A General Theory for Evolving Application Models

H.A. Proper and T.P. van der Weide

Abstract—**In this article we provide a general theory for evolving information systems. This 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.**

Index Terms—**Evolving information systems, temporal information systems, schema evolution, data modelling, type relatedness, predicator set model, ER model.**

I. Introduction

As has been argued in [31] and [11], 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 [10], [24] 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 [28], an informal introduction to this theory is provided, while in [29] the fully elaborated 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 [6], EER [9], NIAM [23], and the generalized object role data modelling technique PSM [17], [14], action modelling techniques such as Task Structures [13], and furthermore object oriented modelling techniques [20]. In [30], the application of the theory presented in this article to the object-role modelling technique PSM, leading to EVORM, is described.

The assumptions underlying our theory 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 [34] a classification for incorporating time in information systems (databases) is presented. However, all these classes do not yet take schema evolution into account. For this reason, we propose a new class: evolving information systems. In [29] a more detailed discussion of the relationship to these classes of information systems is discussed.

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 assumptions on the underlying technique, and limit ourselves to conceptual issues. This paper restricts itself basically to the *way of modelling* of conceptual models. Existing approaches to evolving information systems, such as the GemStone [3], ORION [19], Sherpa [22], and Cocoon [36] systems provide 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 beneficial to have a working 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 II 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 III describes the minimal requirements for an underlying technique, as discussed above. In Section V we introduce the universe for application model evolution. After that, we discuss what constitutes a wellformed application model version. In Section VI the evolution of application models is treated, and some wellformedness rules for such evolutions are formulated.

II. 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

Manuscript received May 25, 1993; revised Jan. 21, 1994.

H.A. Proper is with the Cooperative Information Systems Research Centre, Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane, 4001 Australia; e-mail: erikp@icis.qut.edu.au. http://www.icis.qut.edu.au/~erikp.

T.P. van der Weide is with the Computing Science Institute, University of Nijmegen, Toernooiveld, NL-6525 ED Nijmegen, The Netherlands; e-mail: tvdw@cs.kun.nl.

To order reprints of this article, e-mail: transactions@computer.org, and reference IEEECS Log Number K95051.

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.

A. An Example of Evolution

As an illustration of an evolving universe of discourse, consider a rental store for audio records (LPs). In this store a registration is maintained of the songs that are recorded on the available LPs. In order to keep track of the wear and tear of LPs, the number of times an LP has been lent is registered. The information structure and constraints of this universe of discourse are modelled in Fig. 1 in the style of ER, according to the conventions of [39]. Note the special notation of attributes (`Title`) using a mark symbol (#) followed by the attribute (`#Title`).

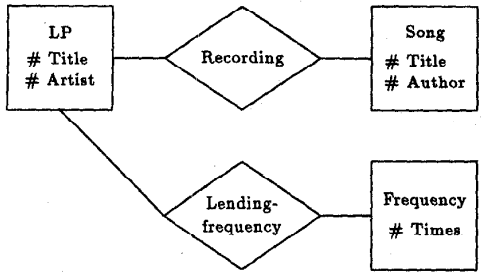


Fig. 1. The information structure of an LP rental store.

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, its 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 [15]. 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 sizable 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. The relation type Medium-type effectuates the subtyping of Medium into LP and CD. In the new situation, the registration of songs on LPs is extended to cover CDs as well. The frequency of lending, however, is not kept for CDs, as CDs are hardly subject to any wear and tear. As a consequence, the application model has evolved to Fig. 2. 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, its 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
```

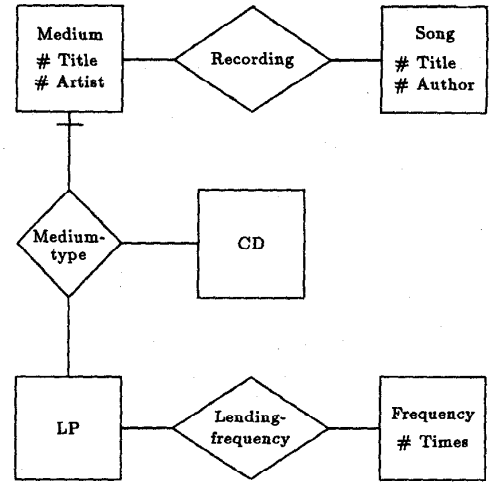


Fig. 2. The information structure of an LP and CD rental store.

After some years, the CDs have become more popular than LPs. Consequently, the rental store has decided to stop renting LPs and to become a CD rental store. Besides, the recording quality of songs on CDs has appeared to be relevant for clients. As this quality may differ from song to song on a single CD, and may for some song be different for recordings on different CDs, 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 Fig. 3. As a result of this evolution step, the action specification `Init-freq` can be terminated, since the lending frequency of CDs 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'
```

Fig. 3. The information structure of a CD rental store.

B. 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.

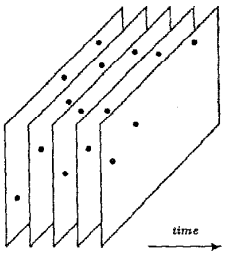


Fig. 4. Evolution modelled by snapshots.

This paper takes another approach, and treats evolution (or rather the time axis) of an application model as a separate concept. This approach has a resemblance to the approach from [33], which, however, is more restricted in the sense that is more directed towards an implementation.

Within our 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 Fig. 4. 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 Fig. 5. Each dotted line corresponds to the evolution of one distinct element.

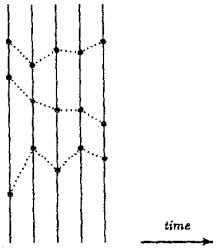


Fig. 5. Evolution modelled by functions over time.

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 Fig. 6. In the theory of evolving application models we will therefore adapt the second alternative.

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., [33], [2], and [18]. 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.

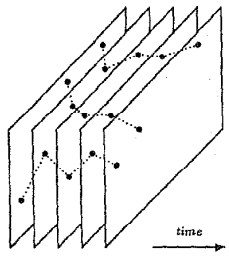


Fig. 6. Deriving snapshots from element evolutions.

C. 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*. (In this paper, the difference between recording and event time [35], and the ability to correct stored information are not taken into consideration. For more details, see [10] or [11].) 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 CDs are added to its assortment, the version of the application model element `Recording` changes from a relation type registrating songs on LPs, to a relation type registrating songs on Media. The removal of LPs from the assortment leads to the change of the application model element `Recording` into a relation type registrating songs on CDs.

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 VI.
2) Time, essential to evolution, is incorporated into the theory through the algebraic structure where $\mathcal{T}$ iss 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., [7], [37], or [1].
   The time axis is the axis along which the application model evolves. With this time axis, an application model history is a (partial) mapping $\mathcal{T} \rightarrowtail \mathcal{AME}$. In this article, $\rightarrowtail$ is used for partial functions, and $\rightarrow$ for total

functions. $\mathcal{AMH}$ 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 [32]. The linear time model is usually chosen in historical databases (see for example [34]).

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 [21].

D. A Dual Vision

The execution of an action at some point of time is referred to as an *event occurrence*.

DEFINITION 1 (*event occurrence sequence domain*). *The domain of sequences of event occurrences is identified by*: $\mathcal{EO} = \mathcal{T} \rightarrowtail \mathcal{M}$.

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:

DEFINITION 2 (*element evolution prefix*). *If* $h : T \rightarrowtail AME$, *then the prefix of h at time t is*:

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

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

DEFINITION 3 (*evolving information system state*). *If* $\mathcal{H} \in \mathcal{AMH}$ *then the state of H at time t is*:

$$H_{|t} = \{h_{|t} \mid h \in H\}.$$

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. For the state operation we have the following property:

LEMMA 1. *If H is an application model history, then*

$$u \le t \Rightarrow (H_{|t})_{|u} = H_{|u}$$

$$t \le u \Rightarrow (H_{|t})_{|u} = H_{|t}.$$

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.

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

DEFINITION 4. *Let* $E \subseteq \mathcal{EO}$ *and* $\mathcal{H} \in \mathcal{AMH}$, *then*:

$$\texttt{Behaves}(E,H) \triangleq \forall_{\langle t,m\rangle \in E}\left[H_{|t}\,[\![\,m]\!]_t\,H_{|\rhd t}\right] \wedge \forall_t\left[H_{|t} \neq H_{|\rhd t} \Rightarrow \exists_m\left[\langle t,m\rangle \in E\right]\right].$$

The first part of the above definition states that every event occurrence 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 occurrence.

The events which are described in our running example are:

1) event E_1 occurring at time t_1: the introduction of CDs
2) event E_2 occurring at time t_2: the abolishment of LPs

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
2) $Store_{|t_2}$: the history of the system after the introduction of CDs, upto the abolishment of LPs (at t_2).
3) $Store_{|t_3} = Store_{|\rhd t_2}$ for points of time later than $\rhd t_2$.

The predicate **Behaves** enforces the following properties:

$$Store_{|t_1}\,[\![E_1]\!]\,Store_{|t_1} \text{ and } Store_{|t_2}\,[\![E_2]\!]\,Store_{|t_3}.$$

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.

At this point, we have demarcated the states and transitions of an evolving information system. Later, we will impose well-formedness 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 5. *Let* $E \in \mathcal{EO}$ *and* $H \in \mathcal{AMH}$, *then*:

$$\texttt{IsEIS}(E,H) \triangleq \texttt{Behaves}(E,H) \wedge \forall_{t \in \mathcal{T}}\left[\texttt{IsAMH}(H_t)\right].$$

III. 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.

A. The information structure universe

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

DEFINITION 6. *The universe $\mathcal{U}_I$ for information structures is determined by the structure*:

$$\mathcal{U}_I = \langle \mathcal{L}, \mathcal{N}, \sim, \rightsquigarrow, \mathtt{IsSch} \rangle.$$

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. In Section III.A.5 we see how ER fits within this framework. For more examples, see [26] and [30]. 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.

A.1. ObjectTypes

The central part of an information structure is formed by its object types (referred to as object classes in object oriented approaches). Two major classes of object types are distinguished. Object types whose instances can be represented directly (denoted) on a medium (strings, natural numbers, etc) form the class of label types $\mathcal{L}$. The other object types, for instance entity types or fact (relation) types, form the class $\mathcal{N}$. The set of all possible object types is defined as: $\mathcal{O} = \mathcal{L} \cup \mathcal{N}$. The example of Fig. 1 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`.

A.2. Type Relatedness

The relation $\sim \subseteq \mathcal{O} \times \mathcal{O}$ expresses type relatedness between object types (see [17]). 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 [38]. 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, [5]). Typically, subtyping and generalisation lead to type related object types. For the data model depicted in Fig. 1, the type relatedness relation is the identity relation: $x \sim x$ for all object types x. 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) $x \sim y \Rightarrow y \sim x$.

A.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 Fig. 2, this 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 Fig. 2 this leads to: `Medium` $\rightsquigarrow$ `LP` and `Medium` $\rightsquigarrow$ `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 [4], [25], and [5]. 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 ancestors, are called *roots*:

$$\mathtt{Root}(x) \triangleq \neg \exists_z [z \rightsquigarrow x].$$

The roots x of an object type y are found by:

$$x\ \mathtt{RootOf}\ y \triangleq (x = y \vee x \rightsquigarrow y) \wedge \mathtt{Root}(x).$$

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

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 Fig. 1 is: x 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.

A.4. Correctness of Information Structures

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

A.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 [30]. For Chen's [6] ER model (extended with subtyping), the information structure universe is as shown below.

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.

Nonlabel Types. The set of nonlabel 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. Thus 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.

B. 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 2. *Any root of an object type is related to that object type*: x RootOf $y \Rightarrow x \sim y$.

Axiom ISU7 may be generalized to:

LEMMA 3. *Sharing a root is equivalent with being type related*: $x \sim y \Leftrightarrow \exists_z [x \sim z \wedge z$ 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 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: $P(y) \wedge \neg$ Root$(y) \Rightarrow \exists_x [P(x) \wedge x \sim y]$.

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 1 (*inheritance schema*). *If P is a strong inheritance property, then the property is preserved by the* RootOf *relation:* $P(x) \wedge x$ RootOf $y \Rightarrow P(y)$.

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

THEOREM 2 (*foundation schema*). *If P is a weak inheritance property, then P originates from root object types:* $P(y) \Rightarrow \exists_x [P(x) \wedge x$ RootOf $y]$.

The result of Lemma 3 can be generalised to the following theorem:

THEOREM 3 (type relatedness propagation). Type relatedness of roots is equivalent with that of object types: $\exists_{z_1 \sim z_2} [z_1$ RootOf $x \wedge z_2$ RootOf $y] \Leftrightarrow x \sim y$.

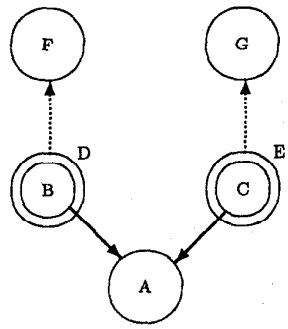


Fig. 7. Data model with propagation of type relatedness.

As an illustration of this theorem, consider the PSM data model from Fig. 7. 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 Fig. 8. The type relatedness of D and E, which itself

follows from the type relatedness of B and C [17], is propagated to F and G by means of the `RootOf` relationship and. In [17], [15], the inheritance of type relatedness via type constructions, e.g., powertyping, is elaborated.

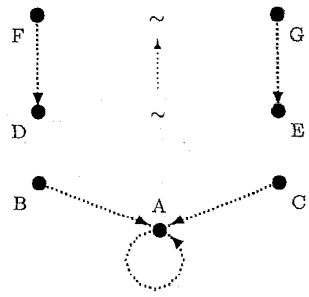


Fig. 8. Root dependency graph showing propagation of type relatedness.

IV. Generalised Application Models

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

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

where the information structure universe $\mathcal{U}_I$ *has been introduced in the previous section.* $\mathcal{D}$ *is a set of underlying concrete domains to be associated to label types. The set* 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 well-formed.* 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 Section II.C). The dependencies of constraints and method on the type level* $(O, \mathcal{L} \times \mathcal{D})$ *are described by the relation* `Depends`. *The information structure universe* $\mathcal{U}_I$ *was introduced in the previous section. The other components of the application model universe are discussed in the remainder of this subsection.*

A. Domains

The separation between concrete and abstract world is provided by the distinction between the information structure I, and the set of underlying (concrete) domains in $\mathcal{D}$ [15]. Therefore, label types in an information structure version will have to be related to domains. An application model version contains a mapping $\mathbf{Dom}_t$ providing the relation between label types and domains. Each domain assignment $\mathbf{Dom}_t$ is bound to: $\mathbf{Dom} = \mathcal{L} \rightarrowtail \mathcal{D}$. Some illustrative examples of such domain assignments, in the context of the rental store running example, are: `Times` $\mapsto$ `Natno`, `Title` $\mapsto$ `String`, where `Natno` and `String` are assumed to be (names of) concrete domains.

B. 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 $\texttt{HasTypes}_t$. The domain for this relation is: $\texttt{HasTypes} = \Omega \times (\wp(O) - \{\phi\})$ where Ω is the set of all possible instantiations of object types. Note that $\texttt{HasTypes}_t$ is a relation rather than a (partial) function. The reason is to support complex generalisation hierarchies. For example, suppose $\{a_1, a_2\}$ is an instance of both D and E in Fig. 7. Then $\{a_1, a_2\}$ is related to both $\{D, F\}$ and $\{E, G\}$ by $\texttt{HasTypes}_t$.

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

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 $\texttt{IsPop} \subseteq \wp(O) \times \wp(\texttt{HasTypes})$. 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 VI.

C. 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 constraint C is treated as a partial function, assigning constraint definitions to object types: $C : O \rightarrowtail \gamma$. 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 [8]). This is later introduced as axiom AMV12.

A constraint c, in an application model version, will be a (usually very sparse) partial function $c : O \rightarrowtail \gamma$, 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: $\mathcal{R} = O \rightarrowtail \gamma$. Enforcing con-

straints on a population is discussed in the next section.

D. 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 $m : O \rightarrowtail \mu$, assigning action specifications to object types. The set of all possible methods is the set of all these mappings: $\mathcal{M} = O \rightarrowtail \mu$. 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 Section II.C.

E. 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 [21]. This leads to the following axiom:

$$\textbf{[AMU1]}\ \gamma \subseteq \mu.$$

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`.

$$\textbf{[AMU2]}\ H\,[\![m]\!]_t\,H' \Rightarrow \big(\texttt{IsAMH}(H) \Rightarrow \texttt{IsAMH}(H')\big).$$

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:

$$\textbf{[AMU3]}\ H\,[\![m]\!]_t\,H' \Rightarrow H = H'_{\downarrow t}.$$

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

$$\textbf{[AMU4]}\ H\,[\![m]\!]_t\,H' \Rightarrow H' = H'_{\downarrow t}.$$

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

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

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:

$$\textbf{[AMU6]}\ \text{If } c \in \mathcal{R} \text{ then } H\,[\![c]\!]_t\ H_1 \Leftrightarrow H[\![c]\!]_t\,H_2.$$

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

F. Evolution Dependency

Every method and constraint will refer to (uses) a number of object types and denotable instances (i.e. directly representable on a communication medium). This relation is provided in the application model universe by means of the dependency relation `Depends`: $\texttt{Depends} \subseteq (\mu \cup \gamma) \times (O \cup \mathcal{L} \times \mathcal{D})$. This relation is modelling technique dependent, but is not subject to evolution.

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

As an example, consider the 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.

V. 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.

A. 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 8. *An application model version over application model universe $\mathcal{U}_\Sigma$ is defined as:*

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

where

$O_t \subseteq O, \mathcal{R}_t \subseteq \mathcal{R}, \mathcal{M}_t \subseteq \mathcal{M}$,

$\texttt{HasTypes}_t \subseteq \texttt{HasTypes}$, and $\texttt{Dom}_t \in \texttt{Dom}$.

From a version of an application model, we can derive the current version

$$I_t = \langle \mathcal{L}_t, \mathcal{N}_t, \sim_t, \rightsquigarrow_t \rangle$$

of the information structure as follows:

$$\mathcal{L}_t = O_t \cap \mathcal{L},$$

$$\mathcal{N}_t = O_t \cap \mathcal{N},$$

$$x \sim_t y \triangleq x \sim y \wedge x, y \in O_t,$$

$$x \rightsquigarrow_t y \triangleq x \rightsquigarrow y \wedge x, y \in O_t.$$

Every application model version must adhere to certain

rules of well-formedness. Some of these rules are modelling technique dependent, and therefore outside the scope of this paper. Nonetheless, some general rules about application model versions can be stated.

A.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 $(x \in O_t)$. 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

$$\exists_{v,t}[v \texttt{HasTypes}_t X].$$

In the remainder of this subsection, a number a rules for instance typings will follow.

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

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

$$X \subseteq O_t.$$

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 III.2) to this axiom shows the presence of roots in instance typings:

LEMMA 4 (*active roots*). *If X is an instance typing, then:*

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

In most traditional data modelling techniques each type hierarchy has a unique root. As a consequence, each instance typing contains a unique root. Some data modelling techniques, however, allow type hierarchies with multiple roots (see Fig. 7). For such modelling techniques, the following axiom guarantees a unique root for each instance typing.

[AMV4] (*unique root*). If X is an instance typing and $x, y \in X$ then: $\texttt{Root}(x) \wedge \texttt{Root}(y) \Rightarrow x = y$.

The above axiom leads to the following strengthening of Lemma 4.

LEMMA 5 (*active root*). *If X is an instance typing, then:*

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

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 $P(x) = x \in O_t$, is a weak inheritance property.

Note that AMV5 cannot be derived from 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 6 (*living roots*).

$$y \in O_t \Rightarrow \exists x[x \in O_t \wedge x \texttt{RootOf} y].$$

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

A.2. Well-Formed Concretisation

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

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

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

[AMV7] (*strong typing of labels*). If $v \texttt{HasTypes}_t X$ and $v \in \cup \mathcal{D}$ then: $x \in X \Rightarrow v \in \texttt{Dom}_t(x)$.

A.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:

$$\texttt{dom}(w) \subseteq O_t.$$

where $\texttt{dom}(w) = \{x | \langle x, y \rangle \in w\}$ 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:

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

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

[AMV10] (*inheritance of definitions*). If $w \in \mathcal{R}_t \cup \mathcal{M}_t$ then the relation P, defined by $P(x) = x \in \mathbf{dom}(w)$, 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 Section IV.F). 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:

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

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: $\forall_{t,H}\,[H[\![d_1]\!]_t \Rightarrow H[\![d_2]\!]_t]$. The intuitive meaning of $d_1 \Vdash d_2$ is: d_1 is at least as restrictive as d_2 (see also [12]).

B. Populations of Information Structures

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

DEFINITION 9. *The population at any point of time, is a mapping Pop*:

$$T \rightarrow (O \rightarrow \wp(\Omega)), \text{ defined by: } \mathtt{Pop}_t(x) = \left\{v \middle| \exists_Y \left[v\mathtt{HasTypes}_t\, Y \wedge x \in Y\right]\right\}.$$

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

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

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

LEMMA 7 (*population distribution*). *Every instance of an object type, is also instance of one of its roots*:

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

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

COROLLARY 1.

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

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.

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

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

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

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

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

From Lemma 7 and Theorem 4 the main typing theorem is derived:

THEOREM 5 (*strong typing theorem*).

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

We will now define what constitutes a wellformed application model version. Let $\Sigma_t = (O_t, \mathcal{R}_t, \mathcal{M}_t, \mathtt{HasTypes}_t, \mathtt{Dom}_t)$:

$$\mathtt{IsAM}(\Sigma_t) \triangleq \mathtt{IsSch}(O_t) \wedge \mathtt{IsPop}(O_t, \mathtt{HasTypes}) \wedge \Sigma_t$$

adheres to the AMV axioms.

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

VI. 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 11. *Application model elements*:

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

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.

A. 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 $X \in \{O, \mathcal{R}, \mathcal{M}, \texttt{HasTypes}, \texttt{Dom}\}$, and $h \in H$

then:

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

where

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

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 12. *Object type histories*:

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

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:

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

concretisation histories:

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

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

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

B. Deriving Application Model Versions

At any point of time t the application model version $\Sigma_t(H) = \langle O_t, \mathcal{R}_t, \mathcal{M}_t, \texttt{HasTypes}_t, \texttt{Dom}_t \rangle$ is easily derived from an application model history H. This is done by defining the five main components, which determine an application version:

DEFINITION 13.

object types: $O_t = \{h(t) \mid h \in H_{type} \wedge h \downarrow t\}$

constraints: $\mathcal{R}_t = \{c(t) \mid c \in H_{constr} \wedge c \downarrow t\}$

methods: $\mathcal{M}_t = \{m(t) \mid m \in H_{meth} \wedge m \downarrow t\}$

population: $\texttt{HasTypes}_t = \{g(t) \mid_{g \in H_{pop} \wedge g \downarrow t}\}$

concretisations: $\texttt{Dom}_t = \{d(t) \mid_{d \in H_{dom} \wedge d \downarrow t}\}$

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

C. 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 $c \in H_{constr}: c \downarrow t \Rightarrow H[T] \llbracket c(t) \rrbracket_t$, where T is the largest time interval such that: $\forall_{t' \in T} [t' \leq t \wedge c(t') = c(t)]$. 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.

D. 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 CDs 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 LPs. 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 8. *If*

$$h_1, h_2 \in H_{type},\ h_1 \downarrow t,\ h_2 \downarrow t,\ \textit{and}\ h_2 \downarrow \rhd t$$

then:

$$h_1(t) \rightsquigarrow h_2(t) \wedge h_1(\rhd t) \sim h_2(\rhd t)$$
$$\wedge \texttt{Root}(h_2(\rhd t)) \Rightarrow \neg h_1 \downarrow \rhd t.$$

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 $g \downarrow \rhd t$

then

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

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:

$$\mathtt{Types}_t(g) \triangleq \bigcup_{X:g\,\mathtt{HasTypes}_t\,X} X.$$

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 introduce IsAMH formally:

DEFINITION 14.

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

E. 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., [33], [2] 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 [30] or [26].

VII. 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 [30], [26], and the conceptual transaction modelling technique Hydra [13], [12], leading to Hydrac [26].

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 [27], [16]. 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.

Acknowledgments

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

[1] J.F. Allen, "Towards a general theory of action and time," *Artificial Intelligence*, vol. 23, pp. 123–154, 1984.

[2] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth, "Semantics and implementation of schema evolution in object-oriented databases," *SIGMOD Record*, vol. 16, no. 3, pp. 311–322, Dec. 1987.

[3] R. Bretl, D. Maier, A. Otis, D.J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams, "The GemStone data management system," W. Kim and F.H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications*. Reading, Mass.: Addison-Wesley, pp. 283–308, 1989.

[4] K.B. Bruce and P. Wegner, "An algebraic model of subtype and inheritance," F. Bancilhon and P. Buneman, eds., *Advances in Database Programming Languages*. Reading, Mass.: ACM Press, Frontier Series, pp. 75–96, 1990.

[5] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–522, Dec. 1985.

[6] P.P. Chen, "The entity-relationship model: Toward a unified view of data," *ACM Trans. on Database Systems*, vol. 1, no. 1, pp. 9–36, Mar. 1976.

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

[8] O.M.F. De Troyer, "The OO-binary relationship model: A truly object oriented conceptual model," R. Andersen, J.A. Bubenko, and A. Sølvberg, eds., *Proc. Third Int'l Conf. CAiSE'91 on Advanced Information Systems Engineering*, vol. 498, Lecture Notes in Computer Science, pp. 561–578, Trondheim, Norway, May 1991. New York: Springer-Verlag, 1991.

[9] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich, "Conceptual modelling of database appli-

cations using an extended ER model," *Data & Knowledge Eng.*, vol. 9, no. 4, pp. 157–204, 1992.

[10] E.D. Falkenberg, J.L.H. Oei, and H.A. Proper, "A conceptual framework for evolving information systems," H.G. Sol and R.L. Crosslin, eds., *Dynamic Modelling of Information Systems*, vol. 2. Amsterdam, The Netherlands: North-Holland, pp. 353–375, 1992.

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

[12] A.H.M. ter Hofstede, "Information modelling in data intensive domains," PhD thesis, Univ. of Nijmegen, Nijmegen, The Netherlands, 1993.

[13] A.H.M. ter Hofstede and E.R. Nieuwland, "Task structure semantics through process algebra," *Software Eng. J.*, vol. 8, no. 1, pp. 14–20, Jan. 1993.

[14] A.H.M. ter Hofstede, H.A. Proper, and T.P. van der Weide, "Data modelling in complex application domains," P. Loucopoulos, ed., *Proc. Fourth Int'l Conf. CAiSE'92 on Advanced Information Systems Engineering*, vol. 593 of Lecture Notes in Computer Science, pp. 364–377, Manchester, United Kingdom, May 1992. New York: Springer-Verlag, 1992.

[15] A.H.M. ter Hofstede, H.A. Proper, and T.P. van der Weide, "Formal definition of a conceptual language for the description and manipulation of information models," *Information Systems*, vol. 18, no. 7, pp. 489–523, 1993.

[16] A.H.M. ter Hofstede, H.A. Proper, and T.P. van der Weide, "Supporting information disclosure in an evolving environment," D. Karagiannis, ed., *Proc. Fifth Int'l Conf. DEXA'95 on Database and Expert Systems Applications*, vol. 856 of Lecture Notes in Computer Science, pp. 433–444, Athens, Greece, Sept. 1994. New York: Springer Verlag, 1994.

[17] A.H.M. ter Hofstede and T.P. van der Weide, "Expressiveness in conceptual data modelling," *Data & Knowledge Eng.*, vol. 10, no. 1, pp. 65–100, Feb. 1993.

[18] R.H. Katz, "Toward a unified framework for version modelling in engineering databases," *ACM Computing Surveys*, vol. 22, no. 4, pp. 375–408, 1990.

[19] W. Kim, N. Ballou, H.-T. Chou, J.F. Garza, and D. Woelk, "Features of the ORION object-oriented database," W. Kim and F.H. Lochovsky, eds., *Object-Oriented Concepts, Databases, and Applications*. Reading, Mass.: ACM Press, Frontier Series, pp. 251–282, 1989.

[20] T. Korson and J. McGregor, "Understanding object oriented: A unifying paradigm," *Comm. ACM*, vol. 33, no. 9, pp. 40–60, Sept. 1990.

[21] L. Lamport, "The temporal logic of actions," Report 79, Digital Systems Research Center, Palo Alto, Calif., Dec. 1991.

[22] G.T. Nguyen and D. Rieu, "Schema evolution in object-oriented database systems," *Data & Knowledge Eng.*, vol. 4, pp. 43–67, 1989.

[23] G.M. Nijssen and T.A. Halpin, *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*. Englewood Cliffs, N.J.: Prentice Hall, 1989.

[24] J.L.H. Oei, H.A. Proper, and E.D. Falkenberg, "Evolving information systems: Meeting the ever-changing environment," *Information Systems J.*, vol. 4, no. 3, pp. 213–233, July 1994.

[25] A. Ohori, "Orderings and types in databases," F. Bancilhon and P. Buneman, eds., *Advances in Database Programming Languages*. Reading, Mass.: ACM Press, Frontier Series, pp. 97–116, 1990.

[26] H.A. Proper, "A theory for conceptual modelling of evolving application domains," PhD thesis, Univ. of Nijmegen, Nijmegen, The Netherlands, 1994.

[27] H.A. Proper and T.P. van der Weide, "Information disclosure in evolving information systems: Taking a shot at a moving target," Technical Report 93-22, Information Systems Group, Computing Science Inst., Univ. of Nijmegen, The Netherlands, 1993.

[28] H.A. Proper and T.P. van der Weide, "Towards a general theory for the evolution of application models," M.E. Orlowska and M. Papazoglou, eds., *Proc. Fourth Australian Database Conf. Advances in Database Research*, pp. 346–362. World Scientific, Brisbane, Australia, Feb. 1993.

[29] H.A. Proper and T.P. van der Weide, "A general theory for the evolution of application models," Technical Report 317, Dept. of Computer Science, Univ. of Queensland, Brisbane, Australia, 1994.

[30] H.A. Proper and T.P. van der Weide, "EVORM: A conceptual modelling technique for evolving application domains," *Data & Knowledge Eng.*, vol. 10, no. 12, pp. 313–359, 1994.

[31] J.F. Roddick, "Dynamically changing schemas within database models," *Australian Computer J.*, vol. 23, no. 3, pp. 105–109, Aug. 1991.

[32] J.F. Roddick and J.D. Patrick, "Temporal semantics in information systems—a survey," *Information Systems*, vol. 17, no. 3, pp. 249–267, 1992.

[33] A.H. Skarra and S.B. Zdonik, "The management of changing types in an object-oriented database," N. Meyrowitz, ed., *Proc. ACM Conf. of Object-Oriented Systems, Languages and Applications (OOPSLA)*, pp. 483–495, Portland, Ore., Sept. 1986.

[34] R. Snodgrass, "Temporal databases status and research directions," *SIGMOD Record*, vol. 19, no. 4, pp. 83–89, Dec. 1990.

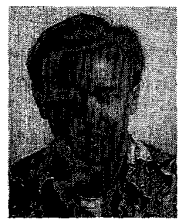
[35] R. Snodgrass and I. Ahn, "A taxonomy of time in databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 236–246, Austin, Texas, 1985.

[36] M.T. Tresch and M.H. Scholl, "Meta object management and its application to database evolution," G. Pernul and A.M. Tjoa, eds., *11th Int'l Conf. Entity-Relationship Approach*, vol. 645 of Lecture Notes in Computer Science, pp. 299–321, Karlsruhe, Germany, Oct. 1992. New York: Springer-Verlag, 1992.

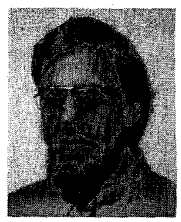
[37] G. Wiederhold, S. Jajodia, and W. Litwin, "Dealing with the granularity of time in temporal databases," R. Andersen, J.A. Bubenko, and A. Sølvberg, eds., *Proc. Third Int'l Conf. CAiSE'91 on Advanced Information Systems Eng.*, vol. 498 of Lecture Notes in Computer Science, pp. 124–140, Trondheim,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

[38] 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*. New York: Springer-Verlag, 1976.

[39] E. Yourdon. *Modern Structured Analysis*. Englewood Cliffs, N.J.: Prentice Hall, 1989.



H.A. Proper received his master's degree from the University of Nijmegen, The Netherlands, in the summer of 1990. In the spring of 1994 he received his PhD from the University of Nijmegen. He is currently at Cooperative Information Systems Research Centre, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia. His main research interests include (evolving) information systems, information retrieval, CASE-tool technology, conceptual modelling, hypertext, and knowledge based systems.



T.P. van der Weide received his master's degree from the Technical University Eindhoven, The Netherlands, in 1975 and the degree of PhD in mathematics and physics from the University of Leiden, the Netherlands, in 1980. He is currently associate professor at the University of Nijmegen, The Netherlands. His main research interests include information systems, information retrieval, hypertext, and knowledge-based systems.