Acta Cybernetica **13** (1997) 103-140.

Framework for Generating Object-Oriented Databases from Conceptual Specifications

György Kovács * Patrick van Bommel †

Abstract

When designing underlying databases of information systems, data are first modelled on conceptual level, then the obtained conceptual data models are transformed to database schemas. The focus of this paper is the transformation of conceptual models into database systems with object-oriented features. The transformation is captured within the framework of a two level architecture. Conceptual models are first mapped to abstract intermediate specifications, which are then transformed to database schemas in a given target environment. This enables us to treat different target systems, such as object-oriented and object-relational systems including the standards ODMG and SQL3, in a uniform way. To express intermediate representations of conceptual models we use F-logic, a logic-based abstract specification language for object-oriented systems. We focus on the first step of the overall transformation, i.e. the mapping of conceptual models into F-logic. Several transformation alternatives are discussed, and a corresponding graphical notation for specifying transformation alternatives is provided.

Keywords: database design, data transformation, conceptual data models, OO models

1 Introduction

It has been generally agreed on that conceptual data modelling is very important when building information systems. This means that data must be modelled first on conceptual level, and then the obtained conceptual model (conceptual schema) must be translated to the external and internal level, according to the three level architecture for information systems modelling ([15]). By doing so, the issues of correctness and efficiency are well-separated, which is quite desirable. In this paper we deal with the transformation of conceptual data models to the internal level.

*Department of Information Systems, Eötvös Loránd University, Múzeum krt. 6-8., H-1088 Budapest, Hungary, email: gykovacs@ullman.inf.elte.hu. Supported by the Hungarian Scientific Research Fund (OTKA W-015176) and Nuffic.

†Department of Information Systems, University of Nijmegen, Toernooiveld 1, NL-6525 ED Nijmegen, The Netherlands, email:pvb@cs.kun.nl

Internal (implementation-oriented) models are considered to be database models that are supported by database management systems (DBMS) running on computers. It is also assumed (required) that some database language is provided for such a model. The well-known relational model ([11], [36]) is the underlying database model of today's RDBMSs and the related database language is typically SQL (see e.g. [13]). However, the relational technology is not always appropriate for some real-life applications, e.g. multimedia or geographical applications, where usually highly structured, complex objects must be stored and manipulated. To overcome the limitations of the relational model, advanced database models have been developed, such as the nested relational model (see e.g. [30], [1], [5]) along with proposals for nested SQL extensions (e.g. [27], [28], [23]) as well as database models with object-oriented features. Database systems with object facilities serve as candidates for next generation database systems. Although a number of such models and query languages have been proposed (see e.g. [2], [32], [4], [31], [19]), there is no object-oriented database model and language yet, that has been commonly accepted. This fact is known and inspired people attempting to define the requirements for next generation DBMSs ([3], [34], [20], [12]). Basically, two main approaches, the pure object-oriented (OO) and the object-relational (OR), compete with each other and seem to co-exist in the future. The same is reflected in the standardisation efforts resulting in the ODMG-93 ([9]) de facto standard for truly object database systems and SQL3 ([24]) for object-relational systems, though some compatibility between them is also aimed.

For data modelling a number of semantic modelling techniques have been developed, such as (extended) ER ([10], [14]), NIAM ([26]) and PSM ([18], [16]). Semantic modelling has been used in practice for long and has proved to be a powerful technique. The mapping of resulting conceptual schemas into relational environments is well-defined (see e.g. [35], [26]) and this process is supported by many CASE-tools. Also, a transformation mechanism to nested relational schemas has been established ([8], [7]). Although there exist OO data modelling techniques, e.g. OMT ([29]), for designig object-oriented databases, the use of traditional semantic modelling will very likely not disappear from system design, but will remain as a powerful alternative, especially in data intensive domains. Indeed, in [33], where semantic modelling (using extended ER) and OO data modelling are compared, it is concluded that even when OO databases are designed the recommended strategy is: (1) creating an EER (or e.g. NIAM) schema; (2) map it to an OO schema; and (3) augment OO schema with behavioral constructs. As a consequence, a mechanism for transforming semantic models into modern database systems is needed.

In this paper we deal with the problem of how to transform conceptual data models into somehow object-oriented database environments. Although valuable previous work has been done on this topic (e.g. [25], [21], [6]), a general unifying mechanism is still missing, which inspired our work. In general, a conceptual data model (conceptual schema) consists of an information structure and a set of integrity constraints, both of which require translation. In addition to the results of existing proposals, advanced modelling constructs (set types, list types, generalisation) are to be considered as part of the structure translation process. Even for

simple constructs additional tranformation alternatives can be recognized. Moreover, a more comprehensive treatment of constraints is necessary in general.

In our approach (similarly to that of [6]) the transformation is captured within the framework of a two level architecture. Conceptual schemas are first transformed to abstract intermediate specifications (design step). Then the obtained intermediate specifications are translated into the final implementation environment (implementation step). This means that in case of different target environments the same design step can be applied and only the implementation step will differ. That is, choosing a new target system requires only the implementation step to be adapted. Thus, as the main benefit of this approach, we gain general applicability. The common design decisions can be factored out in the first step. Here we focus on the design step, the second step is discussed only in very general terms.

For expressing intermediate representations F-logic ([19]), a logic-based abstract specification language for object-oriented systems, is used (cf. [6]). Conceptual models are defined in terms of PSM (Predicator Set Model, [18], [16]), a fully formalized extension of NIAM. However, our approach is easily applicable to other conceptual modelling techniques (e.g. ER) due to the usage of similar constructs.

In the present paper we set up the framework for a general transformation mechanism consisting of two steps as discussed above. It serves as a basis towards working out a comprehensive method for designing modern (OO,OR) databases based on conceptual (semantic) data modelling. We focus on structural aspects and outline a number of alternatives for the translation of information structures into OO systems. A corresponding graphical notation is introduced for illustration purposes. Since the mapping of structures is influenced by simple uniqueness (key) constraints, such constraints are also covered. However, the transformation of complex conceptual constraints in general is beyond the scope of the present paper, though it belongs to the whole picture and is seen as an essential part of the overall transformation, which has to be worked out.

The rest of the paper is organized as follows. In section 2 our approach is presented. In section 3 the conceptual data modelling technique PSM is summarized to an extent needed for the purpose of the paper. Section 4 gives an overview of F-logic, that is used for expressing intermediate specifications. Alternatives for the transformation of conceptual information structures into F-logic (the design step) are discussed in section 5. In section 6 the transformation of an example PSM schema into F-logic is worked out in detail. The implementation step is discussed in general terms in section 7, where also an example schema definition in ODMG-93 is provided. Section 8 contains the conclusions and topics for further work.

2 Approach

The expected end product of the transformation of a conceptual data model is something that can be run on a computer with a given target environment (DBMS) for creating a database. That is, at the end a sequence of statements in the database language of the assumed DBMS has to be generated to create the corresponding

database schema. As it was already mentioned, in our approach the transformation is captured in the framework of a two level architecture, i.e. it is performed in two steps as shown in figure 1. Conceptual schemas are first transformed to abstract intermediate specifications (design step). Then the obtained intermediate specifications are translated into the final implementation environment (implementation step). The task of the second step is the generation of statements in a concrete database language. In [6] a similar approach is taken.

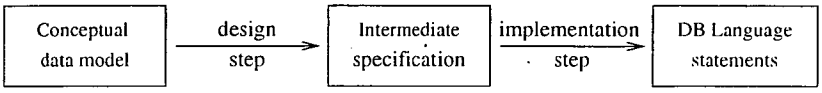

Figure 1: Two level architecture for transformation

There are several advantages of a two level architecture approach, e.g.:

- Provided that the intermediate specification language is general enough to cover all the final target models that are intended to be considered, the design step, which is the more complex and essential part of the whole transformation, becomes the same single task for different target environments and is independent of the choice of the actual target system. The different system specific details must be dealt with in the implementation step only.

- Provided that conceptual models and intermediate specifications have underlying precise formalism, the transformation from the conceptual to the intermediate level can be given algorithmically in a formal framework. This is very fundamental in order to have an automated transformation mechanism.

- Since a given conceptual model may have a number of correct representations on the internal level and possibly the best candidate should be chosen, optimization is important. In a two level transformation optimization can be incorporated at both levels.

We have already made it clear that to express data models on the conceptual level, we will use the Predicator Set Model ([18], [16]), an extension of NIAM ([26]). PSM is a fully formalized expressive modelling technique. It is briefly summarized in section 3. As potential final target environments, truly object-oriented as well as object-relational database systems are considered including the related standards ODMG and SQL3, respectively. Fixing an appropriate specification language for specifying intermediate models is a basic task. When doing so, the following requirements are fundamental to be taken into account:

- The chosen specification language should be implementation-oriented, i.e. it should be able to deal with (important) concepts of implementation environments. At the same time it should provide a high level of abstraction to make it possible for us not to deal with the irrelevant aspects in the design step.

- It must be general enough and support object-oriented concepts to cover object-relational and object-oriented target systems.
- It must be provided with sufficient support for constraint specifications. On the one hand, because the transformation of structures often imply integrity constraints in the target database to be specified. On the other hand, because the translation of conceptual integrity constraints typically (but not always) results in database constraints. Although the general treatment of constraints and their translation is outside the scope of our present paper, this is a very essential perspectival requirement.
- It must have formal syntax and semantics. This makes it possible to define our transformation in a formal framework.

To sum it up, we need an abstract and formal database model with object-oriented facilities, that also allows to specify integrity constraints. After investigating a number of proposals (e.g. [2], [32], [31], [19]) we have concluded that F-logic ([19]) is the one that fulfils our needs the best. Models of other proposals are not general enough and/or are not provided with precise formal syntax and semantics and/or do not deal with constraints at all. Consequently, we use F-logic as an abstract intermediate specification language. In section 4 an overview of F-logic is given based on [19], where it was presented. We note that F-logic has a pure object-oriented view. However, it can serve as an abstract intermediate specification language in case of object-relational database sytems as well. The picture of figure 1 can now be refined to show our approach more concretely, which is depicted in figure 2.

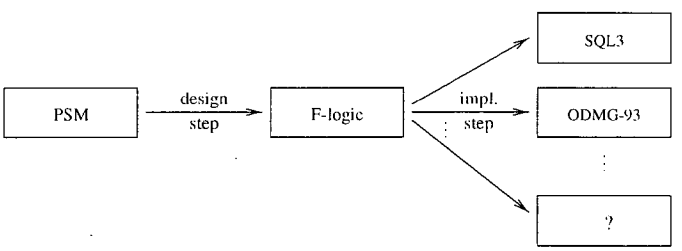

Figure 2: The concrete architecture

3 Conceptual data models

In this section we give a brief overview of the Predicator Set Model (PSM), that is used for expressing conceptual data models, without going into formal details (for details see [18] or [16]). A conceptual data model $\Sigma = \langle \mathcal{I}, \mathcal{C} \rangle$ consists of an information structure $\mathcal{I}$ and a set of integrity constraints $\mathcal{C}$. An information structure $\mathcal{I}$ is a structure consisting of the following basic components:

1. A set $\mathcal{P}$ of *predicators*. A predicator is intended to specify the role played by an object type in a fact type (see below).
2. A set $\mathcal{O}$ of *object types*. Object types are classified as follows:
   (a) Entity types ($\mathcal{E}$) and label types ($\mathcal{L}$). The difference is that labels can, in contrast with entities, be represented (reproduced) on a communication medium. $\mathcal{D}$ is a set of concrete domains (e.g. string, natno) associated with label types via the function $\mathsf{Dom} : \mathcal{L} \longrightarrow \mathcal{D}$.
   (b) Fact types ($\mathcal{F}$). The set $\mathcal{F}$ is a partition of the set of predicators $\mathcal{P}$. The fact type that corresponds with a predicator is obtained by the auxiliary function $\mathsf{Fact} : \mathcal{P} \longrightarrow \mathcal{F}$.
   (c) Power types ($\mathcal{G}$) and sequence types ($\mathcal{S}$). Power types are also called set types. The intention of sequence types is to model list structures.
3. A function $\mathsf{Base} : \mathcal{P} \longrightarrow \mathcal{O}$ specifying the object type associated to a predicator.
4. A function $\mathsf{Elt} : \mathcal{G} \cup \mathcal{S} \rightarrow \mathcal{O}$ specifying the element type of power types and sequence types.
5. A binary relation Spec on object types, capturing specialisation. $a \,\mathsf{Spec}\, b$ is interpreted as "a is a subtype (specialisation) of b", or "b is a supertype of a"". Specialisation of label types is prohibited, and only entity types can act as subtypes ($\mathsf{Spec} \subseteq \mathcal{E} \times \mathcal{O} \backslash \mathcal{L}$). Specialisation networks are acyclic.
6. A binary relation Gen on object types, capturing generalisation. $a \,\mathsf{Gen}\, b$ is interpreted as "a is a generalisation of b", or "b is a specifier of a". Generalisation of label types is prohibited, and only entity types can act as generalised object types ($\mathsf{Gen} \subseteq \mathcal{E} \times \mathcal{O} \backslash \mathcal{L}$). Generalisation networks are acyclic. Furthemore, to avoid conflicting situations, generalised object types cannot be subtypes. The difference between generalisation and specialisation lies in their population (see below).

The connection between (abstract) entity types and (concrete) label types is established by so-called *bridge types*. A fact type f is called a bridge type only if it has the form $f = \{p, q\}$ with $\mathsf{Base}(p) \in \mathcal{L}$ and $\mathsf{Base}(q) \notin \mathcal{L}$. A fact type is called an objectified fact type if it is the base object type of some predicator.

Figure 3 shows an example information structure. In this figure we have a number of entity types, e.g. *Person* and *Project*, represented by circles. Label types, also represented by circles, appear in parentheses, examples are *Date* and *Tel_nr*. By convention, if a label type is an identifier for an entity type, then the label type is represented within the same circle, see e.g. entity type *Person* and its identifying label type *P_id*. Fact types consist of predicators represented by boxes connected with circles for their base object types. For example, fact type *Employment* is a binary fact type consisting of two predicators, *employed_by* and *employs*.

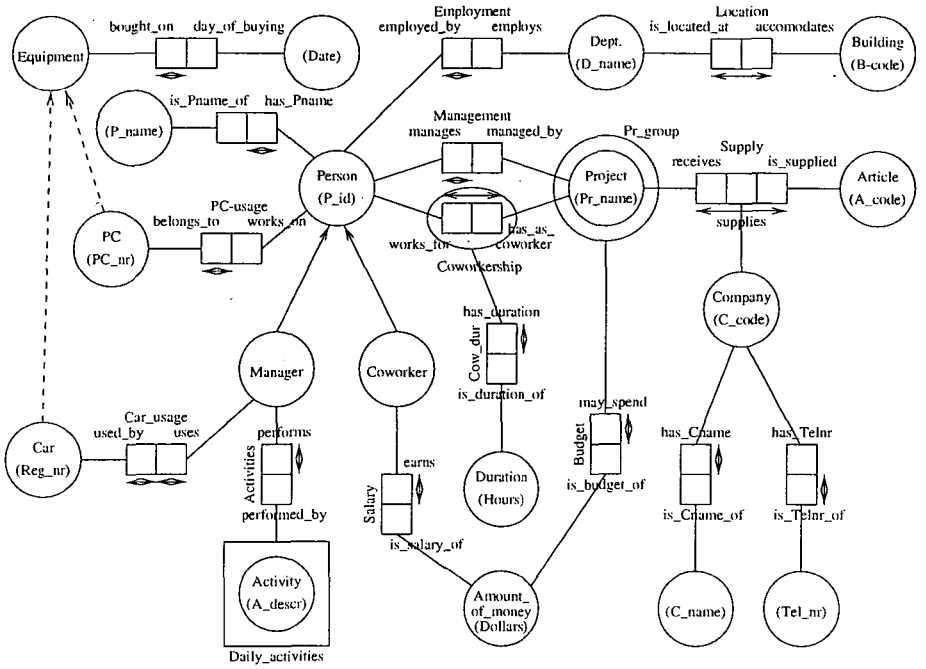

Figure 3: Information structure with uniqueness constraints

Figure 3 also contains representatives of advanced modelling constructs. We have power type *Pr_group* with element type *Project* and sequence type *Daily_activities* with element type *Activity*. Power types and sequence types are represented by circles and boxes, respectively, around their element type.. Entity types *Manager* and *Coworker* are specialised object types (subtypes) represented by solid arrows from subtypes to supertypes. Their common supertype is entity type *Person*. Entity type *Equipment* is a generalised object type with entity types *Car* and *PC* as its specifiers. Generalisations are represented by dashed arrows from specifiers to generalised types. As distinguished fact types, for bridge types many examples can be seen, e.g. $\{has\_Cname, is\_Cname\_of\}$ connecting entity type *Company* with label type *C_name*. As an example of objectified fact types we have fact type *Coworkership*.

Populations and constraints

An information structure is used as a frame for some part of the (real or fictive) world, the so-called Universe of Discourse (UoD). A *state* of the UoD corresponds with a so-called instantiation or population of the information structure, and vice versa. A population $\mathsf{Pop}_{\mathcal{I}}$ of an information structure $\mathcal{I}$ is a value assignment to the object types in $\mathcal{O}$, conforming to the structure as prescribed in $\mathcal{I}$. The population

of a label type comes from the corresponding concrete domain (e.g. string, natural number), while the population of an entity type comes from an abstract domain containing unstructured values.

These domains are part of the *universe of instances* Ω, which is inductively defined as follows. Firstly, all possible atomic instances are contained in Ω. Secondly, Ω contains all possible composed instances such as (a) mappings from predicators to instances, intended for the population of fact types, (b) sets of instances, intended for the population of power types, (c) sequences of instances, intended for the population of sequence types. The basic difference between specialisation (subtyping) and generalisation is that a subtype gets the population (and identification) from its supertype using subtype defining rules, while the population of a generalised object type is the union of the population of its specifiers. For a formal treatment of populations we refer to [17].

Forbidden populations are excluded by so-called (static) integrity constraints. Uniqueness constraints require uniqueness of values in some set of predicators. Graphically such uniquness constraints are represented by double-headed arrows next to the predicators they belong to. For example, in figure 3 we have unique(*employed_by*) expressing that a person may belong to one department only. There is a variety of other constraints, such as total role, occurrence frequency, set, enumeration, power type, sequence type, and specialization constraints. These are not relevant for the purpose of the present paper.

4 F-logic

To express intermediate specifications we use F-logic, an abstract logic-based language for OO systems. In this section we shortly summarize the parts relevant for us. For a comprehensive description of F-logic we refer to [19], where it was presented. In [22] an overview of F-logic from the perspective of our transformation is given.

Basic elements in F-logic are *id-terms*, terms built from function symbols and variables as usual. They denote objects, classes and methods. Ground id-terms are variable-free id-terms playing the role of *logical* object identifiers (*oid*). By means of id-terms *F-molecules* can be constructed, and from F-molecules more complex formulas can be built. F-logic is provided with a model-theoretic semantics defined by means of semantic structures called *F-structures*. F-structures and the satisfaction of formulas are defined in such a way that the commonly known OO features are incorporated. Along with the description of its syntax, below we give an informal summary of the semantics of F-logic, for details see [19]. F-molecules are defined as follows (C, D, O, M, R, A_i-s, R_i-s, AT_i-s and RT_i-s below are id-terms, $k, l \geq 0$).

Is-a assertions of the form $C :: D$ and $O : C$ stating that class C is a subclass of class D and object O is a member of class C, respectively. Each class is subclass and superclass of itself. The subclass relation is transitive, and subclass hierarchies are acyclic. Objects belonging to a class also belong to any superclass of that class.

Structures (signature expressions, see below) are inherited from superclasses.

Object mulecules of the form $O[$ a ';'-separated list of *method expressions* $]$. A method expression can be a *scalar data expression* $M@A_1,...,A_k \rightarrow R$, a *set-valued data expression* $M@A_1,...,A_k \twoheadrightarrow \{R_1,...,R_l\}$, a *scalar signature expression* $M@AT_1,...,AT_k \Rightarrow (RT_1,...,RT_l)$, or a *set-valued signature expression* $M@AT_1,...,AT_k \Rightarrow\!\!> (RT_1,...,RT_l)$. Here O denotes an object or a class. M corresponds to a method. In data expressions it denotes method invocation, while in signature expressions it denotes the signature of some method. The syntactic context of M indicates that the corresponding method is a scalar function ($\rightarrow$, $\Rightarrow$) or a set-valued function ($\twoheadrightarrow$, $\Rightarrow\!\!>$). In scalar data expressions R represents the output of the method M when invoked on object O with arguments $A_1,...,A_k$. In set-valued data expressions R_i-s represent elements of the resulting set. In signature expressions RT_i-s represent the *types* (classes) of the result (scalar case) or the types of the elements of the result (set-valued case) of the method M when invoked on an object of class O with arguments of types $AT_1,...,AT_k$. The output (or the elements of the output, resp.) of the method must belong to *all* the RT_i classes. When only one result type is specified the parentheses may be omitted.

From F-molecules complex formulas (*F-formulae*) can be built by means of logical connectives ($\wedge, \vee, \neg$) and quantifiers ($\forall, \exists$) with their usual interpretation. The implication connective " $\longleftarrow$ " can also be used as usual, i.e. $\varphi \longleftarrow \psi$ is a shorthand for $\varphi \vee \neg\psi$.

F-logic databases (F-programs), well-typing

An F-logic *database*, also called an *F-program*, is basically an arbitrary set of F-formulae. Since this definition is too general, restrictions on the form of the allowed formulas are applied. An F-program **P** consists of a set of *rules*, statements of the form *head* $\longleftarrow$ *body*, where *head* is an F-molecule and *body* is a conjunction of literals (F-molecules or negated F-molecules). The semantics of F-programs is given by Herbrand interpretation, more concretely by *canonic Herbrand models* (H-models). An F-program can be structured in such a way that its schema (declaration) part and data (object base) part are shown separately.

Example 4.1

Let's suppose that we have a simple database about persons, employees and projects. Such a database can be defined by the following F-program:

Schema (declarations)

$employee :: person$

$person[name \Rightarrow string;$
$\quad age \Rightarrow integer]$

$employee[salary \Rightarrow integer;$
$\quad works\_for \Rightarrow\!\!> project]$

$project[title \Rightarrow string;$

$budget \Rightarrow integer;$
$is\_involved@employee \Rightarrow boolean]$

Object base

$smith : person$

$jane : employee$

$natlang : project$

$dbopt : project$

$smith[name \rightarrow "John\ Smith";$
$age \rightarrow 38]$

$jane[name \rightarrow "Eva\ Jane";$
$age \rightarrow 22;$
$salary \rightarrow 5100;$
$works\_for \twoheadrightarrow \{dbopt, natlang\}]$

$natlang[title \rightarrow "Natural\ Language\ Processing";$
$budget \rightarrow 50000]$

$dbopt[title \rightarrow "Database\ Optimization";$
$budget \rightarrow 65000;$
$is\_involved@jane \rightarrow true]$

Note that class membership relations concerning simple objects, e.g. $38 : integer$*, are omitted in the example.* □

In the schema part of example 4.1 we defined the classes *person*, *employee* and *project*. The is-a assertion states that *employee* is a subclass of *person*. The object molecules in the schema contain only signature expressions specifying argument and result types of methods and attributes. For example, *works_for* is a set-valued attribute in class *employee* returning a set of *project* objects for an employee. *is_involved* represents a scalar method with one argument of type *employee*. When it is applied to a project with a given employee it returns *true* or *false*.

The data part of example 4.1 describes the actual content of the database. Objects are identified by logical object identifiers. Is-a assertions represent class memberships. For instance, the example shows that Eva Jane identified by *jane* is an employee. For individual objects the values of attributes, or more generally the results of method invocations with some arguments, are given by data expressions , e.g. $name \rightarrow "Eva\ Jane"$ for employee *jane*, and $is\_involved@smith \rightarrow true$ for project *dbopt*.

The connection between data expressions and signature expressions is not captured by the defintion of F-structures. This link is provided at a meta level by means of well-typing conditions. Informally, an F-program **P** is *well-typed* if canonic H-models of **P** obey the type restrictions given by signature expressions occuring in **P**. This means that (1) for any data expression on any object in **P** there exists a covering signature expression (i.e. the method can be invoked on the object with

the given arguments), and (2) the result of such a data expression is of type prescribed by the covering signature expression. Note that the F-program of example 4.1 is well-typed.

Constraints

In database systems usually a set of integrity constraints is associated with a particular database to disallow invalid database states. F-logic itself does not define the concept of integrity (semantic) constraints. In [6], however, it has been extended for this purpose. We use the extension presented there. An *integrity constraint* in F-logic is an arbitrary F-formula. An *F-program with constraints* is a tuple $(\mathbf{P}, \mathbf{IC})$, where $\mathbf{P}$ is an F-program and $\mathbf{IC}$ is a set of integrity constraints (F-formulae). As a syntactic convention, integrity constraints are preceded by the symbol "!– ". The semantics of an F-program with constraints $(\mathbf{P}, \mathbf{IC})$ is defined by some canonic H-model of $\mathbf{P}$ that is also a model of $\mathbf{IC}$. In order to be a *well-typed F-program with constraints* $(\mathbf{P}, \mathbf{IC})$, $\mathbf{P}$ must be well-typed.

Example 4.2

Let's suppose that in the F-logic database of example 4.1 project titles must be unique. The corresponding F-program now is extended to become an F-program with constraints containing the single constraint as follows:

$$!\!- \; X_1 \doteq X_2 \longleftarrow X_1 : project\,[\,title \rightarrow Y\,] \wedge X_2 : project\,[\,title \rightarrow Y\,]$$

□

In the above example we used $X_1 : project\,[\,title \rightarrow Y\,]$ as a shorthand for $X_1 : project \wedge X_1 : [\,title \rightarrow Y\,]$ (with X_2 analogously). This kind of shorthand is often used in F-logic. The equality predicate $\doteq$ (defined in [19]) expresses that two objects are identical.

In section 6.2 some macros (shorthands) will be introduced for specifying key, mandatory and inverse constraints on F-logic level. Such constraints have to be generated during the transformation of information structures in several situations.

Lists and sets

For the transformation of PSM sequence types we need an F-logic construct that is usually known as *list*. F-logic itself is not provided with list data types, but lists can be modelled by defining a parametric family of classes, see [19] or [22]. We also need to translate PSM power types. Although the concept of set-valued attributes (methods) enable to manage sets, its applicability is limited. Defining attributes of nested sets (sets of sets) is not easy and natural. In [22] we defined a parametric family of classes for this purpose. From now on we assume that the aforementioned parametric classes are defined in each F-program, and $list(t)$ and $set(t)$, where t is a ground id-term denoting a class, can be used in F-programs whenever needed.

5 From PSM to F-logic: the design step

5.1 Framework

In this section we outline the essence of our approach to the transformation of conceptual models into F-logic (design step, see figure 2) by means of activity graphs with different decomposition levels. States denoted by ellipses are input and/or output of activities denoted by rectangles.

Basically, since the final goal is database schema generation, we are interested in schema (data structure + integrity constraints) transformation. However, the semantics of conceptual structures, and more particularly, the constraints are defined in terms of populations. This means that we have to deal with population transformation too. In fact, populations must be transformed anyway when our approach is integrated in an executable transformation mechanism concerning also operations and their transformation. Such an execution model is essential e.g. for optimization.

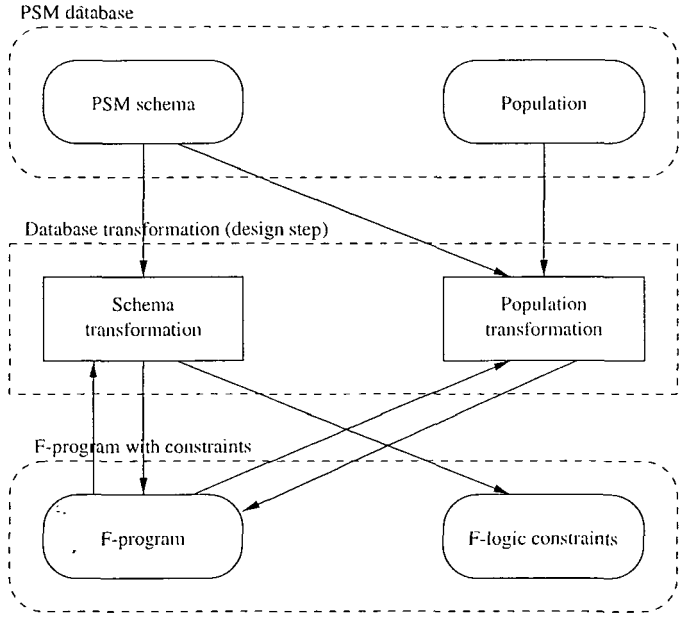

Figure 4: The design step

As depicted in figure 4, a PSM schema $\Sigma = \langle \mathcal{I}, \mathcal{C} \rangle$ together with a population $\mathsf{Pop}_{\mathcal{I}}$ (PSM database) is translated to an F-program with constraints $(\mathbf{P}, \mathbf{IC})$. The activity graph of this figure already shows a first level decomposition to separate the schema from the population and their transformations. The population is represented in $\mathbf{P}$ (object base part), while the schema (structure + constraints) is represented in $\mathbf{P}$ (declaration part) as well as in $\mathbf{IC}$. For the transformation

of populations and constraints (part of the schema) the result of the structure transformation is needed. That is the reason for the upward-directed arrows.

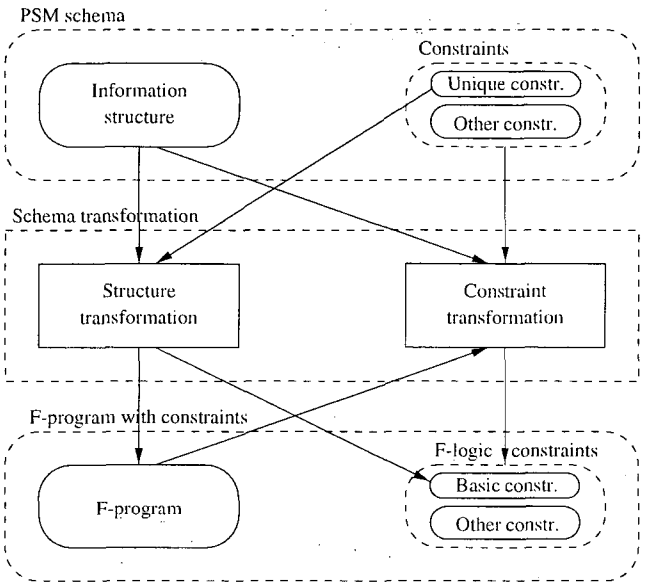

Figure 5: Decomposition of schema transformation

To illustrate the transformation of schemas in more detail, the corresponding part of the diagram of figure 4 is decomposed as depicted in figure 5. The information structure $\mathcal{I}$ becomes (the declaration) part of **P** on the one hand, and some basic constraints (e.g. inverse constraints) are also generated. Figure 5 also shows that the transformation of structures is influenced by uniqueness constraints. Except simple uniqueness constraints (over single predicators), the conceptual constraints in $\mathcal{C}$ are translated to F-logic constraints in **IC** in general. The transformation of constraints takes both the conceptual structure and its internal counterpart as its input. In the rest of section 5 the transformation of information structures is discussed.

5.2 Some preliminary issues

When transforming information structures, for all possible constructs of PSM we have to define their counterparts in F-logic. Basically, an information structure is mapped to F-logic class definitions, i.e. a set of object molecules with signature expressions only. For simplicity, we do not deal with assigning concrete names to the obtained F-logic components. Instead, an abstract notational convention is used, indicating the kind of an F-logic component and the components of the information structure it resulted from. For example, a class is denoted by C_X, where X contains the PSM component(s) to which the class corresponds.

In F-logic no difference is made between attributes and methods of classes. An attribute is considered to be a method without arguments. This enables the uniform treatment of (stored or derived) attributes and general methods. However, in object-oriented database systems a clear distinction between them is often made. This will be respected during our transformation, which is important also because, due to the elimination of certain kinds of redundancy during information analysis, a conceptual (PSM) schema represents data to be stored. From now on, by attributes we mean *stored* attributes. The distinction between (stored) attributes and general methods, however, is made only on syntactic (notational) level, thus preserving their uniform treatment in F-logic. Attributes and methods will be denoted in the form of $Attr_X$ and $Meth_X$, respectively,

For certain kinds of PSM components the translation is quite straightforward, but for others several alternatives are possible. The basic PSM components to be transformed are: object types, predicators, specialisation and generalisation hierarchies. The notion of object types in PSM is very general, it covers a number of more specific concepts, such as entity types, label types, fact types, set types, sequence types. Below we discuss the transformation of different PSM constructs separately. Some (simple) solutions have counterparts presented in [6] and [21], where the transformation is discussed in terms of ER and BRM schemas, respectively. Transformation alternatives presented in the rest of this section is discussed by means of abstract figures. Figure 6 shows how arrows in such figures are interpreted.

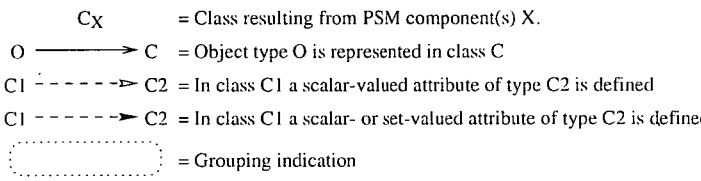

Figure 6: Graphical notation for specifying transformation alternatives

5.3 Entity types

By default, entity types are translated to F-logic classes. The corresponding class *definitions* are object molecules with signature expressions only. As an initial step, an empty object molecule is defined for each entity type. The structure of the class corresponding to an entity type depends on how the fact types in which the entity type is involved are transformed.

Figure 7: Entity types are mapped to classes

Sometimes an entity type simply models values of a label type connected with

it (see e.g. entity type *Duration* in figure 3). In such cases the elimination of the counterpart class of the entity type is reasonable. Then the corresponding class is substituted with the concrete domain of the connected label type.

5.4 Label types and bridge types

A label type represents a set of simple values from a concrete domain. For simplicity, we assume that each conrete domain has a counterpart class built-in F-logic. Since the simple values represented by label types do not have independent existence and may only occur as part of more complex objects, for a label type L, in contrast with entity types, no separate F-logic class is created. It is mapped to the (assumed) built-in F-logic class that corresponds to its concrete domain $\mathsf{Dom}(L)$.

A bridge type is a special binary fact type connecting a non-label type with a label type. Assuming that the involved non-label type is mapped to a class, a bridge type is incorporated as an attribute in that class. If a uniqueness constraint is specified on the predicator with the non-label type as its base, then the attribute is scalar-valued. Otherwise, it is defined to be set-valued. Situations when our assumption does not hold will be mentioned and treated places where they may arise. The translation of bridge types and the related label types is illustrated in figure 8.

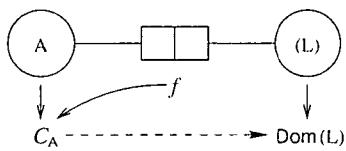

Figure 8: Mapping of bridge types and label types

5.5 Fact types and predicators

In this section we consider non-bridge fact types, when discussing the mapping of fact types into F-logic. The transformation of fact types is not straightforward, there are several possibilities how to transform them and their predicators. The translation of a fact type has a strong effect on the final F-logic counterparts of the object types involved in the fact type via predicators. Alternatives are discussed below.

5.5.1 Trivial mapping

The simplest solution is generating a separate F-logic (relation) class for each fact type. Then each predicator of a fact type results in a scalar-valued attribute in the corresponding class. Figure 9 illustrates this trivial translation.

This transformation is valid only if the base object type of each predicator in f has a corresponding F-logic class. However, this is not necessarily the case, e.g. if

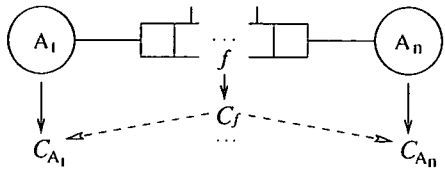

Figure 9: Illustration of trivial mapping

a base object type is a power type. Such cases will be discussed and treated later at the appropriate places.

Since the population of a fact type is a set of tuples, that are in turn *total* functions, it has to be ensured that each attribute of a fact (relationship) object carries some value. This can (has to) be forced by introducing appropriate constraints. On the other hand, fact objects are value-based and are identified by the participating objects. Therefore, a constraint has to be generated requiring that no two different fact objects may correspond to the same combination of participating objects (see also [6]).

Although the trivial mapping is sufficient to store all data, in order to improve query performance additional inverse attributes can be defined in any/all of the classes that correspond to the participating object types. Then such an inverse attribute stores references to relationship objects in which they participate (see also [21]). The type of inverse attributes is influenced by uniqueness constraints. Moreover, if inverse attributes are introduced, then also inverse constraints have to be generated to guarantee integrity.

Introducing inverse attributes with appropriate inverse constraints is a general option in the transformation. In principle, they can be generated in all alternatives that will be discussed for the transformation of fact types. In the sequel we will not explicitly mention the possibility again and again.

5.5.2 Incorporation of fact types

Fact types can be translated in such a way that they are incorporated in classes obtained for their object types, e.g. classes for entity types (see also [21]). In this case fact types are represented as reference attributes in such classes. More precisely, those attributes correspond to predicators constituting fact types.

Binary fact types

First we consider binary fact types as subjects of incorporation. As shown infigure 10, a binary fact type f can be incorporated in any/both of the two classes corresponding to its two base object types. The actual kind of reference attributes (scalar-valued or set-valued) is guided by uniqueness constraints. If f is incorporated in both classes, then an inverse constraint is needed as well.

Note that this way of transforming binary fact types looks very much like the

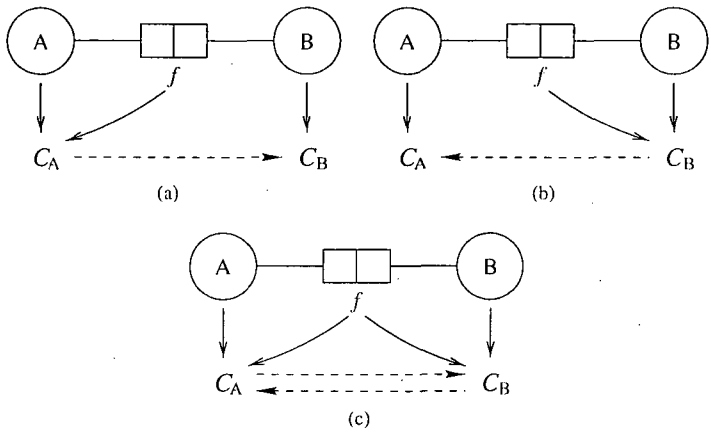

Figure 10: Incorporation of binary fact types

transformation of bridge types (see section 5.4), which is not surprising, since bridge types are always incorporated in the classes for the non-label types involved.

Relational view of incorporation

For binary fact types the incorporation mechanism can be combined with the trivial mapping as follows. A class is introduced for the fact type f similarly to the case of trivial mapping. At the same time, however, one of the two base object types is chosen, around which the related objects are arranged, similarly to the case of incorporation. Then the predicator with the "central" base object type becomes a scalar-valued attribute in the class corresponding to the fact type. The other predicator becomes either a set-valued or a scalar-valued attribute. This new alternative is depicted in figure 11.

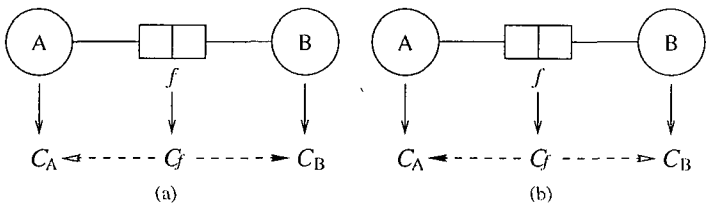

Figure 11: Relational incoporation of binary fact types

Note that if a uniqueness constraint is specified on the predicator with the "central" base object type, then the result is indentical to that of the trivial mapping. The combination of trivial mapping with incorporation can be viewed as a nest operation performed on the result of the trivial mapping. The basic constraints mentioned there are also needed here, but they have to be adapted according to

the different structure.

We note that basically this alternative has relational nature. Since constructs from object-orientation are involved as well (set-valued attributes) it fits in the object-relational approach.

Fact types of higher degree

An n-ary fact type f ($n > 2$) can be incorporated in a class corresponding to a base object type of a predicator in f, for which a uniqueness constraint is specified (provided that each object type involved in f has a counterpart class, for now it is assumed). The situation is depicted in firgure 12.

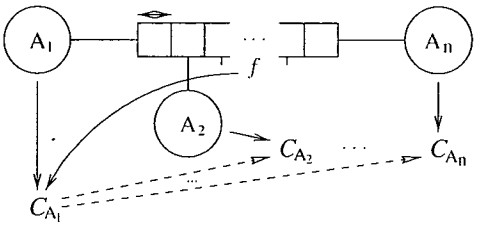

Figure 12: Incorporation of n-ary fact types

As opposed to the binary case, here the uniqueness constraint is required, because without that the concrete relationships between objects cannot be stored unambigously. The result of this transformation is that a class in which f is incorporated will have scalar-valued attributes for referencing related objects. The basic constraints discussed for trivial mapping have to be defined conforming this structure. If f is incorporated in more than one class, then inverse constraints are needed as well.

For the transformation of binary fact types we considered the combination of trivial mapping and incorporation. Since to incorporate fact types of higher degree uniqueness constraints over single predicators are required, we would get back exactly the trivial mapping when applying such a combination. Therefore, this alternative is not considered at all.

Quasi-incorporation

Although our latest note is valid, for n-ary fact types, where $n > 2$, a way to combine incorporation with the relational view can be recognized. The mechanism is slightly different from what we had for binary fact types. It is shown in figure 13. Fact type f is represented in a class for one of its base object types as well as in a subsidiary class denoted by $C_{f'}$. The class C_{A_1} in which f is incorporated has an attribute (scalar- or set-valued) for referencing objects of class $C_{f'}$. Those objects represent combinations of other objects that are related to a given C_{A_1} object via f. The attribute in C_{A_1} is scalar-valued if uniqueness constraint is on

the predicator with base A_1, otherwise set-valued. As usual, structure conforming basic constraints are needed.

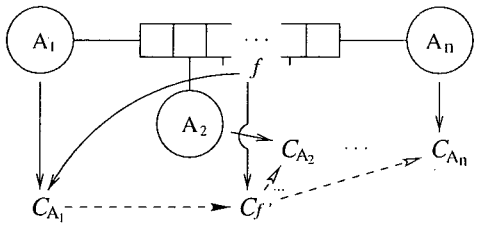

Figure 13: Illustration of quasi-incorporation

Note that, in contrast with pure incorporation of n-ary fact types, here no uniqueness constraint has been required. That is, this kind of transformation is more generally applicable. Note furthermore that quasi-incorporation of binary fact does not make sense, since it would produce a subsidiary class, as an unnecessary extra shell, with a single attribute.

5.5.3 Grouping

Since during information analysis fact types are determined such that they represent elementary recording types ([26]), most of them are binary or ternary in practice. That is, applying e.g. the trivial transformation would result in small but many object classes. Instead of transforming each fact type to a separate class, another alternative is to perform some grouping of fact types that are joinable via common object types before generating F-logic class definitions. Then for fact types in the same group a single class is created.

The grouping mechanism is implemented by means of a so-called *grouping profile*. A grouping *profile* is a set of groupings, where a *grouping* is a set of sets of predicators satisfying certain wellformedness conditions, e.g. conditions that require the joinability (predicators in a set have the same object type as their base) and connectivity of fact types to be grouped.

One particular grouping in the grouping profile implicitly specifies a set of fact types to be grouped together. The result of grouping is that one class is generated for the fact types in one group, and all those fact types are represented in that class. The structure of the class is obtained in such a way that one (scalar-vaued) attribute is defined for each set of predicators (corresponding to one object type) in the grouping and one (scalar-vaued) attribute is defined for each non-grouping predicator occuring in the grouped fact types.

The grouping meachnism is illustrated in figure 14. According to the figure, the grouping profile consists of a single (maximal) grouping. That grouping contains two sets of predicators (with common object types).

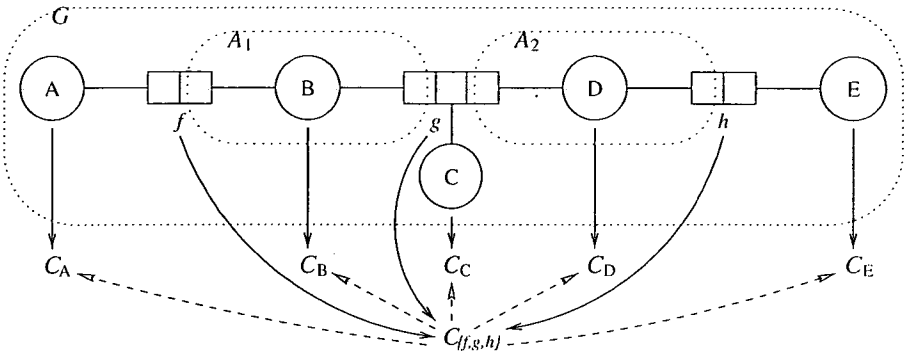

Figure 14: Illustration of the grouping mechanism

5.5.4 OR-like grouping

In section 5.5.2 we discussed the incorporation of fact types in classes corresponding to base object types. For binary fact types the relational variant, resulting from the combination of trivial mapping and incorporation, was also considered. In this section we show how this combined alternative can be further integrated with a restricted version of the grouping mechanism.

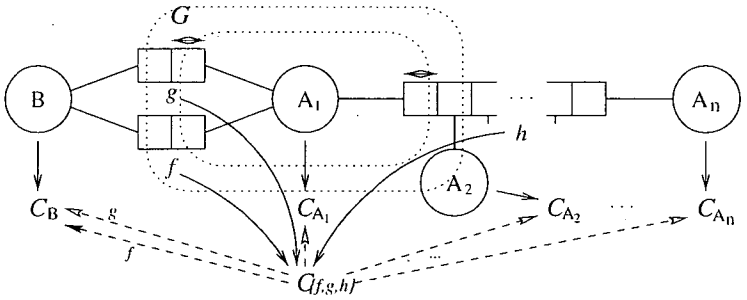

Figure 15: OR-like grouping

The situation is depicted in figure 15. The restrictions on the general grouping (having flat behavior) means that a particular grouping in the grouping profile may consist of only a single set of predicators with common base object type. (Note the difference with figure 14 with respect to the grouping indication.) Similarly to the relational incorporation for binary fact types (see figure 11), a central object type is chosen around which related data coming from several (grouped) fact types is arranged. This gives some OO characteristic to this transformation, while keeping also the relational view of grouping. From the alternatives discussed so far the general idea illustrated by figure 15 can be seen.

Grouping with quasi-incorporation

In figure 13 an alternative way to incorporate fact types of higher degree (called quasi-incorporation) with some relational characteristic was illustrated. There subsidiary classes have been introduced. That mechanism can be combined with (restricted) grouping analogously with the combination presented in the previous section, as shown in figure 16. Since only alternatives discussed earlier are combined in a way that has also been described, we do not explain this combination in more detail.

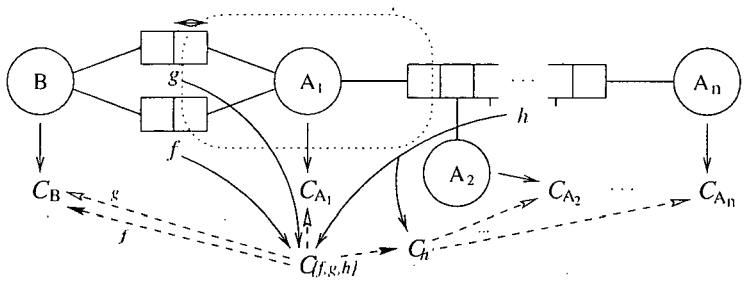

Figure 16: Grouping with quasi-incorporation

5.5.5 Objectification

As said in section 3, a fact type is objectified if it is the base object type of some predicator. The transformation of objectified fact types must be examined with some special care, since it has to be ensured that data attached to facts (and not e.g. to entities) are storable as well. Now we consider how the alternatives for the transformation of fact types are applicable to objectified fact types.

Objectification and trivial mapping

Obviously, the trivial mapping method can be applied to an objectified fact type as to any fact type without problems, which is the most natural solution. During the transformation of the rest of the information structure any of the alternatives can be chosen. Then the (objectified) fact type behaves as if it was an entity type.

Objectification and grouping

In principle, an objectified fact type can be considered for grouping, but not as it stands. In order to be able to transform the other fact types it is involved in, first it has to be unnested with respect to all those fact types. In many cases, however, the same final result can be obtained by applying different transformation alternatives. Therefore, we do not deal with providing an unnest operation for objectifications and do not consider objectified fact types to be grouped in any way.

Incorporation of objectified fact types

In section 5.5.2 the alternative of incorporating fact types in classes for their base object types was discussed. Binary fact types can always be incorporated, but fact types of higher degree can be considered for incorporation at presence of uniqueness constraints over single predicators. Incorporation of objectified fact types is a meaningful option in even more restrictive situations only. The rationale behind this is that beside representation of (objectified) facts, we also have to deal with the representability of facts attached to (objectified) facts.

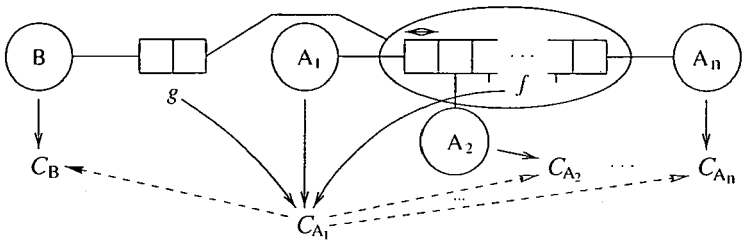

Figure 17: Incorporation of objectified fact type

The mechanism of incorporating objectified fact types is illustrated in figure 17. It shows that if an objectified fact type f is incorporated in the counterpart class of one of its base object types, then the fact types in which f participates are also incorporated in the same class. The unambigous representability is ensured by the required uniqueness constraint (also when f is binary).

Obviously, this transformation requires the fact types with base object type f to fulfill the condition for incorporation. If for any of those fact types this precondition is not satisfied, then f cannot be incorporated either. This leads to a situation that can be captured by recursive checking and evaluation.

At some earlier points of this paper we assumed the existence of classes for base object types when describing transformation alternatives for fact types (also bridge types). We also promised to highlight and treat situations when it's not the case. Now we reached such a situation, because the incorporation mechanism is applied to fact types with base f, but no class for f exists. The special treatment here is that we explicitly prescribed the class in which the incorporation has to be performed instead of the class for f. Note, however, that during the incorporation of f the existence of classes for its base object types is still an assumption (precondition).

The relational-like incorporation for binary fact types (see figure 11) is generally not applicaple to objectified binary fact types, only when the trivial mapping is obtained back due to the required uniqueness constraint. Otherwise the problem of unambigous representability arises.

For similar reason, quasi-incorporation of objectified fact types is only possible at the presence of appropriate uniqueness constraints. The illustrating figure is obtained as a combination of figures 13 and 17 and is presented in figure 18 without further explanation.

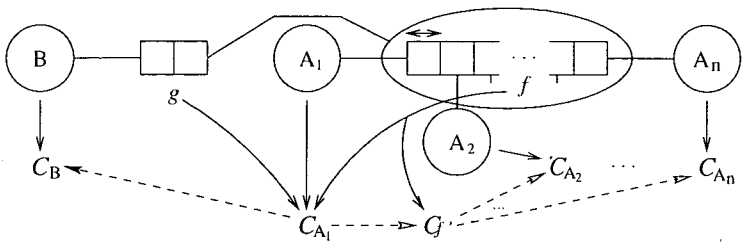

Figure 18: Quasi-incorporation of objectified fact type

5.6 Power types

Power types in PSM represent sets. An instance of a power type is a (non-empty) set of instances of its element type and is identified by its elements. Since in F-logic we have the concept of set-valued attribute, at first sight it might seem that power types can be represented simply by means of set-valued attributes.

However, set-valued attributes have limited applicability. Nested power types (power types of power types) cannot be expressed in F-logic in terms of set-valued attributes. For instance, in our example in figure 3 if budgets belong to groups of groups of projects, then that situation cannot be represented by set-valued attributes. Therefore, for the transformation of power types the parametric class $set(C)$ (discussed in section 4), where the parameter C can be an arbitrary class, will be used.

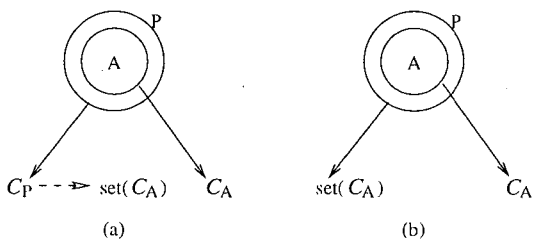

Figure 19: Alternatives for power types

A power type can be transformed such that it is simply represented as a $set(...)$ type wherever it occurs. An alternative solution is that a separate class in defined for the power type with one attribute containing the set elements and possibly other attributes holding references to related objects. The choice depends on its context and/or the designer's preference. This consideration, however, suggests that the transformation of power types with or without defining counterpart classes can be both reasonable solutions, which is shown in figure 19. Our manner of transforming a power type P covers both alternatives in a uniform way. The procedure is as follows:

1. First, as an initial step, a class C_P is introduced with a single scalar-valued attribute $Attr_{elements}$ of type $set(C_{\mathsf{Elt}(P)})$ for containing the set elements.
2. During the transformation of the information structure consider P as if it was an entity type (remember that entity types are always mapped to classes).
3. When the whole information structure has been translated, C_P will or will not contain attributes other than $Attr_{elements}$. If C_P contains no attribute other than $Attr_{elements}$, then optionally the class C_P can be eliminated by substituting it with $set(C_{\mathsf{Elt}(P)})$ where it occurs.

As it can be seen, the translation of a power type requires its element type (not necessarily entity type, it can be e.g. a fact type) to be mapped to a class. To take this into account, it is required that if a fact type is the element type of a power type, then the fact type has to result in a class, i.e. it has to be transformed according to the trivial mapping.

Furthermore, note that although at the end power types do not necessarily result in classes, the assumption, made at several places before, that corresponding classes for participating object types exist when fact types are mapped (see section 5.5) is not violated, because the elimination of the counterpart class for a power type may be performed as a final step only.

5.7 Sequence types

The transformation of sequence types into F-logic is analogous to that of power types, see figure 20. The only differences are that (1) instead of the parametric class $set(C)$ the parametric class $list(C)$ is used, and (2) the implicit attribute is denoted by $Attr_{sequence}$ instead of $Attr_{elements}$. In other aspects the procedure is identical, therefore is not further detailed here. Moreover, similar considerations and notes are valid for the mapping of sequence types what we had for the transformation of power types.

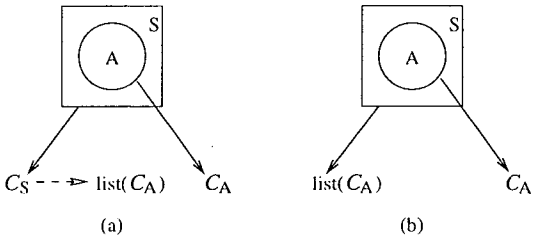

Figure 20: Alternatives for sequence types

5.8 Specialisation and generalisation

Specialisation and generalisation hierarchies of PSM models are translated to subclass hierarchies in F-logic. Both $A\,\mathsf{Spec}\,B$ and $B\,\mathsf{Gen}\,A$ result in $C_A :: C_B$. (C_A is

subclass of C_B). As a consequence, participating object types have to be mapped to classes. Taking the restrictions on Spec and Gen (see section 3) into account this means that: (1) If B is a fact type, then it has to be translated according to trivial mapping. (2) If B is either a power type or a sequence type, then its corresponding class cannot be eliminated.

So far, specialisation and generalisation were treated in the same way. However, they are different concepts, and the difference has to be reflected on F-logic level as well. As mentioned in section 3, the difference lies in the way subtypes and generalised object types get their identification, and in the way their population is derived. A generalised object type inherits identification from its specifiers and its population is the union of the populations of its specifiers. Therefore, in an F-logic class hierarchy that corresponds to a generalisation hierarchy non-leaf level classes may have only object instances that belong to some leaf level class. This can be achieved by introducing appropriate constraints.

A subtype inherits identification from its supertype and its population is a subset of the population of its subtype and is derived by means of a subtype defining rule. When mapping a subtype relationship, the associated subtype defining rule has be translated too, resulting in a rule in the corresponding F-program. The body of the rule is the subtype defining rule translated into F-logic, while the head is an is-a assertion specifying class membership.

5.9 Methodization

In general, an attribute of a class carries direct reference(s) to related object(s), which means that by means of attributes (methods without arguments) only relationships between two objects can be captured. However, it would often be useful to have a device to enable that objects related to a given combination of other objects can be obtained. This can be achieved by defining general methods (method with arguments). A mechanism, called *methodization*, can be introduced for this purpose as a complementary device in order to provide efficient access (querying) of data stored in attributes of classes.

For example, provided that a fact type is mapped to a class, facts (relationships) become stored objects. The attributes of a relationship object contain references to objects that are in relationship. If now we want to know what objects of a given class are in relationship with some combination of objects of some classes, we have to perform an appropriate query. Clearly, it can be very useful to define access *methods* to make querying (traversing relationships) easier and more efficient.

The scale of choices for methods is quite wide. The signatures of methods are strongly influenced by uniqueness constraints. The behavior of methods can be defined in terms of F-logic rules. In [6] a similar technique is introduced, which is called *pivoting*.

Inverse attributes vs methodization

It was said earlier that inverse attributes can be optionally defined beside attributes needed for sufficient storage of data. It improves query performance, but makes update more complex. To ensure integrity, introducing inverse attributes must be done along with generating inverse constraints.

Alternatively, stored inverse attributes can be substituted with *methods* without arguments. In this case the inverse constraints needed for the inverse attributes are converted to rules in the F-program. Those rules define the behavior of the (inverse) methods.

6 Elaborated example

Until now we discussed possible ways to transform conceptual structures to OO database schemas. In this section we transform the PSM schema in figure 3 into F-logic. During this mapping object molecules defining only parts of classes are obtained often. Since, according to the semantics of F-logic, they can be unified to get equivalent more complex object molecules, at the end we will also present the unified result.

6.1 Declarations

Entity types are mapped to classes. As an initial step, for each entity type a an empty object molecule is generated, i.e. we obtain:

$$C_{Equipment}[] \quad C_{Car}[] \quad C_{PC}[] \quad C_{Person}[] \quad C_{Manager}[]$$
$$C_{Coworker}[] \quad C_{Dept.}[] \quad C_{Building}[] \quad C_{Project}[] \quad C_{Article}[]$$
$$C_{Company}[] \quad C_{Activity}[] \quad C_{Duration}[] \quad C_{Amount\_of\_money}[]$$

Furthermore, in the initial step a class is created for each power type and sequence type with a single attribute for containing the set and list elements, respectively:

$$C_{Pr\_group}[\,Attr_{elements} \Rightarrow set(C_{Project})\,]$$
$$C_{Daily\_activities}[\,Attr_{sequence} \Rightarrow list(C_{Activity})\,]$$

Specialisation and generalisation relationship result in subclass relationships:

$$C_{Manager} :: C_{Person} \quad C_{Coworker} :: C_{Person}$$
$$C_{Car} :: C_{Equipment} \quad C_{PC} :: C_{Equipment}$$

Bridge types are incorporated in classes obtained for the involved non-label type. Suppose that with the occuring label types the following concrete domains,

that are assumed to be built-in classes in F-logic, are associated:

$$\mathsf{Dom}(Date) = Date$$
$$\mathsf{Dom}(Dollars) = \mathsf{Dom}(Hours) = Integer$$
$$\mathsf{Dom}(Reg\_nr) = \mathsf{Dom}(PC\_nr) = \mathsf{Dom}(P\_id) = \mathsf{Dom}(P\_name) = String$$
$$\mathsf{Dom}(Pr\_name) = \mathsf{Dom}(D\_name) = \mathsf{Dom}(B\_code) = \mathsf{Dom}(A\_code) = String$$
$$\mathsf{Dom}(C\_code) = \mathsf{Dom}(C\_name) = \mathsf{Dom}(Tel\_nr) = \mathsf{Dom}(A\_descr) = String$$

For identifying bridge types the predicator with non-label base object type X is referred to as has_X. Then the mapping of bridge types yields the following object molecules:

$$C_{Person}[Attr_{has\_P\_id} \Rightarrow String] \qquad C_{Person}[Attr_{has\_Pname} \Rightarrow String]$$
$$C_{Equipment}[Attr_{bought\_on} \Rightarrow Date] \qquad C_{Car}[Attr_{has\_Reg\_no} \Rightarrow String]$$
$$C_{PC}[Attr_{has\_PC\_nr} \Rightarrow String] \qquad C_{Project}[Attr_{has\_Pr\_name} \Rightarrow String]$$
$$C_{Dept.}[Attr_{has\_D\_name} \Rightarrow String] \qquad C_{Building}[Attr_{has\_B\_code} \Rightarrow String]$$
$$C_{Article}[Attr_{has\_A\_code} \Rightarrow String] \qquad C_{Activity}[Attr_{has\_A\_descr} \Rightarrow String]$$
$$C_{Duration}[Attr_{has\_Hour} \Rightarrow Integer] \qquad C_{Am.\_of\_m.}[Attr_{has\_Dollars} \Rightarrow Integer]$$
$$C_{Company}[Attr_{has\_C\_code} \Rightarrow String] \qquad C_{Company}[Attr_{has\_Cname} \Rightarrow String]$$
$$C_{Company}[Attr_{has\_Telnr} \Rightarrow\!\!\!\!\Rightarrow String]$$

Finally, we transform (non-bridge) fact types. In section 5.5 a number of alternatives were discussed. We transform the fact types of figure 3 such that we cover as many alternatives as reasonable according to the complexity of the input schema. In order to identify chosen alternatives unambigously we always refer to the corresponding figures. The specified uniqueness constraints, of course, are taken into account.

The objectified fact type *Coworkership* is mapped according to the trivial mapping (figure 9), which results in:

$$C_{Coworkership}[Attr_{works\_for} \Rightarrow C_{Person};\ Attr_{has\_as\_coworker} \Rightarrow C_{Project}]$$

The binary fact type *Cow_dur* is incorporated (figure 10) in the class obtained for the base object type of its predicator *has_as_duration* yielding:

$$C_{Coworkership}[Attr_{has\_as\_duration} \Rightarrow C_{Duration}]$$

Incorporation of binary fact types in one of the two classes corresponding to their base object types is applied to other fact types as well. Fact types *Salary*, *Activities* and *Budget* are incorporated in classes $C_{Coworker}$, $C_{Manager}$ and C_{Pr_group}, respectively. The following object molecules are generated:

$$C_{Coworker}[Attr_{earns} \Rightarrow C_{Amount\_of\_money}]$$
$$C_{Manager}[Attr_{performes} \Rightarrow C_{Daily\_activities}]$$
$$C_{Pr\_group}[Attr_{may\_spend} \Rightarrow C_{Amount\_of\_money}]$$

Fact type *Car_usage* is incorporated in both classes corresponding to the involved object types (figure 10, alternative (c)), namely in C_{Car} and $C_{Manager}$. Two object molecules are obtained:

$$C_{Car}\,[\,Attr_{used\_by} \Rightarrow C_{Manager}\,]$$
$$C_{Manager}\,[\,Attr_{uses} \Rightarrow C_{Car}\,]$$

In figure 11 a relational-like way of incorporating binary fact types, as the combination of trivial mapping and incorporation (alternatively seen as nesting on the result of trivial mapping), has been shown. Fact type *Location* is mapped according to this way of transformation. Object type *Dept.* is chosen as the central object type. This results in:

$$C_{Location}\,[\,Attr_{is\_located\_at} \Rightarrow C_{Dept.};\ Attr_{accomodates} \Rightarrow\!\!\!\!\Rightarrow C_{Building}\,]$$

To illustrate the alternative coming from the combination of incorporation (OO nature) with restricted grouping (relational nature), fact types *Employment* and *PC_usage* with common object type *Person* are grouped according to the OR-like grouping shown in figure 15. The following single class (object molecule) is generated:

$$C_{\{Employment,\,PC\_usage\}}\,[\,Attr_{\{employed\_by,\,works\_on\}} \Rightarrow C_{Person};$$
$$Attr_{employs} \Rightarrow C_{Dept.};\ Attr_{belongs\_to} \Rightarrow\!\!\!\!\Rightarrow C_{PC}\,]$$

The combination of OR-like grouping and quasi-incorporation has been depicted in Figure 16. To set an example for this mechanism, fact types *Management* and *Supply* are grouped together via object type *Project* such that for fact type *Supply* a subsidiary class $C_{Supply'}$ is introduced. Beside this subsidiary class, a relation class is also defined representing fact type *Management* and partially fact type *Supply*. The obtained object molecules are the following:

$$C_{Supply'}\,[\,Attr_{supplies} \Rightarrow C_{Company};\ Attr_{is\_supplied} \Rightarrow C_{Article}\,]$$
$$C_{\{Management,\,Supply\}}\,[\,Attr_{\{managed\_by,\,receives\}} \Rightarrow C_{Project};$$
$$Attr_{manages} \Rightarrow\!\!\!\!\Rightarrow C_{Person};\ Attr_{receives} \Rightarrow\!\!\!\!\Rightarrow C_{Supply'}\,]$$

By now, each fact type of figure 3 has been translated. We did not exploit every particular alternative discussed in section 5.5 coming from different kinds of combinations. However, all the basic building blocks used in some combination, such as trivial mapping, incorporation, quasi-incorporation, grouping, have been covered.

Class elimination

After completing the transformation of the PSM structure of figure 3, the elimination of classes for power types and sequence types can be considered (see sections 5.6 and 5.7). Since the class C_{Pr_group} contains an attribute other than $Attr_{elements}$,

it cannot be eliminated. The elimination of class $C_{Daily\_activities}$, however, is reasonable. The class is substituted with $list(C_{Activity})$ and is removed.

As mentioned in section 5.3, classes for entity types with label type nature can also be eliminated. In our example such classes are $C_{Duration}$ and $C_{Amount\_of\_money}$ at least. They are, therefore, eliminated. Occurrences are replaced with the built-in classes for the concrete domains of their corresponding label types.

Unified declarations

Due to the fact that in some cases above different parts of class definitions were obtained at different points of the transformation, some classes are defined by means of more than one object molecule. The separate parts can now be put together as presented below. The class eliminations above are taken into account. Clearly, the input schema does not cover all the details of the application domain, which lead to simple (entity) classes in many cases. Those classes likely have additional attributes, which is also indicated below. During the above transformation we used denotations rather than names for classes and attributes. In order to make the example more readable we also give names to classes and attributes now.

$Person\,[\,P\_id \Rightarrow String;\ name \Rightarrow String; \ldots\,]$

$Manager :: Person$

$Manager\,[\,daily\_activities \Rightarrow list(Activity);\ uses\_car \Rightarrow Car; \ldots\,]$

$Coworker :: Person$

$Coworker\,[\,salary \Rightarrow Integer; \ldots\,]$

$Equipment\,[\,bought\_on \Rightarrow Date; \ldots\,]$

$Car :: Equipment$

$Car\,[\,reg\_nr \Rightarrow String;\ used\_by \Rightarrow Manager; \ldots\,]$

$PC :: Equipment$

$PC\,[\,pc\_nr \Rightarrow String; \ldots\,]$

$Project\,[\,title \Rightarrow String; \ldots\,]$

$Department\,[\,name \Rightarrow String; \ldots\,]$

$Building\,[\,B\_code \Rightarrow String; \ldots\,]$

$Article\,[\,A\_code \Rightarrow String; \ldots\,]$

$Activity\,[\,description \Rightarrow String; \ldots\,]$

$Company\,[\,C\_code \Rightarrow String;\ name \Rightarrow String;\ tel\_nrs \Rightarrow\!\!> String; \ldots\,]$

$Pr\_group\,[\,projects \Rightarrow set(Project);\ budget \Rightarrow Integer; \ldots\,]$

$Coworkership\,[\,employee \Rightarrow Person;\ project \Rightarrow Project;\ duration \Rightarrow Integer\,]$

$Dept\_loc[dept \Rightarrow Department;\ building \Rightarrow\!\!> Building]$

$Employment[employee \Rightarrow Person; dept \Rightarrow Department; works\_on \Rightarrow\!\!> PC]$

$Supply[supplier \Rightarrow Company;\ article \Rightarrow Article]$

$Project\_rel[project \Rightarrow Project;\ managers \Rightarrow\!\!> Person;\ receives \Rightarrow\!\!> Supply]$

6.2 Constraints

The result of the transformation of the information structure in figure 3 was influenced by simple conceptual uniqueness constraints. On the other hand, in addition to class definitions, the structure transformation results in some kinds of basic constraints in F-logic, such as key, mandatory and inverse constraints. In this section we provide these constraints for our example.

Uniqueness constraints

F-logic uniqueness (key) constraints are obtained in two ways. Firstly, from the unique representation of facts (relationships) (see also [6]). Secondly, the simple conceptual uniqueness constraints in figure 3 are translated as well. For uniqueness constraints the macro "!– Key$(C, \{A_1, ..., A_n\})$" is defined as follows:

$$!- X \doteq Y \longleftarrow X:C[A_1 \rightarrow(\rightarrow\!\!>) R_1; ...; A_n \rightarrow(\rightarrow\!\!>) R_n] \wedge Y:C[A_1 \rightarrow(\rightarrow\!\!>) R_1; ...; A_n \rightarrow(\rightarrow\!\!>) R_n]$$

The notation $A_i \rightarrow(\rightarrow\!\!>) R_i$ means that if A_i is scalar-valued, then $\rightarrow$ is used, otherwise $\rightarrow\!\!>$. In our example the following F-logic uniqueness constraints are generated:

!– Key$(Person, \{P_id\})$
!– Key$(Car, \{reg_nr\})$
!– Key$(PC, \{pc_nr\})$
!– Key$(Department, \{name\})$
!– Key$(Article, \{A_code\})$
!– Key$(Company, \{C_code\})$
!– Key$(Cow.ship, \{employee, project\})$
!– Key$(Employment, \{employee\})$
!– Key$(Supply, \{supplier, article\})$
!– Key$(Manager, \{uses_car\})$
!– Key$(Car, \{used_by\})$
!– Key$(Project, \{title\})$
!– Key$(Building, \{B_code\})$
!– Key$(Activity, \{description\})$
!– Key$(Pr_group, \{projects\})$
!– Key$(Dept_loc, \{dept\})$
!– Key$(Employment, \{works_on\})$
!– Key$(Project_rel, \{project\})$

Mandatory constraints

Since the population of fact types consists of total functions, it has to be ensured that if a class corresponds to a fact type, then each attribute of a member of that class (fact object) carries some value. Again, a general macro

"!– Mandatory$(C, \{A_1, ..., A_n\})$" is introduced to serve as a shorthand for the following:

$$!- (\exists Y)X\,[\,A_1 \rightarrow(\twoheadrightarrow)\,Y\,] \longleftarrow X:C$$
$$\ldots$$
$$!- (\exists Y)X\,[\,A_n \rightarrow(\twoheadrightarrow)\,Y\,] \longleftarrow X:C$$

In the case of our example the following mandatory constraints are necessary:

$$!- \text{Mandatory}(Coworkership, \{employee, project\})$$
$$!- \text{Mandatory}(Dept\_loc, \{dept, building\})$$
$$!- \text{Mandatory}(Supply, \{supplier, article\})$$

Inverse constraints

When two attributes are considered to be inverses of each other, inverse constraints have to be defined. According to the possible kinds of relationship types between two classes we introduce the following three macros:

$$!- \text{Inverse1-1}(C_1, A_1, C_2, A_2) \stackrel{\text{def}}{=} \begin{array}{l} !- Y:C_2\,[\,A_2 \rightarrow X\,] \longleftarrow X:C_1\,[\,A_1 \rightarrow Y\,] \\ !- Y:C_1\,[\,A_1 \rightarrow X\,] \longleftarrow X:C_2\,[\,A_2 \rightarrow Y\,] \end{array}$$

$$!- \text{Inverse1-N}(C_1, A_1, C_2, A_2) \stackrel{\text{def}}{=} \begin{array}{l} !- Y:C_2\,[\,A_2 \rightarrow X\,] \longleftarrow X:C_1\,[\,A_1 \twoheadrightarrow Y\,] \\ !- Y:C_1\,[\,A_1 \twoheadrightarrow X\,] \longleftarrow X:C_2\,[\,A_2 \rightarrow Y\,] \end{array}$$

$$!- \text{InverseM-N}(C_1, A_1, C_2, A_2) \stackrel{\text{def}}{=} \begin{array}{l} !- Y:C_2\,[\,A_2 \twoheadrightarrow X\,] \longleftarrow X:C_1\,[\,A_1 \twoheadrightarrow Y\,] \\ !- Y:C_1\,[\,A_1 \twoheadrightarrow X\,] \longleftarrow X:C_2\,[\,A_2 \twoheadrightarrow Y\,] \end{array}$$

In our example fact type *Car_usage* has been incorporated in both classes obtained for the two involved entity types, namely in classes *Manager* and *Car*, yielding one attribute in each being inverses of each other. Therefore, an inverse constraint is also required as follows:

$$!- \text{Inverse1-1}(Manager, uses\_car, Car, used\_by)$$

Effects of specialisation and generalisation

The population of subtypes (specialisation) is defined by subtype defining rule. This mechanism has to be translated into F-logic. In our example we have *Manager* Spec *Person* and *Coworker* Spec *Person*. Let $\Psi_{Manager}$ and $\Psi_{Coworker}$ denote the F-logic counterpart of the subtype defining rules for subtypes *Manager* and *Coworker*, respectively. Then we introduce the following two constraints:

$$!- X:Manager \longleftarrow X:Person \wedge \Psi_{Manager}(X)$$
$$!- X:Coworker \longleftarrow X:Person \wedge \Psi_{Coworker}(X)$$

The subtype defining rules are: A person is a manager if he/she plays the role *manages*. A person is a coworker if he/she plays the role *works_for*. It has effect on the final result of the translation of fact types *Management* and *Coworkership*. In class *Project_rel* the result type *Person* is replaced with *Manager*:

$$Project\_rel\,[\,project \Rightarrow Project;\ managers \Rightarrow\!\!> Manager;\ receives \Rightarrow\!\!> Supply\,]$$

In class *Coworkership* the result type *Person* is replaced with *Coworker*:

$$Cow.ship\,[\,employee \Rightarrow Coworker;\ project \Rightarrow Project;\ duration \Rightarrow Integer\,]$$

In case of generalisation it has to be ensured that every instance in the population of a generalised type belongs to the population of one of its specifiers. This can be expressed in terms of constraints on F-logic level. In our example we obtain:

$$!\!-\ (X : Car\ \vee\ X : PC) \longleftarrow X : Equipment$$

6.3 Inverse attributes and methods

As said in section 5.9, a wide range of introducing inverse attributes with inverse constraints as well as methods with the definition of their behavior is possible. To illustrate these general mechanisms, now we introduce an additional inverse attribute and a method. Class *Coworker* is augmented with attribute *involved_in* containing references to *Coworkership* objects in which a given person is involved. The following additional object molecule, that can be unified with the existing one for class *Coworker*, and inverse constraint are defined:

$$Coworker\,[\,involved\_in \Rightarrow\!\!> Coworkership\,]$$
$$!\!-\ \mathsf{Inverse1\text{-}N}(Coworker, involved\_in, Coworkership, employee)$$

As an example of an access method, the method *suppliers* is introduced in class *Project* returning the companies that supply a given article for the project on which the method is invoked. The object molecule defining the signature of the method is the following:

$$Project\,[\,suppliers\,@\,Article \Rightarrow\!\!> Company\,]$$

The semantics of the method is given by means of the following F-logic rule:

$$Y\,[\,suppliers\,@\,W \twoheadrightarrow V\,] \longleftarrow X : Project\_rel\,[\,project \rightarrow Y;\ receives \twoheadrightarrow Z\,] \wedge Z : Supply\,[\,supplier \rightarrow V;\ article \rightarrow W\,]$$

7 About the implementation step

The second step of the transformation (the implementation step, see figure 2) is the translation of intermediate models defined in terms of F-logic into a OO final target environment, e.g. SQL3 or ODMG-93. A sequence of DDL statements in the

corresponding database language has to be generated from F-logic specifications. Unlike in the design step, in the implementation step separate translations have to be defined for different target environments, where all the system specific details must be dealt with. This can be implemented by means of translation tables, containing all the system specific information (e.g. supported data types, correspondece between F-logic "built-in" classes and system specific data types) needed for the generation of DDL statements.

Since in the present paper the main focus is on the design step, we do not further elaborate on the implementation step. However, in the section below an example is given in terms of ODMG-93.

7.1 Example in ODMG-93

Next the F-logic schema in section 6 obtained for the PSM schema of figure 3 is translated into ODMG-93. Beside class declarations (including the additional inverse attribute and method), uniqueness and inverse constraints are translated. Specifying other constraints is not supported directly in ODMG-93, therefore they are not considered here. For the syntax and semantics of ODMG-93 we refer to [9].

```
interface Person
(   extent Persons
    keys P_id   ) : persistent
{   attribute string P_id ;
    attribute string name ; ...  } ;

interface Manager : Person
(   extent Managers
    keys uses_car   ) : persistent
{   relationship List<Activity> daily_activities ;
    relationship Car used_car
            inverse Car:: used_by ; ...  } ;

interface Coworker : Person
(   extent Coworkers   ) : persistent
{   attribute integer salary ;
    relationship Set<Coworkership> involved_in
            inverse Coworkership:: employee ; ...  } ;

interface Activity
(   extent Activities
    keys description   ) : persistent
```

```
{   attribute string description ; ... } ;

interface Project
(   extent Projects
    keys title   ) : persistent
{   attribute string title ;
    ...
    Set<Company> suppliers( in Article art ) ;   } ;

interface Article
(   extent Articles
    keys A_code   ) : persistent
{   attribute string A_code ; ... } ;

interface Company
(   extent Companies
    keys C_code   ) : persistent
{   attribute string C_code ;
    attribute string name ;
    attribute Set<string> tel_nrs ; ... } ;

interface Coworkership
(   extent Coworkerships
    keys (employee, project)   ) : persistent
{   attribute integer duration ;
    relationship Coworker employee
            inverse Coworker:: involved_in ;
    relationship Project project ;   } ;

interface Supply
(   extent Supplies
    keys (supplier, article)   ) : persistent
{   relationship Company supplier ;
    relationship Article article ;   } ;

interface Project_rel
(   extent Project_rels
```

```
    keys project ) : persistent
{   relationship Project project ;
    relationship Set<Manager> managers ;
    relationship Set<Supply> receives ;  } ;
```

8 Conclusions and further research

In this paper we dealt with the transformation of conceptual data models into database environments with object-oriented features, such as ODMG-93 and SQL3. In our approach this transformation is captured within the framework of a two level architecture. Conceptual models are first mapped to intermediate specifications (design step). Then the obtained intermediate specifications are translated into the database language of a given target database system (implementation step). For expressing conceptual models we used the object-role modelling technique PSM (Predicator Set Model), a formalized extension of NIAM. Intermediate specifications are expressed in terms of F-logic, a logic-based abstract specification language for object-oriented systems. The advantages of a two step tranformation mechanism have been discussed.

Here we focused on the first step of the overall transformation. A number of alternatives for the transformation of conceptual structures have been presented, resulting in a collection of design options. Such alternatives were discussed by means of illustrating figures. The mapping of information structures is often influenced by simple uniqueness constraints. Also, transforming structures often imply basic (e.g. key) integrity constraints in the target model to be generated. The treatment of these aspects has been incorporated. The transformation of a real life example conceptual schema into F-logic has been worked out in detail. Moreover, the obtained F-logic specification has been partially translated into ODMG-93, thus illustrating the applicability of the transformation process in practice.

For further research the most fundamental topic is the full formalization of the first (design) step of our transformation mechanism according to the formalisms of PSM and F-logic. Beside the mapping of information structures, the transformation of populations in a formal framework is also essential, since the semantics of an information structure is defined in terms of its possible populations. This is important in order to prove the correctness of the transformation formally. The general treatment of conceptual constraints, that are parts of conceptual schemas, and their translation are to be addressed. Furthermore, issues concerning the second (implementation) step of the overall transformation have to be worked out in more detail. Our present paper has set up the framework and provides the basis for a general automated transformation mechanism that covers all the aspects just desrcibed.

References

[1] S. Abiteboul, P.C. Fischer, and H.J. Schek. *Nested Relations and Complex Objects in Databases.* Springer-Verlag, Berlin, Germany, 1987.

[2] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–173, 1989.

[3] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S.B. Zdonik. The object-oriented database system manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD-89)*, pages 40–57, Kyoto, Japan, 1989. Elsevier Science Publishers.

[4] C. Beeri. A formal approach to object-oriented databases. *Data & Knowledge Engineering*, 5:353–382, 1990.

[5] A. Benczúr, Cs. Hajas, and Gy. Kovács. Datalog Extension for Nested Relations. *Computers Math. Applic.*, 30(12):51–79, 1995.

[6] J. Biskup, R. Menzel, and T. Polle. Transforming an Entity-Relationship Schema into Object-Oriented Database Schemas. In *Proceedings of the International Workshop on Advances in Databases and Information Systems*, pages 67–78, Moscow, June 1995.

[7] P. van Bommel, Gy. Kovács, and A. Micsik. Transformation of database populations and operations from the conceptual to the internal level. *Information Systems*, 19(2):175–191, 1994.

[8] P. van Bommel and Th.P. van der Weide. Reducing the search space for conceptual schema transformation. *Data & Knowledge Engineering*, 8:269–292, 1992.

[9] R.G.G. Catell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Francisco, California, 1994.

[10] P.P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[11] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[12] H. Darwen and C.J. Date. The Third Manifesto. *SIGMOD Record*, 24(1), March 1995.

[13] C.J. Date and H. Darwen. *A Guide to the SQL Standard.* Addison-Wesley, Reading, Massachusetts, 1992.

[14] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, 9(4):157–204, 1992.

[15] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base.* Publ. nr. ISO/TC97/SC5-N695, 1982.

[16] A.H.M. ter Hofstede. *Information Modelling in Data Intensive Domains.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[17] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489–523, October 1993.

[18] A.H.M. ter Hofstede and Th.P. van der Weide. Expressiveness in conceptual data modelling. *Data & Knowledge Engineering*, 10(1):65–100, February 1993.

[19] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.

[20] W. Kim. Object-Oriented Database Systems: Promises, Reality, and Future. In *Proceedings of the 19th VLDB Conference*, pages 676–687, Dublin, Ireland, 1993.

[21] Y. Kornatzky and P. Shoval. Conceptual design of object-oriented database schemas using the binary-relationship model. *Data & Knowledge Engineering*, 14:265–288, 1994.

[22] Gy. Kovács and P. van Bommel. Overview of F-logic from Database Transformation Perspective. Technical Note CSI-N9607, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, 1996.

[23] Gy. Kovács, Cs. Hajas, and I. Quilio. Representations and Query Languages of Nested Relations. In L. Varga, editor, *Proceedings of the 4th Symposium on Programming Languages and Software Tools*, pages 360–373, Visegrád, Hungary, June 1995.

[24] J. Melton. (ISO/ANSI Working Draft) Database Language SQL/Foundation (SQL3). ISO DBL MCI-004 and ANSI X3H2-96-059, March 1996.

[25] J. Nachouki, M.P. Chastang, and H. Briand. From Entity-Relationship Diagram to an Object-Oriented Database. In *Proceedings of the 11th International Conference on the Entity-Relationship Approach*, pages 459–481, 1992.

[26] G.M. Nijssen and T.A. Halpin. *Conceptual Schema and Relational Database Design: a fact oriented approach.* Prentice-Hall, Sydney, Australia, 1989.

[27] P. Pistor and F. Andersen. Designing a Generalized NF^2 Data Model with an SQL-type Language Interface. In *Proceedings of the 12th VLDB Conference*, pages 278–185, Kyoto, Japan, August 1986.

[28] M.A. Roth, H.F. Korth, and D.S. Batory. SQL/NF: a query language for ¬1NF relational databases. *Information Systems*, 12(1):99–114, 1987.

[29] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[30] H.J. Schek and M.H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[31] K.-D. Schewe and B. Thalheim. Fundamental Concepts of Object Oriented Databases. *Acta Cybernetica*, 11(1-2):49–83, 1993.

[32] G.M. Shaw and S.B. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proceedings of the 6th International Conference on Data Engineering*, pages 154–162, 1990.

[33] P. Shoval and S. Shiran. Entity-relationship and object-oriented data modeling - an experimental comparision of design quality. *Data & Knowledge Engineering*, 21(3):297–315, 1997.

[34] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, and D. Beech. Third-Generation Database System Manifesto. *SIGMOD Record*, 19(3):31–44, September 1990.

[35] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, 1986.

[36] J.D. Ullman. *Principles of Database and Knowledge-base Systems*, volume I. Computer Science Press, Rockville, Maryland, 1989.

Received, June 1997