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

(Received 23 July 1992; in final revised form 9 July 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 Conceptualization Principle ([1]) 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 Conceptualization 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 ([1]), 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 cannot be expressed formally, but need to be expressed in natural language, obviously causing interpretation problems ([2]). 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 Ref. [3], the conceptual data modelling technique PSM (Predicator Set Model) has been defined, which is capable of representing complex object structures without violating the Conceptualization Principle. PSM is an extension of PM (Predicator Model [4]) which on its turn is a formalisation of NIAM ([5–7]). 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 originated from the early 1970s, 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 \[8,9\]) 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 \([10]\).

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 \([11]\], FDM \([12]\) or INFOMOD \([13]\).

The organization 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 \(\text{Fact}: \mathcal{P} \rightarrow \mathcal{F}\) yields the fact type in which a given predicator is contained, and is defined by: \(\text{Fact}(p) = f \iff p \in f\).
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 \(\text{Base}: \mathcal{P} \rightarrow \mathcal{O}\). The base of a predicator is the object part of that predicator.
10. A function \(\text{Elt}: \mathcal{G} \cup \mathcal{S} \rightarrow \mathcal{O}\). This function yields the element type of power types and sequence types.

A relation \(\prec \subseteq \mathcal{C} \times \mathcal{O}\). This relation describes the decomposition of schema types.
12. A partial order \(\text{Spec}\) on object types, capturing specialization.
13. A function \(\text{Pater Familias}\): \(\mathcal{O} \rightarrow \mathcal{O}\) yielding the Pater Familias of a given object type.
14. A partial order \(\text{Gen}\) on object types, expressing generalization.

In this approach, the instances of object types are not part of the information structure. Instantiations (populations) will be introduced in Section 2.2.
Example 2.1. Figure 1 shows an information structure diagram visualizing the information structure that consists of:

\[ \mathcal{P} = \{p, q, r, s, t, u, v, w, x\} \quad \mathcal{Q} = \{A, B, C, D, E, f, g, h, i\} \]

\[ \mathcal{F} = \{f, g, h, i\} \quad \mathcal{G} = \{E\} \]

\[ \mathcal{S} = \{A, B, C, D\} \quad \mathcal{L} = \{F\} \]

where \( f = \{p, q\} \), \( g = \{r, s, t\} \), \( h = \{u, v\} \), \( i = \{w, x\} \). With respect to the predicators: \( \text{Base}(p) = B \), \( \text{Base}(q) = A \), \( \text{Base}(r) = f \), etc. Finally, \( \text{Elt}(E) = A \), \( \text{Spec} \) and \( \text{Gen} \) are empty, and \( \bigcap(x) = x \) 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) \( \mathcal{L}, \mathcal{F}, \mathcal{Q}, \mathcal{P}, \mathcal{G} \) and \( \mathcal{S} \) form a partition of \( \mathcal{Q} \).

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 Fig. 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 Fig. 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 Fig. 2, see Ref. [4]).

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 Bridge\( (f) \) qualifies fact type \( f \) as a bridge type, and is an abbreviation for the expression:

\[
\exists_{\mathcal{L}} \{ f = \{ p, q \} \land \text{Base}(p) \in \mathcal{L} \land \text{Base}(q) \notin \mathcal{L} \}
\]

\( \mathcal{L} \) 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:

\[
(\text{PSM2}) \text{Base}(p) \in \mathcal{L} \Rightarrow \text{Bridge} (\text{Fact}(p)).
\]

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

Example 2.2. In Fig. I, \( i \) is a bridge type with \( \text{concr}(i) = x \) and \( \text{abstr}(i) = w \).

2.1.3. Power typing. The concept of power type in PSM forms the data modelling pendant of power sets in conventional set theory ([14]). This notion is the same as the notion of grouping as introduced in the IFO data model [15]. 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).

An example of power typing is the Convoy Problem (based on (Ref. [16]), depicted in Fig. 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 constraint types can be found in Ref. [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 Ref. [17]), without violating the Conceptualization Principle.

The element type of a power type is found by the function \( \text{Elt} \). The relation between a power type \( x \) and its element type \( \text{Elt}(x) \) is recorded in the fact type \( \varepsilon_{x} = \{ \varepsilon^{x}_{x}, \varepsilon^{y}_{y} \} \), where \( \text{Base}(\varepsilon^{x}_{x}) = x \) and \( \text{Base}(\varepsilon^{y}_{y}) = \text{Elt}(x) \). This relation is assumed to be available for each power type. Usually \( \varepsilon_{x} \) 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 \( \varepsilon_{x} \) to an associated polymorphic access operator.

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

\[
(\text{PSM3}) \text{Elt}(x) \notin \mathcal{L}.
\]

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 generalization (see Ref. [3]). 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.

![Fig. 3. A single example of a power type.](image-url)
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 Fig. 4.

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 Elt(x) is recorded in the (implicit) fact type $\varepsilon_x = \{\varepsilon'_x, \varepsilon''_x\}$, where $\text{Base}(\varepsilon'_x) = x$ and $\text{Base}(\varepsilon''_x) = \text{Elt}(x)$. Contrary to power types, this relation $\varepsilon_x$ is augmented with the position of the element in the sequence, via the (implicit) fact type $\@_x = \{\@'_x, \@''_x\}$, where $\text{Base}(\@'_x) = \varepsilon_x$ and $\text{Base}(\@''_x) = I$. 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 to 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 recognized. Though in Ref. [3] 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.

Example 2.4. Activity Graphs are a well-known modelling technique for processes (see Ref. [18]). 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. Ref. [19]), 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 Fig. 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 underlying information structure $\mathcal{S}_x$ for a schema type $x$ is derived from the object types into which $x$ is decomposed: $\mathcal{S}_x = \{y \in \mathcal{O} | x \prec y\}$. Analogously, the special object
classes \( \mathcal{F}, \mathcal{G}, \mathcal{S}, \mathcal{D}, \mathcal{E}, \) and \( \mathcal{B} \), can be derived. The functions \( \text{Base}, \ Elt, \ <, \ \text{Spec}, \ \cap, \) and \( \text{Gen} \), are obtained by restriction to object types within \( \mathcal{O} \). In order to be a proper decomposition, the underlying information structure should form an information structure on its own:

\( (\text{PSM4}) \) (structural nesting) \( x \in \mathcal{E} \Rightarrow \mathcal{S}_x \) is a PSM information structure.

With each schema type \( x \) and each object type \( y \) in its decomposition, an (implicit) fact type \( e_{x,y} = \{ e_x, e_y \} \) is associated, where \( \text{Base}(e_x) = x \) and \( \text{Base}(e_y) = y \). This fact type will enable the transition from a schema object to an object from its decomposition.

2.1.6. Specialization. Specialization, referred to as subtyping in NIAM, is a mechanism for representing one or more (possibly overlapping) subtypes of an object type. Specialization 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 \textit{Adults}, i.e. \textit{Persons} with an \textit{Age} greater or equal than 18, one is interested in the \textit{Cars} they own. This situation is captured by the PSM schema in Fig. 6.

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

\[
\text{Adult} = \text{Person has Age } \geq 18.
\]

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

Specialization relations are organized in so-called specialization “hierarchies”. A specialization 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 \textit{pater familias} (see Ref. [20]). In the example of Fig. 6, the pater familias of \textit{Adult} is \textit{Person}.

Objects inherit all properties from their ancestors in the specialization hierarchy. This characteristic of specialization excludes nonentity 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 nonentity types themselves are specialized. Consequently, nonentity types always act as \textit{pater familias}. For an in depth discussion of specialization, we refer to Ref. [21].

The concept of specialization is introduced as a partial order (asymmetric and transitive) \( \text{Spec} \) on object types, with the convention that a \( \text{Spec} b \) is interpreted as: \( a \) is a subtype (specialization) 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 specialization of label types:

\[
(\text{PSM5}) \) (strictness) \( \text{Spec} \leq \mathcal{B} \times (\mathcal{O} - \mathcal{L}) \)
(\text{PSM6}) \) (asymmetry) \( a \text{ Spec } b \Rightarrow \neg b \text{ Spec } a \)
(\text{PSM7}) \) (transitivity) \( a \text{ Spec } b \land b \text{ Spec } c \Rightarrow a \text{ Spec } c \)

\[
\begin{align*}
\text{Person} & \quad \text{Age} \\
& \quad \text{Adult} \quad \text{Car} \\
& \quad \text{has is-of} \\
& \quad \text{owns owned-by}
\end{align*}
\]

Fig. 6. Example of specialization.
Each specialization hierarchy has a unique top element, the pater familias of the object types in this hierarchy. The *pater familias* is found by the function \( \cap : \emptyset \rightarrow \emptyset \) (which is similar to the top operator from lattice theory). This function has the following properties:

(PSM8) *(cohesion)* \( a \text{ Spec } b \Rightarrow \cap(a) = \cap(b) \)

(PSM9) *(ancestor)* \( a \neq \cap(a) \Rightarrow a \text{ Spec } \cap(a) \)

In the remainder \( \text{spec}(a) \) will be used as an abbreviation for \( \exists x \in x \text{ Spec } a \). 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 Ref. [3]).

**Lemma 2.1.** \( \neg \cap(a) \text{ Spec } b \).

**Corollary 2.1.** Idempotency of \( \cap : \cap(\cap(a)) = \cap(a) \).

2.1.7. **Generalization.** Generalization is a mechanism that allows for the creation of new object types by uniting existing object types. Generalization is to be applied when different object types play identical roles in fact types. Contrary to what its name suggests, generalization is not the inverse of specialization. Specialization and generalization originate from different axioms in set theory [3] and therefore have a different expressive power.

For generalization it typically is required that the generalized object type is covered by its constituent object types (or *specifiers*). Therefore, a decision criterion as in the case of specialization (the subtype defining rule) is not necessary. Furthermore, properties are inherited "upward" in a generalization hierarchy instead of "downward", which is the case for specialization (see also Ref. [15]). This also implies that the identification of a generalized object type depends on the identification of its specifiers. From the nature of generalization, it is apparent that a nonentity type cannot be a generalized object type.

**Example 2.5.** In Fig. 7 we see an example of generalization. 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 \( f \)).

This example also shows that generalization can be used to define recursive object types. This is not possible in the IFO data model [15], where object types are hierarchical structures. In the Logical Data Model (see Ref. [22]), however, object types are directed graphs, which may contain cycles.

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

(PSM10) *(strictness)* \( \text{Gen} \subseteq \emptyset \times (\emptyset - \mathcal{L}) \)

(PSM11) *(asymmetry)* \( a \text{ Gen } b \Rightarrow \neg b \text{ Gen } a \)

(PSM12) *(transitivity)* \( a \text{ Gen } b \land b \text{ Gen } c \Rightarrow a \text{ Gen } c \)

![Fig. 7. Example of generalization.](image)
In the remainder \( \text{gen}(a) \) will be used as an abbreviation for \( \exists x \in \alpha \{ a \in \text{Gen} x \} \). Generalization and specialization can be conflicting due to their inheritance structure. To avoid such conflicts, generalized object types are required to be *pater familias*:

\[
(\text{PSM13}) \quad \text{gen}(a) \Rightarrow \bigcap(a) = a
\]

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 \( \bigcap(x) \). As another example, suppose \( x \in \text{Gen} y \), 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 formalized in the concept of *type relatedness*.

Formally type relatedness is captured by a binary relation \( \sim \) on \( \emptyset \). Two object types are type related if and only if this can be proved from the following derivation rules:

\[
\begin{align*}
(T1) & \quad \vdash x \sim x \\
(T2) & \quad x \sim y \vdash y \sim x \\
(T3) & \quad \vdash x \text{ Spec } y \land y \sim z \vdash x \sim z \\
(T4) & \quad \vdash x \text{ Gen } y \land y \sim z \vdash x \sim z \\
(T5) & \quad \vdash x, y \in \mathcal{S} \land \text{Elt}(x) \sim \text{Elt}(y) \vdash x \sim y \\
(T6) & \quad \vdash x, y \in \mathcal{S} \land \text{Elt}(x) \sim \text{Elt}(y) \vdash x \sim y \\
(T7) & \quad \emptyset \vdash x \sim y
\end{align*}
\]

Example 2.6. In Fig. 8 the only object types that are 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 (UoD). A *state* of the UoD then corresponds to a so-called instantiation or population of the information structure, and vice versa. The idea of states was previously mentioned in Refs [23–25]. Furthermore, a state transition of the UoD 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 \( \text{Pop} \) of an information structure \( \mathcal{S} \) is a value assignment of sets of instances to the object types in \( \emptyset \), satisfying the rules that will follow in the rest of this section. This is denoted as \( \text{IsPop}(\mathcal{S}, \text{Pop}) \).

\( \text{Pop} \) then is a mapping \( \text{Pop} : \emptyset \rightarrow \mathcal{P}(\Omega) \), where \( \Omega \) is the universe of instances that can occur in the population of an information structure \( \mathcal{S} \). This universe of instances is defined in Definition 2.1. The set of all populations is defined as \( \text{POP} = \{ \text{Pop} \} \).

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 \( \text{Dom} : \mathcal{S} \rightarrow \mathcal{D} \). 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 \( D \).

![Fig. 8. Example of information structure.](image-url)
Definition 2.1. The universe of instances \( \Omega \) 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 \( \Theta \) 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 predicate \( p_i \). 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. \( x_1, \ldots, x_n \in \Omega \wedge o_1, \ldots, o_n \in \Theta \Rightarrow \{o_1 : x_1, \ldots, o_n : x_n\} \in \Omega \). 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.

\[(P1) \quad x, y \notin L \land x \neq y \Rightarrow \text{Pop}(x) \cap \text{Pop}(y) = \emptyset\]

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

\[(P2) \quad x \in L \Rightarrow \text{Pop}(x) \subseteq \text{Dom}(x)\]

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

\[(P3) \quad x \in \mathcal{E} \land \text{IsRoot}(x) \Rightarrow \text{Pop}(x) \subseteq \Theta\]

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 predicicators to values of the appropriate type. This is referred to as the Conformity Rule:

\[(P4) \quad x \in \mathcal{F} \land y \in \text{Pop}(x) \Rightarrow y : x \rightarrow \Omega \wedge \forall p \in \mathcal{P}\{y(p) \in \text{Pop} \left( \text{Base}(p) \right) \}\]

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

\[(P6) \quad x \in \mathcal{E} \land y \in \text{Pop}(x) \Rightarrow y \in \mathcal{P} \left( \text{Pop} \left( \text{Elt}(x) \right) \right) \setminus \{\emptyset\}\]

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

\[(P6) \quad x \in \mathcal{E} \Rightarrow \text{Pop}(\varepsilon_x) = \{\{\varepsilon^i_x : u, \varepsilon^i_x : v\} \mid u \in \text{Pop}(x) \wedge v \in u\}\]

The Power Base Rule is a derivation rule for the population of fact type \( \varepsilon_x \). Note that it is not necessary in the Power Base Rule to state that \( v \in \text{Pop}(\text{Elt}(x)) \) 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:

\[(P7) \quad x \in \mathcal{E} \land y \in \text{Pop}(x) \Rightarrow y \in \text{Pop} \left( \text{Elt}(x) \right)^*\]

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

\[(P8) \quad x \in \mathcal{E} \Rightarrow \text{Pop}(\varepsilon_x) = \{\{\varepsilon^i_x : u, \varepsilon^i_x : v\} \mid u \in \text{Pop}(x) \wedge \exists_{\varepsilon_x} \{u[i] = v\}\}

\[(P9) \quad x \in \mathcal{E} \Rightarrow \text{Pop}(@_x) = \{\{\varepsilon^i_x : u, @_x : v\} \mid u \in \text{Pop}(\varepsilon_x) \wedge u(\varepsilon^i_x)[v] = u(\varepsilon^i_x)\} \]

These rules can be used as derivation rules for \( \varepsilon_x \) and \( @_x \).
The population of a composition type consists of populations of the underlying information structure. This is called the Decomposition Rule:

\[(P10) \ x \in \mathcal{C} \land y \in \text{Pop}(x) \Rightarrow \text{IsPop}(\mathcal{F}_x, y).\]

The relation between a composition type and its constituting object types is recorded in the fact type \(\text{IsPop}_{\mathcal{F}}, \mathcal{E}, \mathcal{A}\). Its population is described in the Decomposition Rule, which is a derivation rule:

\[(P11) \ x < y \Rightarrow \text{Pop}(\text{IsPop}_{\mathcal{F}, \mathcal{E}, \mathcal{A}}(x, \mathcal{E}, \mathcal{A}) \subseteq \text{Pop}(y)).\]

**Lemma 2.2.** \(x < y \Rightarrow \forall u \in \text{Pop}(x) \land v \in u(y).\) Applying the Decomposition Rule one can derive that \(\{\varepsilon_{x,y} : u, \varepsilon_{x,y} : v\} \in \text{Pop}(\text{IsPop}_{\mathcal{F}, \mathcal{E}, \mathcal{A}}(x, \mathcal{E}, \mathcal{A})).\) From the Conformity Rule and the fact that \(\text{Base}(\varepsilon_{x,y}) = y\) it then follows that \(v \in \text{Pop}(y).\)

Respecting the specialization hierarchy is reflected by the Specialization Rule:

\[(P12) \ x \text{ Spec } y \Rightarrow \text{Pop}(x) \subseteq \text{Pop}(y)\]

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 Generalization hierarchy is reflected by the Generalization Rule:

\[(P13) \ \text{gen}(x) \Rightarrow \text{Pop}(x) = \bigcup_{y \in \text{gen}(x)} \text{Pop}(y)\]

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

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

- \(\text{Pop}(A) = \{a_1, a_2\}\)
- \(\text{Pop}(B) = \{b_1\}\)
- \(\text{Pop}(C) = \{c_1\}\)
- \(\text{Pop}(D) = \{d_1\}\)
- \(\text{Pop}(E) = \{\{a_1, a_1, a_2\}\}\)
- \(\text{Pop}(F) = \{17\}\)

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 \(a_1, a_2, b_1, c_1, d_1\) come from the abstract domain \(\Theta\) 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 \(\text{Pop}(C)\). In Fig. 9 this population is graphically represented. The population of the implicit fact type \(\varepsilon_F\) can be derived to be:

\(\text{Pop}(\varepsilon_F) = \{\varepsilon_F^a : \{a_1\}, \varepsilon_F^a : \{a_1, a_2\}, \varepsilon_F^a : \{a_1, a_2\}, \varepsilon_F^a : \{a_1, a_2\}, \varepsilon_F^a : \{a_1, a_2\}, \varepsilon_F^a : \{a_1, a_2\}\}.$$

### 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 \(\Sigma = \langle \mathcal{F}, \mathcal{R} \rangle\) be a PSM schema over information structure \(\mathcal{F}\) 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 \(\text{total}(\tau)\) over a set of predicicators \(\tau\) states that object instances in the population of their bases occur at least once in the population.
of these predicates. A uniqueness constraint unique(τ) over a set of predicates τ expresses the uniqueness of combinations of values in these predicates. Examples of a total role constraint and a uniqueness constraint have been discussed in Example 6. The formal semantics of the total(τ) and unique(τ) is given in Refs [4] and [26].

The requirements for structural identification have been presented in Ref. [3], 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:

   \[ \forall x \in \mathcal{X} \exists p \in \mathcal{P} \exists_{\text{total}(τ) \in \mathcal{A}} \left[ \text{Base}(p) = x \land p \in τ \right]. \]

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

2. All object types can be identified

   \[ \forall x \in \mathcal{X} [\text{Identifiable}(x)]. \]

   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 Identifiable(x).

**Fact types**—A fact type x (or, generally, a set of predicates) is identifiable if all components of x are identifiable:

\[ \forall p \in x [\text{Identifiable(\text{Base}(p))}]. \]

**Power types and sequence types**—A power type or sequence type x is identifiable if its element type is identifiable:

\[ \text{Identifiable(\text{Elt}(x))} \]

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

\[ \forall x \in y [\text{Identifiable}(y)]. \]

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

If x is not *pater familias* (\[\overline{x}(x) \neq x\]) 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 generalization. In this case the object type inherits its identification from some of its specifiers. More precisely, if $x$ is a generalized object type ($\text{gen}(x)$), then $x$ is identifiable if:

$$\exists y \in \text{Gen} \; [x \; \text{Gen} \; y \land \text{Identifiable}(y)].$$

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 other type, in any population. The set of possible first names for denominations is defined by:

$$N(x) = \{p \mid \text{Base}(p) = x \land \text{total}(\{p\}) \land \text{unique}(\{p\})\}.$$ 

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 $\tau$ of predicicators such that:

- (uniqueness of denotation) $\text{unique}(\tau)$
- (first name-middle name relatedness) $\forall \tau \in \text{Fact}(p) \exists q \in N(x)$
- (recursion) $\forall \tau \cdot \text{Identifiable}(\text{Base}(p))$

Example 2.8. In Fig. 10 an example of identification in the case of an entity type that is a pater familias is shown. The bases of predicicators $p_1, p_5$ 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 $\{p_1, p_3\}$, 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:

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

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 predicicators, an order of the middle names is defined by Ident: $\mathcal{E} \rightarrow \mathcal{P}^*$. The (partial) function Copred: $\mathcal{P} \rightarrow \mathcal{P}$ is introduced to resolve any ambiguity in the relation between middle names and first names. However, predicicators from the same fact type should be assigned the same copredicator:

$$\text{Copred}(p) = \text{Copred}(q) \Rightarrow \text{Copred}(p) = \text{Copred}(q).$$

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

- Ident(Address) = $\langle p_7, p_9 \rangle$
- Ident(Street) = $\langle p_3, p_5 \rangle$
- Ident(Community) = $\langle p_1, p_3 \rangle$
- Copred($p_9$) = $p_8$
- Copred($p_5$) = $p_13$
- Copred($p_3$) = $p_10$

Fig. 10. Example of complex identification.
Formal definition of a conceptual language

\[ \text{Ident} (\text{Community}) = \langle p_1 \rangle \]
\[ \text{Copred}(p_3) = p_6 \]
\[ \text{Copred}(p_4) = p_2 \]

This allows a short denotation for addresses in the form:
\[ \langle H\text{-nr}, S\text{-name}, C\text{-name} \rangle. \]

Besides for the identification of entity types, \( \text{Ident} \) 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 \( \text{Ident}(f) = \langle p_1, \ldots, p_n \rangle \) 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 [14], or bags [28], 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 \( \lambda \)-calculus notation provided by Ref. [29], will be employed. For instance \( \lambda X.X^2 \) is the polynomial function assigning \( x^2 \) to each \( x \)-value.

As in set theory, \( \emptyset \) denotes the empty multiset, with definition: \( \{x\} \equiv \{e \mid C(e, n)\} \) is a more conventional denotation for a multiset corresponding to bag comprehension, see e.g. [29]. Bag comprehension can be used for intentional denotations of multisets. Extensional denotations are defined by: \( \{a\} \equiv \{a^* \mid q = 1\} \) and \( \{a_1, \ldots, a_n\} \equiv \{a_1\cup\cdots\cup\{a_n\} \}. \) We will write \( e \in^* M \) rather than \( M(e) = n \), and \( e \in M \) for \( M(e) > 0 \).

Besides forming multisets by means of an intentional or extensional specification, they can be formed by the following binary operators:

\[ N \cup M \equiv \lambda x \cdot N(x) + M(x) \]
\[ N \cap M \equiv \lambda x \cdot \min(N(x), M(x)) \]
\[ N - M \equiv \lambda x \cdot \max(N(x) - M(x), 0) \]

The comparison operator \( N \subseteq M \) for multisets is defined as: \( \forall x, (N(x) \leq M(x)) \). From this operator, the \( \subseteq \) comparison is derived in the usual way: \( N \subseteq M \land N \neq M \). This allows for the definition of the powerset of a multiset: \( \mathcal{P}(X) = \{ Y \uparrow \mid Y \subseteq X \} \). Coercions from multiset to set and vice versa are defined by the following functions:
The number of elements in a multiset is counted by $|N| = \sum_{x \in X} N(x)$. 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 $X \times X$:

$$\text{Sqr}(N) = \{<x, x> | x \in N\}$$

and conversely:

$$\pi_1(N) = \lambda x. \sum_{y \in X} N(x, y),$$

$$\pi_2(N) = \lambda y. \sum_{x \in X} N(x, y).$$

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

$$N \circ M \equiv \lambda <x, y>. \bigcup_{a \in X} N(x, a) \times M(a, y),$$

$$N \diamond M \equiv \lambda <x, y>. \bigcup_{a \in X} N(x, a) \times M(y, b),$$

$$N^{-} \equiv \lambda <x, y>. N(y, x),$$

where $N^{-}$ 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 $^{-}$ operations, we define the neutral element: $1_{X \times X} \equiv \lambda <x, x>. 1$. We also define the following operation, being the multiset pendant of a union of a set of sets:

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

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:

$$\max(N) \equiv \max(\text{Set}(N))$$

$$\min(N) \equiv \min(\text{Set}(N))$$

$$\sum(N) \equiv \sum_{x \in X} x \times N(x)$$

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

$$\text{Lin}(<x_1, \ldots, x_n>) = \bigcup_{i < j < n} \{x[i]\},$$

for example $\text{Lin}(<a, b, c, d>) = \{a, a, b, c, d\}$.

### 3.2. Path expressions

The syntax of path expressions is presented as an abstract syntax. In Ref. [31] 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, emphasizing 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. [33]). 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
Formal definition of a conceptual language

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 unifying 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) objects types. It will be convenient to treat these binary relations tuple oriented [33], 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 $\Omega$ in the following way:

$$\Omega_\text{xe} = \{X \mid X \text{ is a multiset over } \Omega \times \Omega\}$$

Path expressions are built around the following syntactical categories: constant, multiset, object type ($\mathcal{O}$), predicator ($\mathcal{P}$) and path expression ($\mathcal{P} \mathcal{E} (\mathcal{P})$). 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:

$$\mu : \mathcal{P} \mathcal{E} \times \text{POP} \rightarrow \Omega_\text{xe}$$

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

<table>
<thead>
<tr>
<th>name</th>
<th>expr</th>
<th>$\mu<a href="%5Ctext%7BPop%7D">\text{expr}</a>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty path</td>
<td>$\emptyset_{\text{xe}}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>neutral path</td>
<td>$1_{\text{xe}}$</td>
<td>$1_{\Omega \times \Omega}$</td>
</tr>
<tr>
<td>constant</td>
<td>$c$</td>
<td>$\text{Sqr}({c})$</td>
</tr>
<tr>
<td>multiset</td>
<td>$X$</td>
<td>$\text{Sqr}(X)$</td>
</tr>
<tr>
<td>object type</td>
<td>$x$</td>
<td>$\text{Sqr} \cdot \text{Multi} \cdot \text{Pop}(x)$</td>
</tr>
<tr>
<td>predicator</td>
<td>$p$</td>
<td>${\langle v(p), v \rangle \uparrow \mid v \in \text{Pop} \cdot \text{Fact}(p)}$</td>
</tr>
</tbody>
</table>

Example 3.1. Suppose $\text{Pop}(g) = \{\{r : h_1, s : e_1\}, \{r : h_2, s : e_1\}, \{r : h_3, s : e_2, e_3\}\}$ in Fig. 8, then:

$$\mu[r](\text{Pop}) = \begin{cases} b_1 & \{r : b_1, s : e_1\} \\ b_2 & \{r : b_2, s : e_1\} \\ b_3 & \{r : b_3, s : e_2, e_3\} \end{cases}$$

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: $P^-$, to isolate the front elements of a path $P$ by: $f P$, to remove multiple occurrences using: $d P$, to count the number of elements in a path expression by: $\text{Cntt}(P)$, to add the elements in a path expression by means of: $\text{Sum}$, and to determine the minimum or maximum element in a path expression by: $\text{Min}$ and $\text{Max}$. The powerset $\mathcal{P}(P)$ of a path expression $P$ yields a path expression with all sets of instances occurring in the first component of $P$. The operators are summarized in the following table:
Example 3.2. In the situation of the previous example:

\[ \mu[s^-](\text{Pop}) = \begin{cases} 
\{ r : b_1, s : c_1 \} & c_1 \\
\{ r : b_2, s : \{ e_1 \} \} & \{ e_1 \} \\
\{ r : b_3, s : \{ e_2, e_3 \} \} & \{ e_2, e_3 \} 
\end{cases} \]

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:

<table>
<thead>
<tr>
<th>name</th>
<th>expr</th>
<th>(\mu<a href="%5Ctext%7BPop%7D">\text{expr}</a>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenate</td>
<td>(P \circ Q)</td>
<td>(\mu<a href="%5Ctext%7BPop%7D">P</a> \circ \mu<a href="%5Ctext%7BPop%7D">Q</a>)</td>
</tr>
<tr>
<td>extend</td>
<td>(P \diamond Q)</td>
<td>(\mu<a href="%5Ctext%7BPop%7D">P</a> \diamond \mu<a href="%5Ctext%7BPop%7D">Q</a>)</td>
</tr>
<tr>
<td>intersection</td>
<td>(P \cap Q)</td>
<td>(\mu<a href="%5Ctext%7BPop%7D">P</a> \cap \mu<a href="%5Ctext%7BPop%7D">Q</a>)</td>
</tr>
<tr>
<td>union</td>
<td>(P \cup Q)</td>
<td>(\mu<a href="%5Ctext%7BPop%7D">P</a> \cup \mu<a href="%5Ctext%7BPop%7D">Q</a>)</td>
</tr>
<tr>
<td>minus</td>
<td>(P - Q)</td>
<td>(\mu<a href="%5Ctext%7BPop%7D">P</a> - \mu<a href="%5Ctext%7BPop%7D">Q</a>)</td>
</tr>
</tbody>
</table>

Example 3.3. In the situation of Example 3.1:

\[ \mu[r \circ s^-](\text{Pop}) = \begin{cases} 
 b_1 & c_1 \\
 b_2 & \{ e_1 \} \\
 b_3 & \{ e_2, e_3 \} 
\end{cases} \]

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

\[ \mu[r \circ e^- \circ e^f \circ e'^-](\text{Pop}) = \begin{cases} 
 b_2 & e_1 \\
 b_3 & e_2 \\
 b_5 & e_3 
\end{cases} \]

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:

<table>
<thead>
<tr>
<th>name</th>
<th>expr</th>
<th>(\mu<a href="%5Ctext%7BPop%7D">\text{expr}</a>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>grouping</td>
<td>(\varphi(P, G))</td>
<td>see below</td>
</tr>
<tr>
<td>ungrouping</td>
<td>(\gamma(P))</td>
<td>(\text{Sqr} \cdot { \cup } \cdot \pi_1 \cdot \mu<a href="%5Ctext%7BPop%7D">P</a>)</td>
</tr>
<tr>
<td>ordering</td>
<td>(\psi(P, G))</td>
<td>see below</td>
</tr>
</tbody>
</table>
Grouping path expression $P$, according to grouping criterion $G$, is performed by the function $\varphi(P, G)$. 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 $g \in \pi_2 \cdot \mu[G](\text{Pop})$, then with $g$ is associated the following class of elements:

$$K_g = \{x \in \pi_1 \cdot \mu[P](\text{Pop}) | \langle x, g \rangle \in \mu[G](\text{Pop})\}.$$

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:

$$\mu[\varphi(P, G)](\text{Pop}) = \text{Multi}(\langle K_g, g \rangle | g \in \pi_2 \cdot \mu[G](\text{Pop}) \land K_g \neq \emptyset).$$

Sorting the result of path expression $P$ into a single sequence, according to a sorting criterion $S$, can be achieved by applying $\psi$ on $P$ and $S$, respectively. The sorting criterion may be weak (for example $S = \text{ass}$), 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 $\text{Pop}$ if:

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

$$0 \leq i < j < |s| \Rightarrow \exists s[i], y_1 \in \mu[P](\text{Pop}) \land \langle s[j], y_2 \rangle \in \mu[P](\text{Pop})$$

$$\land \langle y_2, y_1 \rangle \notin \mu[S](\text{Pop}).$$

The result of sorting now is defined as:

$$\mu[\psi(P, S)](\text{Pop}) = \text{Sqr}(\{s | s \text{ is compatible with } S \text{ over } P \text{ in } \text{Pop}\}).$$

The following construction mechanism for path expressions correspond to the transitive closure of a binary relation:

<table>
<thead>
<tr>
<th>name</th>
<th>expr</th>
<th>$\mu<a href="%5Ctext%7BPop%7D">\text{expr}</a>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>closure</td>
<td>$P^+$</td>
<td>$\text{ds}(\bigcup_{n \in \mathbb{N}} \mu<a href="%5Ctext%7BPop%7D">\text{closure}(n, P)</a>)$</td>
</tr>
</tbody>
</table>

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

$$\text{closure}(0, P) = P,$$

$$\text{closure}(n + 1, P) = \text{closure}(n, P) \circ P.$$  

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.

<table>
<thead>
<tr>
<th>name</th>
<th>expr</th>
<th>$\mu<a href="%5Ctext%7BPop%7D">\text{expr}</a>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>confluence</td>
<td>$[P_1, \ldots, P_n \mid Q]$</td>
<td>see below</td>
</tr>
</tbody>
</table>

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:

$$\mu[[P_1, \ldots, P_n \mid Q]](\text{Pop}) = \bigcup_{x \in \pi_1 \cdot \mu[P](\text{Pop})} \{\langle x_1, \ldots, x_n, x \rangle^\uparrow_{x_1, \ldots, x_n} \in \forall \pi_{\in \mathbb{N}} \cdot \mu[P](\text{Pop})\}$$

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

Formal definition of a conceptual language 505
In order to define the active complement $\neg$ (see Ref. [33]) of a path expression, the set of active elements are introduced:

$$\text{ActVaIs} = \bigcup_{x \in 0} x.$$  

The active complement of a path expression $P$ then is defined as: $\neg P \equiv \text{ActVaIs} - \neg P$. Path expressions are coerced to multi sets by the Rn function: $\text{Rn} \equiv \pi_1 \cdot \mu$.

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.

<table>
<thead>
<tr>
<th>name</th>
<th>$\delta(P, Q)$</th>
<th>(\mu<a href="%5Ctext%7BPop%7D">[\text{expr}]</a>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>element substitution</td>
<td>(\delta(P, Q))</td>
<td>$\text{See below}$</td>
</tr>
<tr>
<td>set constructor</td>
<td>${P_1, \ldots, P_n}$</td>
<td>$\text{Sqr} \cdot \text{Multi}({{x_1, \ldots, x_n}</td>
</tr>
<tr>
<td>sequence constructor</td>
<td>$\langle P_1, \ldots, P_n \rangle$</td>
<td>$\text{Sqr} \cdot \text{Multi}(\langle x_1, \ldots, x_n \rangle</td>
</tr>
</tbody>
</table>

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 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:

$$\mu[[\delta(P, Q)]](\text{Pop}) = \bigcup_{(x, y) \in \delta(P, Q)} \{\langle z, y \rangle | z = \text{subst}(x, \mu[[Q](\text{Pop}))\}.$$

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 U.S.A. This example was a unified example in the special issue of Computing Surveys [34]; the example was first enunciated in Ref. [35]. An excerpt of this schema is presented in Fig. 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 verbalize the mathematical concepts of PSM via some set $\mathcal{N}$ of names. Object types are referenced by a unique name: $\text{ONm} : \mathcal{O} \rightarrow \mathcal{N}$, which is
Fig. 11. Example information structure.
specified in the schema upon their introduction. The (partial) function \( \text{Obj} : \mathcal{N} \rightarrow \mathcal{O} \) is the left-inverse of \( \text{ONm} \), and relates object type names to their corresponding object type:

\[
\forall_{x \in \mathcal{N}} \text{Obj}(\text{ONm}(x)) = x
\]

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

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

\[
\forall_{\mathcal{N}, \mathcal{P}} [\text{ONm}(\text{Fact}(p)) \text{PNm}(p) = p]
\]

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 [36], special names are introduced for predications, 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: \( \text{RNm} : \mathcal{P} \rightarrow \mathcal{N} \). In Fig. 11 role names are added to all predications 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 Ref. [36]). A combination \( nx \text{np} ny \) is valid if there exists a predicator \( p \) such that:

\[
\text{ONm}\left(\text{Base}(p)\right) \sim nx,
\]

\[
\text{RNm}(p) = \text{np},
\]

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

\[
\text{ONm}(\text{Base}(q)) \sim ny.
\]

The combination of \( nx \text{np} ny \) has a unique interpretation in the information structure, if predicator \( p \) is unique. The combination of \( nx \text{np} ny \) 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 predications of binary fact types have unique role names. In the nonbinary 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 \( nx \text{np} ny \). For this purpose, the Path function will be introduced.

As a simple example of this new interpretation, consider the (ternary) election relation in Fig. 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 \( p_1 \odot p_2 \). 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 \( p_1 \odot p_2 \).

**The administration of names**—This leads to a generalization of role names to a partial naming function \( \text{Path} : \mathcal{O} \times \mathcal{O} \times \mathcal{N} \rightarrow \mathcal{P} \mathcal{D} \) that assigns, in a given context, a path expression to a name.
The notation $\text{Path}(x, y, n)$ is used to indicate that $\text{Path}(x, y, n)$ 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 $\text{Path}$ 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 $\text{Path}$ contains the following:

1. Names of (explicit) object types are defined names. The name $\text{ONm}(x)$ of object type $x$ stands for path expression $x$:

$$\text{Path}(x, x, \text{ONm}(x)) = x$$

For implicit object types (such as fact type $\varepsilon$) 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 $\text{PNm}(p)$ describes a path from the base of $p$ to its corresponding fact type:

$$\text{Path}(\text{Base}(p), \text{Fact}(p), \text{PNm}(p)) = p.$$  

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:

$$\text{Path}(\text{Base}(p), \text{Base}(q), \text{RNm}(p)) = p \circ q^-,$$

provided $f$ is not a homogeneous fact type with ambiguous role names (i.e. $f$ consists of predicators $p$, $q$ such that $\text{Base}(p) = \text{Base}(q)$, and also $\text{RNm}(p) = \text{RNm}(q)$). In that case the name receives its interpretation from both roles:

$$\text{Path}(\text{Base}(p), \text{Base}(q), \text{RNm}(p)) = p \circ q^- \cup q \circ p^-.$$ 

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

$$\text{Path}(\text{SortOf}(c), \text{SortOf}(c), \text{CNm}(c)) = c.$$ 

The functions $\text{CNm}$ and $\text{SortOf}$ 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 $\text{Dom} : \mathcal{L} \rightarrow D$. 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 $\text{Dom}$ specifying subtype relations as follows:

$$x \text{ Spec } d \iff \text{Dom}(x) = d.$$ 

These subtype relations do not require subtype defining rules.

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

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.
Fig. 12. Associating concrete domains to label types.

Appropriate names for the object types and predicates 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 \( \text{SortOf} \) be the function that returns the name of this domain for any concrete value:

\[
\text{SortOf}(c) = d \iff c \in d.
\]

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 predicates in any fact type (see function \( \text{Ident} \) in Section 2.3), with the restriction that for functions the predicate corresponding to the result of the function, is first in this ordering. For example, the signature convention for the operator \( + \) could be: \( \text{result}_+, \text{first-argument}_+, \text{second-argument}_+ \). The addition \( 5 + 3 \) then is represented in the population of fact type \( + \) by the tuple \( (8, 5, 3) = (\text{result}_+, 5, \text{first-argument}_+, 3) \).

For relations, special naming conventions can be introduced, for example:

\[
\text{Path}(\text{Nanto}, \text{Nanto}, <) = \text{first-argument}_+ \circ \text{second-argument}_-
\]

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: \( \text{Var} \) for simple variables and \( \mathcal{N} \) for names. The naming conventions for instances of these syntactical categories are as follows: for Information Descriptor: \( \text{P}, \text{P}', \text{P}_1, \text{P}_2, \text{O}, \text{Q} \), for \( \text{Var} \): \( v \), for \( \mathcal{N} : n \).

The semantics of the syntactic category Information Descriptor is specified by the valuation function \( D: \text{Information Descriptor} \times \text{ENV} \rightarrow \mathcal{P} \mathcal{E} \) 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. \( \text{ENV}: \text{Var} \rightarrow \mathcal{P} \mathcal{E} \) denotes the environment containing the current
Formal definition of a conceptual language 511

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 \( N \), 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 summarized by:

\[
D[n](e) = \bigcup_{\text{Path}(x, y, n)} \text{Path}(x, y, n),
\]

\[
D[v](e) = \begin{cases} 
  e(v) & \text{if } e(v) \text{ is defined,} \\
  \emptyset & \text{otherwise.}
\end{cases}
\]

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:

\[
D[\text{born-in}] = \text{Birthyear.born-in}
\]

\[
\cup \text{Birthyear.born-in } \circ \text{being-birthyear-of}
\]

\[
\cup \text{Birthyear.born-in}
\]

\[
\cup \text{Birthstate.born-in } \circ \text{being-birthstate-of}
\]

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

\[
D[P_1, P_2](e) = D[P_1](e) \circ D[P_2](e).
\]

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:

\[
D[\text{born-in Hobby}](e) = D[\text{born-in}](e) \circ \text{Hobby}
\]

\[
= \emptyset.
\]

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:

\[
D[n_1, n_2](e) = \bigcup_{z_1, z_2} \text{Path}(x, z_1, n_1) \circ \text{Path}(z_2, y, n_2).
\]

**Proof**—Suppose \( z_1 \neq z_2 \), then in each population \( \text{Pop} \) of information structure \( \Sigma \) (i.e. \( \text{Is-Pop}(Z, \text{Pop}) \)) \( z_1 \) and \( z_2 \) have no values in common (axiom P1): \( \text{Pop}(z_1) \cap \text{Pop}(z_2) = \emptyset \). As a result, there is no contribution from \( \text{Path}(x, z_1, n_1) \circ \text{Path}(z_2, y, n_2) \) to the result of \( n_1, n_2 \) for any \( x \) and \( y \).

As a next example, the information descriptor born-in State is composed by the concatenation of two atomic information descriptors.

\[
D[\text{born-in State}](e) = D[\text{born-in}](e) \circ \text{State}
\]

\[
= \text{Birthstate.born-in } \circ \text{being-birthstate-of} \circ \text{State}.
\]
As this path expression is homogeneous (see Section 3), it follows that the information descriptor born-in State has the same meaning as President born-in State. The next example concatenates two atomic information descriptors, that both correspond to an object type:

\[
\mathbb{D}[\text{President Person}](e) = \text{President} \circ \text{Person} = \text{President}.
\]

Sometimes, parts of 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:

\[
X_1 \text{ Spec } X_2 \lor X_2 \text{ Gen } X_1 \Rightarrow n_1 n_2 \equiv n_1
\]

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

In the case of objectification, the predicator name can be fruitfully employed to form fluent sentences. For example, suppose \( \text{PNm}(p_4) = \text{in} \) instead of the name presented in Fig. 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:

\[
\mathbb{D}[\text{President in Marriage resulting-in Nr-of-children}](e) = \text{President} \circ \text{in} \circ \text{Marriage} \circ \text{resulting-in} \circ \text{resulting-from}^{-} \circ \text{Nr-of-children}.
\]

**Keywords as information descriptor**—Until now only defined names are introduced for constants, object types and predicates. This naming serves as a verbalization 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 Fig. 13.

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

\[
\text{Path(Base(abstr(b)), Base(concr(b)), WITH)} = \text{abstr(b)} \circ \text{concr(b)}^{-}
\]

\[
\text{Path(Base(concr(b)), Base(abstr(b)), IS-NAME-OF)} = \text{concr(b)} \circ \text{abstr(b)}^{-}
\]

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 Fig. 11, since in such cases bridge types are not visualized.

Example: President \text{WITH} person-name ‘Roosevelt F.D.’ denotes the president with the name ‘Roosevelt F.D.’. Part name \text{IS NAME OF} Party having as member President \text{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 \text{OF} and \text{INVOLVED-IN} are intended to facilitate the manipulation of objectified fact types. They are also useful as shorthands for predicator names. The keyword \text{OF} represents all relations between fact type instances and their constituent object type instances, the keyword \text{INVOLVED-IN} is its inverse:

\[
\text{Path}(x, f, \text{INVOLVED-IN}) = \bigcup_{q \in f, \text{Base}(q) = x} q
\]

\[
\text{Path}(f, x, \text{OF}) = \bigcup_{q \in f, \text{Base}(q) = x} q^{-}
\]

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

\[
\begin{aligned}
&\text{for all } x \in \mathcal{A}: \left\{ \begin{array}{l}
\text{Path(Elt}(x), x, \text{IN}) = e_x^e \circ e_x^m, \\
\text{Path}(x, \text{Elt}(x), \text{CONTAINING}) = e_x^e \circ e_x^m{}
\end{array} \right.
\end{aligned}
\]

Example: Ships can be related to the convoy in which they sail (see Fig. 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 \(e^s\).
Consequently, it relates sequences to the sequence membership relations (instances from \(e_x\)) in which they occur. The keyword OCCURRING-IN, does the reverse, it relates sequence membership relations to the involved sequences:

\[
\text{for all } x \in F: \left\{ \begin{array}{l}
\text{Path}(x, e_x, \text{SEQUENCES}) = e_x', \\
\text{Path}(e_x', x, \text{OCCURRING-IN}) = e_x.
\end{array} \right.
\]

The keyword ELEMENTS is a generic name for predicators \(e_x'.\) 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:

\[
\text{for all } x \in F: \left\{ \begin{array}{l}
\text{Path}(\text{Elt}(x), e_x, \text{ELEMENTS}) = e_x', \\
\text{Path}(e_x', \text{Elt}(x), \text{HAVING}) = e_x.
\end{array} \right.
\]

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

\[
\text{for all } x \in F: \left\{ \begin{array}{l}
\text{Path}(I, e_x, \text{INDICES}) = e_x', \\
\text{Path}(e_x', I, \text{AT-POSITION}) = e_x.
\end{array} \right.
\]

Example: Consider the schema of Fig. 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:

\[
\text{for all } x \in F, x < y: \left\{ \begin{array}{l}
\text{Path}(x, y, \text{COMPRISING}) = e_{x,y}', \\
\text{Path}(y, x, \text{PART-OF}) = e_{x,y}.
\end{array} \right.
\]

Example: Consider Fig. 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:

\[
\begin{align*}
D[P \text{ AND-ALSO } P'](e) &= \neg D[P](e) \cap \neg D[P'](e) \\
D[P \text{ INTERSECTION } P'](e) &= D[P](e) \cap D[P'](e) \\
D[P \text{ OR-ELSE } P'](e) &= \neg D[P](e) \cup \neg D[P'](e) \\
D[P \text{ UNION } P'](e) &= D[P](e) \cup D[P'](e) \\
D[P \text{ BUT-NOT } P'](e) &= \neg D[P](e) - \neg D[P'](e) \\
D[P \text{ MINUS } P'](e) &= D[P](e) - D[P'](e) \\
D[\neg P](e) &= \neg D[P](e)
\end{align*}
\]

To find the presidents that were born in California and served 4 yr 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-NAMED-OF Year can be simplified by the introduction of special names:
Formal definition of a conceptual language

\[ \text{Path(Nr-of-years, Nr, Nr-of-years)} = p \circ q \]  
\[ \text{Path(Nr, Nr-of-years, Nr-of-years)} = q \circ p \]  

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

**Predicate inversion**—Predicate 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:

\[ D[n](e) = \bigcup_{Pn(e) = n} p^{-}. \]

**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 \( 45 + \text{Year being-birthyear-of} \). 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:

\[ P_1 n P_2 = n_0(n \text{ AND-ALSO } n_1 : P_1)n_2 : P_2, \]

where \( n_0, n_1, n_2 \) is the signature convention of the operator with name \( n \).

**Transitive closure**—The information descriptor \( \text{ANY-REPETITION-OF} P \) describes the transitive closure of information descriptor \( P \), and is defined as follows:

\[ D[\text{ANY-REPETITION-OF} P](e) = (D[P](e))^+. \]

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

As another example of the use of the transitive closure consider Fig. 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: \( \text{ANY-REPETITION-OF (Activity being-decomposed-into Activity-graph COMPRISING Activity).} \) This information descriptor relates activities to the activities in their direct or indirect decompositions.

**Correlation**—In order to find the presidents who were inaugurated at an age younger than 45 yr, i.e. inaugurated at least once within 45 yr of their birth year, a convenient formulation is: \( \text{President being-president-of Administration inaugurated-in Year \leq 45 + \text{Year being-birthyear-of THAT President.} \) 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:

\[ D[P \text{ THAT } O](e) = D[P \circ O](e) \cap D[O](e). \]

Usually, the second information descriptor involved (i.e. \( O \)) is the name of an 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:

\[ D[\text{NUMBER-OF } P](e) = \text{Cnt}(D[P](e)). \]

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:

\[ D[\text{SUM } P](e) = \text{Sum}(D[P](e)). \]

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.

\[ D[\text{MIN } P](e) = \text{Min}(D[P](e)), \]

\[ D[\text{MAX } P](e) = \text{Max}(D[P](e)). \]

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:

\[ D[\text{DISTINCT } P](e) = \text{ds}(D[P](e)). \]

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:

\[ D[\text{GROUP } P \text{ BY } Q](e) = \phi(D[P](e), D[Q](e)) \]

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:

\[ D[\text{UNITE } P](e) = \gamma(D[P](e)) \]

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:

\[ D[\text{SORT } P \text{ BY } Q](e) = \psi(D[P](e), D[Q](e)). \]

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:

\[ D[P \text{ PAIRED-WITH } P'](e) = D[P](e) \otimes D[P'](e), \]

\[ D[\text{ALL-SUBSETS-OF } P](e) = \wp(D[P](e)). \]
Formal definition of a conceptual language

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 Fig. 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:

LET \( v \) 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:

\[
A : \text{Assignment} \times \text{ENV} \rightarrow \text{ENV},
\]

which is defined as:

\[
A[\text{LET } v \text{ BE } P](e) = e \oplus \{ v \leftarrow D[P](e) \}.
\]

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

\[
D[A; P](e) = D[P](A[e]).
\]

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:

\[
\begin{align*}
&\quad \text{c} & &\text{constants} \\
&\quad \nu & &\text{variables} \\
&\quad d_1, \ldots, d_k & &\text{denotation of entities} \\
&\quad [q_1 = d_1, \ldots, q_k = d_k] & &\text{denotation of facts} \\
&\quad \{d_1, \ldots, d_k\} & &\text{alternative denotation of facts} \\
&\quad <d_1, \ldots, d_k> & &\text{denotation of power type instances} \\
&\quad \{d_1, \ldots, d_k\} & &\text{denotation of sequence type instances}
\end{align*}
\]

where \( c \) is a Constant name, \( \nu \in \text{Var} \), \( d_i \) is a Constant Denotation and \( q_i \in \text{ran}(\text{PNm}) \).

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

\[
D[L : c](e) = D[L \ c](e).
\]

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

Consider Fig. 10. To denote a concrete address, while only using the constructs that have been introduced so far, one would have two 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 RNm(p_8) = RNm(p_4) = in.

Obviously, one would prefer to write:

Address: 'New York', 'Fifth Avenue', 17
This is an example of an entity denotation. The formal definition of entity denotations uses the functions \( \text{ident} \) and \( \text{Copred} \) introduced in Section 2.3. If \( E \) is the name of an entity type, then:

\[
D[E : d_1, \ldots, d_k](e) = \text{Obj}(E) \bigcap \bigcap_{i=1}^k \mathcal{F}(\text{Copred}(p_i) \cap \text{Obj}(B_i : d_i)(e)),
\]

where \( p_i = \text{ident}(\text{Obj}(E))[i] \) and \( B_i = \text{ONm}(\text{Base}(p_i)) \).

The function \( \text{Ident} \) 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 \( \text{Ident} \) 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:

\[
D[F : [d_1, \ldots, d_k]](e) = \text{Obj}(F) \bigcap \bigcap_{i=1}^k \mathcal{F}(p_i \cap \text{Obj}(N_i : d_i)(e))
\]

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

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

\[
\text{Admin-pers} : [49, 'Eisenhower DD']
\]

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

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:

\[
D[F : [q_1 = d_1, \ldots, q_k = d_k]](e) = \text{Obj}(F) \bigcap \bigcap_{i=1}^k \mathcal{F}(F.q_i \cap \text{Obj}(N_i : d_i)(e))
\]

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

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

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

Evidently, the advantage of this new type of denotation is that the assignments in the function \( \text{Ident} \) 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:

\[
D[G : \{d_1, \ldots, d_k\}](e) = \text{Obj}(G) \bigcap \{D[X : d_i](e), \ldots, D[X : d_k](e)\}
\]

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

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

\[
\text{Convoy} : \{'S101', 'S102'\}
\]

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

\[
D[S : \langle d_1, \ldots, d_k \rangle](e) = \text{Obj}(S) \bigcap \langle D[X : d_i](e), \ldots, D[X : d_k](e)\rangle
\]

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

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

\[
\text{Freight car sequence} : \langle 'FC96', '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 $P : \text{Predicate} \times \text{POP} \times \text{ENV} \rightarrow \text{Bool}$. The basis for LISA-D predicates is the test whether an information descriptor has an empty result. From this basic predicate new predicates can be formed in the usual way, using logical connectives and quantification:

$$P[P](\text{Pop}, e) = \mu \left[ D[P](e) \right](\text{Pop}) \neq \emptyset,$$

$$P[C_1 \text{ AND } C_2](\text{Pop}, e) = P[C_1](\text{Pop}, e) \land P[C_2](\text{Pop}, e)$$

$$P[C_1 \text{ OR } C_2](\text{Pop}, e) = P[C_1](\text{Pop}, e) \lor P[C_2](\text{Pop}, e)$$

$$P[\text{NO } C](\text{Pop}, e) = \neg P[C](\text{Pop}, e)$$

$$P[\text{FOR-EACH } x \text{ IN } P \text{ HOLDS } C](\text{Pop}, e) = \forall x \in \{ x \rightarrow \{ y \} \}$$

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

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

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

A more complex example in the context of activity graphs (see Fig. 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: $\text{NO Activity ANY-REPETITION-OF (Activity being-decomposed-into Activity-graph COMPRISING Activity) THAT Activity.}$

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 $\mathcal{J}$ be an information structure and let $\text{Pop}$ and $\text{Pop}'$ be populations of $\mathcal{J}$ ($\text{IsPop}(\mathcal{J}, \text{Pop})$ and $\text{IsPop}(\mathcal{J}, \text{Pop}')$), then $\text{Pop} \sqsubseteq \text{Pop}'$ if and only if:

$$\forall x \in x \in \text{Pop}(x) \equiv \text{Pop}'(x).$$

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 $U : \text{Update statement} \times \text{POP} \times \text{ENV} \rightarrow \text{POP}$, 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 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 $\text{Pop}'$ of the current population $\text{Pop}$, such that information descriptor $P$ has no empty result in the extended population $\text{Pop}'$. Formally, this meaning is expressed by: $U[\text{ADD } P](\text{Pop}, e)$ is a minimal population $\text{Pop}'$ such that:

1. $\text{IsPop}(\mathcal{J}, \text{Pop}')$,  
2. $\text{Pop} \sqsubseteq \text{Pop}'$ and  
3. $\mu \left[ D[P](e) \right](\text{Pop}') \neq \emptyset$.

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

$$\text{ADD Address: 'New York', 'Fifth Avenue', 17}.$$
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 instance 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 population: 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:

ADD President.

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

ADD President having-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 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: \( U[\text{DELETE } P] \) (Pop, e) is a maximal population Pop' such that:

1. \( \text{lsPop}(\emptyset, \text{Pop}') \),
2. \( \text{Pop}' \subseteq \text{Pop} \) and
3. \( \mu[D[P](e)](\text{Pop}') = \emptyset \).

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 \( \text{Nm} : \emptyset \rightarrow \mathcal{P \delta} \) for object types is derived as follows:
Formal definition of a conceptual language

\[
Nm(X) = \begin{cases} 
X & \text{if } X \text{ is a label type}, \\
\delta(X, \text{Nm(Elt}(X))) & \text{if } X \text{ is a power type} \\
or a sequence type, \\
\delta(X, \bigcup_{X'} \text{Nm}(Y)) & \text{if } X \text{ is a composition type}, \\
\left[\text{Nm}(X_1) \cap P_1, \ldots, \text{Nm}(X_n) \cap P_n\right] & \text{if } X \text{ is an object type, identified as } X(P_1, X_1, \ldots, P_n X_n).
\end{cases}
\]

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 \( \text{Val} : \text{Query} \times \text{ENV} \rightarrow \mathcal{P} \mathcal{E} \) that maps queries on path expressions:

\[
\text{Val}([\text{LIST}\ P_1, \ldots, P_n, P])(e) = [\text{StdNames} \cap D[P_1] (e), \ldots, \text{StdNames} \cap D[P_n] (e) | D[P] (e)],
\]

where StdNames denotes all standard naming conventions: \( \text{StdNames} = \bigcup_{X \in \mathcal{E}} \text{Nm}(X) \). 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 prove properties. In Ref. [36], LISA-D and Task-Structures [37] 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 [38–40] based on EVORM, an extension of PSM supporting evolution [41]. Furthermore, research is conducted providing a better disclosure of the information stored in the information system [42], by means of an approach based on stratified hypermedia architecture [43]. Currently a prototype implementation of LISA-D is being developed.

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.

REFERENCES


Appendix opposite
APPENDIX

Legend of Graphical Symbols

This Appendix contains an overview of the symbols for object types, generalizations and specializations, and graphical constraints used in this paper.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Object type" /></td>
<td>object type</td>
</tr>
<tr>
<td><img src="image" alt="Label type x" /></td>
<td>label type x</td>
</tr>
<tr>
<td><img src="image" alt="Role" /></td>
<td>role</td>
</tr>
<tr>
<td><img src="image" alt="Predicator" /></td>
<td>predicator</td>
</tr>
<tr>
<td><img src="image" alt="y power type of x" /></td>
<td>y power type of x</td>
</tr>
<tr>
<td><img src="image" alt="y sequence type of x" /></td>
<td>y sequence type of x</td>
</tr>
<tr>
<td><img src="image" alt="y is generalisation of x" /></td>
<td>y is generalisation of x</td>
</tr>
<tr>
<td><img src="image" alt="y is specialisation of x" /></td>
<td>y is specialisation of x</td>
</tr>
<tr>
<td><img src="image" alt="Uniqueness constraint over a single fact type" /></td>
<td>uniqueness constraint over a single fact type</td>
</tr>
<tr>
<td><img src="image" alt="Uniqueness constraint over several fact types" /></td>
<td>uniqueness constraint over several fact types</td>
</tr>
<tr>
<td><img src="image" alt="Total role or cover constraint" /></td>
<td>total role or cover constraint</td>
</tr>
<tr>
<td><img src="image" alt="Occurrence frequency constraint or cardinality constraint" /></td>
<td>occurrence frequency constraint or cardinality constraint</td>
</tr>
<tr>
<td><img src="image" alt="Exclusion constraint" /></td>
<td>exclusion constraint</td>
</tr>
<tr>
<td><img src="image" alt="Membership constraint" /></td>
<td>membership constraint</td>
</tr>
<tr>
<td><img src="image" alt="Subset constraint" /></td>
<td>subset constraint</td>
</tr>
<tr>
<td><img src="image" alt="Equality constraint" /></td>
<td>equality constraint</td>
</tr>
<tr>
<td><img src="image" alt="Enumeration constraint" /></td>
<td>enumeration constraint</td>
</tr>
</tbody>
</table>