# Subtyping and Polymorphism in Object-Role Modelling

T.A. Halpin and H.A. Proper[1]
Asymetrix Research Laboratory
Department of Computer Science
University of Queensland
Australia 4072
E.Proper@acm.org

Version of August 23, 1998 at 1:32

**Published as:** T.A. Halpin and H.A. Proper, *Subtyping and Polymorphism in Object-Role Modelling*, Data & Knowledge Engineering, 15(3), 251–281, 1995.

**Abstract**

Although Entity-Relationship (ER) modelling techniques are commonly used for information modelling, Object-Role Modelling (ORM) techniques are becoming increasingly popular, partly because they include detailed design procedures providing guidelines for the modeller. As with the ER approach, a number of different ORM techniques exist. In this paper, we propose an integration of two theoretically well founded ORM techniques: FORM and PSM. Our main focus is on a common terminological framework, and on the notion of subtyping. Subtyping has long been an important feature of semantic approaches to conceptual schema design. It is also the concept in which FORM and PSM differ the most in their formalization. The subtyping issue is discussed from three different viewpoints covering syntactical, identification, and population issues. Finally, a wider comparison of approaches to subtyping is made, which encompasses other ER-based and ORM-based information modelling techniques, and highlights how formal subtype definitions facilitate a comprehensive specification of subtype constraints.

KEYWORDS

Object-Role Modelling, Conceptual Modelling, Information Systems, Subtyping, Polymorphism

[1] Part of this work has been supported by an Australian Research Council grant, entitled: "An expert system for improving complex database design"

## 1 Introduction

Even though Entity-Relationship (ER) modelling techniques are commonly used for information modelling, the use of Object-Role Modelling (ORM) techniques such as NIAM [1], FORM [2], PM [3], PSM [4], BRM [5], NORM [6] and MOON [7], is becoming widespread. Unlike ER diagrams, ORM diagrams are linguistically based, they can be fully populated, and they capture more information (e.g. domains and complex constraints). Since ORM views the application world simply in terms of objects playing roles, it makes no initial use of the notion of attribute; hence the modeller is not required to agonize over whether some feature ought to be modelled in terms of attributes rather than entity types or relationship types.

Another reason for the increasing use of ORM is its inclusion of elaborated design procedures which guide the modeller in the modelling task. A weakness, as well as an advantage, of ER modelling techniques is their lack of details. Most ORM techniques allow a modeller to specify a universe of discourse to a high level of detail, often leading to very elaborate diagrams. While such detail is useful in formulating and transforming a model, for other purposes summary models which hide detail are also required. Recent studies have provided several abstraction mechanisms for ORM, including automatic derivation of ER views from ORM conceptual schemas ([8], [9]). Hence ORM now provides a powerful class of modelling techniques with an elaborated design procedure, as well as ways to hide undesired details from the resulting diagrams when required.

As with the ER approach, a number of different ORM techniques exist. One of the earliest versions of ORM can be found in [10]. The ORM version presented there is called NIAM, and is a variation of ORM which only allows for binary relationship types. A more liberalised version was published in [11] (and later published in English as [12]). The first detailed description of NIAM in English appeared as [1] (in the mean time the second edition of this book as appeared as [13]). Since then, a number of new ORM variations have appeared. Some examples of these variations are: NIAM [1], FORM [2], PM [3], PSM [4], BRM [5], and NORM [6]. PM is essentially a formalisation of NIAM allowing for $n$-ary relationship types. An alternative formalisation of $n$-ary NIAM is found in [14]. FORM and PSM are two formalised extensions of the original NIAM. They allow for more advanced modelling constructs; including sequences, sets, and a notion of polymorphism. BRM is a binary variation of ORM which allows only binary fact types; a formalisation of a binary NIAM called B-ORM can be found in [15]. Finally, NORM is an object-oriented variation of NIAM.

Although each of these ORM techniques has its own special merits, the need for some standardisation has become apparent. Two recent conferences on ORM ([16], [17]) are first attempts in this direction. A standardisation of ORM would certainly be welcomed by CASE-tool vendors, and could lead to a further spread of ORM usage. If a common base for ORM techniques is found, it would enable CASE-tool vendors to make their tools adaptable to different ORM versions by simply flipping an electronic switch corresponding to an appropriate axiom of the underlying formal theory. This has led to the idea of an ORM-Kernel as discussed in [18].

A further advantage of such standardisation is that research results based on a standardised version of ORM may be applied to other ORM versions. In the past some attempts to make research into ORM techniques more generally applicable have been made. For example, issues regarding internal representations of ORM models have been studied in [19], [20] and [21]. In [22] and [23] the conceptual query language LISA-D is presented, which allows for the formulation of queries in a semi-natural language format, closely following the naming conventions in the data model. In [24] and [25] schema evolution of ORM models is discussed.

As a first step in the standardisation of ORM, two popular versions of ORM are partially integrated in this paper. We have chosen FORM and PSM since:

1. FORM has both a theoretical background ([20], [2], [26], [27], [13]) and a practical background in the form of a prototype CASE-tool ([20], [28]), and even a commercially available CASE-tool (InfoModeler, [29]).

2. PSM has an even further elaborated theoretical base ([4], [30], [22], [23]), and to a lesser extent a practical background in the form of a prototype CASE-tool ([31]).

The integration of both ORM versions is part of ongoing research, which currently focusses on an integration of the *ways of modelling* of both techniques/methods. The *way of modelling* provides an abstract description of the underlying modelling concepts together with their inter-relationships and properties. It structures the models used in the information system development (i.e. it provides an abstract language in which to express the models).

In general, a method/technique can be disected in 6 aspects: a way of thinking, a way of working, a way of modelling, a way of controlling, a way of supporting, and a way of communicating. This disection was presented before in more detail in [32], [24] and [25].

Before continuing with a brief discussion of the contents of this article, it should be noted that in Object-Role Modelling the term object is used differently from the way it is used in Object-Oriented modelling techniques. In Object-Oriented Modelling techniques, objects are rather *dynamic* entities incorporating both behaviour and structure.

In this paper, the main focus is on the unification of the subtyping mechanisms of both ORM techniques. An important role in this discussion is played by the notion of identification of object type instances. Besides normal subtyping, PSM also has a mechanism for the introduction of polymorphic types which is also taken into consideration as it is highly intertwined with subtyping. Subtyping has long been an important feature of semantic approaches to conceptual schema design. It is also an important area in which ORM techniques differ, since it involves underlying syntactical, semantical and identification issues. Recent trends, in particular in the field of object oriented databases ([33], [34]) and distributed databases, have also highlighted the importance of subtyping.

Furthermore, some aspects of the ways of communicating of Object-Role models, in particular in relation to subtyping are discussed. PSM and FORM both have associated a graphical way of communicating, and a textual way of communicating: LISA-D ([22]) and FORML ([26]) respectively. In this paper we also discuss some aspects of the graphical representation and definition of subtypes. An elaborated paper on the relation between LISA-D and FORML is planned for the near future.

The structure of the paper is as follows. Section 2 provides a common terminological framework for ORM. In section 3, the FORM approach to subtyping is discussed, and a formalisation in the style of PM is provided. Section 4 outlines the PSM approach to subtyping, together with its formalisation. The integrated approach is discussed in section 5. A wider comparison of subtyping including approaches used in other modelling techniques is made in section 6. There we also discuss how formal subtype definitions facilitate a comprehensive treatment of subtype constraints.

## 2 Terminological Framework

When comparing modelling techniques, it is essential to have a common terminological framework. In this section we briefly introduce such a framework for FORM and PSM. This framework is used to present the approaches to subtyping of FORM and PSM, as well as the unified one. We limit the terminological framework to those aspects needed for the discussions in the remainder of this paper. A more elaborate terminological framework is part of ongoing research in a further integration between PSM and FORM. Relevant background for this unified framework includes a formalisation of PSM ([4], [30], [35]) and NIAM ([14]), descriptions of FORM ([2], [26], [14]), and a first attempt to find such a unified framework ([18]). The terminological framework as proposed in this section is the result of discussions among the authors of PSM and FORM, and is part of a joint research effort to combine PSM and FORM into a general ORM standard.

We assume the reader has a basic knowledge of the concepts underlying ORM or ER. A conceptual schema consists of a number of types ($\mathcal{TP}$). These types may be classified according to various criteria. In the framework, we use two classifications. The first one is based on the underlying structure of the type, and the second one on the way instances can be denoted. In the first classification, the following classes of types are distinguished in an ORM conceptual schema:

1. The set of *basic types* ($\mathcal{BS}$) comprises the object types which do not have a direct underlying structure, and are not the result of subtyping or polymorphy.

2. The set of *relationship types* ($\mathcal{RL}$). Relationship types are used to describe relationships between object types, and are usually provided in the form of a predicate.

3. Instances of some relationship types may be treated as objects that participate in another relationship. Relationship types that are considered to be object types are called *nested object types* or *objectified relationship types* ($\mathcal{NE} \subseteq \mathcal{RL}$).

4. The nested object types are the first class of object types with a direct underlying structure. Besides these, the following object types with an underlying structure exist: *set types* ($\mathcal{ST}$; also known as power types), *bag types* ($\mathcal{BG}$), *sequence types* ($\mathcal{SQ}$) and *schema types* ($\mathcal{SC}$); their instances are sets, bags, sequences or conceptual subschemas respectively.

5. Object types may be specialised into *subtypes* ($\mathcal{SB}$), by specifying a subtype defining rule (the rule which makes the instances special).

6. Existing object types with different underlying structures (called the specifiers) may be united into object types that we call *polymorphic types* ($\mathcal{PL}$). The term "polymorphic" is chosen since it means "many shapes".

The classes $\mathcal{RL}$, $\mathcal{ST}$, $\mathcal{BG}$, $\mathcal{SQ}$, $\mathcal{SC}$, $\mathcal{BS}$, $\mathcal{SB}$ and $\mathcal{PL}$ are mutually exclusive. Based on the above set of basic type classes, the following hierarchy of type classes may be defined:

$$\begin{array}{lcll}
\mathcal{CM} & \triangleq & \mathcal{NE} \cup \mathcal{ST} \cup \mathcal{BG} \cup \mathcal{SQ} \cup \mathcal{SC} & \text{compound types} \\
\mathcal{RT} & \triangleq & \mathcal{CM} \cup \mathcal{BS} & \text{root types} \\
\mathcal{IN} & \triangleq & \mathcal{SB} \cup \mathcal{PL} & \text{inheriting types} \\
\mathcal{OB} & \triangleq & \mathcal{RT} \cup \mathcal{IN} & \text{object types} \\
\mathcal{TP} & \triangleq & \mathcal{RL} \cup \mathcal{OB} & \text{types}
\end{array}$$

In figure 1, this hierarchy is depicted graphically as an ORM schema. The compound types comprise all types that have a direct underlying structure. The root types are the types that are the roots of (possibly empty) typing hierarchies built from subtypes and polymorphic types. The inheriting types are the subtypes or polymorphic types; they inherit the properties (including their identification) from their specifiers.

In the second classification of types, the set of all types $\mathcal{TP}$ is split into two distinct sets reflecting the separation between directly, and indirectly, denotable object types:

1. The *value types* ($\mathcal{VL} \subseteq \mathcal{TP}$) are those types whose instances are directly denotable on a communication medium. This effectively means that there exists some pre-defined denotation function for the instances of the value types (e.g. printing a number as a sequence of ASCII-characters or as a Kanji-character). Value types typically have a pre-defined set of basic operations ($+$, $-$, $<$, $>$, ...). The difference between entity types and value types is not always clear. Consider for instance the object type Date. It is unclear whether Date should be a non-value type or a value type. We simply presume that a concrete CASE-tool will supply a number of predefined value types as a kind of "standard library".

2. The set of *non-value types*: $\mathcal{TP} - \mathcal{VL}$.

The set of entity types (roughly corresponding to ER's notion of entity type) could now be defined as: $\mathcal{EN} \triangleq \mathcal{OB} - \mathcal{VL}$. An example ORM conceptual schema is found in figure 2. Entity types are depicted as named, solid ellipses. Value types are shown as named, broken ellipses. Predicates (relationship types) are shown as named sequences of role-boxes, with the predicate name in or beside the first role of the predicate. A nested object type is shown as a frame around a predicate (e.g. "Estimation"). Arrow-tipped bars over

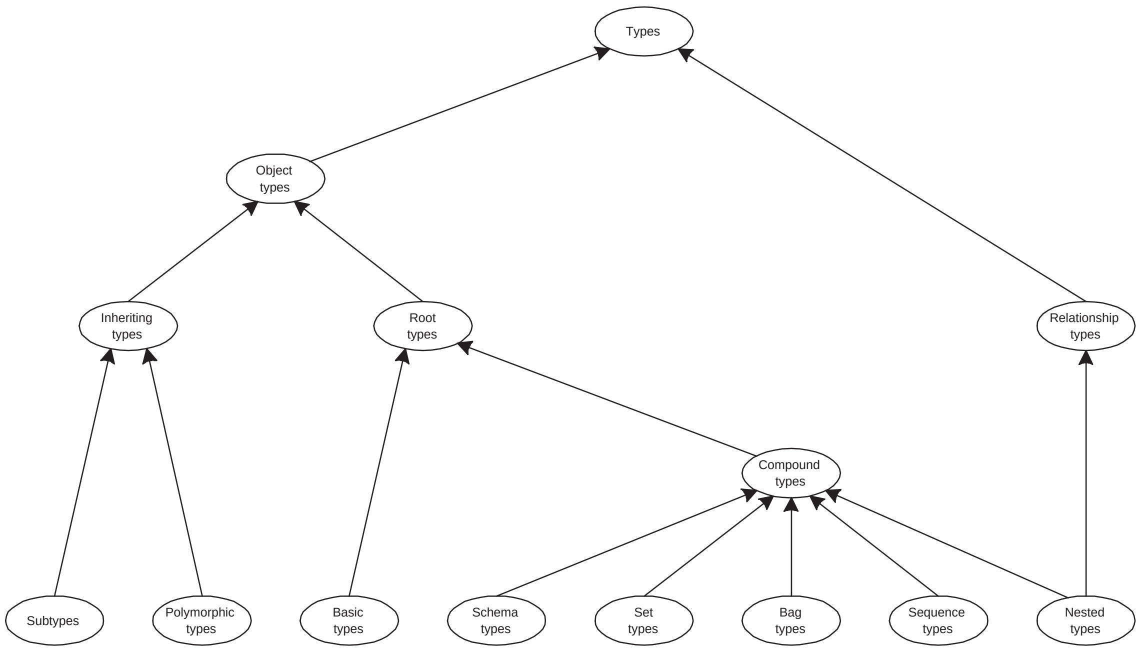


Figure 1: Type hierarchy

one or more role boxes indicate a uniqueness constraint over the role(s). For example, each House has at most one HouseNr. Black dots on the connection between an object type and role-box specify a mandatory role (or total role) constraint. This means the population recorded for the object type must play that role. For example, each house must have an estimated cost, but the date of an estimation need not be known.

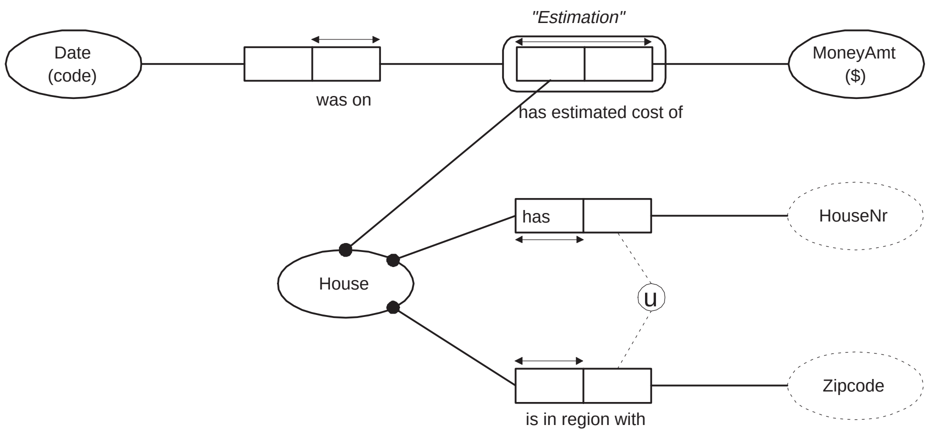


Figure 2: Example ORM schema

Simple identification schemes are shown as reference modes in parentheses (e.g. "(code)", "($)"). This is an abbrevation for reference types which inject the entity types into associated value types (DateCode, and $Value, respectively). The entity type House has a composite identification scheme. The circled "u" is an external uniqueness constraint indicating that each combination of HouseNr and Zipcode refers to at most one House (this is true in some countries, including The Netherlands). For this schema we have:

$$\begin{aligned}\mathcal{VL} &\triangleq \{\mathsf{DateCode}, \mathsf{\$Value}, \mathsf{HouseNr}, \mathsf{Zipcode}\} \\ \mathcal{EN} &\triangleq \{\mathsf{Date}, \mathsf{MoneyAmt}, \mathsf{House}, \mathsf{Estimation}\}\end{aligned}$$

$$\begin{aligned}\mathcal{NE} &\triangleq \{\mathsf{Estimation}\}\\ \mathcal{RL} &\triangleq \{\mathsf{was\ on}, \mathsf{has\ estimated\ cost\ of}, \mathsf{has}, \mathsf{is\ in\ region\ with}\}\end{aligned}$$

In this example, each relationship type has a distinct name. To allow some names (e.g. “has”) to be used with different predicates, a different identification scheme is actually used for predicates (e.g. internal predicate numbers).

In a conceptual schema, all types must be identifiable, meaning that all instances can be referred to by means of a single value or a combination thereof. The identification of a type is provided by means of a so-called reference scheme. Value types are self identifying since they all have an associated denotation function. Compound objects and relationships are usually identified in terms of their underlying structure. Polymorphic types inherit not only their structure from the comprising object types, but their identification as well.

## 3 Subtyping in FORM

A basic concept in data modelling is specialisation, also referred to as subtyping. Specialisation is a mechanism for representing one or more (possibly overlapping) subtypes of a common supertype. Intuitively, a specialisation relation between a subtype and a supertype implies that the instances of the subtype are also instances of the supertype. In this section, a discussion and formalisation of the aspects involved in subtyping as can be found in FORM are provided. Syntactical issues, identification, and the treatment of populations, are discussed consecutively. The formalisation is phrased in terms of the common terminological framework.

### 3.1 Syntax

As an example of a specialisation hierarchy, consider the schema depicted in figure 3. Subtype connections are graphically represented as arrows from the subtype to the supertype. In this example we use “carnivore” and “herbivore” in the sense of “carnivore only” and “herbivore only”; so an omnivore does not belong to these classes. The associated subtype defining rules are:

- **each** Flesh_eater **is an** Animal **that** is of AnimalType **in** $\{$‘carnivore’, omnivore’$\}$
- **each** Plant_eater **is an** Animal **that** is of Animal type **in** $\{$‘herbivore’, ‘omnivore’$\}$
- **each** Carnivore **is an** Animal **that** is of Animal_type ‘carnivore’
- **each** Herbivore **is an** Animal **that** is of Animal_type ‘herbivore’
- **each** Omnivore **is a** Flesh_eater **and also a** Plant_eater

In FORM and PSM, subtype defining rules are formulated in a conceptual (yet formalised) language such as FORML ([36], [27], [13]) or LISA-D ([22], [23]). The subtyping relations in a conceptual schema can be captured formally by the binary relation $\mathsf{SpecOf} \subseteq \mathcal{OB} \times \mathcal{OB}$ with the intuition:

if $x \,\mathsf{SpecOf}\, y$ then ‘$x$ is a specialisation of $y$’, or ‘$x$ is a subtype of $y$’.

The set of subtypes $\mathcal{SB}$ exactly correspond to the object types that receive a supertype via SpecOf, so we have: $\mathcal{SB} \triangleq \{x \mid \exists_y [x \,\mathsf{SpecOf}\, y]\}$. In the example of figure 3, the following hierarchy is depicted:

- $\mathsf{Obj}(\mathsf{Flesh\_eater}) \,\mathsf{SpecOf}\, \mathsf{Obj}(\mathsf{Animal})$
- $\mathsf{Obj}(\mathsf{Plant\_eater}) \,\mathsf{SpecOf}\, \mathsf{Obj}(\mathsf{Animal})$

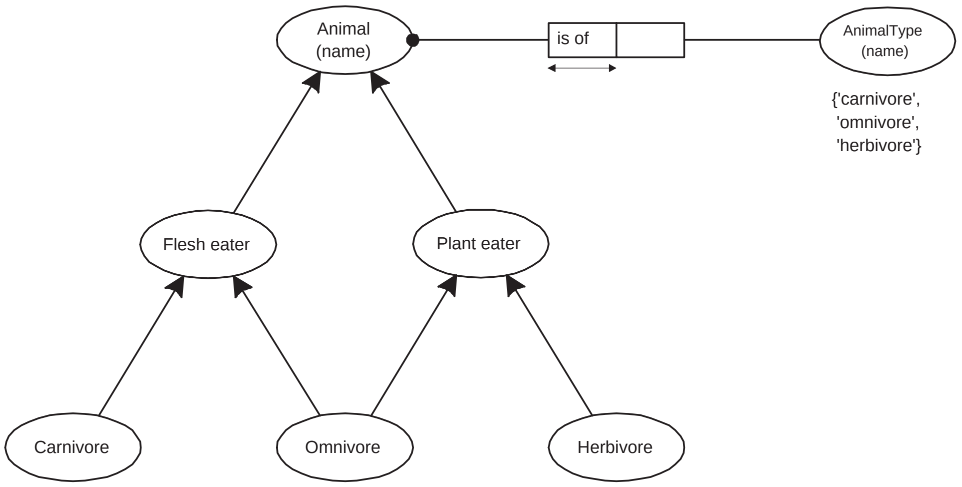


Figure 3: Example of a specialisation hierarchy

- $\mathsf{Obj}(\mathsf{Carnivore})\ \mathsf{SpecOf}\ \mathsf{Obj}(\mathsf{Flesh\_eater})$
- $\mathsf{Obj}(\mathsf{Omnivore})\ \mathsf{SpecOf}\ \mathsf{Obj}(\mathsf{Flesh\_eater})$
- $\mathsf{Obj}(\mathsf{Omnivore})\ \mathsf{SpecOf}\ \mathsf{Obj}(\mathsf{Plant\_eater})$
- $\mathsf{Obj}(\mathsf{Herbivore})\ \mathsf{SpecOf}\ \mathsf{Obj}(\mathsf{Plant\_eater})$

where $\mathsf{Obj}(x)$ refers to the object type with name $x$.

A subtyping hierarchy must adhere to certain rules. In FORM a strict separation between the value types and the non-value types is maintained in the specialisation hierarchy:

**[SF1]** (*separation*) If $x\ \mathsf{SpecOf}\ y$ then: $x \in \mathcal{VL} \iff y \in \mathcal{VL}$

Note: FORM does not support polymorphic types, so we have: $\mathcal{PL} = \varnothing$.

The $\mathsf{SpecOf}$ relation corresponds (semantically) to $\subset$ (i.e. is a proper subset of). Hence it should be transitive:

**[SF2]** (*transitive*) $x\ \mathsf{SpecOf}\ y\ \mathsf{SpecOf}\ z \Rightarrow x\ \mathsf{SpecOf}\ z$

Furthermore, the subtyping relation should not contain any cycles (transitive relations are acyclic if they are irreflexive):

**[SF3]** (*irreflexive*) $\neg x\ \mathsf{SpecOf}\ x$

A subtype hierarchy corresponds to a directed acyclic graph with a unique top. A specialisation hierarchy can thus be considered to be a semi-lattice, where for each pair of subtypes (in the same hierarchy), the least upper bound should exist. For an object type, the unique top nodes in the subtyping graph can be identified by:

$$\mathsf{Top}(x, y) \triangleq (x\ \mathsf{SpecOf}\ y \lor x = y) \land y \notin \mathcal{SB}$$

This subtype graph should, however, have exactly one top element:

**[SF4]** (*unique top element*) $\forall_{x \in \mathcal{TP}} [\mathsf{Top}(x, y_1) \land \mathsf{Top}(x, y_2) \Rightarrow y_1 = y_2]$

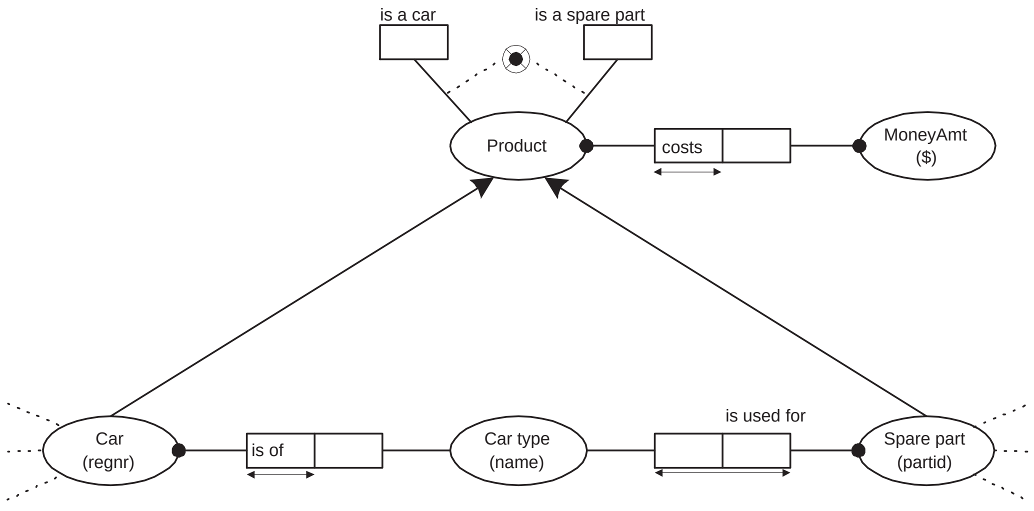


Figure 4: Price-list running example in FORM

This unique top is referred to as the *pater familias* of object type $x$, and is denoted as $\mathsf{Top}(x)$. In figure 3 the pater familias is Animal.

Finally, as a more advanced example of a subtyping in FORM consider figure 4. The depicted universe of discourse is concerned with a price list for individually priced Products in a garage. A Product is either a Car, or a Spare part. The Cars are identified by a registration number, while a Spare part is identified by an id number. Furthermore, every Car is of a type, and every Spare part is used in cars of a certain type. Finally, Products have a price associated to them. The corresponding conceptual schema in FORM is depicted in figure 4. For this schema we have the following subtype defining rules (formulated in FORL):

- **each** Car **is a** Product **that** is a car
- **each** Spare_part **is a** Product **that** is a spare part

Note that the schema depicts only a fragment of a larger universe of discourse, hence the unconnected outgoing arcs from the Car and Spare part object types.

As an illustration of the key differences between FORM and traditional NIAM with respect to subtyping, consider the NIAM version of this universe of discourse as shown in figure 5. It may be argued that the schema in figure 5 suffers from overspecification. A special value type Code needs to be introduced in order to identify Products. FORM allows this to be avoided by using a disjunctive reference schema. Products are always either a Car or a Spare part. Since Cars are identified through their registration nr, and Spare parts thtough their part id, all Products can indeed be identified.

The introduction of the artificial identification in the NIAM schema may be considered as a violation of the *Conceptualisation Principle* (see [37], [1]), since it was not present in the original universe of discourse.

## 3.2 Population

For each database state, a type is populated by a set of instances. The notion of population can be modelled formally by the total function: $\mathsf{Pop} : \mathcal{TP} \rightarrow \wp(\Omega)$. Here "$\Omega$" denotes the domain of values. The population of a subtype is completely determined by the populations of its supertypes and its subtype defining rule:

**[PF1]** If $x \in \mathcal{SB}$ then:

$$\mathsf{Pop}(x) = \bigcap_{y : x \,\mathsf{SpecOf}\, y} \mathsf{Pop}(y) \cap \mu[\![\mathsf{SubtRule}(x)]\!]\,(\mathsf{Pop})$$

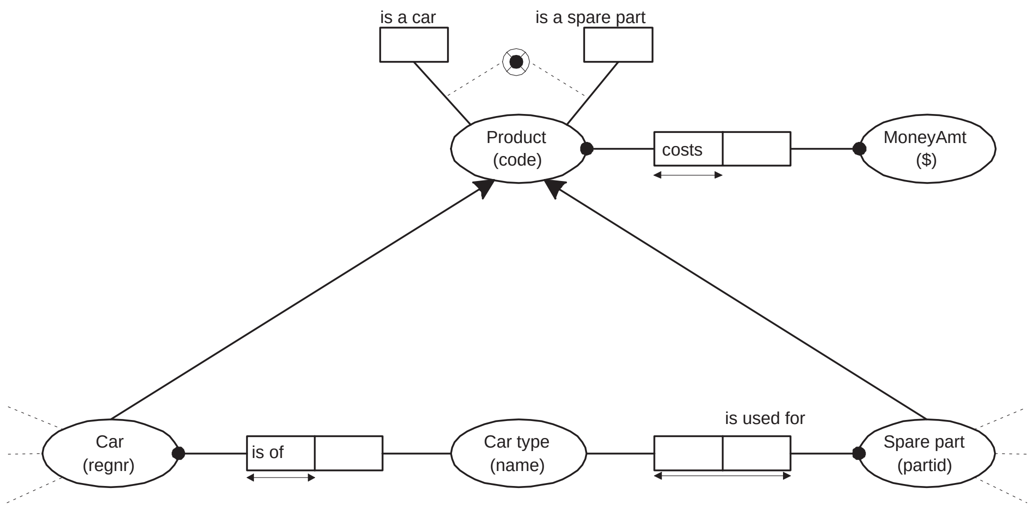


Figure 5: Price-lists in NIAM

SubtRule is a function providing the subtype defining rule for each subtype. Furthermore, $\mu[\![\mathsf{SubtRule}(x)]\!]\,(\mathsf{Pop})$ is presumed to evaluate the subtype defining rule $\mathsf{SubtRule}(x)$ of object type $x$ in population Pop. Basically, we have $\mu[\![\mathsf{SubtRule}]\!] : (\mathcal{TP} \to \wp(\Omega)) \to \wp(\Omega)$. Note that style of notation of $\mu[\![\mathsf{SubtRule}(x)]\!]\,(\mathsf{Pop})$ is the style of denotational semantics of programming languages ([38]). The exact definition of the $\mu$ function depends on the language chosen for the subtype defining rules. Two good candidates in the context of ORM are FORML and LISA-D.

Note that the $\bigcap_{y:x\,\mathsf{SpecOf}\,y} \mathsf{Pop}(y)$ part of the above axiom can actually be regarded as the minimal subtype defining rule since each instance of a subtype should at least be an instance of all its supertypes. So even if there would be no further limitation on the subtype, then this would still be a requirement for the subtype. When implementing a subtyping facility in a CASE-tool this can be presented to the user as a ‘read-only’ part of the subtype defining rule in the dialogue with the user when entering subtype defining rules.

## 3.3 Identification

An important issue in information modelling is identification. In a correct conceptual schema, all types must be identifiable ensuring that all instances can be denoted in terms of values. An object type is identifiable if it has a proper reference scheme. Further elaboration on what a proper reference scheme is can be found in [1] and [4]. Identification of types is required for both the insertion of instances and the representation of query results.

In figure 3, the Animal object type is identified by the implicit reference type Animal has AnimalName. The value type AnimalName corresponds to the names of the animals, which are self-identifying. The subtypes depicted all inherit their identification from the pater familias, i.e. Animal. In this article we formally consider only the rules governing the inheritance of identification in subtyping hierarchies.

Let $\mathsf{Identifiable}(x)$ denote the fact that object type $x$ is identifiable. In FORM, a subtype may inherit its identification from its supertypes (and eventually its pater familias). The inheritance principle is captured in the following derivation rule:

**[IF1]** (*identification inheritance*) $x\,\mathsf{SpecOf}\,y \wedge \mathsf{Identifiable}(y) \;\vdash\; \mathsf{Identifiable}(x)$

A subtype may, however, also have its own reference schema to identify its instances. If all subtypes of a total subtyping provide their own identification, then the supertype is also identifiable:

**[IF2]** (*total subtyping*) $\mathsf{Total}(X, y) \wedge \forall_{x \in X} \left[\mathsf{Identifiable}(x) \wedge x\, \mathsf{SpecOf}\, y\right] \;\vdash\; \mathsf{Identifiable}(y)$

where $\mathsf{Total}(X, y)$ means that the object types in $X$ form a total subtyping of $y$, i.e. for any population $\mathsf{Pop}$ we must have:

$$\mathsf{Pop}(y) = \bigcup_{x \in X} \mathsf{Pop}(x)$$

Note that we restrict ourselves to the identifiability rules concerned with subtyping only, a complete set of axioms for NIAM can be found in [3].

As an example, consider the schema fragment of a conceptual schema of an administration at a university concerned with persons, employees and students as defined in figure 6 (adapted from [26]). The dotted lines attached to the object types indicate that other roles are played but are not shown here. Each person is a student or an employee (possibly both). A student is normally identified by a student number. Employees on the other hand, are preferably identified by their employee number. Student employees are identified by their student numbers like all other students. The circled black dot is a "total union" constraint indicating that Person is the union of Student and Employee. This constraint is implied by the subtype definitions and other details omitted here. Since persons are always students or employees, all persons can be identified. The preferable identification of persons, however, is the employee number. This preference is indicated by the disjunctive expression "(Empnr|Studnr)". In a disjunctive reference scheme like this, if more than one reference mode applies to the same object instance, then out of these reference modes the one that appears earliest in the parenthesized list is preferred. For further discussion of disjunctive and context-dependent reference see [13].

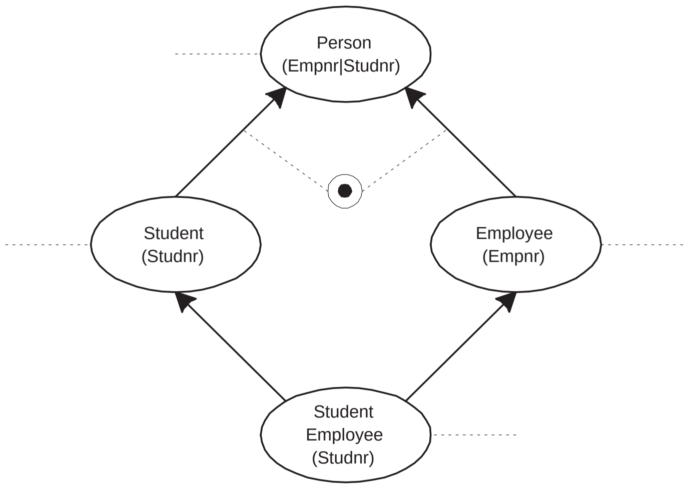


Figure 6: A subtype graph with differing reference schemes

Although natural, disjunctive reference schemes can be awkward to implement. For example, if the application requires common details about all the people to be listed together, a more efficient implementation is obtained if one is prepared to introduce a new, simple global identifier (e.g. Person#). Whether or not this is done, the information system is aware of which student is which employee, since the subtypes inherit all the roles played by their supertypes (including alternative identification roles).

When more than one reference scheme exists for an object type, a primary reference scheme must be chosen. In general, a sequence of reference schemes is associated with object types by the *partial* function: $\mathsf{Ident} : \mathcal{TP} \rightarrowtail \mathsf{RefSch}^{+}$, where RefSch is the domain for reference schemes (which will not be elaborated upon in this paper). A sequence (usually of length one) of reference schemes is associated, since one object type (e.g. Person) may have alternative reference schemas. The sequence then provides the preference of the alternatives. The inheritance rules on identification, and the default identification rules for compound object types and relationship types (identifying instances in terms of their components) can be used to derive a total identification function $\mathsf{Ident}' : \mathcal{TP} \rightarrow \mathsf{RefSch}^{+}$ from Ident.

# 4 Subtyping in PSM

PSM is an extended version of PM. The new modeling concepts introduced by PSM are power types, sequence types, schema types and generalisation. In this paper, only the new concept of generalisation is relevant as it is an extension to subtyping. The relation between the other extra concepts and FORM are discussed in ([39]).

## 4.1 Syntax

The formalisation of subtyping and generalisation presented in this section is based on the one provided in [35] and [4]. However, we have adapted the formalisation according to the unified terminological framework. The subtyping relation in PSM is also modelled as a binary relation $\mathsf{SpecOf} \subseteq \mathcal{OB} \times \mathcal{OB}$, where again $\mathcal{SB} \triangleq \{x \mid \exists_y [x \,\mathsf{SpecOf}\, y]\}$. Analogously to FORM, subtyping adheres to the following rules:

**[SP1]** (*separation*) If $x \,\mathsf{SpecOf}\, y$ then: $x \in \mathcal{VL} \iff y \in \mathcal{VL}$

**[SP2]** (*transitive*) $x \,\mathsf{SpecOf}\, y \,\mathsf{SpecOf}\, z \Rightarrow x \,\mathsf{SpecOf}\, z$

**[SP3]** (*irreflexive*) $\neg x \,\mathsf{SpecOf}\, x$

**[SP4]** (*unique top element*) $\mathsf{Top}(x, y_1) \wedge \mathsf{Top}(x, y_2) \Rightarrow y_1 = y_2$

Besides subtyping, PSM offers an explicit generalisation construct. Although generalisation is often considered to be simply the inverse of specialisation, in PSM this is *not* the case. The difference between subtyping and generalisation lies in the fact that in the case of generalisation, instances and their identifications are inherited upward, whereas in the case of specialisation they are inherited downwards. Generalisation and specialisation are therefore different in nature, and furthermore originate from different axioms in set theory ([4]).

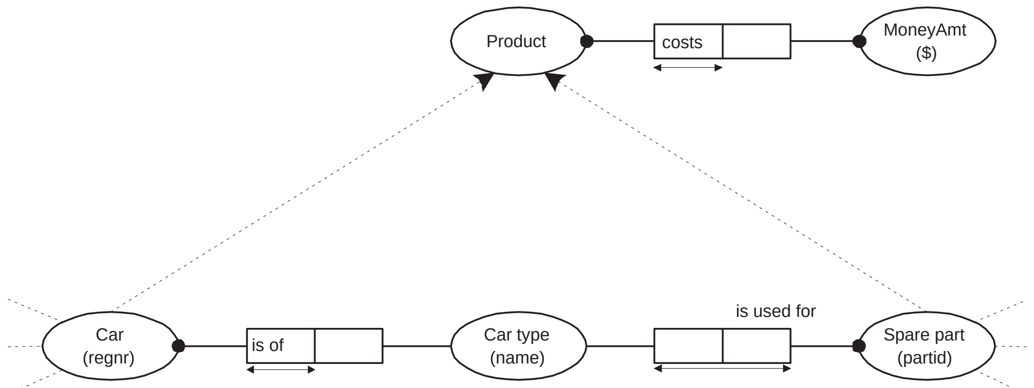


Figure 7: Example of Generalisation in PSM

As a first example for the motivation and use of generalisation, consider the price-list running example from the previous section depicted in figure 4. The corresponding conceptual schema in PSM depicted in figure 7. One might argue that the schema in figure 4 still suffers from overspecification, since two unary relationship types and two proper subtype defining rules are needed to distinguish between products which are a $\mathsf{Spare\ part}$ and those which are a $\mathsf{Car}$. We realise that this is not yet a convincing example of the use of generalisation. Below, we provide two examples that clearly do require the use of a generalisation.

Generalisation is modelled formally by the binary relation: $\mathsf{GenOf} \subseteq \mathcal{OB} \times \mathcal{OB}$, with the intuition:

$x \,\mathsf{GenOf}\, y$ means '$x$ is a generalisation of $y$' or '$y$ is a specifier of $x$'

Note that PSM uses the term "generalised object type" for types in $\mathcal{PL}$ rather than "polymorphic type". In the next section we revisit this issue, and we also put more restrictions on the use of polymorph types providing modellers with a clearer distinction on when to use polymorphism and when to use subtyping. The set of generalised object types is identified by: $\mathcal{PL} \triangleq \left\{x \;\middle|\; \exists_y \left[x \,\mathsf{GenOf}\, y\right]\right\}$. For the example depicted in figure 7 we have:

- $\mathsf{Obj}(\mathsf{Product}) \,\mathsf{GenOf}\, \mathsf{Obj}(\mathsf{Car})$
- $\mathsf{Obj}(\mathsf{Product}) \,\mathsf{GenOf}\, \mathsf{Obj}(\mathsf{Spare\ part})$

Note the reversed correspondence between the binary relation and the direction of the arrows in the diagrams, as opposed to subtyping.

Generalisation also maintains the distinction between the value and non-value types:

**[SP5]** (*separation*) If $x \,\mathsf{GenOf}\, y$ then: $x \in \mathcal{VL} \iff y \in \mathcal{VL}$

The following two familiar rules also hold for generalisation:

**[SP6]** (*transitive*) $x \,\mathsf{GenOf}\, y \,\mathsf{GenOf}\, z \Rightarrow x \,\mathsf{GenOf}\, z$

**[SP7]** (*irreflexive*) $\neg x \,\mathsf{GenOf}\, x$

A generalisation hierarchy does not always have a unique top element. As an illustration of such a situation, suppose that the garage of the discussed example has a fleet of delivery vans to deliver spare parts to smaller garages in the neighbourhood, and that it is desired to keep a record of both the kilometres driven by the cars sold and delivery vans used for the delivery of spare parts. The resulting (fragment of the) universe of discourse could be modelled by the conceptual schema depicted in figure 8. When modelling this universe of discourse in traditional NIAM, or FORM, a common supertype of Vehicle and Product needs to be introduced (e.g. to enable Vehicle and Product to overlap). Doing so is somewhat artificial, especially if no facts (other than required for subtype definition) have to be stored for this common supertype.

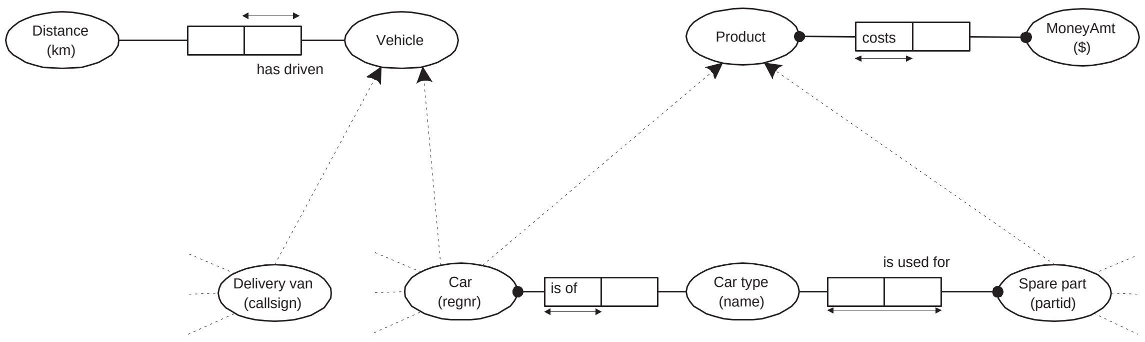


Figure 8: Complex generalisation hierarchy in PSM

As a more advanced (meta modelling) example of generalisation in PSM, consider figure 9. This figure depicts a typical recursive definition of sequences. A sequence either consists of one atom, or is an atom concatenated to an already existing sequence. Note that this example actually proves that sequence types (and so are set types, bag types and schema types) are not elementary concepts. In [39] this is discussed in more detail. However, as argued in [22] and [4], treating them as explicit concepts in the ORM theory allows for the specification of clearer and more elegant rules concerning these types.

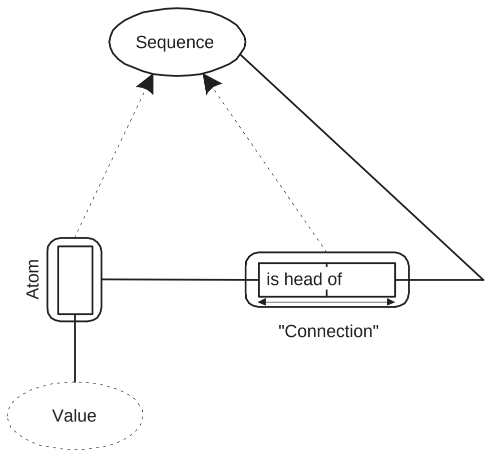


Figure 9: Recursive definition of sequences

## 4.2 Population

The population of a subtype is again completely determined by the populations of its supertypes and the subtype defining rule:

**[PP1]** If $x \in \mathcal{SB}$ then:

$$\mathsf{Pop}(x) = \bigcap_{y:x\,\mathsf{SpecOf}\,y} \mathsf{Pop}(y) \cap \mu[\![\mathsf{SubtRule}(x)]\!]\,(\mathsf{Pop})$$

In the case of generalisation, however, the population is the union of the populations of the specifiers:

**[PP2]** If $x \in \mathcal{PL}$ then:

$$\mathsf{Pop}(x) = \bigcup_{y:x\,\mathsf{GenOf}\,y} \mathsf{Pop}(y)$$

## 4.3 Identification

In PSM, a subtype inherits its identification from its supertype (and eventually its pater familias):

**[IP1]** (*subtype identification inheritance*) $x\,\mathsf{SpecOf}\,y \wedge \mathsf{Identifiable}(y) \;\vdash\; \mathsf{Identifiable}(x)$

PSM does not allow for alternative reference schemes. Therefore upward inheritance in the case of a total subtyping is not present in PSM.

In the case of generalisation, the inheritance is reversed (arrow-wise). A generalised object type inherits the identifications of its specifiers. If one specifier is identifiable, then that is considered enough proof of the identifiability of the generalised object type. The reason why this is the case is that generalisation allows us to construct recursive types. If all specifiers where required to be identifiable, then one would not be able to create recursive types. The rule is now given by:

**[IP2]** (*generalisation identification inheritance*) $x\,\mathsf{GenOf}\,y \wedge \mathsf{Identifiable}(y) \;\vdash\; \mathsf{Identifiable}(x)$

In the example of figure 7, a Car is identified by Reg nr, and a Spare part is identified by a Part id. By applying the above rule it follows that Product is identifiable. For figure 9 things are a bit more complicated. An Atom is identified by its Value. By means of axiom IP2 we can then conclude that Sequence is identifiable. A fact type is identifiable if all object types playing a role in the fact type are identifiable. Therefore, the identifiability of Connection follows from the identifiability of Sequence and Atom. This recursion in the determination of the identifiability is caused by the recursion in the definition of the sequence. As stated before, the possible presence of such recursions is also the reason why axiom IP2 does not require all specifiers to be identifiable.

In PSM not all object types are allowed to provide their own identification. Subtypes and generalised object types must use their inherited identification. So we have for the Ident function:

$$\mathsf{Ident}{\downarrow}x \Rightarrow x \notin \mathcal{IN}$$

where $\mathsf{Ident}{\downarrow}x$ means that function Ident is defined for $x$.

## 4.4 Comparison

The core differences (except for terminology) between PSM and FORM with respect to subtyping are:

1. the presence of polymorphic types (“generalisation”) in PSM.
2. overriding of inherited reference schemes in subtype hierarchies is allowed in FORM.

In this section, we have provided strong arguments in favour of PSM’s notion of generalisation. This notion of generalisation will typically be used when common information needs to be stored for object types with differing (polymorphic) underlying structures. The definition of object types with a recursive underlying structure (see figure 9) is a good example of the expressive power of the generalisation (polymorphism) concept.

A problem with generalised object types is, however, dealing with their disjunctive reference schemes. For instance for LIST Product, the query result will be a mixture of registration numbers for the cars, and part ids, whose values may be drawn from incompatible data types. From a pragmatic point of view such polymorphic query results may be undesirable, since they may complicate queries and updates, make the process of designing screen and form layouts even harder, and implementation can be awkward unless surrogates are used. Therefore, one may argue that in some cases the introduction of an artificial common identification (a Product nr) is appropriate. See section 6.6 of [13] for further discussion of this point.

On the other hand, doing so puts an extra burden on the users who now have to specify a product number in addition to the traditional identifications (although this could be automated to some extent). Furthermore, in some domains such differing identifications (denotations) of instances are very natural. As an example consider figure 10. The depicted schema is concerned with the storage of presentations. A presentation is given on a date by a presenter, and consists of a slide show. One slide show is a sequence of slides, where each slide may be identified by its name in combination with the sequence of fragments that it contains. A fragment on a slide is either a graphical item or a piece of text, being a sequence of sentences. It should be clear that it is hardly possible to model this domain conceptually (and elegantly) by means of ordinary subtyping. Note that the object types Slide show, Fragments and Paragraph correspond to sequence types of the enclosed object types. When LISTing a slide show, one expects that for each slide the graphic fragments are presented as the graphic itself, and the paragraphs as the text contained in the paragraph, rather than as a list of artificial fragment identifiers.

In the unified approach, we incorporate the notion of generalisation as used in PSM (although we rename it to polymorphy). The examples in this section have shown this concept is indeed required. However, care needs to be taken to ensure a clear distinction between polymorphy and subtyping, in particular when to apply either of them.

In the current PSM setup, overriding of inherited reference schemes is not allowed. As a result, the schema depicted in figure 6 is not a valid PSM schema. In the unified approach we adopt FORM’s more liberal view with respect to the inheritance of reference schemes. Furthermore, we will allow specialisation hierarchies to contain multiple roots, allowing us to model situations as depicted in figure 8.

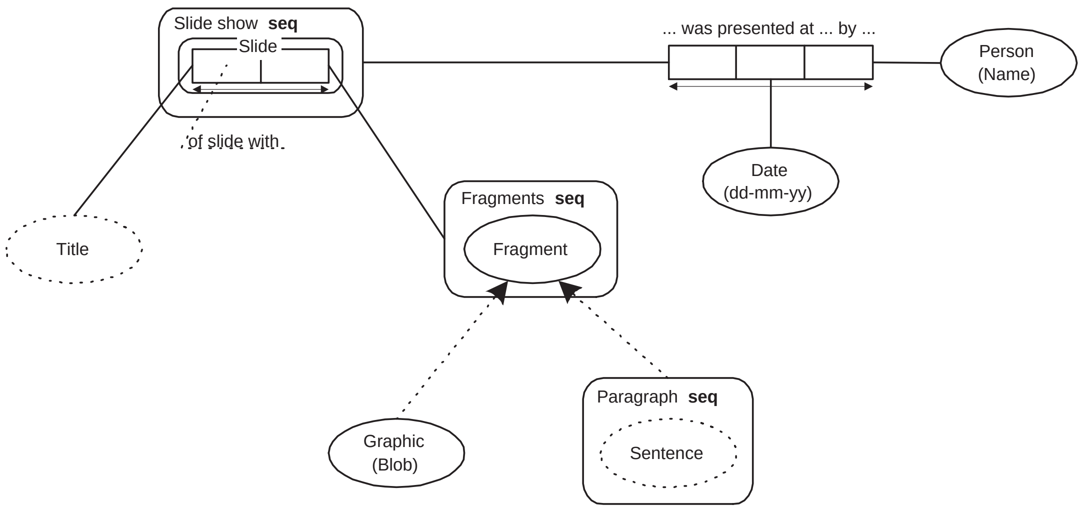


Figure 10: Presentations

## 5 A Unified View on Subtyping

In this section the approaches for subtyping of FORM and PSM are integrated into the unified approach, simply called ORM. In the discussion of the unified approach, we present some axioms which can be used as refinements. The axioms in this class (SW) are referred to as 'electronic SWitches'.

### 5.1 Syntax

In ORM, both specialisation and polymorphy are present. We use the term polymorphy for PSM's generalisation mechanism. Note, however, that we will put limits on the use of this mechanism in order to have a clearer distinction of when to use polymorphy or when to use subtyping. In contrast to PSM, we use the term "generalisation" as the inverse of specialisation (e.g. as in [13]).

Rather than introducing the specialisation and polymorphy relations separately, initially a general inheritance relation $\mathsf{IdfBy} \subseteq \mathcal{OB} \times \mathcal{OB}$ is introduced; with the intuition:

if $x \mathbin{\mathsf{IdfBy}} y$ then '$x$ inherits its identification from $y$'

This relation adheres to the following rules:

**[SO1]** (*transitive*) $x \mathbin{\mathsf{IdfBy}} y \mathbin{\mathsf{IdfBy}} z \Rightarrow x \mathbin{\mathsf{IdfBy}} z$

**[SO2]** (*irreflexive*) $\neg x \mathbin{\mathsf{IdfBy}} x$

The separation between value types and non value types applies for the entire inheritance hierarchy:

**[SO3]** (*separation*) If $x \mathbin{\mathsf{IdfBy}} y$ then: $x \in \mathcal{VL} \iff y \in \mathcal{VL}$

From the transitive $\mathsf{IdfBy}$ relation we can derive the intransitive one ($\mathsf{IdfBy}_1$) as follows:

$$x \mathbin{\mathsf{IdfBy}_1} y \iff x \mathbin{\mathsf{IdfBy}} y \wedge \neg\exists_z \left[x \mathbin{\mathsf{IdfBy}} z \mathbin{\mathsf{IdfBy}} y\right]$$

The finite depth of the identification hierarchy in ORM is expressed by the following schema of induction:

**[SO4]** (*identification induction*) If $F$ is a property for object types, such that:

$$\text{for any } y\text{, we have: } \forall_{x:y \,\mathsf{IdfBy}_1\, x} \left[F(x)\right] \Rightarrow F(y)$$

then $\forall_{x \in \mathcal{OB}} \left[F(x)\right]$

The latter axiom was not explicitly present in the previous discussions, but was always presumed to be implicitly present. In this paper, it is only stated for reasons of completeness. Note that the identification induction schema can be proven from the properties of IdfBy if the axomatic setup were extended with the natural numbers and their axioms (in particular natural number induction). In our formalisation, however, we do not presume the presence of the natural numbers.

In ORM, the identification hierarchy contains two 'flavours', the specialisation flavour and the polymorphy flavour:

1. $\mathsf{SpecOf} \subseteq \mathsf{IdfBy}$ the part of the identification hierarchy concerned with specialisation.
2. $\mathsf{HasMorph} \subseteq \mathsf{IdfBy}$ the part of the identification hierarchy concerned with polymorphism.

Again we have: $\mathcal{SB} \triangleq \left\{x \;\middle|\; \exists_y \left[x \,\mathsf{SpecOf}\, y\right]\right\}$ and $\mathcal{PL} \triangleq \left\{x \;\middle|\; \exists_y \left[x \,\mathsf{HasMorph}\, y\right]\right\}$. The intuition of $x \,\mathsf{HasMorph}\, y$ is that $x$ has as a morph $y$. The identification hierarchy is a direct result of both flavours:

**[SO5]** (*complete span*)
$x \,\mathsf{IdfBy}_1\, y \Rightarrow x \,\mathsf{HasMorph}\, y \vee x \,\mathsf{SpecOf}\, y$ and $x \,\mathsf{HasMorph}\, y \vee x \,\mathsf{SpecOf}\, y \Rightarrow x \,\mathsf{IdfBy}\, y$

Furthermore, the specialisation and polymorphy sub-hierarchies of IdfBy must be complete with respect to transitivity:

**[SO6]** (*transitive complete specialisation*) If $x \,\mathsf{IdfBy}\, y \,\mathsf{IdfBy}\, z$ then:

$$x \,\mathsf{SpecOf}\, y \,\mathsf{SpecOf}\, z \iff x \,\mathsf{SpecOf}\, z$$

**[SO7]** (*transitive complete polymorphy*) If $x \,\mathsf{IdfBy}\, y \,\mathsf{IdfBy}\, z$ then:

$$x \,\mathsf{HasMorph}\, y \,\mathsf{HasMorph}\, z \iff x \,\mathsf{HasMorph}\, z$$

Using the specialisation relation we can also define the set of atomic types $\mathcal{AT}$. They are the types which do not have a direct, or inherited, underlying structure. For $\mathcal{AT}$ we have the following derivation rules:

**[A1]** $x \in \mathcal{BS} \vdash x \in \mathcal{AT}$

**[A2]** $x \,\mathsf{SpecOf}\, y \wedge y \in \mathcal{AT} \vdash x \in \mathcal{AT}$

These two rules are all the rules to derive which elements are in $\mathcal{AT}$.

We allow for specialisation hierarchies to have multiple roots. Therefore no special axiom is needed requiring the presence of a unique root for the specialisation hierarchy. In general, the populations of basic types will be exclusive. However, to be able to model universes of discourse as depicted in figure 8 using the liberalised subtyping notion, we need an explicit mechanism to allow for basic types to share instances. In the example, it should clearly be the case that Product and Vehicle may share instances as Car is a subtype of both. For this purpose we introduce the predicate $\mathsf{Overlap} \subseteq \wp(\mathcal{BS})$. If $\mathsf{Overlap}(X)$, then the population of all basic types in $X$ may overlap. At the end of this section we show how figure 8 may now indeed be modelled using specialisation.

We are now in a position to define the concept of type-related types. Two types are considered to be type-related (or overlapping) if their populations may share instances. The notion of type-relatedness is introduced by a set of derivation rules. For more details on these rules and their importance (e.g. for query optimisation), refer to [22]. First the notion of *structurally-relatedness* is introduced. Structurally-relatedness, denoted by $x \sim_S y$ only takes properties from the information structure into account; so exclusion constraints are not taken into account yet.

Firstly, all types are structurally-related to themselves:

**[TS1]** $x \in \mathcal{TP} \vdash x \sim_S x$

Basic types which are explicitly stated to overlap are structurally-related:

**[TS2]** $\mathsf{Overlap}(X) \land x, y \in X \vdash x \sim_S y$

Bag types, set types, or sequence type, ranging over structurally-related element types ($\mathsf{Elt}(x)$ and $\mathsf{Elt}(y)$) lead to structurally-related types:

**[TS3]** $X \in \{\mathcal{BG}, \mathcal{SQ}, \mathcal{ST}\} \land x, y \in X \land \mathsf{Elt}(x) \sim_S \mathsf{Elt}(y) \vdash x \sim_S y$

Schema types were defined as types with an underlying schema. So every schema type has associated a small conceptual schema, and the *one* instances of a schema types must be a valid population of *that* underlying conceptual schema. As such, schema types allow for hierarchical decomposition of a conceptual schema. The object types contained in the conceptual schema associated to a schema type $x$ is denoted as $\mathcal{TP}_x$. For a more elaborate discussion of the concept of schema type, please refer to [4]. Schema types are structurally-related if the set of object types contained in the schema type are the same:

**[TS4]** $\mathcal{TP}_x = \mathcal{TP}_y \vdash x \sim_S y$

Finally, if an object type is structurally-related to an ancestor in the identification hierarchy, it is also structurally-related to object types lower in the hierarchy:

**[TS5]** $x \,\mathsf{IdfBy}\, y \land y \sim_S z \vdash x \sim_S z$

The above four rules are all the rules to determine structurally-relatedness of object types.

The next step is to take exclusion constraints into consideration, while structurally-relatedness is used as a base. This will limit the number of type-related object types. For this purpose we introduce extra derivation rules using a negative format (i.e. defining when object types are type-unrelated).

Firstly, object types which are not structurally-related are obviously type-unrelated (i.e. disjoint):

**[TR1]** $x \not\sim_S y \vdash x \not\sim y$

Exclusion constraints lead to type-unrelated object types:

**[TR2]** $\mathsf{Exclusion}(X) \land x, y \in X \vdash x \not\sim y$

Note: $\mathsf{Exclusion}(X)$ corresponds to an exclusion constraint on the types in $X$, its semantics are:

$$x, y \in X \Rightarrow \mathsf{Pop}(x) \cap \mathsf{Pop}(y) = \varnothing$$

Type-unrelatedness (incompatibility of types) is also inherited when constructing new types:

**[TR3]** $\mathsf{Elt}(x) \not\sim \mathsf{Elt}(y) \vdash x \not\sim y$

**[TR4]** $x \,\mathsf{IdfBy}\, y \wedge y \not\sim z \vdash x \not\sim y$

These three rules are all the rules to determine type-unrelatedness. The notion of type-relatedness is now defined as:

$$x \sim y \iff x, y \in \mathcal{TP} \wedge \neg(x \not\sim y)$$

Note that it is easy to see that $x \sim y \Rightarrow x \sim_S y$.

For exclusion constraints the following well-formedness axiom may be asserted:

**[SO8]** (*well-formed exclusion constraints*) $\mathsf{Exclusion}(X) \wedge x, y \in X \Rightarrow x \sim_S y$

This axiom basically states that if two object types are explicitly stated to be exclusive, this should not already have been derivable.

The notion of type-relatedness can be used to state the following wellformedness axiom for (multi rooted) specialisation hierarchies. If a specialisation hierarchy has multiple roots, then the roots must be type-related since instances of common subtypes must be instances of all roots:

**[SO9]** (*multi rooted subtypes*) $\mathsf{Top}(z, x) \wedge \mathsf{Top}(z, y) \wedge x \neq y \Rightarrow x \sim y$

Furthermore, the specifiers of a polymorphic type may not all be atomic (i.e. some object types with an underlying structure must be involved).

**[SO10]** (*polymorphic specifiers*) $x \in \mathcal{PL} \Rightarrow \exists_{u \in \mathcal{OB} - \mathcal{AT}} [\mathsf{Bottom}(x, u)]$

Similar to Top for specialisation, Bottom provides the specifiers for polymorphic types:

$$\mathsf{Bottom}(x, y) \iff x \,\mathsf{IdfBy}\, y \wedge y \notin \mathcal{PL}$$

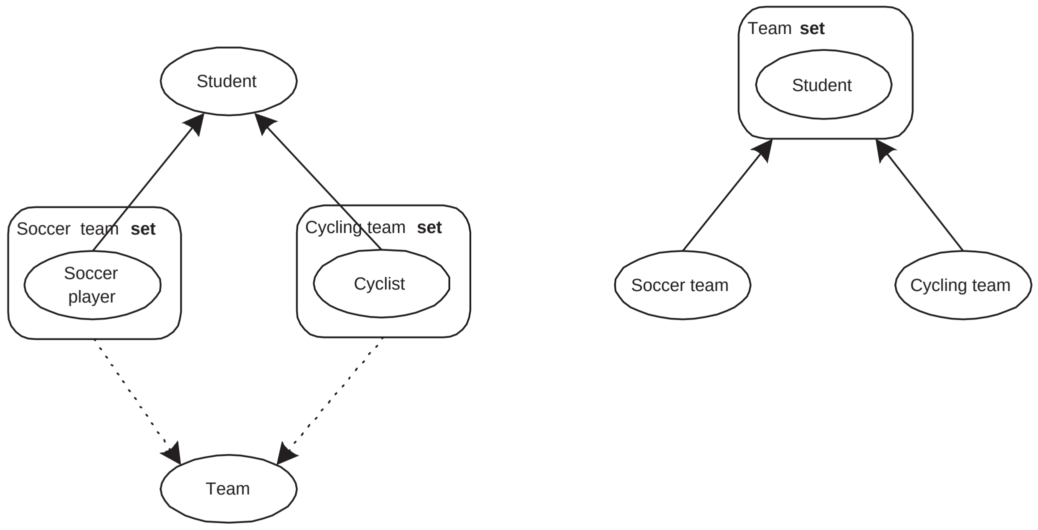


Figure 11: Specialisation and set types

The above rule allows us to make a clear difference between situations were polymorphism should be used and when a subtyping should be used. With this rule, the schema shown in figure 8 becomes illegal. As interesting examples of correct schemas, consider figure 11. The two depicted schemas are equivalent fragments of the same universe of discourse. A decision on the preferred one depends on the way the subtype defining rules are formulated, and whether information needs to be stored that is exclusive to Soccer players or Cyclists. If it is more sensible to define subtype defining rules for Soccer players and Cyclists than for Cycling teams and Soccer teams, or if information specific to individual Soccer players or Cyclists needs to be stored, one should opt for the leftmost fragment. Otherwise the rightmost fragment is the preferred one.

Finally, some "switches" may be introduced that can be used to adapt ORM to one's own requirements. Some people may argue that subtyping is not allowed for value types. This can be enforced by the switch:

**[SW1]** (*no value subtypes*) $x \,\mathsf{SpecOf}\, y \Rightarrow x \notin \mathcal{VL}$

The introduction of polymorphic types can simply be forbidden by the following switch:

**[SW2]** (*no polymorphic types*) $\mathcal{PL} = \varnothing$

For subtyping, the Top operator can be defined in the usual way. Furthermore, subtype hierarchies still have unique tops:

**[SW3]** (*unique top element*) $\mathsf{Top}(x, y_1) \wedge \mathsf{Top}(x, y_2) \Rightarrow y_1 = y_2$

For more possible switches, refer to [18].

## 5.2 Population

The type-relatedness concept allows for the following strong typing rule:

**[PO1]** $x, y \in \mathcal{TP} \wedge x \not\sim y \Rightarrow \mathsf{Pop}(x) \cap \mathsf{Pop}(y) = \varnothing$

Note that this rule implicitly defines the semantics of exclusiveness constraints on object types!

The population of a subtype is again completely determined by the populations of its supertypes, and the subtype defining rule:

**[PO2]** If $x \in \mathcal{SB}$ then:

$$\mathsf{Pop}(x) = \bigcap_{y:x \,\mathsf{SpecOf}\, y} \mathsf{Pop}(y) \cap \mu[\![\mathsf{SubtRule}(x)]\!]\,(\mathsf{Pop})$$

In the case of polymorphism, however, the population is the union of the populations of the specifiers:

**[PO3]** If $x \in \mathcal{PL}$ then:

$$\mathsf{Pop}(x) = \bigcup_{y:x \,\mathsf{HasMorph}\, y} \mathsf{Pop}(y)$$

## 5.3 Identification

In the unified approach, object types are allowed to have alternative reference schemes as in FORM. Therefore, a sequence of reference schemes is associated with object types by the *partial* function: $\mathsf{Ident} : \mathcal{TP} \rightarrowtail \mathsf{RefSch}^+$, where RefSch is the domain for reference schemes. The inheritance rules on identification, and the default identification rules for compound object types and relationship types can be used to derive a total identification function $\mathsf{Ident}' : \mathcal{TP} \to \mathsf{RefSch}^+$ from Ident.

We now have the following derivation rules for the identifiability of object types:

**[IO1]** (*subtyping inheritance*) $\forall_{y:x \,\mathsf{SpecOf}\, y} [\mathsf{Identifiable}(y)] \;\vdash\; \mathsf{Identifiable}(x)$

**[IO2]** (*total subtyping*) $\mathsf{Total}(X, y) \wedge \forall_{x \in X} [\mathsf{Identifiable}(x) \wedge x \,\mathsf{SpecOf}\, y] \;\vdash\; \mathsf{Identifiable}(y)$

**[IO3]** (*polymorphic inheritance*) $x \,\mathsf{HasMorph}\, y \wedge \mathsf{Identifiable}(y) \;\vdash\; \mathsf{Identifiable}(x)$

## 5.4 When to subtype and when to morph

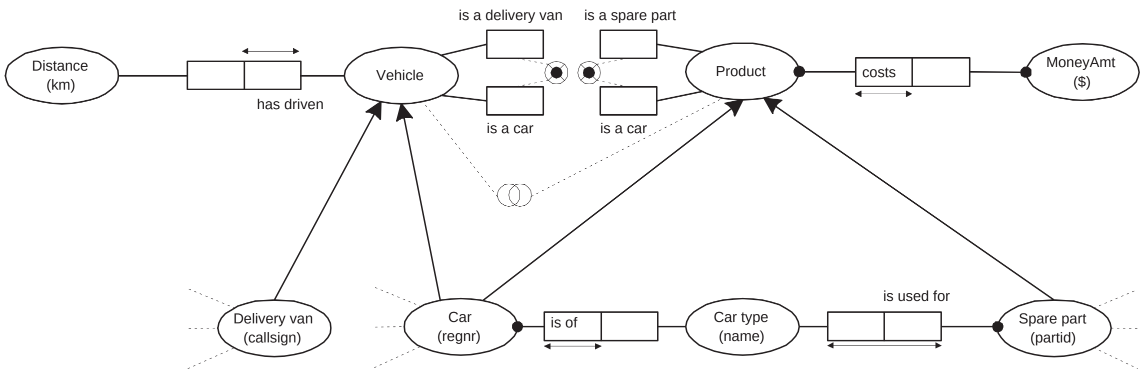
So far, in this article our main attention is on the way of modelling of ORM. However, one aspect of a unified ORM way of working does need some discussion. In the previous section we have argued and demonstrated that both polymorphism and subtyping are needed as modelling concepts. The downside of having two concepts which are so similar to each other is that when modelling a universe of discourse, it is sometimes hard to decide between the two different concepts. Nevertheless, two clear rules can be formulated to decide on the issue:

1. If common facts need to be recorded for types with a (differing) direct underlying structure, a polymorphic type *must* be used to unite the involved types.

   Figures 9 and 10 are examples of such situations. When trying to model such domains with specialisations rather than polymorphic types, one typically introduces artificial object types and identifiers.

2. If common facts need to be recorded for types without a direct underlying structure, a subtyping should be used (see e.g. figures 3 and 6). Even when a hierarchy with multiple tops is required, a subtyping is preferred in such situation!

In some situations, however, more contextual information is required. For instance, for figure 11, the decision is based on the availability of sensible subtype defining rules and the need to store explicit information about the subtypes.



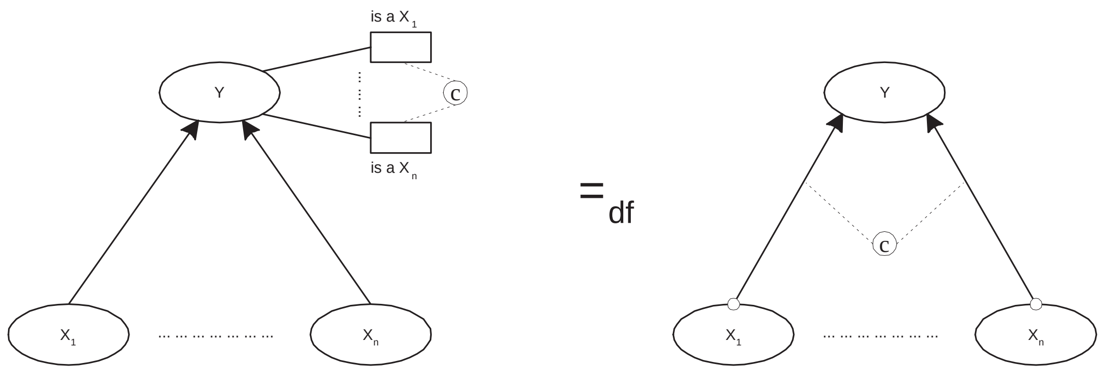
**each** Car **is a** Vehicle **that** is a car **or is a** Product **that** is a car
**each** Delivery_van **is a** Vehicle **that** is a delivery van
**each** Spare_part **is a** Product **that** is a spare part

Figure 12: Multi rooted specialisation

As promised, figure 12 now depicts the same universe of discourse as modelled in figure 8. However, this time we use the liberal notion of subtyping allowing for multiple roots. The fact that the population of object types Vehicle and Product may overlap ($\mathsf{Overlap}(\{\mathsf{Obj}(\mathsf{Vehicle}), \mathsf{Obj}(\mathsf{Product})\})$) is depicted by the ⚭ symbol. Note that one could argue that there is always implicitly a supertype (with appropriate subtype defining rules) present when an ⚭ symbol is used, in this case it would be Vehicle or Product.

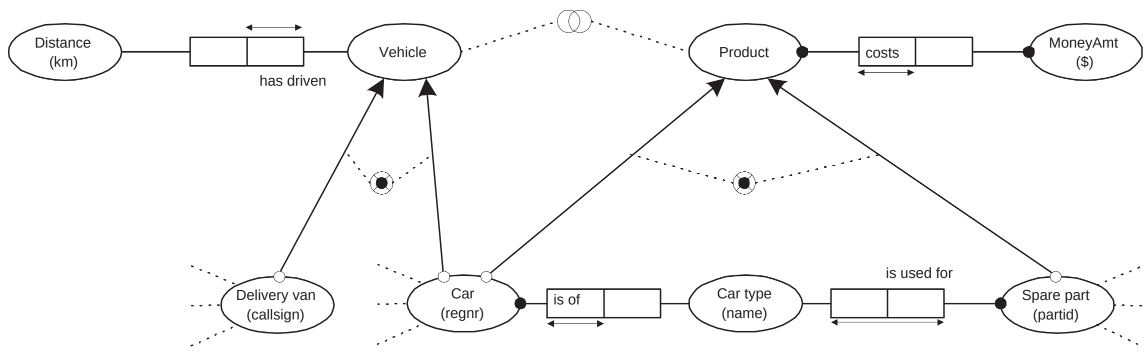
To allow for simpler diagrams if only trivial subtype defining rules are present, the graphical abbreviation as depicted in figure 13 can be used. In this figure, the encircled 'c' stands for any constraint that can be put on subtypes (exhaustion and exclusion). Using this abbreviation, the domain of figure 8 can be modelled as shown in figure 14.

From certain points of view, one might argue that it is possible to completely unite the concepts of polymorphism and specialisation. In the research leading up to this article, we have tried to do so. There are, however, a few issues that have stoped us from making such a step.



**each** $X_1$ **is a** Y **that** is a $X_1$
...
**each** $X_n$ **is a** Y **that** is a $X_n$

Figure 13: Graphical abbreviation for subtypes



**each** Car **is a** Vehicle **that** is a car **or is a** Product **that** is a car
**each** Delivery_van **is a** Vehicle **that** is a delivery van
**each** Spare_part **is a** Product **that** is a spare part

Figure 14: Multi rooted specialisation with abbreviations

1. in specialisation, properties (e.g. constraints and identification) are inherited downward with respect to the arrows,

2. in the case of polymorphism the properties are inherited in the direction of the arrows

Clearly, when uniting specialisation and polymorphism into one concept, this leads to problems. Even when these problems can be solved on the formal side, which is far from a trivial problem, the questions remains how intuitive the resulting diagrams will be. When having two distinct concepts with two distinct graphical representations, users of the technique will be in a much better position to distinguish between the differing semantics of inheritance.

Furthermore, at the moment specialisation and polymorphism correspond to differing constructs from set theory ([4]). Specialisation corresponds directly to the notion of set comprehension. For example, when writing $X = \{x \in Y \mid P(x)\}$, $Y$ corresponds to the supertype, and $X$ to the subtype, while $P$ is the subtype defining rule. Polymorphism, on the other hand, corresponds to set union. If $X$ and $Y$ are two morphs, then $X \cup Y$ corresponds to the polymorphic type uniting $X$ and $Y$.

At the moment interesting research is underway where the semantics of ORM techniques is studied from a category theoretic ([40]) point of view ([41], [42]). In this approach many of the differences between polymorphism and specialisation seem to disapear as category allows for a very general description of properties of data modelling techniques. At the moment, however, it is far from clear whether the distinction from the *modeller's point of view* (and thus the technique) should ever disapear.

## 6 Subtyping and its Context

In this section we discuss the context of subtyping and polymorphy in ORM, and relate them to other approaches such as (E)ER.

### 6.1 Alternative Approaches to Subtyping

The approaches taken to subtyping by the original NIAM ([10], [12]) and the BRM ([5]) variation of NIAM, agree with the subtyping concept used in PSM. The identification in the case of relationships (traditional NIAM does not have compound objects other than relationships) or specialisations is restricted to the inherited identifications.

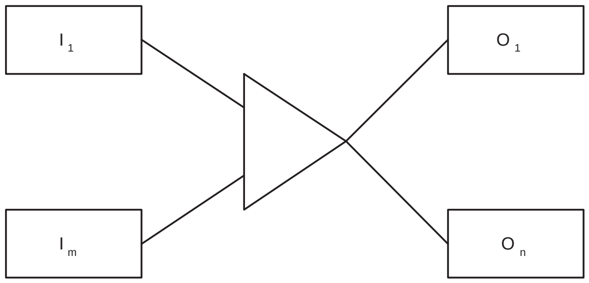


Figure 15: Type construction in EER

IFO ([43]) is an information modeling technique which does not have a very elaborate practical background, but it does have a profound mathematical base. It features a notion of generalisation and specialisation corresponding to PSM's approach. IFO's notion of generalisation corresponds to polymorphy. The specialisation and generalisation relations in IFO are denoted using two different arrows, with differing semantics. However, cyclic definitions (like the sequence type example) are simply not allowed.

In [44] the Entity Category Relationship (ECR) model is introduced. It uses the concept of sub-categories to define specialised types, and categories to build polymorphic types. They also use different notations for building categories and sub-categories, like PSM and IFO. The EER variant as presented in ([45], [46], [47]) uses the notion of type construction. A set of object types (could be one) is generalised into a union, and then specialised into a set of (possibly one) other object types. This is essentially a specialisation and polymorphism of PSM/IFO/ECR combined into one construction. In figure 15 an illustration of such a type construction is provided. For the population we have:

$$\bigcup_{1 \leq i \leq n} O_i \subseteq \bigcup_{1 \leq i \leq m} I_i$$

where the subtypes ($O_i$) could also have associated a subtype defining rule. In [18] the relation between specialisation, polymorphism and EER's type construction is discussed in more detail. It is not so much a semantic difference, but rather a graphical abbreviation.

Another recent EER variant is presented in ([48]). In this EER version, the relationship between supertypes and their subtypes is referred to as the superclass/subclass relationship. This specialisation mechanism corresponds to NIAM's standard subtyping. The only real difference is that subtypes not always have to be defined by a subtype defining rule in which case it is a user-defined subclass. This situation, however, corresponds to the graphical abreviation as introduced in figure 13.

In ([48]) the authors also identify the problem that in some subclass hierarchies more than one common superclass (pater familias in PM terminology) is required. For these situations they introduce the notion of category (which is now slightly different from the category notion used in [44]). Categorisation corresponds to the notion of union in set theory, and indeed corresponds to polymorphy when object types with polymorphic underlying structures are involved. Otherwise, our new notion of specialisation with multiple roots will suffice for such cases. In categorisation hierarchies, overriding of inherited reference schemes is also allowed.

Object oriented information modeling techniques inherently have a subtyping (subclass) mechanism ([33], [49]). Some object oriented information modeling techniques also feature a notion of polymorphic types. The notion of polymorphic type is also known from object oriented programming languages ([50]).

## 6.2 Graphical representation

In practice, several different notations are used to graphically display subtypes or polymorphic types. For simple subtype/polymorphic graphs, Euler diagrams may be used (e.g. as in Oracle's CASE Designer). However this method is unwieldy for more complex cases (e.g. overlapping subtypes). The simple method using solid arrows for specialisation and dotted arrows for polymorphy used in the ORM diagrams is recommended since it captures specialisation/polymorphy graphs of any complexity while allowing plenty of space around subtype nodes to attach roles: as the complexity of the graph grows it is even more important that the exclusion and exhaustion constraints be left implied (from the definitions).

## 6.3 Constraints and Subtyping

Since subtypes are defined (in Object-Role Modelling techniques) in terms of their supertypes using a subtype defining rule, constraints enforced on the supertype may now become implied constraints on the subtype. A more elaborated discussion of this topic can be found in [26]. Consider the ORM conceptual schema depicted in figure 16. In this universe of discourse academics working for a department have exactly one rank (lecturer, senior lecturer or professor) and have been awarded one or more degrees. Each professor holds a unique chair (e.g. information systems). Only senior lecturers may act as counsellors for students. Each student has exactly one counsellor.

The subtype links indicate that each Senior Lecturer and Professor is an Academic. This information is also implied by the subtype definitions. With this approach, subtypes must be well-defined (i.e. each must be formally defined in terms of roles attached to one or more of its supertypes). This has three main advantages.

Firstly, except for rare cases any exclusion and exhaustion constraints on subtypes are now implied by the subtype definitions and the constraints on the defining predicates. For example, an exclusion constraint between $\mathsf{Senior\ Lecturer}$ and $\mathsf{Professor}$ is implied (each $\mathsf{Academic}$ has only one $\mathsf{Rank}$) and hence need not be added to the diagram. This simplifies the diagram, since explicit exclusion and exhaustion constraints are often messy to depict. Most systems that support subtype exclusion and exhaustion (or total union) constraints require these to be explicitly added by the designer, since in these systems subtype definitions are not formally supported. A directly resulting problem of this is that certain inconsistent constraint patterns may be permitted. For instance, suppose the designer adds an exhaustion constraint between $\mathsf{Senior\ Lecturer}$ and $\mathsf{Professor}$ in figure 16. This effectively corresponds to:

$$\mathsf{Pop}(\mathsf{Obj}(\mathsf{Senior\ Lecturer})) \cup \mathsf{Pop}(\mathsf{Obj}(\mathsf{Professor})) = \mathsf{Pop}(\mathsf{Obj}(\mathsf{Academic}))$$

This should be rejected since it is inconsistent with the value constraint, since lecturers (code $'\mathsf{L}'$) may also exist.

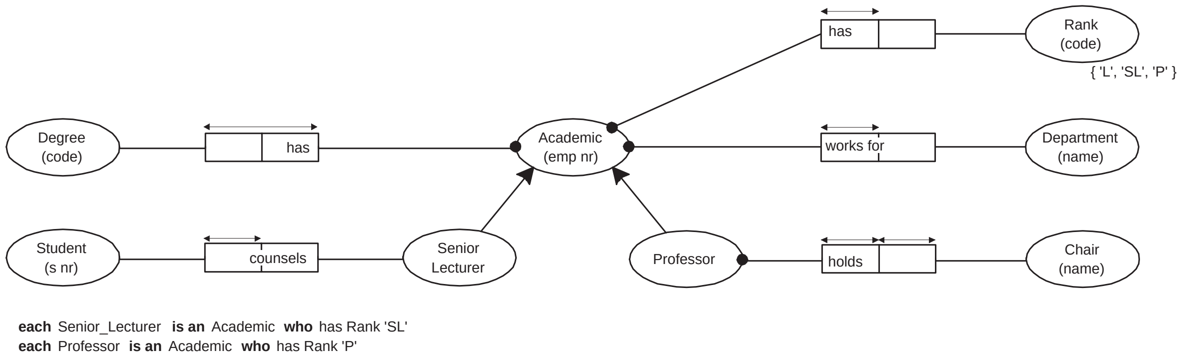


Figure 16: Subtypes are active and well defined

Secondly, an object type depicted as a subtype of another type is always a proper subtype of the former. Hence it is illegal for a subtype to have different, exclusive supertypes. For example, suppose we try to add a further subtype which is a subtype of both Senior Lecturer and Professor. This is rejected at definition time because the definition of the subtype is population-inconsistent (for more on this notion of inconsistency, see [36]). In most systems this violation is not detected unless an exclusion constraint is explicitly asserted between the supertypes, as for instance in RIDL* ([51]).

Thirdly, formal subtype definitions allow specification of constraints that are stronger than totality or exclusion. For example, even if an exclusion constraint is declared between $\mathsf{Senior\ Lecturer}$ and $\mathsf{Professor}$ this still allows us to populate the counselling role with the professors and the chair-holding role with the senior lecturers! Such errors can only be avoided by enforcing the subtype definitions as constraints.

In some situations one wants to have a mechanism to override constraints that are inherited by a subtype from a supertype (or inherited by a polymorphic type from a morph). The population of a subtype is a subset from its supertypes of the specialisation hierarchy. Similary, the population of a polymorphic types a subset of the (united) populations of its morphs. Therefore, in ORM, constraints can only be strengthened in the direction of the inheritance. Formally, we can model this requirement as follows. Let $x$ and $y$ be types such that $x\,\mathsf{SpecOf}\,y$ or $x\,\mathsf{HasMorph}\,y$. If $d_1$ is a constraint associated to type $x$ and this constraint is overrided for type $y$ as constraint $d_2$, then we should have:

$$d_1 \Vdash d_2$$

where $d_1 \Vdash d_2$ is defined as:

$$d_1 \Vdash d_2 \triangleq \forall_p\,[p \text{ is a population of the information stcurture} \land \mathsf{HoldsIn}(d_1, p) \Rightarrow \mathsf{HoldsIn}(d_2, p)]$$

and $\mathsf{HoldsIn}(d, p)$ denotes the fact that constraint $d_1$ holds in population $p$. The intuitive meaning of $d_1 \Vdash d_2$ is: $d_1$ is at least as restrictive as $d_2$ (see also [35]). The idea of overriding constraints in the context of ORM has been discussed before in [6] and [52].

## 7 Conclusion

In this article we have proposed an integrated view on subtyping and polymorphic types in Object-Role Modelling techniques, which is compatible with other information modeling techniques, including ER based techniques. Our integrated view features the definition of subtyping (specialisation), polymorphic types, and overriding of inherited reference schemes in such hierarchies. Furthermore, we have proposed the idea of 'electronic switches', to adapt the formal theory for Object-Role Modeling to one's own wishes and way of thinking with regards to information modelling.

As a next step, a more complete integration of FORM and PSM will be made, and the associated conceptual query and update languages FORML and LISA-D will also be integrated. Furthermore, parts of the results of this research may be incorporated in future versions of an existing commercial CASE-Tool ([29]).

## Acknowledgments

We would like to thank A.H.M. ter Hofstede for his comments and discussions on previous versions of this paper. We would also like to thank the anonymous referees for their comments and suggestions, which have lead to improvements of the original article.

## References

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

[2] T.A. Halpin and M.E. Orlowska. Fact-oriented modelling for data analysis. *Journal of Information Systems*, 2(2):97–119, April 1992.

[3] P. van Bommel, A.H.M. ter Hofstede, and Th.P. van der Weide. Semantics and verification of object-role models. *Information Systems*, 16(5):471–495, October 1991.

[4] 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.

[5] P. Shoval and S. Zohn. Binary-Relationship integration methodology. *Data & Knowledge Engineering*, 6(3):225–250, 1991.

[6] O.M.F. De Troyer. The OO-Binary Relationship Model: A Truly Object Oriented Conceptual Model. In R. Andersen, J.A. Bubenko, and A. Sølvberg, editors, *Proceedings of the Third International Conference CAiSE'91 on Advanced Information Systems Engineering*, volume 498 of *Lecture Notes in Computer Science*, pages 561–578, Trondheim, Norway, May 1991. Springer-Verlag.

[7] H. Habrias. Normalized Object Oriented Method. In *Encyclopedia of Microcomputers*, volume 12, pages 271–285. Marcel Dekker, New York, New York, 1993.

[8] L.J. Campbell and T.A. Halpin. Automated Support for Conceptual to External Mapping. In S. Brinkkemper and F. Harmsen, editors, *Proceedings of the Fourth Workshop on the Next Generation of CASE Tools*, pages 35–51, Paris, France, June 1993.

[9] L.J. Campbell and T.A. Halpin. Abstraction Techniques for Conceptual Schemas. In R. Sacks-Davis, editor, *Proceedings of the 5th Australasian Database Conference*, volume 16, pages 374–388, Christchurch, New Zealand, January 1994. Global Publications Services.

[10] G.M.A. Verheijen and J. van Bekkum. NIAM: an Information Analysis Method. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: A Comparative Review*, pages 537–590. North-Holland/IFIP, Amsterdam, The Netherlands, 1982.

[11] J.J.V.R. Wintraecken. *Informatie-analyse volgens NIAM*. Academic Service, 1985. (In Dutch).

[12] J.J.V.R. Wintraecken. *The NIAM Information Analysis Method: Theory and Practice*. Kluwer, Deventer, The Netherlands, 1990.

[13] T.A. Halpin. *Conceptual Schema and Relational Database Design*. Prentice-Hall, Sydney, Australia, 2nd edition, 1995.

[14] T.A. Halpin. *A logical analysis of information systems: static aspects of the data-oriented perspective*. PhD thesis, University of Queensland, Brisbane, Australia, 1989.

[15] O.M.F. De Troyer. A Logical Formalization of the Binary Object-Role Model. In T.A. Halpin and R. Meersman, editors, *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, pages 28–44, Magnetic Island, Australia, July 1994.

[16] G.M. Nijssen, editor. *Proceedings of the NIAM-ISDM Conference*. NIAM-GUIDE, September 1993.

[17] T.A. Halpin and R. Meersman, editors. *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*. Key Centre for Software Technology, University of Queensland, Brisbane, Australia, Magnetic Island, Australia, July 1994.

[18] S.J. Brouwer, C.L.J. Martens, G.H.W.M. Bronts, and H.A. Proper. Towards a Unifying Object Role Modelling Approach. In T.A. Halpin and R. Meersman, editors, *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, pages 259–273, Magnetic Island, Australia, July 1994.

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

[20] T.A. Halpin. A Fact-Oriented Approach to Schema Transformation. In B. Thalheim, J. Demetrovics, and H.-D. Gerhardt, editors, *MFDBS 91*, volume 495 of *Lecture Notes in Computer Science*, pages 342–356, Rostock, Germany, 1991. Springer-Verlag.

[21] T.A. Halpin. Fact-oriented schema optimization. In A.K. Majumdar and N. Prakash, editors, *Proceedings of the International Conference on Information Systems and Management of Data (CISMOD 92)*, pages 288–302, Bangalore, India, July 1992.

[22] 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.

[23] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. A Conceptual Language for the Description and Manipulation of Complex Information Models. In G. Gupta, editor, *Seventeenth Annual Computer Science Conference*, volume 16 of *Australian Computer Science Communications*, pages 157–167, Christchurch, New Zealand, January 1994. University of Canterbury.

[24] H.A. Proper and Th.P. van der Weide. EVORM: A Conceptual Modelling Technique for Evolving Application Domains. *Data & Knowledge Engineering*, 12:313–359, 1994.

[25] H.A. Proper and Th.P. van der Weide. Information Disclosure in Evolving Information Systems: Taking a shot at a moving target. *Data & Knowledge Engineering*, 15:135–168, 1995.

[26] T.A. Halpin, J. Harding, and C-H. Oh. Automated Support for Subtyping. In B. Theodoulidis and A. Sutcliffe, editors, *Proceedings of the Third Workshop on the Next Generation of CASE Tools*, pages 99–113, Manchester, United Kingdom, May 1992.

[27] T.A. Halpin and J. Harding. Automated Support for Verbalization of Conceptual Schemas. In S. Brinkkemper and F. Harmsen, editors, *Proceedings of the Fourth Workshop on the Next Generation of CASE Tools*, pages 151–161, Paris, France, June 1993.

[28] T.A. Halpin. WISE: a Workbench for Information System Engineering. In V.-P. Tahvanainen and K. Lyytinen, editors, *Next Generation CASE Tools*, volume 3 of *Studies in Computer and Communication Systems*, pages 38–49. IOS Press, 1992.

[29] Asymetrix. *InfoModeler User Manual*. Asymetrix Corporation, 110-110th Avenue NE, Suite 700, Bellevue, WA 98004, Washington, 1994.

[30] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Data Modelling in Complex Application Domains. In P. Loucopoulos, editor, *Proceedings of the Fourth International Conference CAiSE'92 on Advanced Information Systems Engineering*, volume 593 of *Lecture Notes in Computer Science*, pages 364–377, Manchester, United Kingdom, May 1992. Springer-Verlag.

[31] J.W.G.M. Hubbers. Automated Support for Verification & Validation of Graphical Constraints in PSM. Technical Report 93/01, Software Engineering Research Centre (SERC), Utrecht, The Netherlands, 1993.

[32] G.M. Wijers and H. Heijes. Automated Support of the Modelling Process: A view based on experiments with expert information engineers. In B. Steinholz, A. Sølvberg, and L. Bergman, editors, *Proceedings of the Second Nordic Conference CAiSE'90 on Advanced Information Systems Engineering*, volume 436 of *Lecture Notes in Computer Science*, pages 88–108, Stockholm, Sweden, 1990. Springer-Verlag.

[33] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[34] T. Korson and J. McGregor. Understanding Object Oriented: A Unifying Paradigm. *Communications of the ACM*, 33(9):40–60, September 1990.

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

[36] T.A. Halpin and J.I. McCormack. Automated Validation of Conceptual Schema Constraints. In P. Loucopoulos, editor, *Proceedings of the Fourth International Conference CAiSE'92 on Advanced Information Systems Engineering*, volume 593 of *Lecture Notes in Computer Science*, pages 364–377, Manchester, United Kingdom, May 1992. Springer-Verlag.

[37] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base*. Publ. nr. ISO/TC97/SC5/WG3-N695, ANSI, 11 West 42nd Street, New York, NY 10036, 1982.

[38] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, Massachusetts, 1977.

[39] A.H.M. ter Hofstede and Th.P. van der Weide. Fact Orientation in Complex Object Role Modelling Techniques. In T.A. Halpin and R. Meersman, editors, *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, pages 45–59, Townsville, Australia, July 1994.

[40] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[41] E. Lippe and A.H.M. ter Hofstede. A Category Theory Approach to Conceptual Data Modeling. *RAIRO Theoretical Informatics and Applications*, 30(1):31–79, 1996.

[42] A.H.M. ter Hofstede, E. Lippe, and P.J.M. Frederiks. Conceptual Data Modeling from a Categorical Perspective. *The Computer Journal*, 39(3):215–231, August 1996.

[43] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[44] R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept: An extension to the entity-relationship model. *Data & Knowledge Engineering*, 1:75–116, 1985.

[45] U. Hohenstein and G. Engels. Formal Semantics of an Entity-Relationship-Based Query Language. In *Proceedings of the Ninth International Conference on the Entity-Relationship Approach*, Lausanne, Switzerland, October 1990.

[46] 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.

[47] U. Hohenstein and G. Engels. SQL/EER-syntax and semantics of an entity-relationship-based query Language. *Information Systems*, 17(3):209–242, 1992.

[48] R. Elmasri and S.B. Navathe. Advanced data models and emerging trends. In *Fundamentals of Database Systems*, chapter 21. Benjamin Cummings, Redwood City, California, 1994. Second Edition.

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

[50] B. Meyer. *Object-oriented software construction*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[51] O.M.F. De Troyer. RIDL*: A Tool for the Computer-Assisted Engineering of Large Databases in the Presence of Integrity Constraints. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 418–429, Portland, Oregon, 1989.

[52] H.A. Proper. *A Theory for Conceptual Modelling of Evolving Application Domains*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1994.