Formal Definition of a Conceptual Language for the Description and Manipulation of Information Models

A.H.M. ter Hofstede, H.A. Proper, Th.P. van der Weide

Department of Information Systems
University of Nijmegen
Toernooiveld
6525 ED Nijmegen
The Netherlands

Published as: 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), 1993.

Abstract

Conceptual data modelling techniques aim at the representation of data at a high level of abstraction. This implies that conceptual data modelling techniques should not only be capable of naturally representing complex structures, but also the rules (constraints) that must hold for these structures. Contemporary data modelling techniques however, do not provide a language, which on the one hand has a formal semantics and on the other hand leads to natural looking expressions, for formulating these constraints. In this paper such a language is defined for an existing data modelling technique (PSM), which is a generalisation of object-role models (such as ER or NIAM). In this language not only constraints, but also queries and updates can be expressed on a conceptual level.

1 Introduction

Currently, many conceptual data modelling techniques exist. Conceptual data modelling techniques aim at the representation of data at a high level of abstraction. The *Conceptualisation Principle* ([15]) states that a conceptual schema should deal only and exclusively with aspects of the underlying Universe of Discourse (UoD). Any aspect irrelevant to that meaning, e.g. machine efficiency, should be avoided. Contemporary data modelling techniques are not capable of adhering to the Conceptualisation Principle for each UoD. Choices that are not relevant with respect to the UoD have to be made (leading to *overspecification*) or, even worse, the UoD has to be adapted, e.g. extra object types have to be introduced, to meet the requirements of the modelling technique. These problems are caused by the lack of sufficiently powerful construction mechanisms.

Another important principle of conceptual data modelling is the *100% Principle* ([15]), which states that a conceptual schema completely prescribes all the permitted states and transitions of the conceptual data base. This implies that a conceptual data modelling technique should not only be capable of representing complex structures but also rules (constraints) that must hold for these structures. In most modelling techniques such constraints can not be expressed formally, but need to be expressed in natural language, obviously causing interpretation problems ([22]). Besides constraints, it would also be convenient to be able to express queries and updates on a conceptual level. Many query and manipulation languages (e.g. SQL) require a fairly high level of training or are based on a rather primitive data modelling technique (e.g. ER).

In [23], the conceptual data modelling technique PSM (Predicator Set Model) has been defined, which is capable of representing complex object structures without violating the Conceptualisation Principle. PSM is an extension of PM (Predicator Model [4]) which on its turn is a formalisation of NIAM ([31], [42], [17]). This means that all NIAM schemas can be seen as PSM schemas. It also means that the design procedure supporting the construction of NIAM schemas and the NIAM philosophy are not lost, they only need to be extended to support also the additional constructs.

The NIAM analysis method originates from the early seventies, and is based on an analysis method for natural language. The language starts from examples which are (partial) descriptions of the underlying domain provided by domain experts. Such an analysis leads, in a natural way, to an information structure. The use of examples helps to bridge the gap between domain expert and system analyst. It is only obvious that the language for manipulating and querying has the format of a semi natural language. The language RIDL (Reference and IDea Language [9], [29]) was developed for this purpose. However, due to its informal definition, no rigid base for both syntax and semantics was provided, the language never got much acceptance. Furthermore, RIDL was based on the restricted binary version of NIAM ([38]).

The intention of this paper is to make a (re)design of, a strongly extended version of, RIDL. The resulting language is called LISA-D (Language for Information Structure and Access Descriptions), and is based on PSM. Its functionality far exceeds the intended functionality of RIDL. As PSM has been designed as a general object-role modelling technique, LISA-D is (in principle) also applicable to well-known representatives of object-role modelling techniques such as ER ([7]), FDM ([36]) or INFOMOD ([24]).

The organisation of this paper is as follows. In section 2 a summary of the formal definition of PSM is given in order to make this paper self-contained. In section 3 path expressions are introduced. Path expressions form a primitive, yet powerful, language for information manipulation. In section 4 the language LISA-D is introduced and formally defined by means of a translation to path expressions. Information descriptors form the basic syntactical construct of LISA-D and they are used for the definition of constraints, queries and updates in LISA-D. Upon first reading, sections 2 and 3 may be skipped, although they are necessary for a complete understanding of this paper.

2 The Predicator Set Model

This section contains a formal description of PSM. This formal description serves as a platform for a manipulation language, introduced in the next section. The formal description consists of three parts. In the first part, information structures are defined. Information structures capture the syntax of PSM schemas without graphical constraints. The second part deals with instantiations, referred to as populations, of information structures. The third part contains the requirements imposed on a PSM schema that make it possible to uniquely denote abstract instances in terms of concrete instances (labels). This is called structural identification. (This section may be skipped during a first introductory reading).

2.1 The Information Structure

An *information structure* is a structure consisting of the following basic components:

1. A finite set $\mathcal{P}$ of *predicators*.
2. A nonempty set $\mathcal{O}$ of *object types*.

3. A set $\mathcal{L}$ of *label types*. Label types are also object types: $\mathcal{L} \subseteq \mathcal{O}$.
4. A set $\mathcal{E}$ of *entity types* ($\mathcal{E} \subseteq \mathcal{O}$).
5. A partition $\mathcal{F}$ of the set $\mathcal{P}$. The elements of $\mathcal{F}$ are called *fact types*. Fact types are also object types ($\mathcal{F} \subseteq \mathcal{O}$). The auxiliary function $\mathsf{Fact} : \mathcal{P} \rightarrow \mathcal{F}$ yields the fact type in which a given predicator is contained, and is defined by: $\mathsf{Fact}(p) = f \Leftrightarrow p \in f$.
6. A set $\mathcal{G}$ of *power types*. Power types form a special class of object types ($\mathcal{G} \subseteq \mathcal{O}$).
7. A set $\mathcal{S}$ of *sequence types*. Sequence types form a special class of object types ($\mathcal{S} \subseteq \mathcal{O}$).
8. A set $\mathcal{C}$ of *schema types*: $\mathcal{C} \subseteq \mathcal{O}$.
9. A function $\mathsf{Base} : \mathcal{P} \rightarrow \mathcal{O}$. The base of a predicator is the object part of that predicator.
10. A function $\mathsf{Elt} : \mathcal{G} \cup \mathcal{S} \rightarrow \mathcal{O}$. This function yields the element type of power types and sequence types.
11. A relation $\prec \subseteq \mathcal{C} \times \mathcal{O}$. This relation describes the decomposition of schema types.
12. A partial order Spec on object types, capturing specialisation.
13. A function $\sqcap : \mathcal{O} \rightarrow \mathcal{O}$ yielding the *Pater Familias* of a given object type.
14. A partial order Gen on object types, expressing generalisation.

In this approach, the instances of object types are *not* part of the information structure. Instantiations (populations) will be introduced in section 2.2.

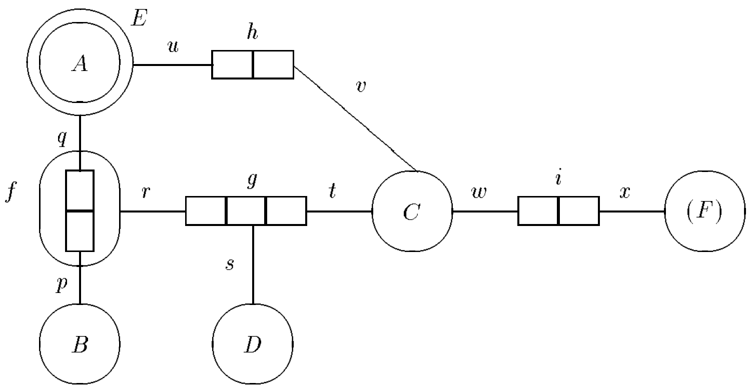


Figure 1: Example information structure

Example 2.1
Figure 1 shows an information structure diagram *visualising the information structure that consists of:*

$$\begin{array}{rclrcl} \mathcal{P} &=& \{p,q,r,s,t,u,v,w,x\} & \mathcal{O} &=& \{A,B,C,D,E,F,f,g,h,i\} \\ \mathcal{F} &=& \{f,g,h,i\} & \mathcal{G} &=& \{E\} \\ \mathcal{S} &=& \varnothing & \mathcal{C} &=& \varnothing \\ \mathcal{E} &=& \{A,B,C,D\} & \mathcal{L} &=& \{F\} \end{array}$$

where $f = \{p,q\}$, $g = \{r,s,t\}$, $h = \{u,v\}$, $i = \{w,x\}$. *With respect to the predicators:* $\mathsf{Base}(p) = B$, $\mathsf{Base}(q) = A$, $\mathsf{Base}(r) = f$, *etc. Finally,* $\mathsf{Elt}(E) = A$, Spec *and* Gen *are empty, and* $\sqcap(x) = x$ *for all object types* x.

Due to the different interpretation that will be given to label types, fact types, power types, sequence types, schema types and entity types, these object types are all considered to be different concepts:

(PSM1) (*separation*) $\mathcal{L}$, $\mathcal{F}$, $\mathcal{G}$, $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{E}$ form a partition of $\mathcal{O}$

2.1.1 Abstract and concrete objects

In data modelling there exists a distinction between objects that can be represented directly and objects that cannot be represented directly. In ER, this distinction is reflected by the difference between entity types and attribute types, while in NIAM and PSM this distinction corresponds to the difference between entity types and label types. Labels can be represented directly on a communication medium, while other objects depend for their representation on labels. As a result, label types are also called *concrete* object types, as opposed to the other object types which are referred to as *abstract* object types. The gap between concrete and abstract object types can only be crossed by special binary fact types, called *bridge types* in the NIAM terminology. We will come back to this in the next subsection.

2.1.2 Fact typing

One of the key concepts in data modelling is the concept of relationship type. Generally, a relation type is considered to represent an association between object types. In figure 2 the graphical representation of a binary relation R between object types X_1 and X_2 is shown, both in the NIAM and ER style. A relation type consists of a number of roles (r_1 and r_2 in figure 2), denoting the way object types participate in that relation type. The connection between an object type and a role is called a predicator (p_1 and p_2 in figure 2, see [4]).

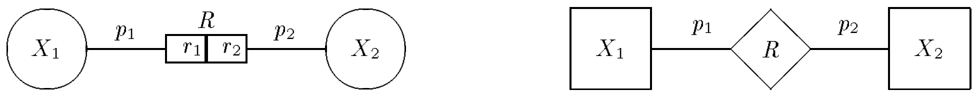


Figure 2: A NIAM relation type, and its corresponding ER diagram

In PSM a relation type is considered to be a *set* of predicators. A relation type is therefore considered to be an association between predicators, rather than between objects types. A relation type (also referred to as *fact type*) may be treated as an object type (*fact objectification*), and can therefore play a role in other relation types.

Bridge types establish the connection between abstract and concrete object types. The term $\mathsf{Bridge}(f)$ qualifies fact type f as a bridge type, and is an abbreviation for the expression

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

$\mathcal{B}$ denotes the set of bridge types. The strict separation between the concrete and abstract level is expressed by the rule that label types may only participate in bridge types:

(PSM2) $\mathsf{Base}(p) \in \mathcal{L} \Rightarrow \mathsf{Bridge}(\mathsf{Fact}(p))$

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

Example 2.2 *In figure 1, i is a bridge type with* $\mathsf{concr}(i) = x$ *and* $\mathsf{abstr}(i) = w$.

2.1.3 Power typing

The concept of *power type* in PSM forms the data modelling pendant of power sets in conventional set theory ([26]). This notion is the same as the notion of grouping as introduced in the IFO data model ([1]). An instance of a power type is a set of instances of its *element type*. Such an instance is identified by its elements, just as a set is identified by its elements in set theory (axiom of extensionality).

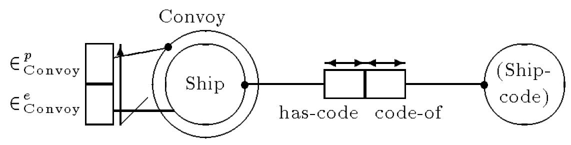


Figure 3: A simple example of a power type

An example of power typing is the *Convoy Problem* (based on [18]), depicted in figure 3. There, the object type *Convoy* is a power type with as element type *Ship*. As a result, each instance of object type *Convoy* is a set of instances of *Ship*. Convoys are identified by their constituent ships, whereas ships are identified by a *Ship-code*, which is a label type. To distinguish label types from entity types in diagrams, label type names are parenthesized. Furthermore, the black dot on the object type *Ship* is an example of a so-called total role constraint, it expresses that each instance of *Ship* has to play the role *has-code*. The arrow above this role is an example of a uniqueness constraint and expresses that instances of *Ship* play the role *has-code* at most once. The formal semantics of these graphical contraint types can be found in [4]. An overview of the drawing conventions is included in the appendix.

This Convoy Problem is *not* expressible in terms of a NIAM or ER schema (see [21]), without violating the Conceptualisation Principle.

The element type of a power type is found by the function Elt. The relation between a power type x and its element type $\mathsf{Elt}(x)$ is recorded in the fact type $\in_x = \{\in^p_x, \in^e_x\}$, where $\mathsf{Base}(\in^p_x) = x$ and $\mathsf{Base}(\in^e_x) = \mathsf{Elt}(x)$. This relation is assumed to be available for each power type. Usually $\in_x$ is treated as an implicit fact type, and not drawn in the information structure diagram. If this fact type is subject to constraints it needs to be made explicit. Note that, in this way, power typing corresponds to a polymorphic type constructor, and the fact type $\in_x$ to an associated polymorphic access operator.

The strict separation between abstract and concrete object types prohibits label types to occur as element type:

(PSM3) $\mathsf{Elt}(x) \notin \mathcal{L}$

2.1.4 Sequence typing

Sequence typing offers the opportunity to represent sequences, built from an underlying element type. This notion is not elementary in PSM, as it is expressible in terms of generalisation (see [23]). Nonetheless, the concept of sequence type is treated as an independent concept in this paper, because this facilitates its use in the manipulation language to be introduced in the remainder of this paper.

Example 2.3 *A train is identified by a T-code, and consists of a locomotive followed by a sequence of freight cars. This Universe of Discourse is modelled in the information structure diagram of figure 4.*

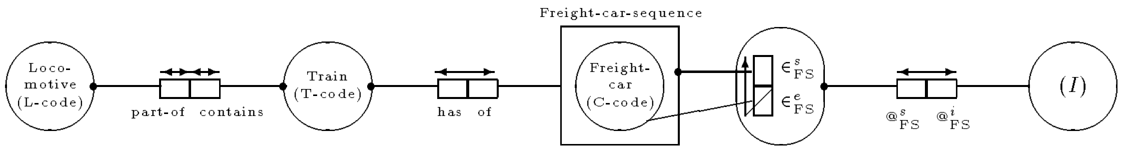


Figure 4: The train composition administration

The element type of a sequence type is also found by the function Elt. The relation between a sequence type x and its element type $\mathsf{Elt}(x)$ is recorded in the (implicit) fact type $\in_x = \{\in^s_x, \in^e_x\}$, where $\mathsf{Base}(\in^s_x) = x$ and $\mathsf{Base}(\in^e_x) = \mathsf{Elt}(x)$. Contrary to power types, this relation $\in_x$ is augmented with the position of the element in the sequence, via the (implicit) fact type $@_x = \left\{@^s_x, @^i_x\right\}$, where $\mathsf{Base}(@^s_x) = \in_x$ and $\mathsf{Base}(@^i_x) = I$. The object type I is the domain for indexes in sequence types. Usually the natural numbers are used for this purpose. The index type is assumed to be a label type ($I \in \mathcal{L}$), which is assumed to be totally ordered and to have a least element.

Note that axiom **PSM3** also applies for sequence types.

2.1.5 Schema typing

A schema type is an object type with an underlying decomposition. The concept of *schema typing* allows for the decomposition of large schemata into, objectified, subschemata. The need for such a mechanism has been generally recognised. Though in [23] it has been argued that schema typing is not an elementary concept, it is considered an independent concept here for the same reason as mentioned for sequence types in the previous section.

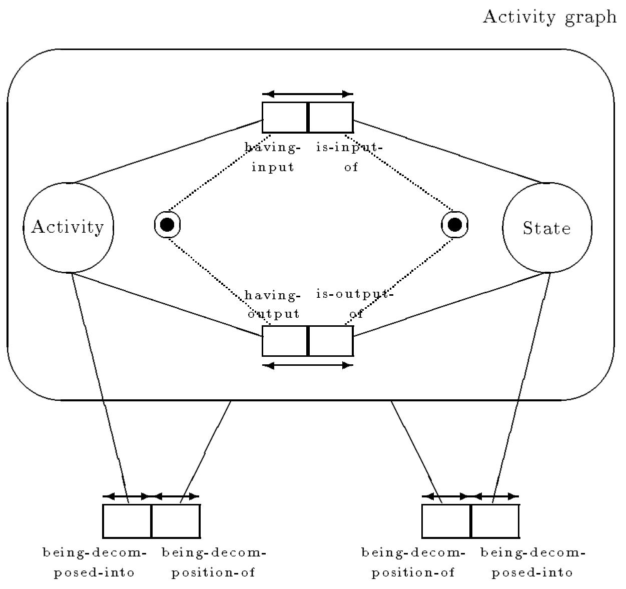


Figure 5: An information structure diagram for Activity Graphs

Example 2.4 *Activity Graphs are a well-known modelling technique for processes (see [35]). Activity graphs are bipartite directed graphs consisting of activities (processes) and states. States, which can be compared to flows in data flow diagrams (see e.g. [43]), can be input for or output of* *activities. In an Activity Graph, both activities and states may be subject to decomposition. This results in the information structure diagram of figure 5.*

Schema types can be decomposed into an underlying information structure via the relation $\prec$, with the convention that $x \prec y$ is interpreted as x is decomposed into y or y is part of the decomposition of x. This underling information structure $\mathcal{I}_x$ for a schema type x is derived from the object types into which x is decomposed: $\mathcal{O}_x = \{ y \in \mathcal{O} \mid x \prec y \}$. Analogously, the special object classes $\mathcal{F}_x$, $\mathcal{G}_x$, $\mathcal{S}_x$, $\mathcal{C}_x$ and $\mathcal{E}_x$ can be derived. The functions Base_x, Elt_x, $\prec_x$, Spec_x, $\sqcap_x$ and Gen_x are obtained by restriction to object types within $\mathcal{O}_x$. In order to be a proper decomposition, the underlying information structure should form an information structure on its own:

(PSM4) (*structural nesting*) $x \in \mathcal{C} \Rightarrow \mathcal{I}_x$ is a PSM information structure

With each schema type x and each object type y in its decomposition, an (implicit) fact type $\in_{x,y} = \{\in^c_{x,y}, \in^d_{x,y}\}$ is associated, where $\mathsf{Base}(\in^c_{x,y}) = x$ and $\mathsf{Base}(\in^d_{x,y}) = y$. This fact type will enable the transition from a schema object to an object from its decomposition.

2.1.6 Specialisation

Specialisation, referred to as subtyping in NIAM, is a mechanism for representing one or more (possibly overlapping) subtypes of an object type. Specialisation is to be applied when only for specific instances of an object type certain facts are to be recorded. Suppose for example that only for *Adults*, i.e. *Persons* with an *Age* greater or equal than 18, one is interested in the *Cars* they own. This situation is captured by the PSM schema in figure 6.

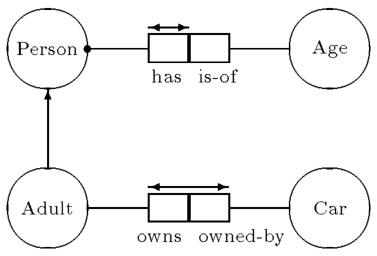


Figure 6: Example of specialisation

A specialisation relation between a subtype and a supertype implies that the instances of the subtype are also instances of the supertype (each *Adult* is also a *Person*). For proper specialisation, it is required that subtypes be defined in terms of one or more of their supertypes. Such a decision criterion is referred to as *Subtype Defining Rule* ([4]). In figure 6 the subtype defining rule for *Adult* is expressed (in LISA-D) as:

Adult = Person has Age $\geq$ 18

As a consequence, identification of subtypes is derived from their supertypes. Therefore, if in the ongoing example *Persons* would be identified by a name, then *Adults* are also identified by that name.

Specialisation relations are organised in so-called specialisation "hierarchies". A specialisation hierarchy is in fact not a hierarchy in the strict sense, but an acyclic directed graph with a unique top. This top is referred to as the *pater familias* (see [10]). In the example of figure 6, the pater familias of *Adult* is *Person*.

Objects inherit all properties from their ancestors in the specialisation hierarchy. This characteristic of specialisation excludes non-entity types (e.g. fact types) occurring as subtypes. Consider for example the case that a ternary fact type is a subtype of a binary fact type. Clearly this leads to a contradiction. No problems occur when non-entity types themselves are specialised. Consequently, non-entity types always act as pater familias. For an in depth discussion of specialisation, we refer to [16].

The concept of specialisation is introduced as a partial order (asymmetric and transitive) Spec on object types, with the convention that $a \,\mathsf{Spec}\, b$ is interpreted as: a is a subtype (specialisation) of b, or b is a supertype of a. Subtypes inherit the structure of their supertypes. A consequence is that only entity types can act as subtype. This, on its turn, prohibits specialisation of label types.

(PSM5) (*strictness*) $\mathsf{Spec} \subseteq \mathcal{E} \times (\mathcal{O} - \mathcal{L})$

(PSM6) (*asymmetry*) $a \,\mathsf{Spec}\, b \Rightarrow \neg b \,\mathsf{Spec}\, a$

(PSM7) (*transitivity*) $a \,\mathsf{Spec}\, b \wedge b \,\mathsf{Spec}\, c \Rightarrow a \,\mathsf{Spec}\, c$

Each specialisation hierarchy has a unique top element, the pater familias of the object types in this hierarchy. The pater familias is found by the function $\sqcap : \mathcal{O} \to \mathcal{O}$ (which is similar to the top operator from lattice theory). This function has the following properties:

(PSM8) (*cohesion*) $a \,\mathsf{Spec}\, b \Rightarrow \sqcap(a) = \sqcap(b)$

(PSM9) (*ancestor*) $a \neq \sqcap(a) \Rightarrow a \,\mathsf{Spec}\, \sqcap(a)$

In the remainder $\mathsf{spec}(a)$ will be used as an abbreviation for $\exists_{x \in \mathcal{O}} [x \,\mathsf{Spec}\, a]$. From these axioms the following important property, stating that a pater familias cannot be a subtype, can be derived (the proof can be found in [23]).

Lemma 2.1 $\neg \sqcap(a) \,\mathsf{Spec}\, b$

Corollary 2.1 *Idempotency of* $\sqcap$*:* $\sqcap(\sqcap(a)) = \sqcap(a)$

2.1.7 Generalisation

Generalisation is a mechanism that allows for the creation of new object types by uniting existing object types. Generalisation is to be applied when different object types play identical roles in fact types. Contrary to what its name suggests, generalisation is *not* the inverse of specialisation. Specialisation and generalisation originate from different axioms in set theory ([23]) and therefore have a different expressive power.

For generalisation it typically is required that the generalised object type is covered by its constituent object types (or *specifiers*). Therefore, a decision criterion as in the case of specialisation (the subtype defining rule) is not necessary. Furthermore, properties are inherited "upward" in a generalisation hierarchy instead of "downward", which is the case for specialisation (see also [1]). This also implies that the identification of a generalised object type depends on the identification of its specifiers. From the nature of generalisation, it is apparent that a non-entity type cannot be a generalised object type.

Example 2.5 *In figure 7 we see an example of generalisation. A formula may be either a single variable, or constructed by some function (say f) from simpler formulas. It is clear that instances from the object type* Formula *inherit the structure (identification) from the specifier from which they originate (*Variable *or f).*

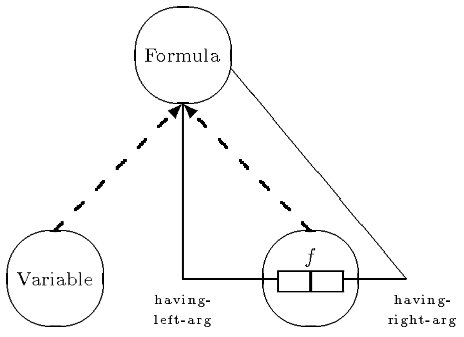


Figure 7: Example of generalisation

This example also shows that generalisation can be used to define recursive object types. This is not possible in the IFO data model ([1]), where object types are hierarchical structures. In the Logical Data Model (see [25]), however, object types are directed graphs, which may contain cycles.

The concept of generalisation is introduced as a partial order (asymmetric and transitive) Gen with the convention that $a \,\mathsf{Gen}\, b$ is interpreted as: a is a generalisation of b, or b is a specifier of a. As generalised objects inherit the structure from the specifier from which they originate, only entity types can act as generalised object types. The strict separation between abstract and concrete object types prohibits the generalisation of label types.

(PSM10) (*strictness*) $\mathsf{Gen} \subseteq \mathcal{E} \times (\mathcal{O} - \mathcal{L})$

(PSM11) (*asymmetry*) $a \,\mathsf{Gen}\, b \Rightarrow \neg b \,\mathsf{Gen}\, a$

(PSM12) (*transitivity*) $a \,\mathsf{Gen}\, b \wedge b \,\mathsf{Gen}\, c \Rightarrow a \,\mathsf{Gen}\, c$

In the remainder $\mathsf{gen}(a)$ will be used as an abbreviation for $\exists_{x \in \mathcal{O}} [a \,\mathsf{Gen}\, x]$. Generalisation and specialisation can be conflicting due to their inheritance structure. To avoid such conflicts, generalised object types are required to be pater familias:

(PSM13) $\mathsf{gen}(a) \Rightarrow \sqcap(a) = a$

2.1.8 Type Relatedness

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

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

(T1) $\vdash x \sim x$

(T2) $x \sim y \vdash y \sim x$

(T3) $x \,\mathsf{Spec}\, y \wedge y \sim z \vdash x \sim z$

(T4) $x\ \mathsf{Gen}\ y \land y \sim z \vdash x \sim z$

(T5) $x, y \in \mathcal{G} \land \mathsf{Elt}(x) \sim \mathsf{Elt}(y) \vdash x \sim y$

(T6) $x, y \in \mathcal{S} \land \mathsf{Elt}(x) \sim \mathsf{Elt}(y) \vdash x \sim y$

(T7) $\mathcal{O}_x = \mathcal{O}_y \vdash x \sim y$

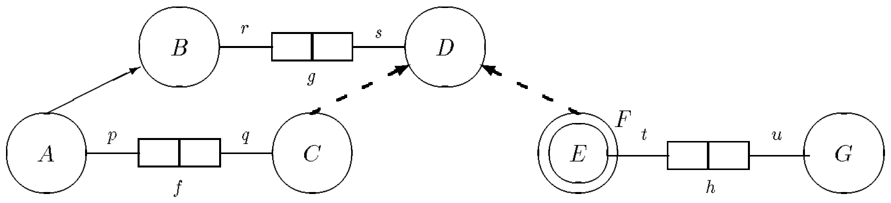


Figure 8: Example information structure

Example 2.6 *In figure 8 the only object types that are type related are A and B, C and D and F and D.*

2.2 Populations

An information structure is used as a frame for some part of the (real) world, the so-called Universe of Discourse (U∘D). A *state* of the U∘D then corresponds to a so-called instantiation or population of the information structure, and vice versa. The idea of states was previously mentioned in [14], [39], [8]. Furthermore, a state transition of the U∘D has a corresponding transition on populations of the information structure. This can be formulated as:

> The Universe of Discourse is *isomorphic* with the set of possible populations of the information structure and a transition relation hereupon.

This is called the *conceptuality property* of information structures. In this paper, a population Pop of an information structure $\mathcal{I}$ is a value assignment of sets of instances to the object types in $\mathcal{O}$, satisfying the rules that will follow in the rest of this section. This is denoted as $\mathsf{IsPop}(\mathcal{I}, \mathsf{Pop})$. Pop then is a mapping $\mathsf{Pop} : \mathcal{O} \rightarrow \wp(\Omega)$, where Ω is the universe of instances that can occur in the population of an information structure $\mathcal{I}$. This universe of instances is defined in definition 2.1. The set of all populations is defined as $\mathsf{POP} = \mathcal{O} \rightarrow \wp(\Omega)$.

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

Definition 2.1 The universe of instances Ω is inductively defined as the smallest set satisfying:

1. $\bigcup D \subseteq \Omega$. Instances from the sorts in the many sorted algebra are elements of the universe of instances.
2. $\Theta \subseteq \Omega$, where Θ is an abstract (countable) domain of (unstructured) values that may occur in the population of entity types.

3. $x_1, \ldots, x_n \in \Omega \wedge p_1, \ldots, p_n \in \mathcal{P} \Rightarrow \{p_1 : x_1, \ldots p_n : x_n\} \in \Omega$. The set $\{p_1 : x_1, \ldots, p_n : x_n\}$ denotes a mapping, assigning x_i to each predicator p_i. These mappings are intended for the population of fact types (see the Conformity Rule).
4. $x_1, \ldots, x_n \in \Omega \Rightarrow \{x_1, \ldots, x_n\} \in \Omega$. Sets of instances may occur as instances of power types (see the Power Base Rule).
5. $x_1, \ldots, x_n \in \Omega \Rightarrow \langle x_1, \ldots, x_n \rangle \in \Omega$. Sequences of instances are used as instances of sequence types (see the Sequence Type Rule). The i-th element of a sequence $\langle x_1, \ldots, x_n \rangle$, i.e. x_i, can be derived using projection, denoted as: $\langle x_1, \ldots, x_n \rangle [i]$.
6. $X_1, \ldots, X_n \subseteq \Omega \wedge O_1, \ldots, O_n \in \mathcal{O} \Rightarrow \{O_1 : X_1, \ldots, O_n : X_n\} \in \Omega$. Assignments of sets of instances to object types are also valid instances. They are intended for the populations of composition types (see the Decomposition Rule).

The first population rule is the *Strong Typing* rule, which expresses that instantiations of abstract object types may only have instances in common, if they are type related.

(P1) $x, y \notin \mathcal{L} \wedge x \not\sim y \Rightarrow \mathsf{Pop}(x) \cap \mathsf{Pop}(y) = \varnothing$

The population of a label type is a set of values, taken from its corresponding concrete domain:

(P2) $x \in \mathcal{L} \Rightarrow \mathsf{Pop}(x) \subseteq \mathsf{Dom}(x)$

Root object types are object types that are neither generalised, nor a subtype. This is formalised as: $\mathsf{IsRoot}(x) \equiv \neg\, \mathsf{gen}(x) \wedge \neg\, \mathsf{spec}(x)$. The population of root entity types is a set of values, taken from the abstract domain Θ.

(P3) $x \in \mathcal{E} \wedge \mathsf{IsRoot}(x) \Rightarrow \mathsf{Pop}(x) \subseteq \Theta$

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

(P4) $x \in \mathcal{F} \wedge y \in \mathsf{Pop}(x) \Rightarrow y :\ x \ \rightarrow\ \Omega \wedge \forall_{p \in x} \left[y(p) \in \mathsf{Pop}(\mathsf{Base}(p)) \right]$

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

(P5) $x \in \mathcal{G} \wedge y \in \mathsf{Pop}(x) \Rightarrow y \in \wp(\mathsf{Pop}(\mathsf{Elt}(x))) - \{\varnothing\}$

The (implicit) fact type $\in_x$ that is provided for each power type x, describes the relation between power type x and its element type $\mathsf{Elt}(x)$. This is described in the *Power Base Rule*:

(P6) $x \in \mathcal{G} \Rightarrow \mathsf{Pop}(\in_x) = \left\{ \{\in_x^p : u, \in_x^e : v\} \mid u \in \mathsf{Pop}(x) \wedge v \in u \right\}$

The Power Base Rule is as a *derivation rule* for the population of fact type $\in_x$. Note that it is not necessary in the Power Base Rule to state that $v \in \mathsf{Pop}(\mathsf{Elt}(x))$ since this follows from the Conformity Rule. The population of a sequence type consists of (nonempty) sequences of instances of the corresponding element type. This is called the *Sequence Type Rule*:

(P7) $x \in \mathcal{S} \wedge y \in \mathsf{Pop}(s) \Rightarrow y \in \mathsf{Pop}(\mathsf{Elt}(x))^{+}$

Indexing in sequence type x is provided by the (implicit) fact types $\in_x$ and $@_x$. This is conceived in the *Sequence Decomposition Rules.*

(P8) $x \in \mathcal{S} \Rightarrow \mathsf{Pop}(\in_x) = \left\{ \left\{\in_x^s : u, \in_x^e : v\right\} \mid u \in \mathsf{Pop}(x) \wedge \exists_{i \in I} \left[u[i] = v\right] \right\}$

(P9) $x \in \mathcal{S} \Rightarrow \mathsf{Pop}(@_x) = \left\{ \left\{@_x^s : u, @_x^i : v\right\} \mid u \in \mathsf{Pop}(\in_x) \wedge u(\in_x^s)[v] = u(\in_x^e) \right\}$

These rules can be used as derivation rules for $\in_x$ and $@_x$.

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

(P10) $x \in \mathcal{C} \wedge y \in \mathsf{Pop}(x) \Rightarrow \mathsf{IsPop}(\mathcal{I}_x, y)$

The relation between a composition type and its constituing object types is recorded in the fact type $\in_{c,d}$. Its population is decribed in the *Decompositor Rule*, which is a derivation rule:

(P11) $x \prec y \Rightarrow \mathsf{Pop}(\in_{x,y}) = \left\{ \left\{\in_{x,y}^c : u, \in_{x,y}^d : v\right\} \mid u \in \mathsf{Pop}(x) \wedge v \in u(y) \right\}$

Lemma 2.2 $x \prec y \Rightarrow \forall_{u \in \mathsf{Pop}(x)} \left[u(y) \subseteq \mathsf{Pop}(y)\right]$

Proof: Assume $u \in \mathsf{Pop}(x)$ and $v \in u(y)$. Applying the Decompositor Rule one can derive that $\left\{\in_{x,y}^c : u, \in_{x,y}^d : v\right\} \in \mathsf{Pop}(\in_{x,y})$. From the Conformity Rule and the fact that $\mathsf{Base}(\in_{x,y}^d) = y$ it then follows that $v \in \mathsf{Pop}(y)$.

$\square$

Respecting the specialisation hierarchy is reflected by the *Specialisation Rule*:

(P12) $x \,\mathsf{Spec}\, y \Rightarrow \mathsf{Pop}(x) \subseteq \mathsf{Pop}(y)$

This rule does not require that instances of subtypes have to fulfil the subtype defining rule associated to the involved subtype. A subtype defining rule is defined as an information descriptor (see section 4). Up to this point no language for the formulation of such rules is available. The subtype defining rule should however also be considered as a population derivation rule, the population of a subtype can be computed using this rule.

Respecting the Generalisation hierarchy is reflected by the *Generalisation Rule*:

(P13) $\mathsf{gen}(x) \Rightarrow \mathsf{Pop}(x) = \bigcup_{x \,\mathsf{Gen}\, y} \mathsf{Pop}(y)$

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

Example 2.7 *A sample population of the information structure of figure 1 is:*

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{Pop}(A) & = & \{a_1, a_2\} & \mathsf{Pop}(f) & = & \left\{\{p : b_1, q : a_1\}, \{p : b_1, q : a_2\}\right\} \\
\mathsf{Pop}(B) & = & \{b_1\} & \mathsf{Pop}(g) & = & \left\{\{r : \{p : b_1, q : a_1\}, s : d_1, t : c_1\}\right\} \\
\mathsf{Pop}(C) & = & \{c_1\} & \mathsf{Pop}(h) & = & \left\{\{u : \{a_1\}, v : c_1\}, \{u : \{a_1, a_2\}, v : c_1\}\right\} \\
\mathsf{Pop}(D) & = & \{d_1\} & \mathsf{Pop}(i) & = & \left\{\{w : c_1, x : 17\}\right\} \\
\mathsf{Pop}(E) & = & \left\{\{a_1\}, \{a_1, a_2\}\right\} & & & \\
\mathsf{Pop}(F) & = & \{17\} & & &
\end{array}$$

It is assumed that the concrete domain of label type F is the set of natural numbers. In the above population 17 comes from this domain and is the only label instance. The instances a_1, a_2, b_1, c_1 and d_1 come from the abstract domain Θ and are considered to be non-denotable by a user. Note that if the instance $\{w : c_2, x : 17\}$ is added to the population of fact type i the conformity rule is violated, since c_2 is not an element of $\mathsf{Pop}(C)$. In figure 9 this population is graphically represented. The population of the implicit fact type $\in_E$ can be derived to be:

$$\mathsf{Pop}(\in_E) = \left\{\{\in_E^p : \{a_1\}, \in_E^e : a_1\}, \{\in_E^p : \{a_1, a_2\}, \in_E^e : a_1\}, \{\in_E^p : \{a_1, a_2\}, \in_E^e : a_2\}\right\}$$

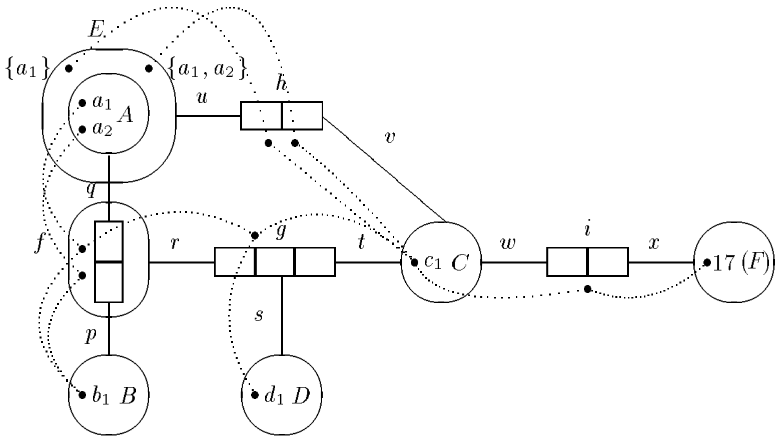


Figure 9: Graphic representation of a population

2.3 Structural identification

Structural identifiability is a schema property that ensures that each population is weakly identified, i.e., in each population each object instance can be identified by some of its properties. This makes it possible to denote abstract instances, e.g. entities, in terms of concrete instances, i.e. labels.

Let $\Sigma = \langle \mathcal{I}, \mathcal{R} \rangle$ be a PSM schema over information structure $\mathcal{I}$ bounded by a set $\mathcal{R}$ of constraints. The important constraints for structural identification are the total role constraint and the uniqueness constraint. Informally, a total role constraint $\mathsf{total}(\tau)$ over a set of predicators τ states that object instances in the population of their bases occur at least once in the population of these predicators. A uniqueness constraint $\mathsf{unique}(\tau)$ over a set of predicators τ expresses the uniqueness of combinations of values in these predicators. Examples of a total role constraint and a uniqueness constraints have been discussed in example 6. The formal semantics of the $\mathsf{total}(\tau)$ and $\mathsf{unique}(\tau)$ is given in [4] and [40].

The requirements for structural identification have been presented in [23], and are only briefly listed here. A PSM schema Σ is *structurally identifiable* iff:

1. Σ is closed over labels, i.e., each label type occurs in some total role constraint:

$$\forall_{x \in \mathcal{L}} \exists_{p \in \mathcal{P}} \exists_{\mathsf{total}(\tau) \in \mathcal{R}} \left[\mathsf{Base}(p) = x \wedge p \in \tau\right]$$

   The motivation behind this is to enforce the absence of unused label values.

2. All object types can be identified:

$$\forall_{x \in \mathcal{O}} \left[\mathsf{Identifiable}(x)\right]$$

   The identification of an object can be seen as a fixed set of properties that provide a unique description in terms of label values.

The predicate Identifiable is defined in terms of the structure of objects. The respective object classes are discussed consecutively.

Label Types
If x is a label type, then obviously $\mathsf{Identifiable}(x)$.

Fact Types
A fact type x (or, generally, a set of predicators) is identifiable if all components of x are identifiable:

$$\forall_{p \in x} [\mathsf{Identifiable}(\mathsf{Base}(p))]$$

Power Types and Sequence Types
A power type or sequence type x is identifiable if its element type is identifiable:

$$\mathsf{Identifiable}(\mathsf{Elt}(x))$$

Composition Types
A composition type x is identifiable if all its constituent object types are identifiable:

$$\forall_{x \prec y} [\mathsf{Identifiable}(y)]$$

Entity Types
If x is an entity type, then the following cases can be distinguished.

If x is *not* pater familias ($\sqcap(x) \neq x$) then x takes (inherits) its identification from its pater familias, provided that the subtype membership is decidable from the subtype defining rules (see section 2).

A second case of identification inheritance arises from object generalisation. In this case the object type inherits its identification from some of its specifiers. More precisely, if x is a generalised object type ($\mathsf{gen}(x)$), then x is identifiable if:

$$\exists_{y \in \mathcal{O}} [x \,\mathsf{Gen}\, y \wedge \mathsf{Identifiable}(y)]$$

This leaves the identification of root object types, in which case we are looking for identification paths (denominations). These denominations form a recipe for uniquely denoting each object of the object type, in any population. The set of possible first names for denominations is defined by:

$$N(x) = \{\, p \mid \mathsf{Base}(p) = x \wedge \mathsf{total}(\{p\}) \wedge \mathsf{unique}(\{p\}) \,\}$$

The identification of object type x now depends on the existence of a set of middle names (constituting a so-called identifier), i.e., a set τ of predicators such that:

- (*uniqueness of denotation*) $\mathsf{unique}(\tau)$
- (*first name-middle name relatedness*) $\forall_{p \in \tau} \exists_{q \in \mathsf{Fact}(p)} [q \in N(x)]$
- (*recursion*) $\forall_{p \in \tau} [\mathsf{Identifiable}(\mathsf{Base}(p))]$

Example 2.8 *In figure 10 an example of identification in the case of an entity type that is a pater familias is shown. The bases of predicators p_1, p_5 and p_9 are label types. Entity type* Address *can be identified by identifier $\{p_7, p_9\}$, which requires the identification of* Street. *This can be achieved by identifier $\{p_3, p_5\}$, which on its turn requires the identification of* Community. *Communities are identified by a* C-name. *As a result, an* Address *can be uniquely denoted in the following format:*

$$(p_9 : \text{H-nr}, p_7 : (p_5 : \text{S-name}, p_3 : (p_1 : \text{C-name})))$$

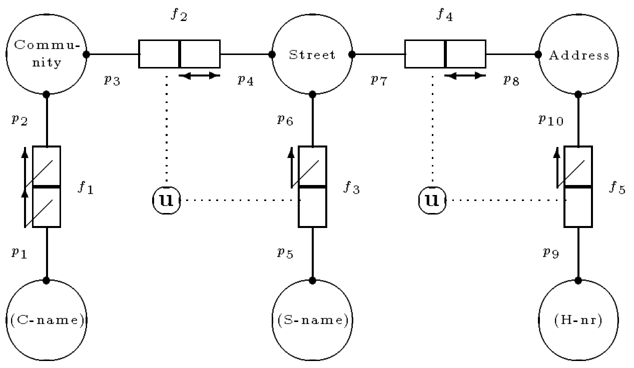


Figure 10: Example of complex identification

Structural identification ensures the existence of denominations for entity types. For each entity type one denomination has to be selected as its standard name. In order to get short denotations for entity types, by omitting predicators, an order of the middle names is defined by $\mathsf{Ident} : \mathcal{E} \rightarrow \mathcal{P}^*$. The (partial) function $\mathsf{Copred} : \mathcal{P} \rightarrowtail \mathcal{P}$ is introduced to resolve any ambiguity in the relation between middle names and first names. However, predicators from the same fact type should be assigned the same copredicator:

$$\mathsf{Fact}(p) = \mathsf{Fact}(q) \Rightarrow \mathsf{Copred}(p) = \mathsf{Copred}(q)$$

Example 2.9 *For example 2.8 the functions* Ident *and* Copred *could be:*

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{Ident}(\text{Address}) & = & \langle p_7, p_9 \rangle & \mathsf{Copred}(p_7) & = & p_8 \\
\mathsf{Ident}(\text{Street}) & = & \langle p_3, p_5 \rangle & \mathsf{Copred}(p_9) & = & p_{10} \\
\mathsf{Ident}(\text{Community}) & = & \langle p_1 \rangle & \mathsf{Copred}(p_5) & = & p_6 \\
 & & & \mathsf{Copred}(p_3) & = & p_4 \\
 & & & \mathsf{Copred}(p_1) & = & p_2
\end{array}$$

This allows a short denotation for addresses in the form:

$$\langle \text{H-nr}, \text{S-name}, \text{C-name} \rangle$$

Besides for the identification of entity types, Ident is extended to provide a standard naming convention for fact types as well. If $f = \{p_1, \ldots, p_n\}$ is a fact type, then $\mathsf{Ident}(f) = \langle p_1, \ldots, p_n \rangle$ determines an order on the predicators in f. This order will be used in section 4.5 to define a standard naming for fact type f.

3 Path Expressions

Path expressions are constructs for expressing derived fact types closely following the underlying information structure. Path expressions can be constructed from elements of the information structure (predicators, object types) and a number of operators. They are evaluated with respect to the current population of the information structure at hand. In its elementary form, a path expression corresponds to a path through the information structure, starting and ending in an object type. Intermediate object instances, though needed for the evaluation of path expressions, are discarded in their final result. The reason for this is uniformity, since this approach always leads to evaluation results in the form of *binary* relations. To compensate for the information that may be lost by discarding intermediate object instances, these binary relations take the form of *multisets* of tuples. More complex forms of path expressions may be inhomogeneous, i.e. resulting in tuples from different domains. Path expressions are thus interpreted as inhomogeneous binary multiset relations. At a first reading of the article, this section may be skipped.

As the semantics of path expressions are defined using multisets, this section starts with a treatment of multisets and operations on multisets. Section 3.2 then presents the formal definition of path expressions. The set of path expressions for a given information structure $\mathcal{I}$, is denoted as $\mathcal{PE}(\mathcal{I})$. In section 4 path expressions will be used to define the semantics of information descriptors in LISA-D.

3.1 Basic Algebraic Operations on Multisets

Multisets ([27]), also known as *multiple membership sets* ([26]), or *bags* ([32]), differ from ordinary sets in that a multiset may contain an element more than once. Multisets over an underlying domain X are elegantly introduced as functions: $X \rightarrow \mathbb{N}$, assigning to each $x \in X$ its frequency. In the definitions of the operations on multisets, the λ-calculus notation provided by [2], will be employed. For instance $\lambda x.x^2$ is the polynomal function assigning x^2 to each x-value.

As in set theory, $\varnothing$ denotes the empty multiset, with definition: $\lambda x.0$. If C is an expression which defines a function $M : X \rightarrow \mathbb{N}$, then: $M \equiv \{\!\!\{ e\uparrow^n \mid C(e,n) \}\!\!\}$ is a more conventional denotation for a multiset corresponding to bag comprehension, see e.g. [3]. Bag comprehension can be used for intentional denotations of multisets. Extentional denotations are defined by: $\{\!\!\{ a \}\!\!\} \equiv \{\!\!\{ a\uparrow^q \mid q = 1 \}\!\!\}$ and $\{\!\!\{ a_1, \ldots, a_n \}\!\!\} \equiv \{\!\!\{ a_1 \}\!\!\} \cup \ldots \cup \{\!\!\{ a_n \}\!\!\}$. We will write $e \in^n M$ rather than $M(e) = n$, and $e \in M$ for $M(e) > 0$. Besides forming multisets by means of an intentional or extentional specification, they can be formed by the following binary operators:

$$\begin{aligned} N \cup M &\equiv \lambda x.N(x) + M(x) \\ N \cap M &\equiv \lambda x.\min(N(x), M(x)) \\ N - M &\equiv \lambda x.\max(N(x) - M(x), 0) \end{aligned}$$

The comparison operator $N \subseteq M$ for multisets is defined as: $\forall_x [N(x) \leq M(x)]$. From this operator, the $\subset$ comparison is derived in the usual way: $N \subseteq M \wedge N \neq M$. This allows for the definition of the powerset of a multiset: $\wp(X) = \{\!\!\{ Y\uparrow^1 \mid Y \subseteq X \}\!\!\}$. Coercions from multiset to set and vice versa are defined by the following functions:

$$\begin{aligned} \mathsf{Set}(N) &\equiv \{x \mid x \in N\} \\ \mathsf{Multi}(S) &\equiv \{\!\!\{ x\uparrow^1 \mid x \in S \}\!\!\} \end{aligned}$$

The number of elements in a multiset is counted by $|N| \equiv \sum_{x \in X} N(x)$. In this paper, a useful class of multisets operations operates on multisets over binary tuples $X \times X$. We start by defining the following coercion operations between multisets over X and $X \times X$:

$$\mathsf{Sqr}(N) \equiv \{\!\!\{ \langle x, x \rangle\uparrow^n \mid x \in^n N \}\!\!\}$$

and conversely:

$$\begin{aligned} \pi_1(N) &\equiv \lambda x. \textstyle\sum_{y \in X} N(x,y) \\ \pi_2(N) &\equiv \lambda y. \textstyle\sum_{x \in X} N(x,y) \end{aligned}$$

We define three extra operations for multisets over $X \times X$:

$$\begin{aligned} N \circ M &\equiv \lambda \langle x, y \rangle. \textstyle\bigcup_{a \in X} N(x,a) \times M(a,y) \\ N \diamond M &\equiv \lambda \langle x, y \rangle. \textstyle\bigcup_{a,b \in X} N(x,a) \times M(y,b) \\ N^{\leftarrow} &\equiv \lambda \langle x, y \rangle.N(y,x) \end{aligned}$$

where $N^{\leftarrow}$ corresponds to the reverse relation, $N \circ M$ to the concatenation of N and M, and $N \diamond M$ to the head-head combinations of N and M. On the $\circ$ and $^{\leftarrow}$ operations, we define the neutral element: $1_{X \times X} \equiv \lambda\langle x, x\rangle.1$. We also define the following operation, being the multiset pendant of a union of a set of sets:

$$\biguplus N \equiv \lambda x. \sum_{A \in N, x \in A} N(A)$$

Note that N is a multiset of sets. By making assumptions on the underlying domains X we can introduce some more interesting operations. If X is an arithmetic domain, then the following operations can be defined:

$$\begin{aligned} \max(N) &\equiv \max(\mathsf{Set}(N)) \\ \min(N) &\equiv \min(\mathsf{Set}(N)) \\ \mathrm{sum}(N) &\equiv \textstyle\sum_{x \in X} x \times N(x) \end{aligned}$$

As in conventional set theory, the concept of *ordered pair* is introduced, and generalised to tuples of arbitrary length (also denoted as sequences). Sequences can be denoted by enumeration, e.g. $\langle a, b, c, d\rangle$. The operator Lin converts a tuple (of any length) to the corresponding multiset:

$$\mathsf{Lin}(\langle x_1, \ldots, x_n\rangle) = \bigcup_{1 \le i \le n} \{\!\!\{x[i]\}\!\!\}$$

for example $\mathsf{Lin}(\langle a, b, c, d, a\rangle) = \{\!\!\{a, a, b, c, d\}\!\!\}$.

3.2 Path Expressions

The syntax of path expressions is presented as an abstract syntax. In [30] the motivation for the use of an abstract syntax is stated as follows:

> *The use of abstract syntax rather than concrete syntax as a basis for studies of programming languages is representative of an important trend in software engineering: the move towards a higher-level view of software objects, emphasising deep structure rather than surface properties. Concepts such as abstract data types are another example of this trend.*

The semantics of path expressions will be defined using denotational semantics (see e.g. [37]). The semantics of each syntactical construct are defined in terms of other syntactical constructs, and ultimately in terms of multisets as defined in the previous subsection. An important role in denotational semantics is played by the environment, representing the state of a program. In the case of path expressions, the environment is the population of the information structure. Information descriptors are evaluated in the context of this environment.

As a path expression corresponds to a (directed) path through the information structure diagram, such a path is interpreted as describing a relation between the object types at its beginning and ending point. However, path expressions may be inhomogeneous, as a result of uniting path expressions with different ending points. In this case, the path expression leads to an inhomogeneous binary relation. Consequently, the semantics of path expressions are defined as binary relations over (multiple) object types. It will be convenient to treat these binary relations tuple oriented ([28]), as opposed to the mapping oriented approach to tuples in the population of fact types. As a result, the domain for these inhomogeneous binary multiset relations is derived from Ω in the following way:

$$\Omega_{\mathcal{PE}} = \{\, X \mid X \text{ is a multiset over } \Omega \times \Omega \,\}$$

Path expressions are built around the following syntactical categories: constant, multiset, object type ($\mathcal{O}$), predicator ($\mathcal{P}$) and path expression ($\boldsymbol{\mathcal{PE}}(\mathcal{I})$). The naming conventions are: c for constants, X for multisets, x for object types, p for predicators and P, Q, G, and $P_1, \ldots, P_n$ for path expressions. The function

$$\mu : \boldsymbol{\mathcal{PE}} \times \mathsf{POP} \to \Omega_{\mathcal{PE}}$$

is used to define the semantics of path expressions. First the atomic path expressions are introduced. Note the use of the function Sqr, necessary due to the interpretation of path expressions as binary relations. The operator $\cdot$ represents functional composition.

name	expr	$\mu[\![\mathsf{expr}]\!](\mathsf{Pop})$
empty path	$\varnothing_{\mathcal{PE}}$	$\varnothing$
neutral path	$1_{\mathcal{PE}}$	$1_{\Omega\times\Omega}$
constant	c	$\mathsf{Sqr}(\{\![c]\!\})$
multiset	X	$\mathsf{Sqr}(X)$
object type	x	$\mathsf{Sqr}\cdot\mathsf{Multi}\cdot\mathsf{Pop}(x)$
predicator	p	$\{\![\langle v(p), v\rangle\!\uparrow^1 \mid v \in \mathsf{Pop}\cdot\mathsf{Fact}(p)]\!\}$

Example 3.1 *Suppose* $\mathsf{Pop}(g) = \{\{r : b_1, s : c_1\}, \{r : b_2, s : \{e_1\}\}, \{r : b_3, s : \{e_2, e_3\}\}\}$ *in figure 8, then:*

$$\mu[\![r]\!](\mathsf{Pop}) = \begin{array}{|c|l|}\hline b_1 & \{r : b_1, s : c_1\} \\ b_2 & \{r : b_2, s : \{e_1\}\} \\ b_3 & \{r : b_3, s : \{e_2, e_3\}\} \\ \hline\end{array}$$

A number of operators and functions are available for the construction of composed path expressions. First the unary operators are introduced. They provide the opportunity to reverse a path P as: $P^{\leftarrow}$, to isolate the front elements of a path P by: $\mathit{f}\!\!\!- P$, to remove multiple occurrences using: $\mathsf{ds}\,P$, to count the number of elements in a path expression by: $\mathsf{Cnt}(P)$, to add the elements in a path expression by means of: Sum, and to determine the minimum or maximum element in a path expression by: Min and Max. The powerset $\wp(P)$ of a path expression P yields a path expression with all sets of instances occurring in the first component of P. The operators are summarised in the following table:

name	expr	$\mu[\![\mathsf{expr}]\!](\mathsf{Pop})$
reverse	$P^{\leftarrow}$	$\mu[\![P]\!](\mathsf{Pop})^{\leftarrow}$
front	$\mathit{f}\!\!\!- P$	$\mathsf{Sqr}\cdot\pi_1\cdot\mu[\![P]\!](\mathsf{Pop})$
distinct	$\mathsf{ds}\,P$	$\mathsf{Multi}\cdot\mathsf{Set}\cdot\mu[\![P]\!](\mathsf{Pop})$
count	$\mathsf{Cnt}\,P$	$\mathsf{Sqr}\left(\{\![\,\lvert\mu[\![P]\!](\mathsf{Pop})\rvert\,]\!\}\right)$
sum	$\mathsf{Sum}\,P$	$\mathsf{Sqr}\left(\{\![\mathrm{sum}\cdot\pi_1\cdot\mu[\![P]\!](\mathsf{Pop})]\!\}\right)$
minimum	$\mathsf{Min}\,P$	$\mathsf{Sqr}\left(\{\![\mathrm{min}\cdot\pi_1\cdot\mu[\![P]\!](\mathsf{Pop})]\!\}\right)$
maximum	$\mathsf{Max}\,P$	$\mathsf{Sqr}\left(\{\![\mathrm{max}\cdot\pi_1\cdot\mu[\![P]\!](\mathsf{Pop})]\!\}\right)$
powerset	$\wp P$	$\mathsf{Sqr}\cdot\wp\cdot\pi_1\cdot\mu[\![P]\!](\mathsf{Pop})$

Example 3.2 *In the situation of the previous example:*

$$\mu[\![s^{\leftarrow}]\!](\mathsf{Pop}) = \begin{array}{|l|l|}\hline \{r : b_1, s : c_1\} & c_1 \\ \{r : b_2, s : \{e_1\}\} & \{e_1\} \\ \{r : b_3, s : \{e_2, e_3\}\} & \{e_2, e_3\} \\ \hline\end{array}$$

A path can be extended in several ways. Most elementary, is path extension by concatenation ($P \circ Q$). The extend operator $\diamond$ also applies to path expressions ($P \diamond Q$), and is built from the head values of both path expressions. Furthermore, the usual set operators ($P \cap Q$, $P \cup Q$ and $P - Q$) are available. These operators are formally described in the following table:

name	expr	$\mu[\![\mathsf{expr}]\!](\mathsf{Pop})$
concatenate	$P \circ Q$	$\mu[\![P]\!](\mathsf{Pop}) \circ \mu[\![Q]\!](\mathsf{Pop})$
extend	$P \diamond Q$	$\mu[\![P]\!](\mathsf{Pop}) \diamond \mu[\![Q]\!](\mathsf{Pop})$
intersection	$P \cap Q$	$\mu[\![P]\!](\mathsf{Pop}) \cap \mu[\![Q]\!](\mathsf{Pop})$
union	$P \cup Q$	$\mu[\![P]\!](\mathsf{Pop}) \cup \mu[\![Q]\!](\mathsf{Pop})$
minus	$P - Q$	$\mu[\![P]\!](\mathsf{Pop}) - \mu[\![Q]\!](\mathsf{Pop})$

Example 3.3 *In the situation of example 3.1:*

$$\mu[\![r \circ s^{\leftarrow}]\!](\mathsf{Pop}) = \begin{array}{|c|c|} b_1 & c_1 \\ b_2 & \{e_1\} \\ b_3 & \{e_2, e_3\} \end{array}$$

A more complex example making use of the implicit fact type between a power type and its element type is:

$$\mu[\![r \circ s^{\leftarrow} \circ \in_F^p \circ \in_F^{e\,\leftarrow}]\!](\mathsf{Pop}) = \begin{array}{|c|c|} b_2 & e_1 \\ b_3 & e_2 \\ b_3 & e_3 \end{array}$$

Special constructs are available for data type conversions. Grouping and ungrouping form the conversion between an object type and a corresponding power type. Ordering is used for the conversion of a path expression into a sequence.

name	expr	$\mu[\![\mathsf{expr}]\!](\mathsf{Pop})$
grouping	$\wp(P, G)$	see below
ungrouping	$\Upsilon(P)$	$\mathsf{Sqr} \cdot \biguplus \cdot \pi_1 \cdot \mu[\![P]\!](\mathsf{Pop})$
ordering	$\psi(P, G)$	see below

Grouping path expression P, according to grouping criterion G, is performed by the function $\wp(P, G)$. The elements to be grouped are obtained from the first component of path expression P. Path expression G specifies a grouping criterion for these elements. Suppose $g \in \pi_2 \cdot \mu[\![G]\!](\mathsf{Pop})$, then with g is associated the following class of elements:

$$K_g = \{\, x \in \pi_1 \cdot \mu[\![P]\!](\mathsf{Pop}) \mid \langle x, g \rangle \in \mu[\![G]\!](\mathsf{Pop}) \,\}$$

The result of grouping is now obtained as the set of all such classes, presented in the format that is used for the interpretation of path expressions:

$$\mu[\![\wp(P, G)]\!](\mathsf{Pop}) \quad = \quad \mathsf{Multi}(\{\, \langle K_g, g \rangle \mid g \in \pi_2 \cdot \mu[\![G]\!](\mathsf{Pop}) \wedge K_g \neq \varnothing \,\})$$

Sorting the result of path expression P into a single sequence, according to a sorting criterion S, can be achieved by applying ψ on P and S respectively. The sorting criterion may be weak (for example $S = \varnothing_{\mathcal{PE}}$), allowing more than one ordering of the elements, or too strong, for which any ordering fails. A sequence s is called compatible with sorting criterion S over P in population Pop if:

1. s contains all elements of $\pi_1 \cdot \mu[\![P]\!]\,(\mathsf{Pop})$ in the same frequency: $\mathsf{Lin}(s) = \pi_1 \cdot \mu[\![P]\!]\,(\mathsf{Pop})$,
2. the order of elements in s does not conflict with the ordering rules from S:

$$0 \le i < j < |s| \Rightarrow \exists_{y_1,y_2} [\langle s[i], y_1\rangle \in \mu[\![P]\!]\,(\mathsf{Pop}) \wedge \langle s[j], y_2\rangle \in \mu[\![P]\!]\,(\mathsf{Pop}) \wedge \langle y_2, y_1\rangle \notin \mu[\![S]\!]\,(\mathsf{Pop})]$$

The result of sorting now is defined as:

$$\mu[\![\psi(P,S)]\!]\,(\mathsf{Pop}) = \mathsf{Sqr}(\{\![s\uparrow^1 \mid s \text{ is compatible with } S \text{ over } P \text{ in } \mathsf{Pop}]\!\})$$

The following construction mechanism for path expressions corresponds to the transitive closure of a binary relation.

name	expr	$\mu[\![\mathsf{expr}]\!]\,(\mathsf{Pop})$
closure	P^+	$\mathsf{ds}\left(\bigcup_{n\in\mathbb{N}} \mu[\![\mathsf{closure}(n,P)]\!]\,(\mathsf{Pop})\right)$

The expression $\mathsf{closure}(n, P)$ represents a closure of path expression P in n steps and is recursively defined as follows:

$$\begin{aligned} \mathsf{closure}(0,P) &= P \\ \mathsf{closure}(n+1,P) &= \mathsf{closure}(n,P) \circ P \end{aligned}$$

A powerful operation on path expressions is the confluence operation. This operator is typically used when different sorts of information are to be integrated. For instance, name, day of birth, salary and address of an employee with a given employee number.

name	expr	$\mu[\![\mathsf{expr}]\!]\,(\mathsf{Pop})$
confluence	$[P_1,\ldots,P_n \mid Q]$	see below

If $P_1,\ldots,P_n,Q$ are path expressions then $[P_1,\ldots,P_n \mid Q]$ is a path expression corresponding to an n-ary relation called the confluence of $P_1,\ldots,P_n$. The meaning of this expression is:

$$\begin{aligned} &\mu[\![\ [P_1,\ldots,P_n \mid Q]\ ]\!]\,(\mathsf{Pop}) \\ &= \bigcup_{x\in\pi_1\cdot\mu[\![Q]\!]\,(\mathsf{Pop})} \{\![\langle\langle x_1,\ldots,x_n\rangle, x\rangle\uparrow^{k_1\times\ldots\times k_n} \mid \forall_{1\le i\le n} [\langle x_i, x\rangle \in^{k_i} \mu[\![P_i]\!]\,(\mathsf{Pop})]\ ]\!\} \end{aligned}$$

The condition in the confluence Q, is not mandatory. By using $1_{\mathcal{PE}}$, the condition is neutralised. As a shorthand, we define: $[P_1,\ldots,P_n] \equiv [P_1,\ldots,P_n \mid 1_{\mathcal{PE}}]$.

In order to define the active complement $\neg$ (see [28]) of a path expression, the set of active elements are introduced:

$$\mathsf{ActVals} = \bigcup_{x\in\mathcal{O}} x$$

The active complemement of a path expression P then, is defined as: $\neg P \equiv \mathsf{ActVals} - \!\!\!\!f\ P$. Path expressions are coerced to *multi* sets by the Rn function: $\mathsf{Rn} \equiv \pi_1 \cdot \mu$.

For path expressions having instances of power, sequence or composition types as front elements, a substitution operator exists. This operator substitutes the elements in these front elements, according to a second path expression. Therefore, a set of instances $\{a, b, c\}$ can be converted to a set $\{x, y, z\}$. Furthermore, a set, or sequence, of path expressions can be converted to a path expression consisting of sets, or sequences, of "ordinary" elements. This is achieved by means of the set or sequence constructor.

name	expr	$\mu[\![\text{expr}]\!]\,(\text{Pop})$
element substitution	$\delta(P,Q)$	see below
set constructor	$\{P_1,\ldots,P_n\}$	$\text{Sqr}\cdot\text{Multi}(\{\ \{x_1,\ldots,x_n\}\ \mid \forall_{1\le i\le n}[x_i\in\pi_1\cdot\mu[\![P_i]\!]\,(\text{Pop})]\ \})$
sequence constructor	$\langle P_1,\ldots,P_n\rangle$	$\text{Sqr}\cdot\text{Multi}(\{\ \langle x_1,\ldots,x_n\rangle\ \mid \forall_{1\le i\le n}[x_i\in\pi_1\cdot\mu[\![P_i]\!]\,(\text{Pop})]\ \})$

Usually the path expressions $P_1,\ldots,P_n$ in the set and sequence constructor will contain just one value. The definition of the element substitution operator is based on the $\text{subst}(p,f)$ operation, which substitutes the components of p by means of the substitution relation f. For instance,

$$\text{subst}(\{a,b,c\},\{\langle x,a\rangle,\langle y,b\rangle,\langle z,c\rangle\})=\{x,y,z\}$$

This leads to the following definition for the element substitution operator:

$$\mu[\![\delta(P,Q)]\!]\,(\text{Pop})=\bigcup_{\langle x,y\rangle\in^n\mu[\![P]\!]\,(\text{Pop})}\{\![\langle z,y\rangle\!\uparrow^n\mid z=\text{subst}(x,\mu[\![Q]\!]\,(\text{Pop}))]\!\}$$

4 Information Descriptors in LISA-D

In this section the abstract syntax and semantics of information descriptors in LISA-D (Language for Information Structure and Access Descriptions) are defined. A concrete syntax for LISA-D falls outside the scope of this paper. A concrete syntax will, however, allow several spellings of the elementary constructs, and also offer the opportunity to use so-called stopwords, i.e., words such as *'the'*, *'a'*.

Information descriptors form the basis of LISA-D, they are used for the specification of constraints (see section 4.6), updates (see section 4.7) and queries (see section 4.8). Most of the examples in this section are taken from a fragment of the so-called *Presidential Database*), regarding the election process of presidents from the USA. This example was a unified example in the special issue of Computing Surveys ([13]); the example was first enunciated in [41]. An excerpt of this schema is presented in figure 11.

4.1 The Underlying Naming Convention

In the previous sections the elements constituting an information structure were introduced as abstract concepts. The intention of the rest of this paper is to describe a language by which populations of information structures can be manipulated (by human beings), in terms of these abstract concepts (to be manipulated by machines). This language should lead to natural expressions. Typical for such languages is the richness to form sentences, even sentences that have no intuitive meaning. The language should be such that it allows for an elegant description for the information need of a user. This does not imply the exclusion of unelegant descriptions, independently of subjective ideas of elegance!

Object type naming

A first requirement is to verbalise the mathematical concepts of PSM via some set $\mathcal{N}$ of names. Object types are referenced by a unique name: $\text{ONm}:\mathcal{O}\rightarrowtail\mathcal{N}$, which is specified in the schema

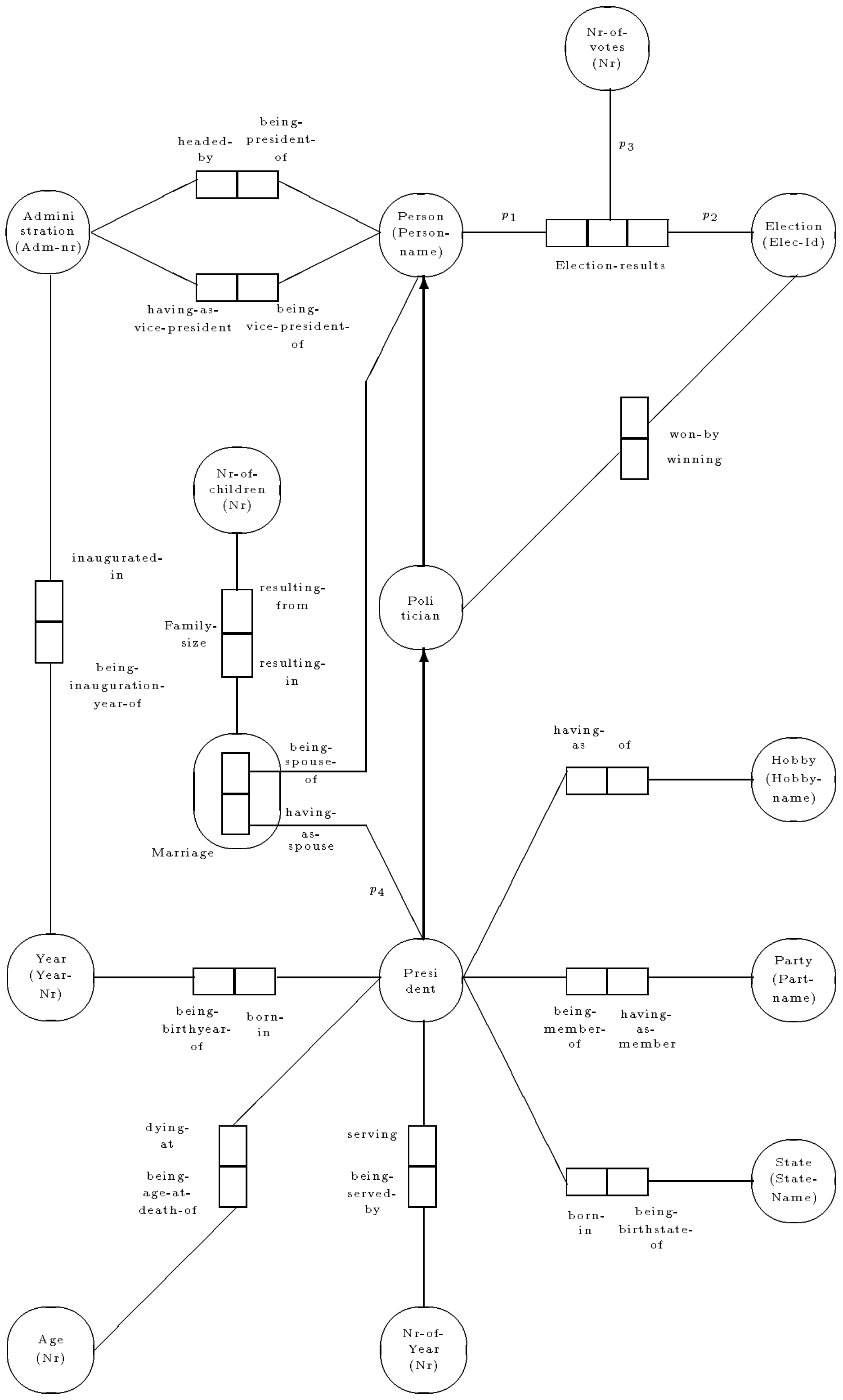


Figure 11: Part of an information structure regarding American presidents

upon their introduction. The (partial) function $\mathsf{Obj} : \mathcal{N} \rightarrowtail \mathcal{O}$ is the left-inverse of ONm, and relates object type names to their corresponding object type:

$$\forall_{x \in \mathcal{O}} [\mathsf{Obj}(\mathsf{ONm}(x)) = x]$$

In order to improve readability, x rather than $\mathsf{Obj}(x)$ will be written. From the context it will be clear whether x is used as an information descriptor, or as a shorthand for $\mathsf{Obj}(x)$.

Predicator naming

Predicators may have assigned a so-called *predicator name* via the (partial) function: $\mathsf{PNm} : \mathcal{P} \rightarrowtail \mathcal{N}$. Predicator names should be unique for predicators belonging to the same fact type. This, however, is not required for predicators of different fact types. The operator $. : \mathcal{N} \times \mathcal{N} \rightarrowtail \mathcal{P}$ retrieves the predicator that is associated with a given name within a fact type (if any):

$$\forall_{p \in \mathcal{P}} [\mathsf{ONm}(\mathsf{Fact}(p)).\, \mathsf{PNm}(p) = p]$$

For unique predicator names, the fact type name qualification may be omitted for readability. Finally, object type names and predicator names should be different.

Role naming

In binary versions of NIAM ([42]), special names are introduced for predicators, to form readable sentences over the information structure. These names, referred to as *role names*, are those names that occur in NIAM schemata close to roles. They are recorded by the (partial) function: $\mathsf{RNm} : \mathcal{P} \rightarrowtail \mathcal{N}$. In figure 11 role names are added to all predicators of binary fact types that are not a bridge type. Role names correspond to special connections (in the form of path expressions) through (binary) fact types. Such special connections are termed *connectors* in this paper. As an example, the sentence Hobby of President specifies all hobbies of presidents, while the sentence Hobby of President having-as-spouse Politician specifies all hobbies of presidents with a spouse involved in politics. In NIAM terminology, such sentences are called *deep structure sentences*. They form the basis of the NIAM modelling technique, and act as a natural language intermediate between application domain expert and system analyst. Such sentences can be interpreted uniquely as path expressions if each valid combination Object-Name Role-Name Object-Name has a unique interpretation in the information structure, and has no ambiguity with respect to its co-role (its co-predicator). This is called the *Role Identification Rule* (see [42]). A combination nx np ny is valid if there exists a predicator p such that:

$$\begin{aligned} \mathsf{ONm}(\mathsf{Base}(p)) &\sim \mathsf{nx} \\ \mathsf{RNm}(p) &= \mathsf{np} \end{aligned}$$

and a predicator $q \in \mathsf{Fact}(p)$, with $q \neq p$, such that:

$$\mathsf{ONm}(\mathsf{Base}(q)) \sim \mathsf{ny}$$

The combination of nx np ny has a unique interpretation in the information structure, if predicator p is unique. The combination of nx np ny is unambiguous with respect to its co-role if predicator q is also unique. The latter condition is automatically fulfilled if only binary fact types are allowed, and if the predicators of binary fact types have unique role names. In the non-binary case however, this latter condition is not fulfilled generally. Furthermore, the requirement of uniqueness of interpretation of role names within a fact type is sometimes felt to be too limiting (for example in the case of homogeneous symmetric binary relations it is natural that both role names are the same). This leads to a different interpretation of combinations nx np ny. For this purpose, the Path function will be introduced.

As a simple example of this new interpretation, consider the (ternary) election relation in figure 11. To find all persons contesting in an election, it would be preferable to formulate Person contesting-in Election. The name contesting-in then is used to denote the path expression $p_1 \circ p_2^{\leftarrow}$. Another example is Nr-of-votes of Person. In this statement, name of is to be interpreted, in the context of Nr-of-votes and Person as path expression $p_3 \circ p_2^{\leftarrow}$.

The administration of names

This leads to a generalisation of role names to a partial naming function $\mathsf{Path} : \mathcal{O} \times \mathcal{O} \times \mathcal{N} \rightarrowtail \mathcal{PE}$ that assigns, in a given context, a path expression to a name. The notation $\mathsf{Path}(x, y, n)\downarrow$ is used to indicate that $\mathsf{Path}(x, y, n)$ is defined for object types x, y and name n. The name n then can be used as a denotation for a path connecting x to y. In this case, name n is qualified as a *defined name*.

The function Path will be filled with a number of predefined names, and may be extended by the user of a LISA-D interpreter. In the sequel all predefined names, or keywords, are introduced. As a notational convention, keywords are written in capitals. For a start, the name function Path contains the following:

1. Names of (explicit) object types are defined names. The name $\mathsf{ONm}(x)$ of object type x stands for path expression x:
$$\mathsf{Path}(x, x, \mathsf{ONm}(x)) = x$$
For implicit object types (such as fact type $\in_x$) no names are assumed. Rather, special keywords are introduced to handle the manipulation of such object types.

2. Predicator names are defined names. If p is a predicator having a predicator name, then the predicator name $\mathsf{PNm}(p)$ describes a path from the base of p to its corresponding fact type:
$$\mathsf{Path}(\mathsf{Base}(p), \mathsf{Fact}(p), \mathsf{PNm}(p)) = p$$

3. Connector names are defined names. If predicator p of binary fact type $f = \{p, q\}$ has associated a connector name, then this name is interpreted as in RIDL:
$$\mathsf{Path}(\mathsf{Base}(p), \mathsf{Base}(q), \mathsf{RNm}(p)) = p \circ q^{\leftarrow}$$
provided f is not a homogeneous fact type with ambiguous role names (i.e. f consists of predicators p, q such that $\mathsf{Base}(p) = \mathsf{Base}(q)$, and also $\mathsf{RNm}(p) = \mathsf{RNm}(q)$). In that case the name receives its interpretation from both roles:
$$\mathsf{Path}(\mathsf{Base}(p), \mathsf{Base}(q), \mathsf{RNm}(p)) = p \circ q^{\leftarrow} \cup q \circ p^{\leftarrow}$$

4. Denotations for label values are defined names. This makes it possible to use such denotations as regular information descriptors. The denotation $\mathsf{CNm}(c)$ of constant c refers to the path expression c, describing a path from $\mathsf{SortOf}(c)$ to $\mathsf{SortOf}(c)$:
$$\mathsf{Path}(\mathsf{SortOf}(c), \mathsf{SortOf}(c), \mathsf{CNm}(c)) = c$$
The functions CNm and SortOf are introduced in the next section.

4.2 Integrating the concrete domains

In section 2.2 the link between an information structure and concrete domains has been described. In this section, this link is described in terms of *schema integration*. This results in a uniform approach both to the actual information structure, and the underlying domains. The resulting information structure is, however, not a proper information structure, as there may be population problems: some concrete domain may have an infinite size, while populations can only be finite.

Suppose $\mathcal{D} = \langle D, F \rangle$ is the underlying concrete domain structure, coupled to the information structure by the function $\mathsf{Dom} : \mathcal{L} \rightarrow D$. To make it possible to use the functions and relations from F (such as $<$ and $+$), the structure $\mathcal{D}$ will be incorporated in the information structure. This section describes the procedure.

The concrete structure $\mathcal{D}$ is predefined as a PSM-schema. The integration then is performed by considering the coupling function Dom specifying subtype relations as follows:

$$x \mathrel{\mathsf{Spec}} d \Leftrightarrow \mathsf{Dom}(x) = d$$

These subtype relations do not require subtype defining rules.

Example 4.1 *In figure 12 these (new) specialisation relations are shown for the schema in figure 10.*

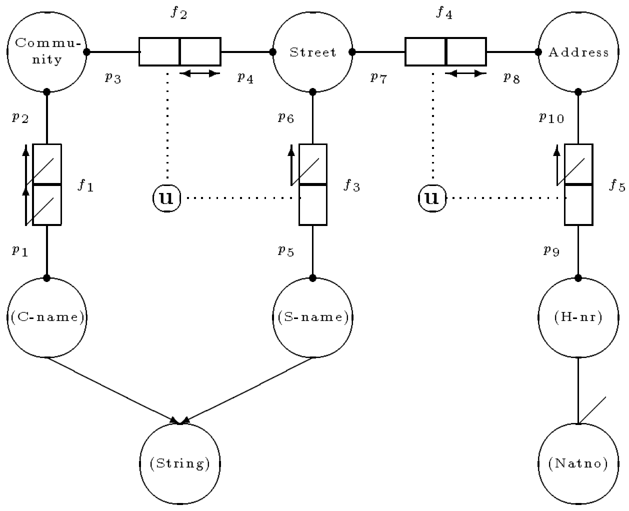


Figure 12: Associating concrete domains to label types

Besides its structure, the population of the concrete structure is also predefined, and may not be subject to change. For example, the domain Natno is populated with the set of all natural numbers, and the relation $<$ on the domain Natno is populated with the set of all tuples with first component smaller than the second component.

Appropriate names for the object types and predicators in the concrete structure are assumed as described in the previous section. On top of that, names (denotations) for concrete values are also assumed. However, as all concrete domains are considered mutually disjoint, each concrete value belongs to precisely one domain. Let SortOf be the function that returns the name of this domain for any concrete value:

$$\mathsf{SortOf}(c) = d \Leftrightarrow c \in d$$

In order to effectively use the functions and relations from F, they are considered as (concrete) fact types. A signature convention is assumed for each concrete fact type. This convention is a complete ordering of the predicators in any fact type (see function Ident in section 2.3), with the restriction that for functions the predicator corresponding to the result of the function, is first in this ordering. For example, the signature convention for the operator $+$ could be: result_+, $\mathsf{first\text{-}argument}_+$, $\mathsf{second\text{-}argument}_+$. The addition $5 + 3$ then is represented in the population of fact type $+$ by the tuple $\langle 8,5,3\rangle = \{\textsf{result}_+ : 8, \textsf{first-argument}_+ : 5, \textsf{second-argument}_+ : 3\}$. For relations, special naming conventions can be introduced, for example:

$$\textsf{Path}(\textsf{Natno}, \textsf{Natno}, <) = \textsf{first-argument}_< \circ \textsf{second-argument}_<^{\leftarrow}$$

4.3 Syntax and Semantics of Information Descriptors

As in natural languages, LISA-D has a very liberal syntax, especially for information descriptors. Some information descriptors are very specific, some are very general, others may not even make sense. Rather than excluding senseless information descriptors syntactically, the semantic interpretation will yield a void meaning for such constructs. Static semantics checks can easily detect such flaws in information descriptors.

LISA-D is built around a number of syntactical categories. In this section the category *Information Descriptor* is introduced. In later sections predicates, updates and queries will follow. The underlying elementary syntactical categories are: *Var* for simple variables and $\mathcal{N}$ for names. The naming conventions for instances of these syntactical categories are as follows: for *Information Descriptor*: P, P', P_1, P_2, O, Q, for *Var*: v, for $\mathcal{N}$: n.

The semantics of the syntactic category *Information Descriptor* is specified by the valuation function $\mathbb{D} : \textit{Information Descriptor} \times \textsf{ENV} \rightarrow \mathcal{PE}$ that maps information descriptors on path expressions. This valuation function is defined inductively on the structure of information descriptors. With each syntactic construct for the syntactic category *Information Descriptor* a recurrence rule is associated. $\textsf{ENV} : \textit{Var} \rightarrow \mathcal{PE}$ denotes the environment containing the current values of variables from the syntactical category *Var*. In a later section, the assignment of values to variables is discussed.

Atomic Information Descriptors

The foundation of information descriptors in LISA-D is formed by the defined names of $\mathcal{N}$, as introduced in the previous section. The meaning of a name is obtained as the sum of all possible interpretations as recorded by the Path-function. Variables form another elementary construct for information descriptors, as they are used to store intermediate results. The meaning of the elementary constructs is summarised by:

$$\begin{aligned} \mathbb{D}[\![n]\!](e) &= \bigcup_{\textsf{Path}\,(x,y,n)\downarrow} \textsf{Path}\,(x,y,n) \\ \mathbb{D}[\![v]\!](e) &= \begin{cases} e(v) & \text{if } e(v) \text{ is defined} \\ \varnothing_{\mathcal{PE}} & \text{otherwise} \end{cases} \end{aligned}$$

Some examples of atomic information descriptors are constant denotations (for example 'Roosevelt F.D.'), names for object types (Year), and role names (born-in). Note that the information descriptor born-in corresponds to two connectors (for simplicity, it is assumed that in the Presidential Database the same names are chosen for predicator names and role names; normally these names will be chosen differently) and two predicator names:

$$\begin{aligned} \mathbb{D}[\![\textsf{born-in}]\!](e) = \; & \textsf{Birthyear.born-in} \\ & \cup\ \textsf{Birthyear.born-in} \circ \textsf{being-birthyear-of}^{\leftarrow} \\ & \cup\ \textsf{Birthstate.born-in} \\ & \cup\ \textsf{Birthstate.born-in} \circ \textsf{being-birthstate-of}^{\leftarrow} \end{aligned}$$

Concatenation of Information Descriptors

Atomic information descriptors by themselves are rather limited. For instance, the atomic information descriptor $\mathsf{born\text{-}in}$ has a very general meaning. More fruitful information descriptors emerge by making combinations. The most fundamental way is concatenation of information descriptors:

$$\mathbb{D}[\![P_1\ P_2]\!](e) \quad = \quad \mathbb{D}[\![P_1]\!](e) \circ \mathbb{D}[\![P_2]\!](e)$$

A crucial effect of the concatenation operator is that it filters out the apparent intention of the user. Both information descriptors P_1 and P_2 may be very ambiguous, if they are used in the context of each other, much of the ambiguity will disappear. The strongest case is when both information descriptors have no meaning in each others context, i.e. when there is no connection from the one to the other. If there is no connection between information descriptors, concatenation will result in an information descriptor with a void meaning:

$$\begin{aligned}\mathbb{D}[\![\mathsf{born\text{-}in\ Hobby}]\!](e) &= \mathbb{D}[\![\mathsf{born\text{-}in}]\!](e) \circ \mathsf{Hobby}\\ &= \varnothing_{\mathcal{PE}}\end{aligned}$$

Note that it can be statically decided (i.e. without the need for evaluation) whether a connection exists between two information descriptors. This is expressed by the first filter property:

Theorem 4.1 (First Filter Property) Suppose n_1 and n_2 are names, then:

$$\mathbb{D}[\![n_1\ n_2]\!](e) = \bigcup_{z_1 \sim z_2} \mathsf{Path}(x, z_1, n_1) \circ \mathsf{Path}(z_2, y, n_2)$$

Proof: Suppose $z_1 \not\sim z_2$, then in each population Pop of information structure Σ (i.e. $\mathsf{IsPop}(\Sigma, \mathsf{Pop})$) z_1 and z_2 have no values in common (axiom **P1**): $\mathsf{Pop}(z_1) \cap \mathsf{Pop}(z_2) = \varnothing$. As a result, there is no contribution from $\mathsf{Path}(x, z_1, n_1) \circ \mathsf{Path}(z_2, y, n_2)$ to the result of $n_1\ n_2$ for any x and y.

□

As a next example, the information descriptor $\mathsf{born\text{-}in\ State}$ is composed by the concatenation of two atomic information descriptors.

$$\begin{aligned}\mathbb{D}[\![\mathsf{born\text{-}in\ State}]\!](e) &= \mathbb{D}[\![\mathsf{born\text{-}in}]\!](e) \circ \mathsf{State}\\ &= \mathsf{Birthstate.born\text{-}in} \circ \mathsf{being\text{-}birthstate\text{-}of}^{\leftarrow} \circ \mathsf{State}\end{aligned}$$

As this path expression is homogeneous (see section 3), it follows that the information descriptor $\mathsf{born\text{-}in\ State}$ has the same meaning as $\mathsf{President\ born\text{-}in\ State}$. The next example concatenates two atomic information descriptors, that both correspond to an object type:

$$\begin{aligned}\mathbb{D}[\![\mathsf{President\ Person}]\!](e) &= \mathsf{President} \circ \mathsf{Person}\\ &= \mathsf{President}\end{aligned}$$

Sometimes, parts of an information descriptors will be added just to make the expression readable by a human being. Semantically, there does not have to be a difference, as is stated in the second filter property:

Theorem 4.2 (Second Filter Property) *Suppose n_1 and n_2 are names of object types X_1 and X_2 respectively, then:*

$$X_1 \,\mathsf{Spec}\, X_2 \vee X_2 \,\mathsf{Gen}\, X_1 \Rightarrow n_1\ n_2 \equiv n_1$$

Proof: Suppose n_1 and n_2 are names of object types $X_1 = \mathsf{Obj}(n_1)$ and $X_2 = \mathsf{Obj}(n_2)$, such that $X_1 \,\mathsf{Spec}\, X_2 \vee X_2 \,\mathsf{Gen}\, X_1$, then in each population Pop of information structure Σ (i.e. $\mathsf{IsPop}(\Sigma, \mathsf{Pop})$): $\mathsf{Pop}(n_1) \subseteq \mathsf{Pop}(n_2)$. As a result, $\mathbb{D}[\![n_1\ n_2]\!]\,(e) = \mathbb{D}[\![n_1]\!]\,(e)$ in each environment e.

□

In case of objectification, the predicator name can be fruitfully employed to form fluent sentences. For example, suppose $\mathsf{PNm}(p_4) = \mathsf{in}$ instead of the name presented in figure 11. Then the information descriptor President in Marriage resulting-in Nr-of-children translates to a path expression connecting presidents with the corresponding number of children:

$$\begin{aligned}&\mathbb{D}[\![\mathsf{President\ in\ Marriage\ resulting\text{-}in\ Nr\text{-}of\text{-}children}]\!]\,(e)\\ &\quad = \mathsf{President} \circ \mathsf{in} \circ \mathsf{Marriage} \circ \mathsf{resulting\text{-}in} \circ \mathsf{resulting\text{-}from}^{\leftarrow} \circ \mathsf{Nr\text{-}of\text{-}children}\end{aligned}$$

Keywords as Information Descriptor

Until now only defined names are introduced for constants, object types and predicators. This naming serves as a verbalisation of the abstract information structure. In this section the keywords are introduced. An important purpose of keywords is to serve as an abstraction mechanism for handling implicit fact types. The keywords are summarized in figure 13.

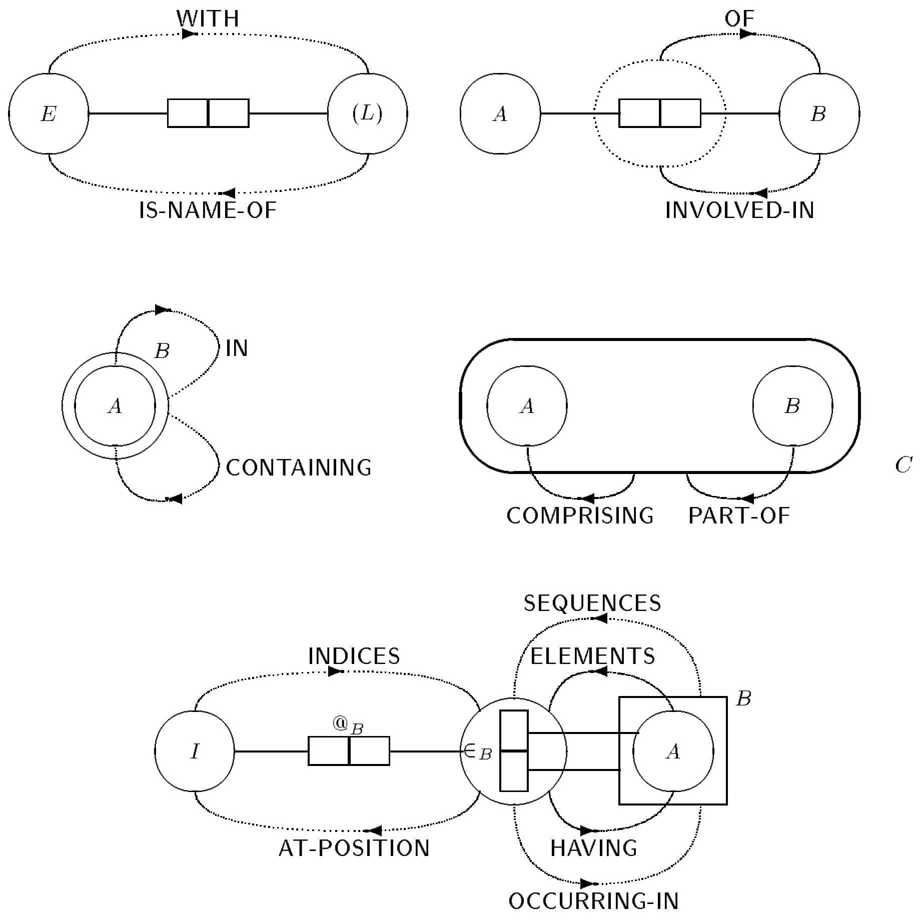


Figure 13: Keywords

Keywords for bridge types.

For relating object types to label types, the keywords WITH and IS-NAME-OF can be used. The keyword WITH relates object types via bridge types to label types, the keyword IS-NAME-OF is its inverse:

$$\text{for all } b \in \mathcal{B}\colon \begin{cases} \mathsf{Path}(\mathsf{Base}(\mathsf{abstr}(b)), \mathsf{Base}(\mathsf{concr}(b)), \mathsf{WITH}) & = & \mathsf{abstr}(b) \circ \mathsf{concr}(b)^{\leftarrow} \\ \mathsf{Path}(\mathsf{Base}(\mathsf{concr}(b)), \mathsf{Base}(\mathsf{abstr}(b)), \mathsf{IS\text{-}NAME\text{-}OF}) & = & \mathsf{concr}(b) \circ \mathsf{abstr}(b)^{\leftarrow} \end{cases}$$

This significantly reduces the need to have role names for predicators from bridge types. The keywords are particularly relevant when entity types are directly identifiable by single label types, which is the case for the entity types in figure 11, since in such cases bridge types are not visualised.

Example: President WITH Person-name 'Roosevelt F.D.' denotes the president with name 'Roosevelt F.D.'. Part-name IS-NAME-OF Party having-as-member President WITH Person-name 'Roosevelt F.D.' results in the *name* of all parties which have president Roosevelt registered as a member.

Keywords for predicator referencing.

The keywords OF and INVOLVED-IN are intended to facilitate the manipulation of objectified fact types. They are also useful as shorthands for predicator names. The keyword OF represents all relations between fact type instances and their constituent object type instances, the keyword INVOLVED-IN is its inverse:

$$\text{for all } x \in \mathcal{O} \text{ and } f \in \mathcal{F}\colon \begin{cases} \mathsf{Path}(x, f, \mathsf{INVOLVED\text{-}IN}) & = & \bigcup_{q \in f, \mathsf{Base}(q)=x} q \\ \mathsf{Path}(f, x, \mathsf{OF}) & = & \bigcup_{q \in f, \mathsf{Base}(q)=x} q^{\leftarrow} \end{cases}$$

The union operator in this definition is required to deal with fact types that contain predicators with identical bases.

Example: The information descriptor President INVOLVED-IN Marriage relates all married presidents to their respective marriages, while the information descriptor Marriage OF President relates all marriages to the presidents involved.

The combination of these keywords can be used to to unite all connections via fact types between two given object types. The information descriptor Administration INVOLVED-IN OF Person, for example, relates administrations to persons that were either president or vice-president of those administrations. The keyword ASSOCIATED-WITH serves as an abbreviation of this combination of keywords allowing for the formulation: Administration ASSOCIATED-WITH Person.

Keywords for power types.

The keywords IN and CONTAINING verbalise the implicit relation between a power type and its underlying element type. The keyword IN relates an element type with its associated power type(s), the keyword CONTAINING is its inverse:

$$\text{for all } x \in \mathcal{G}\colon \begin{cases} \mathsf{Path}(\mathsf{Elt}(x), x, \mathsf{IN}) & = & \in_x^e \circ \in_x^{p\leftarrow} \\ \mathsf{Path}(x, \mathsf{Elt}(x), \mathsf{CONTAINING}) & = & \in_x^p \circ \in_x^{e\leftarrow} \end{cases}$$

Example: Ships can be related to the convoy in which they sail (see figure 3) via the information descriptor Ship IN Convoy. The information descriptor Convoy CONTAINING Ship relates convoys to their constituent ships.

Keywords for sequence types.

The implicit fact types for sequence types capture the indexing relations for sequences. The keyword SEQUENCES is a generic name for predicators $\in_x^s$. Consequently, it relates sequences to the sequence membership relations (instances from $\in_x$) in which they occur. The keyword OCCURRING-IN, does the reverse, it relates sequence membership relations to the involved sequences:

$$\text{for all } x \in \mathcal{S}\text{:} \begin{cases} \mathsf{Path}(x, \in_x, \mathsf{SEQUENCES}) & = \in_x^s \\ \mathsf{Path}(\in_x, x, \mathsf{OCCURRING\text{-}IN}) & = \in_x^{s\,\leftarrow} \end{cases}$$

The keyword ELEMENTS is a generic name for predicators $\in_x^e$. Consequently, it relates elements to the sequence membership relations in which they occur. The keyword HAVING, does the reverse, it relates sequence membership relations to the involved elements:

$$\text{for all } x \in \mathcal{S}\text{:} \begin{cases} \mathsf{Path}(\mathsf{Elt}(x), \in_x, \mathsf{ELEMENTS}) & = \in_x^e \\ \mathsf{Path}(\in_x, \mathsf{Elt}(x), \mathsf{HAVING}) & = \in_x^{e\,\leftarrow} \end{cases}$$

The keyword INDICES relates indices to the associated sequence membership relations, while the keyword AT-POSITION does the reverse:

$$\text{for all } x \in \mathcal{S}\text{:} \begin{cases} \mathsf{Path}(I, \in_x, \mathsf{INDICES}) & = @_x^i \circ @_x^{s\,\leftarrow} \\ \mathsf{Path}(\in_x, I, \mathsf{AT\text{-}POSITION}) & = @_x^s \circ @_x^{i\,\leftarrow} \end{cases}$$

Example: Consider the schema of figure 4. The freight cars that are part of the train with T-code 'NE 99' are described by: Freight-car ELEMENTS OCCURRING-IN Freight-car-sequence of Train WITH T-code 'NE 99' . The trains containing freight car 'A702' are found by: Train has Freight-car-sequence SEQUENCES HAVING Freight-car WITH C-Code 'A702' . The head freight cars of all trains are found by: Freight-car ELEMENTS AT-POSITION 1.

Keywords for composition types.

The keywords COMPRISING and PART-OF deal with the relations between instances of schema types and instances of their constituent object types. The keyword COMPRISING relates instances of schema types to instances of object types of their decomposition, the keyword PART-OF does the reverse:

$$\text{for all } x \in \mathcal{C}, x \prec y\text{:} \begin{cases} \mathsf{Path}(x, y, \mathsf{COMPRISING}) & = \in_{x,y}^c \circ {\in_{x,y}^d}^{\leftarrow} \\ \mathsf{Path}(y, x, \mathsf{PART\text{-}OF}) & = \in_{x,y}^d \circ {\in_{x,y}^c}^{\leftarrow} \end{cases}$$

Example: Consider figure 5. The information descriptor Output PART-OF Activity-graph results in the output relations occurring in activity graphs. The information descriptor Activity-graph COMPRISING Output results in the activity graphs which contain at least one output relation.

Logical connectors and set operators

The LISA-D logical connectors AND-ALSO, OR-ELSE and BUT-NOT have a meaning very similar to that of their logical counterparts. The LISA-D set operators INTERSECTION, UNION and MINUS correspond to the well-known set operators intersection, union, and difference. The logical connectors ignore the values in the second component of the information descriptors involved, the set operators do not. The NOT operator is based on the active complement as defined for path expressions.

$$\begin{aligned} \mathbb{D}[\![P \,\mathsf{AND\text{-}ALSO}\, P']\!](e) &= \text{ꞙ}\,\mathbb{D}[\![P]\!](e) \cap \text{ꞙ}\,\mathbb{D}[\![P']\!](e) \\ \mathbb{D}[\![P \,\mathsf{INTERSECTION}\, P']\!](e) &= \mathbb{D}[\![P]\!](e) \cap \mathbb{D}[\![P']\!](e) \\ \mathbb{D}[\![P \,\mathsf{OR\text{-}ELSE}\, P']\!](e) &= \text{ꞙ}\,\mathbb{D}[\![P]\!](e) \cup \text{ꞙ}\,\mathbb{D}[\![P']\!](e) \end{aligned}$$

$$\begin{aligned}
\mathbb{D}[\![P\,\textsf{UNION}\,P']\!](e) &= \mathbb{D}[\![P]\!](e) \cup \mathbb{D}[\![P']\!](e)\\
\mathbb{D}[\![P\,\textsf{BUT-NOT}\,P']\!](e) &= \not{f}\,\mathbb{D}[\![P]\!](e) - \not{f}\,\mathbb{D}[\![P']\!](e)\\
\mathbb{D}[\![P\,\textsf{MINUS}\,P']\!](e) &= \mathbb{D}[\![P]\!](e) - \mathbb{D}[\![P']\!](e)\\
\mathbb{D}[\![\textsf{NOT}\,P]\!](e) &= \neg\mathbb{D}[\![P]\!](e)
\end{aligned}$$

To find the presidents that were born in California and served four years one can formulate: President(born-in State WITH State-name 'California' AND-ALSO serving Nr-of-years WITH Nr 4).

Remark 4.1 *The use of constructions such as* Year WITH Year-nr *and* Year-nr IS-NAME-OF Year *can be simplified by the introduction of special names:*

$$\begin{aligned}
\textsf{Path}(\textsf{Nr-of-years}, \textsf{Nr}, \textsf{Nr-of-years}) &= p \circ q^{\leftarrow}\\
\textsf{Path}(\textsf{Nr}, \textsf{Nr-of-years}, \textsf{Nr-of-years}) &= q \circ p^{\leftarrow}
\end{aligned}$$

if p and q are the predicators in this bridge type. This would allow the following construction: President(born-in State WITH State-name 'California' AND-ALSO serving Nr-of-years 4). *For all bridge types in the Presidential Database this extension of the* Path*-function is assumed in the remainder of this paper.*

Predicator inversion

Predicator names are introduced as information descriptors that correspond to a path expression consisting of that predicator. The inverse path is obtained via the following construction. Let n be the name of a predicator, then:

$$\mathbb{D}[\![n\,:]\!](e) = \bigcup_{\textsf{PNm}(p)=n} p^{\leftarrow}$$

Binary operators

In section 4.2 the introduction of binary relational operators was discussed. In this section binary operators are introduced as information descriptors, resulting in information descriptors as 45 + 20, or 45 + Year being-birthyear-of. The general format of such an expression is P_1 **n** P_2 where **n** is the name of any concrete binary operator (i.e., ternary fact type). The interpretation of this construct is as follows:

$$P_1\ \mathbf{n}\ P_2 \equiv n_0\ (n\ \textsf{AND-ALSO}\ n_1 :\ P_1)\ n_2 :\ P_2$$

where n_0, n_1, n_2 is the signature convention of the operator with name **n**.

Transitive closure

The information descriptor ANY-REPETITION-OF P describes the transitive closure of information descriptor P, and is defined as follows:

$$\mathbb{D}[\![\textsf{ANY-REPETITION-OF}\,P]\!](e) = (\mathbb{D}[\![P]\!](e))^{+}$$

As an example, consider the construction of formulas as described in example 2.5. Suppose V is an information descriptor describing some set of variables. All formulas that contain variables from V, but are not variables themselves, are obtained by the following information descriptor: Formula ANY-REPETITION-OF (having-left-arg UNION having-right-arg) V. The expression

ANY-REPETITION-OF (having-left-arg UNION having-right-arg) connects formulas to all their subformulas. By concatenating V, the restriction to variables from V is realised. The information descriptor Formula has no effect and is only added to improve readability.

As another example of the use of the transitive closure consider figure 5. According to this schema, activities may have a decomposition, consisting of substates and subactivities. Subactivities may have a decomposition as well. The relation between activities, and their corresponding subactivities, subsubactivities, etc., is captured by the following expression: ANY-REPETITION-OF (Activity being-decomposed-into Activity-graph COMPRISING Activity). This information descriptor relates activities to the activities occurring in their direct or indirect decompositions.

Correlation

In order to find the presidents who where inaugurated at an age younger than 45 years, i.e. inaugurated at least once within 45 years of *their* birth year, a convenient formulation is: President being-president-of Administration inaugurated-in Year $\leq$ 45 + Year being-birthyear-of THAT President This is called a correlation expression. A correlation expression cannot be formulated using the primitives introduced so far. The formal semantics of correlation expressions is defined as:

$$\mathbb{D}[\![P\ \mathsf{THAT}\ O]\!](e) \quad = \quad \mathbb{D}[\![P\ O]\!](e) \cap \mathbb{D}[\![O]\!](e)$$

Usually, the second information descriptor involved (i.e. O) is the name of a object type.

Type coercions

In LISA-D there exist some explicit forms of object type coercion. These can be divided into two groups:

1. Conversion of the population of an information descriptor to a single value. This value can again be used as an information descriptor.
2. Conversion of the population of an information descriptor to a population of a different type.

These coercions are discussed successively. Coercions that lead to a single value of some label type typically perform some computation.

1. The function NUMBER-OF counts the number of elements (including duplicates!) occurring in an information descriptor.

$$\mathbb{D}[\![\mathsf{NUMBER\text{-}OF}\ P]\!](e) \quad = \quad \mathsf{Cnt}(\mathbb{D}[\![P]\!](e))$$

   The number of presidents that were born in Virginia is given by: NUMBER-OF President born-in State 'Virginia'.

2. The function SUM adds the elements occurring in the first component of an information descriptor (including duplicates). This function is only applicable if addition is defined for the elements in the first component of the involved information descriptor.

$$\mathbb{D}[\![\mathsf{SUM}\ P]\!](e) \quad = \quad \mathsf{Sum}(\mathbb{D}[\![P]\!](e))$$

   The total number of children of presidents is found by: SUM Nr IS-NAME-OF Nr-of-children resulting-from Marriage.

3. The functions MIN and MAX calculate the minimal and the maximal element occurring in the first component of an information descriptor. These functions require the existence of an ordering on the elements occurring in the first component of the involved information descriptor.

$$\begin{aligned}\mathbb{D}[\![\mathsf{MIN}\, P]\!](e) &= \mathsf{Min}(\mathbb{D}[\![P]\!](e)) \\ \mathbb{D}[\![\mathsf{MAX}\, P]\!](e) &= \mathsf{Max}(\mathbb{D}[\![P]\!](e))\end{aligned}$$

The highest age of death of a president is found by: MAX Nr OF Age being-age-at-death-of President.

For the second type of coercion the following operators are available:

1. Multiple occurrences are filtered from the result of an information descriptor by the use of the DISTINCT operator:

$$\mathbb{D}[\![\mathsf{DISTINCT}\, P]\!](e) = \mathsf{ds}(\mathbb{D}[\![P]\!](e))$$

An example of the application of this operator is DISTINCT State being-birthstate-of President as some states are birthstate of more than one president.

2. The elements in an information descriptor P can be grouped into sets, according to a certain grouping criterion Q, using the LISA-D group operator:

$$\mathbb{D}[\![\mathsf{GROUP}\ P\ \mathsf{BY}\ Q]\!](e) = \varphi(\mathbb{D}[\![P]\!](e), \mathbb{D}[\![Q]\!](e))$$

The information descriptor GROUP President BY President having-as Hobby groups presidents sharing a hobby.

3. The coercion from sets to elements from these sets is achieved by the UNITE operator. Naturally, it is required that the elements in the first component of the involved information descriptor are sets themselves.

$$\mathbb{D}[\![\mathsf{UNITE}\, P]\!](e) = \Upsilon(\mathbb{D}[\![P]\!](e))$$

For example, the information descriptor UNITE Convoy yields all ships sailing in any convoy.

4. The elements in an information descriptor P can be ordered, according to an ordering criterion Q, using the LISA-D sort operator:

$$\mathbb{D}[\![\mathsf{SORT}\ P\ \mathsf{BY}\ Q]\!](e) = \psi(\mathbb{D}[\![P]\!](e), \mathbb{D}[\![Q]\!](e))$$

The information descriptor SORT President dying-at Age BY Age < Age orders presidents on their age of death.

Generators are operators required for the formulation of special types of constraints.

$$\begin{aligned}\mathbb{D}[\![P\,\mathsf{PAIRED\text{-}WITH}\, P']\!](e) &= \mathbb{D}[\![P]\!](e) \diamond \mathbb{D}[\![P']\!](e) \\ \mathbb{D}[\![\mathsf{ALL\text{-}SUBSETS\text{-}OF}\, P]\!](e) &= \wp(\mathbb{D}[\![P]\!](e))\end{aligned}$$

As an example, the information descriptor President PAIRED-WITH State pairs all presidents with all states, and ALL-SUBSETS-OF Ship yields all possible sets of ships (see figure 3). Obviously, all convoys are part of this information descriptor.

4.4 Assignments

A convenient mechanism to reduce the complexity of expressions is the assignment of subexpressions to variables. The format of an assignment is:

$$\mathsf{LET}\ v\ \mathsf{BE}\ P$$

The effect of such an assignment is a change of the environment. A special operator $\oplus$ is introduced to record such changes. For environment e, $e' = e \oplus \{x \leftarrow c\}$, denotes the same environment as e except for variable x: $e'(x) = c$.

The semantics of assignments is given by the valuation function

$$\mathbb{A} : \text{Assignment } \times \mathsf{ENV} \rightarrow \mathsf{ENV}$$

which is defined as:

$$\mathbb{A}[\![\mathsf{LET}\ v\ \mathsf{BE}\ P]\!]\,(e) \quad = \quad e \oplus \{v \leftarrow \mathbb{D}[\![P]\!]\,(e)\}$$

The meaning of an assignment A in the context of an information descriptor is:

$$\mathbb{D}[\![A; P]\!]\,(e) \quad = \quad \mathbb{D}[\![P]\!]\,(\mathbb{A}[\![A]\!]\,(e))$$

The following assignment may serve as an illustration:
LET Old-Presidents BE President dying-at Age $>$ 90

4.5 Denotations

In this section, constructions are introduced that facilitate the denotation of object instances used in information descriptors considerably. For this purpose, structured constants are introduced via the syntactical category *Constant Denotation*, with the following abstract syntax:

c	*constants*
v	*variables*
$d_1, \ldots, d_k$	*denotation of entities*
$[d_1, \ldots, d_k]$	*denotation of facts*
$[q_1 = d_1, \ldots, q_k = d_k]$	*alternative denotation of facts*
$\{d_1 \ldots, d_k\}$	*denotation of power type instances*
$\langle d_1, \ldots, d_k \rangle$	*denotation of sequence type instances*

where c is a *Constant name*, $v \in$ *Var*, d_j is a *Constant Denotation* and $q_i \in \mathsf{ran}(\mathsf{PNm})$.

Values of a label type named L can be used in information descriptors as follows:

$$\mathbb{D}[\![L : c]\!]\,(e) = \mathbb{D}[\![L\ c]\!]\,(e)$$

The expression Person-name:'Eisenhower D D', for example, is a valid information descriptor.

Consider figure 10. To denote a concrete address, while only using the constructs that have been introduced so far, one would have to write:

```
Address(in Street(in Community WITH C-name 'New York'
                  AND-ALSO
                  WITH S-name 'Fifth Avenue') AND-ALSO WITH  H-nr 17)
```

where it is assumed that $\mathsf{RNm}(p_8) = \mathsf{RNm}(p_4) = \mathsf{in}$.

Obviously, one would prefer to write:

$$\mathsf{Address: 'New\ York', 'Fifth\ Avenue', 17}$$

This is an example of an entity denotation. The formal definition of entity denotations uses the functions Ident and Copred introduced in section 2.3. If E is the name of an entity type, then:

$$\mathbb{D}[\![E : d_1, \ldots, d_k]\!]\,(e) \quad = \quad \mathsf{Obj}(E) \circ \bigcap_{i=1}^{k} \digamma(\mathsf{Copred}(p_i) \circ p_i{}^{\leftarrow} \circ \mathbb{D}[\![B_i :\ d_i]\!]\,(e))$$

where $p_i = \mathsf{Ident}(\mathsf{Obj}(E))[i]$ and $B_i = \mathsf{ONm}(\mathsf{Base}(p_i))$.

The function Ident has been extended to fact types in section 2.3. This extension allows for the denotation of fact type instances as sequences of values. The ordering as defined in the function Ident can then be used to determine which value corresponds to which base. An instance of a fact type named F can therefore be denoted as a structured constant of the form $[d_1, \ldots, d_k]$. The formal interpretation is given by:

$$\mathbb{D}[\![F : [d_1, \ldots, d_k]]\!]\,(e) \quad = \quad \mathsf{Obj}(F) \circ \bigcap_{i=1}^{k} \digamma(p_i{}^{\leftarrow} \circ \mathbb{D}[\![N_i :\ d_i]\!]\,(e))$$

where $p_i = \mathsf{Ident}(\mathsf{Obj}(F))[i]$ and $N_i = \mathsf{ONm}(\mathsf{Base}(p_i))$.

For example, president Eisenhower was president during administration 49. The corresponding instance of fact type Admin-pers can be denoted as:

$$\mathsf{Admin\text{-}pers : [49, 'Eisenhower\ D\ D']}$$

if

$$\mathsf{Ident(Obj(Admin\text{-}pers))} = \langle \mathsf{Admin\text{-}pers.headed\text{-}by}, \mathsf{Admin\text{-}pers.being\text{-}president\text{-}of} \rangle$$

The names of the predicators of a fact type can also be used in the denotation of its instances. In this case, fact type instances of a fact type named F are denoted as structured constants of the form $[q_1 = d_1, \ldots, q_k = d_k]$, where $q_1, \ldots, q_k$ are the names of the predicators of F. The formal interpretation is:

$$\mathbb{D}[\![F : [q_1 = d_1, \ldots, q_k = d_k]]\!]\,(e) \quad = \quad \mathsf{Obj}(F) \circ \bigcap_{i=1}^{k} \digamma(F.q_i{}^{\leftarrow} \circ \mathbb{D}[\![N_i :\ d_i]\!]\,(e))$$

where $N_i = \mathsf{ONm}(\mathsf{Base}(F.q_i))$.

The fact type instance of the previous example can be denoted as:

$$\mathsf{Admin\text{-}pers : [headed\text{-}by = 49, being\text{-}president\text{-}of = 'Eisenhower\ D\ D']}$$

Evidently, the advantage of this new type of denotation is that the assignments in the function Ident need not be known. However, this example demonstrates that denotations of this new form can be far less elegant.

The denotation of an instance of a power type consists of a set of denotations of its elements:

$$\mathbb{D}[\![G : \{d_1, \ldots, d_k\}]\!]\,(e) \quad = \quad \mathsf{Obj}(G) \circ \left\{ \mathbb{D}[\![X :\ d_1]\!]\,(e), \ldots, \mathbb{D}[\![X :\ d_k]\!]\,(e) \right\}$$

where $X = \mathsf{ONm}(\mathsf{Elt}(\mathsf{Obj}(G)))$.

For example, a convoy (see figure 3) consisting of ships 'S101' and 'S102' (instances of label type *S-code*) can be denoted as:

$$\mathsf{Convoy : \{'101','102'\}}$$

The denotation of instances of a sequence type consists of a sequence of denotations of its elements:

$$\mathbb{D}[\![S : \langle d_1, \ldots, d_k \rangle]\!]\,(e) \;=\; \mathsf{Obj}(S) \circ \left\langle \mathbb{D}[\![X :\; d_1]\!]\,(e), \ldots, \mathbb{D}[\![X :\; d_k]\!]\,(e) \right\rangle$$

where $X = \mathsf{ONm}(\mathsf{Elt}(\mathsf{Obj}(S)))$.

A freight car sequence (see figure 4) consisting of freight cars 'FC96' and 'FC99' (instances of label type *FC-code*), respectively, can be denoted as:

$$\mathsf{Freight\text{-}car\text{-}sequence} : \langle \mathsf{'FC96'}, \mathsf{'FC99'} \rangle$$

4.6 Predicates

In this section the extension of LISA-D with the syntactic category *Predicate* is discussed. Information descriptors form the basis for this new category. The names C_1, C_2 are used to denote a predicate. The semantics of predicates is defined by the function $\mathbb{P} \;:\; \mathit{Predicate} \times \mathsf{POP} \times \mathsf{ENV} \;\longrightarrow$ Bool. The basis for LISA-D predicates is the test whether an information desciptor has an empty result. From this basic predicate new predicates can be formed in the usual way, using logical connectives and quantification:

$$\begin{aligned}
\mathbb{P}[\![P]\!]\,(\mathsf{Pop}, e) &= \mu[\![\mathbb{D}[\![P]\!]\,(e)]\!]\,(\mathsf{Pop}) \neq \varnothing \\
\mathbb{P}[\![C_1\,\mathsf{AND}\,C_2]\!]\,(\mathsf{Pop}, e) &= \mathbb{P}[\![C_1]\!]\,(\mathsf{Pop}, e) \wedge \mathbb{P}[\![C_2]\!]\,(\mathsf{Pop}, e) \\
\mathbb{P}[\![C_1\,\mathsf{OR}\,C_2]\!]\,(\mathsf{Pop}, e) &= \mathbb{P}[\![C_1]\!]\,(\mathsf{Pop}, e) \vee \mathbb{P}[\![C_2]\!]\,(\mathsf{Pop}, e) \\
\mathbb{P}[\![\mathsf{NO}\,C]\!]\,(\mathsf{Pop}, e) &= \neg\,\mathbb{P}[\![C]\!]\,(\mathsf{Pop}, e) \\
\mathbb{P}[\![\mathsf{FOR\text{-}EACH}\ x\ \mathsf{IN}\ P\ \mathsf{HOLDS}\ C]\!]\,(\mathsf{Pop}, e) &= \forall_{y \in \mathsf{Rn}[\![\mathbb{D}[\![P]\!]\,(e)]\!]\,(\mathsf{Pop})} \left[\mathbb{P}[\![C]\!]\,(\mathsf{Pop}, e \oplus \{x \leftarrow \{\![y]\!\}\,\})\right]
\end{aligned}$$

The construction $\{x \leftarrow \{\![y]\!\}\}$ is motivated, as each multiset is allowed as path expression. New constructs may be derived as usual, for example:

$$\mathsf{FOR\text{-}SOME}\ x\ \mathsf{IN}\ P\ \mathsf{HOLDS}\ C \equiv \mathsf{NO}\,\mathsf{FOR\text{-}EACH}\ x\ \mathsf{IN}\ P\ \mathsf{HOLDS}\ \ \mathsf{NO}\,C$$

As an example of the use of predicates, we consider the situation that some federal law forbids presidents to be younger than 20 years. This can be formulated as follows: NO President being-president-of Administration inaugurated-in Year $<$ 20 $+$ Year being-birthyear-of THAT President.

A more complex example in the context of activity graphs (see figure 5) is the rule that forbids recursive decomposition of activities (e.g. an activity containing itself as subactivity, either directly or indirectly). The relation between an activity and its subactivities (at any depth) was discussed in the previous section. This leads to the following predicate: NO Activity ANY-REPETITION-OF (Activity being-decomposed-into Activity-graph COMPRISING Activity) THAT Activity .

4.7 Updates

In this section the LISA-D constructs for updating populations are introduced. For a proper introduction, a partial ordering $\sqsubseteq$ on populations of an information structure is useful.

Definition 4.1 *Let $\mathcal{I}$ be an information structure and let* Pop *and* Pop$'$ *be populations of $\mathcal{I}$* ($\mathsf{IsPop}(\mathcal{I}, \mathsf{Pop})$ *and* $\mathsf{IsPop}(\mathcal{I}, \mathsf{Pop}')$)*, then* $\mathsf{Pop} \sqsubseteq \mathsf{Pop}'$ *if and only if:*

$$\forall_{x \in \mathcal{O}} \left[\mathsf{Pop}(x) \subseteq \mathsf{Pop}'(x)\right]$$

Clearly $\sqsubseteq$ is a reflexive partial ordering. The above definition makes it possible to speak of *minimal* (or *maximal*) *populations* with respect to an other population and a condition.

In this section, the syntactic category *Update statement* is introduced. The semantics of LISA-D update statements is given by the function $\mathbb{U} : \textit{Update statement} \times \mathsf{POP} \times \mathsf{ENV} \rightarrow \mathsf{POP}$, which operates on a population in some environment and yields an (updated) population. In LISA-D update statements either add or delete object instances to populations.

Adding instances to a population is performed by the add statement, with the format $\mathsf{ADD}\, P$, where P is any information descriptor. The meaning of this statement is to enforce a minimal extension of the current population, that populates P, i.e. a minimal extension Pop' of the current population Pop, such that information descriptor P has no empty result in the extended population Pop'. Formally, this meaning is expressed by: $\mathbb{U}[\![\mathsf{ADD}\ \ P]\!]\,(\mathsf{Pop}, e)$ is a minimal population Pop' such that:

1. $\mathsf{IsPop}(\mathcal{I}, \mathsf{Pop}')$,
2. $\mathsf{Pop} \sqsubseteq \mathsf{Pop}'$ and
3. $\mu[\![\mathbb{D}[\![P]\!]\,(e)]\!]\,(\mathsf{Pop}') \neq \varnothing$

As an example, the following statement adds the address stated in the beginning of section 4.5 to the current population:

$$\mathsf{ADD\,Address\colon\ 'New\ York'\ 'Fifth\ Avenue'\ 17}$$

If this address is not yet available in the current population, then some (arbitrary) abstract instance is added to the population of entity type Address. This abstract intance is connected (directly or indirectly) to the labels 'New York', 'Fifth Avenue' and 17. Note that if any of these label values is not present in the current population, then this label value is also added. This example shows why it is necessary to speak of *a* minimal population instead of *the* minimal polation: any abstract instance may be added, as long as the requirements are fulfilled.

It is a good convention to use object denotations as objective for the add statement. However, the definition of the add statement makes it possible to formulate such things as

$$\mathsf{ADD\,President}$$

This statement adds an arbitrary president if and only if there are no presidents in the population at hand. An other example is:

$$\mathsf{ADD\,President\ having\text{-}as\ Hobby}$$

This statement assigns an arbitrary hobby to an arbitrary president if and only if such a relation is not available in the current population. Besides, it may lead to the creation of a president, and the creation of a hobby.

Instances can be deleted from a population by the delete statement, with the format $\mathsf{DELETE}\, P$, where P is any information descriptor. The meaning of this statement is to enforce a minimal reduction of the current population, that unpopulates P, i.e., a maximal part Pop' of the current population Pop, such that information descriptor P has an empty result in the reduced population Pop'. Formally, this meaning is expressed by: $\mathbb{U}[\![\mathsf{DELETE}\ \ P]\!]\,(\mathsf{Pop}, e)$ is a maximal population Pop' such that:

1. $\mathsf{IsPop}(\mathcal{I}, \mathsf{Pop}')$,
2. $\mathsf{Pop}' \sqsubseteq \mathsf{Pop}$ and

3. $\mu[\![\mathbb{D}[\![P]\!](e)]\!](\mathsf{Pop}') = \varnothing$

As an example, the statement DELETE President will result in a population, in which the object type President has an empty population. The statement DELETE President having-as Hobby will empty the population of the fact type that relates presidents to their hobbies.

It should be noted that the population resulting from an update statement may not fulfil all constraints. To avoid constraint violations, transactions are introduced. A transaction is a sequence of update statements, enclosed between START-TRANSACTION and END-TRANSACTION. The constraints then serve as invariant relations (i.e., pre- and post-conditions) for these transactions.

4.8 Queries

Basically, queries in LISA-D are formulated using information descriptors. However, an extra language facility (the syntactic category *Query*) is required to formulate a query yielding multiple aspects of some object type. For example one may be interested in the hobbies, the age of death of, and the birth year of presidents from Texas. This is formulated as:

LIST Hobby of, Age being-age-of-death-of, Year being-birth-year-of, President born-in State: 'Texas'

This query will result in a Hobby, Age, Year triple for each president resulting from President born-in State: 'Texas'. The example shows the general format of a query: LIST $P_1, \ldots, P_n, P$.

However, one is not interested in the abstract entities representing Hobby, Age and Year, but in a proper denotation in terms of label values. Such a proper denotation is called the *name* of the entity value. Weak identification is a property, which guarantees a name for each object instantiation. The identification rules from section 2.3 provide a naming convention for all object types. In section 4.5 it is shown how the identification rules are specified within LISA-D. From this specification a naming convention $\mathsf{Nm} : \mathcal{O} \rightarrow \mathcal{PE}$ for object types is derived as follows:

$$\mathsf{Nm}(X) = \begin{cases} X & \text{if X is a label type} \\ \delta(X, \mathsf{Nm}(\mathsf{Elt}(X))) & \text{if } X \text{ is a power type} \\ & \text{or a sequence type} \\ \delta(X, \bigcup_{X \prec Y} \mathsf{Nm}(Y)) & \text{if } X \text{ is a composition type} \\ \left[\mathsf{Nm}(X_1) \circ P_1{}^{\leftarrow}, \ldots, \mathsf{Nm}(X_k) \circ P_k{}^{\leftarrow}\right] & \text{if } X \text{ is an object type, identified} \\ & \text{as } X(P_1\ X_1, \ldots, P_k\ X_k) \end{cases}$$

These standard names form a substitution mechanism to transform instances of abstract entities into concrete label values. The effect of the LIST-statement is to properly list these values.

The semantics of the syntactic category *Query* is specified by the valuation function $\mathbb{L} : \mathit{Query} \times \mathsf{ENV} \rightarrow \mathcal{PE}$ that maps queries on path expressions.

$$\mathbb{L}[\![\mathsf{LIST}\ P_1, \ldots, P_n, P]\!](e) \quad = \quad [\mathsf{StdNames} \circ \mathbb{D}[\![P_1]\!](e), \ldots, \mathsf{StdNames} \circ \mathbb{D}[\![P_n]\!](e) \mid \mathbb{D}[\![P]\!](e)]$$

where StdNames denotes *all* standard naming convention: $\mathsf{StdNames} = \bigcup_{X \in \mathcal{O}} \mathsf{Nm}(X)$. The filtering mechanism will filter out all proper names in the context of its associated path expression.

5 Conclusions and Further Research

In this paper the conceptual language LISA-D based on the data modelling technique PSM, has been introduced. In LISA-D constraints, queries and updates can be expressed in a way closely following the naming in the conceptual schema. This makes LISA-D statements (generally) easy to read and interpret intuitively. The formal foundation of LISA-D makes it possible to implement the language and formally proof properties. In [19], LISA-D and Task-Structures ([20]) have been integrated, resulting in HYDRA.

Further research is necessary to establish the expressive power of LISA-D and to provide the language with a more powerful typing mechanism to support static semantic checks. Research is being performed in the development of a version of LISA-D supporting the (on line) evolution of information systems ([12], [34], [11]) based on EVORM, an extention of PSM supporting evolution ([33]). Furthermore, research is conducted providing a better disclosure of the information stored in the information system ([6]), by means of an approach based on stratified hypermedia architecture ([5]). Currently a prototype implementation of LISA-D is being developed.

6 Acknowledgements

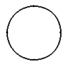
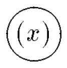
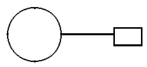
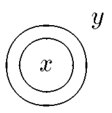
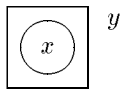
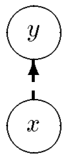
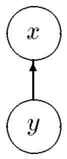
This work has been partially supported by:

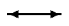
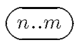
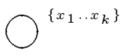
1. SERC (Software Engineering Research Centre), within the context of the SOCRATES project.
2. The Foundation for Computer Science in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), The Netherlands.

Special thanks go to Ernst Lippe, for many very useful ideas, and possibilities for further research.

Appendix: Legend of graphical symbols

This appendix contains an overview of the symbols for object types, generalisations and specialisations, and graphical constraints used in this paper.

	*object type*
	*label type x*
	role
	*predicator*
	*y power type of x*
	*y sequence type of x*
	*y is generalisation of x*
	*y is specialisation of x*

	*uniqueness constraint over a single fact type*
	uniqueness constraint over several fact types
	total role or cover constraint
	*occurrence frequency constraint or cardinality constraint*
	exclusion constraint
	membership constraint
	subset constraint
	equality constraint
	*enumeration constraint*

References

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

[2] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, The Netherlands, Revised Edition, 1984.

[3] E.A. Boiten. *Views of Formal Program Development*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1992.

[4] P. van Bommel, A.H.M. ter Hofstede, and Th.P. van der Weide. Semantics and Verification of Object-Role Models. *Information Systems*, 16(5):471–495, October 1991.

[5] P.D. Bruza and Th.P. van der Weide. Stratified Hypermedia Structures for Information Disclosure. *The Computer Journal*, 35(3):208–220, 1992.

[6] C.A.J. Burgers, H.A. Proper, and Th.P. van der Weide. An Information System organized as Stratified Hypermedia. Technical Report 93-12, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[7] P.P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[8] O.M.F. De Troyer. On Rule-Based Generation of Conceptual Database Updates. In *Proceedings of the IFIP TC 2 Working Conference on Knowledge and Data*, 1986.

[9] O.M.F. De Troyer, R. Meersman, and F. Ponsaert. RIDL User Guide. Research report, International Centre for Information Analysis Services, Control Data Belgium, Inc., Brussels, Belgium, 1984.

[10] O.M.F. De Troyer, R. Meersman, and P. Verlinden. RIDL* on the CRIS Case: A Workbench for NIAM. In T.W. Olle, A.A. Verrijn-Stuart, and L. Bhabuta, editors, *Computerized Assistance during the Information Systems Life Cycle*, pages 375 – 459, Amsterdam, The Netherlands, 1988. North-Holland/IFIP.

[11] E.D. Falkenberg, J.L.H. Oei, and H.A. Proper. A Conceptual Framework for Evolving Information Systems. In H.G. Sol and R.L. Crosslin, editors, *Dynamic Modelling of Information Systems II*, pages 353–375. North-Holland, Amsterdam, The Netherlands, 1992.

[12] E.D. Falkenberg, J.L.H. Oei, and H.A. Proper. Evolving Information Systems: Beyond Temporal Information Systems. In A.M. Tjoa and I. Ramos, editors, *Proceedings of the Data Base and Expert System Applications Conference (DEXA 92)*, pages 282–287, Valencia, Spain, September 1992. Springer-Verlag.

[13] J.P. Fry and E.H. Sibley. Evolution of Data-Base Management Systems. *Computing Surveys*, 8(1):7–42, 1976.

[14] A.L. Furtado and E.J. Neuhold. *Formal Techniques for Data Base Design*. Springer-Verlag, 1985.

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

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

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

[18] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

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

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

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

[22] A.H.M. ter Hofstede and Th.P. van der Weide. Formalisation of techniques: chopping down the methodology jungle. *Information and Software Technology*, 34(1):57–65, January 1992.

[23] 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.

[24] D.A. Jardine and J.J. van Griethuysen. A logic-based information modelling language. *Data & Knowledge Engineering*, 2:59–81, 1987.

[25] G.M. Kuper and M. Vardi. On the Expressive Power of the Logical Data Model. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 180–187, Austin, Texas, 1985. ACM Press.

[26] A. Levy. *Basic Set Theory.* Springer-Verlag, Berlin, Germany, 1979.

[27] A. Lew. *Computer Science: A Mathematical Introduction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[28] D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1988.

[29] R. Meersman. The RIDL Conceptual Language. Research report, International Centre for Information Analysis Services, Control Data Belgium, Inc., Brussels, Belgium, 1982.

[30] B. Meyer. *Introduction to the Theory of Programming Languages.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

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

[32] H. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development.* Springer-Verlag, Berlin, Germany, 1990.

[33] H.A. Proper and Th.P. van der Weide. EVORM: A Conceptual Modelling Technique for Evolving Application Domains. Technical Report 93-16, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[34] H.A. Proper and Th.P. van der Weide. Towards a General Theory for the Evolution of Application Models. In M.E. Orlowska and M. Papazoglou, editors, *Proceedings of the Fourth Australian Database Conference*, Advances in Database Research, pages 346–362. World Scientific, Brisbane, Australia, February 1993.

[35] G. Scheschonk. *Eine auf Petri-Netzen basierende Konstruktions, Analyse und (Teil)Verificationsmethode zur Modellierungsunterstützung bei der Entwicklung von Informationssystemen.* PhD thesis, Berlin University of Technology, Berlin, Germany, 1984. (In German).

[36] D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

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

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

[39] Y. Wand and R. Weber. An Ontological Analysis of some fundamental Information Systems Concepts. In *Proceedings of the Ninth International Conference on Information Systems*, pages 213–226, Minesota, Mineapolis, November 1988.

[40] Th.P. van der Weide, A.H.M. ter Hofstede, and P. van Bommel. Uniquest: Determining the Semantics of Complex Uniqueness Constraints. *The Computer Journal*, 35(2):148–156, April 1992.

[41] S.E. Willner, A.E. Bandurski, W.C. Gorhan, and M.A. Wallace. COMRADE data management system. In *Proceedings of the AFIPS National Computer Conference*, pages 339–345, Montvale, New Jersey, 1973. AFIPS Press.

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

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