A General Theory for the Evolution of Application Models

H.A. Proper[1] and Th.P. van der Weide

Computing Science Institute, University of Nijmegen
Toernooiveld, NL-6525 ED Nijmegen, The Netherlands
E.Proper@acm.org

Abstract

In this article we focus on evolving information systems. First a delimitation of the concept of evolution is provided, resulting in a first attempt to a general theory for such evolutions.

The theory makes a distinction between the underlying information structure at the conceptual level, 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. In terms of these concepts, we propose some axioms on the well-formedness of evolution.

In this general theory, the underlying data model is a parameter, making the theory applicable for a wide range of modelling techniques, including object-role modelling and object oriented techniques.

PUBLISHED AS:

H.A. Proper and Th.P. van der Weide. A General Theory for the Evolution of Application Models. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):984–996, December 1995.

Keywords: Evolving Information Systems, Temporal Information Systems, Schema Evolution, Data Modelling, Type Relatedness, Predicator Set Model, ER Model

1 Introduction

As has been argued in [Rod91] and [FOP92b], 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 reprogramming.

The intention of an evolving information system ([FOP92a], [OPF94]) is to be able to handle updates of all components of the so-called *application model*, containing the information structure, the constraints on this structure, the population conforming to this structure and the possible operations. The theory of such systems should, however, be independent of whatever modelling technique is used to describe the application model. In this paper, we discuss a general theory for the evolution of application models. However, only conceptual aspects are considered, focus is on *what* evolution is, rather than on *how* to implement evolution in a database manegement system. In [PW93], an informal introduction to this theory is provided.

The central part of this theory will make weak assumptions on the underlying modelling technique, making it therefore applicable for a wide range of data modelling techniques such as ER ([Che76], [You89]), EER

[1]Currently at: Department of Computer Science, University of Queensland, Queensland 4072, Australia

([EGH$^+$92]), NIAM ([NH89]), IFO ([AH87]) and the generalized object role data modelling technique PSM ([HW93], [HPW92]), action modelling techniques such as Task Structures ([WHO92], [HN93]), DFD ([BW89]) and ExSpect ([HSV89]), and furthermore object oriented modelling techniques ([KM90]) adhering to the OO typing mechanism as described in [CW85]. In [PW94], the application of the theory presented in this article to the object-role modelling technique PSM, leading to EVORM, is described. Some of the reasons for choosing an object-role modelling technique as a first application of the general theory, are the steadily increasing popularity of object role modelling ([Nij93], [HM94]), and the existence of a formal definition of object-role modelling. An further advantage of object-role modelling is that it is supported by an effective, natural language based, conceptual schema design procedure (way of working). The extension of this technique to cover evolution aspects are not covered by this paper.

The assumptions suppose 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.

In [Sno90] 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 ([TLW91], [MSW92]), the ER model is enhanced with the notion of time, resulting in the ERT model. In TODM ([Ari86]) and ERAE ([DHL$^+$85]), similar strategies are followed, extending the relational model with the notion of time. This makes it possible to handle historical data, over a (non-varying) underlying information structure. In [LS87], [Saa91] and [Saa88] 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, providing first results on the actual implementation of evolution of schemas in database management systems. In [SZ86] a taxonomy for type evolution in object oriented database management systems is provided. The ORION project ([BKKK87], [KBC$^+$89]) offers a more detailed taxonomy, together with a (semi formal) semantics of schema updates restricted to object oriented databases. The ORION system, together with the GemStone ([PS87], [BMO$^+$89]), and Sherpa ([NR89]) systems, are among the first object oriented database systems to support schema/type evolution. In the Cocoon project ([Tre91], [TS92]) 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. This paper differs from these object oriented database management systems, in that the paper focusses on the underlying concepts of evolution rather than implementation of these concepts.

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 ([Kat90], [MBJK90], [JMSV92]). 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. Experiences in the field of version modelling can be fruitfully employed when actually implementing an evolving information system (management system).

A third subclass of research regarding evolving information systems extends a manipulation language for relational models with historical operations, both on population and schema level. An example of this approach can be found in [MS90], in which an algebra is presented allowing relational tables to evolve by changing their arity. This direction is similar to the ORION project ([BKKK87], [KBC$^+$89]), in that a manipulation language is extended with operations supporting schema evolution.

In this paper we consider evolving information systems, and try to abstract from the subclasses mentioned above. Therefore, we take the underlying informaton structuring technique for granted, make only weak

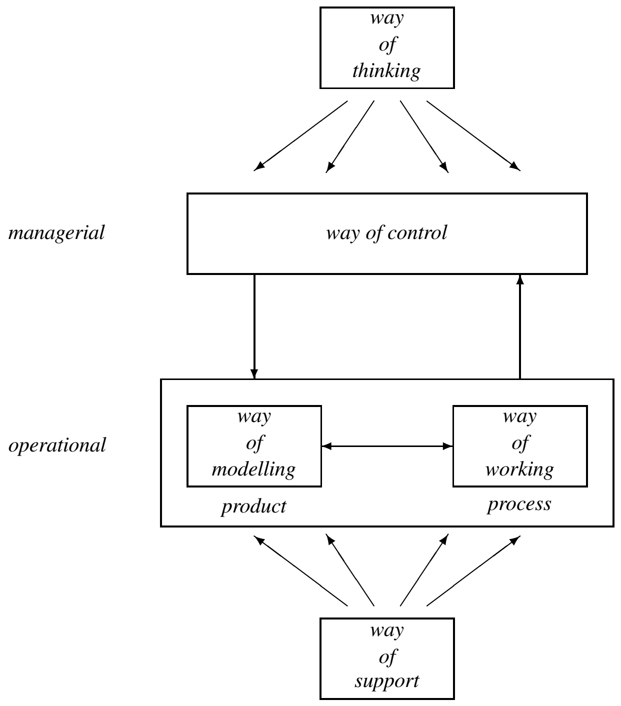


Figure 1: Framework for methodologies

assumptions on the underlying technique, and limit ourselves to conceptual issues. This paper restricts itself basically to the *way of modelling* in the framework for methodologies of figure 1. This framework, taken from [Wij91], presents a more structured view on methodologies, and is based on the original framework of [SWS89]. It makes a distinction between a way of thinking, a way of control, a way of modelling, a way of working and a way of support. The way of thinking is concerned with the philosophy behind the methodology and contains basic assumptions and viewpoints of this methodology. The way of control captures project management. The way of modelling describes the models and model components used in the methodology, while the way of working describes strategies and procedures of how to arrive at specific models. The way of support, finally, is concerned with technique/method support. The GemStone, ORION, Sherpa and Cocoon systems can thus be seen as first attempts for a way of support for evolving information systems. However, to our knowledge, all these systems lack a rigourously formalised underlying way of modelling. Although it is benificial to have a running way of support as soon as possible, having a well thought out underlying way of modelling first has proven its usefullness. At least, this should be the second goal after completing the tool!

The structure of the paper is as follows. In section 2 we describe the approach that has been taken to the concept of evolution, in which evolution is seen (similar as history books) as an ensemble of individual histories of application model elements. As we will not focus on a particular modelling technique, section 3 describes the minimal requirements for an underlying technique, as discussed above. In section 5 we introduce the universe for application model evolution. After that, we discuss what constitutes a wellformed application model version. In section 6 the evolution of application models is treated, and some wellformedness rules for such evolutions are formulated.

2 An Approach to Evolving 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 The extent of the corpus evolutionis

According to [ISO87], 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.

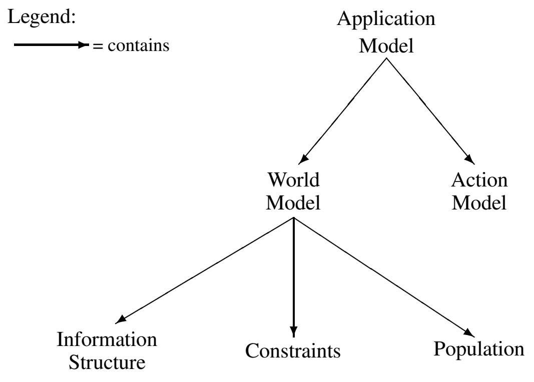


Figure 2: A hierarchy of models

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 (as good as possible) modelled by means of Petri-net like specifications (such as ExSpect or Task Structures), or languages based on 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* ([FOP92a], [PW93]). The resulting hierarchy of models is depicted in figure 2.

The part of an (evolving) information system that is allowed to change over time, will be referred to as *corpus evolutionis*.

In most traditional information systems, the corpus evolutionis is restricted to the population. Nevertheless, some traditional information systems do support modifications of other components from the application model, to a limited extend. 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, which also leads to loss of the old table! In an evolving information system, the entire application model is allowed to evolve on-line, while keeping track of the entire history of the application model. As a result, no information is lost in the course of history. Note that the information capacity (introduced in [Hul86]) may vary in the course of time.

The application model can then be looked upon as the formal denotation of the corpus evolutionis, and 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. Thus, 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.

2.2 An example of evolution

As an illustration of an evolving universe of discourse, consider a rental store for audio records (LP's). In this store a registration is maintained of the songs that are recorded on the available LP's. In order to keep track of the wear and tear of LP's, the number of times an LP has been lent, is registered. The information structure and constraints of this universe of discourse are modelled in figure 3 in the style of ER, according to the conventions of [You89]. Note the special notation of attributes (Title) using a mark symbol (#) followed by the attribute (# Title).

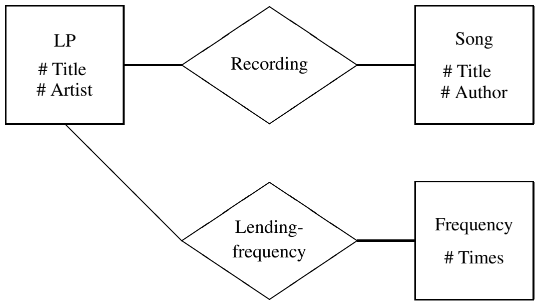


Figure 3: The information structure of an LP rental store

An action specification in this example is the rule Init-freq, stating that whenever a new LP is added to the assortment of the store, it's lending frequency must be set to 0:

```
ACTION Init-freq =
    WHEN ADD Lp:$x$ DO
       ADD Lp:$x$ has Lending-frequency of Frequency:0
```

This action specification is in the style of LISA-D ([HPW93]). Note that the keyword 'has' connects object types to relation types, and the keyword 'of' just the other way around.

After the introduction of the compact disc, and its conquest of a sizeable piece of the market, the rental store has transformed into an 'LP and CD rental store'. This leads to the introduction of the object type Medium as a common supertype (denominator) for LP and CD. This makes CD and LP to subtypes of Medium. Note that some modelling techniques (ECR, PSM, IFO) also feature a construct allowing for the introduction of generalised (polymorphic) types. However, since LPs and CDs are identified by the same properties (Title and Artist), in this case a subtyping relation is the most appropriate solution. The relation type Medium-type effectuates the subtyping of Medium into LP and CD. In the new situation, the registration of songs on LP's is extended to cover CD's as well. The frequency of lending, however, is not kept for CD's, as CD's are hardly subject to any wear and tear. As a consequence, the application model has evolved to figure 4. This requires an update of the typing relation of instances of object type LP, which are now instances of both LP and Medium. Note that this modification can be done automatically.

The action specification Init-freq evolves accordingly, now stating that whenever a medium is added to the assortment of the rental store, it's lending frequency is set to 0 provided the medium is an LP:

```
ACTION Init-freq =
    WHEN ADD Medium:$x$ DO
       IF Lp:$x$ THEN
          ADD Lp:$x$ has Lending-frequency of Frequency:0
```

After some years, the CD's have become more popular than LP's. Consequently, the rental store has decided to stop renting LP's and to become a CD rental store. Besides, the recording quality of songs on CD's has appeared to be relevant for clients. As this quality may differ from song to song on a single CD,

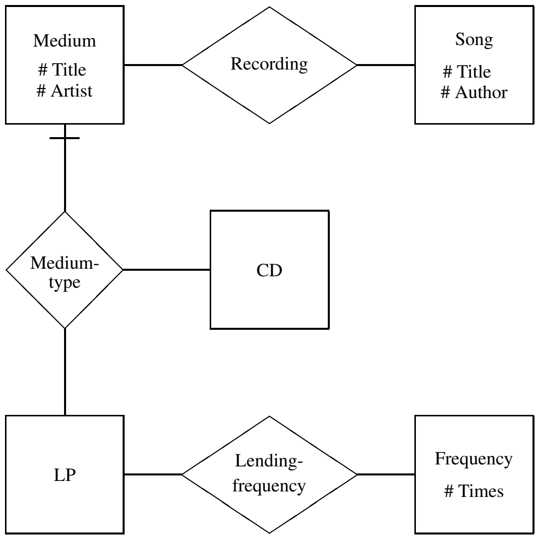


Figure 4: The information structure of a LP and CD rental store

and may for some song be different for recordings on different CD's, the recording quality is added as a (mandatory) attribute to the Recording relation.

This change in the rental store, leads to the information structure as depicted in figure 5. As a result of this evolution step, the action specification Init-freq can be terminated, since the lending frequency of CD's is not recorded anymore. Furthermore, the addition of the mandatory attribute Quality enforces an update of the existing population. In this case, contrary to the previous evolution step, information has to be added to the old population. This could, for example, be effectuated by the following transaction:

```
ADD TO Recording MANDATORY ATTRIBUTE Quality;
UPDATE Recording SET Quality = 'AAD'
```

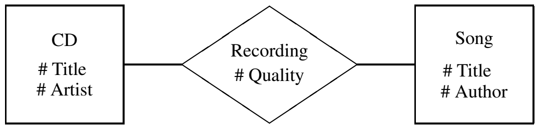


Figure 5: The information structure of a CD rental store

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. A first approach is to model the history of application model elements by adding birth-death relations to all object types in the information structure ([TLW91]), as illustrated in figure 6.

This approach, however, is very limited, as it only enables the modelling of evolution of the population of an information system. For example, the evolution of the Recording relation type can not be modelled in this approach. Evolution of other application model elements than from the population, must then be described by using a meta modelling approach ([SA85]).

This paper takes another approach, and treats evolution (or rather the time axis) of an application model as a separate concept. This still makes it possible to derive the view on the evolution of the application

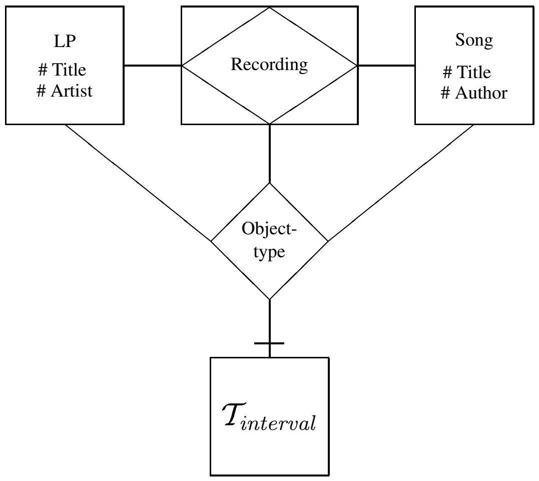


Figure 6: Adding history explicitly

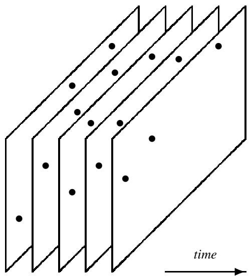


Figure 7: Evolution modelled by snapshots

model of the first approach. It will, in particular, still be possible to derive the view on the evolution of the population as presented in figure 6. Note that this approach has a resemblance to the approach from [SZ86], which, however, is more restricted in the sense that is more directed towards an implementation.

When taking the second approach, there still are two alternatives to deal with the history of application models. The first one is to maintain a version history of application models in their entirety. This alternative leads to a sequence of snapshots of application models, as illustrated in figure 7. The second alternative, is to keep a version history per element, thus keeping track of the evolution of individual object types, instances, methods, etc. This has been illustrated in figure 8. Each dotted line corresponds to the evolution of one distinct element.

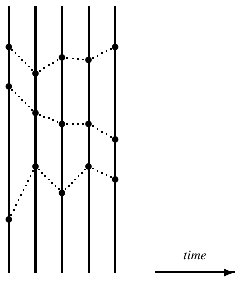


Figure 8: Evolution modelled by functions over time

The major advantage of the second alternative is that it enables one to state rules about, and query, the evolution of distinct application model elements. The first alternative clearly does not offer this oppertunity, as it does not provide relations between successive versions of the application model elements.

Furthermore, the snapshot view from the first alternative can be derived by constituting the application model version of any point of time from the current versions of its components (consequently the view on the evolution of populations of the first approach can be derived as well). This derivation is examplified in figure 9. In the theory of evolving application models we will therefore adapt the second alternative.

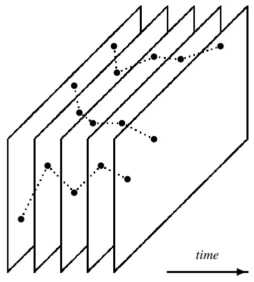


Figure 9: Deriving snapshots from element evolutions

The above discussion has a parallel in the description of the history (evolution) of the world. Many approaches are possible. One may choose to describe the evolution of the world as a sequence of snapshots, where each snapshot contains all facts valid at a given point in time. This (higly inefficient) way of describing the history of the world is not used in practice, as one always wants to know the distinct evolution of persons, municipalities, families, laws, countries, etc. Consequently, if one wants to make a complete description of the evolution of the world, the evolutions of all relevant components of the world (persons, mountains, tectonic plates, ...) have to be described separately. Finally, we realise that the approach we take to the evolution of application models is not new. The described approach is in line with approaches discussed in e.g. [SZ86], [BKKK87], and [Kat90]. However, in this article we try to use this approach as a corner stone for a theory of application model evolution that abstracts as much as possible from underlying concrete modelling techniques and from implementation related details. It is this theory that is the main contribution of this article. The aim of the theory is not to reject or replace any of the existing approaches to schema evolution, but rather to complement it and provide a more elaborate theoretical background.

2.4 Evolving information systems

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 Recording in the rental store. When CD's are added to its assortment, the version of the application model element Recording changes from a relation type registrating songs on LP's, to a relation type registrating songs on Media. The removal of LP's from the assortment leads to the change of the application model element Recording into a relation type registrating songs on CD's.

The domain $\mathcal{AMH}$ for application model histories is determined by the following components:

1. The set $\mathcal{AME}$ is the domain for the evolvable elements of an application model. A formal definition of $\mathcal{AME}$ will be provided in section 6.

[1]In this paper, the difference between recording and event time [SA85], and the ability to correct stored information are not taken into consideration. For more details, see [FOP92a] or [FOP92b].

2. Time, essential to evolution, is incorporated into the theory through the algebraic structure $\mathcal{T}_s = \langle \mathcal{T}, F \rangle$, where $\mathcal{T}$ is a (discrete, totally ordered) time axis, and F a set of functions over $\mathcal{T}$. For the moment, F is assumed to contain the one-step increment operator $\triangleright$, and the comparison operator $\leq$. Several ways of defining a time axis exist, see e.g. [CR87], [WJL91] or [All84].

   The time axis is the axis along which the application model evolves. With this time axis, an application model history is a (partial) mapping $\mathcal{T} \rightarrowtail \mathcal{AME}$.[2] $\mathcal{AMH}$ is the set of all such histories. In a later section, we will pose well-formedness restrictions on histories.

   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 [RP92]. The linear time model is usually chosen in historical databases (see for example [Sno90]).

3. $\mathcal{M}$ is the domain for actions that can be performed on application model histories.

4. The semantics of the actions in $\mathcal{M}$ is provided by the state transition relation on application model histories: $[\![\ ]\!] \subseteq \mathcal{M} \times \mathcal{T} \times \mathcal{AMH} \times \mathcal{AMH}$, where $H\, [\![m]\!]_t\, H'$ means: H' may result after applying action m to H at time t. In business applications, most actions will turn out to be deterministic. However, sometimes it is useful to allow for nondeterminism; for example when external influences can effect the outcome of a process, while these influences themselves are not considered part of the universe of discourse.

   Our way of abstracting the semantics of actions was inspired by the Temporal Logic of Actions as discussed in [Lam91].

In the example of subsection 2.2, the application model history is told by the evolution of its elements, such as LP, CD and Recording. Their histories are summarized in table 1.

	point of time	LP	CD	Recording
time ↓	t_0	object type	*non existent*	relation between LP and Song
	t_1	subtype	subtype	relation between Medium and Song
	t_2	*non existent*	object type	relation between CD and Song

Table 1: Some element histories

2.5 A dual vision

The execution of an action at some point of time is referred to as an *event occurrence*. The domain of sequences of event occurrences is identified by:

Definition 2.1 (*event occurrence sequence domain*)

$$\mathcal{EO} = \mathcal{T} \rightarrowtail \mathcal{M}$$

□

An application model history (H) describes the evolution of an underlying application. A prefix of this history describes the evolution of this application upto some point of time, and forms a *state* of an associated evolving information system. First we introduce prefixing of a single element evolution:

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

Definition 2.2 (*element evolution prefix*)
If $h : \mathcal{T} \rightarrowtail \mathcal{AME}$ then the prefix of h at time t is:

$$h_{|t} = \lambda s.\textbf{if } s \leq t \textbf{ then } h(s) \textbf{ else } h(t) \textbf{ fi}$$

□

Some obvious properties, concerning idempotence, for history prefixing are:

Lemma 2.1 If $h : \mathcal{T} \rightarrowtail \mathcal{AME}$, and $u \leq t$ then:

$$(h_{|t})_{|u} = h_{|u}$$

Lemma 2.2 If $h : \mathcal{T} \rightarrowtail \mathcal{AME}$, and $t \leq u$ then:

$$(h_{|t})_{|u} = h_{|t}$$

The states of an evolving information system, tracking application model history H, are identified by:

Definition 2.3 (*evolving information system state*)
If $H \in \mathcal{AMH}$ then the state of H at time t is:

$$H_{|t} = \left\{ h_{|t} \;\middle|\; h \in H \right\}$$

□

Note that each state of an evolving information system is an application model history as well ($H_{|t} \in \mathcal{AMH}$). States are also referred to as *initial histories*. The result of lemma 2.1 and lemma 2.2 can be generalised to:

Corollary 2.1 If H is an application model history, then

$$\begin{array}{lcl} u \leq t & \Rightarrow & (H_{|t})_{|u} = H_{|u} \\ t \leq u & \Rightarrow & (H_{|t})_{|u} = H_{|t} \end{array}$$

The evolution of an application model is described by an application model history H. Besides, this evolution may be modelled as a sequence E of event occurrences, specifying subsequent changes to initial histories of the application model, starting from the initial application model. Thus the combination of E and H leads to a dual vision on states of evolving information systems. On the one hand, a state results from a set of event occurrences. On the other hand, a state is a prefix of an application model history.

In figure 10, a broader view on this duality is depicted. A user observes the history of a universe of discourse (H_{uod}), and formulates the observed evolution by means of a set of event occurrences (E). These event occurrences result, by means of the Behaves relation, in an application model history (H_{is}) as stored in the information system. This latter application model history must be validated (emperically) against the observed history of the universe of discourse.

The relation between an application model history H, and a set of event occurences E is captured by the Behaves predicate:

Definition 2.4
Let $E \subseteq \mathcal{EO}$ and $H \in \mathcal{AMH}$, then:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathsf{Behaves}(E, H)} \\ \triangleq & \forall_{\langle t,m \rangle \in E} \left[H_{|t} \, [\![m]\!]_t \, H_{|\rhd t} \right] \\ \wedge & \forall_t \left[H_{|t} \neq H_{|\rhd t} \Rightarrow \exists_m \left[\langle t, m \rangle \in E \right] \right] \end{array}$$

□

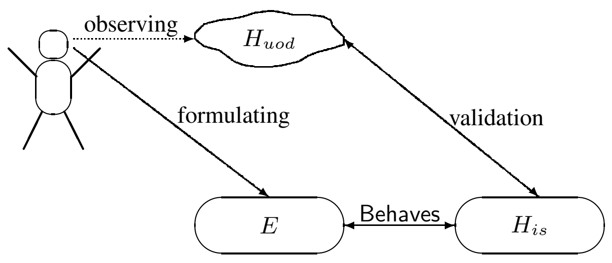


Figure 10: A user observing the history of a universe of discourse

The first part of the above definition states that every event occurence must be reflected in the application model history H. On the other hand, the second part of the definition states that any change in the H must be based on some event occurence.

The events which are described in our running example are:

1. event E_1 occurring at time t_1: the introduction of CD's
2. event E_2 occurring at time t_2: the abolishment of LP's

For simplicity, we assume that no other events (including changes to the population) have taken place. If we refer to application model history of this example by the name $Store$, then the following three different states can be recognized:

1. $Store_{|t_1}$: the initial history of the system, until CD's are introduced.
2. $Store_{|t_2}$: the history of the system after the introduction of CD's, upto the abolishment of LP's (at time t_2).
3. $Store_{|t_3} = Store_{|\rhd t_2}$ for points of time later than $\rhd t_2$.

Then the predicate Behaves enforces the following properties:

1. $Store_{|t_1} \; [\![E_1]\!]_{t_1} \; Store_{|t_2}$
2. $Store_{|t_2} \; [\![E_2]\!]_{t_2} \; Store_{|t_3}$

Due to this property, the communication between user and information system can be transaction oriented. The description of a (convenient) language for this communication falls outside the scope of this paper. For more details see [KBC$^+$89] [BMO$^+$89] [TS92] [Pro94], and [PW95].

At this point, we have demarcated the states and transitions of an evolving information system. Later, we will impose wellformedness restrictions on application model histories, and thus on the states of the evolving information system. We will use IsAMH(H) to denote that H satisfies these restrictions. These restrictions on states imply a restriction on transitions, expressed by the predicate IsEIS:

Definition 2.5
Let $E \in \mathcal{EO}$ and $H \in \mathcal{AMH}$, then:

$$\begin{array}{rcl} \mathsf{IsEIS}(E,H) & \triangleq & \mathsf{Behaves}(E,H) \\ & \wedge & \forall_{t \in \mathcal{T}} \left[\mathsf{IsAMH}(H_t)\right] \end{array}$$

□

2.6 Dissecting application model histories

The current version (snapshot) of an application model is constituted by the current versions of all application model elements. This allows us to reconstruct the sequence of successive versions of the application model. This reconstruction will be described in section 6.

An application model version contains, as its two main components, an information structure version and an action model version (see figure 2). The information structure and action model are assumed to be described in some modelling technique. The only assumption we make on this modelling technique, is that it spans an application model universe, and an information structure universe in particular. These universes provide the state space for the evolution of application models and information structures. In the following two sections, the information structure universe and application model universe are defined respectively. As stated before, in a later section some well-formedness rules are given, limiting the freedom of changes that can be performed upon application model versions in the course of time.

3 Generalised Information Structures

The kernel of the application model universe is formed by the *information structure universe*, fixing the evolution space for information structures. Therefore, we take this universe as a starting point to build the formal framework, as it forms a solid (time and application independent) base for this framework.

3.1 The information structure universe

The information structure universe, for a given modelling technique, is defined as:

Definition 3.1

The universe $\mathcal{U}_{\mathcal{IS}}$ for information structures is determined by the structure:

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

□

where $\mathcal{L}$ are label object types, $\mathcal{N}$ are abstract object types. The relation $\sim$ captures relatedness between object types. Inheritance of identification of object types is described in the relation $\rightsquigarrow$. Finally, the predicate IsSch (is schema) embodies wellformedness of information structures. These components are discussed in more detail in the next subsections.

Further refinements of the information structure universe depend on the chosen data modelling technique (such as NIAM, ER, PSM and Object Oriented data models), and are beyond the scope of the theory. An important refinement of any concrete data modelling technique will be the recognition of relationship types between object types. For instance, the fact that in figure 5 Recording is a relationship type between CD and Song, is of no importance to the general theory. Abstracting from such relationships between object types was essential because this is exactly where most modelling techniques (in particular OO from ER or NIAM) differ the most.

In subsubsection 3.1.5 we will see how ER fits within this framework. For more examples, see [Pro94] and [PW94]. For our purposes, an information structure universe is assumed to provide (at least) the above components, which are available in all conventional high level data modelling techniques.

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 who's 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 $\mathcal{N}$. The set of all possible object types is defined as: $\mathcal{O} = \mathcal{L} \cup \mathcal{N}$. The example of figure 3 contains nine object types: three entity types Record, Song and Frequency, two relation types Recording and Lending-frequency, and four label types Title, Artist, Author and Times.

3.1.2 Type relatedness

The relation $\sim\ \subseteq \mathcal{O} \times \mathcal{O}$ expresses *type relatedness* between object types (see [HW93]).

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 ([WMP$^+$76]). The relation of type relatedness can be recognised in conventional modelling techniques like ER, NIAM, or PSM, as well as in semantic data model approaches including object oriented concepts (see for example [CW85]). Typically, subtyping and generalisation lead to type related object types. For the data model depicted in figure 3, 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 in figure 11. In this example, the administration of a broker for both boats and houses is modelled. 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 specifiers ([AH87],[HW93]). The type relatedness relation for the data model of figure 11 is therefore:

1. Product $\sim$ Boat
2. Product $\sim$ House
3. Product $\sim$ Real estate
4. Real estate $\sim$ Boat
5. Real estate $\sim$ House

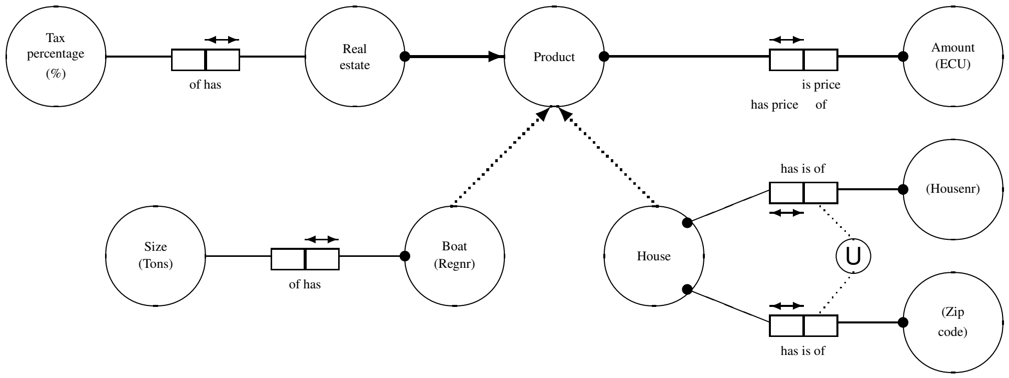


Figure 11: A data model with generalisation and specialisation

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

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

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

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 however, 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 figure 11 this means that instances of Real estate are identified in the same way as instances of Product. The identification of instances from Product depends on the identification of instances from Boat or House (note that an instance from Product is either an instance from Boat or an instance from House). For the data model depicted in figure 4, it means that instances of LP and CD are identified in the same way as instances of Medium.

An object type from which the identification is inherited, is termed an *ancestor* of that object type. The inheritance hierarchy (identification hierarchy) is provided by the relation $x \rightsquigarrow y$, meaning x is an ancestor of y. For figure 4 this leads to: $\mathsf{Medium} \rightsquigarrow \mathsf{LP}$ and $\mathsf{Medium} \rightsquigarrow \mathsf{CD}$. The inheritance relation is both transitive and irreflexive.

[ISU3] (*transitive*) $x \rightsquigarrow y \wedge y \rightsquigarrow z \Rightarrow x \rightsquigarrow z$

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

Similar axioms can be found as properties in literature about typing theory for databases ([BW90], [Oho90] and [CW85]). 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 an ancestor, are called *roots*: $\mathsf{IsRoot}(x) \triangleq \neg \exists_z \left[z \rightsquigarrow x\right]$. The roots x of an object type y are found by:

$$x \, \mathsf{RootOf} \, y \triangleq (x = y \vee x \rightsquigarrow y) \wedge \mathsf{IsRoot}(x)$$

The finite depth of the inheritance hierarchy is expressed by the following schema of induction:

[ISU5] (*ancestor induction*) If $\forall_{x \rightsquigarrow y}[F(x)] \Rightarrow F(y)$ for any y, then $\forall_x[F(x)]$.

From the intuition behind the ancestor relation it follows that object types may have instances in common with their ancestors. This implies that object types not only inherit identification from their ancestors, but type relatedness as well. These requirements are laid down in the following axioms:

[ISU6] (*inheritance of type relatedness*)

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

[ISU7] (*foundation of type relatedness*)

$$x \sim y \wedge \neg \, \mathsf{IsRoot}(y) \Rightarrow \exists_z [x \sim z \wedge z \rightsquigarrow y]$$

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 instance the root relation for figure 3 is: $x \, \mathsf{RootOf} \, x$ for every object type x. When the data model at hand contains specialisation or generalisations, the relations $\rightsquigarrow$ and RootOf will be less trivial. In figure 12, the relation RootOf of the data model in figure 11 is provided. This graph is a so called *root dependency graph*, as it depicts the dependency of object types on their roots.

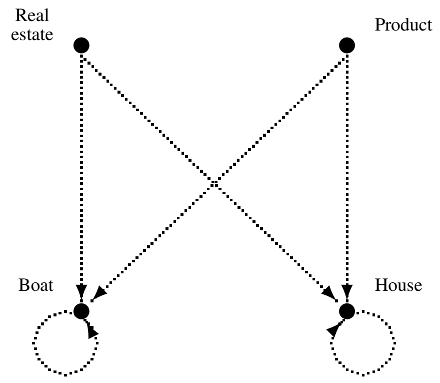


Figure 12: Root dependency graph

3.1.4 Correctness of information structures

An information structure is spanned by a set of object types. Not all sets of object types taken from $\mathcal{O}$ will correspond to a correct information structure. Therefore, a technique dependent predicate $\mathsf{IsSch} \subseteq \wp(\mathcal{O})$ has to be supplied, designating which sets of object types form a correct information structure.

3.1.5 An example: ER

As a brief example of how the general theory can be related to an existing modelling technique, we consider ER in this section. As stated before, a fully elaborated and formalised application of the theory to an object-role modelling technique can be found in [PW94].

For Chen's ([Che76]) ER model (extended with subtyping), the information structure universe will be:

Label Types The set of label types $\mathcal{L}$ 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 $\mathcal{N}$ 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: $x \sim y \triangleq \sqcap(x) = \sqcap(y)$.

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 and ISU2 are immediate consequences of the above definition. The identification hierarchy axioms ISU3, ISU4 and ISU5 directly follow from the nature of subtyping in ER. The axioms that relate type relatedness with the identification hierarchy are also easily verified.

3.2 Properties of information structure universes

The axioms so far try to model the concepts of type relatedness, object type and inheritance. In this section, we derive some usefull properties of information structure universes, illustrating the validity of the ISU axioms at the same time. The first property relates the root relationship to type relatedness:

Lemma 3.1 Any root of an object type is related to that object type:

$$x \,\mathsf{RootOf}\, y \Rightarrow x \sim y$$

Proof:

$x \,\mathsf{RootOf}\, y$
  $\Rrightarrow$ {*definition of* RootOf}[3]
$x \rightsquigarrow y$
  $\Rrightarrow$ {*definition of* $\sim$}
$x \sim x \wedge x \rightsquigarrow y$
  $\Rrightarrow$ {*axiom* ISU6}
$x \sim y$ □

Axiom ISU7 may be generalized to:

Lemma 3.2 (*common roots*) Sharing a root is equivalent with being type related:

$$x \sim y \iff \exists_z [x \sim z \wedge z \,\mathsf{RootOf}\, y]$$

In order to prove this property, and interesting properties to come, two proof schemas concerning *inheritance* and *foundation* of properties are introduced first. We call a property P of object types a *strong inheritance property*, iff for all x, y:

$$P(x) \wedge x \rightsquigarrow y \Rightarrow P(y)$$

Note that axiom ISU6 states that the relation P_x, defined by $P_x(y) = x \sim y$, is a strong inheritance property for all x. A property P will be referred to as a *weak inheritance property* iff, for all y:

$$P(y) \wedge \neg \,\mathsf{IsRoot}(y) \Rightarrow \exists_x [P(x) \wedge x \rightsquigarrow y]$$

Axiom ISU7 states that the relation P_x, defined by $P_x(y) = x \sim y$, is a weak inheritance property for all x. The first proof schema is rather straightforward, and is concerned with inheritance of properties:

Theorem 3.1 (*inheritance schema*) If P is a strong inheritance property, then the property is preserved by the RootOf relation:

$$P(x) \wedge x \,\mathsf{RootOf}\, y \Rightarrow P(y)$$

Proof:

Suppose P is a strong inheritance property, then:

$P(x) \wedge x \,\mathsf{RootOf}\, y$
  $\Rrightarrow$ {*definition of* RootOf}
$P(x) \wedge (x \rightsquigarrow y \vee x = y)$
  $\Rrightarrow$ {*P is a strong inheritance property*}
$P(y)$ □

[3]In proofs, the $\Rrightarrow$ symbol stands for a logical deduction step. A logical equivalence in a proof is denoted by $\equiv$. The motivation for the deduction step is provided in parenthesis.

The second proof schema is concerned with the foundation of properties:

Theorem 3.2 (*foundation schema*) If P is a weak inheritance property, then P originates from root object types.

$$P(y) \Rightarrow \exists_x [P(x) \land x \,\mathsf{RootOf}\, y]$$

Proof:

Suppose P is a weak inheritance property. Now we apply ancestor induction in order to prove the theorem. From the induction hypothesis it follows:

$$\forall_{v \rightsquigarrow w} [P(v) \Rightarrow \exists_z [P(z) \land z \,\mathsf{RootOf}\, v]]$$

From this we will prove:

$$P(w) \Rightarrow \exists_z [P(z) \land z \,\mathsf{RootOf}\, w]$$

Suppose $P(w)$. The $\mathsf{IsRoot}(w)$ case is obvious, due to the reflexivity of the RootOf relation. For $\neg\,\mathsf{IsRoot}(w)$ we get:

$P(w) \land \neg\,\mathsf{IsRoot}(w)$
$\quad \Rrightarrow \{$*P is a weak inheritance property*$\}$

$\exists_u [u \rightsquigarrow w \land P(u)]$
$\quad \Rrightarrow \{$*induction hypothesis*$\}$

$\exists_u [(P(u) \Rightarrow \exists_z [P(z) \land z \,\mathsf{RootOf}\, u]) \land u \rightsquigarrow w \land P(u)]$
$\quad \Rrightarrow \{$*elaborate*$\}$

$\exists_u [\exists_z [P(z) \land z \,\mathsf{RootOf}\, u] \land u \rightsquigarrow w]$
$\quad \Rrightarrow \{$*definition of* $\mathsf{RootOf}\}$

$\exists_z [P(z) \land z \,\mathsf{RootOf}\, w]$ □

With these proof schemata we get an elegant proof of lemma 3.2:

Proof:

We prove both directions separately:

$\Longleftarrow$ Define $P(a) = x \sim a$. We will apply this to axiom ISU6:

$P(y) \land y \rightsquigarrow z \Rightarrow P(z)$
$\quad \Rrightarrow \{$*P is a strong inheritance property*$\}$

$P(y) \land y \,\mathsf{RootOf}\, z \Rightarrow P(z)$
$\quad \Rrightarrow \{$*elaborate*$\}$

$\exists_y [P(y) \land y \,\mathsf{RootOf}\, z] \Rightarrow P(z)$
$\quad \Rrightarrow \{$*definition of P*$\}$

$\exists_y [x \sim y \land y \,\mathsf{RootOf}\, z] \Rightarrow x \sim z$

$\Longrightarrow$ Define $P(a) = x \sim a$. We will apply this to axiom ISU7.

$P(y) \land \neg\,\mathsf{IsRoot}(z) \Rightarrow \exists_z [P(z) \land z \rightsquigarrow y]$
$\quad \Rrightarrow \{$*P is a weak inheritance property*$\}$

$P(y) \Rightarrow \exists_z [P(z) \land z \,\mathsf{RootOf}\, y]$
$\quad \Rrightarrow \{$*definition of P*$\}$

$x \sim y \Rightarrow \exists_z [x \sim z \land z \,\mathsf{RootOf}\, y]$ □

The result of the above lemma can be generalised to the following theorem:

Theorem 3.3 (*type relatedness propagation*) Type relatedness of roots is equivalent with type relatedness of object types:

$$\exists_{z_1 \sim z_2}[z_1 \; \mathsf{RootOf} \; x \wedge z_2 \; \mathsf{RootOf} \; y] \iff x \sim y$$

Proof:

$\exists_{z_1,z_2}[z_1 \sim z_2 \wedge z_1 \; \mathsf{RootOf} \; x \wedge z_2 \; \mathsf{RootOf} \; y]$
$\quad\equiv$ {*apply lemma 3.2 (page 16) to* z_1}
$\exists_{z_2}[x \sim z_2 \wedge z_2 \; \mathsf{RootOf} \; y]$
$\quad\equiv$ {*apply lemma 3.2 (page 16) to* z_2}
$x \sim y$ □

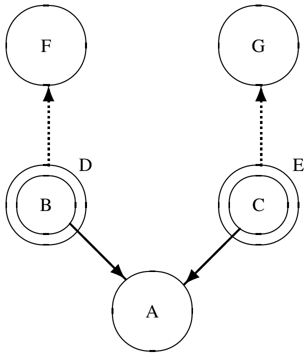


Figure 13: Data model with propagation of type relatedness

As an illustration of this theorem, consider the PSM data model from figure 13. It contains two generalisations, two specialisations, and two power types (D, E). Power types are the data modelling pendant of powersets used in set theory. The instances of object types D and E are sets of instances of B and C respectively. The RootOf relation for this data model, is given in figure 14. The type relatedness of D and E, which itself follows from the type relatedness of B and C ([HW93]), is propagated to F and G by means of the RootOf relationship and theorem 3.3. In [HW93], [HPW93], the inheritance of type relatedness via type constructions, e.g. powertyping, is elaborated.

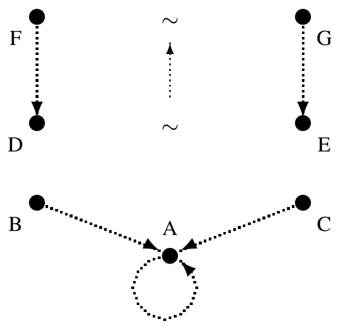


Figure 14: Root dependency graph showing propagation of type relatedness

4 Generalised Application Models

Besides the information structure, an application model contains a number of other elements. The hierarchy of models in figure 2 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 crucial elements of an application model, as they form the fabric of the 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 4.1

An application model universe is spanned by the tuple:

$$\mathcal{U}_\Sigma = \langle \mathcal{U}_{\mathcal{IS}}, \mathcal{D}, \Omega, \mathsf{IsPop}, \gamma, \mu, [\![\ ]\!], \mathsf{Depends} \rangle$$

□

where the information structure universe $\mathcal{U}_{\mathcal{IS}}$ has been introduced in the previous section. $\mathcal{D}$ is a set of underlying concrete domains to be associated to label types. The set Ω 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. γ and μ are the universes for constraint and method definitions respectively. The semantics of both constraints and methods is provided by the ternary predicate $[\![\ ]\!]$ (see subsection 2.4). The dependencies of constraints and method on the type level ($\mathcal{O}$, $\mathcal{L} \times \mathcal{D}$) are described by the relation $\mathsf{Depends}$. The information structure universe $\mathcal{U}_{\mathcal{IS}}$ was introduced in the previous section. The other components of the application model universe are discussed in the remainder of this subsection.

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 $\mathcal{D}$ ([HPW93]). Therefore, label types in an information structure version will have to be related to domains. An application model version contains a mapping Dom_t providing the relation between label types and domains. Each domain assignment Dom_t is bound to:

$$\mathsf{Dom} = \mathcal{L} \rightarrowtail \mathcal{D}$$

Some illustrative examples of such domain assignments, in the context of the rental store running example, are: $\mathsf{Times} \mapsto \mathsf{Natno}$, $\mathsf{Title} \mapsto \mathsf{String}$, where Natno and String are assumed to be (names of) concrete domains.

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 thing, which can evolve by itself. Therefore, (non empty) sets of object types are associated to instances, specifying the object types having this instance as an instantiation. This association is the intuition behind the relation $\mathsf{HasTypes}_t$. The domain for this relation is:

$$\mathsf{HasTypes} = \Omega \times (\wp(\mathcal{O}) - \{\varnothing\})$$

where Ω is the set of all possible instantiations of object types. Note that $\mathsf{HasTypes}_t$ 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 D and E in figure 13. Then $\{a_1, a_2\}\ \mathsf{HasTypes}_t\ \{D, F\}$ and $\{a_1, a_2\}\ \mathsf{HasTypes}_t\ \{E, G\}$.

Another example of such an association is $\langle l_1, \{\mathsf{Medium}, \mathsf{Lp}\}\rangle$, meaning l_1 is an (abstract) instance of entity types Medium and Lp. The population of an object type, traditionally provided as a function $\mathsf{Pop} : \mathcal{O} \rightarrow \wp(\Omega)$, can be derived from the association between instances and object types: $\mathsf{Pop}_t(x) = \{v \mid v\ \mathsf{HasTypes}_t\ Y \wedge x \in Y\}$.

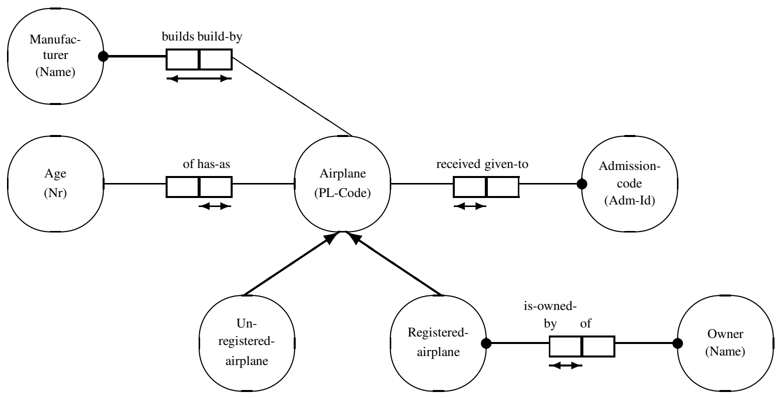
Not all subsets of HasTypes will correspond to a proper population. A population of an information structure will have to adhere to some technique dependent properties. These properties are assumed to be provided by the predicate $\mathsf{IsPop} \subseteq \wp(\mathcal{O}) \times \wp(\mathsf{HasTypes})$. Note that this predicate does not take the validity of constraints in the application model into consideration. This is not yet possible, as constraints may be transition oriented, implying that they can only be enforced in the context of the evolution of the elements. The enforcing of constraints on the (evolution of) populations will therefore be postponed until section 6.

4.3 Constraints

Most data modelling techniques offer a language for expressing constraints, both state and transition oriented. This language describes a set γ of all possible constraint definitions.

Each constraints C is treated as a partial function, assigning constraint definitions to object types: $C : \mathcal{O} \rightarrowtail \gamma$. Constraint C is said to be *owned* by object type x, if x has assigned a constraint definition by constraint C. Each constraint is considered to be an application model element.

Constraints are inherited via the identification hierarchy. However, as in object oriented data modelling techniques, overriding (strengthening) of constraint definition in identification hierarchies is possible (see for instance [De 91]). This will be discussed in more detail in a later section as axiom AMV12.



Registered-airplane	=	Airplane received Admission-code
Unregistered-airplane	=	Airplane BUT-NOT Registered-airplane

Figure 15: Constraint assignment

As an illustration of the assignment of constraints to object types, consider figure 15. The depicted data model is conforming to NIAM, while the subtype defining rules have been formulated in LISA-D. The modelled universe of discourse is concerned with the administration of airplanes. As airplanes should be replaced in time, the age of an airplane is an important attribute. Furthermore, an airplane may be registered by an aviation association, in which case it has associated an admission code. The owner of registered planes is maintained by the administration.

The graphical constraints contained in this data model, are assigned to object types in the following way (conforming to the style of Elisa-D [HPW93]):

$$
\begin{array}{lll}
C_1 : & \mathsf{Manufacturer} & \mapsto \; \mathsf{TOTAL}\ \{\ \mathsf{Manufacturer.builds}\ \} \\
C_2 : & \mathsf{Airplane} & \mapsto \; \mathsf{UNIQUE}\ \{\ \mathsf{Airplane.has\text{-}as}\ \} \\
C_3 : & \mathsf{Admission\text{-}code} & \mapsto \; \mathsf{TOTAL}\ \{\ \mathsf{Admission\text{-}code.given\text{-}to}\ \}
\end{array}
$$

In this example, the expression TOTAL {Manufacturer.builds} denotes the requirement that each Manufacturer plays the role builds. The expression UNIQUE {Airplane.has-as} requires uniqueness of playing the role has-as. All these constraints are owned by a single object type. A more interesting case with respect to inheritance results by adding the following constraint:

All airplanes must have associated a manufacturer or an age. Unregistered airplanes must have both.

The object type assignment for this constraint is:

$$
\begin{aligned}
C_4 :\ & \textsf{Airplane} \\
& \mapsto \\
& \textsf{TOTAL } \{ \textsf{ Airplane.build-by, Airplaine.has-as } \} \\
C_4 :\ & \textsf{Registered-airplane} \\
& \mapsto \\
& \textsf{TOTAL } \{ \textsf{ Airplane.build-by, Airplaine.has-as } \} \\
C_4 :\ & \textsf{Unregistered-airplane} \\
& \mapsto \\
& \textsf{TOTAL } \{ \textsf{ Airplane.build-by } \} \textsf{ AND TOTAL } \{ \textsf{ Airplaine.has-as } \}
\end{aligned}
$$

The constraint C_4 is owned by object types Airplane, Registered-airplane and Unregistered-airplane. Finally, as an illustration of a transition oriented constraint, consider the following constraint for airplanes:

$$
\begin{aligned}
C_5 :\ & \textsf{Airplane} \\
& \mapsto \\
& \textsf{(Unregistered-airplane BEFORE Registered-airplane) EQUALS} \\
& \quad \textsf{Airplane} \\
C_5 :\ & \textsf{Registered-airplane} \\
& \mapsto \\
& \textsf{(Unregistered-airplane BEFORE Registered-airplane) EQUALS} \\
& \quad \textsf{Airplane} \\
C_5 :\ & \textsf{Unregistered-airplane} \\
& \mapsto \\
& \textsf{(Unregistered-airplane BEFORE Registered-airplane) EQUALS} \\
& \quad \textsf{Airplane}
\end{aligned}
$$

stating that every airplane must have been unregistered before being a registered airplane. The expression x BEFORE y selects those instances of x, which came into existance as instance of x, *before* they became an instance of y. The expression x EQUALS y requires the outcome of x to be equal to the outcome of y.

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:

$$\mathcal{R} = \mathcal{O} \rightarrowtail \gamma$$

Enforcing constraints on a population is discussed in the next section.

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 (μ) is determined by the chosen modelling technique for the action model.

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 preleminary definition of the living space of an evolving information system as provided in subsection 2.4.

4.5 Semantics of constraints and methods

The semantics of both methods and constraints are defined by the relation $[\![\ ]\!]$. Therefore, we consider constraints as special methods, as in [Lam91]. This leads to the following axiom:

[AMU1] $\gamma \subseteq \mu$

A direct result of this axiom is: $\mathcal{R} \subseteq \mathcal{M}$. Next, we focus at the semantics of methods, which are described by $[\![\ ]\!]$ as transitions on application model histories. Methods are required to preserve the wellformedness properties specified by IsAMH.

[AMU2] $H\, [\![m]\!]_t\, H' \Rightarrow (\mathsf{IsAMH}(H) \Rightarrow \mathsf{IsAMH}(H'))$

The meaning of a method may depend on the history sofar of an application model. It may, however, not depend on any future behaviour of the application model:

[AMU3] $H\, [\![m]\!]_t\, H' \Rightarrow H = H_{|t}$

Furthermore, the effect of a method is completely known after its completion:

[AMU4] $H\, [\![m]\!]_t\, H' \Rightarrow H' = H'_{|\triangleright t}$

The history of an application model is supposed to be monotoneous. So it is not possible to falsify (correct) the history.

[AMU5] $H\, [\![m]\!]_t\, H' \Rightarrow H_{|t} = H'_{|t}$

Constraints are deemed as a special kind of method, behaving like a guard on application model histories. As a result, constraints are basically predicates. The semantics of constraints are not influenced by the next state:

[AMU6] If $c \in \mathcal{R}$ then

$$H\, [\![c]\!]_t\, H_1 \iff H\, [\![c]\!]_t\, H_2$$

This axiom implies that $H\, [\![c]\!]_t$ is a meaningfull expression.

4.6 Evolution dependency

Methods (and constraints) are usually defined by some syntactic mechanism (language). For example, for figure 15 the specification language LISA-D is used to express non-graphical constraints. The graphical constraints in figure 15 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:

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

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

The interpretation of this relation is as follows: $x\, \mathsf{Depends}\, y$ 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 second action specification from the rental store example:

```
ACTION Init-freq =
   WHEN ADD Medium:$x$ DO
     IF Lp:$x$ THEN
       ADD Lp:$x$ has Lending-frequency of Frequency:$0$
```

This action specification depends on object types Medium, Lp and Frequency. It, furthermore, depends on the domain assignment: Frequency $\mapsto$ 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. This will be formalized in a later section as axiom AMV11.

5 Application Model Versions

In this section, the formal definition of an application model version is provided, containing all components from the hierarchy of models, and the relations among them. First, we give a delimitation of the state space of the application model versions by means of an application model universe.

5.1 Deriving 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 actual application model element versions. At this moment we will identify the domain for such versions:

Definition 5.1

An application model version over application model universe $\mathcal{U}_\Sigma$ is defined as:

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

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

From a version of an application model, 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:

$$\begin{array}{rcl} \mathcal{L}_t & = & \mathcal{O}_t \cap \mathcal{L} \\ \mathcal{N}_t & = & \mathcal{O}_t \cap \mathcal{N} \\ x \sim_t y & \triangleq & x \sim y \wedge x, y \in \mathcal{O}_t \\ x \rightsquigarrow_t y & \triangleq & x \rightsquigarrow y \wedge x, y \in \mathcal{O}_t \end{array}$$

As an overview of the components of an application model version, a meta model is provided in figure 16. This (meta) data model is conforming to the PSM data modelling technique, an extension of the NIAM modelling technique. The object types $\mathcal{R}_t$ and $\mathcal{M}_t$ in figure 16 are power types, the data modelling pendant of power sets in set theory.

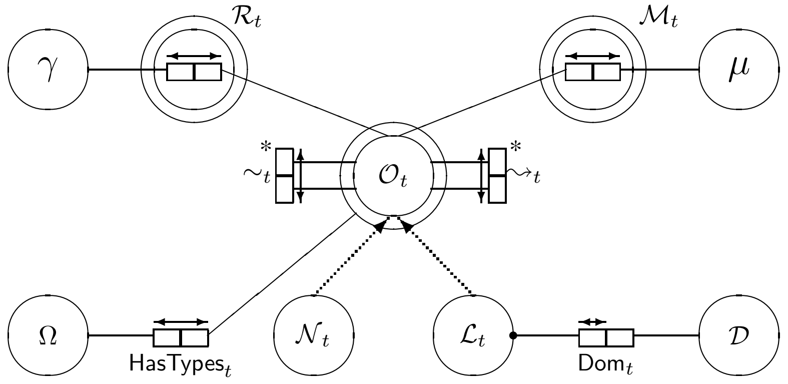


Figure 16: A meta model for information structures

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 can be stated.

5.1.1 Active and living objects

An object type x is called *alive* at a certain point of time t, if it is part of the application model version at that point of time ($x \in \mathcal{O}_t$). Furthermore, an object type x is termed *active* at a certain point of time t, if it is instantiated at that moment, i.e., if there is an *instance typing X at time t* such that $x \in X$. We call X an instance typing at time t if $\exists_{v,t} \left[v \; \mathsf{HasTypes}_t \; X \right]$. In the remainder of this subsection, a number a rules for instance typings will follow.

A first rule of wellformedness states that every active object type must be alive as well. This rule can be popularised as: 'I am active, therefore I am alive'. It is formalised as:

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

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

The next rule of wellformedness 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:

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

We call X an instance typing, if X is an anstance typing at some point of time t. In a later section we will prove a stronger version of this axiom. 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 is also an instance of its ancestors (if any):

[AMV3] (*foundation of activity*)
If X is an instance typing, then the relation P, defined by $P(x) = x \in X$, is a weak inheritance property.

Applying the foundation schema (theorem 3.2) to this axiom shows the presence of roots in instance typings:

Lemma 5.1 (*active roots*) If X is an instance typing, then:

$$y \in X \Rightarrow \exists_x \left[x \in X \wedge x \; \mathsf{RootOf} \; y \right]$$

In most traditional data modelling techniques (ER, NIAM, ...) 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 figure 13). 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:

$$\mathsf{IsRoot}(x) \wedge \mathsf{IsRoot}(y) \Rightarrow x = y$$

Some modelling techniques allow for multiple-rooted type hierarchies. As an example, consider figure 11. The instances of object types, however, are single-rooted. For example, we could have the following instances:

1. $\left\langle h_1, \left\{ \mathsf{House}, \mathsf{Product}, \mathsf{Real\ estate} \right\} \right\rangle$,
2. $\left\langle b_1, \left\{ \mathsf{Boat}, \mathsf{Product} \right\} \right\rangle$

3. $\langle b_2, \{\textsf{Boat}, \textsf{Product}, \textsf{Real estate}\}\rangle$

The instance $\langle w_1, \{\textsf{House}, \textsf{Boat}, \textsf{Product}, \textsf{Real estate}\}\rangle$ is not a valid one, as both House and Boat are root object types. In this case, the value w_1 would inherit its identification from these object types, and would therefore be identified by both an adress and boat registration number, which is a contradiction. The above axiom is intended to exclude such instances.

The above axiom leads to the following strengthening of lemma 5.1:

Lemma 5.2 (*active root*) If X is an instance typing, then:

$$y \in X \Rightarrow \exists!_x [x \in X \wedge x \,\textsf{RootOf}\, y]$$

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*) The relation P, defined by $P(x) = x \in \mathcal{O}_t$, is a weak inheritance property.

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 the foundation schema on axiom AMV5 we get:

Lemma 5.3 (*living roots*)

$$y \in \mathcal{O}_t \Rightarrow \exists_x [x \in \mathcal{O}_t \wedge x \,\textsf{RootOf}\, y]$$

Note that in this case the root x does not have to be unique.

5.1.2 Wellformed concretisation

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

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

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

[AMV7] (*strong typing of labels*) If $v \,\textsf{HasTypes}_t\, X$ and $v \in \bigcup \mathcal{D}$ then:

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

5.1.3 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.

[AMV8] (*alive definitions*) If $w \in \mathcal{R}_t \cup \mathcal{M}_t$ then:

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

where $\textsf{dom}(w) = \{x \mid \langle x, y\rangle \in w\}$ is the domain of function w. For example, constraint C_1 from the airplane example can only be alive if the object type Manufacturer is alive. As a next rule, object types that own the same constraint or method, must be type related.

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

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

For example, object types Registered-airplane and Unregistered-airplane both own constraint C_4. As a result, C_4 constrains populations of both object types. This only makes sense if the object types Registered-airplane and Unregistered-airplane can have values in common, i.e., if they are type related.

Finally, due to inheritance, if a constraint is defined for an ancestor object type, it is defined for all its offspring as well. For example, if a constraint puts a limitation on airplanes, then this constraint is also effective for special kinds of airplanes such as registered airplanes.

[AMV10] (*inheritance of definitions*)
If $w \in \mathcal{R}_t \cup \mathcal{M}_t$ then the relation P, defined by $P(x) = x \in \mathsf{dom}(w)$, is a strong inheritance property.

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

The motivation for the next axiom lies in the following observation (see subsection 4.6). 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.

[AMV11] (*dangling references*)
If $w \in \mathcal{R}_t \cup \mathcal{M}_t$ then:

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

Since every instance from a non-root object type is inherited downwards in the identification hierarchy towards the root object types, constraints on child-object types should be at least as restrictive:

[AMV12] (*strengthening of constraints*) If $c \in \mathcal{R}_t$, then:

$$x \rightsquigarrow y \wedge c{\downarrow}x, y \Rightarrow c(y) \Vdash c(x)$$

where $d_1 \Vdash d_2$ is defined as:

$$d_1 \Vdash d_2 \triangleq \forall_{t,H} \left[H \left[\!\left[d_1 \right]\!\right]_t \Rightarrow H \left[\!\left[d_2 \right]\!\right]_t \right]$$

The intuitive meaning of $d_1 \Vdash d_2$ is: d_1 is at least as restrictive as d_2 (see also [Hof93]). As an illustration of this rule, consider constraint C_4 of the airplane example. For unregistered airplanes is a strengthening of the rule for airplanes. It would not make sense to be more liberal for unregistered airplanes than for airplanes in general, as each unregistered airplane is also an airplane!

5.2 Populations of information structures

A special part of an application model version is its population. This population can be derived from the relation $\mathsf{HasTypes}_t$:

Definition 5.2
The population of an information structure at any point of time, is a mapping $\mathsf{Pop} : \mathcal{T} \rightarrow (\mathcal{O} \rightarrow \wp(\Omega))$, *defined by:*

$$\mathsf{Pop}_t(x) = \left\{ v \;\middle|\; \exists_Y \left[v \,\mathsf{HasTypes}_t\, Y \wedge x \in Y \right] \right\}$$

$\square$

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 5.3

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

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

□

Axiom AMV3 relates instances to the object type hierarchy. This leads to the following property for populations:

Lemma 5.4 (*population distribution*) Every instance of an object type, is also instance of one of its roots:

$$\mathsf{Pop}_t(x) \subseteq \bigcup_{y \, \mathsf{RootOf} \, x} \mathsf{Pop}_t(y)$$

Proof:

Let $i \in \mathsf{Pop}_t(x)$ then:

$i \in \mathsf{Pop}_t(x)$
  $\equiv$ {*definition of* Pop_t}
$\exists_X \left[i \; \mathsf{HasTypes}_t \; X \wedge x \in X \right]$
  $\Rrightarrow$ {*lemma 5.1 (page 24)*}
$\exists_{y,X} \left[i \; \mathsf{HasTypes}_t \; X \wedge x, y \in X \wedge y \; \mathsf{RootOf} \; x \right]$
  $\equiv$ {*definition of* Pop_t}
$\exists_y \left[y \; \mathsf{RootOf} \; x \wedge i \in \mathsf{Pop}_t(y) \right]$
  $\equiv$ {*elaborate*}
$i \in \bigcup_{y \, \mathsf{RootOf} \, x} \mathsf{Pop}_t(y)$. □

The result of the previous lemma can be generalised to extra-temporal populations:

Corollary 5.1

$$\mathsf{Pop}_\infty(x) \subseteq \bigcup_{y \, \mathsf{RootOf} \, x} \mathsf{Pop}_\infty(y)$$

Proof:

Let $v \in \mathsf{Pop}_\infty(x)$ then:

$v \in \mathsf{Pop}_\infty(x)$
  $\equiv$ {*definition of* Pop_∞}
$\exists_t \left[v \in \mathsf{Pop}_t(x) \right]$
  $\Rrightarrow$ {*lemma 5.4 (page 27)*}
$\exists_t \left[v \in \bigcup_{y \, \mathsf{RootOf} \, x} \mathsf{Pop}_t(y) \right]$
  $\equiv$ {*definition of* Pop_∞}
$v \in \bigcup_{y \, \mathsf{RootOf} \, x} \mathsf{Pop}_\infty(y)$ □

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 5.2.

[AMV13] (*exclusive root population*) If $\mathsf{IsRoot}(x)$ and $\mathsf{IsRoot}(y)$ then:

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

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

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 5.1 (*exclusive population*) If $x \not\sim y$ then

$$\bigcup_{z \,\mathsf{RootOf}\, x} \mathsf{Pop}_\infty(z) \;\cap\; \bigcup_{z \,\mathsf{RootOf}\, y} \mathsf{Pop}_\infty(z) \;=\; \varnothing$$

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

Proof:

$x \not\sim y$
$\quad \equiv \{$*theorem 3.3 (page 18)*$\}$
$\neg\exists_{z_1,z_2} [z_1 \,\mathsf{RootOf}\, x \wedge z_2 \,\mathsf{RootOf}\, y \wedge z_1 \sim z_2]$
$\quad \Rrightarrow \{$*elaborate*$\}$
$\forall_{z_1,z_2} [z_1 \,\mathsf{RootOf}\, x \wedge z_2 \,\mathsf{RootOf}\, y \Rightarrow z_1 \not\sim z_2]$
$\quad \Rrightarrow \{$*axiom* AMV13$\}$
$\forall_{z_1,z_2} \left[\begin{array}{c} z_1 \,\mathsf{RootOf}\, x \wedge z_2 \,\mathsf{RootOf}\, y \Rightarrow \\ \mathsf{Pop}_\infty(z_1) \cap \mathsf{Pop}_\infty(z_2) = \varnothing \end{array} \right]$
$\quad \Rrightarrow \{$*elaborate*$\}$
$\bigcup_{z \,\mathsf{RootOf}\, x} \mathsf{Pop}_\infty(z) \cap \bigcup_{z \,\mathsf{RootOf}\, y} \mathsf{Pop}_\infty(z) = \varnothing$ □

From lemma 5.4 and theorem 5.1 the main typing theorem is derived:

Theorem 5.2 (*strong typing theorem*)

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

We are now in a position to define what constitutes a wellformed application model version. Let $\Sigma_t = \langle \mathcal{O}_t, \mathcal{R}_t, \mathcal{M}_t, \mathsf{HasTypes}_t, \mathsf{Dom}_t \rangle$:

$$\begin{array}{rcl} \mathsf{IsAM}(\Sigma_t) & \triangleq & \mathsf{IsSch}(\mathcal{O}_t) \wedge \mathsf{IsPop}(\mathcal{O}_t, \mathsf{HasTypes}_t) \wedge \\ & & \Sigma_t \text{ adheres to the AMV axioms} \end{array}$$

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

6 Evolution of Application Models

As stated before, the evolution of an application model is described by the evolution of its elements. The set $\mathcal{AME}$ was 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 6.1

Application model elements:

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

□

An application model element evolution was defined as a partial function, assigning actual version of application model elements to points of time. Note that the type relatedness and root relationships are defined for the evolution state space as a whole, and are therefore not subject to any evolution.

In this section we will present a set of wellformedness rules for application model histories. These rules represent our *way of thinking* with regards to a wellformed evolution, which is based on strong typing and a strict notion of identification of instances. Alternative *ways of thinking*, and corresponding wellformedness rules may be chosen. For the remainder of this section, let H be some (fixed) application model history.

6.1 Separation of element evolution

The first rule of wellformedness states that the evolution of application model elements is bound to element classes. For example, an object type may not evolve into a method, and a constraint may not evolve into an instance. The motivation behind this rule is strong typing at a theory level. Usually, strong typing leads to better structured models, while type checking provides a means for error detection. This is formalised in the following axiom:

[EW1] (*evolution separation*)

If $X \in \{\mathcal{O}, \mathcal{R}, \mathcal{M}, \mathsf{HasTypes}, \mathsf{Dom}\}$, and $h \in H$ then:

$$h(t) \in X \Rightarrow \mathsf{ran}(h) \subseteq X$$

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

From this axiom it follows that an application model history can be partitioned into the history of its object types, its constraints, its methods, its populations, and its concretisations (of label types):

Definition 6.2

object type histories:

$H_{type} = \{h \in H \mid \exists_t[h(t) \in \mathcal{O}]\}$

constraint histories:

$H_{constr} = \{c \in H \mid \exists_t[c(t) \in \mathcal{R}]\}$

method histories:

$H_{meth} = \{m \in H \mid \exists_t[m(t) \in \mathcal{M}]\}$

population histories:

$H_{pop} = \{g \in H \mid \exists_t[g(t) \in \mathsf{HasTypes}]\}$

concretisation histories:

$H_{dom} = \{d \in H \mid \exists_t[d(t) \in \mathsf{Dom}]\}$ □

In section 3, an application model version was introduced (Σ_t) as the following tuple:

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

6.2 Deriving application model versions

An application model version $\Sigma_t(H) = \langle \mathcal{O}_t, \mathcal{R}_t, \mathcal{M}_t, \mathsf{HasTypes}_t, \mathsf{Dom}_t \rangle$ 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 6.3

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:

$\mathsf{HasTypes}_t = \{g(t) \mid g \in H_{pop} \wedge g{\downarrow}t\}$

concretisations:

$\mathsf{Dom}_t = \{d(t) \mid d \in H_{dom} \wedge d{\downarrow}t\}$ □

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.

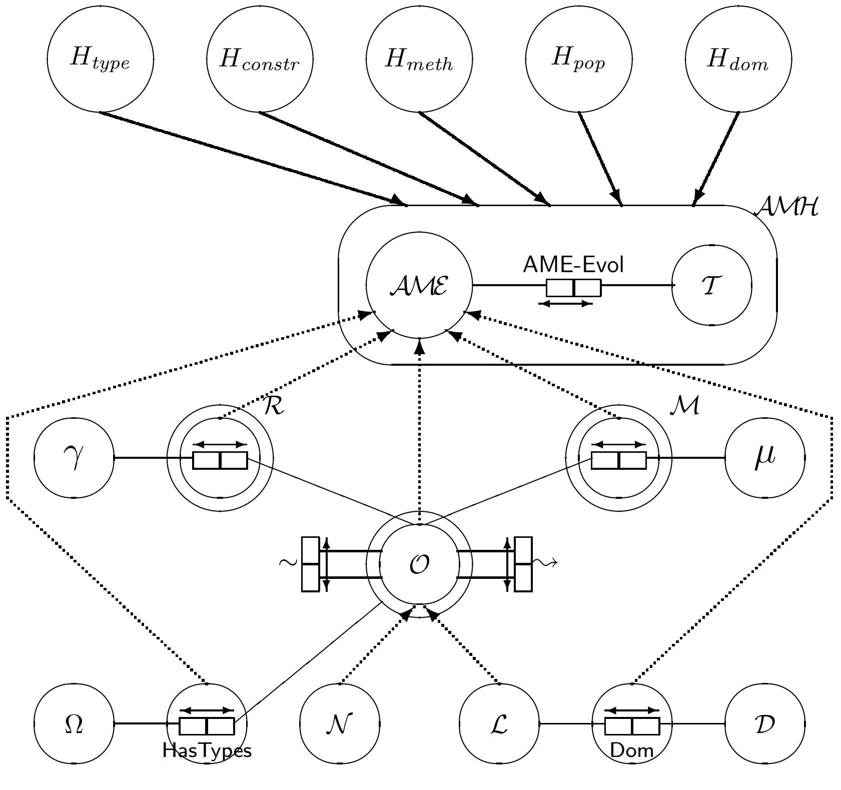


Figure 17: A meta model for the evolution system

As an outline of the hitherto defined concepts, a (meta) data model, relating all defined concepts, is provided in figure 17. The data model depicted there is conform the PSM modelling technique, and uses the notion of schema objectification (object type $\mathcal{AMH}$), and power typing (object type $\wp(\mathcal{O})$). The population of an objectified schema at hand is to be looked upon as one single abstract object instance of the object type corresponding to the objectified schema. Power-typing is, as stated before, the data modelling pendant of power-sets from set theory.

6.3 Enforcing constraints

As a next rule of well-formedness on the evolution of an application model history H, the following axiom states that all constraints must hold:

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

$$c{\downarrow}t \Rightarrow H[T] \, [\![c(t)]\!]_t$$

where T is the largest time interval such that:

$$\forall_{t' \in T}[t' \leq t \wedge c(t') = c(t)]$$

and furthermore:

$$H[T] = \{ h[T] \mid h \in H \}$$

Note that the constraint c is only enforced for the population valid during the validity of the constraint itself.

Remark 6.1

In programming, it is considered bad practice to write self modifying code*, i.e. a program that modifies itself. Analogously, one could formulate a rule stating that actions in an application model should only have effect on the population. Any structural changes (actions, information structure, . . .) should thus be performed by an action specified by the user explicitly (an update request). This can be formulated as:*

If $m \in \mathcal{M}_t$ then

$$H_t \, [\![m]\!]_t \, H_{\triangleright t} \iff H_t^{\mathsf{Pop}} \, [\![m]\!]_t \, H_{\triangleright t}^{\mathsf{Pop}}$$

The H^{Pop} is the restriction of H to the evolution of the elements of HasTypes.

□

6.4 An example of element evolution

As an example of evolution, the following table respresents three object type evolutions ($h_1, h_2, h_3 \in H_{type}$) from the rental store (see section 2):

H_{type}	h_1	h_2	h_3
t_1	o_1	o_2	$-$
t_2	o_1	o_2	$-$
t_3	o_1	o_2	$-$
t_4	o_1	o_3	o_4
t_5	o_1	o_3	o_4

where $t_1, \ldots, t_5 \in \mathcal{T}$, and $o_1, \ldots, o_4 \in \mathcal{O}$ are object types, such that:

o_1	$=$	‘Entity type: Record’
o_2	$=$	‘Fact type: Recording of Song on Record’
o_3	$=$	‘Fact type: Recording of Song on Medium’
o_4	$=$	Entity type: Medium’

Note that the evolution step (from figure 3 to figure 4) takes place at point of time t_4. Two example instance evolutions ($g_1, g_2 \in H_{pop}$), obeying the above schema evolution, are:

H_{pop}	g_1	g_2
t_1	$\langle i_1, \{o_1\}\rangle$	$\langle i_2, \{o_2\}\rangle$
t_2	$\langle i_1, \{o_1\}\rangle$	$\langle i_2, \{o_2\}\rangle$
t_3	$\langle i_1, \{o_1\}\rangle$	$\langle i_3, \{o_2\}\rangle$
t_4	$\langle i_1, \{o_1, o_4\}\rangle$	$\langle i_4, \{o_3\}\rangle$
t_5	$\langle i_1, \{o_1, o_4\}\rangle$	$\langle i_4, \{o_3\}\rangle$

where $i_1, \ldots, i_4 \in \Omega$ are instances such that:

$$\begin{aligned} i_1 &= \text{'Brothers in Arms'} \\ i_2 &= \langle\text{'Money for nothing', 'Brothers in Arms'}\rangle \\ i_3 &= \langle\text{'Brothers in Arms', 'Brothers in Arms'}\rangle \\ i_4 &= \langle\text{'Brothers in Arms', 'Brothers in Arms'}\rangle \end{aligned}$$

The interpretation of this table leads to:

$g_1(t_4) = \langle i_1, \{o_1, o_4\}\rangle$ means: 'Brothers in Arms' is both a Record and a Lp at t_4,

$g_2(t_3) = \langle i_3, \{o_2\}\rangle$ means: Song 'Brothers in Arms' is Recorded on Record 'Brothers in Arms',

$g_2(t_4) = \langle i_4, \{o_3\}\rangle$ means: Song 'Brothers in Arms' is Recorded on Medium 'Brothers in Arms'.

6.5 Evolution of the identification hierarchy

Thus far we discussed the wellformedness of the evolution of application model elements. However, as a result of object type evolution, the identification hierarchy will evolve as well. This evolution is not completely free, some conservatism with respect to such evolution is appropriate. The motivation of this approach is our tendancy to strong typing and strict object identification. In the remainder of this section, we provide some rules which exclude undesirable evolutions. It should be stressed that attacking the wellformedness problem from another vantage-point may result in other rules.

Firstly, the order in the identification hierarchy should not change in one step, since this could lead to conflicting identification schemas in the course of time:

[EW3] (*monotonous ancestors*) If $h_1, h_2 \in H_{type}$, $h_1{\downarrow}t$, $h_2{\downarrow}t$, $h_1{\downarrow}{\rhd}t$ and $h_2{\downarrow}{\rhd}t$ then

$$h_1(t) \rightsquigarrow h_2(t) \wedge h_1(\rhd t) \sim h_2(\rhd t) \Rightarrow h_1(\rhd t) \rightsquigarrow h_2(\rhd t)$$

In the CD store running example, when CD's are a special kind of Medium, the reversal of this relation in one step is excluded by this rule, as this would lead to identification problems for LP's. In the airplane example, registered airplanes are identified as airplanes in general. Suppose registered airplanes need an identification of their own. Then this is only possible after breaking the type relatedness between both object types, i.e., breaking up the identification hierarchy.

This is not only true at the type level, but also at the evolutionary level. A direct consequence of this axiom is that all ancestors of an object type have to be terminated when this object type is promoted to be a root object type:

Lemma 6.1 If $h_1, h_2 \in H_{type}$, $h_1{\downarrow}t$, $h_2{\downarrow}t$ and $h_2{\downarrow}{\rhd}t$ then $h_1(t) \rightsquigarrow h_2(t) \wedge h_1(\rhd t) \sim h_2(\rhd t) \wedge \mathsf{IsRoot}(h_2(\rhd t)) \Rightarrow \neg\, h_1{\downarrow}{\rhd}t$

Proof:

Let $h_1(t) \rightsquigarrow h_2(t) \wedge h_1(\triangleright t) \sim h_2(\triangleright t) \wedge \mathsf{IsRoot}(h_2(\triangleright t))$ and $h_1 \downarrow \triangleright t$, then:

$h_1(t) \rightsquigarrow h_2(t) \wedge h_1(\triangleright t) \sim h_2(\triangleright t) \wedge \mathsf{IsRoot}(h_2(\triangleright t)) \wedge h_1 \downarrow \triangleright t$

$\Rrightarrow$ {*axiom* EW3}

$h_1(\triangleright t) \rightsquigarrow h_2(\triangleright t) \wedge h_1(\triangleright t) \sim h_2(\triangleright t) \wedge \mathsf{IsRoot}(h_2(\triangleright t))$

$\Rrightarrow$ {*definition of* IsRoot}

falsum □

The following rule for identification hierarchy evolution states that the type-instance relation (derived from the relation HasTypes) is to be maintained in the course of evolution. Like the previous rule, the motivation of this rule is to prevent conflicting identification schemas in the course of time. This leads to the axiom of guided evolution:

[EW4] (*guided evolution*) If $g \in H_{pop}$, $g \downarrow t$ and $g \downarrow \triangleright t$ then

$$\exists_{h \in H_{type}} \left[h(t) \sim \mathsf{Types}_t(g) \Rightarrow h(\triangleright t) \sim \mathsf{Types}_{\triangleright t}(g) \right]$$

where $x \sim Y$ is defined as $\exists_{y \in Y} [x \sim y]$. The types that are associated with an instance evolution g, at point of time t, are introduced by:

$$\mathsf{Types}_t(g) \triangleq \bigcup_{X : g \ \mathsf{HasTypes}_t \ X} X$$

As an example, consider the evolution of registered airplanes to an object type with its own identification, within a separate identification hierarchy. Then it would not make any sense if the instances of this object type would not follow this evolution step, the only exception being instances that violate newly introduced constraints. This latter aspect will be elaborated further in the next subsection. Finally, we can live up to our promise of defining IsAMH formally:

Definition 6.4

$$\begin{array}{rcl} \mathsf{IsAMH}(H) & \triangleq & \forall_{t \in \mathcal{T}} \left[\mathsf{IsAM}(\Sigma_t) \right] \\ & \wedge & H \text{ adheres to the EW axioms} \end{array}$$

□

6.6 Propagating modifications

When an element of the application model evolves (is modified), other elements may have to be modified accordingly as these modifications may invalidate others or may result in conflicts. For instance, when the subtyping of object type Medium is terminated in the LP and CD store running example, all its subtypes must be terminated as well. Even more, any relationship type in which such a subtype is involved must be modified or terminated within the same transaction.

Other dependencies can be found, for example in the context of constraints. Whenever a new constraint is added, existing instances may be in conflict with this new rule, and must be adopted to meet the new requirements within the same transaction.

These dependencies are enforced on application model histories by the relations IsSch, IsPop, and Depends, which require at each point in time the population (at that moment) to be in accordance with the information structure version (at that moment). Besides, the information structure version should satisfy the wellformedness rules of the underlying data modelling technique. A detailed discussion of propagation of dependencies can only be given in the context of an application to a concrete modelling technique. When doing so, the issues concerning propagation of changes as discussed in e.g. [SZ86], [BKKK87] come into play. For more details of the propagation of dependencies in the context of some applications of the general theory to existing modelling techniques, refer to [PW94] or [Pro94].

7 Conclusions and Further research

In this paper we presented a first attempt to a general theory for the evolution of application models, supporting evolving information systems. In order to validate the theory, it must be applied to some modelling techniques.

In the mean time the theory has been applied to PSM, resulting in EVORM ([PW94], [Pro94]), and the conceptual transaction modelling technique Hydra ([HN93],[Hof93]), leading to Hydrae ([Pro94]).

Furthermore, based on the notion of evolution as laid down in the axioms of the general theory, a query and manipulation language has been defined supporting the evolution of information systems, and disclosure of information in an evolving context ([PW95], [HPW94]). Query formulation in the context of an evolving information system poses extra requirements for the query language and mechanisms used to formulate the queries, since the underlying conceptual schema evolves in the course of time, and data stored in the old schemas must be retrievable as well.

Remaining issues for further research are the implementation of an actual evolving information system, the development of an adequate modelling procedure to cope with evolution of the universe of discourse and reflect these correctly in the information system, Finally, the consequences of evolution for the internal representation of information structures should be studied in more detail.

Acknowledgements

The investigations were partly supported by the Foundation for Computer Science in the Netherlands (SION) with financial support from the Dutch Organization for Scientific Research (NWO).

We would like to thank the anonymous referees for their many valuable comments on earlier versions of this paper.

References

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[All84] J.F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 1984(23):123–154, 1984.

[Ari86] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, December 1986.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD Record*, 16(3):311–322, December 1987.

[BMO$^+$89] 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, editors, *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Frontier Series, pages 283–308. Addison-Wesley, Reading, Massachusetts, 1989.

[BW89] P. D. Bruza and Th.P. van der Weide. The semantics of data flow diagrams. In N. Prakash, editor, *Proceedings of the International Conference on Management of Data (CISMOD)*, pages 66–78, Hyderabad, India, 1989. McGraw-Hill Publishing Company.

[BW90] K.B. Bruce and P. Wegner. An algebraic model of subtype and inheritance. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, ACM Press, Frontier Series, pages 75–96. Addison-Wesley, Reading, Massachusetts, 1990.

[Che76] P.P. Chen. The entity-relationship model: Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[CR87] J. Clifford and A. Rao. A simple, general structure for Temporal Domains. In C. Rolland, F. Bodart, and M. Leonard, editors, *Temporal Aspects in information Systems*, pages 17–28. North-Holland/IFIP, Amsterdam, The Netherlands, 1987.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

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

[DHL$^+$85] 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.

[EGH$^+$92] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, 9(4):157–204, 1992.

[FOP92a] 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, editors, *Dynamic Modelling of Information Systems II*, pages 353–375. North-Holland, Amsterdam, The Netherlands, EU, 1992. ISBN 0444894055

[FOP92b] 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, editors, *Proceedings of the Data Base and Expert System Applications Conference (DEXA'92)*, pages 282–287, Valencia, Spain, EU, September 1992. Springer Verlag, Berlin, Germany, EU. ISBN 3211824006

[HM94] T.A. Halpin and R. Meersman, editors. *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*. Key Centre for Software Technology, University of Queensland, Brisbane, Australia, Magnetic Island, Australia, July 1994.

[HN93] A.H.M. ter Hofstede and E.R. Nieuwland. Task structure semantics through process algebra. *Software Engineering Journal*, 8(1):14–20, January 1993.

[Hof93] A.H.M. ter Hofstede. *Information Modelling in Data Intensive Domains*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[HPW92] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Data Modelling in Complex Application Domains. In P. Loucopoulos, editor, *Proceedings of the Fourth International Conference CAiSE'92 on Advanced Information Systems Engineering*, volume 593 of *Lecture Notes in Computer Science*, pages 364–377, Manchester, United Kingdom, EU, May 1992. Springer Verlag, Berlin, Germany, EU. ISBN 3540554815

[HPW93] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489–523, October 1993.

[HPW94] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Supporting Information Disclosure in an Evolving Environment. In D. Karagiannis, editor, *Proceedings of the 5th International Conference DEXA'94 on Database and Expert Systems Applications*, volume 856 of *Lecture Notes in Computer Science*, pages 433–444, Athens, Greece, EU, September 1994. Springer Verlag, Berlin, Germany, EU. ISBN 3540584358

[HSV89] K.M. van Hee, L.J. Somers, and M. Voorhoeve. Executable Specifications for Distributed Information Systems. In E.D. Falkenberg and P. Lindgreen, editors, *Information System Concepts: An In-depth Analysis*, pages 139–156. North-Holland/IFIP, Amsterdam, The Netherlands, 1989.

[Hul86] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3):856–886, 1986.

[HW93] A.H.M. ter Hofstede and Th.P. van der Weide. Expressiveness in conceptual data modelling. *Data & Knowledge Engineering*, 10(1):65–100, February 1993.

[ISO87] *Information processing systems – Concepts and Terminology for the Conceptual Schema and the Information Base*, 1987. ISO/TR 9007:1987.
`http://www.iso.org`

[JMSV92] M. Jarke, J. Mylopoulos, J.W. Schmidt, and Y. Vassiliou. DAIDA: An Environment for Evolving Information Systems. *ACM Transactions on Information Systems*, 20(1):1–50, January 1992.

[Kat90] R.H. Katz. Toward a Unified Framework for Version Modelling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, 1990.

[KBC$^+$89] 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, editors, *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Frontier Series, pages 251–282. Addison-Wesley, Reading, Massachusetts, 1989.

[KM90] T. Korson and J. McGregor. Understanding Object Oriented: A Unifying Paradigm. *Communications of the ACM*, 33(9):40–60, September 1990.

[Lam91] L. Lamport. The Temporal Logic of Actions. Report 79, Digital, Systems Research Center, Palo Alto, California, December 1991.

[LS87] U.W. Lipeck and G. Saake. Monitoring dynamic integrity constraints based on temporal logic. *Information Systems*, 12(3):255–269, 1987.

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, 1990.

[MS90] E. McKenzie and R. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.

[MSW92] P. McBrien, A.H. Seltviet, and B. Wangler. An Entity-Relationship Model Extended to describe Historical Information. In A.K. Majumdar and N. Prakash, editors, *Proceedings of the International Conference on Information Systems and Management of Data (CISMOD 92)*, pages 244–260, Bangalore, India, July 1992.

[NH89] G.M. Nijssen and T.A. Halpin. *Conceptual Schema and Relational Database Design: a fact oriented approach*. Prentice-Hall, Sydney, Australia, 1989. ASIN 0131672630

[Nij93] G.M. Nijssen, editor. *Proceedings of the NIAM-ISDM Conference*. NIAM-GUIDE, September 1993.

[NR89] G.T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data & Knowledge Engineering*, 4:43–67, 1989.

[Oho90] A. Ohori. Orderings and Types in Databases. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, ACM Press, Frontier Series, pages 97–116. Addison-Wesley, Reading, Massachusetts, 1990.

[OPF94] J.L.H. Oei, H.A. Proper, and E.D. Falkenberg. Evolving Information Systems: Meeting the Ever-Changing Environment. *Information Systems Journal*, 4(3):213–233, 1994.

[Pro94] H.A. Proper. *A Theory for Conceptual Modelling of Evolving Application Domains*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, EU, 1994. ISBN 909006849X

[PS87] D.J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In N. Meyrowitz, editor, *Proceedings of the ACM Conference of Object-Oriented Systems, Languages and Applications (OOPSLA)*, pages 111–117, Orlando, Florida, October 1987.

[PW93] 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.P. Papazoglou, editors, *Proceedings of the Fourth Australian Database Conference*, Advances in Database Research, pages 346–362, Brisbane, Australia, February 1993. World Scientific, Singapore. ISBN 981021331X

[PW94] H.A. Proper and Th.P. van der Weide. EVORM - A Conceptual Modelling Technique for Evolving Application Domains. *Data & Knowledge Engineering*, 12:313–359, 1994.

[PW95] H.A. Proper and Th.P. van der Weide. Information Disclosure in Evolving Information Systems: Taking a shot at a moving target. *Data & Knowledge Engineering*, 15:135–168, 1995.

[Rod91] J.F. Roddick. Dynamically changing schemas within database models. *The Australian Computer Journal*, 23(3):105–109, August 1991.

[RP92] J.F. Roddick and J.D. Patrick. Temporal semantics in information systems - A survey. *Information Systems*, 17(3):249–267, 1992.

[SA85] R. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 236–246, Austin, Texas, 1985.

[Saa88] G. Saake. *Spezifikation, Semantik und Überwachung von Objektlebensläufen in Datenbanken*. PhD thesis, Technische Universität Braunsweig, Braunsweig, Germany, 1988. (In German).

[Saa91] G. Saake. Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, 6(1):47–73, 1991.

[Sno90] R. Snodgrass. Temporal Databases Status and Research Directions. *SIGMOD Record*, 19(4):83–89, December 1990.

[SWS89] P.S. Seligmann, G.M. Wijers, and H.G. Sol. Analyzing the structure of I.S. methodologies, an alternative approach. In R. Maes, editor, *Proceedings of the First Dutch Conference on Information Systems*, Amersfoort, The Netherlands, EU, 1989.

[SZ86] A.H. Skarra and S.B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In N. Meyrowitz, editor, *Proceedings of the ACM Conference of Object-Oriented Systems, Languages and Applications (OOPSLA)*, pages 483–495, Portland, Oregon, September 1986.

[TLW91] C. Theodoulidis, P. Loucopoulos, and B. Wangler. A conceptual modelling formalism for temporal database applications. *Information Systems*, 16(4):401–416, 1991.

[Tre91] M.T. Tresch. A Framework for Schema Evolution by Meta Object Manipulation. In *Proceedings of the 3d International Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1991. Institut für Informatik, TU Clausthal.

[TS92] M.T. Tresch and M.H. Scholl. Meta Object Management and its Application to Database Evolution. In G. Pernul and A.M. Tjoa, editors, *11th International Conference on the Entity-Relationship Approach*, volume 645 of *Lecture Notes in Computer Science*, pages 299–321, Karlsruhe, Germany, October 1992. Springer-Verlag.

[WHO92] G.M. Wijers, A.H.M. ter Hofstede, and N.E. van Oosterom. Representation of Information Modelling Knowledge. In V.-P. Tahvanainen and K. Lyytinen, editors, *Next Generation CASE Tools*, volume 3 of *Studies in Computer and Communication Systems*, pages 167–223. IOS Press, 1992.

[Wij91] G.M. Wijers. *Modelling Support in Information Systems Development*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 1991. ISBN 9051701101

[WJL91] 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, editors, *Proceedings of the Third International Conference CAiSE'91 on Advanced Information Systems Engineering*, volume 498 of *Lecture Notes in Computer Science*, pages 124–140, Trondheim, Norway, May 1991. Springer-Verlag.

[WMP$^+$76] 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.

[You89] E. Yourdon. *Modern Structured Analysis*. Printice-Hall, Englewood Cliffs, New Jersey, 1989.