ELSEVIER

Data & Knowledge Engineering 12 (1994) 313–359

DATA & KNOWLEDGE ENGINEERING

# EVORM: A conceptual modelling technique for evolving application domains

H.A. Proper*, Th.P. van der Weide

*Department of Information Systems, University of Nijmegen, Toernooiveld, NL-6525 ED Nijmegen, The Netherlands*

Received 29 June 1993; revised 22 December 1993; accepted 20 January 1994

**Abstract**

In this paper we present EVORM, a data modelling technique for evolving application domains. EVORM is the result of applying a general theory for the evolution of application domains to the object role modelling technique PSM, a generalisation of ER, EER, FORM and NIAM.

First the general theory is presented. This theory describes a general approach to the evolution of application domains, abstracting from details of specific modelling techniques. This theory makes a distinction between the underlying information structure and its evolution on the one hand, and the description and semantics of operations on the information structure and its population on the other hand. Main issues within this theory are object typing, type relatedness and identification of objects.

After a (short) introduction to PSM, this general theory is applied, resulting in EVORM. Besides having a right of its own, the usefulness of the general theory is demonstrated by interpreting its abstract results, resulting in more intuitive rules for EVORM.

*Key words:* Schema evolution; Conceptual modelling; Evolving information systems; Temporal information systems; Data modelling; Predicator set model

## 1. Introduction

As has been argued in [43] and [17], there is a growing demand for information systems, not only allowing for changes of their information base, but also for modifications in their underlying structure (conceptual schema and specification of dynamic aspects). In case of snapshot databases, structure modifications will lead to costly data conversions and re-programming. The intention of an evolving information system ([16]) is to be able to handle updates of all components of the so-called *application model*, containing the information

* *Corresponding author.* Email: erikp@cs.kun.nl


*SSDI:* 0169-023X(94)0002-V

structure, the constraints on this structure, the population conforming to this structure and the possible operations.

In [49] a classification for incorporating time in information systems (databases) is presented. This classification makes a distinction between rollback, historical and temporal information systems (databases). However, all these classes do not yet take schema evolution into account. For this reason, we propose a new class: evolving information systems.

We mention some examples of research regarding these first three classes. In the TEMPORA project [51, 33], the ER model is enhanced with the notion of time, resulting in the ERT model. In TODM [3] and ERAE [15, 14], similar strategies are followed, extending the relational model with the notion of time. This makes it possible to handle historical data, over a (nonvarying) underlying information structure. In [32, 46 and 45] the focus is on the monitoring of dynamic constraints, i.e. constraints over such historical data. Dynamic constraints restrict temporal evolutions, i.e. state sequences of databases. Historical data, however, are considered in their approach only as a means for implementing a monitor. Only the object domains may vary in the course of time.

Within the class of evolving information systems, extensions of object oriented modelling techniques with a time dimension (both on instance and type level) can be seen as a first subclass. In [48] a taxonomy for type evolution in object oriented databases is provided. The ORION project [4, 30] offers a more detailed taxonomy, together with a (semi formal) semantics of schema updates restricted to object oriented databases. The ORION system, together with the GemStone system [38, 7], are among the first object oriented database systems to support schema/type evolution. In [52] and [53] an approach to the evolution of schemas in object oriented databases is followed in which schema objects (e.g. object types) are considered to be objects like others (from the application). We will do a similar thing, and consider objects of both levels as objects describing an evolution in the course of time.

The second subclass of evolving information systems can be found in the field of version modelling, which can be seen as a restricted form of evolving information systems [29, 35, 28]. An important requirement for evolving information systems, not covered by version modelling systems, is that changes to the structure can be made on-line. In version modelling, a structural change requires the replacement of the old system by a new system, and a costly conversion of the old population into a new population conforming to the new schema.

A third subclass of research regarding evolving information systems extends a manipulation language for relation models with historical operations, both on population and schema level. An example of this approach can be found in [34], in which an algebra is presented allowing relational tables to evolve by changing their arity. This direction is similar to the ORION project [4, 30], in that a manipulation language is extended with operations supporting schema evolution.

In Fig. 1 we see a framework, based on [55] (see also [47]), presenting a structured view on modelling methods. It makes a distinction between a way of thinking, a way of controlling, a way of modelling, a way of working, a way of communicating and a way of supporting. The *way of thinking* is concerned with the philosophy behind the method and contains basic assumptions and viewpoints of this method. The *way of controlling* deals with managerial aspects of system development, providing a mechanism to control the way of working. The *way of working* describes the process of system development, the (sub)tasks to be performed,

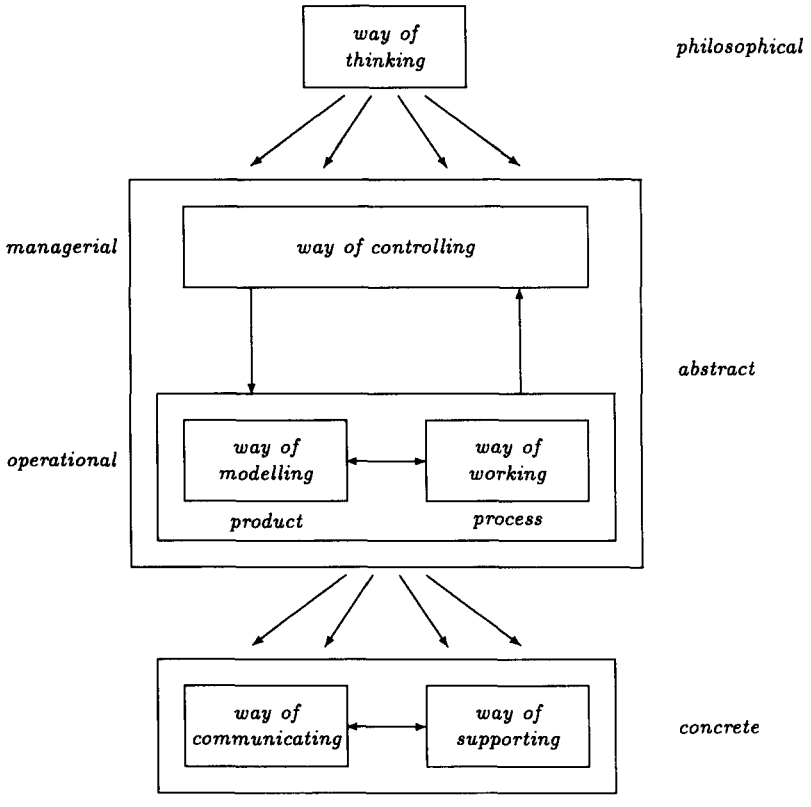



Fig. 1. Framework for methodologies.

and in which order. The *way of modelling* provides a mathematical (abstract) description of the underlying concepts, their properties and behaviour. The concrete level is a materialization of the development process. The *way of communicating* describes how the abstract notions are visualized (communicated) to human beings, for example in the style of a conceptual language (such as Elisa-D). Usually, the way of communicating provides a graphical notation. It may very well be the case that different methods are based on the same way of modelling, but use a different graphical notation. The *way of supporting* deals with tools supporting the development process.

In this paper, we first describe the underlying way of thinking for evolving information systems used in this paper. Next we provide a general way of modelling (see also [42]), making only weak assumptions on the underlying method. As a result, this approach is applicable for a wide range of data modelling methods, such as ER [11], NIAM [36] and PSM [26, 24], action modelling methods such as Task Structures [23], DFD [9] and ExSpect [21] and furthermore object oriented modelling methods [31]. Since in our approach the main focus is on object identity, we postulate a typing mechanism for objects, a type relatedness relation expressing which object types may share instances, and a hierarchy on object types expressing inheritance of identification.

This typing mechanism is captured by a set of rules (ISU: Information Structure Universe),

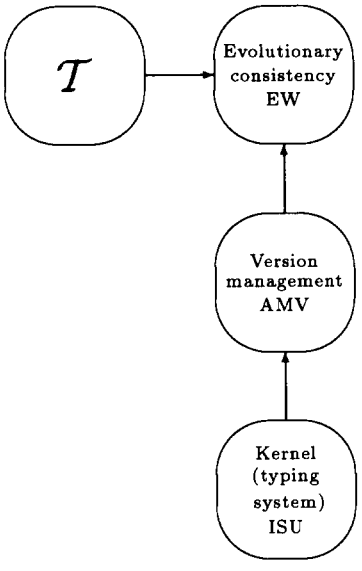



Fig. 2. Axiomatic framework.

and forms the basis for version management. This leads to a set of rules (AMV: Application Model Version) describing wellformedness of versions. The version management, on its turn, serves together with a time axis $\mathcal{T}$ as the base for rules (EW: Evolution Wellformedness), describing what constitutes a wellformed evolution of an information system. These dependencies are illustrated in Fig. 2.

When applying the general evolution theory to a concrete (data) modelling technique, the modelling technique must provide:

(1) a typing system conforming to the typing axioms of the general theory (ISU axioms),
(2) wellformedness rules for versions of the models (AMV axioms).

This enables us to accomplish the main goal of this paper, the introduction of the data modelling technique EVORM (Evolutionary Object Role Modelling) as an application of the general theory on PSM, a snapshot-oriented technique for data modelling. The application of the general theory on PSM also provides a good test case for the general theory.

PSM originated from PM [6] being a formalisation of NIAM [36]. Another formalisation of NIAM, resulting in FORM, can be found in [19, 20]. PSM can be regarded as a common base for object role modelling techniques like NIAM, FORM, ER [11], EER [27] and IFO [1].

Although the introduction of the general theory is a substantial and essential part of this paper, focus is on the introduction of EVORM, contrary to [40] which addresses the general theory itself.

## 2. Modelling the evolution of information systems

In this section we discuss our approach to evolving information systems. We start with a hierarchy of models, which together constitute a complete specification of (a version of) a

universe of discourse (application domain). Using this hierarchy, we are able to identify that part of an information system that may be subject to evolution. From this identification, the difference between a traditional information system, and its evolving counterpart, will become clear. This is followed by a discussion on how the evolution of an information system is modelled.

### 2.1. A hierarchy of models

According to [18], a conceptual (i.e. complete and minimal) specification of (a version of) a universe of discourse consists of the following components:

(1) an *information structure*, a set of *constraints* and a *population* conforming to these requirements.
(2) a set of *action specifications* describing the transitions that can be performed by the system.

The set of action specifications in such a specification is referred to as the *action model*. The action model describes all possible transitions on populations, and is usually modelled by means of Petri-net like specifications (such as ExSpect or Task Structures), or languages such as SQL. The *world model* encompasses the combination of information structure, constraints and population. A conceptual specification of a universe of discourse, containing both the action and world model, is called an *application model* [16, 42]. The resulting hierarchy of models is depicted in Fig. 3.

The application model of a universe of discourse is denoted in terms of object types, constraints, instantiations, action specifications, etc. As a collective noun for these modelling concepts the term *application model element* is used. In an evolving information system, the complete application model, described as a set of application model elements, is allowed to change in the course of time.

In most traditional information systems, however, the evolvable part of the application model is restricted to the population. Nevertheless, some traditional information systems do support modifications of other components from the application model, to a limited extent. For example, adding a new table in an SQL system is easily done. However, changing the arity of a table, or some of its attributes, will result in a time consuming table conversion,

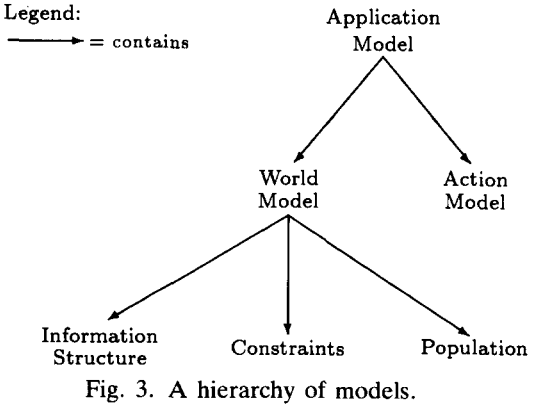



Fig. 3. A hierarchy of models.

which also leads to loss of the old table! In an evolving information system, the entire application model is allowed to evolve on-line, without loss of any information. The application model can then be looked upon as the formal denotation of the corpus evolutionis.

### 2.2. An example of evolution

As an illustration of an evolving universe of discourse, consider an insurance company for cars. For each policy sold, the insured car and client are recorded. Every insured car has associated its registration number and type (Opel Corsa 1.2S, Ford Sierra 1.8, etc.). A client is identified by name and address. The information structure of this universe of discourse is modelled in Fig. 4 in the style of ER. Note the special notation of attributes (`Type`) using a mark symbol (#) followed by the attribute (`#Type`).

After some time, the insurance company noticed a substantial difference between damage claims made for private cars, and for company cars. Rather than raising overall policy prices, a price differentiation was effectuated. For company owned cars, prices for new policies were increased by some percentage. Prices for new policies for private cars, however, were made dependent on the car usage, measured in kilometers per year.

As these changes in price only involve new policies, the current population of the schema did not have to be altered. The evolved information structure is depicted in Fig. 5. The differentiation between private and company cars, has led to a subtyping of cars, and the dependency of the policy price on the amount of driven kilometers has led to the introduction of an extra entity type (`Kilometrage`) and relation type (`Usage`). As a result of this change, instances of `Car` are also distributed over object types `Company car` and `Private car`, according to the subtype defining rule associated with this subtyping. Furthermore, a method to initialize the kilometrage of private cars is introduced:

```
WHEN ADD Car:x
  IF Private car:x THEN
    ADD Private Car:x has a Usage of Kilometrage:0
```

using a Lisa-D like notation ([25]).

A large number of small companies, not intensively using their cars, started to protest against new policy pricing, threatening to accommodate their policies elsewhere. Thereupon, the insurance company decided to differentiate pricing for business cars on usage as well. As a result, subtyping cars into business cars and private cars was abolished. A further means to be more competitive, was found in the introduction of a reduction for clients not claiming much damage. This reduction depends on the number of damage free years. This requires an adaptation of the kilometrage initialization method:

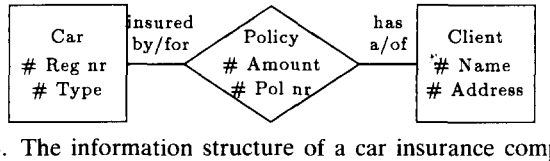



Fig. 4. The information structure of a car insurance company.

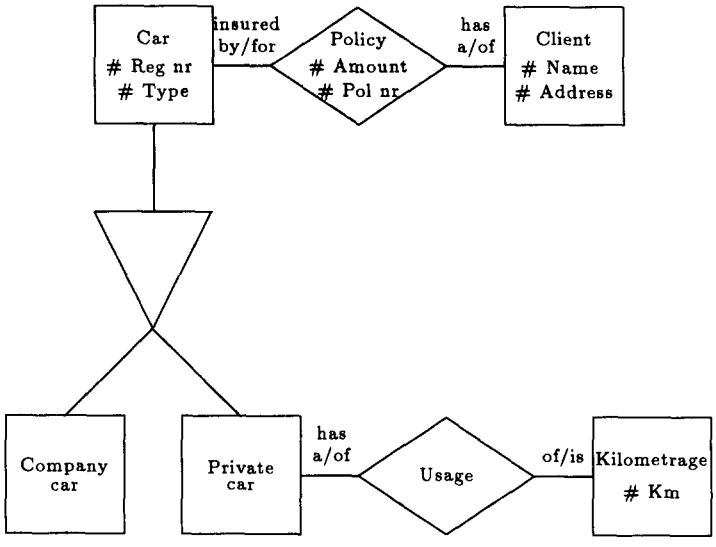



Fig. 5. Car insurance with differentiated pricing.

```
WHEN ADD Car:x
  ADD Car:x has a Usage of Kilometrage:0
```

For the information structure, this leads to

(1) the abolishment of the subtyping of `Car` into `Company car` and `Private car`,

(2) the introduction of the attribute `# Reduction` for relationship `Policy`.

This results in Fig. 6. The abolishment of the subtyping for cars requires an extension of the `Usage` relation to (former) `Company cars`. Note that instances of the `Usage` relation for `Private cars` automatically become instances of the modified `Usage` relation, as each instance of (former) `Private car` is also an instance of `Car`. The introduction of the reduction, also requires a change in the current population of the information structure, as an initial reduction must be issued. This could, for example, be effectuated by the following transaction (in the style of SQL):

```
ADD TO Policy MANDATORY ATTRIBUTE Reduction;
UPDATE Policy SET Reduction = '20%'
```

### 2.3. *The approach*

The three ER schemata, and the associated action specifications, as discussed above, correspond to three distinct snapshots of an evolving universe of discourse. Several approaches can be taken to the modelling of this evolution (see for a more elaborate discussion

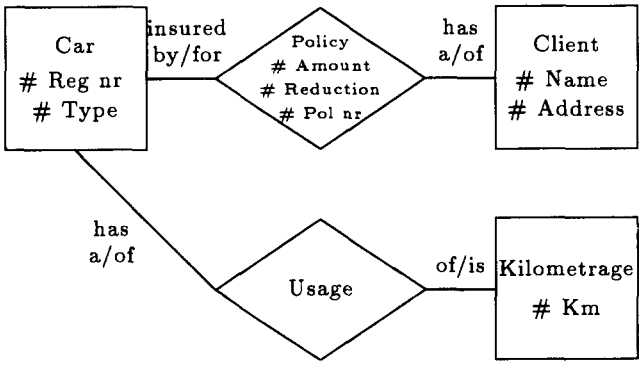



Fig. 6. The final information structure.

[42, 40]). In this paper, we treat evolution of an application model as a separate concept. We will maintain the evolution of distinct application model elements, thus keeping track of the evolution of individual object types, instances, methods, etc. This has been illustrated in Fig. 7. Each dotted line corresponds to the evolution of one distinct element.

This approach enables one to state rules about, and query, the evolution of distinct application model elements. Furthermore, a snapshot view, showing the distinct versions of the application models in the course of time, can be derived by constituting the application model version of any point of time from the current versions of its components. This derivation is examplified in Fig. 8.

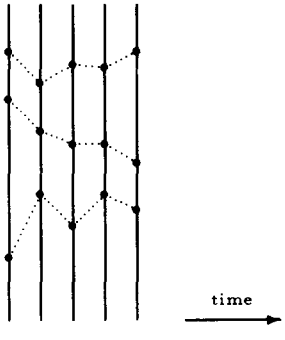



Fig. 7. Evolution modelled by functions over time.

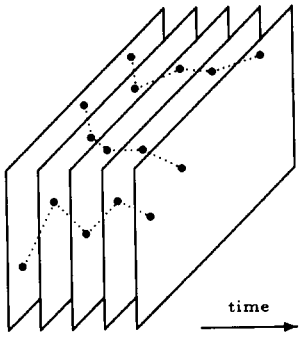



Fig. 8. Deriving snapshots from element evolutions.

## 2.4. The formal model

We are now in a position to formally introduce evolving information systems. The intention of an evolving information system is to describe an *application model history*.[1] An application model history in its turn, is a set of (*application model*) *element evolutions*. Each element evolution describes the evolution of a specific application model element. An element evolution is a partial function assigning to points of time the actual occurrence (version) of that element.

An example of an element evolution is the evolution of the relation type named `Usage` in the insurance company. This relation starts out as an association between private cars and kilometrages. After the abolishment of the differentation between company car and private car, the version of the application model element `Usage` is changed to a relation between (all) cars and kilometrages.

The domain $\mathcal{AMH}$ for application model histories is determined by the *living space* of the evolving information system. The living space is defined by the following components:

(1) Time, essential to evolution, is incorporated into the theory through the algebraic structure $\mathcal{T} = \langle T, F \rangle$, where $T$ is a (discrete, totally ordered) time axis, and $F$ a set of functions over $T$. For the moment, $F$ is assumed to contain the one-step increment operator $\triangleright$, and the comparison operator $\leqslant$. Several ways of defining a time axis exist, see e.g. [12, 54 or 2].

Other time models are possible, for example, in distributed systems a relative time model might be used. For a general survey on time models, see [44]. The linear time model is usually chosen in historical databases (see for example [49]).

(2) The set $\mathcal{AME}$ is the domain for the evolvable elements of an application model, and

[1] In this paper, the difference between recording and event time [50], and the ability to correct stored information are not taken into consideration. For more details, see [16] or [17].

thus each element evolution $h$ has the signature of a partial mapping: $h : T \rightarrowtail \mathcal{AME}$[2]. A formal definition of $\mathcal{AME}$ will be provided in Section 5.

Consequently, an application model history $H$ is a set of (partial) mappings:

$$H \subseteq T \rightarrowtail \mathcal{AME}$$

or, $H \in \wp(T \rightarrowtail \mathcal{AME})$. The set $\mathcal{AMH}$ is thus identified by:

$$\mathcal{AMH} = \wp(T \rightarrowtail \mathcal{AME}) .$$

In a later section, we will pose wellformedness restrictions on histories.

(3) $\mathcal{M}$ is the domain for actions that can be performed on application model histories. The semantics of these actions are provided by the state transition relation on application model histories:

$$[\![\ ]\!] \subseteq \mathcal{M} \times T \times \mathcal{AMH} \times \mathcal{AMH}$$

where $H[\![m]\!]_t H'$ means: $H'$ may result after applying action $m$ to $H$ at time $t$. Usually, however, actions are deterministic.

In this article we do not take the semantics of actions into consideration, and focus on data modelling aspects, with a special emphasis on object identity.

## 3. Generalised application models

The kernel of the application model universe is formed by the *information structure universe*, fixing the evolution space for information structures. The application model universe is a demarcation for the evolution of application models. This universe is centered around the information structure universe, as all other elements of the application model (see Fig. 3) refer to the information structure.

In this section, we first introduce the information structure universe, mainly focussing on object typing, type relatedness and identification of objects. After that, we derive a number of properties of the information structure universe, which will be useful in a later section, when applying the general theory to the concrete data modelling technique PSM. A special class of properties is concerned with the inheritance of identification, resulting in the identification hierarchy. In a concrete modelling technique, several flavours of inheritance will be distinguishable. In Subsection 3.3 we pay special attention to the partitioning of the identification hierarchy in those flavours. In this paper, the focus is on data modelling aspects of evolution. The remaining components of application model universe are briefly addressed in Subsection 3.4.

### 3.1. The information structure universe

The information structures which are part of the application models, are bound to the information structure universe. The information structure universe, for a given modelling

[2] In this paper, $\rightarrowtail$ is used for partial functions, and $\rightarrow$ for total functions.

technique, is determined by a set $\mathcal{O}$ of object types, together with relations $\sim$ and $\rightsquigarrow$ defined over it. The relation $\sim$ captures relatedness between object types. Inheritance of identification of object types is described in the relation $\rightsquigarrow$.

The actual universe then is formed by all subsets of $\mathcal{O}$. An information structure version is identified by a set of object types $\mathcal{O}_t \subseteq \mathcal{O}$, where $\sim$ and $\rightsquigarrow$ specify the structure within the version.

Not all sets of object types will correspond to a correct information structure, therefore we introduce the criterion `IsSch` (is schema) determining the set of proper information structure universes. In the sequel, the information structure universe will be defined by its components, which will be explained further below.

**Definition 3.1.** $\mathcal{U}_{\mathcal{G}}$ for information structures is determined by the structure:

$$\mathcal{U}_{\mathcal{G}} = \langle \mathcal{L}, \mathcal{N}, \sim, \rightsquigarrow, \texttt{IsSch} \rangle$$

where $\mathcal{O} = \mathcal{L} \cup \mathcal{N}$.

$\mathcal{L}$ are label object types, $\mathcal{N}$ are abstract object types. The components of the information structure universe are discussed in more detail in the next subsections.

Further refinements of the information structure universe depend on the chosen data modelling technique. In Section 6 this will be elaborated for the PSM case, resulting in EVORM. In the general theory, an information structure universe is assumed to provide (at least) the above components, which are available in all conventional high level data modelling techniques. These components are discussed below.

### *3.1.1 Object types*

The central part of an information structure is formed by its object types (referred to as object classes in object oriented approaches). Two major classes of object types are distinguished. Object types whose instances can be represented directly (denoted) on a medium (strings, natural numbers, etc.) form the class of label types $\mathcal{L}$. The other object types, for instance entity types or fact (relation) types, form the class $\mathcal{N}$. For an information structure version $\mathcal{O}_t$, the set of actual label types and non-label types is defined by: $\mathcal{L}_t = \mathcal{L} \cap \mathcal{O}_t$ and $\mathcal{N}_t = \mathcal{N} \cap \mathcal{O}_t$. The validity of $\mathcal{O}_t$ is designated by the predicate `IsSch`.

The example of Fig. 4 contains the following object types: entity types `Car`, and `Client`, relation type `Policy`, and label types `Name`, `Reg-nr`, `Type`, `Pol-nr`, `Amount` and `Address`.

### *3.1.2 Type relatedness*

The relation $\sim \subseteq \mathcal{O} \times \mathcal{O}$ expresses *type relatedness* between object types (see [26]). Object types $x$ and $y$ are termed type related ($x \sim y$) iff populations of object types $x$ and $y$ *may* have values in common in any version of the application model. Type relatedness corresponds to mode equivalence in programming languages [56]. Typically, subtyping and generalisation lead to type related object types. For the data model depicted in Fig. 4, the type relatedness relation is the identity relation: $x \sim x$ for all object types $x$.

An example of a more complex type relatedness relation is provided in the PSM data model

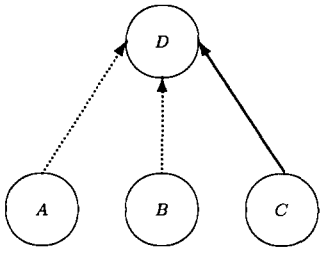



Fig. 9. A data model with generalisation and specialisation.

in Fig. 9. In this example, $A$, $B$, $C$, $D$ are object types, the solid arrow stands for a subtyping (specialisation) relation, whereas the dotted arrows represent generalisations. A major difference between generalisation and specialisation is that the population of subtypes is defined by means of a subtype defining rule in terms of the population of the supertype, whereas a generalised object type directly inherits the complete populations from its specificers [1, 26]. The type relatedness relation for the data model of Fig. 9 is therefore: $D \sim A$, $D \sim B$, $D \sim C$, $C \sim A$ and $C \sim B$.

According to the intuitive meaning of type relatedness, this relation is required to be reflexive and symmetrical:

**[ISU1]** (*reflexive*). $x \sim x$.

**[ISU2]** (*symmetrical*). $x \sim y \Rightarrow y \sim x$.

Note that the relation $\sim$ is not transitive in general. For example, in Fig. 9 we have $A \sim D$ and $D \sim B$, but not $A \sim B$. The separation of the concrete and abstract worlds has the following consequence for type relatedness:

**[ISU3]** (*separation*). $x \sim y \Rightarrow x, y \in \mathscr{L} \vee x, y \in \mathscr{N}$.

*3.1.3 The identification hierarchy*

In data modelling, a crucial role is played by the notion of object identification: each object type of an information structure should be identifiable. In a subtype hierarchy, a subtype inherits its identification from its super type, whereas in a generalisation hierarchy the identification of a generalised object type is inherited from its specifiers. For the data model depicted in Fig. 9 this means that instances of $C$ are identified in the same way as instances of $D$. The identification of instances from $D$ depends on the identification of instances from $A$ or $B$ (note that an instance from $D$ is either an instance from $A$ or an instance from $B$). For the data model depicted in Fig. 5, it means that instances of `Private-car` and `Company-car` are identified in the same way as instances of `Car`.

An object type from which identification is inherited, is termed an *ancestor* of that object type. The identification hierarchy is provided by the relation $x \rightsquigarrow y$, meaning $x$ is an ancestor of $y$. For Fig. 5 this leads to: `Car`$\rightsquigarrow$`Company-car` and `Car`$\rightsquigarrow$`Private-car`. The identification hierarchy corresponding to Fig. 9 is:

$$\left.\begin{array}{l} A \rightsquigarrow D \\ B \rightsquigarrow D \end{array}\right\} D \rightsquigarrow C \,.$$

The identification hierarchy is both transitive and irreflexive.

**[ISU4]** (*irreflexive*). $\neg x \rightsquigarrow x$.

**[ISU5]** (*transitive*). $x \rightsquigarrow y \rightsquigarrow z \Rightarrow x \rightsquigarrow z$.

Similar axioms can be found as properties in literature about typing theory for databases [8, 37 and 10]. The difference between these properties and ours lies in the abstraction of an underlying structure of object types and their instances. As we do not make any assumption on these structures, such properties must be stated as axioms. Another reason is that the inheritance hierarchy is intertwined with type relatedness, requiring appropriate axioms.

Object types without ancestor, are called *roots*: $\texttt{Root}(x) \stackrel{\triangle}{=} \neg\exists_z[z \rightsquigarrow x]$. We will write $x \underline{\rightsquigarrow} y$ as an abbreviation for $x = y \vee x \rightsquigarrow y$. The roots $x$ of an object type $y$ are found by:

$$x \texttt{ RootOf } y \stackrel{\triangle}{=} \texttt{Root}(x) \wedge x \underline{\rightsquigarrow} y \,.$$

This relation is idempotent:

**Corollary 3.1** (*idempotency*).

$$x \texttt{ RootOf } y \Rightarrow x \texttt{ RootOf } x \,.$$

Note that the identification hierarchy of Fig. 9 object types $C$ and $D$ have multiple roots. Next we focus at direct ancestors of object types within the identification hierarchy. We call $p$ a direct ancestor (a parent) of $x$, denoted as $\texttt{Parent}(p, x)$, if:

$$p \rightsquigarrow x \wedge \neg\exists_z[p \rightsquigarrow z \rightsquigarrow x] \,.$$

The existence of direct ancestors is postulated by:

**[ISU6]** (*direct ancestors*).

$$a \rightsquigarrow x \Rightarrow \exists_p[a \underline{\rightsquigarrow} p \wedge \texttt{Parent}(p, x)] \,.$$

The (complete) identification of a non-root object type is derived from the identification properties of its ancestors. Thus, the identification of a non-root object type can only be complete if all its ancestors are. This is expressed by the following schema of induction:

**[ISU7]** (*parent induction*).

$$\forall_{x:\texttt{Parent}(p,x)}[F(p)] \Rightarrow F(x), \text{ then } \forall_{x \in \mathcal{O}}[F(x)] \,.$$

Note that the base step ($\texttt{Root}(x) \Rightarrow F(x)$) is contained in this schema of induction, as root object types have no ancestors. This axiom leads to the following, more convenient induction schema:

**Theorem 3.1** (*ancestor induction*).
*If for all* $x$: $\forall_{a:\, a \rightsquigarrow x}[F(a)] \Rrightarrow F(x)$, *then* $\forall_{x \in \mathcal{O}}[F(x)]$.

**Proof.** Suppose $F$ has property $\forall_{a:\, a \rightsquigarrow x}[F(a)] \Rrightarrow F(x)$.

Consider $G(x) = \forall_{a:\, a \rightsquigarrow x}[F(a)]$. Using parent induction, we will prove the stronger property $\forall_{x \in \mathcal{O}}[G(x)]$, which directly implies $\forall_{x \in \mathcal{O}}[F(x)]$.

Suppose all parents $p$ of $x$ have the property $G(p)$. Let $a$ be an ancestor of $x$, then there exists a parent $p$ between $a$ and $x$ (axiom ISU6). From the induction hypothesis we conclude $G(p)$, and thus $F(a)$. As all ancestors $a$ of $x$ have property $F(a)$, we conclude $F(x)$, and consequently $G(x)$. □

*3.1.4 Inheritance of type relatedness*

In this section we introduce two special types of properties with respect to inheritance, for which we will prove general theorems in Subsection 3.2. A strong kind of inheritance occurs when a property is preserved from a parent to all its children. Therefore, we say that property $P$ *is preserved by* relation $R$, if for all $x$, $y$:

$$P(x) \wedge R(x, y) \Rrightarrow P(y) .$$

A weak kind of inheritance is when a property can be traced back to root object types. In this case, properties of object types are a reflection of properties of their ancestors. Therefore, we say that property $P$ *is reflected by* relation $R$, if for all $x$:

$$P(x) \wedge \exists_a[R(a, x)] \Rrightarrow \exists_a[P(a) \wedge R(a, x)] .$$

If a relation $P$ is both reflected and preserved by relation $R$, then $P$ is said to be *filled by* relation $R$. In this case, if some object type has the property, then a complete subhierarchy (containing this object type) has this property, i.e. is filled by this property. Note that the naming conventions used (preserved, reflected, filled) are adopted from Category Theory [5].

We will introduce the inheritance of type relatedness as a filled property. For this purpose, the relation $P_x$ is defined by $P_x(y) \triangleq x \sim y$ for all $x$. From the intuition behind the ancestor relation it follows that each instance of an object type originates from some ancestor. Since type relatedness captures the intuition of object types sharing instances, this property may be enforced by the requirement of $P_x$ being reflected by $\rightsquigarrow$. Note that instances of object types not necessarily originate from *all* ancestors (e.g., a multi-rooted hierarchy such as in Fig. 9). On the other hand, an instance of an object type is also an instance of all its children. This implies that object types not only inherit identification from their ancestors, but type relatedness as well (preservation). These requirements are laid down in the following axiom:

**[ISU8]** (*inheritance and foundation of type relatedness*). The relation $P_x$ is filled by $\rightsquigarrow$, for all $x$.

Some immediate consequences are:

**Corollary 3.2.** $x \rightsquigarrow y \Rrightarrow x \sim y$.

**Corollary 3.3.** $x$ `RootOf` $y \Rightarrow x \sim y$.

Some examples of properties which are also filled by $\rightsquigarrow$ are:

(1) *true*.
(2) *false*.
(3) The relation $Q_r$, defined by $Q_r(x) \stackrel{\triangle}{=} r$ `RootOf` $x$.
(4) The relation $I_i$, defined as $I_i(x) \stackrel{\triangle}{=}$ '$i$ is an instance of $x$'.

### *3.1.5 An example: ER*

For every data model from conventional data modelling techniques, an ancestor and root relation can be derived. If no specialisations or generalisations are present in a particular data model, the associated ancestor relation will be empty. As a result, the root relation will then be the identity relation.

For Chen's [11] ER model (extended with subtyping), the information structure universe will be:

*Label types*. The set of label types $\mathscr{L}$ in ER corresponds to the printable attribute types. Note that in some ER versions, entity types can be used as attribute for other entity types.

*Non-label types*. The set of non-label types $\mathscr{N}$ is defined as the set of relationship types, entity types and associative object (entity) types.

*Inheritance*. Traditional ER only contains the notion of subtyping. So for each subtype $x$ of a supertype $y$ we have: $y \rightsquigarrow x$. The complete inheritance relation $\rightsquigarrow$ is then obtained by applying the transitive closure.

*Type relatedness*. Two subtypes of the same supertype are type related. Furthermore, subtyping is the only way in ER to make type related object types. Furthermore, a subtyping hierarchy has a unique top element. Let $\sqcap(x)$ denote the unique top element of the subtyping hierarchy containing object type $x$. As a result, type relatedness for ER is defined as: $x \sim y \stackrel{\triangle}{=} \sqcap(x) = \sqcap(y)$.

*Schema wellformedness*. The predicate `IsSch` can be described according to ER rules. This will be omitted in this paper.

The information structure universe axioms are easily verified. The type relatedness axioms ISU1, ISU2 and ISU3 are immediate consequences of the above definition. The identification hierarchy axioms ISU4, ISU5, ISU6 and ISU7 directly follow from the nature of subtyping in ER.

## *3.2. Properties of information structures*

In this section we present a number of properties for information structure universes, that will prove to be useful. The first general theorem is concerned with inheritance of properties, and states that preservation of a property implies the validity of the property for all descendants.

**Theorem 3.2** (*inheritance schema*). *If property P is preserved by* $\rightsquigarrow$, *then it is also preserved by* `RootOf`:

$$P(x) \wedge x \texttt{ RootOf } y \Rightarrow P(y) .$$

**Proof.** Let property $P$ be preserved by $\rightsquigarrow$. Suppose $P(x) \wedge x$ `RootOf` $y$. Then we conclude $P(x) \wedge x \underline{\rightsquigarrow} y$ from the definition of `RootOf`, and thus, as $P$ is preserved by $\rightsquigarrow$, we have $P(y)$. □

The second general theorem is concerned with the foundation of properties:

**Theorem 3.3** (*basic foundation schema*). *If property P is reflected by* $\rightsquigarrow$, *then it is also reflected by* `RootOf`:

$$P(y) \wedge \neg \texttt{Root}(y) \Rightarrow \exists_x [P(x) \wedge x \texttt{ RootOf } y] .$$

**Proof.** Suppose $P$ is reflected by $\rightsquigarrow$. We apply ancestor induction, and assume that for all ancestors $a$ of some object type $y$, the property has been proven. In order to prove the property for $y$, we suppose $P(y)$. Let furthermore, $y$ to be a non-root. As a result, $a \rightsquigarrow y$ for some $a$. Applying the induction hypothesis, we find object type $x$, such that $P(x) \wedge x$ `RootOf` $a$. From the transitivity of $\rightsquigarrow$ we get $P(x) \wedge x$ `RootOf` $y$. □

From this theorem, and the observation that $y$ `RootOf` $y$ if `Root`$(y)$, the following, more convenient, formulation of the (base) foundation schema can be derived:

**Lemma 3.1** (*foundation schema*). *If property P is reflected by* $\rightsquigarrow$, *then*:

$$P(y) \Rightarrow \exists_x [P(x) \wedge x \texttt{ RootOf } y]$$

*3.2.1 Filled properties*

As stated before, a property $P(x)$ which is both reflected and preserved by a relation $R$, is said to be filled by relation $R$. For properties filled by $\rightsquigarrow$, the inheritance can be traced from parents:

**Lemma 3.2.** *If property P is filled by* $\rightsquigarrow$, *then P is reflected by* `Parent`:

$$P(x) \wedge \neg \texttt{Root}(x) \Rightarrow \exists_p [P(p) \wedge \texttt{Parent}(p, x)] .$$

**Proof.** Let $P$ be filled by $\rightsquigarrow$, and furthermore suppose $P(x) \wedge \neg$`Root`$(x)$. As $P$ is reflected by $\rightsquigarrow$, we have $P(a) \wedge a \rightsquigarrow x$ for some $a$. By applying axiom ISU6 we know $a \underline{\rightsquigarrow} p \wedge$ `Parent`$(p, x)$ for some $p$ (see also Fig. 10). From $P(a)$ and $a \underline{\rightsquigarrow} p$ and the preservation of $P$ by $\rightsquigarrow$, we then conclude $P(p)$. □

*3.2.2 Some special inheritance properties*

In this section we discuss some special inheritance properties filled by `RootOf`, which will be used in proofs in the sequel.

*The relation* $P_x$. The first inheritance property under consideration is the relation $P_x$, which,

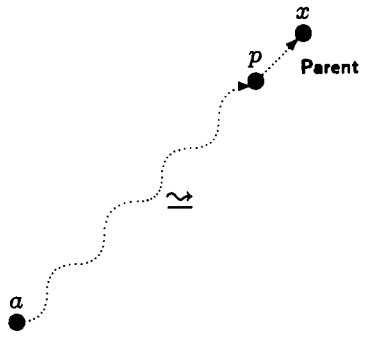



Fig. 10. Existing ancestors.

for all $x$, was defined as $P_x(y) \equiv x \sim y$. Applying Theorem 3.2, Lemma 3.1 and Lemma 3.2 respectively on relation $P_a$ yields:

**Corollary 3.4.** *Relation* $\sim$ *is preserved by* `RootOf`:

$$a \sim x \wedge x \texttt{ RootOf } y \Rightarrow a \sim y\,.$$

**Corollary 3.5.** *Relation* $\sim$ *is reflected by* `RootOf`, *and can be formulated stronger* as:

$$a \sim y \Rightarrow \exists_x [a \sim x \wedge x \texttt{ RootOf } y]\,.$$

**Corollary 3.6.** *Relation* $\sim$ *is reflected by* `Parent`:

$$a \sim x \wedge \neg \texttt{Root}(x) \Rightarrow \exists_p [a \sim p \wedge \texttt{Parent}(p, x)]\,.$$

If two object types are type related, then they may share instances. Using the above results, this implies that if two object types share a root, they should be type related. This is formulated in the following lemma:

**Lemma 3.3.** $v \texttt{ RootOf } x \wedge v \texttt{ RootOf } y \Rightarrow x \sim y$.

**Proof.** Suppose $v \texttt{ RootOf } x \wedge v \texttt{ RootOf } y$, then $x \sim v$ and $v \texttt{ RootOf } y$. Applying corollary 3.4 yields $x \sim y$. □

The following theorem, which is illustrated in Fig. 11, shows that type relatedness of object types is equivalent to type relatedness of roots:

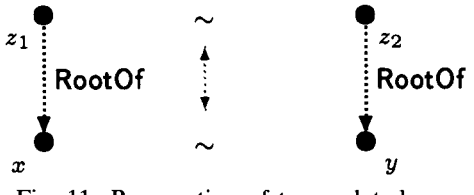



Fig. 11. Propagation of type relatedness.

**Theorem 3.4** (*type relatedness propagation*).

$$x \sim y \Leftrightarrow \exists_{z_1,z_2}[z_1 \sim z_2 \wedge z_1 \,\mathtt{RootOf}\, x \wedge z_2 \,\mathtt{RootOf}\, y] .$$

**Proof.**

$\Rightarrow$ Suppose $x \sim y$. From Corollary 3.5 follows the existence of an $r$ such that $x \sim r \wedge r \,\mathtt{RootOf}\, y$. When applying Corollary 3.5 to $r \sim x$, the existence of an $s$ such that $r \sim s \wedge s \,\mathtt{RootOf}\, x$ follows.

$\Leftarrow$ Suppose $z_1 \sim z_2 \wedge z_1 \,\mathtt{RootOf}\, x \wedge z_2 \,\mathtt{RootOf}\, y$ for some $z_1$ and $z_2$. As $z_1 \sim z_2 \wedge z_1 \,\mathtt{RootOf}\, x$, we conclude from Corollary 3.4 that $x \sim z_2$. Another application of Corollary 3.4 yields $x \sim y$. □

This theorem allows on its term for the formulation of the following theorem, expressing the intuition that if two object types share all their roots, they share all their type related object types as well.

**Theorem 3.5.**

$$x \sim y \wedge \forall_r[r \,\mathtt{RootOf}\, y \Rightarrow r \,\mathtt{RootOf}\, z] \Rightarrow x \sim z .$$

**Proof.** Suppose $x \sim y$, then $z_1 \sim z_2 \wedge z_1 \,\mathtt{RootOf}\, x \wedge z_2 \,\mathtt{RootOf}\, y$ for some $z_1$ and $z_2$. As each root of $y$ is also a root of $z$, we conclude then $z_1 \sim z_2 \wedge z_1 \,\mathtt{RootOf}\, x \wedge z_2 \,\mathtt{RootOf}\, z$, and thus $x \sim z$. This proof is illustrated by means of Fig. 12. □

*The relation True.* Next we consider the property *True*. In this case, Theorem 3.2 leads to an obvious statement. Applying Lemma 3.1, however, results in:

**Corollary 3.7.** $\forall_{y \in \mathcal{O}} \exists_x] x \,\mathtt{RootOf}\, y]$.

*The relation $Q_r$*

The property $Q_r$, which has been defined as $Q_r(x) \triangleq r \,\mathtt{RootOf}\, x$, is filled by $\rightsquigarrow$. For this property, Theorem 3.2 and Lemma 3.1 are trivial statements. From Lemma 3.2 we get:

**Corollary 3.8.**

$$r \,\mathtt{RootOf}\, x \wedge \neg \mathtt{Root}(x) \Rightarrow \exists_p[r \,\mathtt{RootOf}\, p \wedge \mathtt{Parent}(p, x)] .$$

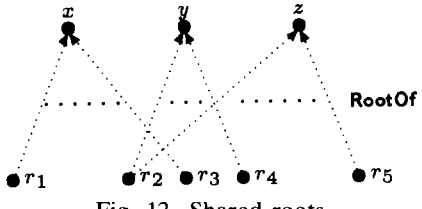



Fig. 12. Shared roots.

### 3.3. Filtering a hierarchy

In the remainder of this article, when applying the general evolution theory to PSM, we will have to prove some properties on sub-hierarchies of the identification hierarchy. For instance the identification hierarchy restricted to specialisation only, or to generalisation only. Generally, we call a binary relation $R \subseteq \mathcal{O} \times \mathcal{O}$ a *filter relation* on the identification hierarchy $\rightsquigarrow$ if:

**[F1]** (*transitivity completeness*). If $x \rightsquigarrow y \rightsquigarrow z$, then:

$$R(x, y) \wedge R(y, z) \Leftrightarrow R(x, z)$$

**[F2]** (*choice completeness*). If $\texttt{Parent}(p, x)$ and $\texttt{Parent}(q, x)$, then

$$R(p, x) \Rightarrow R(q, x) .$$

The filtered identification hierarchy $\rightsquigarrow_R$ then is defined by:

$$x \rightsquigarrow_R y \triangleq R(x, y) \wedge x \rightsquigarrow y .$$

In this case we will speak of $R$-ancestors rather than ancestors. As before, $x \underline{\rightsquigarrow}_R y$ is used as a shorthand for $x = y \vee x \rightsquigarrow_R y$. We call $p$ a direct $R$-ancestor (an $R$-parent) or $x$, denoted as $\texttt{Parent}_R(p, x)$, if:

$$p \rightsquigarrow_R x \wedge \neg \exists_a [p \rightsquigarrow_R a \rightsquigarrow_R x] .$$

In the resulting sub-hierarchy, object types may have no $R$-ancestors:

$$\texttt{Root}_R(x) \triangleq \neg \exists_z [z \rightsquigarrow_R x] .$$

Such object types are denoted as $R$-roots, and are found by:

$$x \, \texttt{RootOf}_R \, y \triangleq x \underline{\rightsquigarrow}_R y \wedge \texttt{Root}_R(x) .$$

The following lemma states under what condition $R$-parents are guaranteed.

**Lemma 3.4.**

$$\texttt{Parent}(p, x) \wedge R(p, x) \Leftrightarrow \texttt{Parent}_R(p, x) .$$

**Proof.**

$\Rightarrow$ Suppose $\texttt{Parent}(p, x) \wedge R(p, x)$. Then obviously $p \rightsquigarrow_R x$. Now suppose for some $z$ we have $p \rightsquigarrow_R z \rightsquigarrow_R x$. Then also $p \rightsquigarrow z \rightsquigarrow x$, which contradicts $\texttt{Parent}(p, x)$.

$\Leftarrow$ Suppose $\texttt{Parent}_R(p, x)$, then obviously $p \rightsquigarrow x$ and $R(p, x)$. Now suppose for some $z$ we have $p \rightsquigarrow z \rightsquigarrow x$. From $R(p, x)$ and axiom F1 we then conclude $R(p, z) \wedge R(z, x)$, which contradicts $\texttt{Parent}_R(p, x)$. □

Some direct consequences are:

**Corollary 3.9.**

$$\texttt{Parent}_R(p,x) \wedge \texttt{Parent}(q,x) \Rightarrow \texttt{Parent}_R(q,x)\,.$$

**Corollary 3.10.**

$$\forall_{p:\texttt{Parent}(p,x)}[F(p)] \Rightarrow \forall_{q:\texttt{Parent}_R(q,x)}[F(q)]\,.$$

Filtering an inheritance relation leads to a proper information structure universe:

**Theorem 3.6.** *If $R$ is a filter relation, then $\langle \mathscr{L}, \mathscr{N}, \sim, \rightsquigarrow_R, \texttt{IsSch}\rangle$ is an information structure universe.*

**Proof.** We will prove $\rightsquigarrow_R$ versions of the ISU axioms. The validity of $\text{ISU1}_R$, $\text{ISU2}_R$, $\text{ISU3}_R$ and $\text{ISU4}_R$ is obvious, while $\text{ISU5}_R$ is a direct consequence of F1. The remaining axioms:

**axiom** $\text{ISU6}_R$. Suppose $a \rightsquigarrow_R x$, then $a \rightsquigarrow x \wedge R(a,x)$. Let $p$ be a parent of $x$ between $a$ and $x$. From axiom F1 we conclude that $p$ is also an $R$-parent of $x$ between $a$ and $x$ (see Lemma 3.4). The existence of an $R$-parent then follows from the existence of a parent of $x$ between $a$ and $x$.

**axiom** $\text{ISU7}_R$. Let $F$ be a property such that $\forall_{a:\texttt{Parent}_R(a,x)}[F(a)] \Rightarrow F(x)$. Suppose furthermore that $\forall_{q:\texttt{Parent}(q,x)}[F(q)]$, then from Corollary 3.10 it follows $\forall_{q:\texttt{Parent}_R(q,x)}[F(q)]$. From the first assumption follows $F(x)$. As a result: $\forall_{q:\texttt{Parent}(q,x)}[F(q)] \Rightarrow F(x)$.
From axiom ISU7 follows: $\forall_{x\in\mathscr{O}}[F(x)]$.

**axiom** $\text{ISU8}_R$. From axiom $\text{ISU8}_R$ and observation $x \rightsquigarrow_R y \Rightarrow x \rightsquigarrow y$, the preservation by $\rightsquigarrow_R$ directly follows. For the reflection by $\rightsquigarrow_R$, suppose $x \sim y \wedge \neg\texttt{Root}_R(y)$. Then $a \rightsquigarrow_R y$ for some $a$. From axiom $\text{ISU6}_R$ we conclude $a \underline{\rightsquigarrow}_R p \wedge \texttt{Parent}(p,y)$ for some $p$. From axiom F2 and Lemma 3.4, we conclude $R(q,y)$ for all parents $q$ of $y$. As $P_x$ is reflected by Parent (see Corollary 3.6), we have a parent $q$ such that $x \sim q$ and $\texttt{Parent}_R(q,y)$. So, $x \sim q \wedge q \rightsquigarrow_R y$. □

As a direct result of the above theorem, all properties of $\rightsquigarrow$ will, a forteriori, hold for $\rightsquigarrow_R$. Next we focus on the relation between inheritance properties in an identification hierarchy, and a filtered version of this hierarchy.

**Lemma 3.5.** *If $P$ is preserved by $\rightsquigarrow$, and $R$ a filter relation on $\rightsquigarrow$, then $P$ is also preserved by $\rightsquigarrow_R$.*

**Proof.** Suppose $P$ is preserved by $\rightsquigarrow$, and $R$ a filter relation on $\rightsquigarrow$. Suppose $P(x) \wedge x \rightsquigarrow_R y$, then obviously $P(x) \wedge x \rightsquigarrow y$, and thus $P(y)$. Thus, $P$ is also preserved by $\rightsquigarrow_R$. □

If a relation $P$ is reflected by $\rightsquigarrow$, then $P$ does not have to be reflected by $\rightsquigarrow_R$. If $P(x)$, and $x$ is not an R-root, then there exists an ancestor $a$ such that $P(a)$, as $P$ is reflected by $\rightsquigarrow$. Furthermore, there exists an R ancestor $a'$. However, we are not able to prove that $a'$ has property $P$. This is illustrated in Fig. 13. Nevertheless, if $P$ is also preserved by $\rightsquigarrow$, we can prove the following:

**Lemma 3.6.** *If $P$ is filled by $\rightsquigarrow$, and $R$ a filter relation on $\rightsquigarrow$, then $P$ is filled by $\rightsquigarrow_R$.*

**Proof.** Suppose $P$ is filled by $\rightsquigarrow$, and $R$ a filter relation on $\rightsquigarrow$. From the previous lemma we know that $P$ is preserved by $\rightsquigarrow_R$.

Suppose $P(y) \wedge \neg \mathtt{Root}_R(y)$. As $y$ is not an $R$-root, for some $z'$ we have $z' \rightsquigarrow_R y$. Using axiom ISU6 we find a parent $p'$ between $z'$ and $y$, such that $R(p', y)$. Thus, by axiom F2, for all parents $p$ of $y$ we have $R(p, y)$.

As $P$ is a reflected by $\rightsquigarrow$, some ancestor $a$ of $y$ has the property $P(a) \wedge a \rightsquigarrow y$. Furthermore, as $P$ is preserved by $\rightsquigarrow$, and due to axiom ISU6, there is a $p$ such that $P(p) \wedge \mathtt{Parent}(p, y)$. Since $R(p, y)$ for any ancestor of $y$, we have: $P(p) \wedge p \rightsquigarrow_R y$. Thus, $P$ is also filled by $\rightsquigarrow_R$. This is illustrated by Fig. 13. □

The following property states that the existence of $R$-roots is bounded by the original `RootOf` relation:

**Lemma 3.7.** $r\ \mathtt{RootOf}\ x \Leftrightarrow \exists_s [r\ \mathtt{RootOf}\ s \wedge s\ \mathtt{RootOf}_R\ x]$.

**Proof.**

$\Rightarrow$ $Q_r$ is preserved by $\rightsquigarrow$, and thus also preserved by $\rightsquigarrow_R$. Applying the foundation schema (Lemma 3.1) yields the result.

$\Leftarrow$ If $r\ \mathtt{RootOf}\ s$ and $s\ \mathtt{RootOf}_R\ x$, we have $\mathtt{Root}(r)$ and $r \rightsquigarrow s \rightsquigarrow x$, and thus $r\ \mathtt{RootOf}\ x$. □

We will call filter relations $R_1, \ldots, R_n$ over $\mathcal{O} \times \mathcal{O}$ an *identification signature* for the identification hierarchy $\rightsquigarrow$, if they span $\rightsquigarrow$:

$$\mathtt{Parent}(x, y) \Rightarrow \exists!_i [R_i(x, y)] .$$

A direct result of this definition is:

**Lemma 3.8.** *If $R_1, \ldots, R_n$ are an identification signature of $\rightsquigarrow$, then $\mathtt{Parent}_{R_1}, \ldots \mathtt{Parent}_{R_n}$ forms a partition of* `Parent`.

**Proof.** From the definition of an identification structure immediately follows that if $i \neq j$, then

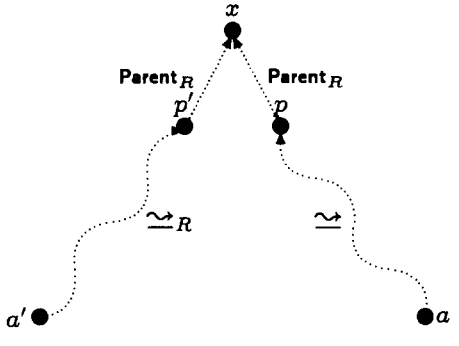



Fig. 13. R-perfect inheritance.

$\texttt{Parent}_{R_i}$ and $\texttt{Parent}_{R_j}$ are disjunct. Furthermore, we obviously have $\texttt{Parent}_{R_i} \subseteq \texttt{Parent}$, thus:

$$\bigcup_{1 \leq i \leq n} \texttt{Parent}_{R_i} \subseteq \texttt{Parent}\,.$$

Conversely, from the definition of an identification structure follows:

$$\texttt{Parent} \subseteq \bigcup_{1 \leq i \leq n} \texttt{Parent}_{R_i}\,. \quad \square$$

The notion of identification signature will be useful when partitioning the identification hierarchy of PSM into generalization and specialization (see Subsection 7.1).

### *3.4. The remaining components*

Besides the information structure, an application model contains a number of other elements. The hierarchy of models in Fig. 3 describes how an application model is constructed from other (sub)models. However, this hierarchy disregards relations that must hold between the submodels, for example, how a population relates to the information structure. These relations are the crucial elements of an application model.

An application model version provides a complete description of the state of the information system at some point of time. Such an application model version is bound to the *application model universe* $\mathcal{U}_{\Sigma}$.

**Definition 3.2.** *An application model universe is spanned by the tuple*:

$$\langle \mathcal{U}_{\mathcal{I}}, \mathcal{D}, \Omega, \texttt{IsPop}, \gamma, \mu, [\![\,]\!], \texttt{Depends} \rangle$$

where the information structure universe $\mathcal{U}_{\mathcal{I}}$ has been introduced in the previous section. $\mathcal{D}$ is a set of underlying concrete domains to be associated to label types. The set $\Omega$ is derived from these concrete values, and is a domain for instantiating abstract object types. The predicate IsPop checks if such an instantiation is wellformed. $\gamma$ and $\mu$ are the universes for constraint and method definitions respectively. The semantics of both constraints and methods is provided by the quaternary predicate $[\![\,]\!]$ (see Subsection 2.4). The dependencies of constraints and methods on the type level ($\mathcal{C}, \mathcal{L} \times \mathcal{D}$) are described by the relation Depends.

The information structure universe $\mathcal{U}_{\mathcal{I}}$ was introduced in the previous subsection. The other components of the application model universe are discussed in the remainder of this subsection.

#### *3.4.1 Domains*

The separation between concrete and abstract world is provided by the distinction between the information structure $\mathcal{I}$, and the set of underlying (concrete) domains in $\mathcal{D}$ [25]. An application model version at point of time $t$, therefore, contains a mapping $\texttt{Dom}_t : \mathcal{L} \rightarrow \mathcal{D}$ providing the relation between label types and domains in that version. The domain of these domain assignments is defined as: $\texttt{Dom} = \mathcal{L} \times \mathcal{D}$, so $\texttt{Dom}_t \subseteq \texttt{Dom}$. Some illustrative examples of such domain assignments, in the context of the car insurance running example, are:

Km↦Natno, Name↦String, where Natno and String are assumed to be (names of) concrete domains.

*3.4.2 Instances*

The population of an information structure is not, as usual, a partial function that maps object types to sets of instances. Rather, an instance is considered to be an independent application model element, which evolves by itself. Therefore an instance version is an association between a value and a (non-empty) set of object types, specifying the object types having this value in their population in that version. This association provides the intuition behind the relation $\texttt{HasTypes}_t$, where $t$ is the current point in time.

The expression $x\ \texttt{HasTypes}_t\ T$ states that the value $x \in \Omega$ has associated all types from $T$ (where $T \subseteq \mathcal{O}$, and $T \neq \emptyset$). The set $\Omega$ is the domain for values in instantiations. The domain for the relation $\texttt{HasTypes}_t$ is: $\texttt{HasTypes} = \Omega \times \wp^+(\mathcal{O})$, where $\wp^+$ denotes the powerset operation excluding the empty set.

Note that $\texttt{HasTypes}_t$ is a relation rather than a (partial) function. The reason is to support complex generalisation hierarchies. For example, suppose that $\{a_1, a_2\}$ is an instance of both power types $D$ and $E$ in Fig. 14. (Note that power types are graphically represented as a circle around the corresponding element type.) This can either be modelled by the single instance $\{a_1, a_2\}\ \texttt{HasTypes}_t\ \{D, E, F, G\}$ or by the two instances (with the same value $\{a_1, a_2\}$) $\{a_1, a_2\}\ \texttt{HasTypes}_t\ \{D, F\}$ and $\{a_1, a_2\}\ \texttt{HasTypes}_t\ \{E, G\}$. The difference between these two options comes to the fore, when considering evolution of instances. The second way of modelling allows us to describe the evolution of both instantiations, although they have the same value ($\{a_1, a_2\}$), separately. A concrete example of such a situation, would be when object type $A$ is a set of students, $B$ is the set of students participating in practicum groups, and $C$ is the set of students playing soccer. The object types $E$ and $D$, then correspond to the soccer teams, and the practicum groups respectively. Now consider the value $\{a_1, \ldots, a_{14}\}$. This value may well be a soccer team, and a practicum group at the same time! In this case, it is obvious that one wants to model the evolution of both instantiations separately, i.e. two instances with the same value, and differing sets of associated object types.

The population of an object type in a version, traditionally provided as a function $\mathcal{O} \rightarrow \wp(\Omega)$, can be derived from the association between instances and object types:

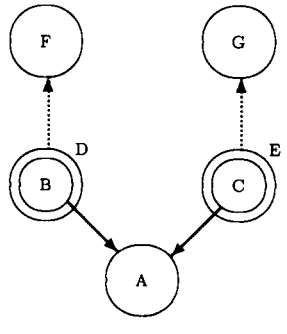



Fig. 14. Complex generalisation hierarchy.

$$\mathtt{Pop}_t(x) = \{v \mid \exists_Y[v\ \mathtt{HasTypes}_t\ Y \wedge x \in Y]\}\ .$$

This does not necessarily lead to a proper population of the information structure version at point in time $t$. A population of an information structure will have to adhere to some technique dependent properties. These properties are assumed to be provided by the predicate $\mathtt{IsPop} \subseteq \wp(\mathcal{O}) \times \wp(\mathtt{HasTypes})$. Note that a set of object types $O$ completely determines an information structure. The intuition behind the expression $\mathtt{IsPop}(O, H)$ is thus: $H$ is a proper instantiation of the information structure $O$. The definition of this predicate will be given in Section 7.

*3.4.3 Constraints*

Most data modelling techniques offer a language for expressing constraints, both state and transition oriented for a given $\mathcal{U}_g$. This language describes a set $\gamma$ of all possible constraint definitions.

Constraints are treated as application model elements, that assign constraint definitions to (some) object types. A constraint $c$ is said to be *owned* by an object type $x$, if $x$ has assigned a constraint definition by constraint $c$.

Constraints are inherited via the identification hierarchy. However, as in object oriented data modelling techniques, overriding constraint definitions in identification hierarchies is possible (see for instance [13]).

A constraint $c$, in an application model version, will be a (usually very sparse) partial function $c: \mathcal{O} \rightarrowtail \gamma$, providing for every object type a *private* definition of the constraint. Each modelling technique will have its own possibilities to formulate inheritance rules, thus governing the mapping $c$. The domain for constraints is defined as: $\mathcal{R} = \mathcal{O} \rightarrowtail \gamma$. Enforcing constraints on a population is discussed in the next section.

*3.4.4 Methods*

The action model part of an application model version will be provided as a set of action specifications. The domain for action definitions ($\mu$) is determined by the chosen modelling technique for the action model. As this paper focuses on the evolution of PSM data models, we will not take action modelling techniques into consideration.

The modelling technique dependent, inheritance mechanism for constraints can be used for methods as well. A method $m$ is regarded as a partial function $m: \mathcal{O} \rightarrowtail \mu$, assigning action specifications to object types. The set of all possible methods is the set of all these mappings:

$$\mathcal{M} = \mathcal{O} \rightarrowtail \mu$$

This definition provides the formal foundation of the methods in the preliminary definition of the living space of an evolving information system as provided in Subsection 2.4.

The semantics of methods and constraints are defined by the relation $[\![\ ]\!]$ and are treated similarly, but as stated at the end of Subsection 2.4 the semantics are considered to be outside the scope of this paper.

*3.4.5 Syntactic Conformity*

Methods (and constraints) are usually defined by some syntactic mechanism (language). For example, for Fig. 4 the specification language LISA-D could be used to express non-graphical

constraints. The graphical constraints used in NIAM or ER schemes form another example of the use of a (graphical!) syntactic mechanism.

Every method and constraint will refer to (uses) a number of object types and denotable instances (i.e. directly representable on a communication medium). This relation is provided in the application model universe by means of the dependency relation `Depends`:

$$\texttt{Depends} \subseteq (\mu \cup \gamma) \times (\mathcal{O} \cup \mathcal{L} \times \mathcal{D}) .$$

This relation is modelling technique dependent, but is not subject to evolution.

The interpretation of this relation is as follows: $x$ `Depends` $y$ means that if $y$ is not alive in an application model version, then $x$ has no meaning in that version. A consequence is that, in case of evolution of application models, when $y$ evolves to $y'$, then $x$ must be adapted appropriately.

As an example, consider the following constraint for the car insurance example:

`NEVER Client having Policy with Amount > 10000`

stating that no price of a policy should ever exceed 10000 Ecu. This action specification depends on object types `Client`, `Policy` and `Amount`. It, furthermore, depends on the domain assignment: `Amount` $\mapsto$ `Natno`. If one of the object types, or the domain assignment, is terminated or changed, the action specification has to be terminated or changed accordingly.

## 4. Application Model Versions

The (description of the) evolution of an application domain (i.e., an application model history) has been introduced as a set of application model element evolutions. Therefore, an application model version can be determined by the application model element versions at that point in time. At this moment we will identify the domain for such versions:

**Definition 4.1.** *An application model version at point in time t over an application model universe is determined by:*

$$\Sigma_t = \langle \mathcal{O}_t, \mathcal{R}_t, \mathcal{M}_t, \texttt{HasTypes}_t, \texttt{Dom}_t \rangle$$

*where* $\mathcal{O}_t \subseteq \mathcal{O}$, $\mathcal{R}_t \subseteq \mathcal{R}$, $\mathcal{M}_t \subseteq \mathcal{M}$, $\texttt{HasTypes}_t \subseteq \texttt{HasTypes}$ *and* $\texttt{Dom}_t \subseteq \texttt{Dom}$.

From a version of an application model at a given point in time $t$, we can derive the current version $\mathcal{I}_t = \langle \mathcal{L}_t, \mathcal{N}_t, \sim_t, \rightsquigarrow_t \rangle$ of the information structure as follows:

$$\mathcal{L}_t = \mathcal{O}_t \cap \mathcal{L}$$

$$\mathcal{N}_t = \mathcal{O}_t \cap \mathcal{N}$$

$$x \sim_t y \stackrel{\Delta}{=} x \sim y \wedge x, y \in \mathcal{O}_t$$

$$x \rightsquigarrow_t y \stackrel{\Delta}{=} x \sim y \wedge x, y \in \mathcal{O}_t .$$

Note that `IsSch` can be used to determine whether information structure version $\mathcal{O}_t$ is valid:

$\texttt{IsSch}(\mathcal{O}_t)$. Every application model version must adhere to certain rules of well-formedness. Some of these rules are modelling technique dependent, and therefore outside the scope of this paper. Nonetheless, some general rules about application model versions will be stated.

### 4.1. Active and living objects

An object type evolution $h: T \rightarrowtail \mathcal{O}$ is called *alive* at a certain point of time $t$, if it is part of the application model version at that point of time ($h(t) \in \mathcal{O}_t$). Furthermore, an object type $h(t)$ is termed *active* at $t$, if there is an *instance typing* $X$ at $t$ such that $h(t) \in X$. We call $X$ an instance typing if $\exists_v[v\ \texttt{HasTypes}_t\ X]$.

A first rule of well-formedness states that every active object type must be alive as well. This rule can be popularised as: 'I am active, therefore I am alive', or: 'ghost object types can not exist'. This is formalised as:

**[AMV1]** (*active life*). If $X$ is an instance typing at $t$, then:

$$X \subseteq \mathcal{O}_t .$$

The next rule of well-formedness states that sharing an instance at any point of time, is to be interpreted as a proof of type relatedness:

**[AMV2]** (*active relatedness*). If $X$ is an instance typing, then:

$$x, y \in X \Rightarrow x \sim y .$$

From the very nature of the root relation it follows that instances are included upwards, towards the roots. As a result, every instance of an object type should also be an instance of its ancestors (if any):

**[AMV3]** (*foundation of activity*). If $X$ is an instance typing, then the property $A_X$ defined by $A_X(x) \stackrel{\Delta}{=} x \in X$ is reflected by $\rightsquigarrow$.

Applying the foundation schema (Lemma 3.1) to this axiom shows the presence of roots in instance typings:

**Lemma 4.1** (*active roots*). *If $X$ is an instance typing, then:*

$$y \in X \Rightarrow \exists_x[x \in X \wedge x\ \texttt{RootOf}\ y] .$$

In most traditional data modelling techniques (such as ER, NIAM and FORM), each type hierarchy has a unique root. As a consequence, each instance typing contains a unique root. Some data modelling techniques (such as PSM), however, allow type hierarchies with multiple roots (see Fig. 14). For such modelling techniques, the following axiom guarantees a unique root for each instance typing.

**[AMV4]** (*unique root*). If $X$ is an instance typing and $x, y \in X$ then:

$$\texttt{Root}(x) \wedge \texttt{Root}(y) \Rightarrow x = y \ .$$

Axiom AMV3 has a structural pendant as well: every living object type is accompanied by one of its ancestors (if any). This is stipulated in the following axiom:

**[AMV5]** (*foundation of live*). For all points of time $t$, the property $L_t$ defined by $L_t(x) \stackrel{\triangle}{=} x \in \mathcal{O}_t$ is reflected by $\rightsquigarrow$.

Note that axiom AMV5 can not be derived from axiom AMV3. The reason is that a non-root object type may be alive, yet have no instance associated. By applying Lemma 3.1, we also have:

**Corollary 4.1** (*living roots*). *The property $L_t$ is reflected by* `RootOf`.

### 4.2. Well-formed concretisation

In a valid application model version each label type is *concretised* by associating a domain. Therefore, the domain providing function $\texttt{Dom}_t$ is a (total) function from alive label types to domains:

**[AMV6]** (*full concretisation*). $\texttt{Dom}_t : \mathcal{L}_t \rightarrow \mathcal{D}$.

Domain assignment for label types, should not be conflicting with inheritance relations:

**[AMV7]** (*domain inclusion*). If $l_1, l_2 \in \mathcal{L}$, then:

$$l_1 \rightsquigarrow l_2 \Rightarrow \texttt{Dom}_t(l_2) \subseteq \texttt{Dom}_t(l_1) \ .$$

Furthermore, the instances of label types must adhere to this domain assignation:

**[AMV8]** (*strong typing of labels*). If $v \ \texttt{HasTypes}_t \ X$ and $v \in \bigcup \mathcal{D}$ then:

$$x \in X \Rightarrow v \in \texttt{Dom}_t(x) \ .$$

### 4.3. Constraints and methods

Methods, and thus constraints, are defined as mappings from object types to method and constraint definitions respectively. This implies that object types, owning a constraint or a method, must be alive.

**[AMV9]** (*alive definitions*). If $w \in \mathcal{R}_t \cup \mathcal{M}_t$ then:

$$\texttt{dom}(w) \subseteq \mathcal{O}_t \ ,$$

where $\texttt{dom}(w) = \{x \mid \langle x, y \rangle \in w\}$ is the domain of function $w$. Furthermore, object types that own the same constraint or method, are type related.

**[AMV10]** (*type related definitions*). If $w \in \mathcal{R}_t \cup \mathcal{M}_t$ then:

$$x, y \in \mathtt{dom}(w) \Rightarrow x \sim y .$$

Finally, due to inheritance, if a constraint is defined for an ancestor object type, it is defined for its offspring as well.

**[AMV11]** (*inheritance of definitions*). For all $w \in \mathcal{R}_t \cup \mathcal{M}_t$, the property $D_w$ defined by $D_w(x) \stackrel{\Delta}{=} x \in \mathtt{dom}(w)$ is preserved by $\rightsquigarrow$.

Note that the inheritance direction for populations is reverse to the inheritance direction for methods (and constraints).

The motivation for the following axiom lies in the following observation. The definition of a constraint or a method refers to a set of object types, and domain concretisations. Thus, if a method or constraint definition is alive, then all these referred items should be alive at that same moment.

**[AMV12]** (*dangling references*). If $w \in \mathcal{R}_t \cup \mathcal{M}_t$ then:

$$w(x) \ \mathtt{Depends}\ y \Rightarrow y \in \mathcal{O}_t \cup (\mathcal{L}_t \times \mathcal{D}_t).$$

### 4.4. Populations of information structures

A special part of an application model version is its population. As stated in the previous subsection, this population can be derived from the relation $\mathtt{HasTypes}_t$. A direct consequence of this definition and axiom AMV8 is:

**Corollary 4.2.** $x \in \mathcal{L}_t \Rightarrow \mathtt{Pop}_t(x) \subseteq \mathtt{Dom}_t(x)$.

It will be convenient to have an overview of all instances that ever lived. We will refer to this population as the extra-temporal population.

**Definition 4.2.** The extra-temporal population of an application model is a mapping $\mathtt{Pop}_\infty$: $\mathcal{O} \rightarrow \wp(\Omega)$, defined by

$$\mathtt{Pop}_\infty(x) = \bigcup_{t \in T} \mathtt{Pop}_t(x) .$$

Next we focus at strong typing, which is considered to be a property to hold on each moment: if $x \not\sim y$, then their populations may never share instances. The following axiom is sufficient to guarantee this property, as we will show in Theorem 4.2.

**[AMV13]** (*exclusive root population*). If $\mathtt{Root}(x)$ and $\mathtt{Root}(y)$ then:

$$x \not\sim y \Rightarrow \mathtt{Pop}_\infty(x) \cap \mathtt{Pop}_\infty(y) = \emptyset .$$

If roots are not type related, then their extra-temporal populations are disjoint.

For (extra-temporal) populations, some interesting properties hold. The proofs can be found in [40].

**Lemma 4.2.**

$$\mathtt{Pop}_t(x) \subseteq \bigcup_{y:y\ \mathtt{RootOf}\ x} \mathtt{Pop}_t(y)\,.$$

**Proof.** Let $i \in \mathtt{Pop}_t(x)$, then from the definition of $\mathtt{Pop}_t$ follows the existence of an instance typing $X$ such that: $i\ \mathtt{HasTypes}_t\ X \wedge x \in X$.

Due to Lemma 4.1, we then also have a root $y$ of $x$ such that $y \in X$.

As a result we have: $i\ \mathtt{HasTypes}_t\ X \wedge y \in X$. From the definition of $\mathtt{Pop}_t$ then follows that $i \in \mathtt{Pop}_t(y)$. □

By means of the following theorem the nature of type relatedness, captured for roots in the above axiom, is generalised to object types in general:

**Theorem 4.1** (*exclusive population*). *If* $x \not\sim y$ *then*

$$\bigcup_{z:z\ \mathtt{RootOf}\ x} \mathtt{Pop}_\infty(z) \cap \bigcup_{z:z\ \mathtt{RootOf}\ y} \mathtt{Pop}_\infty(z) = \emptyset\,.$$

*The populations of object types which are not type related, have no values in common.*

From Lemma 4.2 and Theorem 4.1 the main typing theorem is derived:

**Theorem 4.2** (*strong typing theorem*).

$$x \not\sim y \Rightarrow \mathtt{Pop}_\infty(x) \cap \mathtt{Pop}_\infty(y) = \emptyset$$

We are now in a position to define what constitutes a good application model version ($\Sigma_t = \langle \mathcal{O}_t, \mathcal{R}_t, \mathcal{M}_t, \mathtt{HasTypes}_t, \mathtt{Dom}_t \rangle$):

**Definition 4.3.**

$$\mathtt{IsAM}(\Sigma_t) \triangleq \mathtt{IsSch}(\mathcal{O}_t) \wedge \mathtt{IsPop}(\mathcal{O}_t, \mathtt{HasTypes}_t) \wedge \Sigma_t \text{ adheres to the AMV axioms}\,.$$

In the next section, this predicate will be used to define what a proper application model history (IsAMH) is.

## 5. Evolution of application models

The evolution of an application model is described by the evolution of its elements. The set $\mathcal{AME}$ has been introduced as the set of all evolvable elements of an application model. Its formal definition, in terms of components of $\mathcal{U}_\Sigma$, is:

**Definition 5.1.** *Application model elements:*

$$\mathcal{AME} = \mathcal{O} \cup \mathcal{R} \cup \mathcal{M} \cup \texttt{HasTypes} \cup \texttt{Dom}.$$

An application model history has been introduced as a set of application model element evolutions (see Subsection 2.4). In this section we discuss some wellformedness rules for application model histories. For a more elaborate discussion on such rules, see [40]. For the remainder of this section, let $H$ be some (fixed) application model history.

### 5.1. Separation of element evolution

The first rule of wellformedness states that the evolution of application model elements is bound to classes. For example, an object type may not evolve into a method, and a constraint may not evolve into an instance. This leads to the following axiom:

**[EW1]** (*evolution separation*). If $A \in \{\mathcal{O}, \mathcal{R}, \mathcal{M}, \texttt{HasTypes}, \texttt{Dom}\}$, and $h \in H$ then:

$$h(t) \in A \Rightarrow \texttt{ran}(h) \subseteq A,$$

where $\texttt{ran}(h) = \{y \mid \langle x, y \rangle \in h\}$.

As a result, an application model history can be partitioned into the history of its object types $H_{type}$, its constraints $H_{constr}$, its methods $H_{meth}$, its populations $H_{pop}$, and its concretizations (of label types) $H_{dom}$. An application model version at a given point of time $t$, is easily derived from an application model history $H$. This is done by defining the five main components, which determine an application version:

**Definition 5.2.**
*object types:* $\mathcal{O}_t = \{h(t) \mid h \in H_{type} \wedge h{\downarrow}t\}$,
*constraints:* $\mathcal{R}_t = \{c(t) \mid c \in H_{constr} \wedge c{\downarrow}t\}$,
*methods:* $\mathcal{M}_t = \{m(t) \mid m \in H_{meth} \wedge m{\downarrow}t\}$,
*population:* $\texttt{HasTypes}_t = \{g(t) \mid H_{pop} \wedge g{\downarrow}t\}$,
*concretisations:* $\texttt{Dom}_t = \{d(t) \mid d \in H_{dom} \wedge d{\downarrow}t\}$.

In this definition, $f{\downarrow}t$ is used as an abbreviation for $\exists_s[\langle t, s \rangle \in f]$, stating that (partial) function $f$ is defined at time $t$.

### 5.2. Enforcing constraints

The next rule of well-formedness on the evolution of an application model $\Sigma$ states that for every population of an application model version, all constraints in that version must hold. However, constraints do not cause restrictions beyond their lifetime. The intuition behind is the closed world assumption, applied to an application model history, with respect to changes outside the lifetime of the constraint.

**[EW2]** (*constraints hold*). For all $c \in H_{constr}$:

$$c{\downarrow}t \Rightarrow H_{\mathtt{birth}(c,t)|t}[\![c(t)]\!]_t ,$$

where the birth of constraint $c$ is found by $\mathtt{birth}(c, t) = \min\{b \mid \forall_{b \leq u \leq t}[c(u) = c(t)]\}$. Note that the semantics of both methods and constraints have been introduced by $[\![\ ]\!]$ (see Subsection 2.4). The restriction of an application model history to a period of time is defined by $H_{b|t} = \{h_{b|t} \mid h \in H\}$, where:

$$\begin{aligned} h_{b|t} = \lambda u.&\ \mathtt{if}\ b \leq u \leq t\ \mathtt{then}\ h(u) \\ &\ \mathtt{else\ if}\ u < b\ \mathtt{then}\ h(b) \\ &\ \mathtt{else}\ h(t) \\ &\ \mathtt{fi\ fi}\ . \end{aligned}$$

Finally, we can live up to our promise of defining IsAMH formally:

**Definition 5.3.**

$$\mathtt{IsAMH}(H) \triangleq \forall_{t \in T}[\mathtt{IsAM}(\Sigma_t)] \wedge H \text{ adheres to the EW axioms}.$$

## 6. The EVORM information structure universe

In this section we introduce the information structure universe for the modelling technique EVORM. This technique is based on the modelling technique PSM, and results after applying the evolution theory from the previous sections. As a result, each information structure version constitutes a proper PSM schema.

### *6.1. Information structure universe*

In the EVORM information structure universe, the following components can be identified:

(1) A finite set $\mathscr{P}$ of *predicators*. Predicators correspond to roles in NIAM and (E)ER.
(2) A non-empty finite set $\mathscr{O}$ of *object types*.
(3) A set of *label types* $\mathscr{L}$. As every label type is an object type we have: $\mathscr{L} \subseteq \mathscr{O}$. Label types correspond to value types in (E)ER.
(4) A partition $\mathscr{F}$ of the set $\mathscr{P}$. The elements of $\mathscr{F}$ are called *fact types*. All fact types are object types, so: $\mathscr{F} \subseteq \mathscr{O}$. Fact types correspond to relationship types or attribute types in (E)ER.
(5) A set $\mathscr{G}$ of *power types*. Every power type is an object type, hence $\mathscr{G} \subseteq \mathscr{O}$.
(6) A set $\mathscr{S}$ of *sequence types*. Each sequence type is an object type, therefore $\mathscr{S} \subseteq \mathscr{O}$.
(7) A set $\mathscr{C}$ of *schema types*. Any schema type is an object type as well, so: $\mathscr{C} \subseteq \mathscr{O}$.
(8) A function Base: $\mathscr{P} \rightarrow \mathscr{O}$. The base of a predicator is the object type part of that predicator.
(9) A function Elt: $\mathscr{G} \cup \mathscr{S} \rightarrow \mathscr{O}$. This function yields the *element type* of a power type or sequence type.

(10) A partial order $\texttt{IdfBy} \subseteq \mathcal{O} \times \mathcal{O}$ on object types, capturing the inheritance hierarchy.
(11) A partial order $\texttt{Spec} \subseteq \texttt{IdfBy}$ specifying that part of the identification hierarchy which is concerned with specialisation.
(12) A partial order $\texttt{Gen} \subseteq \texttt{IdfBy}$ specifying that part of the identification hierarchy which is concerned with generalisation.
(13) A relation $\prec \subseteq \mathcal{C} \times \mathcal{O}$, describing the decomposition of schema types.
(14) Not every set of object types will lead to a correct schema. Therefore, the relation (set) $\texttt{IsSch} \subseteq \mathcal{O}$ will designate which schemas are correct. This predicate will be defined formally, together with the IsPop predicate, in the next section providing the wellformedness rules for EVORM (see Fig. 2).

For fact types we define the auxiliary function $\texttt{Fact}: \mathcal{P} \rightarrow \mathcal{F}$, yielding the fact type in which a given predicator is contained. This fact type is identified by: $\texttt{Fact}(p) = f \Leftrightarrow p \in f$. In the remainder of this section, we shortly formulate rules for this information structure universe in terms of EU axioms. For a more complete discussion of these axioms, see [26] and [22]. After that, the predicate IsSch is introduced by EU-axioms.

Label types, entity types, fact types, sequence types and schema types will all be interpreted differently:

**[EU1].** $\mathcal{L}$, $\mathcal{E}$, $\mathcal{F}$, $\mathcal{G}$, $\mathcal{S}$, $\mathcal{C}$ form a partition of $\mathcal{O}$.

### *6.2. Abstract and concrete objects*

Bridge types establish the connection between abstract and concrete object types. In (E)ER they are referred to as attribute types. The term $\texttt{Bridge}(f)$ qualifies fact type $f$ as a bridge type, and is an abbreviation for the expression

$$\exists_{p,q}[f = \{p, q\} \wedge \texttt{Base}(p) \in \mathcal{L} \wedge \texttt{Base}(q) \notin \mathcal{L}] .$$

The set of all bridge types in the information structure universe is denoted by $\mathcal{B}$. The strict separation between the concrete and abstract level is expressed by the following rules. Firstly, label types may only participate in bridge types:

**[EU2].** $\texttt{Base}(p) \in \mathcal{L} \Rightarrow \texttt{Bridge}(\texttt{Fact}(p))$.

Secondly, bridge types may not be used to build other object types. Consequently, they cannot be used for objectification:

**[EU3].** $\neg\texttt{Bridge}(\texttt{Base}(p))$.

The predicators that constitute a bridge type $b = \{p, q\}$ can be extracted by the operators concr and abstr. These operators are defined by $\texttt{concr}(b) \in b \wedge \texttt{Base}(\texttt{concr}(b)) \in \mathcal{L}$ and $\texttt{abstr}(b) \in b \wedge \texttt{Base}(\texttt{concr}(b)) \notin \mathcal{L}$ respectively.

### 6.3. Power typing

The element type of a power type is found by the function Elt. The relation between a power type $x$ and its element type $\texttt{Elt}(x)$ is recorded in the fact type $\in_{x,\texttt{Elt}(x)}$. In general an implicit fact type $\in_{x,y}$ will be used as a bridge between complex object types $x$ (power types, sequence types and schema types), and their elementary types $y$. This fact type is defined by:

$$\in_{x,y} = \{\in^{c}_{x,y}, \in^{e}_{x,y}\},$$

where $\texttt{Base}(\in^{c}_{x,y}) = x$ and $\texttt{Base}(\in^{e}_{x,y}) = y$. With respect to power types, this relation is assumed to be available for each power type.

Usually, implicit fact types are not drawn in an information structure diagram. Only if such a fact type is subject to constraints, or used in an objectification, it needs to be made explicit. Note that, in this way, power typing corresponds to a polymorphic type constructor, and the fact type $\in_{x,\texttt{Elt}(x)}$ to an associated polymorphic access operator. The strict separation between abstract and concrete object types prohibits label types to occur as element type:

**[EU4].** $\texttt{Elt}(x) \notin \mathscr{L}$.

### 6.4. Sequence typing

The element type of a sequence type is also found by the function Elt. The relation between a sequence type $x$ and its element type $\texttt{Elt}(x)$ is recorded by the implicit fact type $\in_{x,\texttt{Elt}(x)}$. Contrary to power types, this relation $\in_{x,\texttt{Elt}(x)}$ is augmented with the position of the element in the sequence, via the implicit fact type $@_{x,\texttt{Elt}(x)} = \{@^{s}_{x,\texttt{Elt}(x)}, @^{i}_{x,\texttt{Elt}(x)}\}$, where $\texttt{Base}(@^{s}_{x,\texttt{Elt}(x)}) = \in_{x,\texttt{Elt}(x)}$ and $\texttt{Base}(@^{i}_{x,\texttt{Elt}(x)}) = \mathrm{I}$. The object type I is the domain for indexes in sequence types. Usually the natural numbers are used for this purpose. The index type is assumed to be a label type ($\mathrm{I} \in \mathscr{L}$), which is assumed to be totally ordered and to have a least element. Note that axiom EU4 also applies for sequence types.

### 6.5. Schema types

Schema types can be decomposed into an underlying information structure via the relation $\prec$, with the convention that $x \prec y$ is interpreted as $x$ is decomposed into $y$ or $y$ is part of the decomposition of $x$.

This underlying information structure $\mathscr{I}^{x}$ for a schema type $x$ is derived from the object types into which $x$ is decomposed: $\mathscr{O}^{x} = \{y \in \mathscr{O} \mid x \prec y\}$. Analogously the special object classes $\mathscr{F}^{x}$, $\mathscr{G}^{x}$, $\mathscr{S}^{x}$, $\mathscr{C}^{x}$ and $\mathscr{E}^{x}$ can be derived.

With each schema type $x$ and each object type $y$ in its decomposition, the implicit fact type $\in_{x,y}$ is associated. These fact types enable the transition from a composed object to an object from its decomposition.

### 6.6. Identification hierarchy

The identification hierarchy in EVORM is defined as the partial order (asymmetric and transitive) IdfBy on object types, with the convention that $a$ IdfBy $b$ is interpreted as: $a$

*inherits its identification from b*. As non-root object types inherit the structure of their parents, they are atomic, i.e. entity or label types. However, abstract and concrete object types should not be mixed up:

**[EU5]** (*strictness*). $\texttt{IdfBy} \subseteq \mathscr{E} \times (\mathscr{O} \setminus \mathscr{L}) \cup \mathscr{L} \times \mathscr{L}$.

The nature of a partial order is expressed by:

**[EU6]** (*asymmetry*). $x\ \texttt{IdfBy}\ y \Rightarrow \neg y\ \texttt{IdfBy}\ x$.

**[EU7]** (*transitivity*). $x\ \texttt{IdfBy}\ y\ \texttt{IdfBy}\ z \Rightarrow x\ \texttt{IdfBy}\ z$.

We define $\texttt{IdfBy}_1$ as the one step counterpart of IdfBy:

$$x\ \texttt{IdfBy}_1\ y \triangleq x\ \texttt{IdfBy}\ y \wedge \neg\exists_z[x\ \texttt{IdfBy}\ z\ \texttt{IdfBy}\ y]\,.$$

In EVORM, all object types in the identification hierarchy have direct ancestors:

**[EU8]** (*direct ancestors*). $x\ \texttt{IdfBy}\ y \Rightarrow x\ \texttt{IdfBy}_1\ y \vee \exists_p[x\ \texttt{IdfBy}_1\ p\ \texttt{IdfBy}\ y]$.

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

**[EU9]** (*identification induction*). If $F$ is a property for object types, such that for all $y$: $\forall_{x:y\ \texttt{IdfBy}_1\ x}[F(x)] \Rightarrow F(y)$, then $\forall_{x \in \mathscr{O}}[F(x)]$.

The identification hierarchy is a result of specialisation and generalisation:

**[EU10]** (*complete span*).
(1) $x\ \texttt{Spec}\ y \vee x\ \texttt{Gen}\ y \Rightarrow x\ \texttt{IdfBy}\ y$,
(2) $x\ \texttt{IdfBy}_1\ y \Rightarrow \texttt{Gen}\ y \vee x\ \texttt{Spec}\ y$.
In the next subsections, the relations Spec and Gen will be refined. As a result, Spec and Gen will be filter relations of IdfBy.

### *6.7. Specialisation*

The concept of specialisation is modelled as a partial order (asymmetric and transitive) Spec on object types. The intuition behind $a\ \texttt{Spec}\ b$ is: $a$ is a specialisation of $b$, or $a$ is a subtype of $b$.

**[EU11]** (*transitivity completeness*). If $x\ \texttt{IdfBy}\ y\ \texttt{IdfBy}\ z$ then:

$$x\ \texttt{Spec}\ y\ \texttt{Spec}\ z \Leftrightarrow x\ \texttt{Spec}\ z\,.$$

Note that the asymmetry of Spec follows from the asymmetry of IdfBy, as $\texttt{Spec} \subseteq \texttt{IdfBy}$.

On specialisation hierarchies, we define the *pater familias* relation. This relation represents the root relation, if we restrict ourselves to specialisation based inheritance. The pater familias relation is identified by:

$$\sqcap(x, y) \triangleq (x \texttt{ Spec } y \vee x = y) \wedge \neg \texttt{spec}(y) ,$$

where $\texttt{spec}(x)$ is a shorthand for $\exists_y[x \texttt{ Spec } y]$. Each specialisation hierarchy, contrary to generalisation, has a unique top element. This is stipulated by the following axiom:

**[EU12]** (*unique pater familias*).

$$\sqcap(x, y) \wedge \sqcap(x, z) \Rightarrow y = z .$$

This axiom allows us to regard the pater familias relation $\sqcap$ as a partial function, and write $\sqcap(x) = y$ instead of $\sqcap(x, y)$. In a later subsection we provide a proof for the existence of a pater familias for all object types, thus proving that $\sqcap$ is a total function on $\mathcal{O}$.

### 6.8. Generalisation

The concept of generalisation is introduced as a partial order Gen. The expression $a \texttt{ Gen } b$ stands for: $a$ is a generalisation of $b$, or $b$ is a specifier of $a$. In the sequel $\texttt{gen}(x)$ will be used as an abbreviation for $\exists_{y \in \mathcal{O}}[x \texttt{ Gen } y]$.

**[EU13]** (*transitivity completeness*). If $x \texttt{ IdfBy } y \texttt{ IdfBy } z$ then:

$$x \texttt{ Gen } y \texttt{ Gen } z \Leftrightarrow x \texttt{ Gen } z .$$

Generalisation and specialisation can be conflicting due to their inheritance structure. To avoid such conflicts, generalised object types are required to be pater familias:

**[EU14]**, $\texttt{gen}(x) \Rightarrow \neg \texttt{spec}(x)$.

The different nature of specialisation and generalisation is stipulated in the next lemma, which directly follows from the above axiom.

**Lemma 6.1.** $x \texttt{ Spec } y \Rightarrow \neg x \texttt{ Gen } y$.

*Basic specifiers* of a generalised object type are defined analogously to the pater familias for a specialised object type:

$$\sqcup(x, y) \triangleq (x \texttt{ Gen } y \vee x = y) \wedge \neg \texttt{gen}(y) .$$

Note that uniqueness of basic specifier is not required. As a shorthand, we will write $\sqcup(x)$ for the set $\{y \mid \sqcup(x, y)\}$.

### 6.9. Type relatedness

Intuitively, object types can, for several reasons, have values in common in some instantiation. For example, each value of object type $x$ will, in any instantiation, also be a value of object type $\sqcap(x)$. As another example, suppose $x$ Gen $y$, then any value of $y$ in any population will also be a value of $x$. A third example, where object types may share values is when two power types have element types that may share values.

Formally, for EVORM, type relatedness is captured by a binary relation $\sim$ on $\mathcal{O}$. Two object types are type related if and only if this can be proven from the following derivation rules:

**[TR1].** $x \in \mathcal{O} \vdash x \sim x$.

**[TR2].** $x \sim y \vdash y \sim x$.

**[TR3].** $y\ \mathtt{IdfBy}\ x \wedge x \sim z \vdash y \sim z$.

**[TR4].** $x, y \in \mathcal{G} \wedge \mathtt{Elt}(x) \sim \mathtt{Elt}(y) \vdash x \sim y$.

**[TR5].** $x, y \in \mathcal{S} \wedge \mathtt{Elt}(x) \sim \mathtt{Elt}(y) \vdash x \sim y$.

**[TR6].** $\mathcal{I}_x = \mathcal{I}_y \vdash x \sim y$.

**Example 6.1.** In Fig. 15 the only object types that are type related are $A$ and $B$, $C$ and $D$ and $F$ and $D$.

### 6.10. Valid EVORM versions

Let $\mathcal{O}_t$ be a set of object types spanning an information structure version at point of time $t$. From this version, we derive the EVORM information structure:

$$\mathcal{I}_t = \langle \mathcal{F}_t, \mathcal{G}_t, \mathcal{S}_t, \mathcal{C}_t, \mathcal{E}_t, \mathcal{L}_t, \mathcal{P}_t \rangle .$$

The set of fact types in the EVORM information structure version is defined as $\mathcal{F}_t = \mathcal{F} \cap \mathcal{O}_t$. The other components are derived analogously. The set of predicators, on the other hand, is defined as: $\mathcal{P}_t = \cup \mathcal{F}_t$.

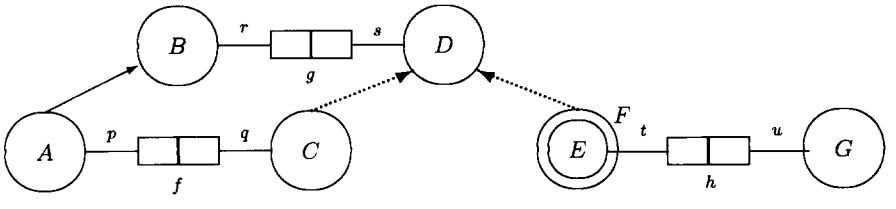



Fig. 15. Example information structure.

For the information structure derived from $\mathcal{O}_t$, we have the following time-conformity rules:

**[EV1]**. $p \in \mathcal{P}_t \Rightarrow \texttt{Base}(p) \in \mathcal{O}_t$.

**[EV2]**. $x \in \mathcal{G}_t \cup \mathcal{S}_t \Rightarrow \texttt{Elt}(x) \in \mathcal{O}_t$.

For schema types, each underlying information structure version should be a proper information structure version on its own:

**[EV3]**. $x \in \mathcal{C}_t \Rightarrow \texttt{IsSch}(\mathcal{O}_t^x)$, where $\mathcal{O}_t^x = \{y \in \mathcal{O}_t \mid x \prec y\}$. The predicate `IsSch` is introduced later in this section. Furthermore, the foundation of live axiom (AMV5) should hold for decomposition as well:

**[EV4]**. If $x \in \mathcal{C}_t$, then the property $DL_t^x$ defined as $DL_t^x(y) \triangleq y \in \mathcal{O}_t^x$ is reflected by `IdfBy`.

The axioms on information structure versions allow us to define what we regard as a good EVORM information structure version:

**Definition 6.1.** $\texttt{IsSch}(\mathcal{O}_t) \triangleq \mathcal{O}_t$ adheres to the EV axioms.

This definition provides the wellformedness predicate for (schema) versions in EVORM (see Fig. 2). In the next section, we will prove that EVORM provides a proper typing mechanism.

## 7. EVORM application model universe

In this section we describe EVORM as an application model universe. We show how EVORM spans an information structure universe, and prove that this universe is a proper information structure universe as defined in Subsection 3.1. The information structure universe of EVORM is more detailed than that of the general theory, as more concepts are recognised. As a result, versions of the EVORM information structure universe have associated (besides the AMV axioms) more rules regarding wellformedness. We distinguish two classes of additional rules. The first class takes wellformedness of the information structure into account, leading to the predicate `IsSch`. The second class poses restrictions on populations, resulting in the predicate `IsPop`. Note that the predicates `IsSch`, `IsPop` and the AMV axioms form the definition of the predicate `IsAM` (see Definition 4.3). With respect to wellformedness of evolution (see Definition 5.3), no extra rules besides the EW axioms are supposed.

The application model universe for EVORM is defined by the tuple (see Definition 3.2):

$$\mathcal{U}_{\Sigma}^{\texttt{orm}} = \langle \mathcal{U}_{\mathcal{G}}^{\texttt{orm}}, \mathcal{D}, \Omega, \texttt{IsPop}, \gamma, \mu, [\![\,]\!], \texttt{Depends} \rangle .$$

In the next subsection we describe the components $\mathcal{U}_{\mathcal{G}}^{\texttt{orm}}$, $\mathcal{D}$, $\Omega$ and `IsPop`. As we restrict ourselves in this paper to data modelling aspects, the components $\gamma$, $\mu$, $[\![\,]\!]$, and `Depends` fall outside the scope of this paper, and are omitted. After that, we describe the extra rules for

populations, leading to the definition of the underlying domain $\Omega$ for instances, and the relation IsPop.

### 7.1. The information structure universe

The elements for the EVORM information structure universe were introduced in Subsection 6.1. In this subsection we show how this universe fits within the general theory of Subsection 3.1. The EVORM information structure universe is spanned by:

$$\mathcal{U}_{\mathcal{I}}^{\mathrm{orm}} = \langle \mathcal{L}, \mathcal{N}, \sim, \rightsquigarrow, \mathit{IsSch} \rangle .$$

$\mathcal{L}$ is the set of label types that build the information structure universe (see Subsection 3.1). The set $\mathcal{N}$ of *non-label object types* consists of:

$$\mathcal{N} = \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{S} \cup \mathcal{C} .$$

The type relatedness relation $\sim$ for EVORM has already been defined by the TR axioms. The inheritance hierarchy $\rightsquigarrow$ of EVORM, corresponds to the relation $\mathtt{IdfBy}^{-1}$. The relation IsSch has been introduced in Definition 6.1.

#### 7.1.1 Verifying the axioms

The EVORM information structure is a proper information structure conforming to the general evolution theory, thus providing a correct typing system (see Fig. 2):

**Theorem 7.1.** $\mathcal{U}_{\mathcal{I}}^{\mathrm{orm}}$ *is an information structure universe.*

**Proof.** All axioms ISU1 to ISU8 hold:
(1) The axioms ISU1 and ISU2 follow directly from axioms TR1 and TR2.
(2) From axiom EU5 and the TR axioms, axiom ISU3 directly follows.
(3) Axioms ISU4 to ISU7 correspond to axioms EU6 to EU9.
(4) Axiom ISU8 is treated in the following two lemmas below. □

In order to prove the correctness of the $\sim$ and $\rightsquigarrow$ relation of EVORM with respect to the axioms of the general evolution theory, all that remains to be done is to prove that axiom ISU8 holds for the EVORM $\sim$ and $\rightsquigarrow$ relation as well. This is proven in the following two lemmas:

**Lemma 7.1.** *EVORM type relatedness is preserved by* $\rightsquigarrow$:

$$x \sim y \wedge y \rightsquigarrow z \Rightarrow x \sim z .$$

**Proof.** By identification induction on $z$. Suppose each parent of $z$ has the property in question. Now let $x \sim y \wedge y \rightsquigarrow z$. From axiom ISU6 (which has already been proven for EVORM) we conclude the existence of $p$ such that $y \underline{\rightsquigarrow} p \wedge \mathtt{Parent}(p, z)$. From the induction hypothesis we then conclude $x \sim p$. Now by applying axiom TR3 on $x \sim p \wedge \mathtt{Parent}(p, z)$ the result follows. □

**Lemma 7.2.** *EVORM type relatedness is reflected by* $\rightsquigarrow$*:*

$$x \sim y \wedge \neg \mathtt{Root}(y) \Rightarrow \exists_z [x \sim z \wedge z \rightsquigarrow y] .$$

**Proof.** We use parent induction on $x$. As induction hypothesis, let all parents of $x$ have the property. Now suppose $x \sim y \wedge \neg \mathtt{Root}(y)$. The case $x = y$ directly follows from Lemma 7.1. So assume $x \neq y$.

The validity of $x \sim y$ is a result of the application of the derivation rules for $\sim$ (see Section 6.9). Consider a minimal length proof of $x \sim y$. As $y$ is not a root object type, we can conclude $y \in \mathscr{E}$. As a result, the last step in the proof of $x \sim y$ is either an application of axiom TR2 or axiom TR3.

(1) Axiom TR2 is the last step.
Removing this last step then leads to a minimal length proof of $y \sim x$. Now we can conclude that the last step of this proof was an application of axiom TR3. As a result, for some $p$ we have $\mathtt{Parent}(p, y) \wedge p \sim x$, from which the result directly follows.

(2) Axiom TR3 is the last step.
As a result, for some parent $p$ of $x$ we have $\mathtt{Parent}(p, x) \wedge p \sim y$. As $p$ is a parent of $x$, we can apply the induction hypothesis, leading to $p \sim z \wedge p \rightsquigarrow z$ for some $z$. From this, and applying axiom TR3, the result directly follows. □

As stated before, we are able to prove that $\mathtt{Spec}^{-1}$ and $\mathtt{Gen}^{-1}$ are filter relations for $\mathtt{IdfBy}^{-1}$. Even more, they are an identification signature for $\rightsquigarrow$:

**Theorem 7.2.** *The relations* $\mathtt{Spec}^{-1}$ *and* $\mathtt{Gen}^{-1}$ *are an identification signature of* $\mathtt{IdfBy}^{-1}$.

**Proof.**

- Axiom F1 follows for both relations, directly from axiom EU11 and EU13.
- Axiom F2 can be proven for Gen as follows:
  Let $x\ \mathtt{IdfBy}_1\ p$ and $x\ \mathtt{IdfBy}_1\ q$, and $x\ \mathtt{Gen}\ p$. From axiom EU10 follows $x\ \mathtt{Gen}\ q \vee x\ \mathtt{Spec}\ q$. Applying axiom EU14, and $x\ \mathtt{Gen}\ p$, yields $x\ \mathtt{Gen}\ q$.
  The proof for Spec goes analogously.
- From axiom EU10, and axiom EU14, follows:

$$x\ \mathtt{IdfBy}_1\ y \Rightarrow \exists!_{R \in \{\mathtt{Gen}, \mathtt{Spec}\}} [R(x, y)] . \quad \square$$

*7.1.2 Results*

As a result of these last two theorems, the general evolution theory presented in the first half of this paper is applicable. Therefore the properties of the identification hierarchy as proven in Subsection 3.3 also hold for the identification hierarchies from EVORM models. The following two theorems, which are an application of Lemma 3.3 to the definition of $\sqcap$ (see Subsection 6.7) and $\sqcup$ (see Subsection 6.8), are a first result:

**Theorem 7.3.** $\sqcap(x) = \sqcap(y) \Rightarrow x \sim y$.

**Proof.** Applying Lemma 3.3 for $\rightsquigarrow_{\mathtt{Spec}}$ yields:

$$v\ \mathtt{RootOf}_{\mathtt{Spec}}\ x \wedge v\ \mathtt{RootOf}_{\mathtt{Spec}}\ y \Rightarrow x \sim y\ .$$

Combining the definitions of $\mathtt{RootOf}_{\mathtt{Spec}}$ and $\sqcap$, together with the uniqueness of a pater familias, leads to:

$$\sqcap(x) = \sqcap(y) \Rightarrow x \sim y\ .\ \square$$

**Theorem 7.4.** $\sqcup(x) \cap \sqcup(y) \neq \emptyset \Rightarrow x \sim y$.

**Proof.** Applying Lemma 3.3 (page 19) for $\rightsquigarrow_{\mathtt{Gen}}$ yields:

$$v\ \mathtt{RootOf}_{\mathtt{Gen}}\ x \wedge v\ \mathtt{RootOf}_{\mathtt{Gen}}\ y \Rightarrow x \sim y\ .$$

Combining the definitions of $\mathtt{RootOf}_{\mathtt{Gen}}$ and $\sqcup$ leads to:

$$\sqcup(x) \cap \sqcup(y) \neq \emptyset \Rightarrow x \sim y\ .\ \square$$

A further result is that $\sqcap$ and $\sqcup$ are total functions. This follows by applying Corollary 4.1 for $\mathtt{RootOf}_{\mathtt{Spec}}$ and $\mathtt{RootOf}_{\mathtt{Gen}}$ respectively. Finally, Theorem 7.3 can even be strengthened to:

**Theorem 7.5.** $\sqcap(x) = \sqcap(y) \wedge y \sim z \Rightarrow x \sim z$.

**Proof.** Apply Theorem 3.5 with $\rightsquigarrow_{\mathtt{Spec}}$. $\square$

For versions we have, when applying Corollary 4.1 to Gen and Spec respectively, the following corollaries:

**Corollary 7.1.** $a \in \mathcal{O}_t \Rightarrow \sqcup(a) \cap \mathcal{O}_t \neq \emptyset$.

**Corollary 7.2.** $a \in \mathcal{O}_t \Rightarrow \sqcap(a) \in \mathcal{O}_t$.

### *7.2. Populations of information structure versions*

A version of a population at $t$ is a mapping:

$$\mathtt{Pop}_t : \mathcal{O} \rightarrow \wp(\Omega)\ ,$$

where $\Omega$ is the universe of instances that can occur in the population of the information structure universe. For EVORM, the set of instances $\Omega$ is defined in terms of two base sets.

The first set provides the concrete instances. An information structure can only be populated if a link is established between label types and concrete domains. The instances of label types then come from their associated concrete domain. Formally this link has been established by the function $\mathtt{Dom}_t : \mathcal{L} \rightarrow \mathcal{D}$. The range of this function, i.e. $\mathcal{D}$, is the set of concrete domains (e.g. string, natno). These concrete domains form the carriers of a many sorted algebra $\langle \mathcal{D}, F \rangle$, where $F$ is the set of operations (e.g. $+$) on the sorts in $\mathcal{D}$.

The second set provides the atomic abstract instances: $\Theta$. They are used to populate the root entity types. The universe of instances $\Omega$ is inductively defined as the smallest set satisfying:

(1) $\cup \mathcal{D} \subseteq \Omega$. Instances from the sorts in the many sorted algebra are elements of the universe of instances.
(2) $\Theta \subseteq \Omega$, where $\Theta$ is an abstract (countable) domain of (unstructured) values that may occur in the population of entity types.
(3) If $x_1, \ldots, x_n \in \Omega$ and $p_1, \ldots, p_n \in \mathcal{P}$ then $\{p_1{:}x_1, \ldots, p_n{:}x_n\} \in \Omega$ as well. The set $\{p_1{:}x_1, \ldots, p_n{:}x_n\}$ denotes a mapping, assigning $x_i$ to each predicator $p_i$. These mappings are intended to populate fact types.
(4) If $x_1, \ldots, x_n \in \Omega$ then $\{x_1, \ldots, x_n\} \in \Omega$ as well. Sets of instances may occur as instances of power types.
(5) If $x_1, \ldots, x_n \in \Omega$ then $\langle x_1, \ldots, x_n \rangle \in \Omega$. Sequences of instances are used as instances of sequence types (see the Sequence Type Rule).
(6) If $X_1, \ldots, X_n \subseteq \Omega$ and $O_1, \ldots, O_n \in \mathcal{O}$ then $\{O_1{:}X_1, \ldots, O_n{:}X_n\} \in \Omega$ as well. Assignments of sets of instances to object types are also valid instances. They are intended for the populations of schema types.

The population of a root entity type is a set of values, taken from the abstract domain $\Theta$.

**[P1].** If $x \in \mathcal{E}_t$ and $\mathtt{Root}(x)$ then:

$$\mathtt{Pop}_t(x) \subseteq \Theta\,.$$

The population of a fact type is a set of tuples. A tuple $t$ in the population of a fact type $f$ is a mapping of all its predicators to values of the appropriate type. This is referred to as the *Conformity Rule*:

**[P2].** If $x \in \mathcal{F}_t$ and $y \in \mathtt{Pop}_t(x)$ then:

$$y{:}\, x \rightarrow \Omega \wedge \forall_{p \in x}[y(p) \in \mathtt{Pop}_t(\mathtt{Base}(p))]\,.$$

The population of a power type consists of (nonempty) sets of instances of the corresponding element type. This is called the *Power Type Rule*:

**[P3].** If $x \in \mathcal{G}_t$ and $y \in \mathtt{Pop}_t(x)$ then:

$$y \in \wp^+(\mathtt{Pop}_t(\mathtt{Elt}(x)))\,.$$

The population of a sequence type consists of (nonempty) sequences of instances of the corresponding element type. This is called the *Sequence Type Rule*:

**[P4].** If $x \in \mathcal{S}_t$ and $y \in \mathtt{Pop}_t(s)$ then:

$$y \in \mathtt{Pop}_t(\mathtt{Elt}(x))^+$$

The population of a composition type consists of populations of the underlying information structure. This is called the *Decomposition Rule*:

**[P5].** If $x \in \mathcal{C}_t$ and $y \in \mathtt{Pop}_t(x)$ then:

$$\mathtt{IsPop}(\mathcal{I}_t^x, y)\,.$$

The nature of generalisation requires the following rule:

**[P6].** $x\texttt{Gen}y \Rightarrow \texttt{Pop}_t(y) \subseteq \texttt{Pop}_t(x)$.

The P axioms lead to the definition of the IsPop predicate, completing the well-formedness rules on versions of EVORM models:

**Definition 7.1.** $\texttt{IsPop}(\mathcal{O}_t, \texttt{Pop}_t) \triangleq \texttt{Pop}_t$ and $\mathcal{O}_t$ adhere to the *P* axioms.

Now let $\Sigma_t$ be a correct application model version ($\texttt{IsAM}(\Sigma_t)$). We focus at the population in this version, and reconsider the results from Subsection 4.4. Respecting the specialisation hierarchy is reflected by the *Specialisation Rule*, which follows directly from Lemma 4.2 and Axiom EU12.

**Corollary 7.3.** $\texttt{Pop}_t(x) \subseteq \texttt{Pop}_t(\sqcap(x))$.

This rule does not require that instances of subtypes have to fulfil the subtype defining rule associated to the involved subtype. A subtype defining rule is defined as an information descriptor (see [25]). Up to this point no language for the formulation of such rules is available. The subtype defining rule should however also be considered as a population derivation rule, the population of a subtype can be computed using this rule.

Respecting the Generalisation hierarchy is reflected by the *Generalisation Rule*, which follows from Lemma 4.2 and Axiom P6.

**Corollary 7.4.** $\texttt{Pop}_t(x) = \bigcup_{y \in \sqcup(x)} \texttt{Pop}_t(y)$.

The *Generalisation Rule*, which clearly is a derivation rule, requires that the population of a generalised object type ($x$) is completely covered by the populations of its specifiers.

### *7.3. The running example*

In this section we describe the example of Subsection 2.2 in terms of EVORM. As the information structure versions are most adequately represented by the drawing technique of the underlying data modelling technique (PSM in case of EVORM), we concentrate in this section on describing the evolution steps. In the first evolution step

(1) a subtyping of object type Car into object types Private Car and Company Car is introduced,
(2) the object type Kilometrage, identified by Km is created,
(3) the relationship type Usage is added.

This is denoted in the style of Elisa-D (see [41, 39]) as follows:

```
CREATE HISTORY Private Car AS
  ENTITY TYPE Private Car SUBTYPE OF Car
```

```
CREATE HISTORY Company Car AS
  ENTITY TYPE Company Car SUBTYPE OF Car;
CREATE HISTORY Usage Dimension AS
  ENTITY TYPE Kilometrage (Km);
CREATE HISTORY Usage AS
  FACT TYPE Usage (has a:of (Private Car), of:is (Kilometrage))
```

Note that application model histories may have the same name as object types. This will not lead to ambiguity. In the next evolution step the subtyping of Car is abolished. This is communicated to the information system by:

```
TERMINATE HISTORY Private Car, Company Car;
MODIFY HISTORY Usage TO
  FACT TYPE Usage (has a:of (Car), of:is (Kilometrage))
```

Note that object types in the extra temporal schema are allowed to bear the same name.

## 8. Conclusions

In this paper we presented a way of modelling for evolving application domains in the form of a general theory, and an application of this theory to the data modelling technique PSM resulting in EVORM. In this application, we introduced four classes of axioms for EVORM:

- EU: typing mechanism and information structure universe
- TR: type relatedness
- EV: schema wellformedness
- P: population wellformedness .

In Fig. 16, these classes are related to the original framework of the general theory as depicted in Fig. 2.

The next step is to find suitable representation mechanisms for the concepts of the theory, i.e. a proper way of communicating. In a forthcoming paper [41], we will present a way of communicating which leads to the formulation of queries and updates in a semi-natural language.

This language is strongly related to the language which is to be used by domain experts to describe the underlying Universe of Discourse. As a result, the communication language has a strong intuitive meaning for users, as it resembles their way of talking within their Universe of Discourse. This approach corresponds to the way of thinking from the NIAM modelling method. This language will provide the possibility to query across boundaries of schema versions. This is not possible in traditional relational algebra based languages (such as [34]).

Future research may address an effective way of working, based on this way of communicating. For the efficiency of a development process, based on this way of working, a way of

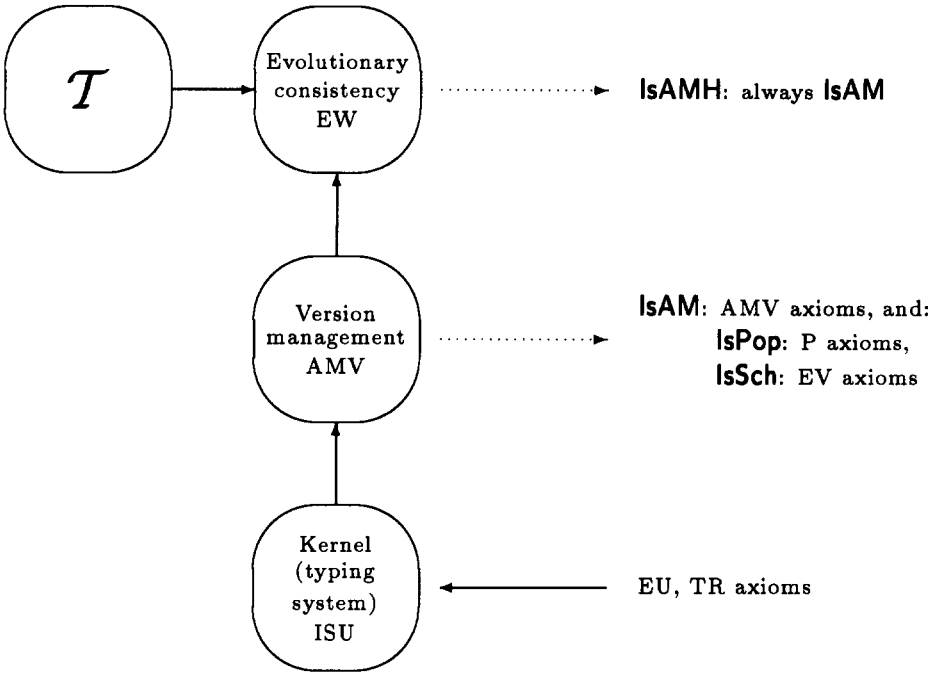



Fig. 16. Axiomatic framework, revisited.

controlling has to be developed. Finally, an Evolving Information Systems Management System has to be implemented, leading to a way of supporting.

## Acknowledgement

We would like to thank Udo Lipeck for his contributive remarks. The remarks of the anonymous referees resulted in many improvements.

## References

[1] S. Abiteboul and R. Hull, IFO: A formal semantic database model, *ACM Trans. Database Syst.* 12(4) (Dec. 1987) 525–565.
[2] J.F. Allen, Towards a general theory of action and time, *Artificial Intelligence* (23) (1984) 123–154.
[3] G. Ariav, A temporally oriented data model, *ACM Trans. Database Syst.* 11(4) (Dec. 1986) 499–527.
[4] J. Banerjee, W. Kim, H.J. Kim and H.F. Korth, Semantics and implementation of schema evolution in Object-Oriented Databases, *SIGMOD Rec.* 16(3) (Dec. 1987) 311–322.
[5] M. Barr and C. Wells, *Category Theory for Computing Science* (Prentice-Hall, Englewood Cliffs, NJ, 1990).
[6] P. van Bommel, A.H.M. ter Hofstede and Th.P. van der Weide, Semantics and verification of object-role models, *Informat. Syst.* 16(5) (Oct. 1991) 471–495.
[7] R. Bretl, D. Maier, A. Otis, D.J. Penney, B. Schuchardt, J. Stein, E.H. Williams and M. Williams, The GemStone data management system, in: W. Kim and F.H. Lochovsky, eds., *Object-Oriented Concepts, Databases and Applications* (ACM Press, Frontier Series, Addison-Wesley, Reading, MA, 1989) 283–308.

[8] K.B. Bruce and P. Wegner, An algebraic model of subtype and inheritance, in: F. Bancilhon and P. Buneman, eds., *Advances in Database Programming Languages* (ACM Press, Frontier Series, Addison-Wesley, Reading, MA, 1990) 75–96.

[9] P.D. Bruza and Th.P. van der Weide, The semantics of data flow diagrams, in: N. Prakash, ed., *Proc. Int. Conf. on Management of Data*, Hyderabad, India (1989).

[10] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism, *ACM Comput. Surveys* 17(4) (Dec. 1985) 471–522.

[11] P.P. Chen, The Entity-Relationship model: Toward a unified view of data, *ACM Trans. Database Syst.* 1(1) (March 1976) 9–36.

[12] J. Clifford and A. Rao, A simple, general structure for temporal domains, in: C. Rolland, F. Bodart and M. Leonard, eds., *Temporal Aspects in Information Systems* (North-Holland/IFIP, Amsterdam, 1987) 17–28.

[13] 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, eds., *Proc. Third Int. Conf. CAiSE'91 on Advanced Information Systems Engineering*, vol. 498 of *Lecture Notes in Computer Science*, Trondheim, Norway (May 1991) (Springer-Verlag, Berlin) 561–578.

[14] E. Dubois, J. Hagelstein and A. Rifaut, Formal requirements engineering with ERAE, *Philips J. Res.* (43) (1988) 393–414.

[15] E. Dubois, J. Hagelstein, E. Lahou, A. Rifaut and F. Williams, A formalisation of entities, relationships, attributes, and events, Philips Manuscript M105, Philips Research Laboratory, Brussels, Belgium, 1985.

[16] E.D. Falkenberg, J.L.H. Oei and H.A. Proper, A conceptual framework for evolving information systems, in: H.G. Sol and R.L. Crosslin, eds., *Dynamic Modelling of Information Systems II* (North-Holland, Amsterdam, The Netherlands, 1992) 353–375.

[17] E.D. Falkenberg, J.L.H. Oei and H.A. Proper, Evolving information systems: Beyond temporal information systems, in: A.M. Tjoa and I. Ramos, eds., *Proc. Data Base and Expert System Applications Conf. (DEXA 92)*, Valencia, Spain (Sep. 1992) (Springer-Verlag, Berlin) 282–287.

[18] J.J. van Griethuysen, ed., Concepts and terminology for the conceptual schema and the information base, Publ. nr. ISO/TC97/SC5-N695, 1982.

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

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

[21] K.M. van Hee, L.J. Somers and M. Voorhoeve, Executable specifications for distributed information systems, in: E.D. Falkenberg and P. Lindgreen, eds., *Information System Concepts: An In-depth Analysis* (North-Holland/IFIP, Amsterdam, The Netherlands, 1989) 139–156.

[22] A.H.M. ter Hofstede, *Information modelling in data intensive domains*, PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[23] A.H.M. ter Hofstede and E.R. Nieuwland, Task structure semantics through process algebra, *Software Eng. J.* 8(1) (Jan. 1993) 14–20.

[24] A.H.M. ter Hofstede, H.A. Proper and Th.P. van der Weide, Data modelling in complex application domains, in: P. Loucopoulos, ed., *Proc. Fourth Int. Conf. CAiSE'92 on Advanced Information Systems Engineering*, vol. 593 of *Lecture Notes in Computer Science*, Manchester, United Kongdom (May 1992) (Springer-Verlag) 364–377.

[25] 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, *Informat. Syst.* 18(7) (1993) 489–523.

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

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

[28] M. Jarke, J. Mylopoulos, J.W. Schmidt and Y. Vassiliou, DAIDA: An environment for evolving information systems, *ACM Trans. Informat. Syst.* 20(1) (Jan. 1992) 1–50.

[29] R.H. Katz, Toward a unified framework for version modelling in engineering databases, *ACM Comput. Surv.* 22(4) (1990) 375–408.
[30] W. Kim, N. Ballou, H.-T. Chou, J.F. Garza and D. Woelk, Features of the ORION object-oriented database, in: W. Kim and F.H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications* (ACM Press, Frontier Series, Addison-Wesley, Reading, MA, 1989) 251–282.
[31] T. Korson and J. McGregor, Understanding object oriented: A unifying paradigm. *Commun. ACM* 33(9) (Sept. 1990) 40–60.
[32] U.W. Lipeck and G. Saake, Monitoring dynamic integrity constraints based on temporal logic, *Informat. Syst.* 12(3) (1987) 255–269.
[33] P. McBrien, A.H. Seltviet and B. Wangler, An Entity-Relationship Model extended to describe historical information, in: A.K. Majumdar and N. Prakash, eds., *Proc. Int. Conf. on Information Systems and Management of Data* (*CISMOD 92*), Bangalore, India (July 1992) 244–260.
[34] E. McKenzie and R. Snodgrass, Schema evolution and the relational algebra, *Informat. Syst.* 15(2) (1990) 207–232.
[35] J. Mylopoulos, A. Borgida, M. Jarke and M. Koubarakis, Telos: Representing knowledge about information systems, *ACM Trans. Informat. Syst.* 8(4) (1990) 325–362.
[36] G.M. Nijssen and T.A. Halpin, *Conceptual Schema and Relational Database Design: A Fact Oriented Approach* (Prentice-Hall, Sydney, Australia, 1989).
[37] A. Ohori, Orderings and types in databases, in: F. Bancilhon and P. Buneman, eds., *Advances in Database Programming Languages* (ACM Press, Frontier Series, Addison-Wesley, Reading, MA, 1990) 97–116.
[38] D.J. Penney and J. Stein, Class modification in the GemStone Object-Oriented DBMS, in: N. Meyrowitz, ed., *Proc. ACM Conf. Object-Oriented Systems, Languages and Applications* (*OOPSLA*), Orlando, FL (Oct. 1987) 111–117.
[39] H.A. Proper, A theory for conceptual modelling of evolving application domains, PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1994 (forthcoming).
[40] H.A. Proper and Th.P. van der Weide, A general theory for the evolution of application models, Technical Report 92-26, Department of Information Systems, University of Nijmegen, Nijmegen, The Netherlands, 1992.
[41] H.A. Proper and Th.P. van der Weide, Information disclosure in evolving information systems: Taking a shot at a moving target, Technical Report 93-22, Information Systems Group, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, 1993.
[42] H.A. Proper and Th.P. van der Weide, Towards a general theory for the evolution of application models, in: M.E. Orlowska and M. Papazoglou, eds., *Proc. Fourth Australian Database Conf., Advances in Database Research* (World Scientific, Brisbane, Australia, Feb. 1993) 346–362.
[43] J.F. Roddick, Dynamically changing schemas within database models, *The Australian Comput. J.* 23(3) (Aug. 1991) 105–109.
[44] J.F. Roddick and J.D. Patrick, Temporal semantics in information systems – A survey, *Informat. Syst.* 17(3) (1992) 249–267.
[45] G. Saake, Spezifikation, Semantik und Überwachung von Objektlebensläufen in Datenbanken, PhD thesis, Technische Universität Bruansweig, Braunsweig, Germany, 1988 (in German).
[46] G. Saake, Descriptive specification of database object behaviour, *Data & Knowledge Eng* 6(1) (1991) 47–73.
[47] P.S. Seligmann, G.M. Wijers and H.G. Sol, Analyzing the structure of I.S. methodologies, an alternative approach, in: R. Maes, ed., *Proc. First Dutch Conf. on Information Systems* (1989).
[48] A.H. Skarra and S.B. Zdonik, The management of changing types in an object-oriented database, in: N. Meyrowitz, ed., *Proc. ACM Conf. Object-Oriented Systems, Languages, and Applications* (*OOPSLA*), Portland, OR (Sept. 1986) 483–495.
[49] R. Snodgrass, Temporal databases status and research directions, *SIGMOD Rec.* 19(4) (Dec. 1990) 83–89.
[50] R. Snodgrass and I. Ahn, A taxonomy of time in databases, in: *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, Austin, TX (1985) 236–246.
[51] C. Theodoulidis, P. Loucopoulos and B. Wangler, A conceptual modelling formalism for temporal database applications, *Informat. Syst.* 16(4) (1991) 401–416.

[52] M.T. Tresch, A framework for schema evolution by meta object manipulation, in: *Proc. 3d Int. Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria (Sep. 1991) (Institut für Informatik, TU Clausthal).

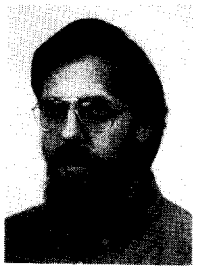
[53] M.T. Tresch and M.H. Scholl, Meta object management and its application to database evolution, in: G. Pernul and A.M. Tjoa, eds., *11th Int. Conf. on the Entity-Relationship Approach*, vol. 645 of *Lecture Notes in Computer Science*, Karlsruhe, Germany (Oct. 1992, Springer-Verlag) 299–321.

[54] G. Wiederhold, S. Jajodia and W. Litwin, Dealing with the granularity of time in temporal databases, in: R. Andersen, J.A. Bubenko and A. Sølvberg, eds., *Proc. Third Int. Conf. CAiSE'91 on Advanced Information Systems Engineering*, vol. 498 of *Lecture Notes in Computer Science*, Trondheim, Norway (May 1991. Springer-Verlag) 124–140.

[55] G.M. Wijers, Modelling support in information systems development, PhD thesis, Delft University of Technology, Delft, The Netherlands, 1991.

[56] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.T. Meertens and R.G. Fisker, *Revised Report on the Algorithmic Language ALGOL 68* (Springer-Verlag, Berlin, Germany, 1976).

**H.A. Proper** received his masters degree from the University of Nijmegen, the Netherlands in 1990. He is currently a Ph.D. student at the University of Nijmegen, the Netherlands, and expects to receive his Ph.D. before the summer of 1994. His main research interests include (evolving) information systems, information retrieval, hypertext and knowledge based systems.



**Th.P. van der Weide** received his masters degree from the Technical University Eindhoven, the Netherlands in 1975, and the degree of Ph.D. in Mathematics and Physics from the University of Leiden, the Netherlands in 1980. He is currently associate professor at the University of Nijmegen, the Netherlands. His main research interests include information systems, information retrieval, hypertext and knowledge based systems.