Deriving and Paraphrasing Information Grammars using Object-oriented Analysis Models

P.J.M. Frederiks
ICT-N Innovation Centre
Philips Semiconductors
Gerstweg 2
6534 AE Nijmegen
The Netherlands
paul.frederiks@philips.com

Th.P. van der Weide*
Computing Science Institute
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands
tvdw@cs.kun.nl

Abstract

In this paper the focus is on object-oriented analysis of information systems. We assume that the communication within an application domain can be described by a logbook of events. In our view, the purpose of the analysis phase is to model the structure of this logbook. The resulting conceptual model is referred to as the information architecture, and is an integration of three formal object-oriented analysis models with each a specific view on the application domain. Furthermore, the information architecture forms an abstraction of an underlying grammar, called the information grammar, for the communication within the application domain. This grammar can be used to validate the information architecture in a textual format by informed users. In addition, the information grammar can be used to obtain the relevant data and processes of the application domain, and serves as a basis for the query language of users with the information system.

1 Introduction

The evolution of description techniques of information systems has in the course of time shifted from machine-oriented techniques into more and more human-oriented approaches (see e.g. [Bub86] and [FW96c]). Rather than focusing only on the structure of the information, the way the information is used to capture the communication language within the application domain, or *Universe of Discourse* (UoD) ([Gri82]), is discussed. This evolution is reflected by a change of (1) the *architecture* (from a file-oriented to a communication-oriented information system architecture), (2) the *man-machine communication* (from using files and cards to using advanced query facilities), and (3) the *analysis methods* (from program construction methods to conceptual analysis methods). A central role in this evolution is played by the so-called *information grammar* (see e.g. [NH89], [WZ96], [HPW94], [Lek93]). This grammar forms the basis for all communication within the UoD and all communication with the information system. As a result, the information grammar can be used (1) to obtain the relevant data types and process types of the UoD, and (2) as a basis for the query language of users with the information system. Usually, this grammar is depicted using graphical models, see e.g. NIAM ([Hal95]) and PSM ([HW93]). As a consequence, these models can be validated by informed users readily, as each model corresponds with a part of the information grammar covering a part of the communication in the UoD.

In order to find the information grammar the NIAM method starts with a initial textual specification of the UoD. The goal of the modeling process is to obtain a complete consistent formal specification from the initial specification. In order to structure this way of working, the (sentences in the) initial specification should be in accordance with some unifying format. In NIAM these sentences are called *elementary sentences*. Elementary sentences provide a simple and effective handle for obtaining the underlying conceptual model of so-called *Snapshot Information Systems* ([Dat91]), i.e. information systems where only the current state of the UoD is relevant. As a result, the information grammar found refers to *static* aspects of the UoD.

*Corresponding author

Even though these textual initial specifications are an important aid in modeling information systems, they still are too poorly structured. This lack of structure results in several problems during the modeling process. For example, a system analyst has to come up with a number of properties of the UoD, which do not follow directly from the initial specification. One of these properties is the order and history of events, i.e. the *dynamic* aspects of the UoD are lost in such specifications. Consequently, the system analyst has to reconstruct this order.

In this paper the notion of *logbook* ([BFW96]) is introduced as a common basis for various models to be produced during system analysis. A logbook has a unifying format which contains a complete description of the history of some UoD. Generally a logbook will report on changes of *instances* within the UoD, as well as on changes of the *structure* (information structure) of the UoD. A logbook may thus be seen as the sequence of all events which describe the evolution of the UoD. From a logbook, not only each past state of the UoD can be derived, but also the current state. As a consequence, a UoD basically corresponds to the set of all acceptable logbooks. The goal of the modeling process then is to obtain a formal description of this set of acceptable logbooks. The logbook is intended as a structuring mechanism for initial specifications in the context of Snapshot Information Systems as well as *Temporal Information Systems* ([Sno90]). Furthermore, it supports the development of *Evolving Information Systems* (e.g. [PW95a], [Tre91], [NR89]).

In order to model a logbook, conceptual analysis methods have to provide analysis models which capture both the static and dynamic aspects of the UoD described in that logbook. Many experts believe that *object-orientation* is a step forward in developing better information systems and they claim that *object-oriented methods* have such analysis models, such as *OOA* ([CY90]), *OMT* ([RBP$^+$91]), *Booch* ([Boo91]), *OOSE* ([JCJO92]) and *UML* ([BRJ99]). However, many of these object-oriented analysis models (1) do not have a formal foundation, (2) lack a clear mechanism for integrating different models (e.g. data and process models), (3) are too complex or too weak, and (4) are focussed on data and processes instead of communication, and are therefore less suitable for finding the information grammar, i.e., a description of the structure of the logbook of the UoD.

In this paper three formally founded object-oriented analysis models are introduced. Each model describes a specific abstract view on the logbook. Each such abstraction provides a specific view on the nature of the UoD (information grammar). We distinguish the following models: (1) the *object action involvement model* describing the main structure of the UoD, (2) the *object property model* dealing with static aspects of the UoD, and (3) the *object life model* which describes dynamic aspects of the UoD. Together these models compose a conceptual model, called the *information architecture* (cf. [Kri94]), describing the structure of the communication of the UoD, i.e. the structure of the logbook, which is an integrated system with respect to (1) the graphical notations used, (2) the formal foundations, and (3) the composition and validation of the rules of the underlying information grammar. The information grammar is validated by paraphrasing into semi-natural language sentences. In this paper we assume that the sentences in the initial specification are liguistically preprocessed. For example, the difference between verbs and nouns is of importance, but the recognition of verbs and nouns is outside the scope of this paper. Furthermore, the processing of constraints has been studied in other papers (for example [HPW93, Bur96]) but will not be discussed in this context.

The organization of this paper is as follows. In the sections 2, 3, and 4 for each analysis model, respectively, the formal foundation and relation with the information grammar is described. Some related work with respect to the use of natural language aspects for conceptual (object-oriented) modeling is described in section 5. Finally, conclusions and directions for further research are presented in section 6. The appendices summarize the graphical notation, the rationale behind the concept of logbook and a comparison of the models introduced in this paper.

2 Object action involvement model

In this section syntax and semantics of the object action involvement model (or OAI model for short) are discussed. Furthermore, it is indicated how to obtain that part of the information grammar which is described by the object action involvement model. The information grammar is validated by a so-called paraphrasing mechanism.

2.1 Informal introduction

The object action involvement model provides a structured description of the structure of events in the UoD stating which (abstract) object types are involved in what action types and in what role. A special kind of involvement indicates which object type(s) is (are) responsible for what action type. These object types thus

are seen as the initiators of that action type. External initiators of action types, so-called *subject types*, fall outside the scope of this paper. The model may also contain composed object types such as generalized object types and group types.

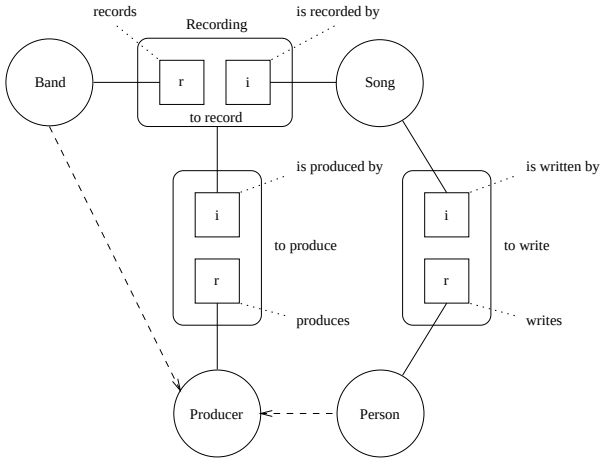

Figure 1: An object action involvement model

Figure 1 shows a sample object action involvement model. In this model *Band*, *Person*, *Producer*, *Recording* and *Song* are object types involved in the action types *to produce*, *to record* and *to write*. An r states that the corresponding object type is *responsible* in the execution of the associated action type, whilst an i denotes (passive) involvement. The dotted arrows represent a generalization (kind of inheritance) relation. Both *Band*s and *Person*s can act as *Producer*, i.e., a *Producer* is either a *Band* or a *Person*. Note that object type *Recording* is the objectification of action type *to record*.

2.2 Syntax

Formally, an *object action involvement model* is a structure $\mathcal{OAI}$ consisting of the following basic aspects:

1. A set $\mathcal{O}$ of *object types*. The set of object types is partitioned into *elementary* object types ($\mathcal{E}$) and *complex* or *composed* object types ($\mathcal{Q}$).

2. A set $\mathcal{P}$ of *predicators*. Predicators are abstractions of roles in actions. The role of predicator p is played by object type $\mathsf{Actor}(p)$.

3. An *action type* corresponds to a set of predicators. The set $\mathcal{A}$ of action types forms a partition of $\mathcal{P}$. Action types are composed object types ($\mathcal{A} \subseteq \mathcal{Q}$).

   The action type associated with predicator p is denoted by $\mathsf{Action}(p)$. We use the notation $p \in a$ as a shorthand for $\mathsf{Action}(p) = a$.

4. Predicators are *characterized* by the function χ. If $\chi(p) = r$, then the object type $\mathsf{Actor}(p)$ is designated as being *responsible* for this action. If $\chi(p) = i$, then $\mathsf{Actor}(p)$ is involved only in that action.

5. A set $\mathcal{G}$ of *group types*. Group types form a special class of composed object types, i.e., $\mathcal{G} \subseteq \mathcal{Q}$. Group types are used for example to handle multiple roles of an object type in an action type.

6. A set $\mathcal{S}$ of *sequence types*. Sequence types form a special class of composed object types, i.e. $\mathcal{S} \subseteq \mathcal{Q}$. Sequence types are used to express ordered lists.

7. The function $\mathsf{Elt} : \mathcal{G} \cup \mathcal{S} \rightarrow \mathcal{O}$ yields the underlying *element type* of group types and sequence types.

8. A set $\mathcal{M}$ of *module types* ($\mathcal{M} \subseteq \mathcal{Q}$). Module types can be used to express model *decomposition*, which is especially useful in the case of large application domains. The relation $\mathsf{compr} \subseteq \mathcal{M} \times \mathcal{O}$ describes how a module type is decomposed into its components, where $x \,\mathsf{compr}\, y$ expresses that y is part of module x.

   The decomposition structure forms a hierarchy of module types. The top element of this hierarchy is obtained by the predicate $\mathsf{Main}(x) \equiv \neg\exists_y [y \,\mathsf{compr}\, x])$.

The predicate $x \,\mathsf{relative}\, y \equiv \forall_z [z \,\mathsf{compr}\, x \Rightarrow z \,\mathsf{compr}\, y]$ indicates that object type x occurs in the same decompositions as object type y. We use $x \,\mathsf{family}\, y$ as a shorthand for $x \,\mathsf{relative}\, y \wedge y \,\mathsf{relative}\, x$. Note that this relation is an equivalence relation.

The predicate $\mathsf{IsParent}(x, y) \equiv x \,\mathsf{compr}\, y \wedge \neg\exists_z [x \,\mathsf{compr}\, z \wedge z \,\mathsf{compr}\, y]$ states that object type y is introduced by module type x. We will use $\mathsf{Parent}(x)$ to denote the unique parent of object type x (if $\neg\, \mathsf{Main}(x)$).

The object action involvement model associated with module type x is denoted as $\mathcal{OAI}_x$, and is derived by restricting the basic components of $\mathcal{OAI}$ to the spanning set $\mathcal{O}_x = \{x\} \cup \{y \in \mathcal{O} \mid x \,\mathsf{compr}\, y\}$ of object types[1].

For convenience we introduce $\mathsf{Locals}(m) \equiv \{x \mid \mathsf{IsParent}(m, x)\}$ to denote the set of all object types within a module m.

9. The relation gen expresses the *generalization* structure for object types. If x is a generalization of y, then this is denoted as $x \,\mathsf{gen}\, y$. y is called a *specifier* of x. We use $\mathsf{IsGen}(x) \equiv \exists_y [x \,\mathsf{gen}\, y]$ to indicate that x is a generalized object.

   Generalized object types are usually inhomogeneous, as the specifiers have a different structure, i.e. specifiers do not share instances. Generalized object types are used to reduce schema redundancy, by relating similar action types of specifiers to the generalization.

   In section 4.3.4 the *generalization defining rule* is introduced as a rule for being member of a generalized object type.

10. The relation spec is used to describe *specialization*. If x is a specialization of y, then this is denoted as $x \,\mathsf{spec}\, y$. x is also called a *subtype* of y. We use $\mathsf{IsSpec}(x) \equiv \exists_y [x \,\mathsf{spec}\, y]$ to indicate that x is a specialized object.

    Contrary to generalization, specialization is a restriction mechanism. Specialized object types have the same structure as their supertype, but will participate in extra action types. Each specialization hierarchy will have a unique top element, referred to as its pater familias. The predicate $\sqcap(x, y) \equiv \neg\, \mathsf{IsSpec}(y)$ (if $x \,\mathsf{spec}\, y$, while $x = y$ otherwise) state that y is the pater familias of x.

    A subtype thus can be seen as a special grouping of instances of some object type, while a generalization is a grouping of instances of different object types.

    In section 4.3.4 the *subtype defining rule* is introduced as a rule for being member of a subtype.

The structure $\mathcal{OAI}$ must satisfy a number of axioms of well formedness to be a valid OAI model. The axioms are summarized by:

General rules	OAI-1	*(no cycles)*	$\mathsf{Actor}(p) \neq \mathsf{Action}(p)$
	OAI-2	*(born objects)*	$\exists_{p \in \mathcal{P}} [\mathsf{Actor}(p) = x]$
Decomposition	OAI-3	*(structure* compr*)*	compr is a partial order
	OAI-4	*(main module)*	$\exists!_x [\mathsf{Main}(x)]$
	OAI-5	*(unique parent)*	$\mathsf{IsParent}(x_1, y) \wedge \mathsf{IsParent}(x_2, y) \Rightarrow x_1 = x_2$
	OAI-6	*(decomposition validity)*	$x \in \mathcal{M} \Rightarrow \mathcal{OAI}_x$ is a valid OAI model
Subtyping	OAI-7	*(structural relatedness)*	spec $\cup$ gen $\subseteq \mathcal{E} \times \mathcal{O}$
	OAI-8	*(structure* gen*)*	gen is a partial order
	OAI-9	*(structure* spec*)*	spec is a partial order
	OAI-10	*(enforcing pater familias)*	$\sqcap(x, y) \wedge \sqcap(x, z) \Rightarrow y = z$
	OAI-11	*(*gen-spec *harmony)*	$\mathsf{IsGen}(x) \Rightarrow \neg\, \mathsf{IsSpec}(x)$
	OAI-12	*(*spec-compr *harmony)*	$x \,\mathsf{spec}\, y \Rightarrow y \,\mathsf{relative}\, x$
	OAI-13	*(*gen-compr *harmony)*	$x \,\mathsf{gen}\, y \Rightarrow x \,\mathsf{relative}\, y$

Lemma 1 $\neg x \,\mathsf{relative}\, y \iff \exists_z [z \,\mathsf{compr}\, x \wedge \neg z \,\mathsf{compr}\, y]$

Example 1 Figure 2 shows a graphical representation of the following algebraic description:

[1] Note that $\mathcal{O}_x = \{x\}$ if x is not a module type

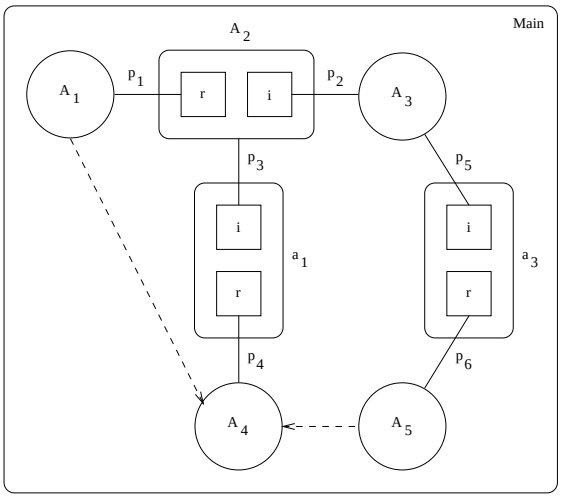

Figure 2: A sample object action involvement model

$\mathcal{O}$	$\mathcal{A}$
Main	a_1
A_1	A_2
A_2	a_3
A_3	
A_4	
A_5	
a_1	
a_3	

subtyping
A_4 gen A_1
A_4 gen A_5

$\mathcal{P}$	Action	Actor	χ
p_1	A_2	A_1	r
p_2	A_2	A_3	i
p_3	a_1	A_2	i
p_4	a_1	A_4	r
p_5	a_3	A_3	i
p_6	a_3	A_5	r

The relation compr is obvious, and therefore omitted.

2.3 Inheritance

The concept of type relatedness states whether object types can have instances in common in some instantiation and is used to enforce a consistent typing of the instances of the sample sentences. The notation $\mathsf{TypeRel}(x, y)$ is used to indicate that x is type related with y. For example, if object type x is a specialization of object type y, then $\mathsf{TypeRel}(x, y)$.

Once a consistent typing is achieved the concept of *inheritance* can be used to express the existence of common action types and properties for object types.

Intuitively, object types can share action types and properties. For example, object types *Person* and *Band* inherit action type *to produce* from their generalized object type *Producer* (see figure 1). An object type x has the potential to inherit action types and properties from an object type y, denoted as $\mathsf{Inherits}(x, y)$, iff this can be proven using the derivation rules introduced in the remainder of this section. The relation Inherits thus describes the *class hierarchy* as introduced in conventional object-oriented analysis methods (see e.g. [CY90]). The concept of inheritance is used to provide a more eloquent paraphrasing mechanism (see section 2.7). Section 2.6 discusses the semantics of the inheritance relation.

Inheritance rules	I-1	*(reflexive)*	$\vdash \mathsf{Inherits}(x, x)$
	I-2	*(specialization inheritance)*	x spec $y \wedge \mathsf{Inherits}(y, z) \vdash \mathsf{Inherits}(x, z)$
	I-3	*(generalization inheritance)*	y gen $x \vdash \mathsf{Inherits}(x, y)$

2.4 Events of object action involvement model

The semantics of an object action involvement model is the set of its possible *logbooks*, where the events are conforming to the structure layed down in the object action involvement model. The expression $\mathsf{IsLog}(\mathcal{OAI}, \mathsf{L})$

expresses that L is a logbook of object action involvement model $\mathcal{OAI}$. A logbook L of an object action involvement model is in this approach a mapping:

$$\mathsf{L} : \mathcal{O} \rightarrow \wp^{\mathrm{fin}}(\mathsf{Time} \times \Omega)$$

where Ω includes a set of *object identifiers*, and Time the set of *time stamps*. This instantiation mechanism can be described formally via the category **TimeStampSet**, using the category theoretical approach from [HLF96], [FHL97], and [HLW97], where it has been shown to be a general population mechanism that enforces all usual properties of populations. For example, the requirement that the time stamp of an action may not precede the time stamp of its involved instances is formulated within this framework.

As an example, the population of action type A_2 (to record) can be described as:

$$\{\langle \text{21-06-1967}, p_1 : \text{The Rolling Stones}, p_2 : \text{Paint It Black}\rangle\,,\\ \langle \text{29-04-1991}, p_1 : \text{Playful Plebs}, p_2 : \text{I Want You}\rangle\}$$

The labels are used to denote the abstract instances to improve readability (labels are introduced in the object property model, see section 3).

2.5 Naming and verbalization rules

The object action involvement model is made communicable by naming its elements and providing rules to verbalize its structure. For the naming elements we assume a name space `Names` and a qualification operator $\odot : \texttt{Names} \times \texttt{Names} \rightarrow \texttt{Names}$ such that:

$$x_1 \odot y_1 = x_2 \odot y_2 \Rightarrow x_1 = x_2 \wedge y_1 = y_2$$

The action types which are also involved in some predicator, i.e. action types which are objectified, are treated in a special way. First we introduce the following auxiliary predicate:

$$\mathsf{Objectified}(x) \equiv x \in \mathcal{A} \wedge \exists_p [\mathsf{Actor}(p) = x]$$

Object types involved in some predicator are called *active*. The set $\mathcal{I}$ of all active object types is defined by:

$$\mathcal{I} = \mathcal{O} \setminus \mathcal{A} \cup \{x \mid \mathsf{Objectified}(x)\}$$

In the remainder of this paper the terms *object type* and *active object type* are used interchangeably whenever this will not lead to confusion.

The naming structure consists of the following components:

1. The functions INm, ANm, and PNm assign a name to active object types, action types, and predicators, respectively. The object type names and action type names are used to identify object types and action types within their context. We assume names for object types and action types to be different.

   Predicator names are used to denote how the actor is involved in the action associated with that predicator.

2. The functions IVerbs and AVerbs assign a set of verbalization rules to each active object type and action type, respectively. The verbalizing sets are used for paraphrasing the object action involvement model to natural language sentences. Each verbalization rule corresponds to the structure of some sample sentences from the sample logbook.

The naming structure has to fulfill a number of requirements. A first requirement is that different object types can be denoted differently. As names for object types need not to be unique, unique object denotations are formed by qualification with the module denotations in which they are introduced. For this purpose, the family name $\mathsf{FNm}(x)$ of object type x is introduced as its shortest qualification leading to a unique denotation:

$$\mathsf{FNm}(x) = \begin{cases} \mathsf{INm}(x) & \text{if } \mathsf{INm}(x) \text{ is unique or } \mathsf{Main}(x) \\ \mathsf{FNm}(\mathsf{Parent}(x)) \odot \mathsf{INm}(x) & \text{otherwise} \end{cases}$$

Object types should have unique family names:

[N-1] *(unique family name)* $\quad \mathsf{FNm}(x) = \mathsf{FNm}(y) \Rightarrow x = y$

As a consequence, object types within the same module must have different names.

Lemma 2 $\mathsf{INm}(x) = \mathsf{INm}(y) \wedge x \neq y \Rightarrow \neg x \,\mathsf{family}\, y$

Proof:
Suppose $\mathsf{INm}(x) = \mathsf{INm}(y) \wedge x \neq y$. As object types have a unique *family name* and $x \neq y$ we know that $\mathsf{FNm}(x) \neq \mathsf{FNm}(y)$. Together with the fact that each object type has a unique parent and $\mathsf{INm}(x) = \mathsf{INm}(y)$ it holds that $\mathsf{FNm}(\mathsf{Parent}(x)) \neq \mathsf{FNm}(\mathsf{Parent}(y))$ and thus $\mathsf{Parent}(x) \neq \mathsf{Parent}(y)$. As a consequence $\neg\, \mathsf{Parent}(x) \,\mathsf{compr}\, y$. On the other hand $\mathsf{Parent}(x) \,\mathsf{compr}\, x$. Applying lemma 1 leads to $\neg x \,\mathsf{relative}\, y$ and thus $\neg x \,\mathsf{family}\, y$. ◇

Action types may also have the same name, as long as they can be distinguished by the (family) names of their actors:

[**N-2**] *(action naming)* $\mathsf{ANm}(a) = \mathsf{ANm}(b) \wedge \neg a \,\mathsf{family}\, b \Rightarrow \exists_{p \in a \wedge q \in b} [\mathsf{Actor}(p) \neq \mathsf{Actor}(q)]$

A unique naming for action types thus is obtained by involving the actors. Let action type a consist of predicators $p_1, \ldots, p_k$, ordered alphabetically by the family name of the corresponding actors. Then an action type is uniquely denoted within its parent module $\mathsf{Parent}(a)$ as:

$$\mathsf{ANm}'(a) = \begin{cases} \mathsf{ANm}(a) & \text{if } \mathsf{ANm}(a) \text{ is unique} \\ \mathsf{ANm}(a) \odot \mathsf{FNm}(\mathsf{Actor}(p_1)) \odot \ldots \odot \mathsf{FNm}(\mathsf{Actor}(p_k)) & \text{otherwise} \end{cases}$$

This is the basis for the family name $\mathsf{FNm}(a)$ for action type a:

$$\mathsf{FNm}(a) = \begin{cases} \mathsf{ANm}'(a) & \text{if } \mathsf{ANm}'(a) \text{ is unique} \\ \mathsf{FNm}(\mathsf{Parent}(a)) \odot \mathsf{ANm}'(a) & \text{otherwise} \end{cases}$$

An analogous requirement is posed for predicator names:

[**N-3**] *(predicator naming)* $\mathsf{PNm}(p) = \mathsf{PNm}(q) \wedge p \neq q \Rightarrow \mathsf{Action}(p) \neq \mathsf{Action}(q)$

We can uniquely denote each predicator as follows:

$$\mathsf{FNm}(p) = \begin{cases} \mathsf{PNm}(p) & \text{if } \mathsf{PNm}(p) \text{ is unique} \\ \mathsf{FNm}(\mathsf{Action}(p)) \odot \mathsf{PNm}(p) & \text{otherwise} \end{cases}$$

A verbalization rule of a schema element (object type or predicator) is a context-free grammar rule describing how to put it into words. The verbalization of a schema element is a set of associated verbalization rules. The verbalization of a schema element is also referred to as its concrete type. Note that verbalization rules for object types can be enriched for example with synonyms of object type names using lexica such as *WordNet* ([MBF+93]).

The first restriction on forming verbalization rules states that different object types have different concrete typing:

[**V-1**] *(unique object verbalization)* $\mathsf{IVerbs}(x) = \mathsf{IVerbs}(y) \Rightarrow x = y$

Action types have extra verbalizations, originating from the sentence structures derived from the sample sentences.

[**V-2**] *(unique action verbalization)* $\mathsf{AVerbs}(a) = \mathsf{AVerbs}(b) \Rightarrow a = b$

Example 2 By naming the elements of figure 2 appropriately, figure 1 can be constructed:

$\mathcal{I}$	INm
Main	*Main*
A_1	*Band*
A_2	*Recording*
A_3	*Song*
A_4	*Producer*
A_5	*Person*

$\mathcal{A}$	ANm
a_1	*to produce*
A_2	*to record*
a_3	*to write*

$\mathcal{P}$	PNm
p_1	*agent*
p_2	*object*
p_3	*object*
p_4	*agent*
p_5	*object*
p_6	*agent*

Note that some predicators have the same predicator name. Using family names provides a unique name:

$\mathcal{P}$	FNm
p_1	*to record* $\odot$ *agent*
p_2	*to record* $\odot$ *object*
p_3	*to produce* $\odot$ *object*
p_4	*to produce* $\odot$ *agent*
p_5	*to write* $\odot$ *object*
p_6	*to write* $\odot$ *agent*

The verbalizations of these schema elements are as follows:

$\mathcal{I}$	IVerbs
A_1	"Pop group"
A_2	"the recording of" $\langle p_2 \rangle$ "recorded by" $\langle p_1 \rangle$

$\mathcal{A}$	AVerbs
a_1	$\langle p_4 \rangle$ "produces" $\langle p_3 \rangle$
a_1	$\langle p_3 \rangle$ "is produced by" $\langle p_4 \rangle$
A_2	$\langle p_1 \rangle$ "records" $\langle p_2 \rangle$
A_2	$\langle p_2 \rangle$ "is recorded by" $\langle p_1 \rangle$
a_3	$\langle p_6 \rangle$ "writes" $\langle p_5 \rangle$
a_3	$\langle p_5 \rangle$ "is written by" $\langle p_6 \rangle$

Note that object types can also be verbalized by their object name. This is omitted in the table. However, the table has an entry which provides an alternative verbalization for bands. Furthermore, the table provides a more elaborated object verbalization for recordings.

2.6 Binding mechanism

In section 2.3 we introduced a scheme describing potential *inheritance* between object types. This is usually referred to as the class hierarchy. In this section we describe what will actually be inherited. The binding relation is required during execution of the information grammar when the meaning of a user sentence is to be investigated by an interpreter of the information grammar.

Each object type in the object action involvement model is involved in a number of action types. However, an action type name may be overloaded, leaving the question what action is meant in the context of a particular object type. Associating a name with an action type, in the context of an object type, is called *binding*. The primitive binding rules are described by the predicate $\mathsf{Has} \subseteq \mathcal{O} \times \mathtt{Names} \times \mathcal{A}$ defined as follows:

[B-1] *(primitive binding)* $\quad \vdash \mathsf{Has}(\mathsf{Actor}(p), \mathsf{ANm}(\mathsf{Action}(p)), \mathsf{Action}(p))$

This primitive binding predicate is extended over the class hierarchy, leading to the predicate Binds, as follows:

[B-2] *(basic binding)* $\quad \mathsf{Has}(x, n, a) \vdash \mathsf{Binds}(x, n, a)$

[B-3] *(inheritance)* $\quad \neg\, \mathsf{Has}(x, n, a) \wedge \mathsf{Inherits}(x, y) \wedge \mathsf{Binds}(y, n, a) \vdash \mathsf{Binds}(x, n, a)$

Next we consider binding at run time. Consider an instance i, then this instance has associated a number of object types at each moment. The predicate $\mathsf{HasType}(\mathsf{L}, i, x)$ states that at the end of logbook L instance i is of type x. The set of all object types associated with instance i after L is referred to as the class hierarchy of i after L. To identify the meaning of a name n in the context of this instance, we determine whether object type x, associated with i after L, assigns a meaning to name n:

$$\mathsf{IsAwareOf}(\mathsf{L}, i, n, x) \equiv \mathsf{HasType}(\mathsf{L}, i, x) \wedge \exists_a [\mathsf{Binds}(x, n, a)]$$

From the class hierarchy we select the deepest descendants (types) of instance i knowing n in the context of L from this partial family tree:

$$\mathsf{IsDefiner}(\mathsf{L}, i, n, x) \equiv \mathsf{IsAwareOf}(\mathsf{L}, i, n, x) \wedge \forall_y [\mathsf{IsAwareOf}(\mathsf{L}, i, n, y) \wedge \mathsf{Inherits}(y, x) \Rightarrow x = y]$$

If there is more than one definer, the name n has more than one meaning (*multiple inheritance*). Multiple *inheritance* can be excluded by the following axiom:

[**B-4**] *(no multiple inheritance)* $\mathsf{IsDefiner}(\mathsf{L}, i, n, x) \wedge \mathsf{IsDefiner}(\mathsf{L}, i, n, y) \Rightarrow x = y$

In a later section, properties of object types are introduced. The inheritance of properties is derived via the action types which update and inspect these properties.

The role of inheritance for conceptual modeling and consequences of multiple inheritance remain an actual research topic. For further readings about (multiple) inheritance the reader is referred to [Bra83], [Car84], [Wil96], or [SD96].

2.7 Paraphrasing mechanism

Next we introduce a *paraphrasing mechanism* for the object action involvement model. The paraphrasing mechanism is presented as a context-free grammar, which will be introduced in the rest of this section. The rules of this grammar are referred to as *paraphrasing rules*. The paraphrasing mechanism should be able to produce a textual description of the structure of all events that may occur in the UoD. A sketch of an algorithm in pseudo code, which collects all grammar rules for the information grammar, is presented in [FW96b]. For an example of an experiment using this paraphrasing mechanism, see [DFW96].

2.7.1 Context

Textual analysis and paraphrasing can be seen as counterpart of each other in the following sense. During the analysis phase textual diversifications (e.g. word order, modalities) are mapped onto single elementary sentences (NIAM), also reffered to as structured sentences (KISS ([Kri94, HVH90])). Parsing may be seen as a many-to-one function.

On the other hand paraphrasing may be seen as a one-to-many function by introducing syntactical variation to improve readability (c.f. [The00]. As a result paraphrasing may introduce ambiguity, without necessarily having a negative effect on interpretation.

A usual requirement for a paraphrasing mechanism is that analyzing its result will produce the original model. This relation between analysis and paraphrasing is expressed as:

$$\text{analysis} \circ \text{paraprase} \circ \text{analysis} = \text{analysis}$$

where $\circ$ is functional composition. In general the function

$$\text{analysis} \circ \text{paraprase}$$

will not be the identity function.

Ambiguity can occur at several levels. For example, word ambiguity is the ambiguity resulting from words having more than one meaning. Syntactical ambiguity means that a sentence can be assigned more than one syntactical structure. Anaphora related ambiguity concerns inter-sentence references, for example *The guitar player*$_1$ *entered the studio*$_2$*;* **She**$_1$ *did not see her guitar in* **there**$_2$. Text of discourse ambiguity refers to ambiguity over a complete text.

In [ASU86] guidelines are discussed to avoid the introducion of ambiguity while constructing a grammar. For example, syntactical ambiguity may be avoided by keeping open and closed symbols different.

The paraphrasing mechanism introduced in this paper is constructed to avoid syntactical ambiguity.

2.7.2 General format of paraphrasing rules

The verbalization rules for action types, and the names for object and action types form the point of departure for the formulation of the paraphrasing rules. We will generate paraphrasing rules in the style of *affix grammars* (see for example [Kos70]), a family of two-level grammars.

The first level of the grammar consists of context-free production rules containing affixes. The second level defines the domains of these affixes by *meta rules*. We will distinguish two types of meta rules: domain *independent* (e.g. singular or plural form *affixes*), and domain *dependent* (e.g. generalizations). The production rules on the first level are either provided by the domain expert (verbalization rules) or deduced from the algebraic description of the object action involvement model.

2.7.3 Meta rules

The domain independent meta rules provide the domains for the affixes $\langle \Gamma \rangle$, $\langle$number$\rangle$ and $\langle$mode$\rangle$:

$\langle \Gamma \rangle$ :: $r \mid i$
$\langle$number$\rangle$ :: singular | plural
$\langle$mode$\rangle$ :: actual | structural

The role characterization is enclosed in the first meta rule. The second meta rule provides a mechanism to produce sentences using nouns in plural or singular form. An example of the usage of this meta rule is presented by the introduction of the paraphrasing rules for group types. The last meta rule is used to paraphrase either the actions (events) within a module or the structure of the module.

Furthermore, we assume the existence of a meta rule which has as domain the natural numbers. This meta rule provides the paraphrasing mechanism with the possibility to indicate a preferential treatment of the way object types should be paraphrased.

For each predicator a (domain dependent) meta rule is introduced stating all possible actors of that predicator. The set of possible *actors* for a predicator p, denoted as $\mathsf{Actors}(p)$, is defined as the set of all object types which can be involved in the corresponding action type ($\mathsf{Action}(p)$), i.e.

$$\mathsf{Actors}(p) = \{x \mid \mathsf{Binds}(x, \mathsf{ANm}(\mathsf{Action}(p)), \mathsf{Action}(p)\}$$

In the context of figure 1 and example 1 predicator p_4, which is contained in action type *to produce*, has as possible actors: *Producer*, *Band*, and *Person*.

For each predicator the following meta rule is introduced:

$\langle p \rangle$ :: $|_{x \in \mathsf{Actors}(p)}\ \mathsf{FNm}(x)$

For our example, this results in:

$\langle p_4 \rangle$ :: Producer | Band | Person

2.7.4 Object types

The verbalization rules for object types form the starting point for constructing their paraphrasing rules. For object type x the following paraphrasing rules are introduced:

$\langle \mathsf{FNm}(x) \rangle$: $|_{i=1}^{n(x)}\ \langle \mathsf{FNm}(x)\,[i] \rangle$
$\langle \mathsf{FNm}(x)\,[i] \rangle$: v_i

where v_i is the i-th verbalization rule of x and $n(x) = |\mathsf{IVerbs}(x)|$ the number of verbalization rules for x. Note that if x has a single verbalization rule v, this can be simplified to:

$\langle \mathsf{FNm}(x) \rangle$: v

The following rules are generated from our running example:

$\langle$Person$\rangle$: "Person"
$\langle$Band$\rangle$: $\langle$Band$\,[1]\rangle$ | $\langle$Band$\,[2]\rangle$
$\langle$Band$\,[1]\rangle$: "Band"
$\langle$Band$\,[2]\rangle$: "Pop group"

In order to obtain better readable sentences, plural form formulations of object types may be required. Since the name of an object type is assumed to be singular, the domain expert has to provide the corresponding plural form whenever necessary. Alternatively, one could use a lexical database to deliver the necessary morphosytactic information ([Hop97]). In general this leads to the following paraphrasing rules:

$\langle \mathsf{FNm}(x)\,[\text{Number}]\rangle$: $\langle \mathsf{FNm}(x)\,[\text{Singular}]\rangle \mid \langle \mathsf{FNm}(x)\,[\text{Plural}]\rangle$

For example if we consider an object type *Band* to be a group of *Person*s the following paraphrasing rules are added to the information grammar:

$\langle$Person [Number]$\rangle$: $\langle$Person [Singular]$\rangle \mid \langle$Person [Plural]$\rangle$

$\langle$Person [Singular]$\rangle$: $\langle$Person$\rangle$
$\langle$Person [Plural]$\rangle$: "Persons"

2.7.5 Action types

Let action type a consist of predicators $p_1, \ldots, p_k$. The paraphrasing rules for action type a are constructed in a number of steps. The first rule is:

$$\langle \mathsf{FNm}(a)\rangle : \begin{cases} \langle \mathsf{FNm}(a)\,[r]\rangle \mid \langle \mathsf{FNm}(a)\,[i]\rangle & \text{if } \exists_{p,q\in a}\,[\chi(p) = r \wedge \chi(q) = i] \\ \langle \mathsf{FNm}(a)\,[r]\rangle & \text{if } \forall_{p\in a}\,[\chi(p) = r] \\ \langle \mathsf{FNm}(a)\,[i]\rangle & \text{otherwise} \end{cases}$$

This can be further elaborated by extending these rules with a choice of actors for each role in action type a:

$\langle \mathsf{FNm}(a)\,[\Gamma]\rangle$: $\langle \mathsf{FNm}(a)\,[\Gamma, \mathsf{FNm}_{(x_1)}, \ldots, \mathsf{FNm}_{(x_k)}]\rangle$

where each $x_i \in \mathsf{Actors}(p_i)$. Each such paraphrasing rule is actualized by employing a verbalization rule from $\mathsf{AVerbs}(a)$. Let $\omega_0\,\langle p_{n_1}\rangle\,\omega_1 \ldots \omega_{l-1}\,\langle p_{n_l}\rangle\,\omega_l$ be such a verbalization rule, then this leads to the following paraphrasing rule:

$\langle \mathsf{FNm}(a)\,[\Gamma,\, \mathsf{FNm}_{(x_1)},\ldots,\mathsf{FNm}_{(x_k)}]\rangle$: $\omega_0\,\langle p_{n_1}\,[\mathsf{FNm}_{(x_{n_1})}]\rangle\,\omega_1 \ldots \omega_{l-1}\,\langle p_{n_l}\,[\mathsf{FNm}_{(x_{n_l})}]\rangle\,\omega_l$

Next we work out the predicator variants. For predicator p hooked to x the following rule is generated:

$\langle p\,[\mathsf{FNm}_{(x)}]\rangle$: $\langle \mathsf{FNm}(x)\rangle$

Note that the system analyst may decide for a special naming strategy, for example:

$\langle p\,[\mathsf{FNm}_{(x)}]\rangle$: $\langle \mathsf{FNm}(x)\,[\text{Plural}]\rangle$

Example 3 As an example of a paraphrasing rule for action types, we consider action type *to produce*.

$\langle$to produce$\rangle$: $\langle$to produce $[r]\rangle \mid \langle$to produce $[i]\rangle$
$\langle$to produce $[r]\rangle$: $\langle$to produce [r, Recording, Producer]$\rangle \mid$
$\langle$to produce [r, Recording, Band]$\rangle \mid$
$\langle$to produce [r, Recording, Person]$\rangle$
$\langle$to produce $[i]\rangle$: $\langle$to produce [i, Recording, Producer]$\rangle \mid$
$\langle$to produce [i, Recording, Band]$\rangle \mid$
$\langle$to produce [i, Recording, Person]$\rangle$

$\langle$to produce [r, Recording, Producer]$\rangle$: $\langle p_4\,[\text{Producer}]\rangle$ produces $\langle p_3\,[\text{Recording}]\rangle$
$\langle$to produce [r, Recording, Band]$\rangle$: $\langle p_4\,[\text{Band}]\rangle$ produces $\langle p_3\,[\text{Recording}]\rangle$
$\langle$to produce [r, Recording, Person]$\rangle$: $\langle p_4\,[\text{Person}]\rangle$ produces $\langle p_3\,[\text{Recording}]\rangle$

$\langle$to produce [i, Recording, Producer]$\rangle$: $\langle p_3\,[\text{Recording}]\rangle$ is produced by $\langle p_4\,[\text{Producer}]\rangle$
$\langle$to produce [i, Recording, Band]$\rangle$: $\langle p_3\,[\text{Recording}]\rangle$ is produced by $\langle p_4\,[\text{Band}]\rangle$
$\langle$to produce [i, Recording, Person]$\rangle$: $\langle p_3\,[\text{Recording}]\rangle$ is produced by $\langle p_4\,[\text{Person}]\rangle$

$\langle p_4\,[\text{Producer}]\rangle$: $\langle$Producer$\rangle$
$\langle p_4\,[\text{Band}]\rangle$: $\langle$Band$\rangle$
$\langle p_4\,[\text{Person}]\rangle$: $\langle$Person$\rangle$
$\langle p_3\,[\text{Recording}]\rangle$: $\langle$Recording$\rangle$

2.7.6 Paraphrasing structure

The subtyping structure is paraphrased via a rule which describes the generalization structure, and a rule for the specialization structure. Suppose the generalization structure of a module m consists of the generalizations x_1 gen $y_1, \ldots, x_k$ gen y_k. Then this is described by the rule:

$$\begin{aligned}\langle \text{Gen} \odot \ \mathsf{FNm}(m)\rangle : \ & \langle \text{Gen} \odot \ \mathsf{FNm}(x_1) \odot \ \mathsf{FNm}(y_1)\rangle \\ & \vdots \\ & \langle \text{Gen} \odot \ \mathsf{FNm}(x_k) \odot \ \mathsf{FNm}(y_k)\rangle\end{aligned}$$

with for each each generalization relation x gen y:

$$\langle \text{Gen} \odot \ \mathsf{FNm}(x) \odot \ \mathsf{FNm}(y)\rangle : \langle \mathsf{FNm}(y)\rangle \ \text{“is a”} \ \langle \mathsf{FNm}(x)\rangle \ \text{“.”}$$

The specialization structure of a module m is handled analogously.

Group types of a module m are paraphrased via a special rule with $\langle \text{Group} \odot \ \mathsf{FNm}(m)\rangle$ as left hand side nonterminal. Suppose module m has groups $g_1, \ldots, g_k$. Then the group structures are described in the following way:

$$\langle \text{Group} \odot \ \mathsf{FNm}(m)\rangle : \langle \text{Group} \odot \ \mathsf{FNm}(g_1)\rangle \ldots \langle \text{Group} \odot \ \mathsf{FNm}(g_k)\rangle$$

with for each group type g:

$$\langle \text{Group} \odot \ \mathsf{FNm}(g)\rangle : \langle \mathsf{FNm}(g)\rangle \ \text{“is a group of”} \ \langle \mathsf{FNm}(\mathsf{Elt}(g))\,{}_{[\text{plur}]}\rangle \ \text{“.”}$$

Paraphrasing of sequence types is almost the same as for group types. Suppose module type m has sequence types $s_1, \ldots, s_k$. This results in the following paraphrasing rule:

$$\langle \text{Seq} \odot \ \mathsf{FNm}(m)\rangle : \langle \text{Seq} \odot \ \mathsf{FNm}(s_1)\rangle \ldots \langle \text{Seq} \odot \ \mathsf{FNm}(s_k)\rangle$$

with for each sequence type s:

$$\langle \text{Seq} \odot \ \mathsf{FNm}(s)\rangle : \langle \mathsf{FNm}(s)\rangle \ \text{“is a sequence of”} \ \langle \mathsf{FNm}(\mathsf{Elt}(s))\,{}_{[\text{plur}]}\rangle \ \text{“.”}$$

The last decomposable object types to consider are the module types. Suppose module type m consists of object types $x_1, \ldots x_n$, such that $x_i \in \mathsf{Locals}(m)$. The paraphrasing rule for module type m then is:

$$\begin{aligned}\langle \text{Mod} \odot \ \mathsf{FNm}(m)\rangle : \ & \langle \mathsf{FNm}(m)\rangle \ \text{“is a composition of”} \\ & \mathsf{FNm}(x_1) \text{“,”} \ldots \text{“,”} \ \mathsf{FNm}(x_n) \text{“.”}\end{aligned}$$

3 Object property model

In this section syntax and semantics of the object property model are discussed. Furthermore, it is indicated how to obtain that part of the information grammar which is described by the object property model. This part of the information grammar is also validated by a paraphrasing mechanism.

3.1 Informal Introduction

Properties bridge the gap between abstract (elementary or composed) object types and concrete (label) types. The properties of an object type form a *state record* which reports about the current state of an object. Obviously the object types in figure 1 have properties. For example, instances of object type *Band* have a name. This property (of having a name) is initialized whenever a band is set up. During the life of a band its band name may be inspected via retrieval actions. In this particular example there are no explicit update action types for band names, except for its initialization (which can be seen as an update). Figure 3 shows an example of an object property model where *(Name)* denotes a concrete object type. The *trigger* relation, which is a relation between an action type and an update action type, states that action type *to set up* causes an update of property *Band name*. The dashed box specifies which update action type and retrieval action type belong to the property *Band name*. The shorthand graphical representation for the object property model for figure 3 is given in figure 4.

The object property models of figures 3 and 4 requires the following additional constraints:

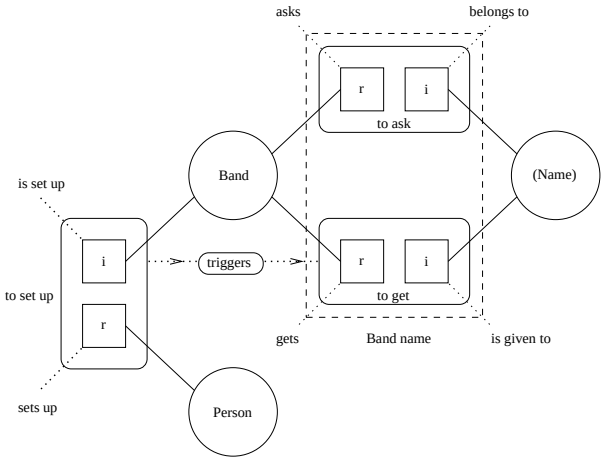

Figure 3: An object property model

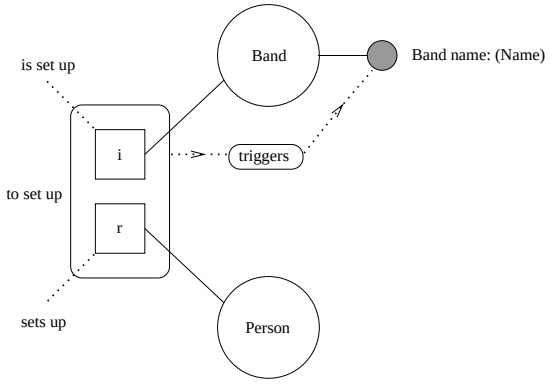

Figure 4: Shorthand notation

1. When a band is set up it also gets a band name.
2. A band name belongs to a band only when it is given to that band.

This can be generalized as follows:

1. Modifications are a side-effect of some action type. They can thus be represented by a trigger mechanism.
2. A property can only be retrieved after it has been set.

The property model describes what properties object types have and which action types trigger their modifications. The execution order of action types (and thus also retrieval- and update action types) are considered in the object life model.

3.2 Syntax

In the object action involvement model only *abstract* object types are considered. In this section the set of object types $\mathcal{O}$ is extended with a set $\mathcal{L}$ of *concrete* object types, the so-called *label types*, i.e. $\mathcal{L} \subseteq \mathcal{O}$. They can, in contrast with abstract object types, be represented on a communication medium. The extension of the set of object types with the set of label types needs a number of refinements on the formalization of the object action involvement model, if a strict separation between abstract object types and concrete object types is required:

1. Label types are not involved in a subtyping hierarchy, i.e. $\mathsf{spec} \cup \mathsf{gen} \subseteq \mathcal{E} \times \mathcal{O} \setminus \mathcal{L}$.
2. The strict separation between abstract and concrete object types prohibits label types to act as an element type, i.e. $\mathsf{Elt}(x) \notin \mathcal{L}$.

Formally, the *object property model* is a structure $\mathcal{OP}$ consisting of the following basic aspects:

1. A set $\mathcal{B}$ of properties. A property *bridges* the gap between object types and label types. With each property p, a *retrieval* and *update* action type is associated ($\mathsf{Retrieve}(p)$ and $\mathsf{Update}(p)$ respectively).

   Retrieval and update operations for properties are binary action types. We will use the abbreviation $r \in \mathsf{Bridge}(p)$ for $\mathsf{Action}(r) = \mathsf{Retrieve}(p) \vee \mathsf{Action}(r) = \mathsf{Update}(p)$. The predicate $\mathsf{Object}(p) = x \equiv x \notin \mathcal{L} \wedge \exists_{r \in \mathsf{Bridge}(p)} [\mathsf{Actor}(r) = x]$ extracts the associated object type, while the associated label type is obtained by $\mathsf{Label}(p) = x \equiv x \in \mathcal{L} \wedge \exists_{r \in \mathsf{Bridge}(p)} [\mathsf{Actor}(r) = x]$.
2. A relation $\mathsf{Triggers}$ between action types and update action types for properties, where $\mathsf{Triggers}(a, \mathsf{Update}(p))$ states that action type a *triggers* action type $\mathsf{Update}(p)$.
3. A relation IsDen between object types and *properties*. The expression $\mathsf{IsDen}(x, p)$ states that property p should be used to get (one of) the attribute(s) denoting instances of object type x. We introduce $\mathsf{Den}(x)) = \{p \mid \mathsf{IsDen}(x, p)\}$ as the set of properties used to denote the instances of object type x.

The object property model has to fulfill the following rules:

Property rules	OP-1	*(binary bridges)*	$\mathsf{Binary}(\mathsf{Retrieve}(p)) \wedge \mathsf{Binary}(\mathsf{Update}(p))$
	OP-2	*(unique property owner)*	$\exists!_x [\mathsf{Object}(p) = x]$
	OP-3	*(unique property domain)*	$\exists!_x [\mathsf{Label}(p) = x]$
	OP-4	*(label involvement)*	$\mathsf{Actor}(r) \in \mathcal{L} \Rightarrow \exists_p [r \in \mathsf{Bridge}(p)]$
	OP-5	*(passive label involvement)*	$\mathsf{Actor}(r) \in \mathcal{L} \Rightarrow \chi(r) = i$
	OP-6	*(property initialization)*	$\exists_a [\mathsf{Triggers}(a, \mathsf{Update}(p))]$
Denoting instances	OP-7	*(denotational correctness)*	$\mathsf{IsDen}(x, p) \Rightarrow \exists_y [\mathsf{Inherits}(x, y) \wedge \mathsf{Object}(p) = y]$
	OP-8	*(denotable objects)*	$\mathsf{Den}(x) \neq \emptyset$

Example 4 Figure 5 shows a graphical representation which includes the following algebraic description (with respect to the object property model):

$\mathcal{L}$	$\mathcal{B}$	Retrieve	Update	Triggers
L	p	$\mathsf{Retrieve}(p)$	$\mathsf{Update}(p)$	$\mathsf{Triggers}(a_4, \mathsf{Update}(p))$

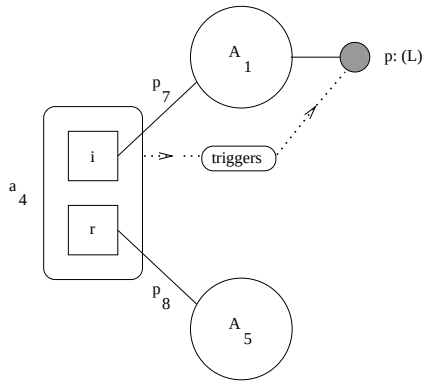

Figure 5: Object property model

3.3 Populating the object property model

The property model is an extension to the object action involvement model. It provides more structure for the way that label types are involved in this model. In order to populate label types the set Ω of object identifiers is extended with concrete values (labels). As a result, the population mechanism of the object action involvement model is still valid.

3.4 Naming and verbalization rules

The naming and verbalization functions (INm, ANm, IVerbs, AVerbs) for object types and action types of the object action involvement model did not assume label types, retrieval action types, and update action types to have a name nor verbalization rules. In this section, names and verbalization rules for label types will be assumed (and thus INm and IVerbs can be applied to label types), and standard names and verbalization rules for retrieval and update action types are introduced.

Each property p is also assigned a name and set of verbalization rules using $\mathsf{BNm}(p)$ and $\mathsf{BVerbs}(p)$, respectively. Properties may have the same name as long as they can be distinguished by the family names of the owners of the property:

[**N-4**] *(property naming)* $\quad \mathsf{BNm}(p) = \mathsf{BNm}(q) \wedge p \neq q \Rightarrow \mathsf{Object}(p) \neq \mathsf{Object}(q)$

We can uniquely denote each property as follows:

$$\mathsf{FNm}(p) = \begin{cases} \mathsf{BNm}(p) & \text{if } \mathsf{BNm}(p) \text{ is unique} \\ \mathsf{FNm}(\mathsf{Object}(p)) \odot \mathsf{BNm}(p) & \text{otherwise} \end{cases}$$

Example 5 In the context of example 4, $\mathsf{BNm}(p)$ can be chosen *Band name* whereas $\mathsf{BVerbs}(p)$ can be the set of verbalization rules $\{$"Band name", "Pop group name"$\}$.

Names and verbalization rules for retrieval action types and update action types are generated as follows. Let p be a property with $\mathsf{Retrieve}(p) = \{r_1, r_2\}$ and $\mathsf{Update}(p) = \{u_1, u_2\}$, such that $\mathsf{Actor}(r_1)$ and $\mathsf{Actor}(s_1)$ are (abstract) object types. This leads to:

$$\begin{aligned}
\mathsf{ANm}(\mathsf{Retrieve}(p)) &= \textit{to ask} \odot \mathsf{FNm}(p) \\
\mathsf{ANm}(\mathsf{Update}(p)) &= \textit{to get} \odot \mathsf{FNm}(p) \\
\mathsf{AVerbs}(\mathsf{Retrieve}(p)) &= \{\langle r_1 \rangle \text{ "asks" } \langle \mathsf{FNm}(p) \rangle, \\
&\quad \langle \mathsf{FNm}(p) \rangle \text{ "belongs to" } \langle r_1 \rangle\} \\
\mathsf{AVerbs}(\mathsf{Update}(p)) &= \{\langle u_1 \rangle \text{ "gets" } \langle \mathsf{FNm}(p) \rangle, \\
&\quad \langle \mathsf{FNm}(p) \rangle \text{ "is given to" } \langle u_1 \rangle\}
\end{aligned}$$

3.5 Inheritance of properties

As stated in section 2.3 object types can inherit besides action types also properties. In Section 2.6 the semantics of the inheritance relation with respect the inheritance of action types has been discussed. The semantics of the inheritance of properties is analogous and is based on the observation that properties can be modeled via their retrieval and update action types.

The definitions of Has and Binds have to be defined on properties in order to apply the axioms for binding. Object type x has a property p with name $\mathsf{BNm}(p)$, denoted as $\mathsf{Has}(x, n, p)$, where:

$$\mathsf{Has}(x, n, p) \equiv \mathsf{Has}(x, \mathsf{ANm}(\mathsf{Update}(p)), \mathsf{Update}(p)) \wedge \mathsf{BNm}(p) = n$$

Object type x binds name n to property p, denoted as $\mathsf{Binds}(x, n, p)$, where:

$$\mathsf{Binds}(x, n, p) \equiv \mathsf{Binds}(x, \mathsf{ANm}(\mathsf{Update}(p)), \mathsf{Update}(p)) \wedge \mathsf{BNm}(p) = n$$

Lemma 3 $x \,\mathsf{spec}\, y \vee y \,\mathsf{gen}\, x \Rightarrow \forall_n [\exists_p [\mathsf{Binds}(y, n, p)] \Rightarrow \exists_q [\mathsf{Binds}(x, n, q)]]$

Proof:
This lemma is only proven for the case $x \,\mathsf{spec}\, y$. The proof for $y \,\mathsf{gen}\, x$ is analogous.

Suppose $x \,\mathsf{spec}\, y$. Furthermore, suppose object type y binds name n to property p. Then we have to prove that object type x binds the same name to a property q. We distinguish two cases:

1. Suppose $\mathsf{Has}(x, n, q)$. As a result of axiom B-2 we know that $\mathsf{Binds}(x, n, q)$.
2. Suppose $\neg\, \mathsf{Has}(x, n, q)$. $x \,\mathsf{spec}\, y$ and axiom I-2 leads to $\mathsf{Inherits}(x, y)$. Via axiom B-3 we can conclude $\mathsf{Binds}(x, n, q)$.

$\diamond$

The binding relation for properties states what property names can be used in the context of an object type, and the properties corresponding to such names. The *state record* of an object type is introduced as the set of all properties which can be bound by that object type:

$$\mathsf{StatRec}(x) = \{p \in \mathcal{B} \mid \exists_n [\mathsf{Binds}(x, n, p)]\}$$

Next we consider binding of properties at run time. The predicates IsAwareOf and IsDefiner are reused. The expression $\mathsf{IsAwareOf}(\mathsf{L}, i, \mathsf{ANm}(\mathsf{Update}(p)), x)$ answers the question whether instance i, in the context of object type x, after logbook L is aware of property p. The predicate $\mathsf{IsDefiner}(\mathsf{L}, i, \mathsf{ANm}(\mathsf{Update}(p)), x)$ states whether object type x, which is a proper type of instance i after logbook L, has introduced this property p.

3.6 Extended paraphrasing mechanism

In this section the paraphrasing mechanism of the object action involvement model is extended with rules for paraphrasing properties. Furthermore, a way to paraphrase instances will be sketched at the end of this section.

3.6.1 Meta rules

The domain independent affix rule ⟨mode⟩ is extended with trigger for paraphrasing trigger relations, i.e.

⟨mode⟩ :: actual | structural | trigger

In order to paraphrase properties another domain dependent meta rule is introduced which summarizes all properties within a module m:

$\langle \text{All props} \odot \mathsf{FNm}(m) \rangle :: |_{p \in \mathcal{B}_m} \mathsf{FNm}(p)$

where $\mathcal{B}_m$ is the set of all properties which belong to object types within a module m, i.e.

$$\mathcal{B}_m = \{p \mid \mathsf{Retrieve}(p) \in \mathsf{Locals}(m) \wedge \mathsf{Update}(p) \in \mathsf{Locals}(m)\}$$

Example 6 Suppose object type *Recording* of figure 1 has properties *Studio number* and *Tape number*. This leads to the following meta rule (where the property of object type *Band* is also included):

$\langle \text{All props} \odot \mathsf{FNm}(m) \rangle$ :: Band name | Studio number | Tape number

3.6.2 Property triggers

As stated before, properties can be modeled using action types. For action types a mechanism to achieve their paraphrasing rules is already presented in section 2.7.5. This mechanism can be applied for the retrieval and update action types of property p, resulting in rules for $\langle \mathsf{FNm}(\mathsf{Update}(p)) \rangle$ and $\langle \mathsf{FNm}(\mathsf{Retrieve}(p)) \rangle$.

However, the resulting set of paraphrasing rules is not sufficient for paraphrasing triggers. Since triggers form a relevant part of the object property model the set of paraphrasing rules for action types is extended with rules for paraphrasing triggers. Let action type a trigger the update action type b for property p, i.e. $\mathsf{Triggers}(a, b)$. This leads to the following paraphrasing rule:

$$\langle \mathsf{FNm}(b)\,{}_{[\mathsf{FNm}(p)]} \rangle : \text{“when”}\ \langle \mathsf{FNm}(a) \rangle\ \text{“then”}\ \langle \mathsf{FNm}(b) \rangle$$

which brings us back to the paraphrasing rules for action types presented in section 2.7.5.

Example 7 The paraphrasing rule for the trigger $\mathsf{Triggers}(a_4, \mathsf{Update}(p))$ in the context of figures 4 and 5 is:

$$\langle \text{to get}\,{}_{[\text{Band name}]} \rangle : \text{“when”}\ \langle \text{to set up} \rangle\ \text{“then”}\ \langle \text{to get} \odot \text{Band name} \rangle$$

3.6.3 Paraphrasing structure

Suppose module m has object types $x_1, \ldots, x_k$. Then the properties of these object types are described by the rule:

$$\langle \text{Props} \odot \mathsf{FNm}(m) \rangle : \langle \text{Props} \odot \mathsf{FNm}(x_1) \rangle \ldots \langle \text{Props} \odot \mathsf{FNm}(x_k) \rangle$$

The properties which belong to object type x of module m are paraphrased via a special rule. Let $\mathsf{StatRec}(x) = \{p_1, \ldots, p_l\}$. Then this is described by the rule:

$$\begin{aligned} \langle \text{Props} \odot \mathsf{FNm}(x) \rangle : \ & \langle \mathsf{FNm}(x) \rangle\ \text{“has”}\ \langle \mathsf{FNm}(p_1) \rangle\ \text{“denoted as”}\ \langle \mathsf{FNm}(\mathsf{Label}(p_1)) \rangle\ \text{“.”} \\ & \vdots \\ & \langle \mathsf{FNm}(x) \rangle\ \text{“has”}\ \langle \mathsf{FNm}(p_l) \rangle\ \text{“denoted as”}\ \langle \mathsf{FNm}(\mathsf{Label}(p_l)) \rangle\ \text{“.”} \end{aligned}$$

3.6.4 Instances

Properties are used to denote instances of object types. Using these instance denotations for the paraphrasing mechanism leads to instantiated sentences. Usually, not all properties will be required for instance denotation.

An instance of object type x is only denotable if it has a non-empty set of properties, i.e. $\mathsf{StatRec}(x) \neq \emptyset$.

The paraphrasing mechanism is equipped with one more domain independent meta rule:

$$\langle \text{sort} \rangle :: \text{typed} \mid \text{instantiated}$$

This affix rule is used to switch between sentences on a type level and sentences on an instance level. The sentence *Band records Song* is on a type level while the sentence *The Rolling Stones record Paint It Black* is an instantiated sentence.

The meta rule requires a number of extensions of the paraphrasing rules for abstract object types and action types. For each abstract object type x with verbalization rule v, its rule:

$$\langle \mathsf{FNm}(x) \rangle : v$$

is changed in:

$$\begin{aligned} &\langle \mathsf{FNm}(x)\,{}_{[\text{typed}]} \rangle : v \\ &\langle \mathsf{FNm}(x)\,{}_{[\text{instantiated}]} \rangle : \mid_{y \in \mathsf{L}(x)} \langle x : y \rangle \end{aligned}$$

where $\mathsf{L}(x)$ is a logbook of abstract object type x, i.e., an instantiation of x. An instance of an object type is a tuple consisting of a time stamp, and an object identifier (for abstract object types) or a label (for concrete object types). The projection functions π_{t} and π_{l} are used to extract the time component of an instance, and its object identifier or label, respectively. For readability, these operators are often omitted in the sequel.

Let $\mathsf{Den}(x) = \{p_1, \ldots, p_k\}$. Then for each instance y from $\mathsf{L}(x)$ the following rule is generated:

$$\langle x : y \rangle : \langle \mathsf{FNm}(x)\,[\text{typed}] \rangle \text{ “}(\mathsf{Denote}(y, p_1), \ldots, \mathsf{Denote}(y, p_k))\text{”}$$

where $\mathsf{Denote}(y, p_j)$ is the value of property p_j of instance y, i.e. $\mathsf{Denote}(y, p_j) = \pi_{\mathsf{l}}(l_i)$ with l_i a *valid* and *updated* denotation of property p_j for y. This valid and updated denotation is also referred to as the *current* denotation, $\mathsf{CurDen}(y, p, l)$ where

$$\mathsf{CurDen}(y, p, l) \equiv \begin{cases} y \in \mathsf{L}(\mathsf{Object}(p)) \wedge l \in \mathsf{L}(\mathsf{Label}(p)) \\ \wedge\ \exists_{v \in \mathsf{L}(\mathsf{Update}(p))} \left[\pi_{\mathsf{l}}(v) = \langle y, l \rangle \wedge \forall_{w \in \mathsf{Update}(p)} \left[\pi_{\mathsf{t}}(w) \leq \pi_{\mathsf{t}}(v) \right] \right] \end{cases}$$

The paraphrasing rules for action types are also modified. The affixes typed and instantiated are passed through all paraphrasing rules in the same way as the affixes for role characterization, i.e. the affixes r and i. Finally, the predicator variants, $(\langle p\,[\mathsf{FNm}(x)] \rangle)$ with $x \in \mathsf{Actors}(p)$, are also extended with the affixes typed and instantiated. As a result their paraphrasing rules are changed into:

$$\langle p\,[\mathsf{FNm}(x),\text{typed}] \rangle : \langle \mathsf{FNm}(x)\,[\text{typed}] \rangle$$
$$\langle p\,[\mathsf{FNm}(x),\text{instantiated}] \rangle : \langle \mathsf{FNm}(x)\,[\text{instantiated}] \rangle$$

Example 8 Suppose $y \in \mathsf{L}(x)$ with $\mathsf{Den}(x) = \{p\}$. Let $\mathsf{FNm}(x) = \textit{Band}$, $\mathsf{FNm}(p) = \textit{Band name}$, and $\mathsf{Denote}(y, p) =$ The Rolling Stones. This leads to the following revision and extension of the information grammar:

$$\langle \text{Band}\,[\text{typed}] \rangle : \langle \text{Band} \rangle$$
$$\langle \text{Band}\,[\text{instantiated}] \rangle : \langle x : y \rangle$$

$$\langle x : y \rangle : \langle \text{Band}\,[\text{typed}] \rangle \text{ “(The Rolling Stones)”}$$

$$\langle p_4\,[\text{Band,typed}] \rangle : \langle \text{Band}\,[\text{typed}] \rangle$$
$$\langle p_4\,[\text{Band,instantiated}] \rangle : \langle \text{Band}\,[\text{instantiated}] \rangle$$

For more information on paraphrasing instances, see [HPW97].

4 Object life model

In this section syntax and semantics of the object life model are discussed. Furthermore, it is indicated how to obtain that part of the information grammar which is described by the object life model. This part of the information grammar is also validated by a paraphrasing mechanism.

4.1 Informal Introduction

The object action involvement model states which object types are involved in what action type. However, this model does not state the order in which actions may be performed. The object property model provides only an ordering (the trigger relation) between action types, which cause an update of the state record of an object type, and their corresponding update action types.

Object life models elaborate on both the object action involvement model and the object property model. The goal of the object life model is to describe the so-called *course of life* for each object type. The course of life of an object type describes the set of possible sequences in which action types (including retrieval action types and update action types) may be invoked subsequently. Such a sequence is also referred to as a *(process) trace* and describes the behavior or *histories* of individual objects of that type. The first action in this sequence is called the *initialization* or *birth action*.

Reconsider the figures 1 and 3. The object action involvement model states that (1) a band is set up by a person, (2) a band can record songs, and (3) a band can produce recordings during its life. We also assume bands to be able to participate in the following actions: (1) a band is joined by a person, (2) a band is left by a person, and (3) a band disbands. Furthermore, the object property model states that a band has the property *Band name*.

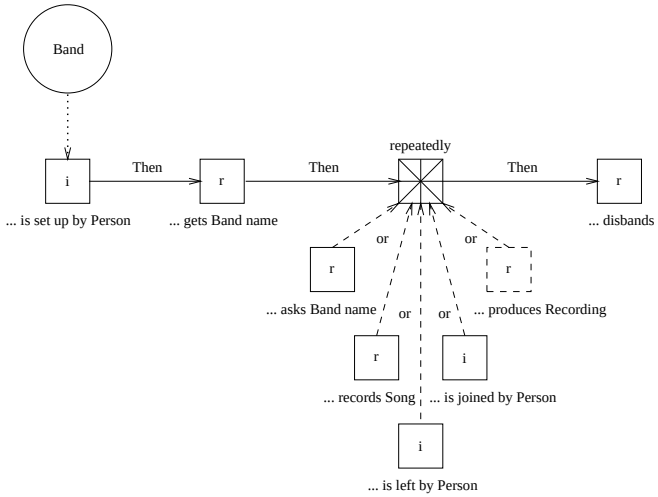

Figure 6: An object life model including retrieval actions

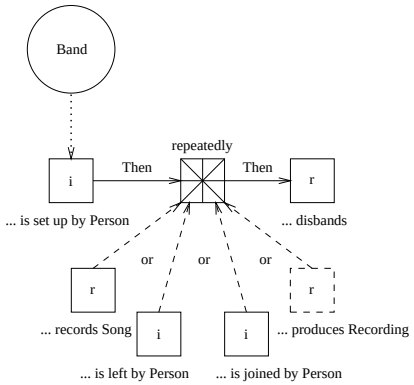

Figure 7: Shorthand notation

Figure 6 shows the life cycle of object type *Band*. This model expresses that an object of type band is born when it is set up by a person; after that the band gets its name (via a retrieval action type). Then a band can repeatedly (zero or more times): be joined or left by a person, record a song, produce a recording, and ask for its name. Finally, the life of a band is (explicitly) ended after it disbands. Note that additional constraints are necessary to exclude histories where:

1. a person leaves a band before joining that band,
2. a person joins a band already having that person as a member.

Such restrictions on histories can not be expressed by the object life model graphically and will be the subject of section 4.3.5.

The dotted arrow in figure 6 connects the corresponding object type with its course of life via its birth action. The course of life consists of generalized predicators which are mutually connected by either a solid arrow or a dashed arrow. The solid arrows denote the sequential order of generalized predicators whereas the dashed arrows denote a decomposition relation between generalized predicators and structural predicators. The predicators used in the object life model are those predicators which have already appeared in the object action involvement and object property model, and which have the corresponding object type as actor, i.e. an object type connected with

that predicator. Surrogates, denoted with dashed boxes, refer to predicators which appear in a generalization or specialization hierarchy. Furthermore, surrogates are used to express multiple occurrences of predicators in the course of life of an object type. The symbols which can be used for an object life model are summarized in appendix A.

Figure 7 is a shorthand notation for figure 6.

4.2 Syntax

The *object life model* extends the object action involvement model and the object property model with the notion of *generalized predicator*. The model adds the following aspects:

1. The set $\mathcal{C}$ of *generalized predicators* consists of the set $\mathcal{P}$ of predicators, and the following sets:
   (a) a set $\mathcal{D}$ of *surrogates* or *deputies*
   (b) a set $\mathcal{X}$ of *repetition predicators*.
   (c) a set $\mathcal{Y}$ of *choice predicators*.
   (d) a set $\mathcal{Z}$ of *merge predicators*.

   In the context of the object life model, *predicators* and *surrogates* are also referred to as *tasks* or *task predicators*. The set of tasks is denoted with $\mathcal{T}$. Generalized predicators are also referred to as *components*. Repetition predicators, choice predicators and merge predicators are also called *structural components* or *structural predicators*.
2. The function $\mathsf{Pred} : \mathcal{D} \to \mathcal{P}$ assigns to each surrogate the predicator from which the surrogate is a derative.
3. The partial function $\mathsf{Succ} : \mathcal{C} \mapsto \mathcal{C}$ describes the *sequential* order of components. Component $\mathsf{Succ}(c)$ is called the *successor* of component c.
4. The partial function $\mathsf{Decomp} : \mathcal{C} \mapsto \mathcal{C} \setminus \mathcal{T}$ provides initial components of structural components. The expression $\mathsf{Decomp}(c) = d$ states component c to be a *decomposition alternative* of structural component d.
5. The relation $\mathsf{Connect} \subseteq \mathcal{C} \times \mathcal{C}$ is the irreflexive transitive closure of both sequential order and decomposition. The direct connection $\mathsf{Connect}_1$ is obtained by: $c\,\mathsf{Connect}_1\,d \equiv c\,\mathsf{Connect}\,d \wedge \neg\exists_e\,[c\,\mathsf{Connect}\,e \wedge e\,\mathsf{Connect}\,d]$.

   It will be convenient to introduce the reflexive extension of the connection relation: $c\,\mathsf{Connect}^*\,d \equiv c = d \vee c\,\mathsf{Connect}\,d$.
6. The relation $\mathsf{HasInit} \subseteq \mathcal{O} \times \mathcal{C} \setminus \mathcal{Z}$ describes which components initialize object types.

   $x\,\mathsf{Owns}\,c$ is used as an abbreviation for $d\,\mathsf{Connect}^*\,c$ with $x\ \mathsf{HasInit}\ d$ for some d.

The following properties must hold for object life models:

General rules	OL-1	*(no cycles)*	$c\,\mathsf{Connect}\,d \Rightarrow \neg d\,\mathsf{Connect}\,c$
	OL-2	*(transitivity)*	$c\,\mathsf{Connect}\,d \wedge d\,\mathsf{Connect}\,e \Rightarrow c\,\mathsf{Connect}\,e$
	OL-3	*(connectivity induction)*	if $\forall_x\,[x\ \mathsf{HasInit}\ c \Rightarrow \phi(c)] \wedge \forall_d\,[d\,\mathsf{Connect}\,c \Rightarrow \phi(d)] \Rightarrow \phi(c)$ then $\forall_c\,[\phi(c)]$
	OL-4	*(connection)*	$c\,\mathsf{Connect}_1\,d \iff \mathsf{Succ}(c) = d \vee \mathsf{Decomp}(d) = c$
	OL-5	*(preconnector)*	$\neg x\ \mathsf{HasInit}\ c \Rightarrow \exists_d\,[d\,\mathsf{Connect}_1\,c]$
	OL-6	*(executability)*	$\exists!_x\,[x\,\mathsf{Owns}\,c]$
	OL-7	*(repetition decomposition)*	$c \in \mathcal{X} \wedge \mathsf{Decomp}(d) = c \wedge \mathsf{Decomp}(e) = c \Rightarrow d = e$
	OL-8	*(minimal tasks)*	$\mathsf{Actor}(t)\,\mathsf{Owns}\,t$
	OL-9	*(auto initialization)*	$\mathsf{Objectified}(a) \Rightarrow \forall_{p \in a}\exists_c\,[a\,\mathsf{HasInit}\,c \wedge \mathsf{Pred}(c) = p]$

A first observation is that two components are either in a decomposition relation or a successor relation with each other.

Lemma 4 $\mathsf{Decomp}(d) = c \Rightarrow \neg\,\mathsf{Succ}(d) = c$

Proof:
Suppose $\mathsf{Decomp}(d) = c$. Then we have to prove that $\neg\, \mathsf{Succ}(d) = c$. Assume that $\mathsf{Succ}(d) = c$. From $\mathsf{Decomp}(d) = c$ we can derive (via the definitions of Connect and $\mathsf{Connect}_1$) that $c\,\mathsf{Connect}\,d$. In the same way from $\mathsf{Succ}(d) = c$, $d\,\mathsf{Connect}\,c$ is derived. Using axioms 4.2 and 4.2 a cyclic structure is found, and so we can conclude $\neg\, \mathsf{Succ}(d) = c$. ◇

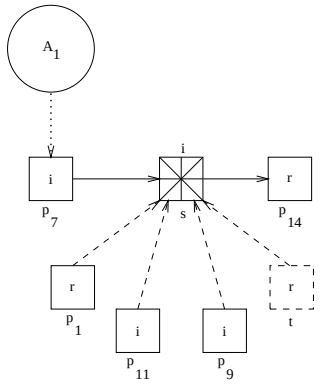

Figure 8: Object life model

Example 9 Consider the object life model displayed in figure 8 which is an abstract view on the object life model of *Band* displayed in figure 7. This model is described by:

$\mathcal{T}$	$\mathcal{D}$	$\mathcal{X}$	$\mathcal{Y}$	$\mathcal{Z}$
p_1	t	i	s	
p_7				
p_9				
p_{11}				
p_{14}				
t				

Succ	$\mathcal{C}$
p_7	i
i	p_{14}

Decomp	$\mathcal{C} \setminus \mathcal{T}$
s	i
p_1	s
p_9	s
p_{11}	s

HasInit
$A_1\,\mathsf{HasInit}\,p_7$

Pred	$\mathcal{P}$
t	p_4

4.3 Traces of object life model

An object life model can be transformed into a *process algebra* ([BW90]) expression describing a formal semantics for the course of life of its corresponding object type (see section 4.3.1). Expressions in process algebra can be rewritten, using a number of rewrite rules of [BW90], showing semantic equivalence of object life models (see section 4.3.2). Furthermore, an expression in process algebra can be seen as a generator of *traces* (section 4.3.3).

Each process algebra expression describes a narrowed view on the course of life of its corresponding object type, i.e. it does not necessarily describe parts of the courses of life of type related object types. The object trace space of an object type is derived from its own object life model and that of its type related object types. The object trace space of an object type describes *histories* for its objects. However, the object trace space can be defined too wide, i.e. it allows invalid histories. In section 4.3.4 the relation between object trace spaces and object histories are described leading to a number of domain independent restriction rules on object histories. Domain dependent restriction rules on object histories are discussed in section 4.3.5. Such restriction rules are also referred to as *property functions* or *constraints*. Finally, in section 4.3.6, a special property function is discussed which can compute the state of instances for a particular history.

4.3.1 Trace generator

The narrowed course of life of object type x can be defined as

$$E_x = \bigoplus_{x\ \mathsf{HasInit}\ c} E_c$$

The semantics of component c is denoted as E_c. This function will be recursively defined in the sequel of this section. For each component c a translation to process algebra equations is defined:

$$E_c = \begin{cases} \mathsf{Exec}(c) & \text{if } c \text{ has no successor} \\ \mathsf{Exec}(c) \odot E_{\mathsf{Succ}(c)} & \text{otherwise} \end{cases}$$

The execution of a single component is described by the function Exec. We distinguish the following cases:

1. The execution of task t consists of the execution of its corresponding action type in the role specified by predicator t. Thus:
$$\mathsf{Exec}(t) = t$$

2. The execution of a surrogate q coincides with that of its associated predicator:
$$\mathsf{Exec}(q) = \mathsf{Exec}(\mathsf{Pred}(q))$$

3. The execution of a choice s consists of the execution of one of its alternatives:
$$\mathsf{Exec}(s) = \bigoplus_{\mathsf{Decomp}(c)=s} E_c$$

4. A repetition symbol i leads to a repeated execution of its body. Let c be the unique decomposition of i, i.e. $\mathsf{Decomp}(c) = i$. Then:
$$\mathsf{Exec}(i) = \bigoplus_{n \geq 0} E_c^n = E_c^*$$
Note that $E_c^0 = \varepsilon$.

5. The last symbol to be defined is the merge symbol p:
$$\mathsf{Exec}(p) = \mathop{\|}_{\mathsf{Decomp}(c)=p} E_c$$

Example 10 The semantics in terms of example 9 is defined as follows:

$$\begin{array}{lcllcl} E_{A_1} & = & E_{p_7} & E_{p_7} & = & p_7 \odot E_i \\ E_i & = & E_s^* \odot E_{p_{14}} & E_s & = & E_{p_1} \oplus E_{p_9} \oplus E_{p_{11}} \oplus E_t \\ E_{p_1} & = & p_1 & E_{p_9} & = & p_9 \\ E_{p_{11}} & = & p_{11} & E_t & = & p_4 \\ E_{p_{14}} & = & p_{14} & & & \end{array}$$

The resulting process algebra equation for the course of life of object type *Band* is:

$$E_{A_1} = p_7 \odot (p_1 \oplus p_9 \oplus p_{11} \oplus p_4)^* \odot p_{14}$$

4.3.2 Life equivalence

An advantage of process algebra is that it contains rewrite rules which can be used to prove semantic equivalence. However, there are some courses of life that are observational the same which cannot be proven equivalent within the system of axioms. For example, the process algebra equation $X \odot (Y \oplus Z) = X \odot Y \oplus X \odot Z$ does not hold for process variables in general and is as a consequence not an axiom of the family of algebras. However, for concrete atomic processes the equation is correct. The theory of *bisimulation* ([BW90]) offers an extended notion of equivalence which can deal with such courses of life.

4.3.3 Object trace space

As shown above the semantics of the life of an object type x is expressed by an algebraic expression E_x. The set $\mathsf{Traces}(E_x)$, or $\mathsf{Traces}(x)$ for short, denotes all possible traces of tasks belonging to object type x and is called

the *object trace space* of x. The set of all possible traces of all tasks is denoted as $(\mathcal{T})^* \cup \{\lambda\}$ with λ denoting the empty trace. Axioms with respect to the object trace space are:

$$\begin{aligned} \mathsf{Traces}(\varepsilon) &= \{\lambda\} \\ \mathsf{Traces}(a) &= \{\lambda, a\} \\ \mathsf{Traces}(a \odot X) &= \{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \mathsf{Traces}(X)\} \\ \mathsf{Traces}(X \oplus Y) &= \mathsf{Traces}(X) \oplus \mathsf{Traces}(Y) \end{aligned}$$

Until now *recursive equations* and *recursive specifications* were not considered. An example of a recursive equation is $X = X \odot a$ and an example of a recursive specification is $E = \{X = X \odot a, Y = a \odot X\}$. The trace axioms are not sufficient to deal with recursive specifications. However, in [BW90] a number of definitions and axioms are presented which deal with a large number of recursive specifications.

4.3.4 Object histories

Let L be a logbook of some object action involvement model and object property model. Then this logbook can be transformed into a population of the meta-model of logbooks (see [BFW96]). This population provides a detailed description of the *history* of the UoD. For our purposes, an abstract view on this history is sufficient. This abstract view is obtained by cumulating all actions that have occurred:

$$\mathsf{History}(\mathsf{L}) = \bigcup_{a \in \mathcal{A}} \mathsf{L}(a)$$

Thus, generally, a history is a subset of $\mathsf{Time} \times \Omega$. The history of an instance v is obtained by considering only those actions in which v plays a role:

$$\mathsf{History}(v, \mathsf{L}) = \{\langle t, f\rangle \in \mathsf{History}(\mathsf{L}) \mid \exists_p [\pi_{\mathsf{i}}(f(p)) = v]\}$$

At each moment, an instance can be involved in at most one action:

[**L-1**] *(indivisible atomic actions)* $a, b \in \mathsf{History}(v, \mathsf{L}) \wedge \pi_{\mathsf{t}}(a) = \pi_{\mathsf{t}}(b) \Rightarrow a = b$

As a result, the history $\mathsf{History}(v, \mathsf{L})$ can be considered as a sequence of actions. Let $\mathsf{Head}(\mathsf{History}(v, \mathsf{L}))$ be the action first occurring in history $\mathsf{History}(v, \mathsf{L})$. Then the expression $\mathsf{Head}(\mathsf{History}(v, \mathsf{L}))$ must be the birth action of v. With each action in $\mathsf{History}(v, \mathsf{L})$ the function:

$$\mathsf{Preds} : \Omega \times \mathsf{History}(v, \mathsf{L}) \to \wp(\mathcal{P})$$

associates the predicator(s) corresponding to the role(s) being played by v in this action.

$$\mathsf{Preds}(v, \langle t, f\rangle) = \{p \in \mathcal{P} \mid \pi_{\mathsf{i}}(f(p)) = v\}$$

Note that $\mathsf{Preds}(v, \langle t, f\rangle)$ may contain more than one predicator. However, an object can not be multiply involved in its birth action.

[**L-2**] *(passive birth)* $|\mathsf{Preds}(\mathsf{Head}(\mathsf{History}(v, \mathsf{L})))| = 1$

The root type of v is defined as the actor of this single predicator. This object type is denoted as $\mathsf{Type}(v)$. During its life, an instance may become (temporarily) a member of specializations and generalizations.

Next we derive all sequences of predicators describing the life of v. This set of sequences is denoted as $\mathsf{Traces}(v, \mathsf{L})$ and equals the concatenation of the actions in the history of v ($\mathsf{Concat}(\mathsf{History}(v, l))$). The function Concat is defined by:

$$\mathsf{Concat}(\mathsf{L}) = \begin{cases} \lambda & \text{if } \mathsf{L} \text{ is empty} \\ \mathsf{Perms}(\mathsf{Preds}(v, a)) \cdot \mathsf{Concat}(\mathsf{Tail}(\mathsf{L})) & \text{if } a = \mathsf{Head}(\mathsf{L}) \end{cases}$$

where $\mathsf{Perms}(R)$ yields the set of all traces of predicators in R. The life of instance v should be in accordance with the object trace space from its root type, its specializations and its generalizations. Suppose instance v has root type $x = \mathsf{Type}(v)$.

[**L-3**] *(root life)* $\mathsf{Restrict}(\mathsf{Traces}(v, \mathsf{L}), x) \in \mathsf{Traces}(x)$

The function $\mathsf{Restrict}(\omega, x)$ restricts trace ω to object type x and is inductively defined by:

$$\begin{aligned}
\mathsf{Restrict}(\lambda, x) &= \lambda \\
\mathsf{Restrict}(p \cdot \omega, x) &= \begin{cases} p \cdot \mathsf{Restrict}(\omega, x) & \text{if } \mathsf{Actor}(p) = x \vee \mathsf{Actor}(\mathsf{Pred}(p)) = x \\ \mathsf{Restrict}(\omega, x) & \text{otherwise} \end{cases}
\end{aligned}$$

Instances of object type x can be a member of each specialization of x. The next rule describes the life from the perspective of a specialization.

[L-4] *(specialized life)* $\quad y \,\mathsf{spec}\, x \Rightarrow \mathsf{Restrict}(\mathsf{Traces}(v, \mathsf{L}), y) \in \mathsf{Traces}^*(y)$

The following rule is the analogon for generalization.

[L-5] *(generalized life)* $\quad y \,\mathsf{gen}\, x \Rightarrow \mathsf{Restrict}(\mathsf{Traces}(v, \mathsf{L}), y) \in \mathsf{Traces}^*(y)$

Furthermore, the rules for being member of subtypes and generalizations must be obeyed. Each specialization and each generalization has associated a predicate stating its extensionality. Let $\mathsf{SubRule}_x$ be the membership predicate for a specialized object type x. Then instance v is a member of x iff the expression $\mathsf{SubRule}_x(v, \mathsf{L})$ is true.

[L-6] *(valid membership (1))*
$\langle t, f\rangle \in \mathsf{History}(v, \mathsf{L}) \wedge p \in \mathsf{Preds}(v, \langle t, f\rangle) \wedge \mathsf{IsSpec}(\mathsf{Actor}(p)) \Rightarrow \mathsf{SubRule}_{\mathsf{Actor}(p)}(v, \mathsf{History}(t, \mathsf{L}))$

where $\mathsf{History}(t, \mathsf{L})$ provides a snapshot of L at time t. An analogous requirement should hold for generalized object types.

[L-7] *(valid membership (2))*
$\langle t, f\rangle \in \mathsf{History}(v, \mathsf{L}) \wedge p \in \mathsf{Preds}(v, \langle t, f\rangle) \wedge \mathsf{IsGen}(\mathsf{Actor}(p)) \Rightarrow \mathsf{GenRule}_{\mathsf{Actor}(p)}(v, \mathsf{History}(t, \mathsf{L}))$

4.3.5 Restricting object histories

The object life model provides a framework for histories of instances of each object type. However, this demarcation may be too wide. Further restrictions, also called *history constraints*, can be described in terms of the properties of objects. Such restrictions may involve objects of several types. Restrictions on histories can be defined with so-called *property functions*.

Property functions assign values to histories. These values can be taken from Ω. The value $\perp$ (undefined) is used to make property functions total functions.

Example 11 Reconsider the course of life of a *Band* (see figure 7). Suppose $b \in \mathsf{L}(\text{Band})$ and $p, q \in \mathsf{L}(\text{Person})$. The following histories for bands obviously are not valid[2]:

h_1: *to set up*(b, p)*; to leave*(b, q)
h_2: *to set up*(b, p)*; to join*(b, p)*; to join*(b, p)

The problem with these histories is that a person can only leave a band after joining this band. This constraint is referred to as BM1. Furthermore, only new members can join a band. This constraint is referred to as BM2. In order to express these constraints, we introduce the property function

$$\mathsf{BandMembers} : \{\mathsf{History}(b, \mathsf{L}) \mid \mathsf{L} \text{ a logbook}\} \to \Omega$$

as follows:

$$\begin{aligned}
\mathsf{BandMembers}(\langle\rangle) &= \emptyset \\
\mathsf{BandMembers}(\textit{to set up}(b,p)) &= \{p\} \\
\mathsf{BandMembers}(h; \textit{to join}(b,p)) &= \begin{cases} \perp & \text{if } p \in \mathsf{BandMembers}(h) \\ \mathsf{BandMembers}(h) \cup \{p\} & \text{otherwise} \end{cases} \\
\mathsf{BandMembers}(h; \textit{to leave}(b,p)) &= \begin{cases} \perp & \text{if } p \notin \mathsf{BandMembers}(h) \\ \mathsf{BandMembers}(h) - \{p\} & \text{otherwise} \end{cases} \\
\mathsf{BandMembers}(h; \textit{to disband}(b)) &= \emptyset
\end{aligned}$$

[2]For readability, time stamps are omitted, and action type names are added; furthermore we denote histories in a sequence notation as in programming languages.

where $\langle\rangle$ denotes the empty history, and h stands for any valid object history of *Band b*. This leads to the following formulation for both BM1 and BM2:

$$\mathsf{BandMembers}(h) \neq \bot$$

The verbalization of such constraints requires a language to formulate constraints. For this purpose we use the query and constraint language *Elisa-D* ([PW95b]), which is suitable for restricting information grammars.

```
LET is_member_of BE
  ALL Person joins Band EVER MINUS ALL Person leaves Band EVER
```

The construction `LET ... BE ...` is a mechanism to extend the information grammar with new rules for forming sentences. The construction `ALL P EVER` gathers all instances of information descriptor `P` from the past. The constraints BM1 and BM2 can be expressed as:

```
CONSTRAINT
  BM1: NOT Person leaves Band BUT NOT is_member-of THAT Band
  BM2: NOT Person joins Band AND ALSO is_member-of THAT Band
```

4.3.6 Evaluating state records

In section 3.5 the concept of state record has been introduced as the set of all properties of an object type. In this section we show how values of properties of instances can be computed from histories.

The *state record* of an instance v after logbook L contains those properties which are introduced by the types of v. The state of an instance v at the end of a logbook L can be defined by the property function:

$$\mathsf{State}(v, \mathsf{L}) : \{p \mid \exists_x [\mathsf{IsDefiner}(\mathsf{L}, v, \mathsf{ANm}(\mathsf{Update}(p)), x)]\} \rightarrow \Omega$$

with $\mathsf{State}(v, \mathsf{L})(p) = \mathsf{Eval}(\mathsf{History}(v, \mathsf{L}), p)$, where the function Eval is defined in the following way:

$$\begin{array}{rcll} \mathsf{Eval}(\langle\rangle, p) & = & \bot & \\ \mathsf{Eval}(h; \langle t, f\rangle, p) & = & \mathsf{Eval}(h, p) & \text{if } \langle t, f\rangle \notin \mathsf{L}(\mathsf{Update}(p)) \\ \mathsf{Eval}(h; \langle t, \langle v, m\rangle\rangle, p) & = & m & \text{if } \langle t, \langle v, m\rangle\rangle \in \mathsf{L}(\mathsf{Update}(p)) \end{array}$$

4.4 Extended paraphrasing mechanism

Paraphrasing the object life model is not straightforward as it requires not only a single sentence. In the rest of this section we give some rules of the thumb. Paraphrasing object life models is still a topic for research (see also [Dal95]).

4.4.1 Meta rules

The final driver for executing the information grammar should be able to focus on paraphrasing the lifes of its object types. Therefore, the domain independent affix rule ⟨mode⟩ is extended with an affix life, i.e.

⟨mode⟩ :: actual | structural | trigger | life

4.4.2 Course of life

The life of an object type starts with its initialization, followed by a sequence of components. Suppose object type x starts its life with component c. This leads to the following grammar rule:

⟨life of x⟩ : ⟨seq c⟩

A sequence of components is elaborated as follows:

$$\langle \text{seq } c \rangle : \begin{cases} \langle c \rangle \text{ “.” “Then” } \langle \text{seq } d \rangle & \text{if } c \,\mathsf{Connect}_1\, d \\ \langle c \rangle \text{ “.”} & \text{otherwise} \end{cases}$$

Before paraphrasing rules for components are introduced an auxiliary predicate InDisplay is defined. $\mathsf{InDisplay}(x, {}_{[D]})$ states whether affix (non)terminal x occurs in the display D of nonterminal $\langle \mathsf{FNm}(a)\,{}_{[D]} \rangle$. For example, predicate $\mathsf{InDisplay}(r, {}_{[\text{r,Recording,Producer}]})$ yields true.

We distinguish the following cases for component c:

1. Suppose c is a predicator, i.e. $c \in \mathcal{P}$. If there are no other object types using a surrogate of c, $\neg\exists_{y,d}\,[x \neq y \wedge \mathsf{Pred}(d) = c \wedge y\,\mathsf{Owns}\,d]$, the following rule is added:

   $\langle c \rangle : \langle \mathsf{FNm}(\mathsf{Action}(c)) \rangle$

   In case there are surrogates of c used in the course of life of other object types, a more precise rule is necessary:

   $\langle c \rangle : \langle \mathsf{FNm}(\mathsf{Action}(c))\,{}_{[D]} \rangle$

   such that $\mathsf{InDisplay}(\mathsf{FNm}(x), {}_{[D]})$.

2. Suppose c is a surrogate, i.e. $c \in \mathcal{D}$. If there are no other object types using a surrogate of its corresponding predicator, $\neg\exists_{y,d}\,[x \neq y \wedge \mathsf{Pred}(d) = \mathsf{Pred}(c) \wedge y\,\mathsf{Owns}\,d]$, the following rule is added:

   $\langle c \rangle : \langle \mathsf{FNm}(\mathsf{Action}(\mathsf{Pred}(c))) \rangle$

   Otherwise the following (more sophisticated) rule is required:

   $\langle c \rangle : \langle \mathsf{FNm}(\mathsf{Action}(\mathsf{Pred}(c)))\,{}_{[D]} \rangle$

   such that $\mathsf{InDisplay}(\mathsf{FNm}(x), {}_{[D]})$.

3. Suppose c is a repetition predicator, i.e. $c \in \mathcal{X}$. Furthermore, component d is a decomposition of c, i.e. $\mathsf{Decomp}(d) = c$. This leads to the following rule:

   $\langle c \rangle :$ “repeatedly” $\langle \text{frag } d \rangle$

4. Suppose c is a choice predicator, i.e. $c \in \mathcal{Y}$. Furthermore, components $d_1, \ldots, d_k$ are decompositions of c. This leads to the following rule:

   $\langle c \rangle : \langle \text{frag } d_1 \rangle$ “or” … “or” $\langle \text{frag } d_k \rangle$

5. Suppose c is a merge predicator, i.e. $c \in \mathcal{Z}$. Furthermore, components $d_1, \ldots, d_k$ are decompositions of c. This leads to the following rule:

   $\langle c \rangle :$ “interleaved” $\langle \text{frag } d_1 \rangle$ “or” … “or” $\langle \text{frag } d_k \rangle$

The nonterminal $\langle \text{frag } d \rangle$ leads to the description of the course of life starting from components d. If the component d is not followed by another component, i.e. $\neg\exists_d\,[e\,\mathsf{Connect}_1\,d]$, no special care is required. Otherwise, the description leads to a text fragment within some other sentence. Therefore, this is marked explicitly with brackets:

$$\langle \text{frag } d \rangle : \begin{cases} \text{“[”} \langle \text{seq } d \rangle \text{“]”} & \text{if } d \text{ has a successor} \\ \langle d \rangle & \text{otherwise} \end{cases}$$

5 Related work

Using natural language for problem specification is not new in the field of computer science. In the early seventies *syntactically-oriented programming* methods, like step-wise refinement, were introduced, see e.g. [Dij76] or [Mee78]. The step-wise refinement method paraphrases the problem in a top-down fashion, using simple control structures like `IF THEN ELSE FI`. Also the use of natural language in information modeling is not new. The conceptual data modeling techniques *EER* (as described in [BCD+95]), NIAM (as described in [DO90]), and PSM (as described in [HPW94] and [CW93]) are based on such an approach.

5.1 Grammatical analysis

Linguistic approaches to object-oriented modeling have been taken before:

- For the OOA method ([CY90]) a simple linguistic approach is described, which only considers *nouns* and *verbs*. Although this method offers the system analyst not more than a rule of thumb, it is still very useful for the determination of the global structure of the system, i.e. this method supports the analyst in finding object types and action types.
- In the *SACIS* method (see e.g. [Gra94]) verbs and nouns are further refined. Besides these two refinements the SACIS method provides grammatical views for attributes, operations, associations and events.
- The *KISS* method ([Kri94]) is a further refinement, because it allows for a deep analysis of the informal specification. Besides finding object types, via direct objects and indirect objects in sentences, the *connection* and *direction* between object types is investigated by focusing on verbs (predicates) and prepositions. A novelty is the use of *gerunds*. Gerunds turn out to be objectified action types.
- The NIBA project ([FKM+00]) focusses on natural language based information requirements analysis. The authors introduce an intemediate model called CPM (Conceptual Predesign Model) to bridge the gap between natural language and formal model.
- In [BDT97] RADD-NLI (Rapid Application and Database Development - Natural Language Interface) is presented as a mechanism to capture behavioral information from verbal descriptions. This interface has been added as a component of the workbench RADD (Rapid Application and Database Development) which has been developed on the basis of HERM ([Tha00]), an extended entity relationship model.
- The SynDiKATe system presents a framework for automatically acquiring knowledge from text and automatically representing it in a formal structure ([HR00]). The authors introduce a system architecture integrating requirements obtained from the analysis of single sentences. Furthermore, this system can handle referentially linked sentences.
- The Grammalizer project ([HVH90]) involved the design and implementation of a case tool which is based on text analysis. This project is part of the LIKE project, which is further elaborated in the next subsection.

All these methods in some way use grammatical analysis of the initial specification but lack a mechanism for paraphrasing the models obtained by these methods to natural language sentences. The way of working of these methods, during analysis, consists of collecting lists of candidate object types, action types, etc.

5.2 Paraphrasing

We discuss the following approaches to paraphrasing:

- In [RP92] *generative grammar* ([Cho65]) is used for the generation of natural language sentences out of conceptual schemata. The proposed framework is supported by an expert design system, known as *OICSI* (French acronym for intelligent tool for information system design). This approach is not particularly focused on object-oriented analysis.
- A similar and according to the authors ([MG94]) less sophisticated tool is *LOLITA*, which supports a linguistic approach for developing object-oriented systems. A data dictionary is used to find semantic relations between the object types that are detected in the informal specification. Furthermore, alternative formulations may be suggested by the system. Besides this feature, LOLITA is able to automatically identify ambiguities, inconsistencies, to correct misspellings and guess new words.
- In [Joh95] the author identifies related object types using *Galois connections*. In order to verify that the proposed correspondences between object types are consistent with the semantics of the conceptual schemas, the modeling formalism utilizes linguistic instruments. The instruments used are in particular *case grammars* and *speech act theory*. The modeling formalism described can be used to support *schema integration*.

- The *LIKE* project (Linguistic Instruments in Knowledge Engineering) has (among other things) resulted in a method called *COLOR-X* ([Bur96]) which is an abbreviation for Conceptual Linguistically based Object-oriented Representation language for Information and Communication Systems (ICS is abbreviated to X). The use of a lexicon during the modeling phase has been covered by [Hop97]. The static and dynamic part of object-oriented modeling is described in [BR95b] and [BR95a], respectively. The expert language is captured in graphical models which can be translated to an intermediate language called Conceptual Prototyping Language (CPL). *CPL* ([Dig89]) is a formal modeling technique, based on *functional grammars* ([Dik89]), which can be used for specification as close as possible to the informal specification. As a result the paraphrasing of the graphical models is based on CPL. Building on LIKE results, [SRD00] determine organization primitives using grammatical analysis, and elaborate on the relation between functional grammar and CPL.

The main difference between the above mentioned paraphrasing approaches and ours is that they are semantically based whilst ours is syntactically based. The restriction to syntactical aspects is a simplification, yet it does not seem to affect expressiveness too dramatically.

6 Conclusion

This paper has started with the observation that information system architectures have evolved from machine-oriented techniques into a more and more human-oriented approach. In this approach the information grammar, i.e. the grammar which covers the dynamic and static aspects of the communication in a UoD, plays a central role. The focus of an object-oriented analysis is to provide a mechanism to registrate what may occur in the UoD. The notion of a logbook has been used as a unifying framework for such a registration. A logbook describes both static and dynamic aspects of the UoD.

The rules which govern the contents of a logbook for some UoD are layed down subsequently in three object-oriented analysis models. These analysis models are specific views on the overall model, forming the information architecture. The information architecture describes the information grammar.

The analysis models can also be seen as a growing awareness of the communication within the UoD. Each analysis model also provides the opportunity for validation based on feedback via a paraphrasing mechanism of the associated part of the information grammar. Furthermore the integration of these three object-oriented analysis models has been discussed. The object-oriented modeling analysis technique in this paper is part of the analysis method PSM^2 , for which also a way of working is elaborated. Based on this way of working, a provisional educational program has been designed to train a group of 20 persons in an insurance and banking company to become system analyst, in the context of building document information systems. It appeared that workflow aspects could well be described in terms of object life models. The resulting models were transformed into models of a specific document management system. These transformation rules appeared to be quite intuitive. A more embedded discussion on PSM^2 for object-oriented analysis can be found in [Fre97].

Verification and design issues with respect to information architectures form a subject for further research. A first approach may be found in [FW96d] and [KB97].

A Graphical symbols

This appendix summarizes the graphical notation of the $\mathcal{OAI}$ model (figure 9), the $\mathcal{OP}$ model (figure 10) and the $\mathcal{OL}$ model (figure 11).

B Logbooks

The simplest description of events occurring in a UoD consists of the following components for each event: (1) *when* does it occur, (2) *what* is happening, (3) *who* are associated, (4) what is the *role* of each associate and (5) which *properties* of a associates are relevant. A logbook is a set of such event descriptions, ordered by their time-stamps in some unifying format (seconds, hours, days, etc.). In the sequel of this section logbooks are introduced informally. For a formal treatment of logbooks see [BFW96].

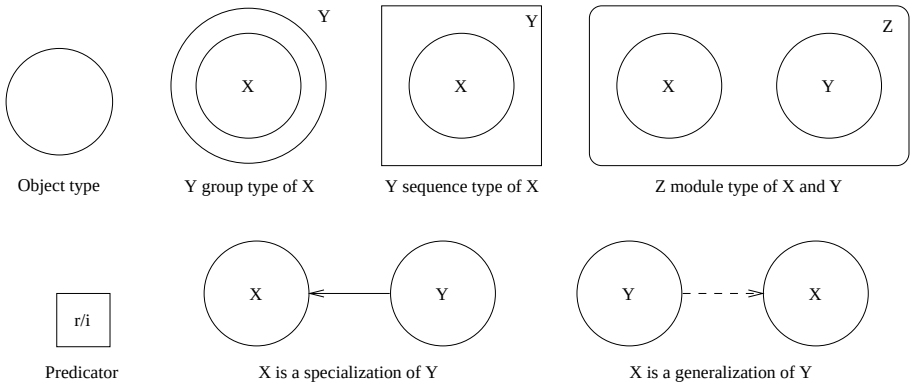

Figure 9: Symbols of object action involvent model

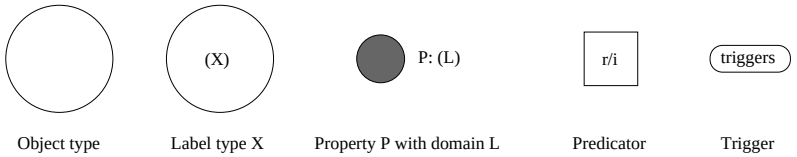

Figure 10: Symbols of object property model

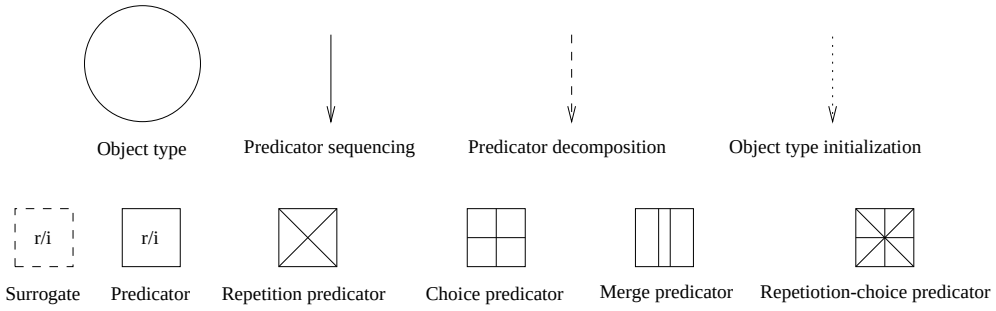

Figure 11: Symbols of object life model

B.1 Informal specifications

The starting-point for constructing a logbook is an informal specification consisting of natural language sentences describing events with a time stamp. Obtaining an informal specification usually is an incremental and iterative process ([RP96], [FKW95], [FKW96]) which requires skills for both the domain expert and system analyst ([FW96a]). As an example of an informal specification consider table 1, where the unifying time format is chosen to be on a daily basis.

01-05-1963:	Mick Jagger and Keith Richards set up band 'The Rolling Stones'.
03-05-1967:	Mick Jagger and Keith Richards write Paint It Black.
21-06-1967:	The Rolling Stones record, on tape number 666 in studio number 11, Paint It Black.
23-06-1967:	Mick Jagger and Keith Richards produce the recording, on tape number 666 in studio number 11, of Paint It Black.
12-12-1988:	A. Knijff writes the song 'I Want You'.
10-02-1989:	P. Frederiks and A. Knijff set up The Playful Plebs.
29-04-1991:	The band 'Playful Plebs' record, in studio number 2 on tape number 3, the song 'I Want You'.
29-04-1991:	The band 'Playful Plebs' record, in studio number 2 on tape number 3, the song 'Long Way To Go'.
05-05-1991:	H. Honer produces the recording, recorded in studio number 2 on tape number 3, of the song 'I Want You'.

Table 1: Example informal specification

B.2 Meta-model of logbooks

Formally, the format of a logbook is defined by the meta model of figure 12. This meta model is presented as a PSM schema which is a formalized and extended version of the object-role modeling technique NIAM. For more literature about the background and formalization of PSM, the reader is referred to [HW93] or [Hof93].

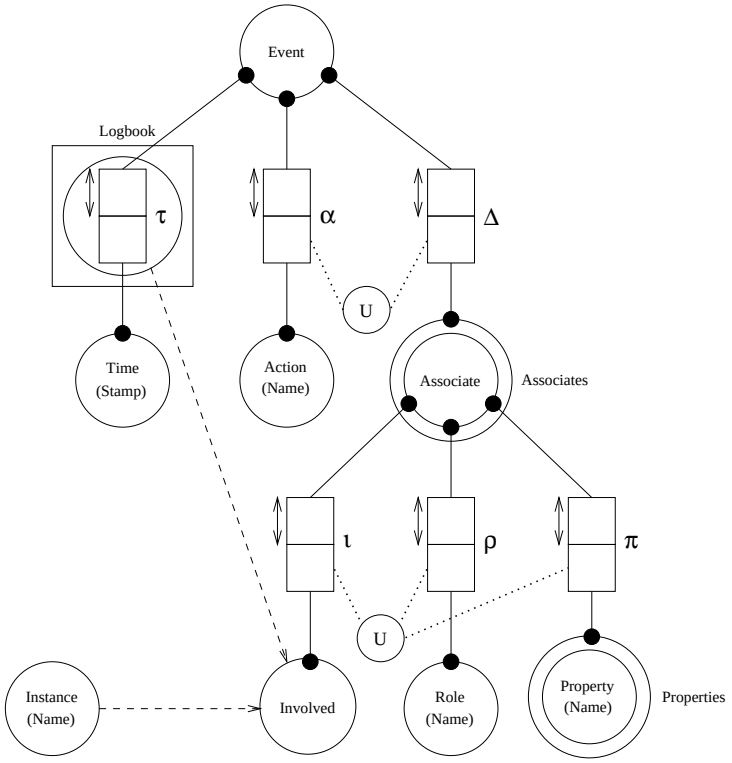

Figure 12: Meta-model for the format of a logbook

In figure 12 we have a number of entity types, e.g. *Event*, represented by circles. Label types, also represented by circles, appear between parentheses. Examples of label types are *Name* and *Stamp*[3]. The convention is used that if a label type is an identifier for an entity type, the label type is represented within the same circle. For example, in the logbook meta-model, an *Action* is identified by its *Name*. Fact types consist of predicators represented by boxes connected with circles for their associated object types. For example fact type τ is a binary fact type consisting of two predicators, with associated object types *Event* and *Time*. Object type *Properties* is modeled as a group type with underlying element type *Property*. A *Logbook* is seen as a sequence type of tuples consisting of an *Event* and *Time* part. Finally, the information structure of a logbook contains a generalization hierarchy. Object type *Involved* is either an *Instance* or an *Event/Time* tuple.

The PSM schema of figure 12 also contains a number of constraints. A *Total role* constraint, denoted by a black dot, expresses that each instance of the corresponding object type must be involved in the corresponding fact type. For example each *Event* must have a *Time* component. The arrows and 'circles with an *U*' represent so-called *uniqueness* constraints over a predicator and set of predicators, respectively. The uniqueness constraint is used to specify that each instance of the corresponding object type is involved *only once* in the corresponding fact type *if* it plays that role. Note that both a total role constraint and uniqueness constraint over a predicator states that an instance of the corresponding object type is involved *exactly once* in the corresponding fact type.

In order to be able to represent logbooks in a way as presented in table 1, some additional (non-graphical) constraints are required. The PSM modeling technique is equipped with the query language Lisa-D to express such constraints. Lisa-D as such falls outside the scope of this paper, for more information see [HPW93]. The additional constraints are informally expressed as:

1. the order of the tuples of a logbook is according to the order of its time stamps, and
2. if two events occur at the same time, these events can not be the same.

Table 2 (see appendix C) represents a population of the meta-model of the logbook in figure 12 and thus is a unifying format for the sample logbook of table 1. Note that the action *to write* may have multiple agents. However, each of these events is elementary, and thus can not be split in smaller events. As a consequence, the modeling technique must be able to handle the case of multiple agents on a conceptual level.

A population of the logbook meta-model can be obtained by grammatical analysis of the sentences of the informal specification. Methods such as KISS ([Kri94]) and SACIS (as described in [Gra94]) provide the analyst with clues for grammatical analysis of informal specifications. Grammatical analysis as such fall outside the scope of this paper, for more readings see [Fre97].

B.3 Views on logbooks

During the several stages of the modeling process the sentences of a logbook are analyzed according to three perspectives, leading to a number of abstractions of the logbook, also referred to as the *analysis models*. Each abstraction provides a specific view on the nature of the application domain, and results in a corresponding model. The models together compose a conceptual model of the UoD. This conceptual model is also referred to as the *information architecture*. The information architecture composes the following analysis models:

- *object action involvement model*,
- *object property model*, and
- *object life model*.

The purpose of the *object action involvement model* is to develop a first approximation to the information grammar, covering the main structure of the logbook language. This model provides an abstraction (typing) for three columns of the logbook, i.e. *Action*, *Involved*, and *Role* (see table 2). Special is that different types of events may have the same type of action. Eventually this can lead to the introduction of generalized object types (see section 2).

The *object property model* deals with static aspects of the application domain, by modeling *properties* of object types. The properties of an object type form a so-called *state record* which reports about the current state of an object. Usually properties are set during the birth of an instance of the associated object type. Properties

[3]The population of label type *Stamp* is assumed to be an ordered set.

are only useful if they can be retrieved via *retrieval* action types and updated by so-called *update* action types. Strictly spoken, properties thus can also be modeled via its initialization action type, retrieval action types and update action types. The object property model is introduced to provide a more simple description mechanism dealing with properties. This model is obtained inter alia by consulting the *Property* column of the logbook.

The object action involvement model does not restrict the order of the action types. The *object life model* elaborates on the object action involvement model. The object life model considers the application domain from a historical perspective, and describes for each object type the *course of life* of its instances. Each object type starts with its birth action type (creation). From the object action involvement model it can be derived in which action types an object type can be involved. Using the column *Time* of the logbook, clues for the ordering of action types can be derived. Note that the object life model describes a possible order in which action types can be performed for each object type. By collecting all life courses the (restrictions on the) dynamics of the UoD can be derived.

C Sample logbook population

Time	Event			
	Action	Associate		
		Involved	Role	Properties
01-05-1963	set up	Mick Jagger	agent	$\emptyset$
		Keith Richards	agent	$\emptyset$
		The Rolling Stones	object	$\emptyset$
03-05-1967	write	Mick Jagger	agent	$\emptyset$
		Keith Richards	agent	$\emptyset$
		Paint It Black	object	$\emptyset$
21-06-1967	record	Paint It black	object	$\emptyset$
		The Rolling Stones	agent	$\emptyset$
23-06-1967	produce	$\left\langle \text{21-06-1967}, \left\langle \text{record}, \left\{ \begin{array}{l} \langle \text{Paint It black}, \text{object}, \emptyset \rangle, \\ \langle \text{The Rolling Stones}, \text{agent}, \emptyset \rangle \end{array} \right\} \right\rangle \right\rangle$	object	$\left\{ \begin{array}{l} \text{tape number 666,} \\ \text{studio number 11} \end{array} \right\}$
		Mick Jagger	agent	$\emptyset$
		Keith Richards	agent	$\emptyset$
12-12-1989	writes	I Want You	object	$\emptyset$
		A. Knijff	agent	$\emptyset$
10-02-1989	set up	P. Frederiks	agent	$\emptyset$
		A. Knijff	agent	$\emptyset$
		The Playful Plebs	object	$\emptyset$
29-04-1991	record	Playful Plebs	agent	$\emptyset$
		I Want You	object	$\emptyset$
29-04-1991	record	Playful Plebs	agent	$\emptyset$
		Long Way To Go	object	$\emptyset$
05-05-1991	produces	H. Honer	agent	$\emptyset$
		$\left\langle \text{29-04-1991}, \left\langle \text{record}, \left\{ \begin{array}{l} \langle \text{I Want You}, \text{object}, \emptyset \rangle, \\ \langle \text{Playful Plebs}, \text{agent}, \emptyset \rangle \end{array} \right\} \right\rangle \right\rangle$	object	$\left\{ \begin{array}{l} \text{tape number 3,} \\ \text{studio number 2} \end{array} \right\}$

Table 2: A sample population of figure 12

D Overview of the analysis models

Model	focus on	sentence structures
$\mathcal{OAI}$	objects, actions, roles, specialization, generalization, complex objects and decomposition	⟨noun⟩ ⟨verb⟩ ⟨noun⟩ ⟨proposition⟩* ⟨noun⟩ is a ⟨noun⟩ ⟨noun⟩ is a group of ⟨noun⟩ ⟨noun⟩ is a sequence of ⟨noun⟩ ⟨noun⟩ is a composition of ⟨noun⟩*
$\mathcal{OP}$	objects, birth actions, properties, triggers	⟨noun⟩ has ⟨noun⟩ When ⟨sentence of $\mathcal{OAI}$⟩ then ⟨sentence of $\mathcal{OAI}$⟩
$\mathcal{OL}$	course of life for each object	ordering the generated sentences of $\mathcal{OAI}$ using (mainly temporal) conjunctions

Table 3: Model overview

Acknowledgment

The authors would like to thank Patrick van Bommel, Caspar Derksen, Stijn Hoppenbrouwers and Kees Koster for their comments and suggestions on earlier versions of this paper. Their contributions are herewith gratefully acknowledged. Furthermore, the authors thank the anonymous referees for their valuable comments, which resulted in many improvements to this paper.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[BCD+95] E. Buchholz, H. Cyriaks, A. Düsterhöft, H. Mehlan, and B. Thalheim. Applying a Natural Language Dialogue Tool for Designing Databases. In *Proceedings of the First International Workshop on Applications of Natural Language to Databases (NLDB'95)*, pages 119–133, Versailles, France, June 1995.

[BDT97] E. Buchholz, A. Düsterhöft, and B. Thalheim. Capturing information on behaviour with the radd-nli – linguistic and knowledge-based approach. *Data & Knowledge Engineering*, 23:33–46, 1997.

[BFW96] P. van Bommel, P.J.M. Frederiks, and Th.P. van der Weide. Object-Oriented Modeling based on Logbooks. *The Computer Journal*, 39(9), 1996.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City, California, 1991.

[BR95a] J.F.M. Burg and R.P. van de Riet. COLOR-X: Linguistically-based Event Modeling: A General Approach to Dynamic Modeling. In J. Iivari, K. Lyytinen, and M. Rossi, editors, *The Proceedings of the Seventh International Conference on Advanced Information System Engineering*, Lecture Notes in Computer Science, pages 26–39, Jyvaskyla, Finland, 1995. Springer-Verlag.

[BR95b] J.F.M. Burg and R.P. van de Riet. COLOR-X: Object Modeling profits from Linguistics. In *Proceedings of the KB&KS'95, the Second International Conference on Building and Sharing of Very Large-Scale Knowledge Bases*, Enschede, The Netherlands, 1995.

[Bra83] R.J. Brachman. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantics Networks. *IEEE Computer*, 16(10):30–36, 1983.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[Bub86] J.A. Bubenko. Information System Methodologies - A Research View. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: Improving the Practice*, pages 289–318. North-Holland, Amsterdam, The Netherlands, 1986.

[Bur96] J.F.M. Burg. *Linguistic Instruments In Requirements Engineering.* PhD thesis, Free University, Amsterdam, The Netherlands, 1996.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, Cambridge, United Kingdom, 1990.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In *International Symposium on Semantics of Data types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67, Berlin, Germany, June 1984. Springer-Verlag.

[Cho65] N. Chomsky. *Aspects of the theory of syntax.* MIT Press, Cambridge, Massachusetts, 1965.

[CW93] M.A. Collignon and Th.P. van der Weide. An Information Analysis Method Based on PSM. In G.M. Nijssen, editor, *Proceedings of NIAM-ISDM.* NIAM-GUIDE, September 1993.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Yourdon Press, New York, New York, 1990.

[Dal95] H. Dalianis. Aggregation, Formal Specification and Natural Language Generation. In *Proceedings of the First International Workshop on Applications of Natural Language to Databases (NLDB'95)*, pages 135–149, Versailles, France, June 1995.

[Dat91] C.J. Date. *An Introduction to Data Base Systems*, volume 1. Addison-Wesley, Reading, Massachusetts, 5th edition, 1991.

[DFW96] C.F. Derksen, P.J.M. Frederiks, and Th.P. van der Weide. Paraphrasing as a Technique to Support Object-Oriented Analysis. In R.P. van de Riet, J.F.M. Burg, and A.J. van der Vos, editors, *Proceedings of the Second Workshop on Applications of Natural Language to Databases (NLDB'96)*, pages 28–39, Amsterdam, The Netherlands, June 1996.

[Dig89] F.P.M. Dignum. *A Language for Modelling Knowledge Bases.* PhD thesis, Free University, Amsterdam, The Netherlands, 1989.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Dik89] S.C. Dik. *The Theory of Functional Grammar. Part I: The Structure of the Clause.* Floris Publications, Dordrecht, The Netherlands, 1989.

[DO90] L. Dunn and M.E. Orlowska. A Natural Language Interpreter for the Construction of Conceptual Schemas. In B. Steinholz, A. Sølvberg, and L. Bergman, editors, *Proceedings of the Second Nordic Conference CAiSE'90 on Advanced Information Systems Engineering*, volume 436 of *Lecture Notes in Computer Science*, pages 175–194, Stockholm, Sweden, 1990. Springer-Verlag.

[FHL97] P.J.M. Frederiks, A.H.M. ter Hofstede, and E. Lippe. A unifying framework for conceptual data modelling concepts. *Information and Software Technology*, 39(1):15–25, January 1997.

[FKM+00] G. Fliedl, C. Kop, H.C. Mayr, W. Mayerthaler, and C. Winkler. Linguistically based requirements engineering – the niba-project. *Data & Knowledge Engineering*, 35:111–120, 2000.

[FKW95] P.J.M. Frederiks, C.H.A. Koster, and Th.P. van der Weide. Object-Oriented Analysis using Informal Language. Technical Report CSI-R9516, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, December 1995.

[FKW96] P.J.M. Frederiks, C.H.A. Koster, and Th.P. van der Weide. Validation of Object-Oriented Analysis Models using Informal Language. Technical Report CSI-R9609, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, May 1996.

[Fre97] P.J.M. Frederiks. *Object-Oriented Modeling based on Information Grammars.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1997.

[FW96a] P.J.M. Frederiks and Th.P. van der Weide. Cognitive Requirements for Natural Language Based Conceptual Modeling. Technical Report CSI-R9610, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, June 1996.

[FW96b] P.J.M. Frederiks and Th.P. van der Weide. Formalization, Integration, and Validation of Object-Oriented Analysis Models leading to an Information Grammar. Technical Report CSI-R9625, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, December 1996.

[FW96c] P.J.M. Frederiks and Th.P. van der Weide. From a File-Oriented View to an Object-Oriented View. Technical Report CSI-R9601, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, January 1996.

[FW96d] P.J.M. Frederiks and Th.P. van der Weide. Verification and Design for Information Architectures. Technical Note CSI-N9611, December 1996.

[Gra94] I. Graham. *Object-oriented Methods.* Addison-Wesley, Reading, Massachusetts, 1994.

[Gri82] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base.* Publ. nr. ISO/TC97/SC5-N695, 1982.

[Hal95] T.A. Halpin. *Conceptual Schema and Relational Database Design.* Prentice-Hall, Sydney, Australia, 2nd edition, 1995.

[HLF96] A.H.M. ter Hofstede, E. Lippe, and P.J.M. Frederiks. Conceptual Data Modeling from a Categorical Perspective. *The Computer Journal*, 39(3):215–231, August 1996.

[HLW97] A.H.M. ter Hofstede, E. Lippe, and Th.P. van der Weide. A categorical framework for conceptual data modeling: Definition, application, and implementation. *Acta Informatica*, 34(12):927–963, 1997.

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

[Hop97] J. Hoppenbrouwers. *Conceptual Modeling and the Lexicon.* PhD thesis, Tilburg University, Tilburg, The Netherlands, 1997.

[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. Grammar Based Information Modelling. Technical Report CSI-R9414, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, October 1994.

[HPW97] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Exploiting Fact Verbalisation in Conceptual Information Modelling. *Information Systems*, 22(6/7):349–385, 1997.

[HR00] U. Hahn and M. Romacker. Content management in the syndikate system – how technical documents are automatically transformed to text knowledge bases. *Data & Knowledge Engineering*, 35:137–159, 2000.

[HVH90] J. Hoppenbrouwers, B. van der Vos, and S. Hoppenbrouwers. Structures and conceptual modelling: Grammalizing for kiss. *Data & Knowledge Engineering*, 23:79–92, 1990.

[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.

[JCJO92] I. Jacobson, M. Christerson, M. Jonsson, and P. van Overgaard. *OO Software Engineering, A Use Case Driven Approach.* Addison-Wesley, Reading, Massachusets, 1992.

[Joh95] P. Johannesson. Supporting Schema Integration by Linguistic Instruments. In *Proceedings of the First International Workshop on Applications of Natural Language to Databases (NLDB'95)*, pages 41–56, Versailles, France, June 1995.

[KB97] G. Kovács and P. van Bommel. From Conceptual Model to OO Database via Intermediate Specification. *Acta Cybernetica*, 13:103–140, 1997.

[Kos70] C.H.A. Koster. Affix grammars. In *ALGOL 68 implementation, Proceedings of the IFIP Working Conference on ALGOL 68 Implementation 1970*, pages 95–109, Amsterdam, The Netherlands, 1970. North-Holland.

[Kri94] G. Kristen. *Object Orientation, the KISS Method: From Information Architecture to Information System.* Addison-Wesley, Reading, Massachusetts, 1994.

[Lek93] H. van der Lek. On the Structure of an Information Grammar. In G.M. Nijssen, editor, *Proceedings of NIAM-ISDM.* NIAM-GUIDE, September 1993.

[MBF+93] G.A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K.J. Miller. Five Papers on WordNet. Technical Report, Cognitive Science Laboratory, Princeton University, 1993.

[Mee78] L.G.L.T. Meertens. Program text and program structure. In P.G. Hibbard and S.A. Schuman, editors, *Constructing quality software*, pages 271–283, Amsterdam, The Netherlands, May 1978. North-Holland. IFIP WG 2.1/WG 2.4 Working Conference, Novosibirsk.

[MG94] L. Mich and R. Garigliano. A Linguistic Approach to the Development of Object Oriented Systems using the NL System LOLITA. In E. Bertino and S. Urban, editors, *Proceedings of the International Symposium, ISOOMS '94: Object-Oriented Methodologies and Systems*, volume 858 of *Lecture Notes in Computer Science*, pages 371–386, Palermo, Italy, September 1994. Springer-Verlag.

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

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

[PW95a] 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.

[PW95b] 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.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[RP92] C. Rolland and C. Proix. A Natural Language Approach For Requirements Engineering. 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 257–277, Manchester, United Kingdom, 1992. Springer-Verlag.

[RP96] D. Redmond-Pyle. Software development methods and tools: some trends and issues. *Software Engineering Journal*, 11(2):99–103, March 1996.

[SD96] M. Snoeck and G. Dedene. Generalization/specialization and role in object oriented conceptual modeling. *Data & Knowledge Engineering*, 19:171–195, September 1996.

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

[SRD00] A.A.G. Steuten, R.P. van der Riet, and J.L.G. Dietz. Linguistically based conceptual modeling of business communication. *Data & Knowledge Engineering*, 35:121–136, 2000.

[Tha00] B. Thalheim. *Foundations of Database Technology.* Springer Heidelberg, 2000.

[The00] M. Theune. *From Data to Speech: Language Generation in Context.* PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2000.

[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.

[Wil96] C.P. Willis. Analysis of inheritance and multiple inheritance. *Software Engineering Journal*, 11:215–224, July 1996.

[WZ96] R. Weber and Y. Zhang. An analytical evaluation of NIAM's grammar for conceptual schema diagrams. *Information Systems Journal*, 6(2):147–170, April 1996.