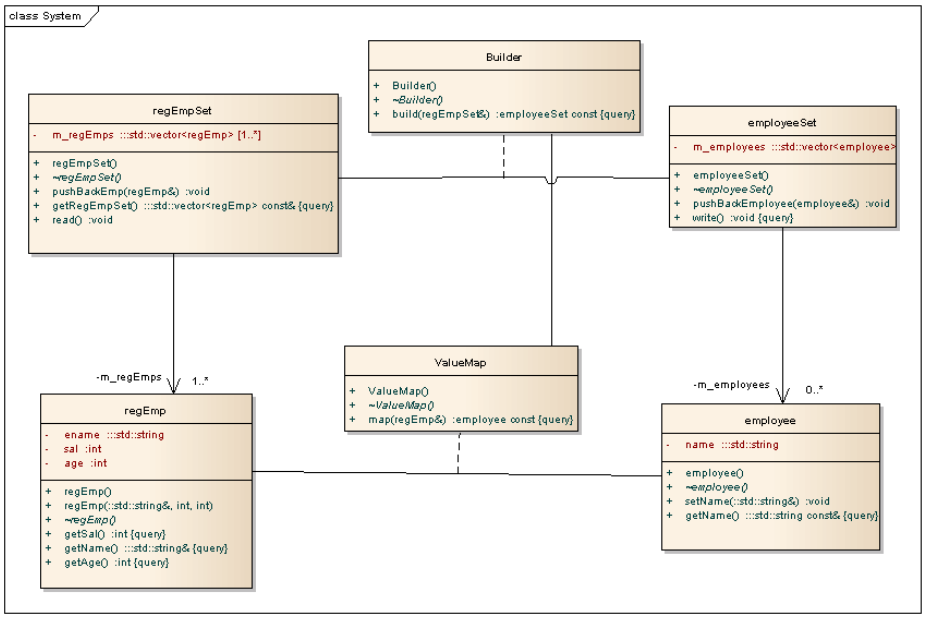

Fig. 2. The class diagram with a simple mapping corresponding to Fig. 1

There are two classes on the source side: a class of type *regEmpSet* and a class of type *regEmp* connected with the previous by aggregation with cardinality *1..**. The class *regEmp* contains two attributes: *ename* of type *string* and *sal* of type *int*. On the target side there are two classes: *employeeSet* and *employee* connected by aggregation with the same cardinality as the previous aggregation from the source side. The class *regEmpSet* from the source is connected to the class *employeeSet* from the target by an *association class*: *Builder*. In the same way the class *regEmp* from the source is connected to the class *employee* from the target by an *association class*: *ValueMap*. Between these two association classes there is an *association* which helps the class *Builder* to access the functionality of the class *ValueMap*. The *Builder* association class iterates through the source set using the function *build*. In each iteration by the help of the association class *ValueMap* each *regEmp* is mapped to a *employee* using the function *map*. These both functions, *Builder::build* and *ValueMap::map* are defined by OCL post-condition expressions.

Translating *value nodes*, *single elements* and *multiple elements*. The previously named classes translate the Clip structure to UML. Both *Set* classes: *regEmpSet* and *employeeSet*, represent the *multiple nodes* in Clip language: *regEmp[0..*]* and *employee [0..*]*. The other two classes *regEmp* and *employee* put together all *value nodes* and *single elements* that structurally belong to the *multiple elements* such as *regEmp[0..*]* and *employee [0..*]*.

Translating *value mappings* and *builders*. The semantic of Clip *value mappings* and *builders* is achieved in the UML translation through artifacts of the class diagram and the OCL expressions. We use in the class diagram the association class *ValueMap* that connects the source type *regEmp* to the target type *employee*. In the UML translation of Clip *value mapping*, the generation of the target object from the source object is done by the help of the function *ValueMap::map*. This mapping function is defined in OCL as follows:

```
context ValueMap::map(rEmp: regEmp): employee
post: result = e: employee and e.name = rEmp.ename
```

In OCL [11, Section 7.6.2] it is mentioned that an operation could be defined by a postcondition. The object that is returned by the operation can be referred to by the keyword `result`. In our case the target to source mapping is defined by the equality of the names. Other mapped attributes could be here similarly detailed if necessary.

In the UML translation of Clip *builder*, the generation of the target set from the source set is achieved by the use of the function *Builder::build*. This function iterates over the set *regEmpSet* generating the set *employeeSet* and by this models the Clip *builder*. This mapping function is defined in OCL as follows:

```
context Builder::build(rEmpSet: regEmpSet): employeeSet
post: result = rEmpSet.m_regEmps->select(r | r.getSal() > 11000)
    ->collect(r: regEmp | ValueMap.map(r): emplyee)
```

The mapping definition starts from the source set *regEmpSet* and selects only those objects from the source that have a salary greater as 11000 creating a set. In the next step we obtain another set of type *employeeSet* from this set. This is done by the use of the function *collect* that applies to each object of type *regEmp* the function *ValueMap::map*. The result is an object of type *employee*. Further the function *collect* inserts all this newly created objects in a set which is the return value of the function *Builder::build*. The class *Builder* is the translation of the Clip *builder* because it iterates on the source set, it selects the nodes to be mapped to the target by the help of the OCL *select* function and then it creates a totally different set using the *collect* function. The function *collect* uses the association to the class *ValueMap* to effectively map each object from the source to the target. The class *ValueMap* translates the Clip *value mappings*.

Context propagation. Consider the Clip mapping with context propagation shown in Fig. 3. For each *dept* from the source a *department* in the target is created and for each *regEmp* of a *dept* an *employee* of a *department*. The mapped *regEmps* are only those with a salary greater than 11000. The mapping is performed with the help of two *builders*, each with a *build node* and a *context arc* connecting the *build nodes*. The *builder* started from *dept* acts as an outer iterator on the *builder* started from *regEmp*, an inner iterator. This has the effect that all *regEmps* of each distinguished *dept* from the source are mapped as the *employees* of the corresponding *department* in the target. If the *context arc* is omitted in Clip then all the *employees* are connected to each of the *departments*.

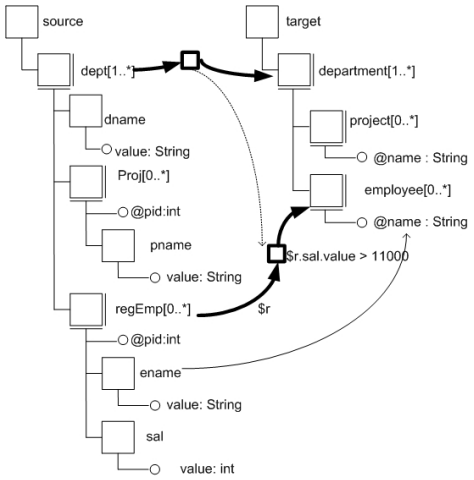

Fig. 3. A Clip mapping with context propagation (adapted from [13, Fig. 4])

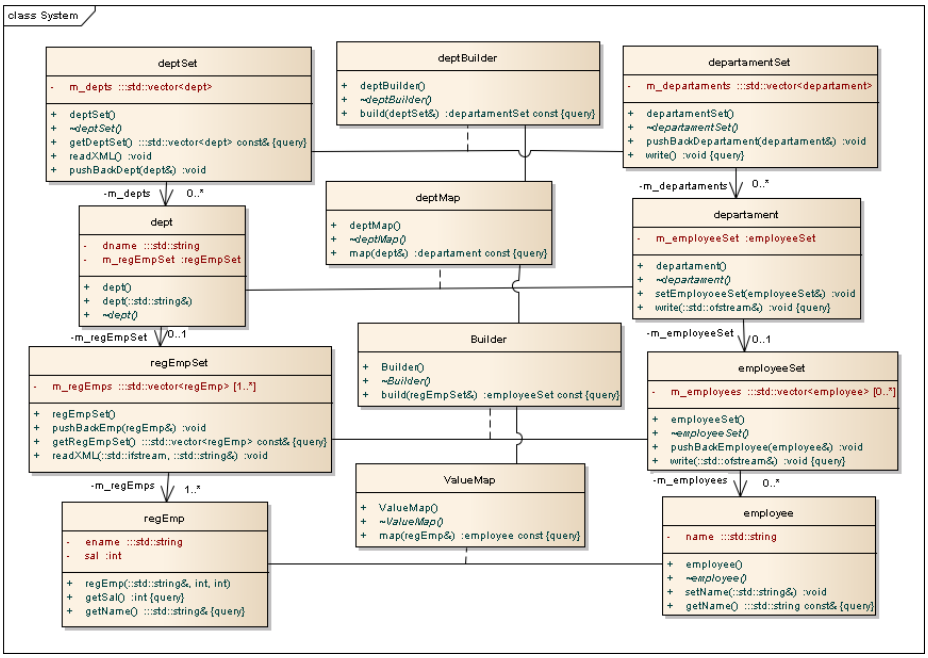

Fig. 4. The class diagram with context propagation corresponding to Fig. 3

The Uml structure. The Uml class diagram in Fig. 4 presents the structure and the Ocl expressions define the operations used to map the source to the target. Supplementary to the previous case, in which only employees were mapped from source to target, in this case departments with employees are mapped. The mapping of the employees was already presented above. The classes and the associations between them are reused and we repeat only the essential facts. The employee is represented on the source side by the help of two classes: a class of type *regEmpSet* and a class of type *regEmp* connected with the previous by aggregation with cardinality 1..*. On the target side there are two classes: *employeeSet* and *employee* connected by aggregation with the same cardinality as the previous aggregation from the source side. The class *regEmpSet* from the source is connected to the class *employeeSet* from the target by an association class: *Builder*. In the same way the class *regEmp* from the source is connected to the class *employee* from the target by an association class: *ValueMap*. Between these two association classes there is an association which helps the class *Builder* to access the functionality of the class *ValueMap*. The *Builder* association class iterates through the source set using the function *build*. In each iteration, the association class *ValueMap* maps each *regEmp* to an *employee* using the function *map*. Both functions *Builder::build* and *ValueMap::map* are defined by Ocl post-condition expressions.

The association class *ValueMap* connects the source type *regEmp* to the target type *employee*. The mapping function is defined in Ocl as follows:

```
context ValueMap::map(rEmp: regEmp): employee
post: result = e: employee and e.name = rEmp.ename
```

The generation of the target set from the source set is achieved by the use of the function *Builder::build*. This function iterates over the set *regEmpSet* generating the set *employeeSet*:

```
context Builder::build(rEmpSet: regEmpSet): employeeSet
post: result = rEmpSet.m_regEmps->select(r | r.getSal() > 11000)
    ->collect(r: regEmp | ValueMap.map(r): employee)
```

The mapping definition starts from the source set *regEmpSet* and selects only those objects from the source that have a salary greater than 11000 creating a set. In the next step we obtain another set of type *employeeSet* from this set. This is done by the use of the function *collect* that applies to each object of type *regEmp* the function *ValueMap::map*. The result is an object of type *employee*. Further the function *collect* inserts all created objects in a set which is the return value of the function *Builder::build*.

The department is represented on the source side by the help of two classes: a class of type *deptSet* and a class of type *dept* connected with the previous by aggregation with cardinality 1..*. On the target side there are two classes: *departmentSet* and *department* connected by aggregation with the same cardinality as the previous aggregation from the source side. The class *deptSet* from the source is connected to the class *departmentSet* from the target by an association class: *deptBuilder*. In the same way the class *dept* from the source

is connected to the class *department* from the target by an association class: *deptMap*. Between these two association classes there is an association which helps the class *deptBuilder* to access the functionality of the class *deptMap*. The *deptBuilder* association class iterates through the source set using the function *deptBuilder::build*. In each iteration, the association class *deptMap* maps each *dept* to an *department* using the function *deptMap::map*. Both functions *deptBuilder::build* and *deptMap::map* are defined by OCL post-condition expressions.

```
context deptMap::map(dep: dept): department
post: result = d: department and
d.m_employeeSet = Builder.build(dep.m_regEmpSet)
```

The input object of this operation is of type *dept* and the object that results is of type *department*. The input object includes a set of source type *regEmpSet* which is mapped to a set of target type *employeeSet*. This is done by the function *Builder::build*, already presented. If needed, other target attributes, based on source attributes, could be defined here. The translation of Clip *builder*, generating a *department* set from a *dept* set is done by the function *deptBuilder::build*. This function iterates over the source set generating the target set and so translating the Clip *builder*:

```
context deptBuilder::build(dSet: deptSet): departmentSet
post: result = dSet.m_depts->asSet()
    ->collect(r: dept | deptMap.map(r): department)
```

The postcondition starts from the source set *deptSet* and could select only those departments fulfilling some conditions. In this case there are no conditions so all the departments are selected. The next step is to obtain from this set another set of type *departmentSet*. This is done by the function *collect* that applies to each object of type *dept* the function *deptMap::map*. The result is an object of type *department*. Further the function *collect* inserts all this newly created objects in a set which is the return value of this function. The class *deptBuilder* is the translation of the Clip *builder* because it iterates on the source set, it selects the nodes to be mapped to the target by the help of the OCL *select* function and then it creates a totally different set by the use of *collect* function.

A more complex mapping. A more complex Clip mapping is presented in Fig. 5. The hierarchy of builders, i.e. builders connected by context arcs, enforces the propagation of the outer iterator context to the inner iterators on *Proj* and *regEmp*. The authors explain in [13] that Clip can achieve, through this configuration, the mapping of *depts* with *Projs* and *regEmps* from the source to the target without loss of the structure what no other state-of-the-art-tools like Clio can do. The main idea of translating from Clip into UML, developed in the previous case, is used and the results are presented in the class diagram Fig. 6. The class diagram has on the source side six classes: *dept*, *Proj* and *regEmp* and the set variant of each. The association class *deptBuilder* using the other association class *deptMap* triggers the two inner iterators of the association classes *Builder* and *projBuilder*. At each step in the outer iteration the function *deptMap::map*

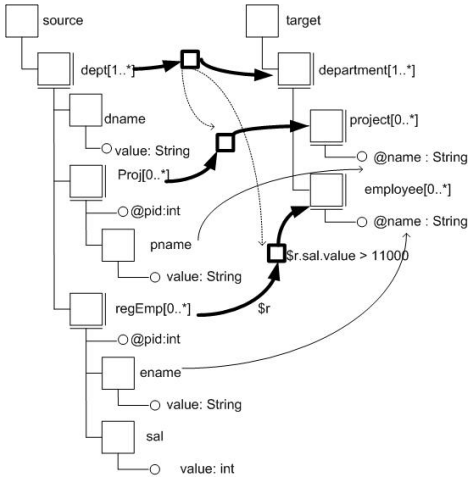

Fig. 5. A more complex Clip mapping (adapted from [13, Fig. 5])

is used and the inner iterations transform all the qualified *regEmp* and *Proj* objects from the source to *employee* and *project* objects of the target types and then the outer iteration attached them to the corresponding *department*. The both inner iterators, the associations classes: *projBuilder* and *Builder* are triggered using of associations existing between them and the associations classes: *deptMap*, the outer iterator. All the functions are defined in OCL and are similar to those used in the precedent two cases.

The OCL expressions for the mappings included in the association classes *ValueMap* and *Builder* were already presented in the first two cases. The corresponding OCL expressions for the association classes *projMap* and *projBuilder* are presented next.

```
context projMap::map(p: Proj): project
post: result = pr: project and pr.name = p.pname
```

This OCL expression defines the function *projMap::map* which maps a source object of type *Proj* to a target object of type *project*.

```
context projBuilder::build(pSet: ProjSet): projectSet
post: result = pSet.m_Projs
    ->Set()->collect(r: Proj | projMap.map(r): project)
```

This OCL expression defines the function *projBuilder::build* which maps a source set of *Proj* objects to a target set of *project* objects.

The OCL expression for the association class *deptMap* is similar to that of the second case. This function has to define the mapping of two sets at the same time.

```
context deptMap::map(dep: dept): department
post: result = d: department and
```

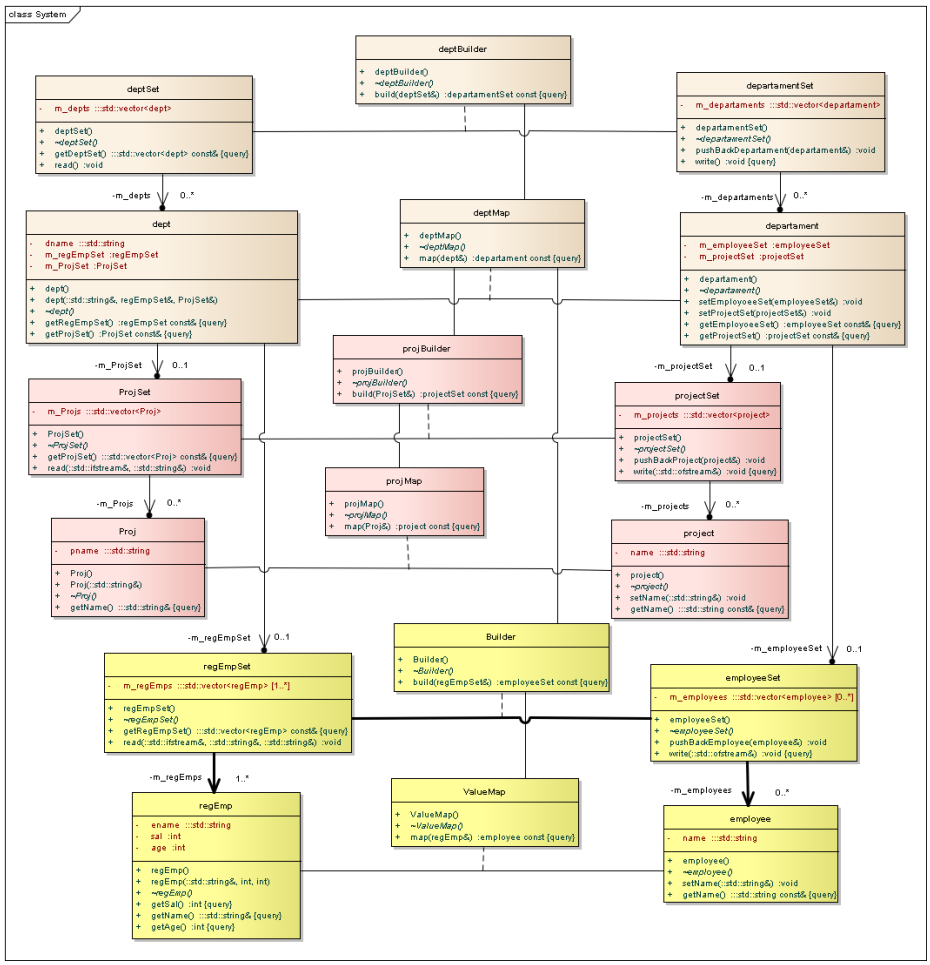

Fig. 6. The class diagram with a more complex mapping corresponding to Fig. 5

```
d.m_employeeSet = Builder.build(dep.m_regEmpSet) and
d.m_projectSet = projBuilder.build(dep.m_ProjSet)
```

The translations of the Clip *builders* or *value mappings* to UML are associations classes using respectively functions named *build* or *map*. This association classes are used alternatively at successive levels developing a hierarchy of functions. The function named *build* is always situated at the top level of the hierarchy and *map* at the bottom level. The function *build* from the top level or from another level of the hierarchy calls always only one function named *map* from a successive lower level and vice versa: a *map* function calls one ore more functions named *build* from a successive lower level. As already mentioned the function at the lowest level, the bottom level of the hierarchy, is always named *map*. This successive levels of alternations using functions named *build* and *map* are translating the Clip feature called Context Propagation Tree or CPT to UML.

The OCL expression of the function *deptBuilder::build* is identical to the one with the same name in the second case above.

This proves again that UML diagram produces the same mapping as the Clip diagram.

4 Nested Mappings and Tree-Like Data Structures

A join constrained by a CPT. A join in Clip is achieved by a *context*

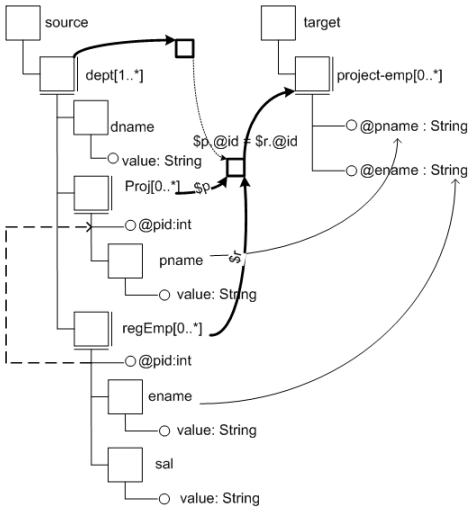

Fig. 7. A Clip mapping with a join constrained by a CPT (adapted from [13, Fig. 6])

propagation tree as shown in Fig. 7. The result is a flattened list of employees

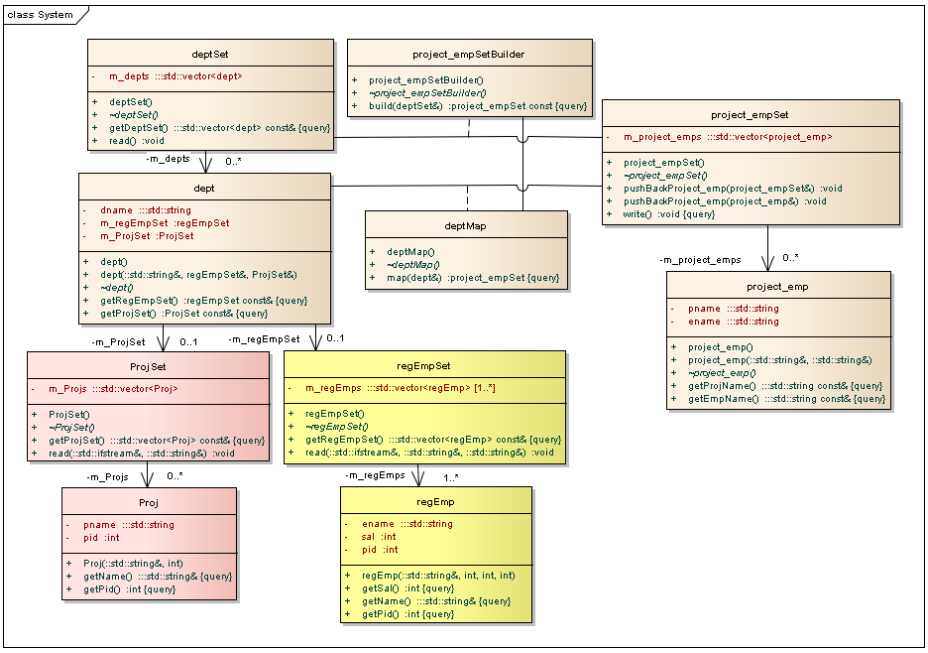

Fig. 8. A join, the class diagram corresponding to Fig. 7

and projects in which they work. The class diagram in Fig. 8 differs from the previous on the target side where instead of the class *departmentSet* the class *project_empSet* is introduced. As in Clip no *department* is created in the target. When a *build node* is reached from two or more *builders*, Clip computes a Cartesian product or a join if a condition involving two different variables is present. The UML translation is based on the definition of the Cartesian Product in OCL by [1]. The association class *project_empSetBuilder* starts the iteration over the elements of the set *deptSet*. Each iteration maps one object of type *dept* to a set of objects *project_emp* obtained from the join of the *Proj* and *regEmp* objects of each *dept* on the attribute *pid*. The OCL expressions define the join by constructing first a Cartesian Product and then a selection of the elements with the same attribute *pid* associating each employee with the projects in which he works. In this case the association class *deptMap* creates from each object *dept* a set of *project_emp*. The association class *project_empSetBuild* using this functionality maps the set of *dept* objects to the union of sets of *proj_emp* objects. We define the OCL expression for the function *deptMap::map* as:

```
context deptMap::map(dep: dept): project_empSet
def: projProdEmp = dep.m_ProjSet->collect(p: Proj | dep.m_regEmpSet
    ->collect(e: regEmp | Proj_regEmp: Tuple {Proj, regEmp}))
```

This is the Cartesian Product of the two sets included in an *dept*. The result is a set of tuples composed of a *regEmp* and a *Proj* each.

```
def: projJoinPidEmp = projProdEmp->select(Proj_regEmp |
Proj_regEmp.Proj.pid = Proj_regEmp.regEmp.pid)
```

The join is obtained by its definition from the Cartesian Product by selecting those tuples with the same *pid*.

```
post: result = projJoinPidEmp->collect(Proj_regEmp |
project_emp( Tuple {pname = Proj_regEmp.Proj.pname, ename =
Proj_regEmp.regEmp.ename}))
```

The result of this operation is a set of *project_emp* objects containing the name of the project and the name of the employee working in that project. The OCL definition of the function *project_empSetBuilder::build* uses the function *deptMap::map*.

```
context project_empSetBuilder::build(dSet: deptSet): project_empSet
post: result = dSet.m_depts->Set()->iterate(r: dept;
peS: project_empSet = {} | peS.pushBackProject_emp(deptMap.map(r)))
```

This function iterates over the set of *dept* objects and produces from each of them a set of *project_emp* objects and these sets are inserted in the *project_empSet*, a union of sets. This ensures that the Clip join and the described UML class diagram produce the same mapping.

A mapping with grouping and join. *Group nodes* are used to group source

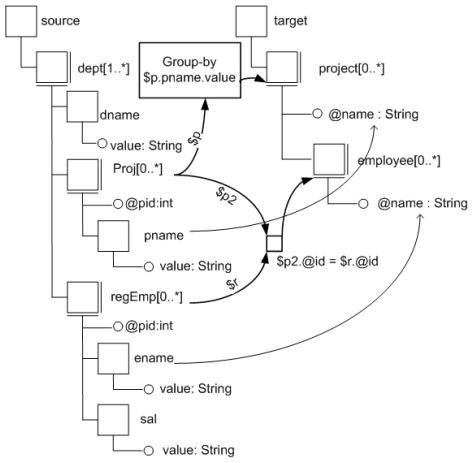

Fig. 9. A Clip mapping with a join and grouping (adapted from [13, Fig. 7])

data on attributes. Fig. 9 depicts such a construct. The result of a *group node* is a sequence of elements selected by the grouping attributes. The number of created sequences on the target equals the number of distinct values of the grouping attributes from the source. In Fig. 9 the *Projs* are grouped by *pname*. The *Projs*

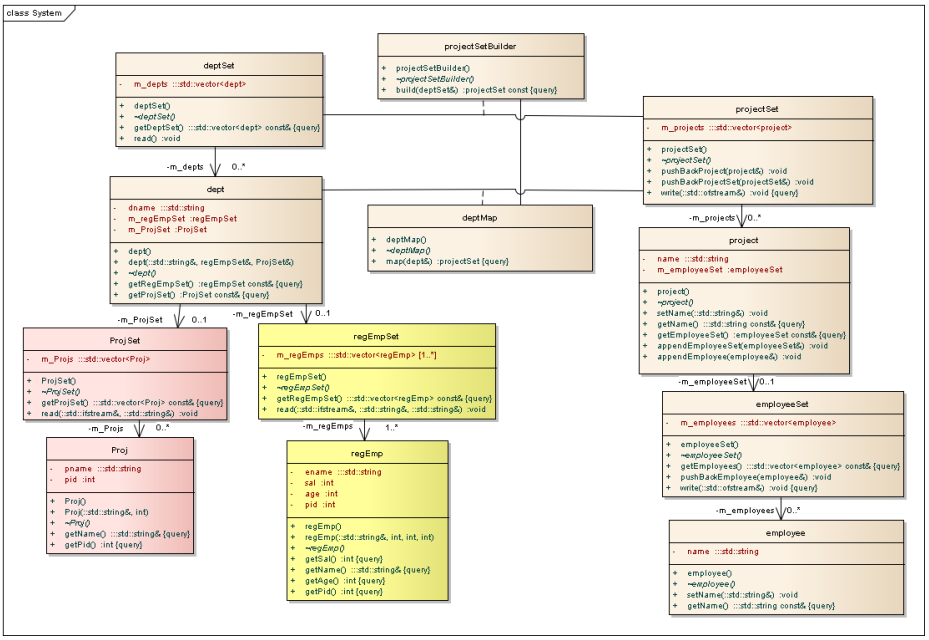

Fig. 10. A join and grouping, the class diagram corresponding to Fig. 9

and *regEmps* are joined by *pid* and finally the *employees* on the target are created and added to the *project* by name independently of the *dept* in which they work. The class diagram in Fig. 10 is the corresponding translation for grouping in Clip. The association class *projectSetBuilder* starts the iteration over the elements of the set *deptSet*. Each iteration using the function *deptMap::map* maps one object of type *dept* to an object of type *projectSet*. This set is obtained from the join of the *Proj* and *regEmp* objects of each *dept* on the attribute *pid*. On the target each *project* includes its *employee*. The function *deptMap::map* inserts each *project* into the *projectSet*. Each insert groups the *project* objects by name. In this case OCL expressions do not give a constructive solution but the OCL constraints define the possible implementations. The attribute *m_projects* of the type *projectSet* from the target is specified in OCL by the following expression:

```
context projectSet.m_projects
inv: self->isUnique(p: project | p.name )
```

This means that the elements of the set, the *project* objects, are unique by name. In this way the grouping by project name is achieved. In the UML diagram the type *project* has a set of objects of type *employee*. Because of this structure the only possible grouping is to attach all the employees to the project in which they work. If two or more projects have the same name by the uniqueness of the project name the employees of this projects are again grouped together. This is valid by the structure of the UML diagram also for projects in different

departments. The OCL expressions give the definition of the join by constructing first a Cartesian Product and then a selection of the elements with the same attribute *pid* associating each employee with the projects in which he works. It follows the OCL expression for the function *deptMap::map*:

```
context deptMap::map(dep: dept): projectSet
```

The OCL expressions defining the Cartesian Product and the join on *pid* have already been presented in the previous subsection.

```
def: result_lhs = projJoinPidEmp->collect(Proj_regEmp |
Tuple {pname = Proj_regEmp.Proj.pname, ename = Proj_regEmp.regEmp.ename})
```

This OCL expression creates all the tuples from the source that are to be grouped on *project* name in the target by the mapping.

```
def: result_rhs = projectSet.m_projects
    ->collect(p | p.m_employeeSet.m_employees
        ->collect(e | proj_emp: Tuple {p: project, e: employee}))
```

This OCL expression creates the Cartesian Product of the tuples from the target.

```
post: result = projectSet(result_lhs) and
result_lhs->forAll(pe | result_rhs->exists(proj_emp |
proj_emp.pname = pe.pname and proj_emp.ename = pe.ename))
```

This OCL expression defines the constraint that all tuples from the source must have a correspondent in the target. All elements from the target are created only from the source so it is not necessary to show that all elements from the target are only those that are created by the mapping. Every possible implementation must fulfill these constraints. The association class *deptMap* connects the class *dept* from the source with class *projectSet* from the source. The function *deptMap::map* transforms a *dept* to a *projectSet*. The association class *projectSetBuilder* connects the class *deptSet* from the source with class *projectSet* from the target. The function *projectSetBuilder::build* transforms the source to the target. The OCL definition of the function *projectSetBuilder::build* uses the function *deptMap::map*.

```
context projectSetBuilder::build(dSet: deptSet): projectSet
post: result = dSet.m_depts->Set()->iterate(r: dept;
pS: projectSet = {} | pS.pushBackProjectSet(deptMap.map(r)))
```

Inverting the nesting hierarchy. Another Clip example, a *group node* with hierarchy inverting is presented in Fig. 11. The source data is mapped to the target and as in the previous example, grouped on attributes, i.e. the mapping groups the *projects* by *name*. The *departments* are nested under the grouped *projects*, recall that in the source the *depts* have nested *Projs* hence the hierarchy inverting.

The class diagram Fig. 12 translates the grouping with inverting hierarchy from Clip to UML/OCL. The mapping is started by iterator from the association class *projectSetBuilder* on the elements of the set of *deptSet*.

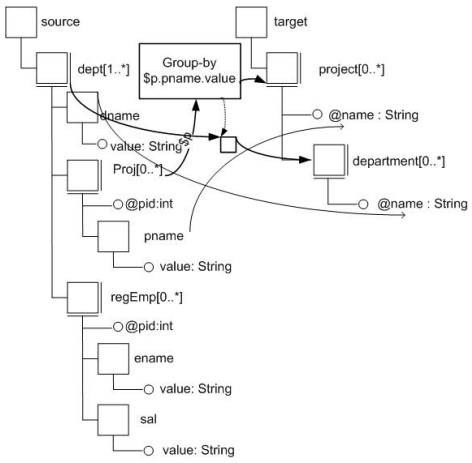

Fig. 11. A join and grouping with hierarchy inverting, the class diagram (adapted from [13, Fig. 8])

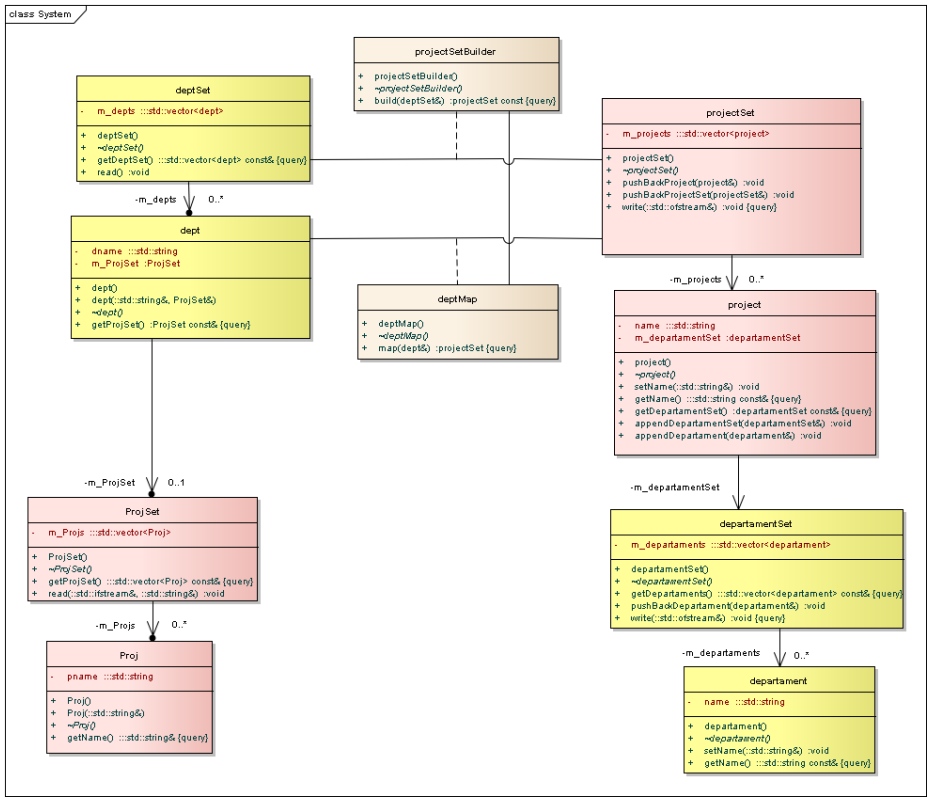

Fig. 12. A join and grouping with hierarchy inverting, the class diagram corresponding to Fig. 11

The function *deptMap::map* in each iteration maps one object of type *dept* to an object of type *projectSet*.

This set is obtained from the *Proj* elements of the current *dept* of the iteration. The current same *department* is inserted to each *project* object created by mapping from a *Proj* object. This operation actually inverts the hierarchy. Each inserted *project* is grouped by name. In this case OCL expressions do not give a constructive solution but the OCL constraints define the possible implementations.

The attribute *m_projects* of the type *projectSet* from the target is specified by the following OCL.

```
context projectSet.m_projects
inv: self->isUnique(p: project | p.name )
```

This means that the elements of the set, the *project* objects, are unique by name. In this way the grouping by project name is achieved.

In the UML diagram the type *project* has a set of objects of type *department*. Because of this structure the only possible grouping is to attach all the departments to the project belonging to it. If two ore more projects have the same name by the uniqueness of the project name the different departments are again grouped together. This is valid by the structure of the UML diagram also for projects in different departments.

We present next the OCL expression defining the function *deptMap::map*.

```
context deptMap::map(dep: dept): projectSet
def: result_lhs = dept.m_ProjSet
    ->collect(Proj | Tuple {pname = Proj.pname, dname = dept.dname})
```

This expression creates a set of tuples containing the name of the current department and each of the names of the projects of the current department that in this iteration are to be inverted.

```
def: result_rhs = projectSet.m_projects
    ->collect(p | p.m_departmentSet.m_departments
    ->collect(d | Tuple {pname = p.name, dname = d.name}))
```

This OCL expression computes the Cartesian Product of the target.

```
post: result_lhs->forAll(dp | result_rh
    ->exists(DP | dp.pname = DP.pname and dp.dname = DP.dname))
```

The constraint that we impose is that the elements from the source to be inverted in this iteration are included in the target. The uniqueness of the project name and the UML structure assure that the target actually inverts the hierarchy. All the elements from the target are the result of the mapping so no supplementary element exist in the target to those created by the mapping.

The association class *projectSetBuilder* connects the class *deptSet* from the source with class *projectSet* from the target. The function *projectSetBuilder::build* transforms the source to the target.

The OCL definition of the function *projectSetBuilder::build* uses the function *deptMap::map*.

```
context projectSetBuilder::build(dSet: deptSet): projectSet
post: result = dSet.m_depts->Set()->collect(r: dept;
pS: projectSet = {} | pS.pushBackProjectSet(deptMap.map(r)))
```

The UML solution produces also in this case the same final result as Clip.

A mapping with aggregates. Aggregate functions are presented in the last

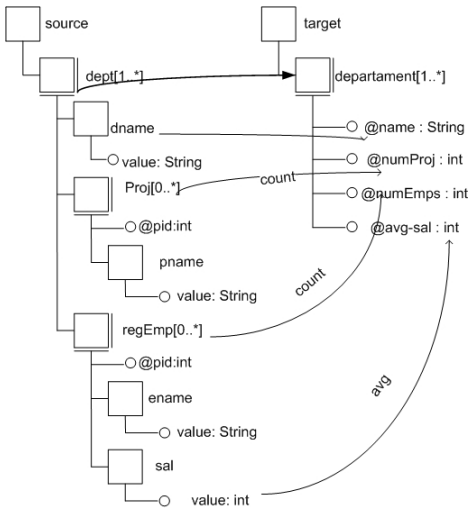

Fig. 13. A mapping with aggregates (adapted from [13, Fig. 9])

Clip mapping, Fig. 13. The class diagram Fig. 14 translates this mapping to UML. The *dept* objects from the source are mapped to the target as *department* objects. Target aggregate values are calculated for the source nested elements: *Projs* and *regEmps*.

The function *deptBuilder::build* starts the mapping iterating over the *deptSet*. In each iteration a *dept* object is mapped from the source to the target creating a *department* object. In this way the source *deptSet* is mapped to the target *departmentSet*. The mapping function is *depMap::map*. This function calls the functions that create the aggregates having as input sets. In a *dept* object there are two sets included: *ProjSet* and *regEmpSet*. The two functions *projBuilder::build* and *Builder::build* create the aggregate objects *projectProperties* and respectively *employeeProperties* which are again in the object *department* included. We present next the OCL expressions defining the mapping starting from the function creating aggregates for the *regEmpSet*.

```
context Builder::buildNumEmps(rEmpSet: regEmpSet): int
post: result = rEmpSet.m_regEmps->size()
```

This OCL expression defines a function taking as input a *regEmpSet* and returning the size of the set.

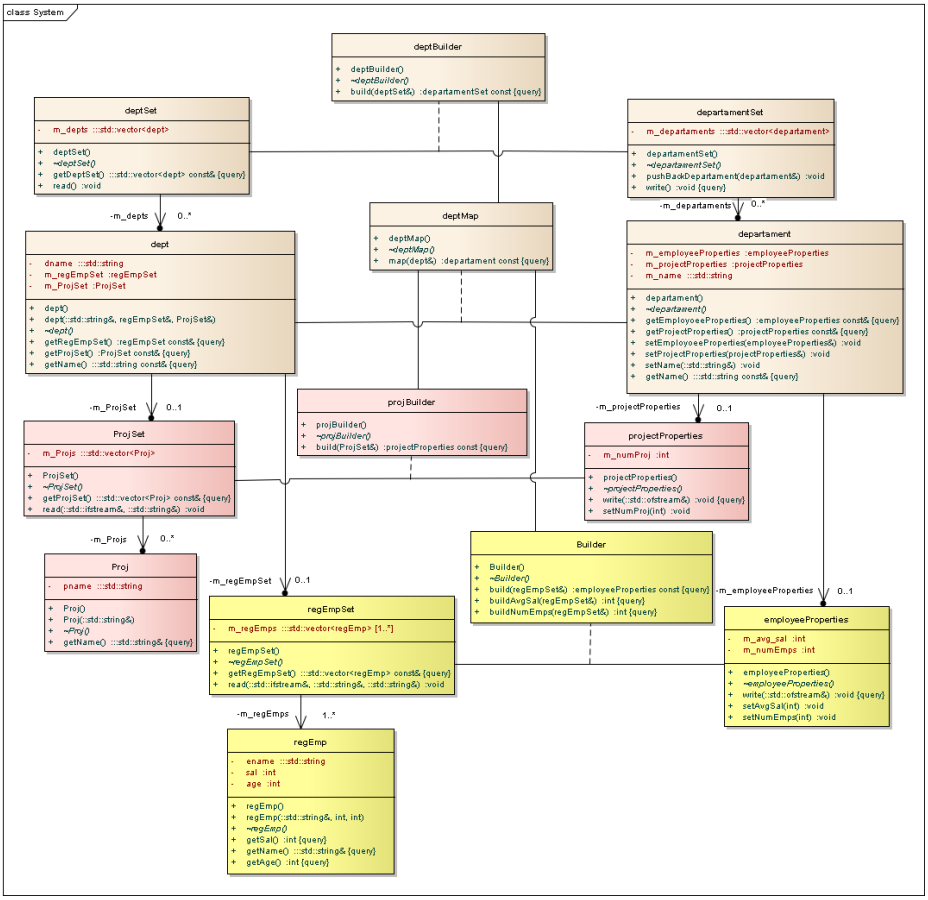

Fig. 14. A mapping with aggregates, corresponding to Fig. 13

```
contest Builder::buildAvgSal(rEmpSet: regEmpSet): int
post: result = rEmpSet.m_regEmps
    ->collect(r: regEmp | sal)->sum /  rEmpSet.m_regEmps->size()
```

This OCL expression defines a function using the same input and returning an average salary.

```
context Builder::build(rEmpSet: regEmpSet): employeeProperties
post : result = employeeProperties and
employeeProperties.m_avg_sal = buildAvgSal(rEmpSet) and
employeeProperties.m_numEmps = buildNumEmpsl(rEmpSet)
```

Now we use the previous two results in creating a *employeeProperties* object from a *regEmpSet*.

The following OCL expression describes the creation of an object *projectProperties* containing the number of the projects belonging to a *dept* object.

```
context projBuilder::build(pSet: ProjSet): projectProperties
post: result = projectProperties and
employeeProperties.m_numProj = pSet->size()
```

Now we have all the elements for mapping the source to the target creating a *department* from a *dept*.

```
context depMap::map(de: dept): departament
post: result = departement and
department.m_employeeProperties = Builder.build(de.m_regEmpSet) and
department.m_projectProperties = projBuilder.build(de.m_ProjSet)
```

The functions *Builder.build* and *projBuilder.build* create the desired aggregated values.

The mapping of sets of *dept* to sets of *department* is presented in the following expression.

```
context deptBuilder::build(dSet: deptSet): departamentSet
post: result = deptSet.m_depts->Set()->collect(r | deptMap.map(r))
```

As in previous cases the UML/OCL translation describes the same mapping as the Clip mapping.

5 Second Order and Target Dependencies

Second Order Dependencies. In [12] the authors consider the schema *mapping problem* defined as translating an instance of the source schema to an instance of the target schema. The *primary path* is defined as the set of elements found on the path from the root to an intermediate node or leaf in the tree structure of the source and target. The mapping is materialized with the use of *correspondences*: elements of source and target connected with arrows. The *correspondences* are modeled as *interpretations*, source-to-target referential constraints. In this context the *nested referential integrity constraint* is presented

and defined using the *primary path*, a large class of referential constraints that include relational foreign key and XML schema's key reference. Nested dependencies support the translation of the nested structure of the source and the target. This source-to-target dependencies are compiled into low level, language independent *rules*. These *rules* are used to obtain the transformations in concrete languages like XSLT or XQUERY for XML Schemas or SQL for relational schemas. For the creation of the new values in the target, that are not specified, one-to-one Skolem functions are used. In [7] the authors, based on [12], introduce the *nested mappings*. The mappings are defined by lines, *correspondences* connecting the source elements with target elements. Constraints, describing the mapping can be generated from these lines by tools like Clio or directly by human experts. The *nested mapping* is introduced allowing common subexpressions to be factored out. Grouping is another feature introduced by the use of *nested mappings*. The *nested mappings* are strictly more expressive as the mappings from [12] but less expressive as languages used for composition of s-t tgds as presented in [6, 9]. The authors note that for the relational model the *nested mappings* are a sub-language of the second-order tgds (SO tgds): every *nested mappings* can be rewritten, via Skolemization into a equivalent SO tgd but not vice-versa. In [13] the authors, based on [12, 7], introduce more complex mappings using second-order logical formulas expressing grouping and aggregate by means of functions. Recall the nested tgds of Fig. 7, the grouping, from Clip [13, IV. Language Semantics] expressing the fact that for each join on *pid* from the source there must be a *employee* nested inside of a *project* of the target. The correlation between the outer mapping and the inner mapping is achieved through the variable p', a *project* on the target, defined in the outer mapping and used in the inner mapping (supplementary the name of the *project* must be unique). This correlating element p' is the grouping element of the mapping. This could be expressed in first-order logic but the structure of the nested tgds is not preserved. This was the reason why the special Skolem function was introduced to solve this problem.

Our approach UMAP using UML class diagrams and OCL constraints does not explicitly use *nested mappings* [7], a sub-language of SO tgd, as Clip [13] does. Instead the diagram structure and the constraints define the possible query implementations that execute the mapping. A translation of UML/OCL to first-order predicate logic is given in [2]. The UMAP mapping, translated to first-order logic, could be compared with the nested mappings used by Clip. Clip introduces a special kind of Skolem function to express the grouping and the aggregates. A translation of the UML/OCL to nested tgds via a translation to first-order logic would need the same special Skolem function, showing the equivalence of the both mappings.

Target dependencies, target *egds* and *tgds*. UMAP allows the use of target *egds* and target *tgds*. The definition of an explicit mapping by nested *tgds* from [13, IV. Language Semantics] uses two expressions: C_1 and C_2. The first is a source expression. The second has three kinds of target conditions and could be used as *s-t tgds* but, by their definition, not as target *egds* or target *tgds*. In our translation, constraints could be defined for each element in the target

schema. An element of the target schema defined using an OCL expression to be unique is an example of using an target *egd*. The usage of the target *tgd* is possible through an OCL expression on the target. Special measures must be taken to limit the infinite cascading of tuples created in the target by restricting the target *tgds* to *weekly acyclic set of tgds* [5].

6 Conclusion

In this paper we have introduced UMAP, a new universal layer for schema mapping languages. Schema mappings are modeled by standardized UML class diagrams and OCL expressions. By restricting the UML artifacts to well-understood elements (e.g., classes, associations, aggregations, methods, and straightforward post-conditions and invariants), there is a well-defined semantics. This allows us to semi-automatically translate UMAP specifications to a broad range of target languages (like first-order logic, C++, Java, or XML). We have modeled a set of common schema mapping operations in UMAP, starting with basic source-to-target dependencies over join and grouping operations to more advanced mappings like second-order dependencies or target dependencies. We translated several core features of Clip to UMAP (see [4] for a translation of all Clip artifacts). There is also an implementation available (see `http://www.dbai.tuwien.ac.at/research/project/umap`) generating C++ code showing the translation of typical Clip language elements to our UML-based formalism, both for its core features and tree-like data structures, proving that our approach works in practical usage aimed at industrial application. UMAP can be seen as a new middleware for high-level visual schema mapping languages. Therefore we propose to use UMAP as a back-end when creating new visual mapping languages. This allows the designer to use a common basic language under the hood (like a visual schema mapping language assembler) without having to worry about generation of bindings for target languages. As future work we plan to implement interfaces and semi-automatic compilation to several targets for additional schema mapping languages and tools.

References

1. David H. Akehurst and Behzad Bordbar. On querying UML data models with OCL. In *UML 2001*, volume 2185 of *LNCS*, pages 91–103. Springer, 2001.
2. Bernhard Beckert, Uwe Keller, and Peter Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *VERIFY, FLoC*, 2002.
3. Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1–2):70–118, 2005.
4. Florin Chertes. DBAI-TR-2012-76. Technical report, DBAI, Institute of Information Systems, Vienna University of Technology, 2012.
5. Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224. Springer, 2003.

6. Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Transactions on Database Systems*, 30(4):994–1055, 2005.
7. Ariel Fuxman, Mauricio A. Hernandez, Howard Ho, Renee J. Miller, Paolo Papotti, and Lucian Popa. Nested mappings: schema mapping reloaded. In *VLDB 2006*, pages 67–78. ACM, 2006.
8. Maurizio Lenzerini. Data integration: a theoretical perspective. In *PODS 2002*, pages 233–246. ACM, 2002.
9. Alan Nash, Philip A. Bernstein, and Sergey Melnik. Composition of mappings given by embedded dependencies. In *PODS 2005*, pages 172–183. ACM, 2005.
10. Object Management Group. *Unified Modeling Language 2.4.1*, 2011. `www.omg.org`.
11. Object Management Group. *Object Constraint Lang. 2.3.1*, 2012. `www.omg.org`.
12. Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *VLDB 2002*, pages 598–609. Morgan Kaufmann, 2002.
13. Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: a visual language for explicit schema mappings. In *ICDE 2008*, pages 30–39. IEEE, 2008.