Formal semantics of advanced information models

Arthur ter Hofstede and Patrick van Bommel

Radboud University Nijmegen

November 2017

Contents

1 Introduction **5**

1.1 Problem area 5
1.2 A framework for describing IS methods 5
1.3 Information modelling 6
1.4 Data intensive domains 7
1.5 Information modelling requirements 7
1.6 Meta-modelling 15
1.7 Issues for discussion 15

2 Information structures **17**

2.1 Basics 17
2.2 Data model populations 36
2.3 Population rules 40
2.4 Type relatedness 41

3 Two examples of meta-models **43**

3.1 Meta-model of entity-relationship diagrams 43
3.2 Meta-model of data flow diagrams 45

4 Integrity constraints **47**

4.1 Introduction 47
4.2 Uniqueness constraints 47
4.3 Occurrence frequency constraints 59
4.4 Total role constraints 61
4.5 Set constraints 62
4.6 Enumeration constraint 65
4.7 Power type constraints 65
4.8 Specialisation constraints 68
4.9 Subtype defining rules 69
4.10 Schema type constraints 70
4.11 Relational algebra 71

5 Identification and verification **81**

5.1 Introduction . . . 81
5.2 Weak identification . . . 81
5.3 Structural identification . . . 83
5.4 Theoretical results . . . 90
5.5 Verification . . . 91
5.6 Complexity of verification . . . 95
5.7 Self-meta-modelling . . . 98
5.8 Motivation from the axioms of set theory . . . 102
5.9 Data models for context-free grammars . . . 103

6 Path expressions **109**

6.1 Introduction . . . 109
6.2 Multisets . . . 109
6.3 Syntax and semantics of path expressions . . . 111
6.4 Information descriptors . . . 120

A Mathematical Notation **129**

A.1 Functions . . . 129
A.2 Sets and tuples . . . 130

B Graphical Notation **131**

C Further formalization **133**

C.1 Properties of specialisation . . . 133
C.2 Population rules . . . 133
C.3 Object types in relational expressions . . . 135
C.4 Derivation rules for dependency of object types . . . 135

D Further application **137**

D.1 Chipkaarten vanuit informatiekundig perspectief . . . 137
D.2 Miracles To Save The Realm . . . 138

Bibliography **139**

Chapter 1

Introduction

1.1 Problem area

The development of large software applications is a difficult problem. It appears that this problem only increases as applications tend to become more and more complex. The development of information systems, one of the more frequently occurring applications, is a typical example of this problem. Information systems that are reliable, delivered in time, and meet users' expectations are rare.

In order to improve the *productivity* of developers and the *quality* of the resulting information systems, methods and techniques have been introduced (see e.g. [65]). The different aspects of information systems development methods can be captured in the framework presented in [76]. This framework will be discussed in the next section.

1.2 A framework for describing IS methods

The goal of the framework of [76] is to provide a better understanding of information systems development. In this framework, graphically represented in figure 1.1, a distinction is made between a *way of thinking*, a *way of controlling*, a *way of modelling*, a *way of working* and a *way of supporting*.

The focus of this text is the way of modelling (this text is based on [35]). The way of modelling encompasses the *modelling concepts* used in information systems development, the *rules* regarding these modelling concepts, their *interrelationships*, and their *representations*. As such, the way of modelling consists of a (possibly integrated) set of *techniques*. The syntax of a technique states which models are *well-formed*, while the semantics states the formal meaning of well-formed models.

The way of working structures the strategy determining the manner in which information systems are developed. Strategy deals with the identification of relevant tasks in the development process and their feasible order. Naturally, the way of modelling and the way of working are closely related, since intermediate models may influence the course of the modelling process, and modelling tasks adapt and create models.

The way of thinking involves the philosophical starting points and basic objectives of a method, while the way of controlling deals with project management aspects. The way of supporting is defined as a collection of tools. Tools are *supporting means* for performing and facilitating systems development tasks. Tools may be automated (e.g. database management systems, code generators) or not automated (e.g. whiteboards, templates).

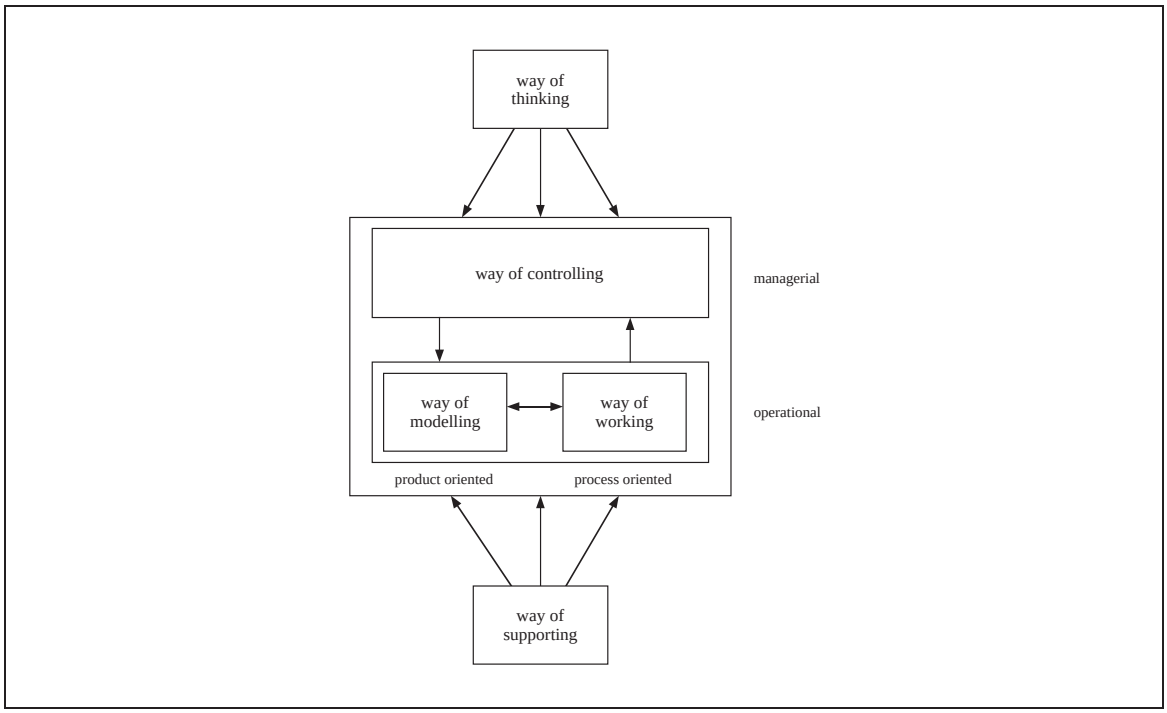

Figure 1.1: Framework for understanding information systems development

1.3 Information modelling

Most methods for the development of information systems distinguish a number of phases as part of their way of controlling. An important and very difficult goal of the *early* phases of systems development is the acquisition and representation of requirements. This part of system modelling is commonly referred to as *requirements engineering*. Requirements engineering often turns out to be the bottleneck of systems development, since the acquisition of requirements is notoriously difficult and it is a well-known fact that the later in the development process an error is detected, the more expensive it is to correct it (see e.g. [17]).

Information modelling is defined as that part of requirements engineering which involves a high-level, problem-oriented and implementation independent description of an information system. Information modelling aims at a description of *what* an information system does, or should do, as opposed to *how* the information system should do that. Information modelling only deals with functional requirements, not with non-functional requirements (e.g. real-time requirements). Information modelling results in an *information model* or *conceptual model*.

Information systems development methods usually contain, as part of their way of modelling, several techniques dealing with (aspects of) information modelling. Many information systems development methods however, have a much broader scope as they cover large parts of the systems life cycle. Therefore, they also include techniques that deal with aspects not considered to be part of information modelling, e.g. dialogue specifications or screen design.

An information model models part of a real or postulated world. This area of concern is referred to as the *Universe of Discourse*, see [29]. An alternative term is *application domain*, or simply, *domain*. Using the terminology of [74], a Universe of Discourse corresponds to an information system in the *broader* sense. An information system in the *broader* sense covers *all* informational aspects of the business system, while an information system in the *narrower* sense only covers its computerised aspects. Information modelling

therefore concerns information systems in the broader sense. Business modelling on the other hand aims at a description of all aspects of the business system including its organisational structure, objectives, critical success factors etc.

An information model can be viewed from different perspectives. The most common perspectives are the data perspective and the process perspective. Sometimes also the behaviour perspective is added (see e.g. [59]). Often, information systems development methods tend to be dominated by one perspective. In the next section focus will be on application domains for which the data perspective is the most important perspective.

1.4 Data intensive domains

In the field of information systems, *data intensive* domains play a prominent role. These domains are particularly suited for database applications. In such applications large quantities of (more or less) structured data, and few operations, have to be dealt with. Consequently, the data perspective is an important perspective.

Data intensive domains vary from simple to very complex. Many administrative systems can be characterised as simple, their implementation does not pose severe problems in general. This is different for complex data intensive domains. Examples of complex data intensive domains can be found in the areas of office automation, multi-media, web site modelling, networked databases (e.g. Internet or Intranet), and CAD/CAM. These areas require advanced data modelling concepts. Consider e.g. the representation of document structures in the field of office automation and multi-media or products in the field of CAD/CAM. Another area with (complex) application domains, which can be classified as data intensive, is *meta-modelling*, which deals with the representation of method knowledge. This area is discussed in more detail in section 1.6.

1.5 Information modelling requirements

An important role of a conceptual model is to provide a common understanding of the Universe of Discourse involved. This implies that a conceptual model should have a unequivocal meaning. To this end an information modelling technique should have a well-defined *formal* semantics. In addition to that, the information modelling technique should have sufficient *expressive power* to describe the Universe of Discourse.

As conceptual models play a crucial role in the communication with domain experts, they should be *comprehensible*. Insight in the meaning of a conceptual model might be improved by its execution. Therefore, conceptual models have to be *executable*.

Finally, a conceptual model serves as a basis for the future implementation of the (computerised parts of the) information system. Apart from the requirement mentioned before, which states that an information modelling technique should have a formal semantics, this also implies that an information modelling technique should be on a *conceptual* level. This prevents implementation decisions from having to be made in too early a phase, which could lead to suboptimal solutions during actual implementation. Also, models which are not on a conceptual level, are generally not easily understood by domain experts.

Now, each of the requirements discussed above, will be discussed in more detail.

1.5.1 Formal foundation

It has often been emphasized that requirement specification languages should have a rigorous formal basis (see e.g. [13], [71], [67], [48], [42]). However, in the "*Methodology Jungle*" in the area of information systems development (a term first introduced in [2]), most organisations and research groups have defined their own methods. The techniques advocated in these methods usually do not have a formal foundation. In some

cases their syntax is defined, but attention is hardly ever paid to their (formal) semantics. The discussion of numerous examples, mostly with the use of pictures, is a popular style for the "definition" of new concepts and their behaviour. This has led to *fuzzy* and *artificial* concepts in information systems development methods (see also [10]).

The resistance to formalisation in the field of information systems has been overcome. In several papers the need for formal foundations in this field has been addressed (see e.g. [63], [26], [41], [70], [24], [36]). In [30] it is noted that without a formal approach it is difficult to avoid deficiencies such as inconsistency, lack of structure, overspecification, incompleteness, ambiguity, and redundancy. These problems are caused by the fact that informal notations can be ambiguous and do not allow for sophisticated (automated) support and formal reasoning. Each of these problems will be addressed and illustrated in the following sections in the context of information modelling.

Ambiguity

As stated before, many techniques have been introduced by discussing examples only. If that occurs, different interpretations will easily arise in cases not covered by these examples. Worse, situations that are not covered by any example are hardly ever recognised. The intention behind examples is to give hands on experience, in order to try to convey the general idea. This may however be difficult and sometimes even impossible. As an example, we consider data models as presented in [58].

In figure 1.2 a simple data model, taken from [58], is shown. In this figure, a uniqueness constraint is expressed over a so-called objectified fact type (*Enrollment*). The schema of figure 1.2 is a well-formed schema. The meaning of the schema requires knowledge of the meaning of the uniqueness constraint. The semantics of this constraint is explained in [58] by stating that it expresses a key on the ternary relation that is the result of *flattening* objectified fact type *Enrollment*. This key is shown in figure 1.3 and implies that for *Person* the combination of *Subject* and *Position* is unique.

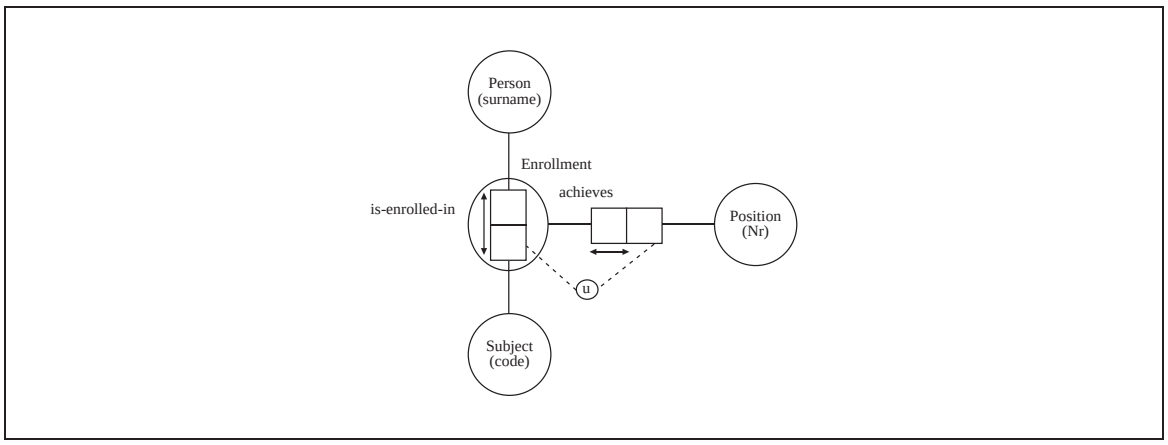

Figure 1.2: Uniqueness constraint over objectification

This example does not seem difficult to understand, but does certainly not convey the general idea. The relation between the fact types in figure 1.2 on the one hand and the fact type in figure 1.3 on the other hand is not clear. As stated before, this relation is to be found by flattening the objectified fact type *Enrollment*. Flattening is not an operation which is easily explained intuitively.

Furthermore the reader might get the impression, that the schemata of figures 1.2 and 1.3 are equivalent (which they are *not*, as contrary to figure 1.3, in figure 1.2 a *Position* is not required for each recorded combination of a *Person* and a *Subject*), which would allow for switching freely between both alternatives, depending on the taste of an information analyst. As another example, what is the meaning of the uniqueness constraints in figures 1.4 and 1.5?

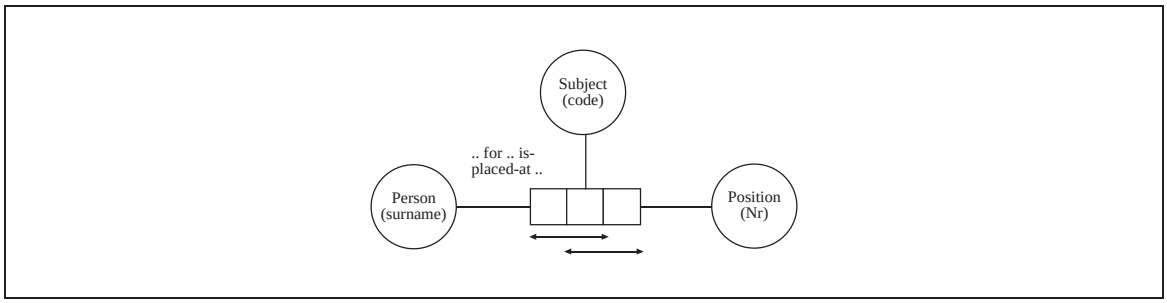

Figure 1.3: Semantics of uniqueness constraint

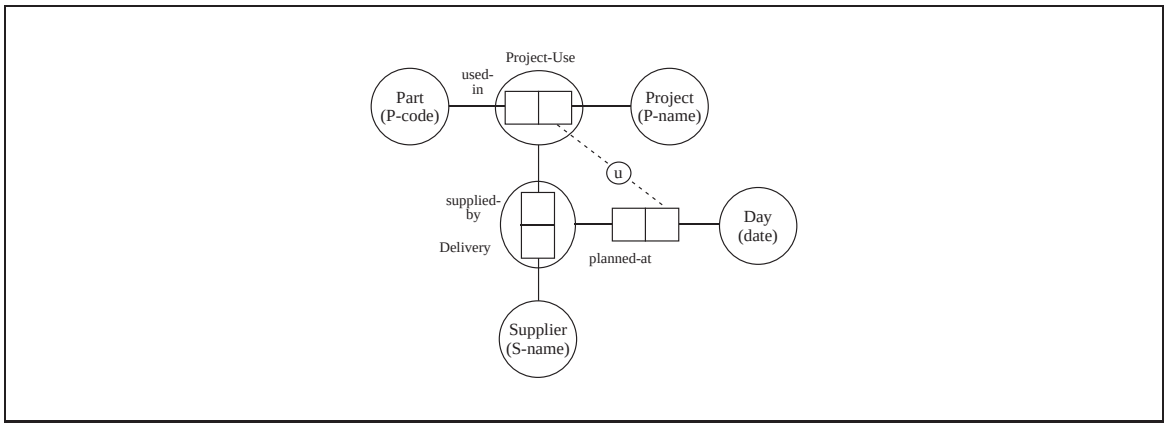

Figure 1.4: Complex uniqueness constraint

The consequences of different interpretations among persons, or persons and machines, can be disastrous. Interpretation problems will certainly cause the resulting system to malfunction.

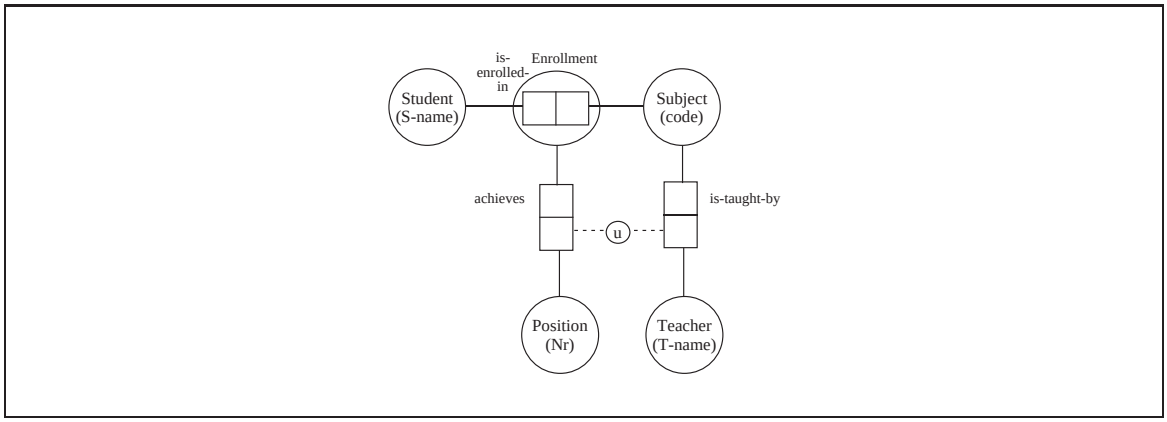

Figure 1.5: Another complex uniqueness constraint

Automated support

A second reason for the need of formalisation is that a formal model provides a good clue for implementation. The better the formalism, the easier the implementation. In fact, an implementation can be seen as yet another, but enormously more detailed formal description.

Not only the implementation benefits from a formal description. It also provides scope for much more sophisticated and extended support to information analysts. Due to the lack of formality of the underlying concepts of the supported techniques, automated tools are often not able to perform sophisticated verification checks, nor give warnings on suspect constructs.

Consider the data models of figures 1.6 and 1.7.

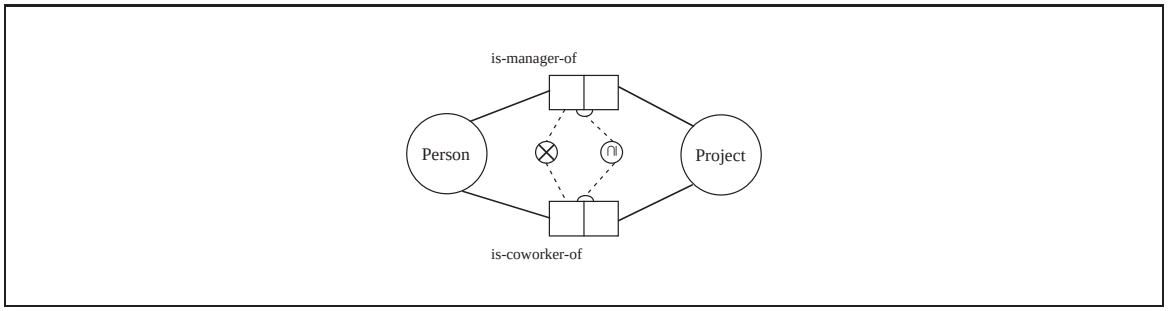

Figure 1.6: Inconsistent data model

The inconsistencies in these models are the following.

1. In figure 1.6, the subset constraint states that a *Person* managing a certain *Project* should also be a coworker of that Project. The exclusion constraint in figure 1.6 expresses that a *Person* is never both a manager and a coworker. Clearly, the only populations of this schema that satisfy both requirements are those populations with no managers.
2. The subset constraint of figure 1.7 states that a *Company* that produces a certain *Product* also promotes that *Product*. The uniqueness constraint on the upper fact type states that a *Company* may produce several *Products*, and that a *Product* may be produced by several *Companies*. The uniqueness constraint on the lower fact type states that a *Company* may promote at most one *Product*. The only populations that satisfy these three constraints, are those populations where each *Company* produces at most one *Product*.

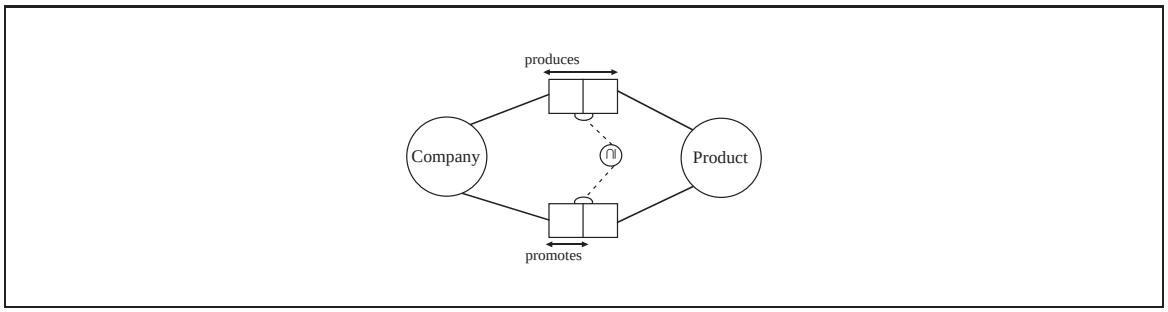

Figure 1.7: A suspect data model

In both cases we have well-formed schemata, i.e. schemata that fulfil the syntactical rules. If automated tools are to detect the inconsistencies mentioned above, they need to be aware of the semantic definition of the data modelling technique used (and in this case, of constraints in particular).

Checking whether a model has certain properties, using the rules of a technique, is called *verification*. If these properties are formally expressed and computable, it is possible to perform verification automatically. Checking whether a model is well-formed according to a certain technique, is an example of verification. Often however, verification not only deals with well-formedness, but also deals with more complicated checks (as e.g. required in schema 1.6), usually referred to as static semantic checks.

Another important issue regarding automated support is *validation*, the question whether a well-formed model meets its intended requirements in the Universe of Discourse. In contrast with verification, this cannot be checked by a computer, as the Universe of Discourse is only available in a highly informal format. Validation however, can be *supported* by tools. By executing a specification, a domain expert can improve his (or her) understanding of the meaning of that specification. By offering a domain expert the possibility to specify a population of a certain data model, which is considered to be correct, it can be checked whether that population satisfies the constraints in that data model, and therefore, whether these constraints meet their intended meaning.

An extended form of validation is testing on *plausibility*. Certain constructs are accepted during verification, yet are most probably incorrect. In some cases these implausibilities can be detected automatically. The implausibility of the schema of figure 1.7 is an example. Another example is a data flow diagram that contains a process that produces information out of nothing (i.e. a process without input flows), or a process that only consumes information (i.e. a process without output flows). Another example would be a program fragment

$$\mathsf{i} := 1;\ \mathsf{S};\ \mathsf{i} := 2$$

where program fragment S does not contain an application of variable i, directly or indirectly. In this case, the assignment $\mathsf{i} := 1$ is superfluous. It is not plausible that the programmer writes useless statements, so a warning should be given.

Properties of modelling techniques

A third reason for formalisation is that a formal definition makes it possible to formulate and prove properties. For example, one could look at the expressive power of a technique and compare it with the expressive power of other techniques. A conclusion could be, that a certain technique allows a much more precise specification than another technique and that this other technique provides insufficient possibilities to exclude invalid situations.

Furthermore, models specified according to the same technique can be compared and (dis)proved equivalent. This is important in order to abstract from the peculiarities of an information analyst. The introduction of a set of transformation rules, transforming models into equivalent models, offers the opportunity to describe normal forms for well-formed models. A normal form is a base for comparing different models, probably originating from different information analysts trying to model the same Universe of Discourse.

Also, properties of specific well-formed models according to some technique can be formulated and proved. An example of a property of a data model is structural identifiability, i.e. the question whether every entity type is identifiable. For process models it is interesting to determine whether they are free of deadlock or starvation.

From the above it is clear that properties can be proved both for models in a technique and for a technique itself.

1.5.2 Conceptual level

An important principle that is included in the definition of information modelling is the *Conceptualisation Principle* ([29]). This principle states that conceptual models should deal only and exclusively with aspects of the Universe of Discourse. Any aspects irrelevant to that meaning should be avoided. Examples of these conceptually irrelevant aspects are e.g. (see [29]) aspects of (external or internal) data representation,

physical data organisation and access as well as all aspects of particular external user representation such as message formats and data structures.

The Conceptualisation Principle is important because in the early phases of systems development no implementation decisions should have to be made. Such decisions easily become outdated and impede successful implementation. Furthermore, they tend to lead to models that are difficult to comprehend and to communicate.

As an example, sometimes many-to-many relations are not allowed, but need to be replaced by one-to-many relations (see figure 1.8). This restriction is a clear example of a violation of the Conceptualisation Principle.

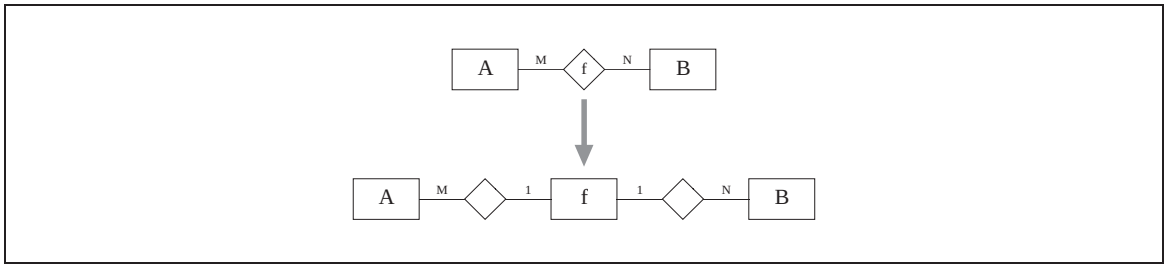

Figure 1.8: Transformation of many-to-many relations

As another example, the traditional Relational Model ([12]) also violates the Conceptualisation Principle as this technique is based on the notion of tables. Tables are a particular kind of data structure, and therefore not on a conceptual level.

1.5.3 Expressive power

The *100 Percent Principle* ([29]) states that a conceptual model (or information model) should describe all relevant static and dynamic aspects of the Universe of Discourse. An information modelling technique should therefore be capable of modelling the data and process perspective of an application domain adequately. This implies that an information modelling technique should have sufficient *expressive power*.

As an illustration of the notion of expressive power, consider entity structure diagrams. In these diagrams, the central notion is *action*. An action can be an ordered sequence of other actions, it can be a choice between actions, an iteration of another action (zero or more times) or it can be an atomic action. As an example of an entity structure diagram consider figure 1.9 taken from [47].

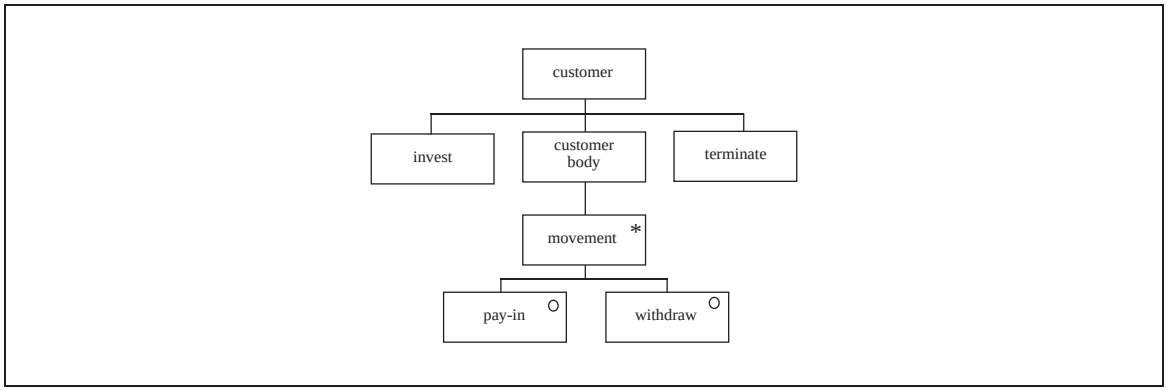

Figure 1.9: Example of entity structure diagram

In the action diagram in figure 1.9, the actions *invest*, *pay-in*, *withdraw* and *terminate* are atomic. The action *movement* represents a choice between the actions *pay-in* and *withdraw*. The action *customer body* consists of a number of *movements*. Finally, the action *customer* first performs the action *invest*, then *customer body* and then *terminate*.

Entity structure diagrams correspond to regular expressions, which have a very limited expressive power. A simple domain in which an action c consists of first performing an action a a number of times and then an action b the *same* number of times, cannot be modelled in entity structure diagrams. The reason is that the language $\{ a^n b^n \mid n \in \mathbb{N} \}$ is not regular (see e.g. [53]). Regular expressions are less powerful than context-free grammars which on their turn are less powerful than Turing machines. The *Church-Turing Thesis* states that every computable function can be computed on a Turing machine (or alternatively: is recursive).

As another example consider the data flow diagram of figure 1.10. Process *Report Production* has flows *order information* and *product information* as input and flows *monthly report* and *management report* as output. From this diagram it cannot be derived whether both *order information* and *product information* are necessary to produce a *monthly report* or a *management report*. This is caused by the fact that in data flow diagrams, it is not possible to express the precise input-output relations. To this end, the technique lacks expressive power.

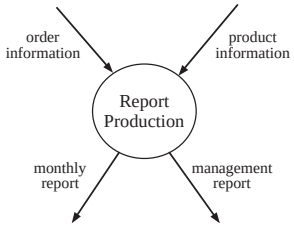

Figure 1.10: Process with input and output flows

An example in the context of data modelling would be the situation of projects having coworkers and managers. This can be modelled as in figure 1.11. The constraint stating that a person cannot manage a project of which (s)he is a coworker, cannot be expressed in the original Entity-Relationship model. Therefore, such constraints had to be expressed in natural language.

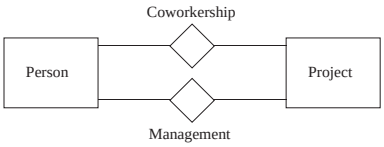

Figure 1.11: Example data model

Lack of expressive power might result in violations of the Conceptualisation Principle. Choices that are not relevant with respect to the Universe of Discourse then might have to be made, leading to *overspecification*, or even worse, the Universe of Discourse might have to be adapted. As an illustration consider the Relational Model. As [49] notes, the Relational Model cannot represent complex nested entities (e.g. compound documents). When an analyst attempts to represent such entities in a relational schema, these entities have to be flattened.

1.5.4 Executable

Validating requirements as early as possible might prevent errors in later stages of systems development. As stated before, the later an error is detected the more it will cost to correct it. To make validation possible in the early phases of system development it is important that a specification can be executed. This enhances the understanding of the meaning of a specification and its implications considerably. Therefore, it is to be preferred that an information modelling technique yields executable models. Note that in order to be executable, a technique should be formal. A formal technique on the other hand, is not necessarily executable.

1.5.5 Comprehensible

Since one of the important roles of conceptual models is to reach a common understanding of the Universe of Discourse, not only between information analysts, but especially between domain experts and information analysts, it is vital that a conceptual model is *comprehensible*. While the previous requirement stated that an information model should be interpretable by a machine, this requirement states that it should be interpretable by human beings. In [20] it is stressed that languages for conceptual modelling should be *easy to use* and *easy to learn*.

One way of achieving comprehensibility is by offering structuring mechanisms. In this way a specification remains surveyable. An often used structuring mechanism is decomposition. In *state transition diagrams* for example, a state in a state transition diagram may be decomposed into another state transition diagram.

Often, comprehensibility is achieved by the use of graphical notations. The importance of *visual* formalisms can be described as follows ([33]): *visual, because they are to be generated, comprehended and communicated by humans, and formal, because they are to be manipulated, maintained, and analyzed by computers.*

In [71] several reasons are given why a graphical representation is more comprehensible than its textual counterpart:

1. Graphics is in two dimensions, while text is in one dimension. The former gives an additional degree of freedom in presentation.

2. Graphics is more useful in showing the hierarchical structure of complex systems and more natural in describing parallelism.

3. A person reading graphics can do so selectively, depending on the level of details required. If he reads text, he has to do so linearly.

4. There is a limit to the number of concepts which can be reasonably held in the short-term memory of the human mind ([57]). A person reading graphics can start off generally and go down to detail after some degree of familiarisation. If one is reading text, then one has to start off with detail and abstract the skeleton concepts afterwards.

Of course, it is not always possible, or desirable, to fully represent a specification graphically. In [15] extensions to data models were proposed that allow for the graphical specification of complex constraints. These extensions tend however to clutter the diagrams and make them hard to read.

Finally, specifications should be in a style close to intuition. Specifications resembling natural language for example are close to the human intuition. An important presumption is then that the formal semantics is close to the intuitive semantics. In [4] this is stated as a requirement for assigning formal semantics.

1.6 Meta-modelling

A *meta-modelling technique* is a technique that represents the way of working and the *structural* and *representational* aspects of the way of modelling. This definition implies that a meta-model does not capture the formal *semantics* of techniques, only their syntactical aspects. This is the main distinction between formalisation and meta-modelling.

A meta-modelling technique can be considered as a special kind of information modelling technique. Representation of the way of working of a method requires a process perspective, while representation of the way of modelling requires a data perspective. The only distinction is that representational aspects are not important in the case of information modelling, but are important in the case of meta-modelling. Representational aspects refer to the way models of a technique are represented, they concern graphical or textual conventions. For example, the fact that processes in a variant of data flow diagrams are represented as boxes and flows as arrows. The distinction between structural and representational aspects corresponds to the distinction between abstract syntax and concrete syntax in the field of programming languages (see e.g. [56]). In the remainder of this book no attention is paid to representational aspects in the context of meta-modelling. This issue has been addressed in [38] and [37].

Since a meta-modelling technique is a special kind of information modelling technique, the same requirements for information modelling techniques apply to meta-modelling techniques. In particular, meta-models should be *comprehensible*. Comprehensibility is less important for formal models. Due to the fact that formal models should also capture formal semantics, which requires the use of mathematics, comprehensibility for persons not familiar with mathematics can usually not be achieved. This explains the aforementioned difference between meta-modelling and formalisation.

Meta-modelling deals with complex application domains as meta-models often contain complex object types, for example when a modelling technique allows for decomposition. It is not natural to represent these object types as flat structures ([75]). Another reason for its complexity is that the syntactical rules for techniques can be quite complicated. For example, in data flow diagrams it is required that processes are hierachically decomposed. To express these kinds of rules, a powerful constraint language is required.

1.7 Issues for discussion

Issues for discussion are e.g. the following:

1. Development of information models related to the Internet, e.g. database applications made accessible via the WWW. An overview of relevant topics is presented in [7]. What is the role of Unified Modelling Language (UML) here, and how does this compare to the data models we use? How can data models be used in the context of client-server applications?

2. Integration of the theory behind information modelling with the theory behind information retrieval (in the sense of *document* retrieval.

3. Description of visual data models ([14]): *Both the way we look at data through a database system, and the nature of data we ask such a system to manage, have drastically evolved, moving from text, images, sound, and video to compositions of these media as known from multimedia applications. Visual representations are used extensively within new user interfaces and are an essential requirement for accessing data through a database system. Powerful visual approaches are used for data manipulation, including the investigation of three-dimensional display techniques and animated interfaces.*

4. How does information modelling fit in approaches such as Rapid Application Development, Iterative or Evolutionary System Development, and Joint Application Development? The idea here is that first a (restricted) user interface is built. Then a suitable conceptual information model is defined, along with a corresponding database representation to be implemented. After validating this prototype

with the users, the system is extended by (a) adding more functionality to the user interface and (b) introducing new object types and constraints in the information model.

5. Development of systems in which the groupware principle is important. Are existing information modelling techniques suitable for this purpose, or do we need additional instruments?

Chapter 2

Information structures

2.1 Basics

In this section the syntax of schemata without constraints (e.g. total role constraints and uniqueness constraints) is defined. A schema without constraints is referred to as an *information structure*. We first give an overview. This overview will be treated in more detail in later sections.

2.1.1 Overview

An *information structure* is a structure consisting of specific basic components. These components include the following sets:

1. A finite set $\mathcal{P}$ of *predicators* or *roles*. A predicator is intended to specify the role played by an object type in a fact type. This is discussed in section 2.1.3. We assume the reader is familiar with *sets*. An overview of basic definitions is given in section 2.1.2.
2. A nonempty finite set $\mathcal{O}$ of *object types*.
3. A set $\mathcal{L}$ of *label types* (see section 2.1.3). Label types are also object types: $\mathcal{L} \subseteq \mathcal{O}$.
4. A set $\mathcal{E}$ of *entity types* (see section 2.1.3). Entity types are also object 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* to be further explained in section 2.1.3. Fact types are also object types: $\mathcal{F} \subseteq \mathcal{O}$. The auxiliary function $\mathsf{Fact} : \mathcal{P} \to \mathcal{F}$ yields the fact type in which a given predicator is contained, and is defined by: $\mathsf{Fact}(p) = f \iff p \in f$. Some general properties of fucntions are treated in section 2.1.2.
6. A set $\mathcal{G}$ of *power types* (see section 2.1.4). Power types form a special class of object types: $\mathcal{G} \subseteq \mathcal{O}$.
7. A set $\mathcal{S}$ of *sequence types* (see section 2.1.5). Sequence types form a special class of object types: $\mathcal{S} \subseteq \mathcal{O}$.
8. A set $\mathcal{C}$ of *schema types*, where $\mathcal{C} \subseteq \mathcal{O}$. These will be discussed in section 2.1.6.

Besides the above sets, an information structure contains the following functions and relations:

1. A function $\mathsf{Base} : \mathcal{P} \to \mathcal{O}$. The base of a predicator is the object type associated to that predicator.
2. A function $\mathsf{Elt} : \mathcal{G} \cup \mathcal{S} \to \mathcal{O}$. This function yields the element type of power types and sequence types.

3. A relation $\prec \subseteq \mathcal{C} \times \mathcal{O}$ describing the decomposition of schema types.
4. A binary relation Spec on object types, capturing specialisation.
5. A binary relation Gen on object types, capturing generalisation.
6. A many-sorted algebra $\mathcal{D} = \langle D, F \rangle$, with D a set of concrete domains (e.g. string, natno) and F a set of operations (e.g. $+$).
7. A function $\mathsf{Dom} : \mathcal{L} \to D$ from the label types to the set D of concrete domains. The instances of a label type come from its associated domain (see section 2.2).

Figure 2.1 shows the graphical representation of an information structure, i.e. an *information structure diagram*, that is defined by:

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

where $f = \{p,q\}$, $g = \{r,s,t\}$, $h = \{u,v\}$ and $i = \{w,x\}$. With respect to the predicators, we have $\mathsf{Base}(p) = B$, $\mathsf{Base}(q) = A$, $\mathsf{Base}(r) = f$, etc. Furthermore, $\mathsf{Elt}(E) = A$, $D\,\mathsf{Spec}\,B$, $C\,\mathsf{Gen}\,D$, and $C\,\mathsf{Gen}\,G$. Finally, it is assumed that $\mathsf{Dom}(F) = \mathbb{N}$, which is not graphically represented. The set of natural numbers $\mathbb{N}$ has to be one of the sorts of the many-sorted algebra $\mathcal{D}$. This algebra is not important for this example and is therefore omitted.

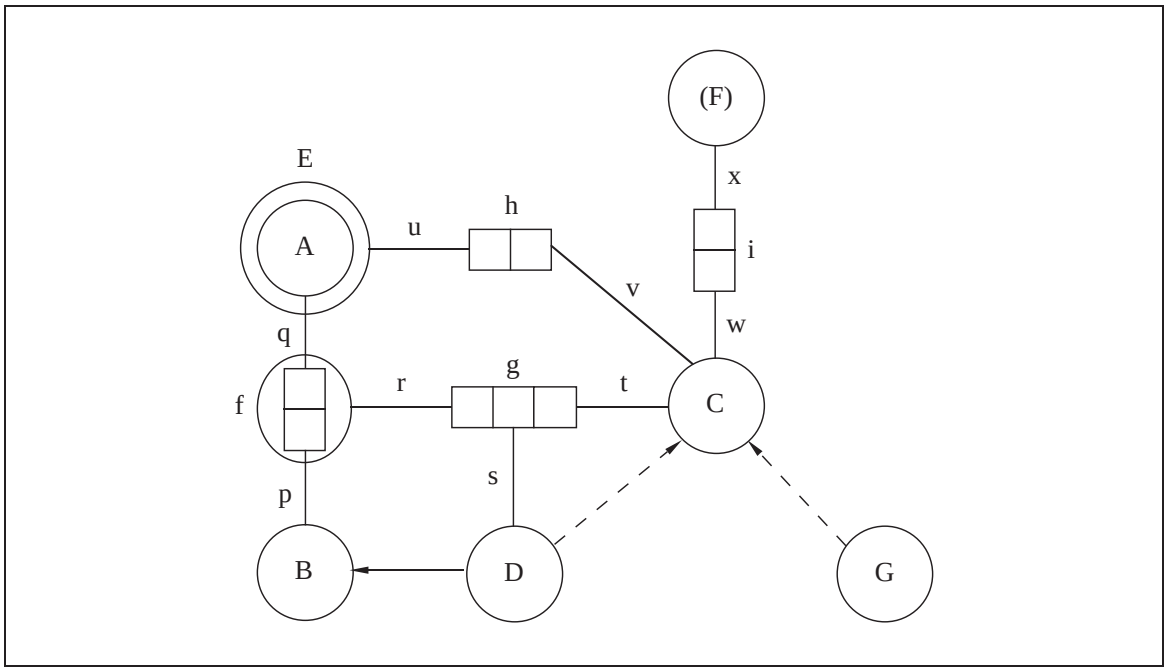

Figure 2.1: Example information structure

Due to the different interpretation that will be given to label types, entity types, fact types, power types, sequence types, and schema types, these kinds of object types are all considered to be different. Therefore, they do not share elements:

$$|\mathcal{L} \cup \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{S} \cup \mathcal{C}| = |\mathcal{L}| + |\mathcal{E}| + |\mathcal{F}| + |\mathcal{G}| + |\mathcal{S}| + |\mathcal{C}|$$

Furthermore, these object types are the only possible kinds of object types:

$$\mathcal{L} \cup \mathcal{E} \cup \mathcal{F} \cup \mathcal{G} \cup \mathcal{S} \cup \mathcal{C} = \mathcal{O}$$

2.1.2 Sets and functions

We often use *sets* and *functions*. In this section we give an overview of basic definitions. Let X be a set. Using the membership operator, we write $x \in X$ if x is an element of X. If x is not an element of X we write $x \notin X$. We write $Y \subseteq X$ if Y is a subset of X. We write $Y \subset X$ if Y is a proper subset of X, i.e. $Y \subseteq X$ and $Y \neq X$. We have the following basic set operators:

union:	$X \cup Y$
intersection:	$X \cap Y$
difference:	$X - Y$
product:	$X \times Y$

These operators can be defined in terms of the membership operator. Note that $X \cup Y = Y \cup X$. This commutativity also holds for intersection. Difference and product are noncommutative. It is obvious that intersection and difference do not result in new elements:

$$\begin{array}{lll} X \cap Y & \subseteq & X \\ X - Y & \subseteq & X \end{array}$$

Furthermore, union with a subset has no effect:

$$X \subseteq Y \;\; \Rightarrow \;\; X \cup Y = Y$$

A *relation* is a subset of a product. Let $h \subseteq X \times Y$ be a relation. The relation h is called a function, if each element of X is used at most once in h:

$$\forall_{(a,b),(c,d) \in h)} \, [a = c \; \Rightarrow \; b = d]$$

We then write $h : X \rightarrow Y$. Now X is called the domain of h, and Y the range of h. If h is a function, we often use the function notation $h(a) = b$ instead of $(a, b) \in h$.

Note that a function is a set, because it is a relation. Therefore, the usual set operators may be applied to functions. As an example, if h_1 and h_2 are functions, then $h_1 \cap h_2$ is a function as well. Also $h_1 - h_2$ is a function, but $h_1 \cup h_2$ is not necessarily a function.

2.1.3 Label types, entity types and fact types

Labels versus entities. In many conceptual data modelling techniques, a distinction exists between objects that can be represented directly and objects that cannot be represented directly. Often this distinction corresponds with the difference between entities and labels. Labels can be represented directly on a communication medium, while entities depend for their representation on labels. As a result, label types are also called *concrete* object types (or lexical ot's, or printable ot's), as opposed to entity types which are referred to as *abstract* object types (or non-lexical ot's or NOLOT's). The gap between concrete and abstract object types can only be crossed by special binary relationship types, called *bridge types* in [78] and *reference types* in [58]. Typical examples of label types are *Name*, *Number* and *Code*. A typical example of an entity type is *Person*. Graphically, label types can be distinguished from entity types by the fact that their names are parenthesized.

Identification deals with the question whether instances of object types can be denoted in terms of one or more labels. Identification is discussed in depth in section 5.2.

Relationships. One of the key concepts in data modelling is the concept of relationship type. Generally, a relationship type is considered to represent an association between object types. In data modelling, a

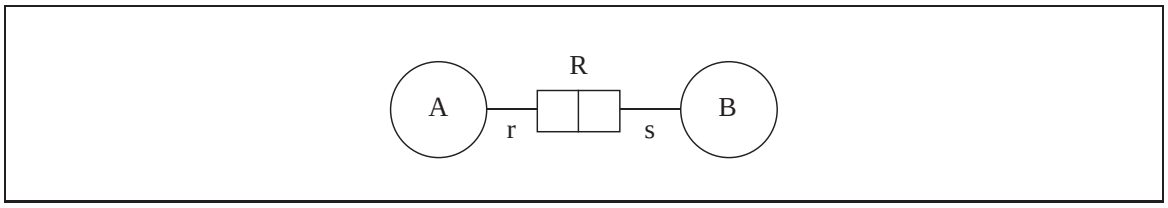

Figure 2.2: A binary fact type

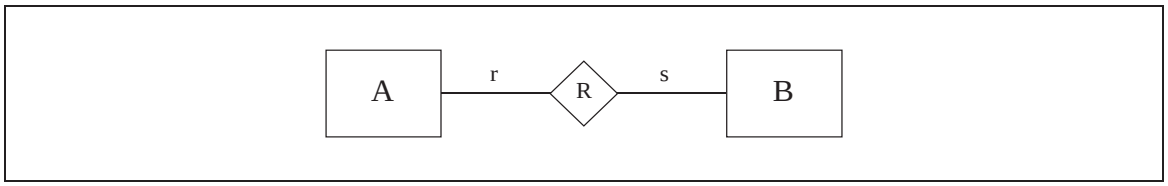

Figure 2.3: A different representation for figure 2.2

relationship type often is referred to as a *fact type*. In figure 2.2, the graphical representation of a binary fact type R between object types A and B is shown:

A different graphical representation is shown in figure 2.3. A fact type consists of a number of *roles* (r and s in figure 2.2), denoting the way object types participate in that fact type. Roles are also referred to as *predicators* (a term first introduced in [23]). This term reflects that a fact type can be considered a predicate. A predicator then denotes a position in that predicate.

In figure 2.2, the roles r and s seem to be ordered, as a consequence of the graphical layout. This view corresponds to the *tuple oriented approach*, where a relation is defined as a subset of a Cartesian product. However, often it is not desirable to enforce a specific order between the roles in a fact type. A more suitable approach then is to use the mapping mechanism to describe relations, the so-called *mapping oriented approach*, see also [54]. In our approach a fact type is therefore considered to be a *set* of predicators.

Fact types may be treated as object types, a process referred to as *fact objectification*. Fact objectification is sometimes called *aggregation* (see for example [43]). A fact type is always considered to be an object type. Graphically however, a fact type is only represented as an object type (by encircling it) if it plays a role in other fact types.

Functional representation. For the visualisation of complex operations on schemata a functional drawing style can be convenient. This is explained by the fact that operations exist of which the application does not necessarily yield a valid schema. In the resulting schemata, roles may participate in more than one fact type. The functional drawing style supports the representation of such schemata. A schema represented in the functional style is a labeled directed graph. An arrow labeled r is drawn from an object type A to a fact type f if and only if A plays role r in f. The diagram of figure 2.2 is represented functionally in figure 2.4.

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

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

The set $\mathcal{B} = \{\, f \in \mathcal{F} \mid \mathsf{Bridge}(f) \,\}$ contains all the bridge types. As a consequence of the strict separation between the concrete and the abstract level, label types usually may only participate in bridge types:

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

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

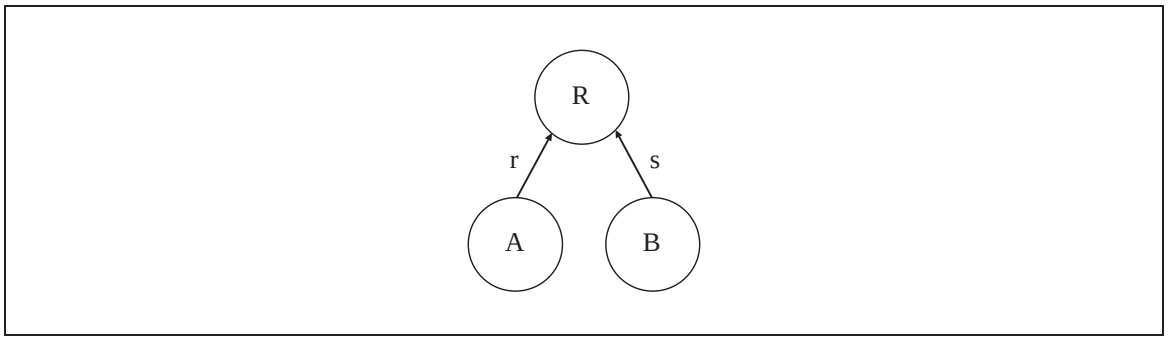

Figure 2.4: The functional drawing style

Example 2.1
In figure 2.1, i is a bridge type with $\mathsf{concr}(i) = x$ *and* $\mathsf{abstr}(i) = w$. □

2.1.4 Power types

The concept of *power type* forms the data modelling pendant of power sets in conventional set theory. An instance of a power type is a (nonempty) set of instances of its *element type*. The difference with power sets in set theory is that a power type does not generally contain *all* possible sets of instances of its element type. An instance of a power type is identified by its elements, just as a set is identified by its elements in set theory (axiom of extensionality), see also [43]. Power typing corresponds to the notion of *grouping* as present in [1], the notion of *user-controllable grouping classes* in [32] and the notion of *association* in [8].

A simple example of the application of power types can be found in the *Convoy Problem* (taken from [32]), depicted in figure 2.5. In this diagram, the additional circle around object type *Ship* represents object type *Convoy* defined as a power type with 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.

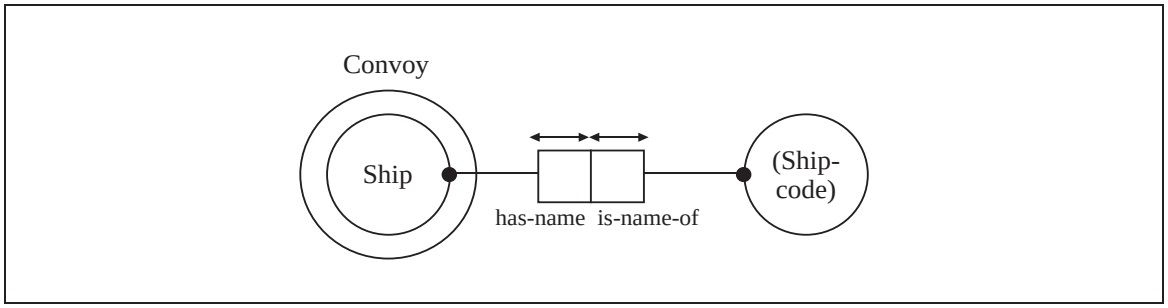

Figure 2.5: A simple example of a power type

In the schema of figure 2.5 some integrity constraints occur. The black dot on the object type *Ship* graphically represents a *total role constraint* and expresses that each instance of *Ship* must play the role *has-name*. The double-headed arrow above the role *has-name* represents a *uniqueness constraint* and expresses that instances of *Ship* play this role at most once. These constraint types are discussed in detail in sections 4.4 and 4.2.

A convoy as power type is *not* expressible in terms of models having no explicit power typing construct. Consider for example the schema from figure 2.6. This schema only implicitly states that a convoy consists of a number of ships: by means of role name *contains*. Contrary to figure 2.5, a convoy cannot be identified by

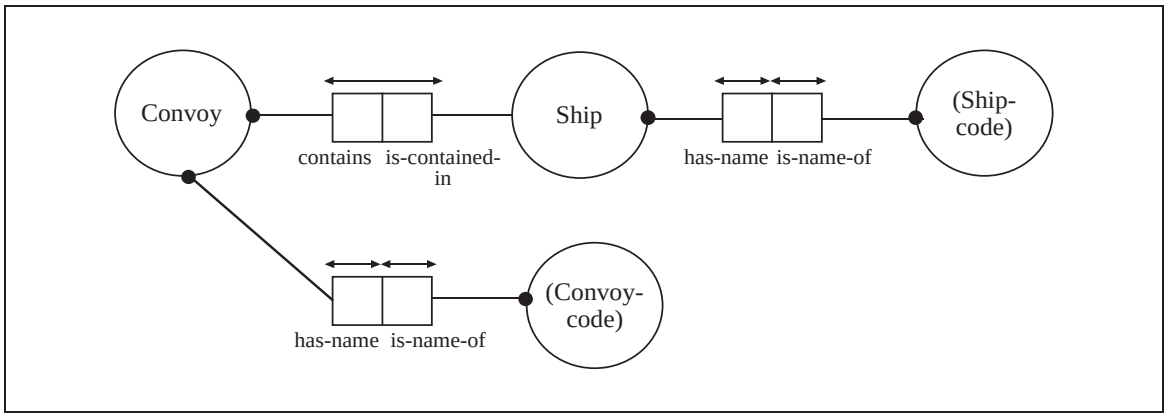

Figure 2.6: Convoy Problem without power type

its ships in this schema, and needs another form of identification, for example by the artificial introduction of a *Convoy-code*. This clearly is a violation of the Conceptualisation Principle.

Normally one would not like a convoy to be identified by its constituent ships. So a power type is not preferred here. A power type would imply that a convoy would become a different convoy when it looses one of its ships. A more realistic example is that of simple chemical reactions, a Universe of Discourse described in [22]. A chemical reaction transforms a set of input substances with their associated quantities, and produces a set of output substances in corresponding quantities.

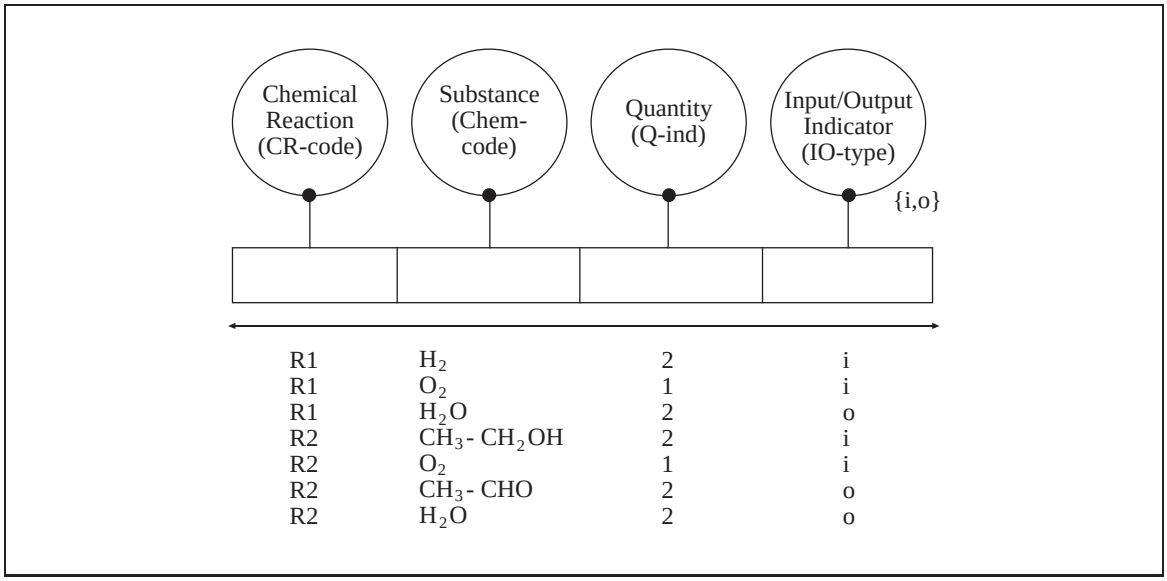

Figure 2.7: Chemical reactions without power type

This Universe of Discourse could be modelled without power types in terms of a quarternary fact type, as shown in figure 2.7 (together with a sample population). In this fact type, label type *CR-code* is used to identify chemical reactions. The entity type *Substance* describes which substances are subject to the chemical reaction and the entity type *Quantity* describes in what quantity. The *Input/Output indicator* makes the distinction between input and output substances.

A first problem with this solution is the superfluous (artificial) identification of a chemical reaction. Only

some chemical reactions are sufficiently important to have a name of their own. The others are only identified by their input and output substances and associated quantities. A second problem is that this solution does not allow for the addition of a chemical reaction by one elementary update. This is caused by the fact that in the schema of figure 2.7 several fact type instances are needed to denote one reaction.

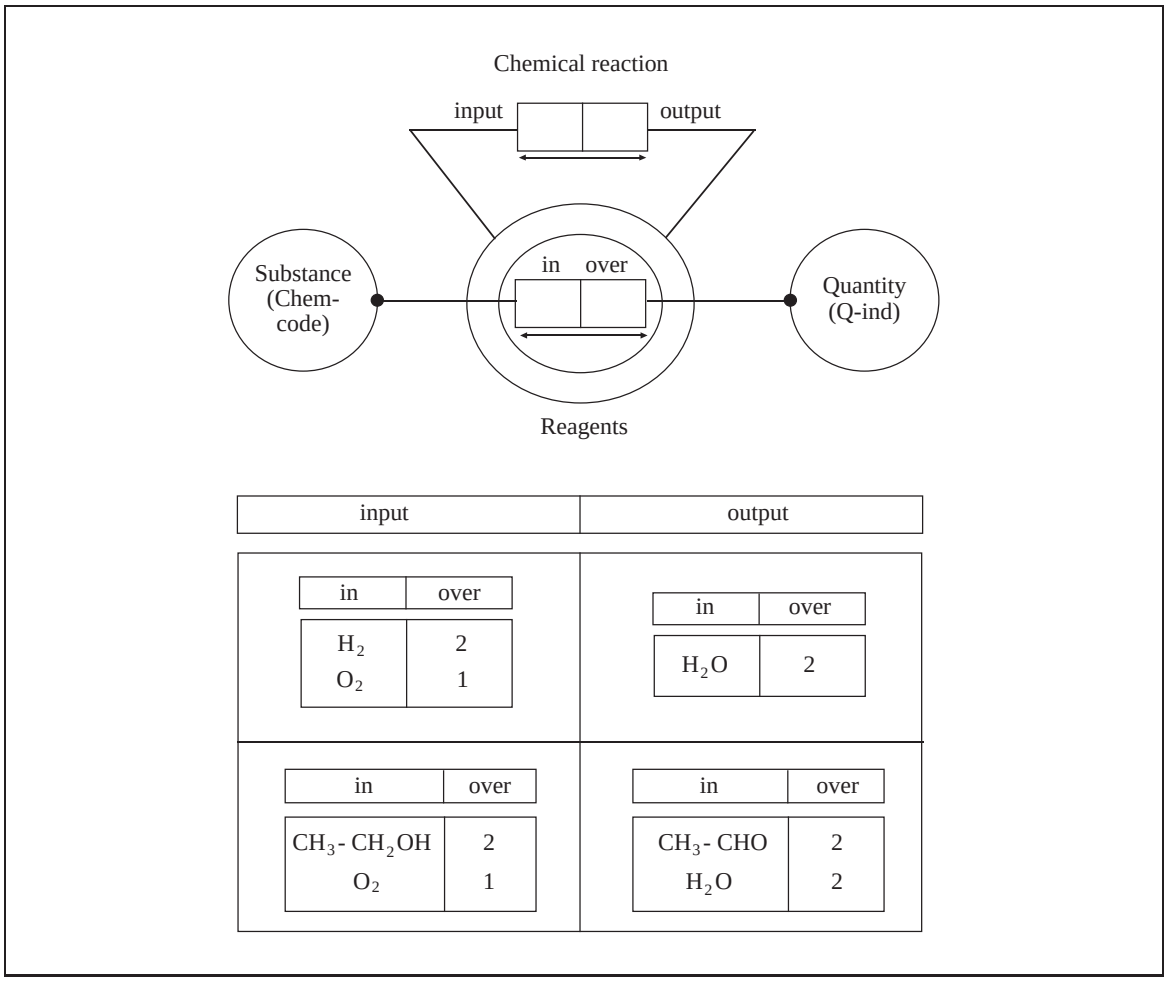

Figure 2.8: Chemical reactions with power type

The use of a power type makes it possible to model this Universe of Discourse more adequately. In the schema of figure 2.8, a chemical reaction is modelled as a relationship between a set of input *Reagents*, and a set of output *Reagents*. This schema is better understood by studying a sample population (e.g. the population given in figure 2.8).

As can be seen from the sample population, the solution of figure 2.8 solves the aforementioned update problem. In this schema, a chemical reaction corresponds to one fact type instance. The consequence is that an update operation of a chemical reaction can be considered as a single operation. Furthermore, neither a separate identification for chemical reactions nor an *Input/Output indicator* is needed.

About graphical conventions, it should be mentioned that in order to avoid confusion between fact objectification and power typing, a fact type occurring as an element type should be represented as an object type. This is the case for the fact type between *Substance* and *Quantity* in figure 2.8.

The element type of a power type is found by the function Elt. The relation between a power type x and its

element type $\mathsf{Elt}(x)$ is recorded in the fact type $\in_x = \{\in_x^p, \in_x^e\}$, where $\mathsf{Base}(\in_x^p) = x$ and $\mathsf{Base}(\in_x^e) = \mathsf{Elt}(x)$. This fact type is assumed to be available for each power type. Usually $\in_x$ is treated as an implicit fact type, and not drawn in the information structure diagram. If this fact type is subject to constraints it needs to be graphically represented.

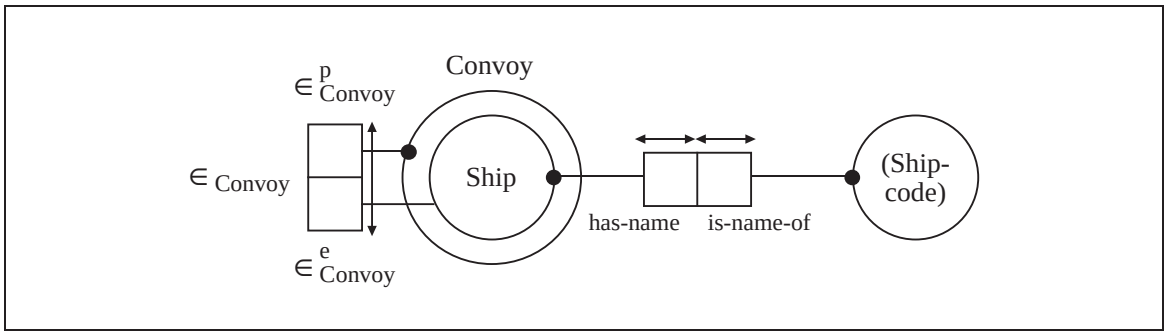

Figure 2.9: Implicit fact type for power type *Convoy*

Example 2.2
Figure 2.9 extends figure 2.5 with the implicit fact type associated to power type Convoy*.* □

Example 2.3
In figure 2.1 the implicit fact type associated to power type E was omitted for simplicity's sake. The precise definition should include fact type $\in_E = \{\in_E^p, \in_E^e\}$ in $\mathcal{F}$, with $\mathsf{Base}(\in_E^p) = E$ and $\mathsf{Base}(\in_E^e) = A$. □

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

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

2.1.5 Sequence types

Sequence types can be compared to power types. The differences are that, in the case of sequence types, the ordering of elements is important and elements may occur more than once. An instance of a sequence type is a sequence (tuple) of instances of its element type.

A simple example of a sequence type is shown in figure 2.10. In this schema, a *Train* is identified by a *T-code*, has a sequence of *Freight-cars* and is controlled by a *Locomotive*. The sequence type *Freight-car-sequence* is represented by a square around its element type *Freight-car*. Without explicit sequence types, the representation of sequences requires the introduction of an entity type that explicitly records the ordering. In figure 2.11 the model of 2.10 is represented in an alternative way. Note that in the schema of figure 2.11, constraints are necessary to ensure that *Positions* are consecutive and start with 1. These constraints are not necessary in schema 2.10.

As another, less trivial, example, consider function overloading in a language like C++ ([69]). In C++, a function name may represent several functions, each with different function definitions. However, functions with the same name must differ in number, type or ordering of their input parameters. The combination of an input parameter type sequence and a function name then uniquely identifies the function involved. From this identification also the output parameter type can be determined. This Universe of Discourse is modelled in figure 2.12. In this figure, the encircled u is an example of the graphical representation of a uniqueness constraint over several fact types. This constraint states that the combination of *Parameter-type-sequence* and *Function-name* uniquely determines a *Function*.

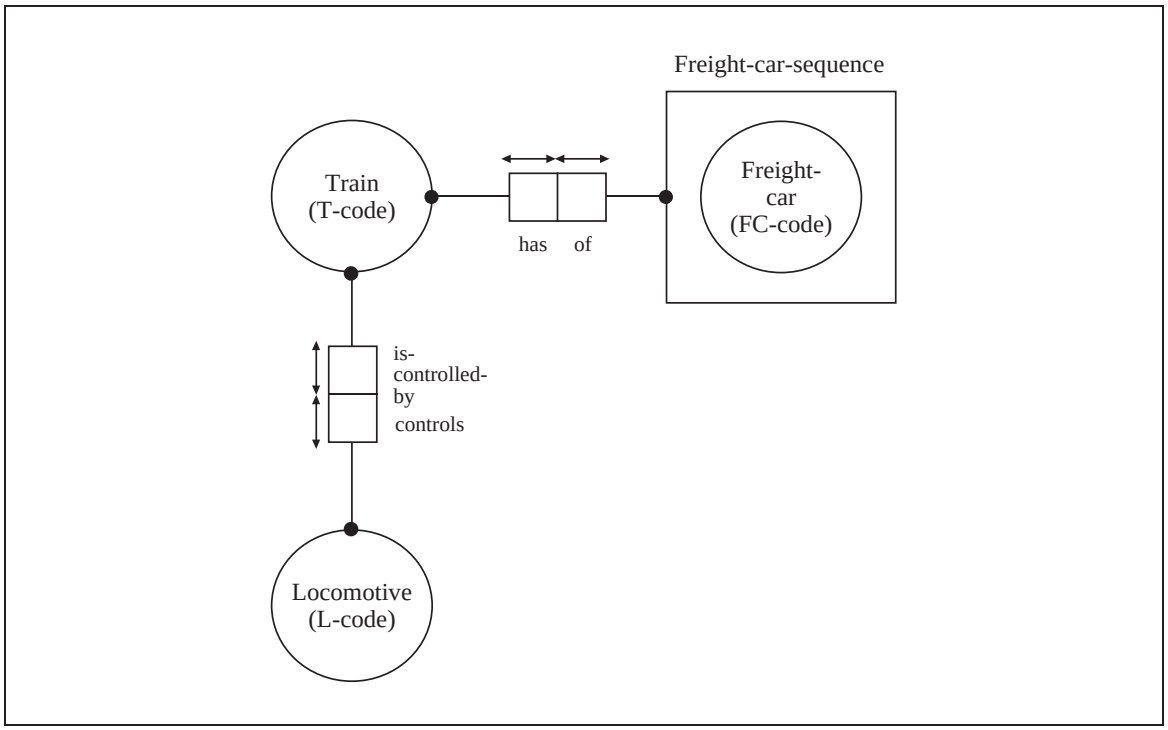

Figure 2.10: Train composition

A final example of the use of sequence types can be found in the meta-model of entity structure diagrams (see section 1.5.3), shown in figure 2.13. From this meta-model it follows that an *Action* can be decomposed into a sequence of other *Actions*, that it can be a repetition of another *Action* and that it can be a choice between a number of *Actions*. The exclusion constraint (represented by the crossed circle) ensures that for each *Action* at most one of these options is possible. Exclusion constraints are formally defined in section 4.5. Other, more complex, constraints are omitted in this meta-model.

The notion of sequence type is not elementary, as it is expressible in terms of generalisation and fact objectification. This is elaborated upon in section 2.1.8, which deals with generalisation. Despite the fact that the concept of sequence type is not elementary, it is treated as an independent concept, because this facilitates its use in database languages based on path expressions (see e.g. [39]).

The element type of a sequence type is also found by the function Elt. The relation between a sequence type x and its element type $\mathsf{Elt}(x)$ is recorded in the implicit fact type $\in_x = \{\in_x^s, \in_x^e\}$, where $\mathsf{Base}(\in_x^s) = x$ and $\mathsf{Base}(\in_x^e) = \mathsf{Elt}(x)$. Contrary to power types, this fact type $\in_x$ is augmented with the position of the element in the sequence, via the implicit fact type $@_x = \{@_x^s, @_x^i\}$, where $\mathsf{Base}(@_x^s) = \in_x$ and $\mathsf{Base}(@_x^i) = \mathrm{I}$. The object type I is the domain for indices in sequence types. To be more precise, I is a label type ($\mathrm{I} \in \mathcal{L}$) having as domain the set of natural numbers ($\mathsf{Dom}(\mathrm{I}) = \mathbb{N}$). This object type is only present in an information structure if this information structure contains sequence types.

Example 2.4

Figure 2.14 extends figure 2.10 with the implicit fact types associated with sequence types. In this figure Fcs *is a shorthand for Freight-car-sequence.* □

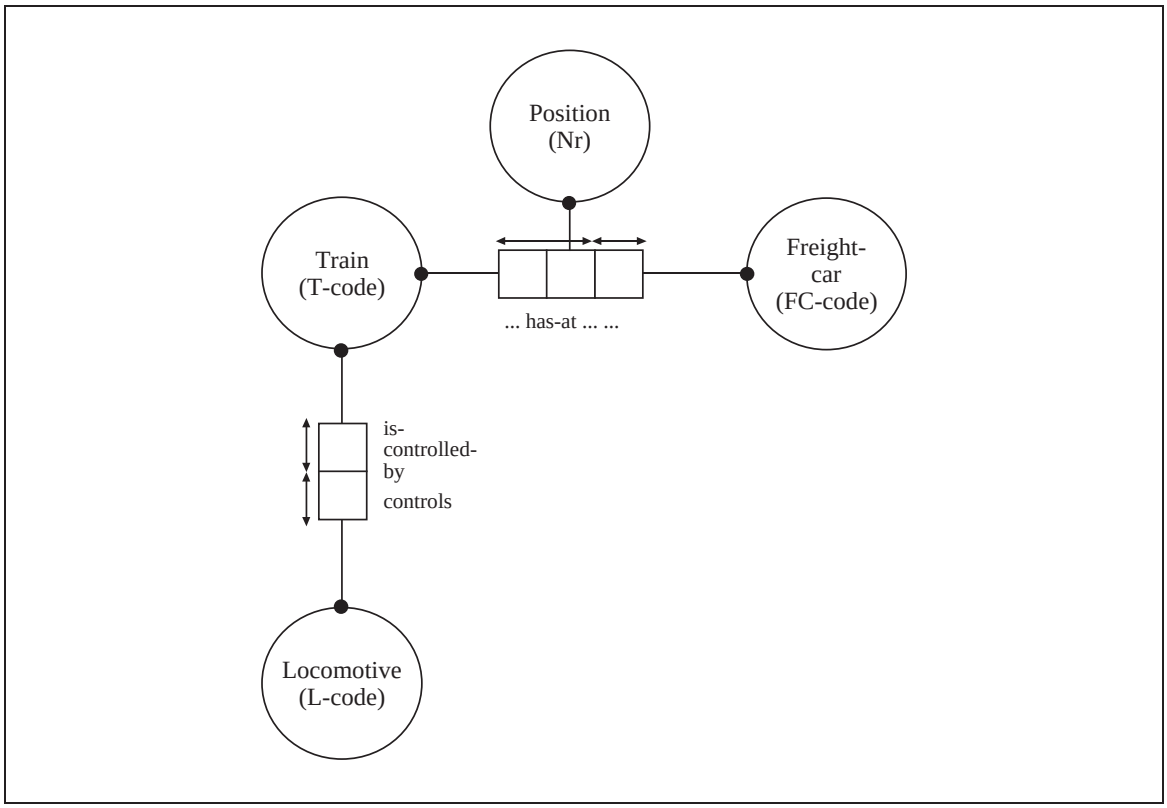

Figure 2.11: Train composition without explicit sequence type

2.1.6 Schema types

Schema objectification allows to define part of a schema as an object type. This object type is referred to as *schema type*. An instance of a schema type is an instantiation of the associated schema part. Consequently, this schema part should be a valid schema.

An example of the use of schema types can be found in the meta-model of Activity Graphs. Activity Graphs [62] are used for modelling processes and information passing between processes. Activity Graphs are bipartite directed graphs consisting of activities (processes) and states. States can be input for, or output of activities. In an Activity Graph, both activities and states may be subject to decomposition.

Two examples of Activity Graphs are shown in figure 2.15. In this figure, S_1 is an example of a state and A_1 an example of an activity. State S_1 is input for activity A_1 and activity A_1 has state S_2 as output. The Activity Graph on the right represents the decomposition of activity A_1.

The meta-model of Activity Graphs is shown in figure 2.16. In this figure, *Activity-graph* is a schema type. The decomposition relations are modelled by the binary fact types connecting *Activity-graph* and *Activity*, and *Activity-graph* and *State*. The total role constraints express that each *State* is input or output of an *Activity* and that each *Activity* has a *State* as input or output.

Schema objectification is not elementary. Figure 2.17 shows how a schema type can be modelled using fact objectification and power typing. The idea is to construct a power type for each object type that is to take part in the schema type. Each of these power types is the base of a predicator that is part of a fact type. This fact type is to relate sets of instances of the object types involved in the schema objectification, which are part of the same schema instance. As such, this fact type models the schema type. In figure 2.17

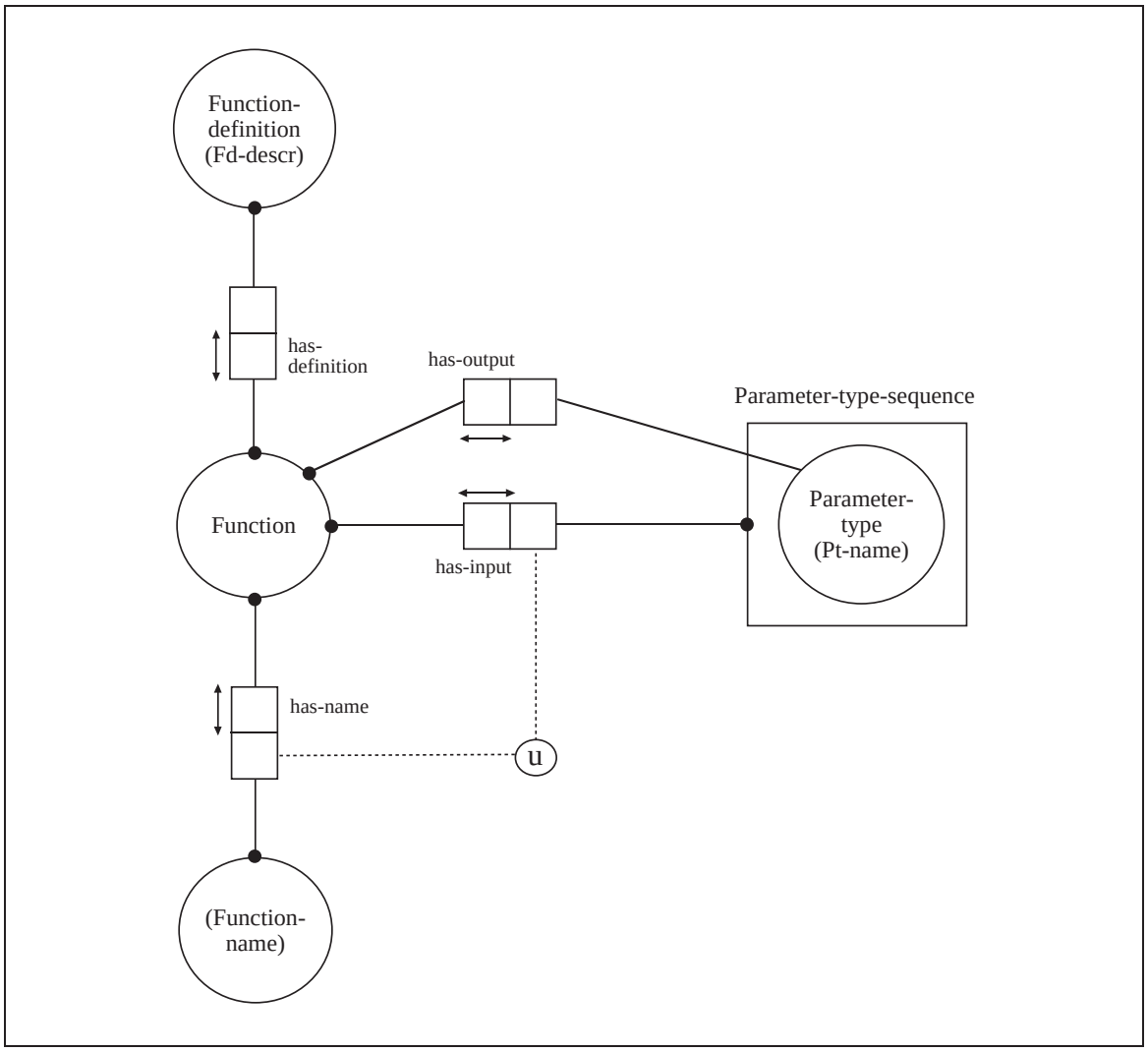

Figure 2.12: Function overloading in languages like C++

constraints that ensure that an instance of a schema type is a valid population of its associated schema, are omitted. Furthermore, it should be noted that this construction is only correct for schema types containing populations that assign nonempty sets of instances to each of the object types in their decomposition, as power types are not allowed to contain the empty set. We will not consider constructions for schema types that may contain partially empty populations.

Though schema objectification is not an elementary concept, it is considered an independent concept for the same reason as mentioned for sequence types in the previous section.

Schema types are decomposed into their 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".

The underlying information structure $\mathcal{I}_x$ of a schema type x is derived from the object types into which x is decomposed: $\mathcal{O}_x = \{y \in \mathcal{O} \mid x \prec y\}$. Analogously, the special object classes $\mathcal{L}_x$, $\mathcal{E}_x$, $\mathcal{F}_x$, $\mathcal{G}_x$, $\mathcal{S}_x$ and $\mathcal{C}_x$ can be derived. The functions Base_x, Elt_x, $\prec_x$, Spec_x and Gen_x are obtained by restriction to object types within $\mathcal{O}_x$. The following axiom ensures that a schema type is always decomposed into an information

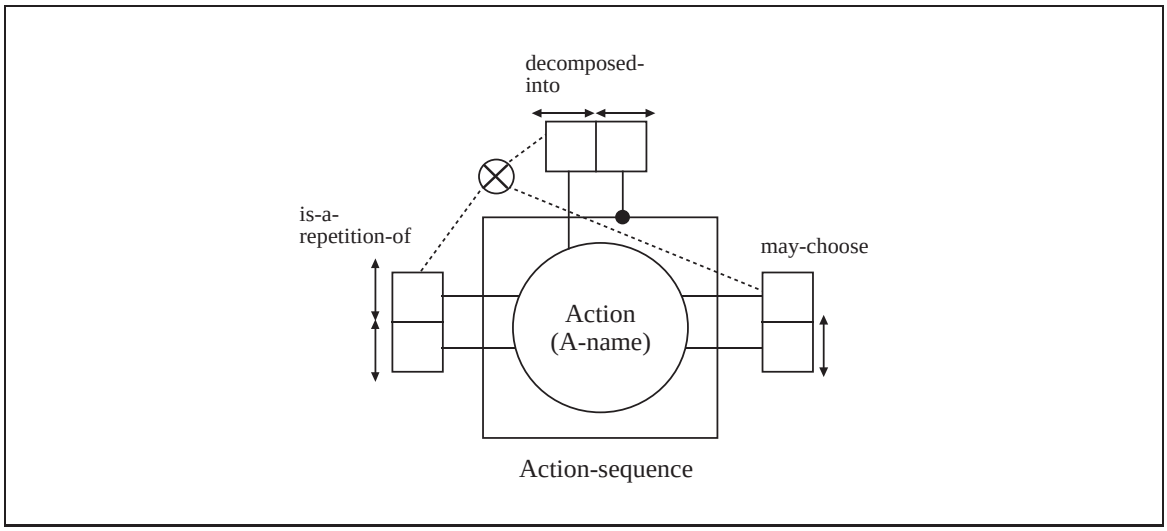

Figure 2.13: Meta-model of entity structure diagrams

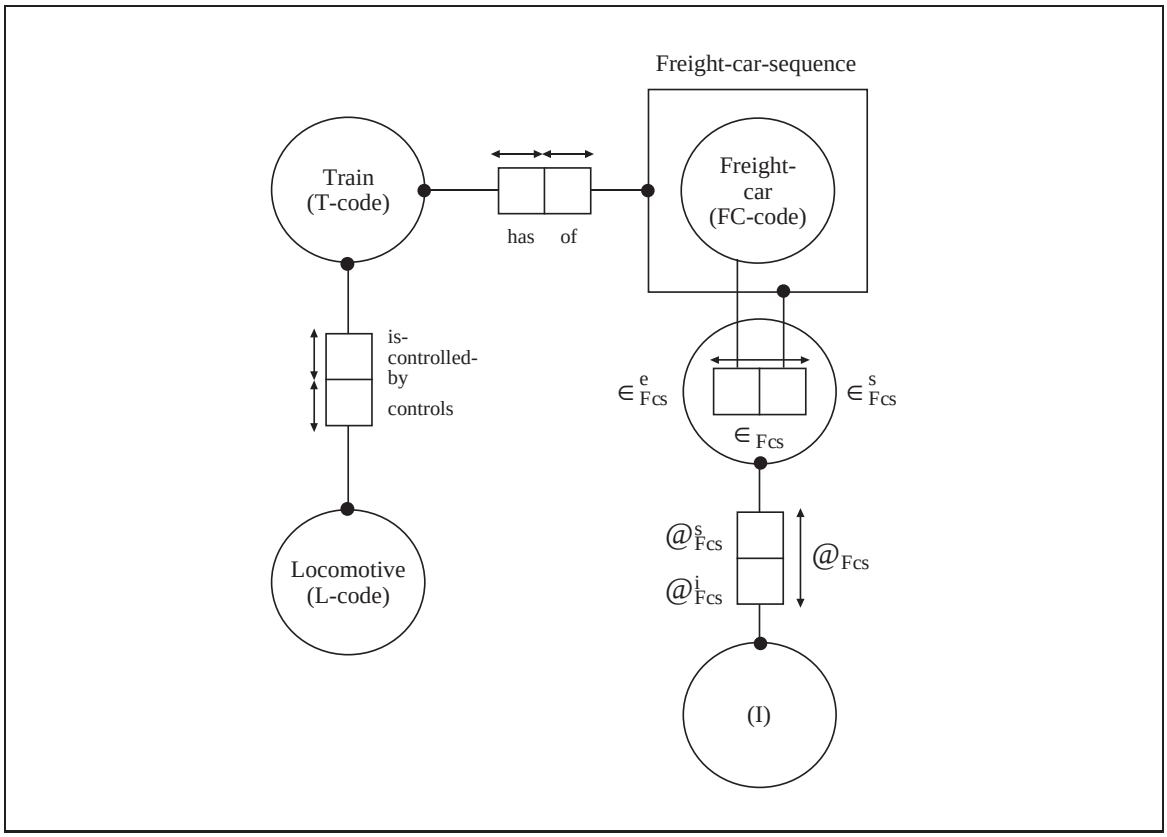

Figure 2.14: Implicit fact types for sequence type *Freight-car-sequence*

structure:

$$x \in \mathcal{C} \Rightarrow \mathcal{I}_x \text{ is an information structure}$$

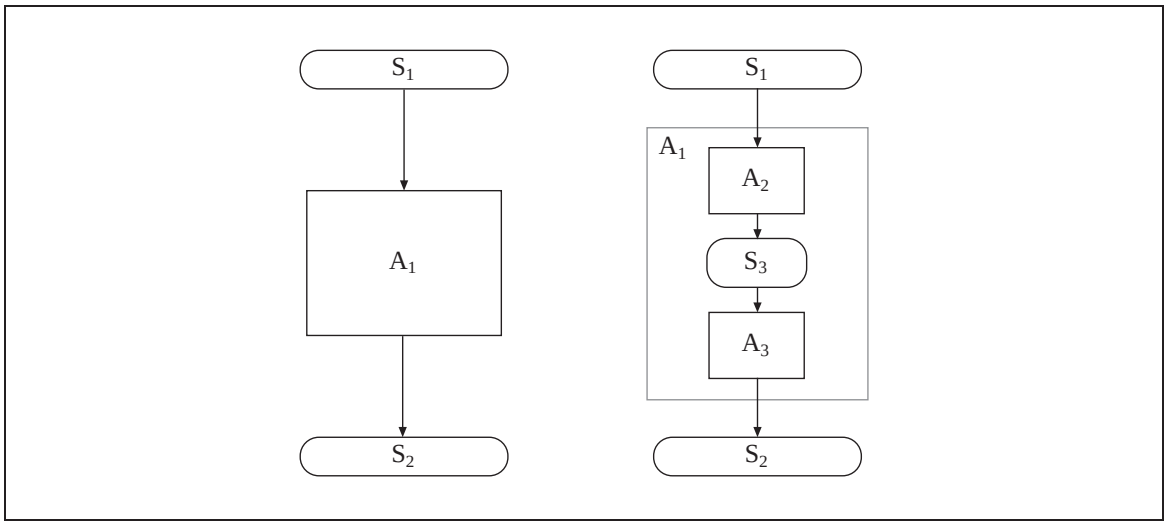

Figure 2.15: Sample Activity Graphs

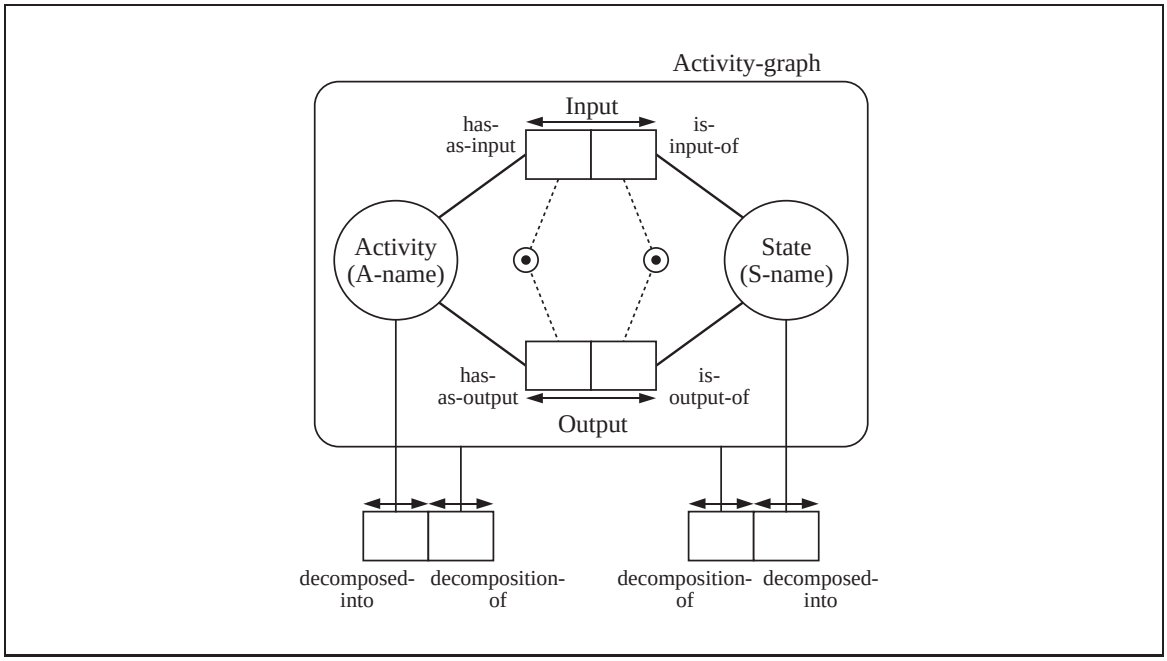

Figure 2.16: Meta-model of Activity Graphs

To each schema type x and each object type y in its decomposition, an implicit fact type $\in_{x,y} = \left\{\in^c_{x,y}, \in^d_{x,y}\right\}$ is associated, where $\mathsf{Base}(\in^c_{x,y}) = x$ and $\mathsf{Base}(\in^d_{x,y}) = y$.

2.1.7 Specialisation

Specialisation, also referred to as *subtyping*, is a mechanism for representing one or more (possibly overlapping) subtypes of an object type. Specialisation is to be applied when certain facts are to be recorded for specific instances of an object type only. Suppose for example that only for adults, i.e. persons with an

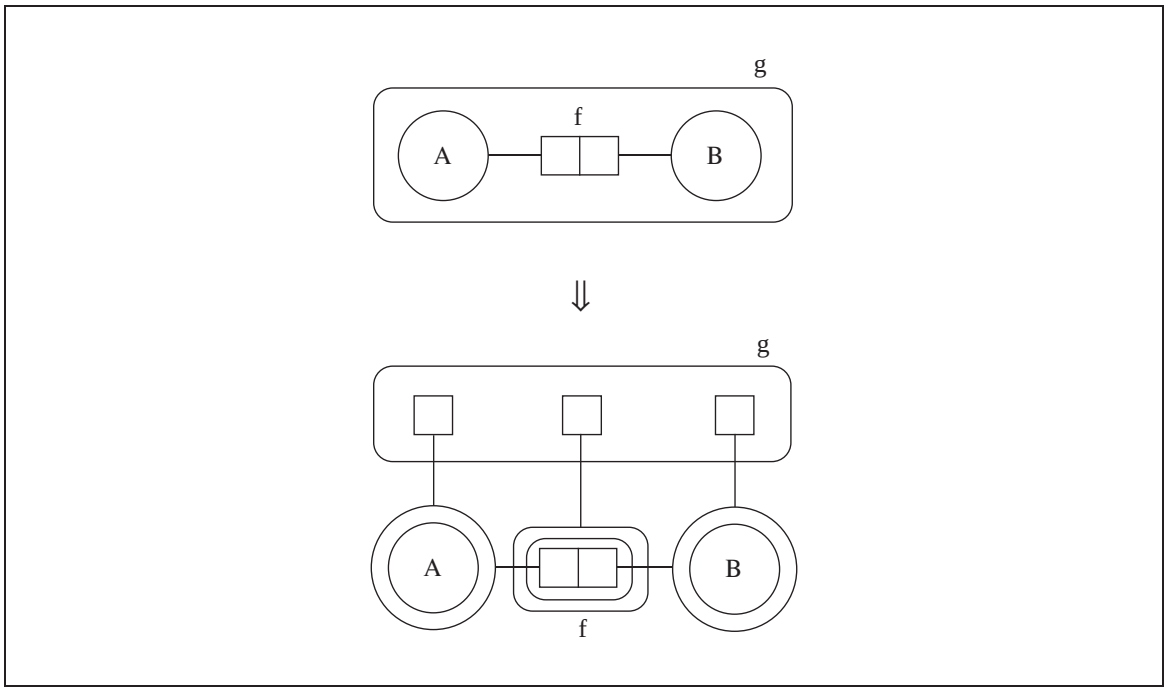

Figure 2.17: Modelling a schema type

age greater than or equal to 18, one is interested in the cars they own. This situation is captured by the schema in figure 2.18.

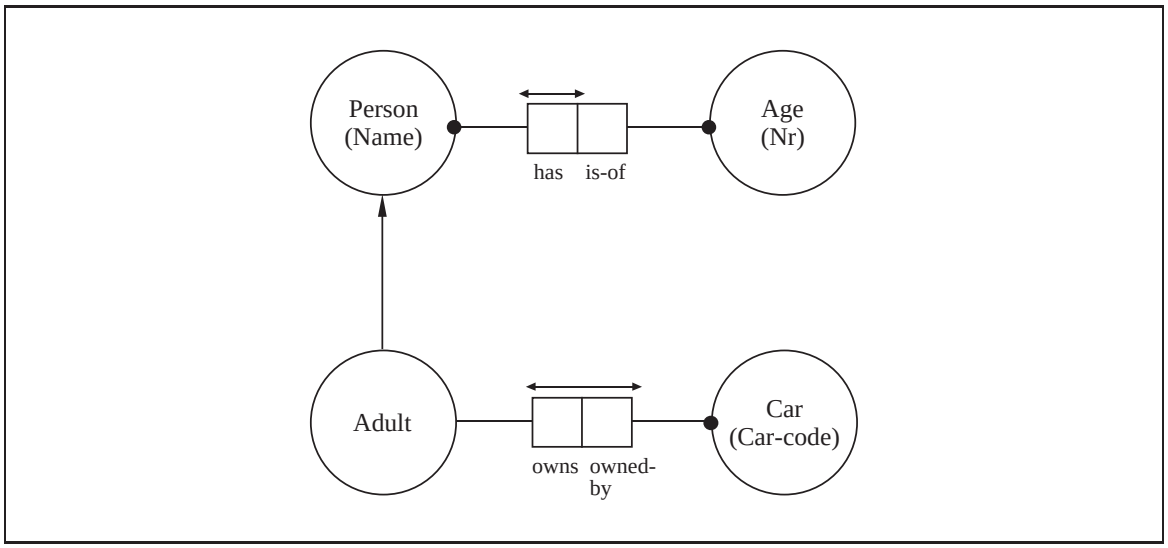

Figure 2.18: A simple example of specialisation

A specialisation relation between a subtype and a supertype implies that the instances of the subtype are also instances of the supertype (each *Adult* is a *Person*). For proper specialisation, it is required that subtypes are defined in terms of one or more of their supertypes. Such a decision criterion is referred to as

a *subtype defining rule*. In figure 2.18 the subtype defining rule for *Adult* is:

$$\mathsf{Adult} = \mathsf{Person\ has\ Age\ WITH\ Nr} \geq 18$$

The language used in the definition of this subtype defining rule is described in e.g. [39].

Identification of subtypes is derived from their supertypes. Therefore, as in the ongoing example *Persons* are identified by a name, *Adults* are also identified by that name.

Specialisation relations are organised in so-called specialisation "hierarchies". A specialisation hierarchy is in fact not a hierarchy in the strict sense, but an acyclic directed graph with a unique top. This top is referred to as the *pater familias* (see [19]). In the example of figure 2.18, the pater familias of *Adult* is *Person*.

Objects inherit all properties from their ancestors in the specialisation hierarchy. This characteristic of specialisation prevents non-entity types (e.g. fact types) from acting as a subtype. Consider for example the case that a ternary fact type is a subtype of a binary fact type. Clearly this leads to a contradiction. Problems do not occur when non-entity types themselves are specialised. Consequently, non-entity types always act as pater familias, i.e. each non-entity type is pater familias of itself and possibly of a collection of entity types.

As a more elaborate example of specialisation, consider a company where only for managers the telephone number is recorded, only for married employees the number of children they have, and only for married managers their life insurance. This Universe of Discourse can be modelled as in the diagram of figure 2.19 (adapted from [21]).

In this figure, the entity type *Employee* is the pater familias of the specialisation hierarchy. The entity type *Married-manager* is a subtype of both *Manager* and *Married-employee*. The subtype defining rules are:

Manager = Employee is-manager-of
Married-employee = Employee is-married
Married-manager = Employee (is-married AND-ALSO is-manager-of)

The concept of specialisation is defined as a binary relation Spec, with the convention that $a\,\mathsf{Spec}\,b$ is interpreted as "a is a subtype (specialisation) of b", or "b is a supertype of a". The strict separation between abstract and concrete object types prohibits label types to be specialised. Furthermore, as subtypes inherit the structure of their supertypes, only entity types can act as subtype (strictness):

$$\mathsf{Spec} \subseteq \mathcal{E} \times \mathcal{O} \setminus \mathcal{L}$$

Specialisation networks are acyclic:

$$a\,\mathsf{Spec}^{+}\,b \Rightarrow \neg b\,\mathsf{Spec}^{+}\,a$$

In this definition Spec^{+} represents the transitive closure of Spec, which can be defined in the usual way. The reflexive transitive closure of Spec (which is not standard due to the signature of Spec) is defined as follows:

$$\mathsf{Spec}^{*} = \mathsf{Spec}^{+} \cup \{\, \langle x, x\rangle \mid x \in \mathcal{O} \,\}$$

This relation is used in the definition of the pater familias relation $\sqcap$:

$$\sqcap(a, b) = a\,\mathsf{Spec}^{*}\,b \wedge \neg\,\mathsf{spec}(b)$$

where $\mathsf{spec}(b)$ is an abbreviation for $\exists_{x\in\mathcal{O}}\,[b\,\mathsf{Spec}\,x]$. In the rest of this section several properties of specialisation are given. The following lemma states that each object type has at least one pater familias.

Lemma 2.1 $\forall_{a\in\mathcal{O}}\exists_{b\in\mathcal{O}}\,[\sqcap(a, b)]$

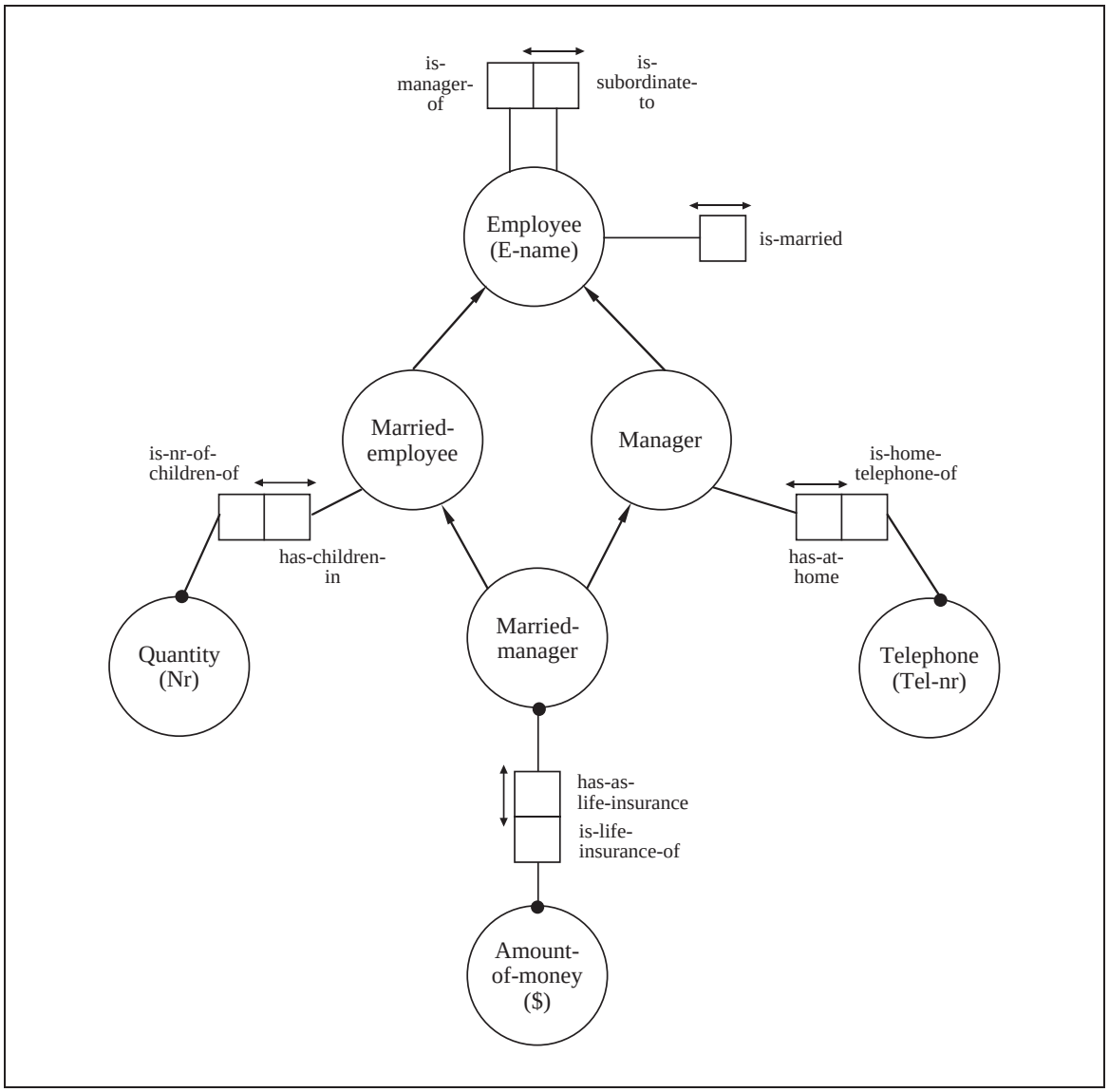

Figure 2.19: Example of a specialisation hierarchy

The following trivial lemma states that an object type is not a subtype if and only if it is its own pater familias.

Lemma 2.2 $\neg\,\mathsf{spec}(x) \iff \sqcap(x,x)$

A consequence of this lemma is that non-entity types have themselves as pater familias.

Corollary 2.1 $x \notin \mathcal{E} \Rightarrow \sqcap(x,x)$

While lemma 2.1 states that each object type has at *least* one pater familias, the following axiom ensures that an object type has at *most* one pater familias:

$$\sqcap(a,b) \wedge \sqcap(a,c) \Rightarrow b = c$$

Combining this axiom with lemma 2.1 yields that each object type has precisely one pater familias, i.e. $\forall_{a\in\mathcal{O}}\exists!_{b\in\mathcal{O}}\,[\sqcap(a,b)]$. For each a this unique b is denoted as $\sqcap(a)$. From the definition of $\sqcap$ it follows directly that $a\,\mathsf{Spec}^*\,\sqcap(a)$ and $\neg\,\mathsf{spec}(\sqcap(a))$.

Lemma 2.3 Idempotency of $\sqcap$: $\sqcap(\sqcap(a)) = \sqcap(a)$

A subtype has the same pater familias as its supertype(s).

Lemma 2.4 $a\,\mathsf{Spec}\,b \Rightarrow \sqcap(a) = \sqcap(b)$

2.1.8 Generalisation

Generalisation is a mechanism that allows for the creation of new object types by uniting existing object types. Contrary to what its name suggests, generalisation is *not* the inverse of specialisation. Specialisation and generalisation originate from different axioms in set theory (see section 5.8) and therefore have a different expressive power.

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

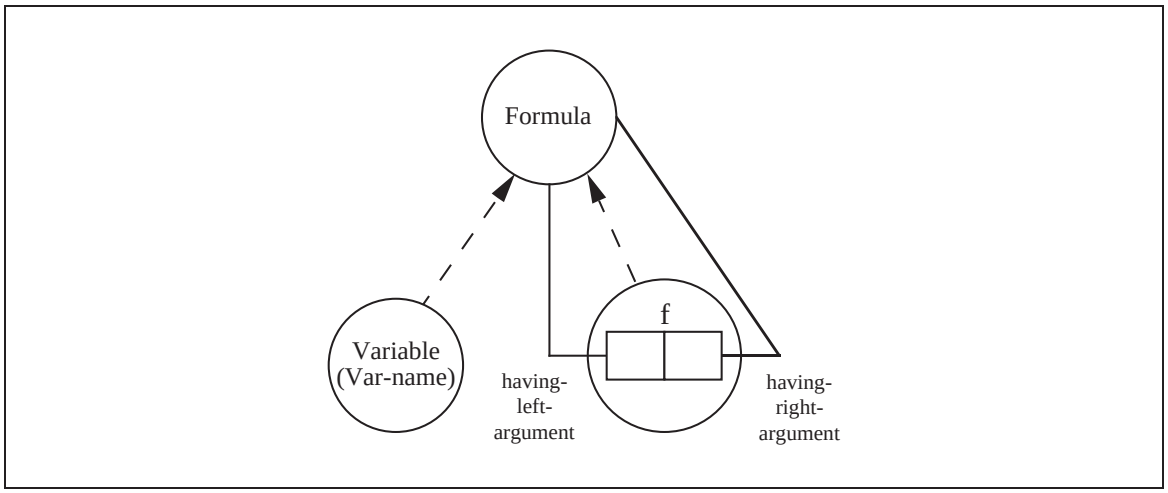

Figure 2.20: Example of generalisation

Figure 2.20 contains an example of generalisation, using dashed arrows (generalisation) rather than solid arrows (specialisation). A formula may be either a single variable, or constructed by some function (say f) from simpler formulas. From the schema of figure 2.20 it is clear that instances from the object type *Formula* inherit the structure from the specifier from which they originate (*Variable* or f).

The above example demonstrates that generalisation can be used to define recursive object types. This is not possible in the data model in [1], where object types are hierarchical structures. In the data model in [50], however, object types are directed graphs, which may contain cycles.

The notions specialisation and generalisation as presented here, can be compared to specialisation and generalisation as presented in [43]. The only differences are that in their approach no subtype defining rule

for specialisation is required and that they require specifiers to be disjunct. As the schema of figure 2.21 shows, specifiers of a generalised object type are not always disjunct. In this schema, the object types *Plant-eater* and *Flesh-eater* have the *Omnivores* in common. As a side remark, note that in this particular Universe of Discourse it is assumed that herbivores, omnivores and carnivores are elementary objects for which an identification is known (*H-id*, *O-id* and *C-id* resp.). Therefore, generalisation is used instead of specialisation.

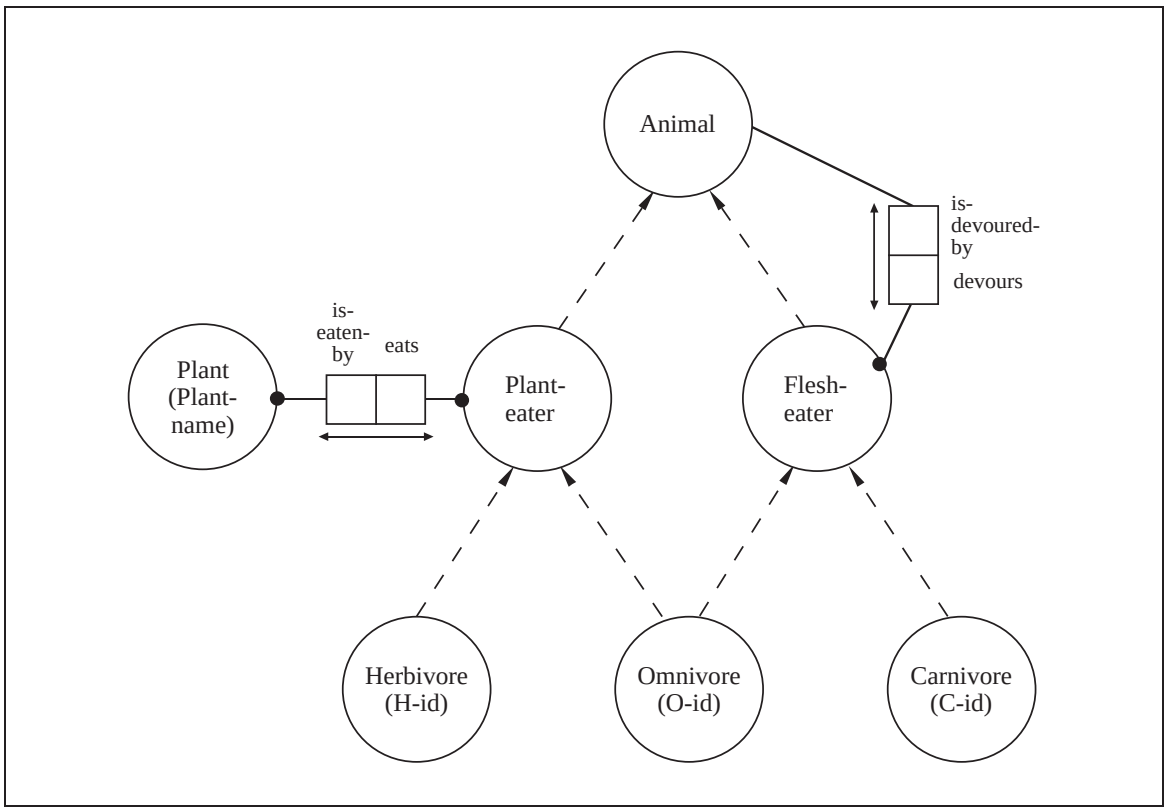

Figure 2.21: Another example of generalisation

As stated in section 2.1.5, sequence types are not elementary and can be modelled using generalisation and fact objectification. This is demonstrated in figure 2.22. In this figure, the unary fact type *One-element-sequence* captures the sequences consisting of only one element. The binary fact type *Extend* models the fact that a sequence extended at the end with an element, is again a sequence.

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

$$\mathsf{Gen} \subseteq \mathcal{E} \times \mathcal{O} \setminus \mathcal{L}$$

Generalisation networks are acyclic:

$$a\,\mathsf{Gen}^+\,b \Rightarrow \neg b\,\mathsf{Gen}^+\,a$$

In the remainder, $\mathsf{gen}(a)$ is used as an abbreviation for $\exists_{x\in\mathcal{O}}\,[a\,\mathsf{Gen}\,x]$.

Generalisation and specialisation can be conflicting due to their inheritance structure. Consider for example figure 2.23. In this figure, A is a specialisation of C as well as a generalisation of B. The specialisation

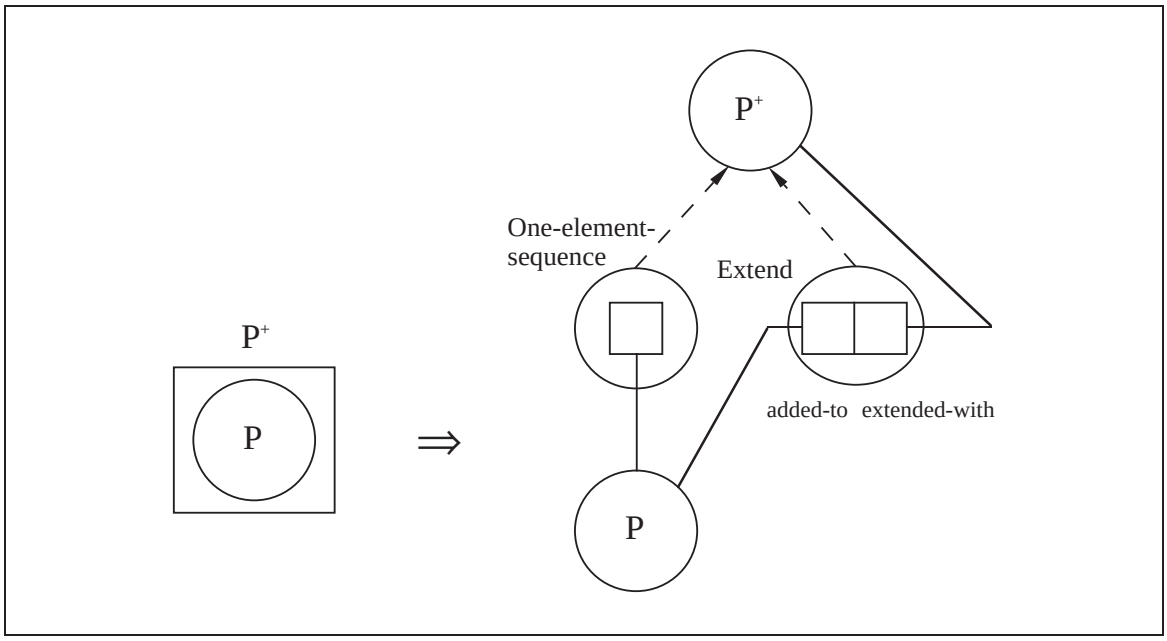

Figure 2.22: Modelling a sequence type

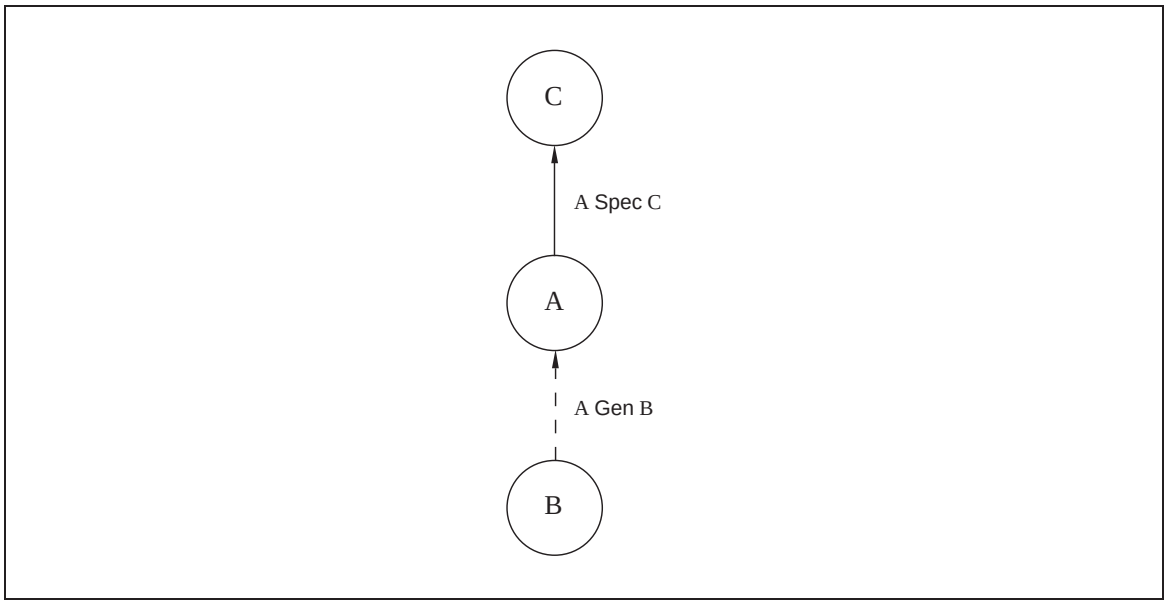

Figure 2.23: Conflicting generalisation and specialisation

relation requires the identification of A to depend on the identification of C, while the generalisation relation requires the identification of A to depend on the identification of B. In terms of populations this contradiction can be formulated in a different way. In this case, specialisation requires that, in every population, the instances of object type A are those instances of object type C that satisfy the subtype defining rule, while generalisation requires the instances of A to be exactly the instances of object type B.

To avoid such conflicts, generalised object types cannot be subtypes:

$$\mathsf{gen}(a) \Rightarrow \neg\, \mathsf{spec}(a)$$

2.1.9 Specialisation versus generalisation

As stated before, generalisation and specialisation are quite different notions. Some situations can only be solved using specialisation (e.g. when complex subtype defining rules are involved) and some situations can only be solved using generalisation (e.g. when recursive structures are involved, as in the formula example in figure 2.20). Sometimes, generalisation is modelled as specialisation. This leads to violations of the Conceptualisation Principle as will be discussed in the following example.

Consider a price list for individually priced products. A product is either a car or a house. A car is identified by a registration number, while a house is identified by the combination of its postal code and house number. For each product the price should be recorded. This Universe of Discourse could be modelled as the schema in figure 2.24.

We will argue that this schema suffers from overspecification. First, a special label type, *P-code*, has to be introduced in order to identify *Product*. This is necessary since specialisation requires the subtypes *Car* and *House* to inherit their identification from their supertype *Product*. Secondly, a partition constraint between the subtypes *Car* and *House* is needed to express that these subtypes cannot have instances in common and that the union of their instantiations is the instantiation of the object type *Product*. Thirdly, a special fact type and a special object type, *Product Type*, are required for the formulation of the subtype defining rules of *Car* and *House* (see figure 2.24). However, these extra object types are not relevant from a conceptual point of view. Their introduction should therefore be considered as a violation of the Conceptualisation Principle.

Figure 2.25 shows a more appropriate schema for this Universe of Discourse. In this schema, the label type *P-code* is no longer needed, since *Product* inherits its identification from *Car* and *House*. Furthermore, the partition constraint follows directly from the fact that specifiers of a generalised object type are disjunct, unless explicitly derivable otherwise (as in figure 2.21), and the fact that a generalised object type is covered by its specifiers. Finally, it should be remarked that as cars can be identified by their registration number, the name of the label type *Reg-nr* may be put below the name of the entity type *Car*. This convention is only allowed for identifying label types (see also section 5.2).

2.2 Data model populations

2.2.1 Let's enter the area of semantics

A data model consists of an information structure and a set of integrity constraints. In the *syntax* of a data model, we deal with the question whether the model satisfies the basic rules. For example, a role in a fact type is connected to exactly one object type. If we define a role connected to two object types, we would certainly produce incorrect syntax.

Syntax deals with basic structural properties of the information structure and of the constraints. Syntax does not deal with meaning. Meaning is the area of *semantics*. The semantics of a data model is usually defined in terms of the set of possible populations. A data model specifies all kinds of object types, whereas a population of a data model specifies the corresponding objects, also called data values. As a very simple example, a data model may contain an object type *person* and a population may contain the objects *john* and *mary*. Some people prefer the term instantiation rather than population.

2.2.2 State space

So we will define populations! But populations of what exactly? In our approach, a population Pop belongs to an information structure of a data model. Recall that an information structure consists of the following

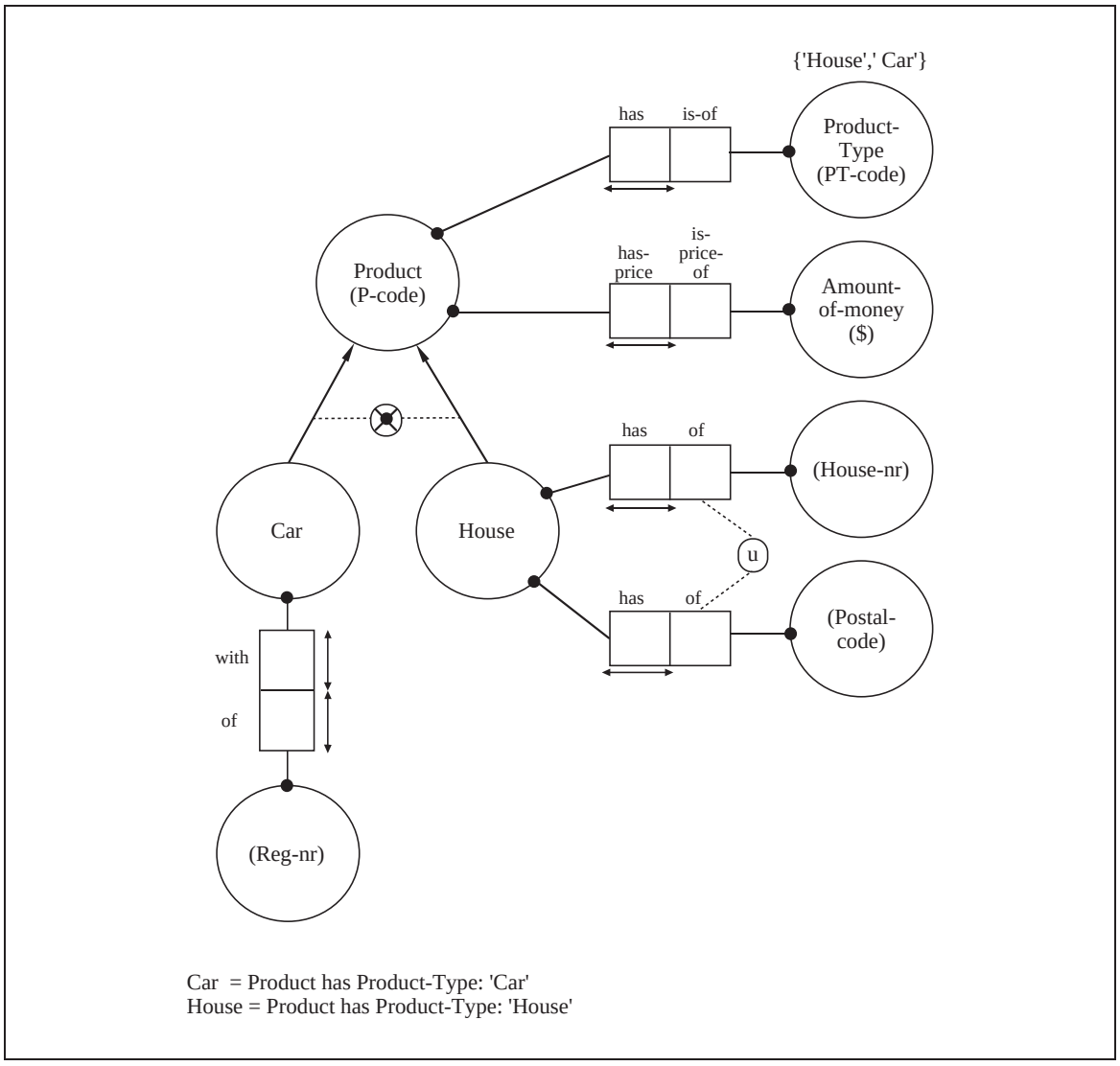

Figure 2.24: Specialisation (with overspecification) instead of generalisation

components:

$$\mathcal{I} = \langle \mathcal{P}, \mathcal{O}, \mathcal{L}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{S}, \mathcal{C}, \mathsf{Base}, \mathsf{Elt}, \prec, \mathsf{Spec}, \mathsf{Gen}, \mathcal{D}, \mathsf{Dom} \rangle$$

Wooow, does this give you a headache? Then wait a minute. Otherwise, go a step further and define a population Pop as an assignment, which assigns a set of instances to each object type in $\mathcal{O}$. The expression $\mathsf{IsPop}(\mathcal{I}, \mathsf{Pop})$ denotes that Pop is a population of the information structure $\mathcal{I}$. In that case Pop is a mapping from object types to sets of instances:

$$\mathsf{Pop} : \mathcal{O} \to \wp(\Omega)$$

Here Ω is the universe of instances that can occur in populations of information structures and $\wp(\Omega)$ denotes

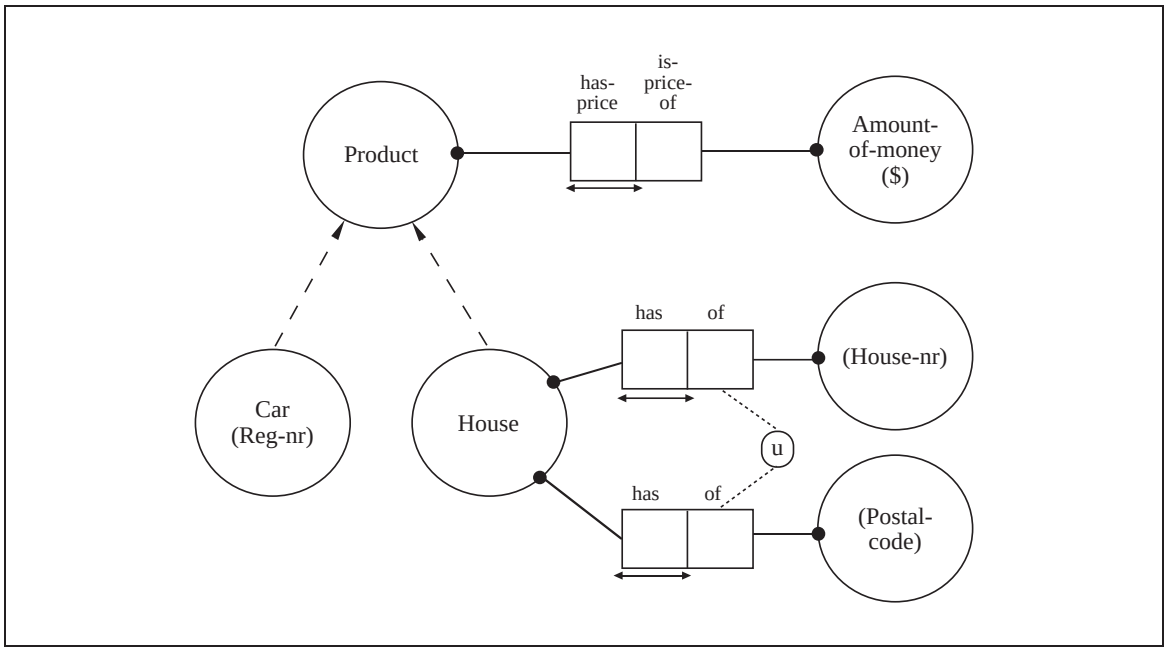

Figure 2.25: Example of generalisation (avoiding overspecification)

the powerset of that universe. So if $x \in \mathcal{O}$ is an object type, then $\mathsf{Pop}(x)$ is a subset of Ω. The state space of an information structure $\mathcal{I}$ is the set of all populations of that structure, defined as:

$$\mathsf{POP}_{\mathcal{I}} = \left\{ \mathsf{Pop} : \mathcal{O} \to \wp(\Omega) \;\middle|\; \mathsf{IsPop}(\mathcal{I}, \mathsf{Pop}) \right\}$$

The state space $\mathsf{POP}_{\mathcal{I}}$ contains populations and we hope that these popualtions correspond with states of the real world.

2.2.3 The universe of instances

Instances of a label type come from the domain associated to that label type via the domain function $\mathsf{Dom} : \mathcal{L} \to D$. We may for example specify that the domain of label type *name* is a string of characters. Instances of entity types are abstract values, which do not come from a concrete domain. Instead, entities come from a special domain Θ, which is a set of unstructured values.

We now define the universe of instances. Instances from this universe are the only kind of instances that may occur in the population of any information structure. In this definition $\mathcal{P}'$ is the set of all predicators that may occur in information structures and $\mathcal{O}'$ is the set of all object types that may occur in information structures. These sets are necessary to make the definition of Ω independent of specific information structures.

Although the following may look a bit complex, don't worry. It will all become clear. If not, don't hesitate to ask your neighbour or your teacher! Here we go. The universe of instances Ω is inductively defined as the smallest set satisfying the following conditions:

1. Elements of the concrete domains are contained in the universe of instances:

$$\bigcup D \subseteq \Omega$$

So the universe of instances contains things such as numbers and strings of characters.

2. Elements of the abstract domain Θ are contained in the universe of instances:

$$\Theta \subseteq \Omega$$

3. The universe of instances contains mappings:

$$x_1, \ldots, x_n \in \Omega \ \wedge p_1, \ldots, p_n \in \mathcal{P}' \ \Rightarrow \ \{p_1 : x_1, \ldots p_n : x_n\} \in \Omega$$

Here, the set $\{p_1 : x_1, \ldots, p_n : x_n\}$ denotes a mapping, assigning x_i to each predicator p_i. These mappings are intended for the population of fact types.

4. The universe of instances contains sets:

$$x_1, \ldots, x_n \in \Omega \ \Rightarrow \ \{x_1, \ldots, x_n\} \in \Omega$$

Sets of instances may occur as instances of power types.

5. The universe of instances contains sequences:

$$x_1, \ldots, x_n \in \Omega \ \Rightarrow \ \langle x_1, \ldots, x_n \rangle \in \Omega$$

Sequences of instances may occur as instances of sequence types.

6. The universe of instances contains assignments:

$$X_1, \ldots, X_n \subseteq \Omega \ \wedge O_1, \ldots, O_n \in \mathcal{O}' \ \Rightarrow \ \{O_1 : X_1, \ldots, O_n : X_n\} \in \Omega$$

Assignments of sets of instances to object types are also valid instances. They are intended for the populations of schema types.

Now you know what the universe of instances Ω is. So now you know the objects that may occur in populations of object types. Let's consider an example. A sample population of the information structure of figure 2.1 is the following:

$$\begin{array}{lcllcl}
\mathsf{Pop}(A) & = & \{a_1, a_2\} & \mathsf{Pop}(f) & = & \{\{p : b_1, q : a_1\}, \{p : b_1, q : a_2\}\} \\
\mathsf{Pop}(B) & = & \{b_1\} & \mathsf{Pop}(g) & = & \{\{r : \{p : b_1, q : a_1\}, s : b_1, t : g_1\}\} \\
\mathsf{Pop}(C) & = & \{b_1, g_1\} & \mathsf{Pop}(h) & = & \{\{u : \{a_1\}, v : b_1\}, \{u : \{a_1, a_2\}, v : g_1\}\} \\
\mathsf{Pop}(D) & = & \{b_1\} & \mathsf{Pop}(i) & = & \{\{w : b_1, x : 17\}\} \\
\mathsf{Pop}(F) & = & \{17\} & \mathsf{Pop}(E) & = & \{\{a_1\}, \{a_1, a_2\}\} \\
\mathsf{Pop}(G) & = & \{g_1\} & & &
\end{array}$$

In the above population 17, which is the only label, comes from the set of natural numbers $\mathbb{N}$. Recall that $\mathsf{Dom}(F) = \mathbb{N}$. The instances a_1, a_2, b_1 and g_1 come from the abstract domain Θ and are considered to be non-denotable by a user. The population of the implicit fact type $\in_E$ can be derived as follows:

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

Note that if the instance $\{w : c_1, x : 17\}$ is added to the population of fact type i we will have severe problems, because c_1 is not an element of $\mathsf{Pop}(C)$. In order to avoid such problems, we will introduce the Conformity Rule in section 2.3.

2.3 Population rules

2.3.1 Basic population rules

The first population rule is the Strong Typing Rule, which expresses that instantiations of non-label types can only have instances in common, if they are type related:

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

The notion of type relatedness is further elaborated in section 2.4. The population of a label type is a set of values, taken from its corresponding concrete domain:

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

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

$$\mathsf{Root}(x) \quad \Rightarrow \quad \mathsf{Pop}(x) \subseteq \Theta$$

The population of a fact type is a set of tuples. A tuple t in the population of a fact type f is a mapping of all its predicators to values. The value assigned to a predicator should occur in the population of the base of that predicator. These requirements are captured by the Conformity Rule:

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

Next we will consider several more advanced population rules.

2.3.2 Population rules for specialisation and generalisation

Respecting the specialisation hierarchy is captured by the Specialisation Rule:

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

Respecting the generalisation hierarchy is captured by the the Generalisation Rule:

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

The Generalisation Rule, which clearly is a derivation rule, requires that the population of a generalised object type is the union of the populations of its specifiers.

2.3.3 Population rules for power types and sequence types

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

The implicit fact type $\in_x$ that is provided for each power type x, relates sets in the population of power type x, to their elements in the population of element type $\mathsf{Elt}(x)$. This is called the Power Base Rule. The Power Base Rule is a *derivation rule* for the population of fact type $\in_x$.

The population of a sequence type consists of nonempty sequences of instances of the corresponding element type. This is called the Sequence Type Rule. The population of the index type I is the set of possible indices. This set can be derived from the sequences occurring in the populations of sequence types using the Active Index Rule.

The populations of the implicit fact types $\in_x$ and $@_x$, provided for each sequence type x, are given by Sequence Decomposition Rules. These rules act as derivation rules for $\in_x$ and $@_x$.

2.3.4 Population rules for schema types

The population of a schema type consists of populations of the underlying information structure. This is called the Decomposition Rule. According to this rule we will see situations in which $\mathsf{Pop}_1 \in \mathsf{Pop}(x)$ for some schema type x. Here Pop_1 is a population of the information structure in schema type x and thus Pop_1 is not the same as Pop.

The relations between instances occurring in the population of a schema type and instances occurring in the population of its constituing object types are recorded in the implicit fact types $\in_{c,d}$. Their population is prescribed by the Decompositor Rule, which is a derivation rule. Instances occurring in an instance of a schema type, which is, as stated before, a population, are also instances of an object type part of the decomposition of that schema type.

2.4 Type relatedness

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

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

[T1] $\vdash x \sim x$

[T2] $x \sim y \vdash y \sim x$

[T3] $\sqcap(x) = \sqcap(y) \wedge y \sim z \vdash x \sim z$

[T4] $x\,\mathsf{Gen}\,y \wedge y \sim z \vdash x \sim z$

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

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

[T7] $\mathcal{O}_x = \mathcal{O}_y \vdash x \sim y$

Two predicators, p and q, are called type related if their bases are type related: $\mathsf{Base}(p) \sim \mathsf{Base}(q)$. Now let us discuss an example. Consider the information structure with complex power types in figure 2.26. This structure has an interesting type relatedness. In this structure, all object types are type related, except C and B, C and E, and C and F. So $C \not\sim B$ and $C \not\sim E$ and $C \not\sim F$.

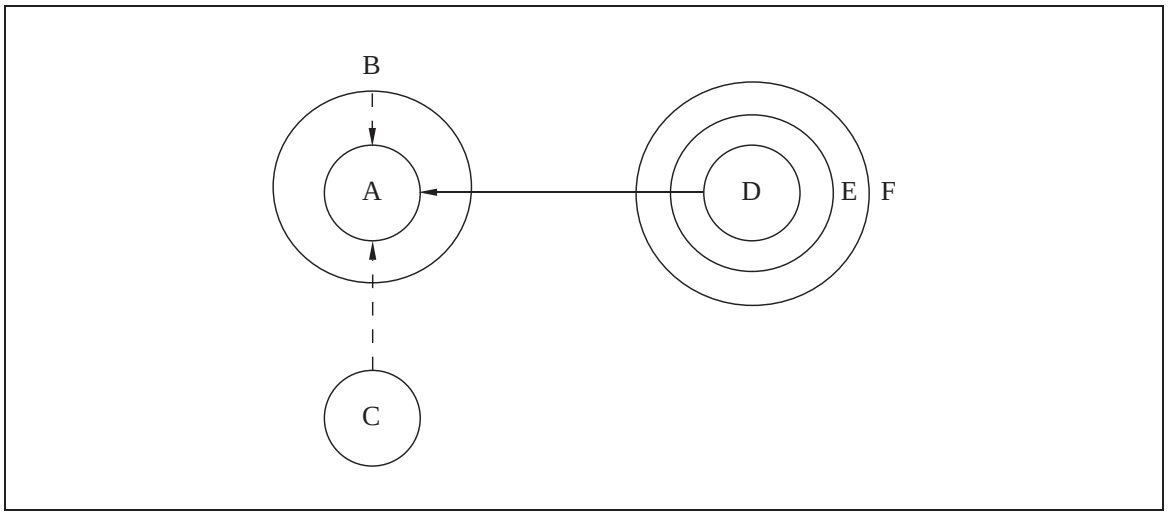

Figure 2.26: Information structure with complex power types

Chapter 3

Two examples of meta-models

3.1 Meta-model of entity-relationship diagrams

According to [80], an entity-relationship diagram is a network model that describes the stored data layout of a system at a high level of abstraction.

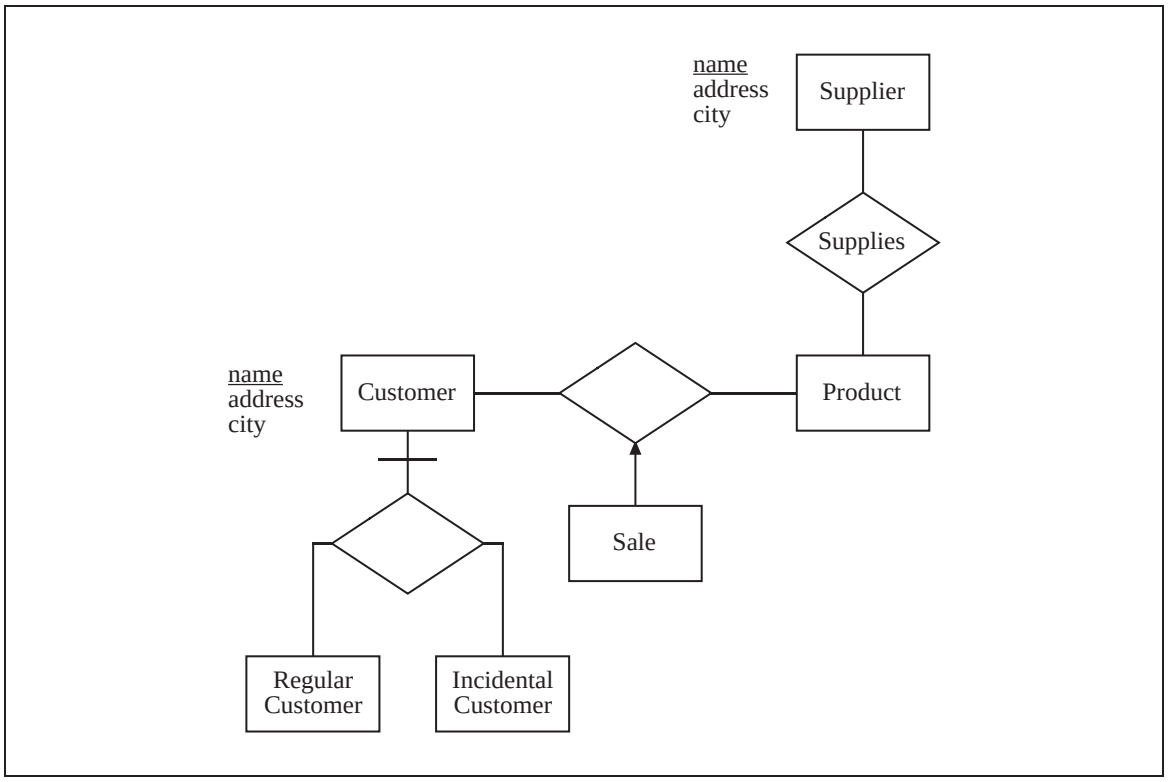

Figure 3.1: Example of an entity-relationship diagram

In figure 3.1 an example of a diagram according to [80] is shown. In this diagram, *Product*, *Sale*, *Supplier* and *Customer* are object types. The object type *Customer* has *Regular Customer* and *Incidental Customer* as subtypes. The object type *Sale* is a so-called associative object type indicator of the relationship between

Customer and *Product*. In this example, only the object types *Customer* and *Supplier* have attributes: *name*, *address* and *city*. In both cases, the data element *name*, which is underlined, identifies the object type involved.

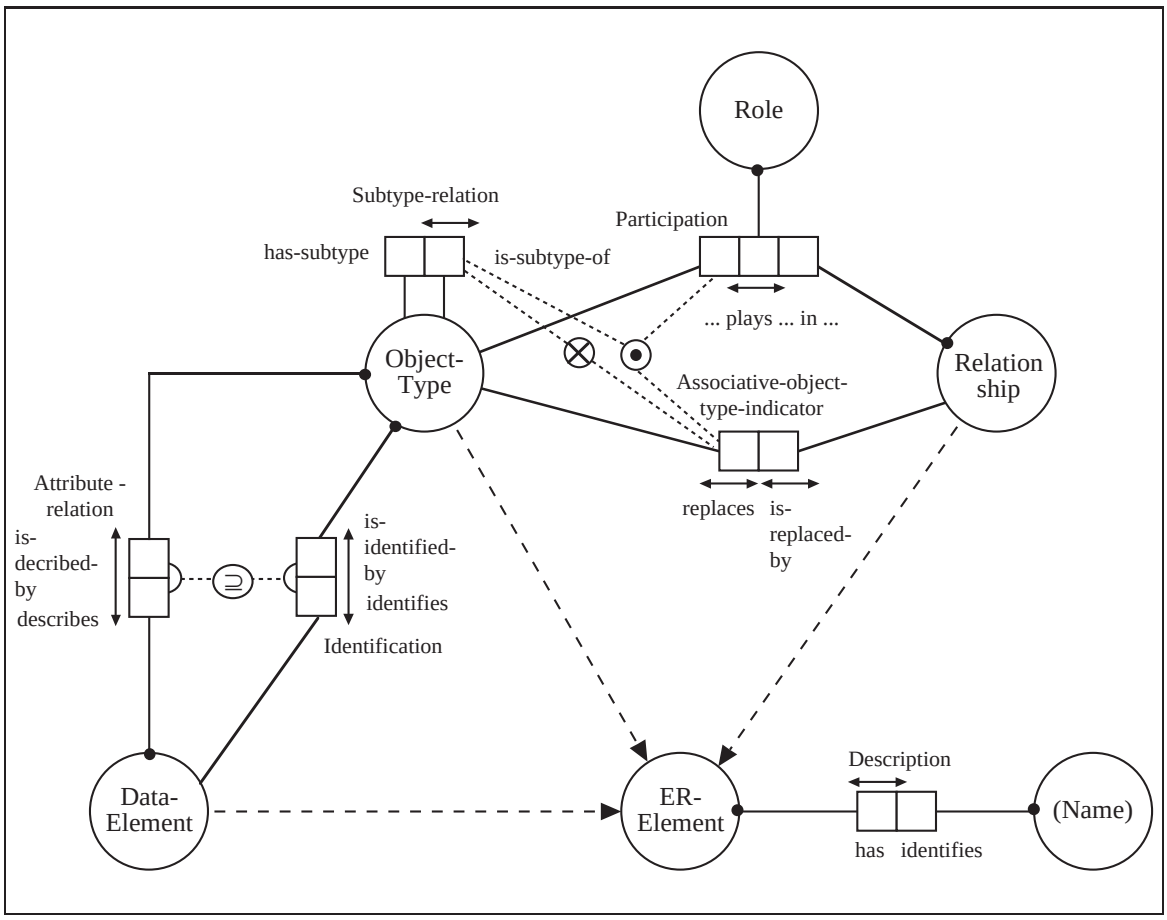

Figure 3.2: Meta-model of entity-relationship diagram

Figure 3.2 contains the meta-model. In this meta-model *Object-Type*, *Relationship* and *Data-Element* are generalised into *ER-Element*, because they all have a *Name*. We first consider *Object-Types*[1]. The fact type *Participation* captures the participation of *Object-Types* in *Relationships* via *Roles*. The fact type *Associative-object-type-indicator* captures the *Relationships* that are treated as *Object-Types*. *Object-Types* can have several subtypes, but can be subtype of at most one other *Object-Type*.

Next we consider some integrity constraints. The exclusion constraint between the roles with role names *is-subtype-of* and *replaces* expresses that an *Object-Type* cannot be both a subtype and an associative object type indicator. The total role constraint on three roles ensures that an *Object-Type* plays at least one of these three roles, i.e. is a subtype, an associative object type indicator or participates in a relationship.

An *Object-Type* is described by a number of *Data-Elements*. A subset of these *Data-Elements* serves as the identification of that *Object-Type*. This is ensured by the subset constraint.

[1]More precisely: we first consider the object type *Object-Type*.

3.2 Meta-model of data flow diagrams

According to [80], a data flow diagram pictures a system as a network of functional processes. The main components of a data flow diagram are processes, flows, data stores and terminators. An overview of the main components and their graphical representations can be found in figure 3.3.

A process transforms input into output. Processes have a process specification or are decomposed into another diagram. Each process has a number. Control processes are a special kind of process. A control process does not process data, but coordinates other processes. The operation of a control process is modelled by means of a state transition diagram. Terminators represent external processes communicating with the system under consideration. Data stores model collections of data "at rest". Data stores may be external, which means that they are used for communication with the outside world.

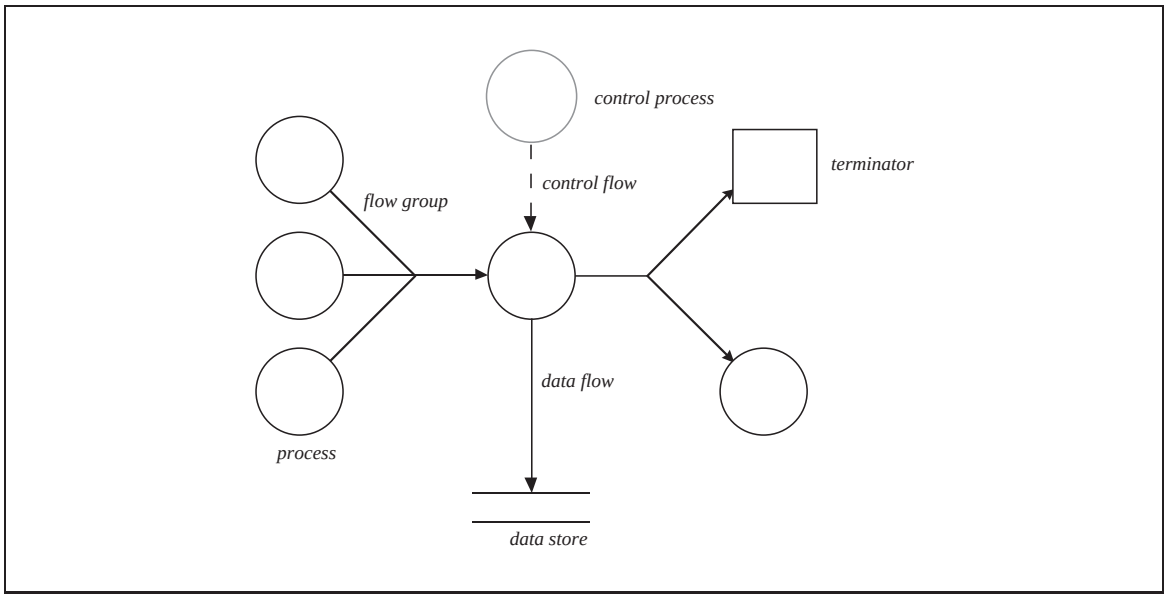

Figure 3.3: The main notions of data flow diagrams

Flows represent data "in motion". Several types of flows exist. A simple flow has a source and a destination. Processes, data stores and terminators can be source or destination of simple flows. A complex flow consists of a set of flows converging to one other flow or a flow diverging into a set of other flows. Control flows represent triggers, i.e. signals or interrupts.

In figure 3.4 a meta-model is shown, based on the previous discussion. Some of the graphical constraints in this meta-model deserve some further explanation. The two exclusion constraints attached to binary fact types express that the source and the destination of a *Data-Flow* are different and that the source and the destination of a *Control-Flow* are different. The two uniqueness constraints each over two fact types, express that no two *Data-Flows* with the same *Name* have the same *DFD-Object* as destination and that no two *Data-Flows* with the same *Name* have the same *DFD-Object* as source. The occurrence frequency constraint on the role with role name *relates-to-lower-level* expresses that a *Data-Flow* is related to at most two other *Data-Flows* on a lower decomposition level. The exclusion constraint attached to the power type *Flow-Group* states that a *Data-Flow* does not occur in more than one *Flow-Group*.

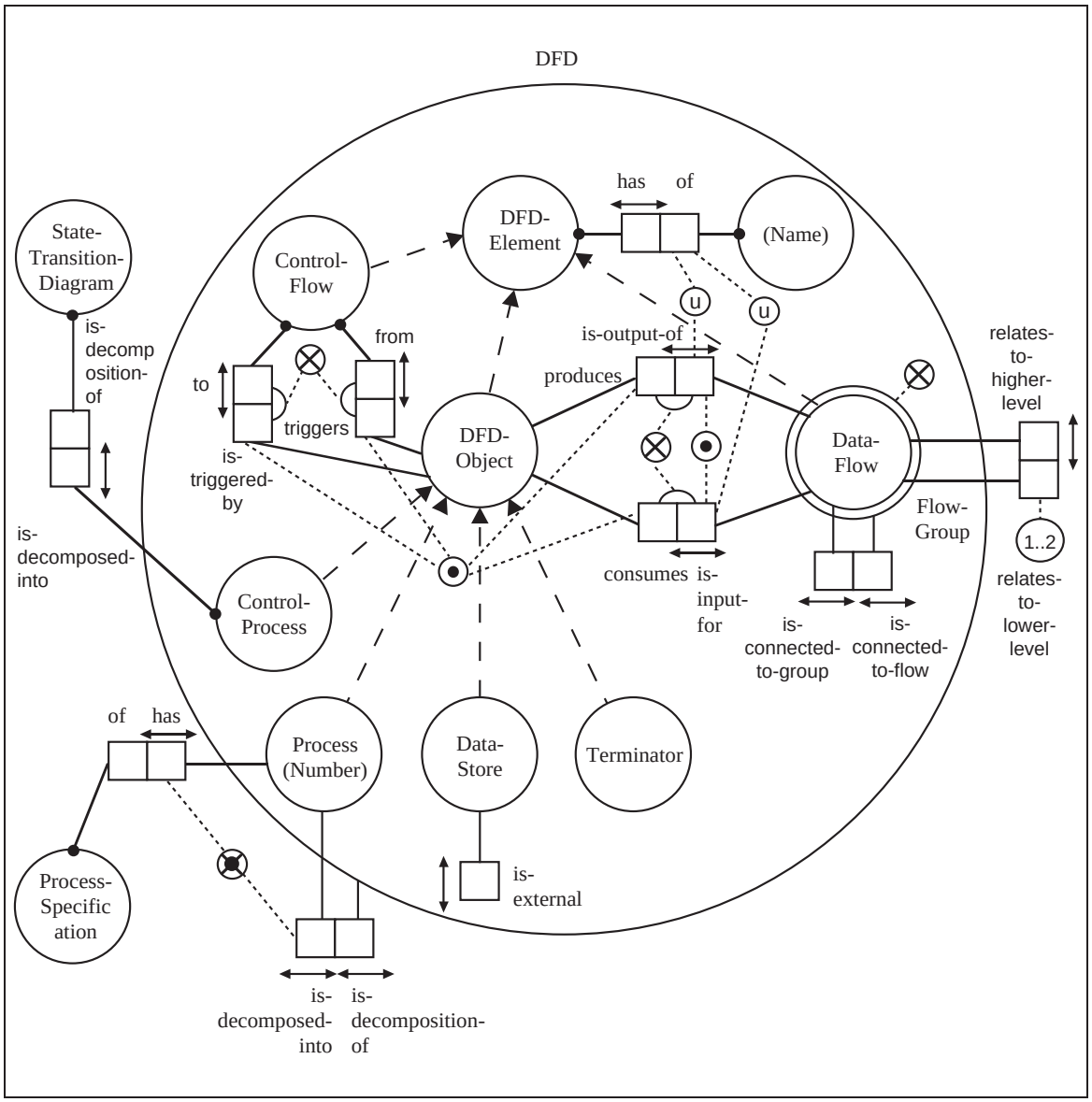

Figure 3.4: Meta-model of data flow diagrams

Chapter 4

Integrity constraints

4.1 Introduction

Usually, not all possible populations of an information structure are valid, because some populations do not correspond with states in the Universe of Discourse. Static constraints are used to restrict the set of possible populations of an information structure to those populations that do have a corresponding state in the Universe of Discourse. Dynamic constraints are used to exclude unwanted transitions between populations.

Naturally, it is not possible, nor even desirable, to specify all static constraints graphically. Some types of static constraints however, occur frequently. A graphical representation then facilitates their specification and improves comprehensibility. Furthermore, some of these constraint types are important for the translation of a conceptual model to a concrete implementation. Uniqueness constraints for example, can be used as a base for efficient access mechanisms (indexes).

In this chapter, a number of graphical constraints are considered. Some of these constraints are very powerful and consequently have a complicated semantics. So let's start with a formal context. A schema $\Sigma = \langle \mathcal{I}, \mathcal{R} \rangle$ consists of an information structure $\mathcal{I}$ and a set of constraints $\mathcal{R}$. A population of a schema should be a population of its information structure and satisfy its constraints:

$$\mathsf{IsPop}(\Sigma, \mathsf{Pop}) = \mathsf{IsPop}(\mathcal{I}, \mathsf{Pop}) \wedge \forall_{r \in \mathcal{R}} \left[\mathsf{Pop} \models r \right]$$

The set of all possible graphical constraints of an information structure $\mathcal{I}$ is denoted by $\mathbf{\Gamma}(\mathcal{I})$. Consequently, $\mathcal{R} \subseteq \mathbf{\Gamma}(\mathcal{I})$. The set of all populations of a schema Σ is defined as:

$$\mathsf{POP}_\Sigma = \left\{ \mathsf{Pop} \in \mathsf{POP}_\mathcal{I} \;\middle|\; \mathsf{IsPop}(\Sigma, \mathsf{Pop}) \right\}$$

4.2 Uniqueness constraints

The uniqueness of values in some set of predicators has become a widely used concept in database technology, both for guaranteeing integrity and as a base for efficient access mechanisms. Three types of uniqueness constraints exist: uniqueness constraints restricted to a single fact type, uniqueness constraints over several joinable fact types and uniqueness constraints over objectification.

A uniqueness constraint is based on a nonempty set of predicators $\tau \subseteq \mathcal{P}$. A population satisfies the uniqueness constraint τ, denoted as $\mathsf{Pop} \models \mathsf{unique}(\tau)$, if and only if τ is an identifier (also called a key) of the derived relation $\xi(\tau)$. This derived relation will be discussed in the rest of this section.

4.2.1 Uniqueness constraints in a single fact type

If a uniqueness constraint τ does not exceed the boundaries of a single fact type f, that is $\tau \subseteq f$, then $\xi(\tau) = f$. In this situation, τ is a key of fact type f. A fact type can have several keys. An example of a simple uniqueness constraint is given in figure 4.1. This uniqueness constraint $\tau = \{p, q\}$ excludes the following population of fact type f:

$$\mathsf{Pop}(f) = \{\{p : a_1, q : b_1, r : c_1\}, \{p : a_1, q : b_1, r : c_2\}\}$$

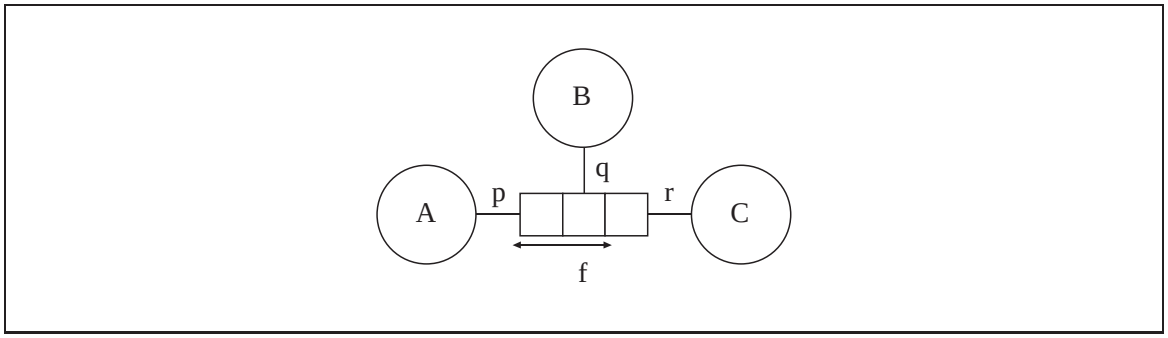

Figure 4.1: Uniqueness constraint over a single fact type

4.2.2 Uniqueness constraints requiring joins

If more than one fact type is involved in a uniqueness constraint, these fact types must be joined. Let $f = [A, B]$ and $g = [B, C]$ be two fact types. A uniqueness constraint over A and C specifies that $\{A, C\}$ is a key of the derived relation ξ defined as follows:

```
select *
from   f,g
where  f.B = g.B
```

More formally, a uniqueness constraint τ specifies a key for the derived fact type $\xi(\tau)$ defined as follows:

$$\xi(\tau) = \sigma_{C(\tau)} \bowtie_{f \in \mathsf{Facts}(\tau)} f$$

Here $\bowtie$ is an algebraic join operator which corresponds with the from-part of SQL queries, and σ is an algebraic selection operator which corresponds with the where-part of SQL queries. The where-condition $C(\tau)$ is defined as follows:

$$C(\tau) = \bigwedge_{p,q \in \rho(\tau), p \sim q} p = q$$

Here $\rho(\tau) = \bigcup \mathsf{Facts}(\tau) \backslash \tau$ is the set of predicators that occur in the fact types involved in the uniqueness constraint, but not in the uniqueness constraint itself. We consider some examples. In figure 4.2, the uniqueness constraint $\tau = \{p, s, v\}$ is specified. This constraint expresses that $\{A, E, G\}$ is a key of ξ defined as follows:

```
select *
from   f,g,h
where  f.B = g.D and
       g.F = h.F
```

An example of a population of the fact types f, g and h that is excluded by this uniqueness constraint is:

$$\mathsf{Pop}(f) = \left\{\begin{matrix}\{p:a_1, q:b_8\},\\ \{p:a_1, q:b_9\}\end{matrix}\right\} \qquad \mathsf{Pop}(g) = \left\{\begin{matrix}\{r:b_8, s:e_1, t:f_6\},\\ \{r:b_9, s:e_1, t:f_7\}\end{matrix}\right\}$$
$$\mathsf{Pop}(h) = \left\{\begin{matrix}\{u:f_6, v:g_1\},\\ \{u:f_7, v:g_1\}\end{matrix}\right\}$$

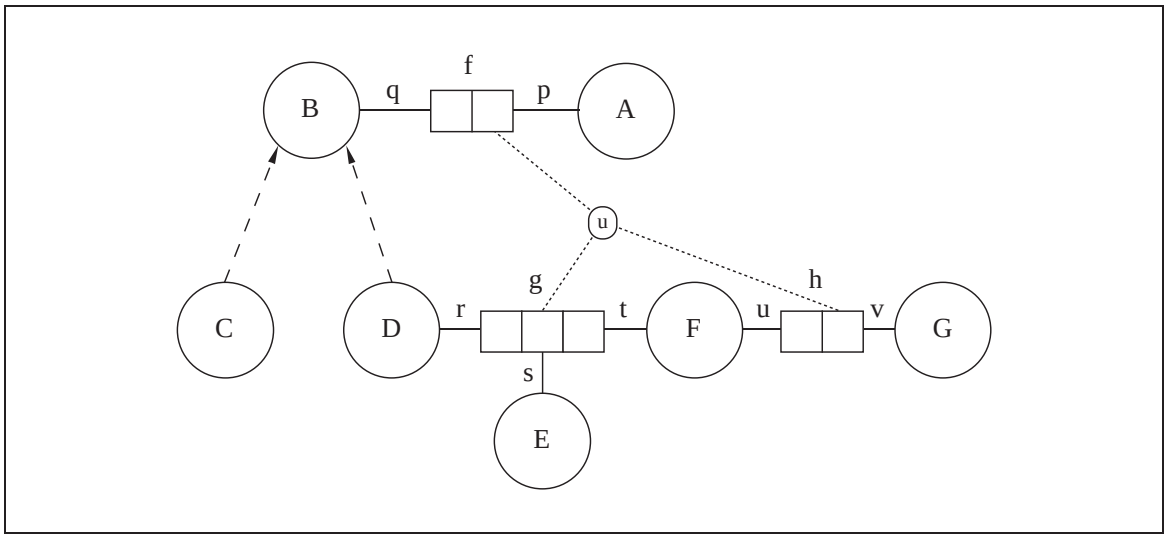

Figure 4.2: Example of a complex uniqueness constraint

To consider figure 4.2 in terms of the algebraic operators, we have $\mathsf{Facts}(\tau) = \{f, g, h\}$, $\rho(\tau) = \{q, r, t, u\}$ and therefore, $C(\tau) = (q = r \wedge t = u)$. The condition $\mathsf{unique}(\tau)$ thus requires that τ is a key of the following derived relation:

$$\xi(\tau) = \sigma_{q=r \wedge t=u}(f \bowtie g \bowtie h)$$

Figure 4.3 shows a peculiar uniqueness constraint $\tau = \{r, t\}$. This constraint expresses that $\{B, C\}$ is a key of ξ defined as follows:

```
select *
from   f,g
where  p=s and q=s
```

A population of the fact types f and g that is excluded by the constraint in figure 4.3 is:

$$\mathsf{Pop}(f) = \left\{\begin{matrix}\{p:a_1, q:a_1, r:b_1\},\\ \{p:a_2, q:a_2, r:b_1\}\end{matrix}\right\} \qquad \mathsf{Pop}(g) = \left\{\begin{matrix}\{s:a_1, t:c_1\},\\ \{s:a_2, t:c_1\}\end{matrix}\right\}$$

In terms of the algebraic operators, we have $\rho(\tau) = \{p, q, s\}$, and consequently $\{r, t\}$ is an identifier as follows:

$$\mathsf{identifier}(\sigma_{p=s \wedge q=s}(f \bowtie g), \{r, t\})$$

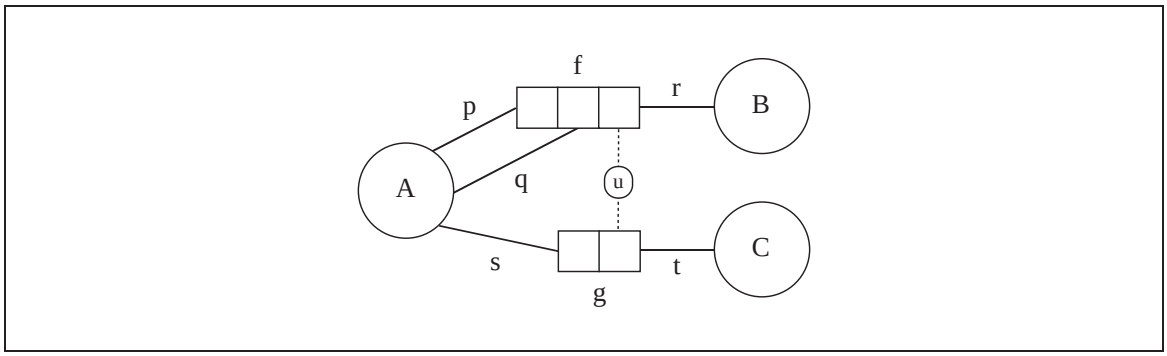

Figure 4.3: Another uniqueness constraint

Figure 4.3 demonstrates that the graphical notation of uniqueness constraints is not powerful enough to allow for alternative join conditions, for example $q = s$ only.

In figure 4.4, a uniqueness constraint over two implicit fact types of power types is shown. This constraint is joinable via common object types because $\in^e_D$ and $\in^e_E$ are type related. This constraint expresses that each set in the population of power type D should not have more than one element in common with sets in the population of power type E.

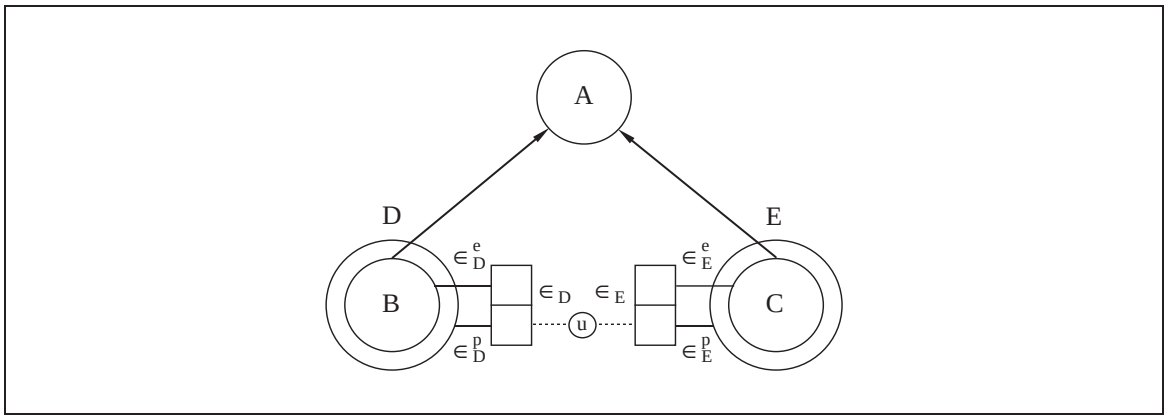

Figure 4.4: Uniqueness constraint over power types

The uniqueness constraint $\tau = \{p, u\}$ in figure 4.5, is not joinable via common object types because q and t are not type related. So this constraint is syntactically incorrect.

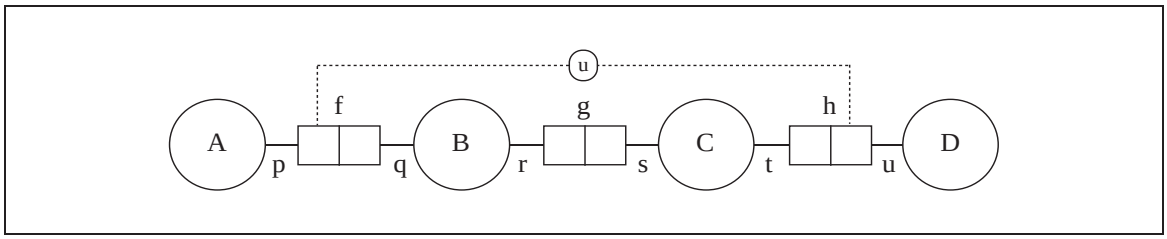

Figure 4.5: Syntactically incorrect uniqueness constraint

4.2.3 Uniqueness constraints over objectification

The most complex type of uniqueness constraint involves objectification. If objectification is involved in a uniqueness constraint, we must perform unnesting (also called flattening). Let $f = [A, B]$ and $g = [f, C]$ be two fact types. A uniqueness constraint over A and C specifies that $\{A, C\}$ is a key of the derived relation ξ which is obtained by unnesting fact type g. This derived relation is a ternary relation $[A, B, C]$. Consider the following population:

```
f  = [ A,  B ]          g = [ f,  C ]      derived relation
t1 =   a1  b1                 t1  c1           a1 b1 c1
t2 =   a1  b2                 t2  c2           a1 b2 c2
```

The above population satisfies the uniqueness constraint over A and C, because $\{A, C\}$ is a key of the derived relation. Note that this population also satisfies a uniqueness constraint over B and C. Now we will consider a population which does not satisfy the uniqueness constraint over A and C:

```
f  = [ A,  B ]          g = [ f,  C ]      derived relation
t1 =   a1  b1                 t1  c1           a1 b1 c1
t2 =   a1  b2                 t2  c1           a1 b2 c1
```

The above population does not satisfy the uniqueness constraint over A and C, because $\{A, C\}$ is not a key of the derived relation. Note that this population does satisfy a uniqueness constraint over B and C. Next we will consider some more complex uniqueness constraints over objectification. Consider the uniqueness constraint in figure 4.6.

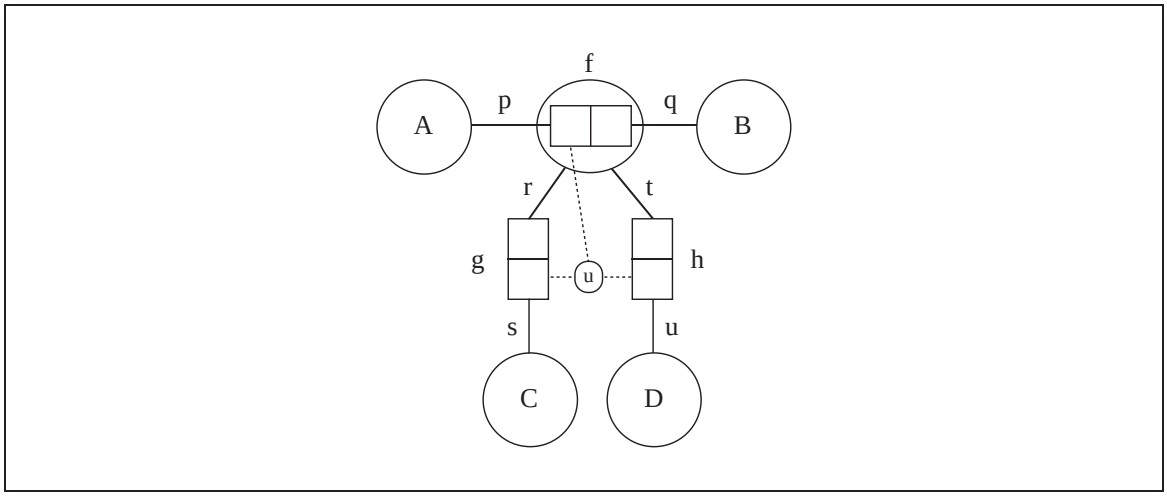

Figure 4.6: A uniqueness constraint over objectification

The uniqueness constraint in figure 4.6 expresses that $\{A, C, D\}$ is a key of the derived relation obtained by unnesting fact type g via predicator r and fact type h via predicator t. This derived relation has the form $[A, B, C, D]$. The uniqueness constraint in figure 4.6 requires unnesting, because predicator p is involved in that constraint. Therefore the simple uniqueness constraint over $\tau = \{s, u\}$ would not require unnesting and can be treated by joining g and h according to the definitions from section 4.2.2.

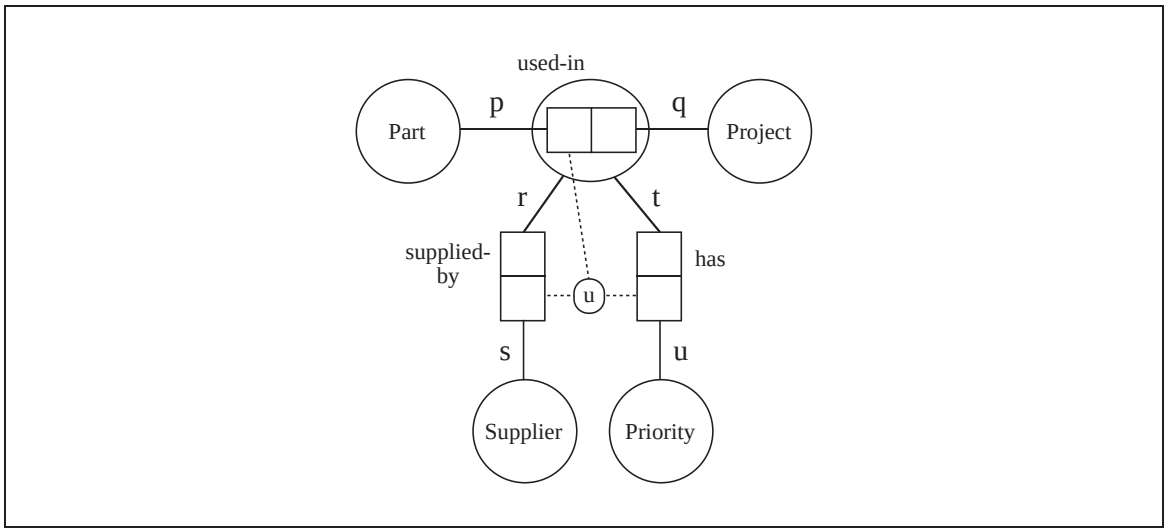

Figure 4.7: A schema regarding parts and suppliers

4.2.4 Practical examples

As a practical example of the uniqueness constraint of figure 4.6, consider figure 4.7.

The schema in figure 4.7 models that *Parts* are used in *Projects* and that *Parts* can be supplied for particular *Projects* by a *Supplier* with a certain *Priority*. Informally, the uniqueness constraint states that no two different *Projects* use the same *Part* supplied by the same *Supplier*, with the same *Priority*. As another practical example of this uniqueness constraint, consider figure 4.8.

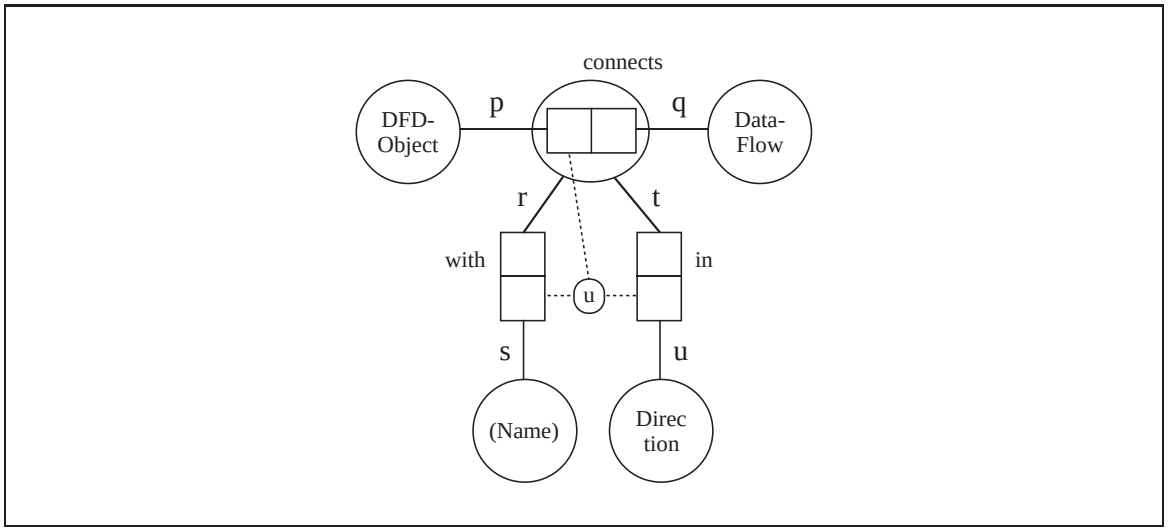

Figure 4.8: Part of meta-model for data flow diagrams

The uniqueness constraint in figure 4.8 is derived from the meta-model of data flow diagrams as presented in [73]. In this meta-model, the uniqueness constraint models the fact that no two *Data-Flows* connected to the same *DFD-Object*, with the same *Name* have the same *Direction*.

4.2.5 Using the functional representation

We sometimes prefer to view complex uniqueness constraints in the functional representation. The functional representation of the information structure of figure 4.6 is shown in figure 4.9.

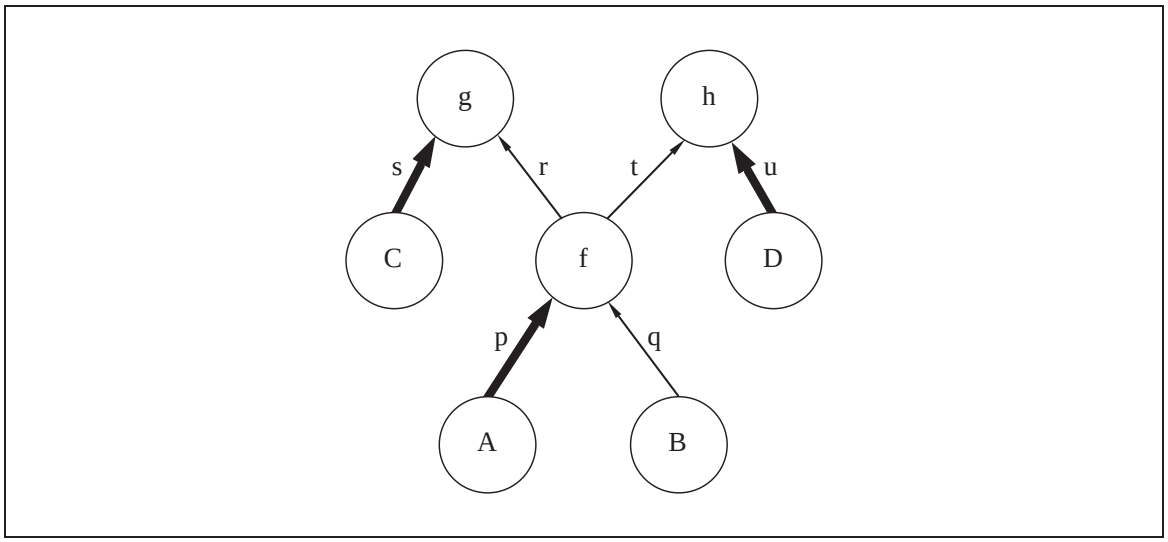

Figure 4.9: Associated functional representation of figure 4.6

In figure 4.9 we see the uniqueness constraint over $\tau = \{p, s, u\}$ drawn by thick arrows. In this representation, two tops g and h exist that are suitable for unnesting, g via predicator r, and h via predicator t. Suppose fact type g is chosen. The choice for fact type g leads to the functional representation of figure 4.10.

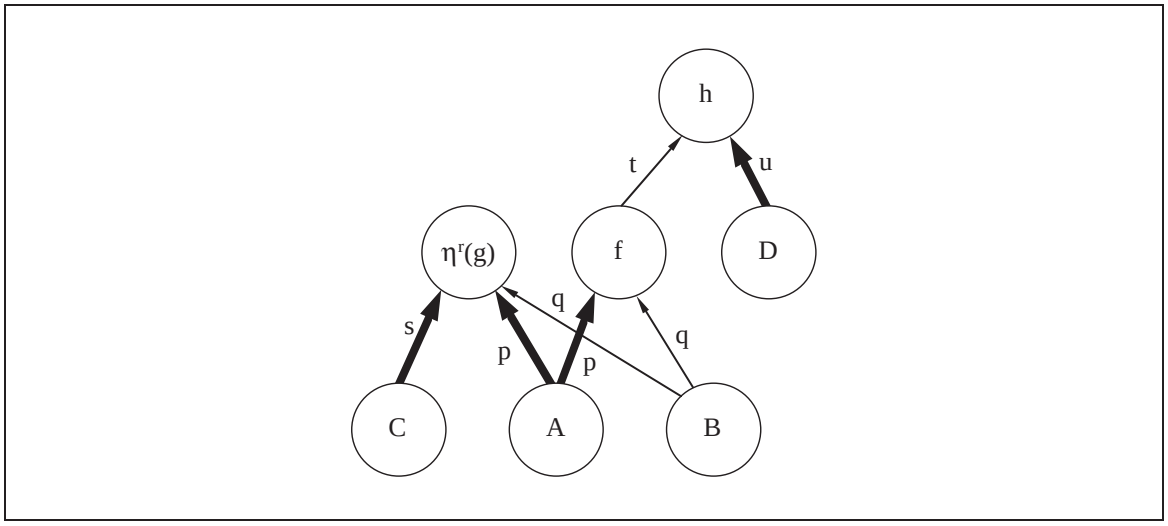

Figure 4.10: Functional representation after the first pass

In figure 4.10 we see the algebraic operation $\eta^r(g)$ which expresses that fact type g has been unnested via predicator r. Instead of $\eta^r(g)$ we may also write *unnest(r,g)*. Now fact type h can be unnested via predicator t. This results in the functional representation of figure 4.11.

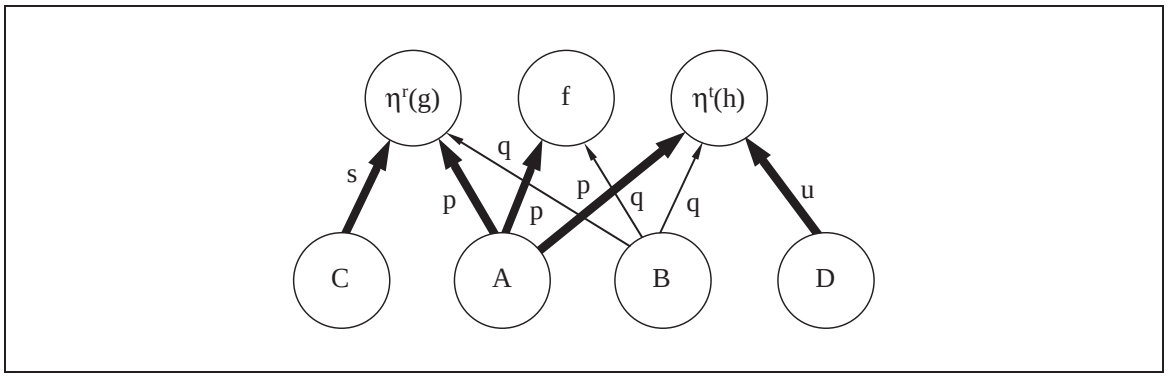

Figure 4.11: Functional representation after the second pass before reduction

In figure 4.11 fact type h has been unnested. Fact type f can now be omitted. This reduction has been performed in figure 4.12.

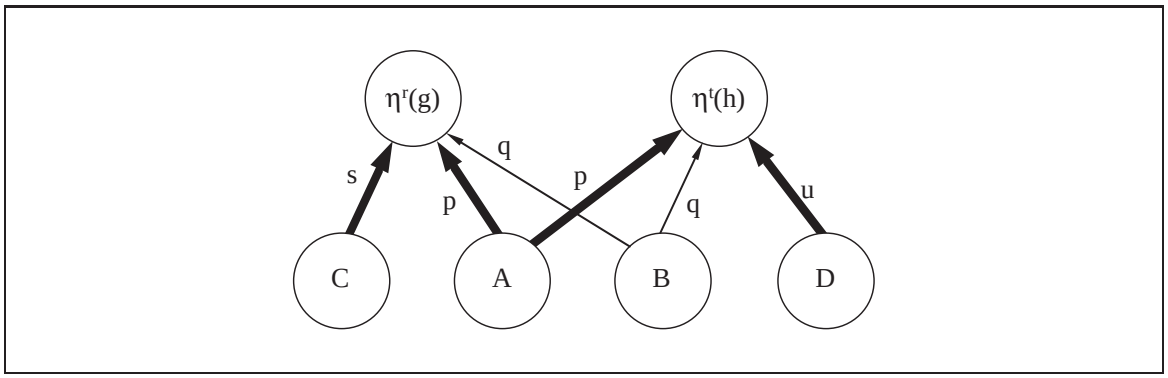

Figure 4.12: Functional representation after the second pass after reduction

In figure 4.12 we see the results $\eta^r(g)$ and $\eta^t(h)$ of unnesting fact type g and h. No unnesting is needed anymore. We can now apply the join discussed in section 4.2.2. So this constraint expresses that $\{A, C, D\}$ is a key of the derived relation $[A, B, C, D]$. In terms of algebraic operators, the semantics of this uniqueness constraint is that $\{p, s, u\}$ is a key of the derived relation ξ, where ξ is the join of the unnesting results:

$$\mathsf{identifier}\left(\eta^r(g) \bowtie \eta^t(h), \{p, s, u\}\right)$$

Looking at this join, it can be understood, intuitively, why fact type f has been reduced. The reason is that this fact type would not have contributed to this join. A population, of the fact types f, g and h, that is excluded by this uniqueness constraint is:

$$\mathsf{Pop}(f) = \left\{\begin{matrix}\{p : a_1, q : b_1\},\\ \{p : a_1, q : b_2\}\end{matrix}\right\} \qquad \mathsf{Pop}(g) = \left\{\begin{matrix}\{r : \{p : a_1, q : b_1\}, s : c_1\},\\ \{r : \{p : a_1, q : b_2\}, s : c_1\}\end{matrix}\right\}$$

$$\mathsf{Pop}(h) = \left\{\begin{matrix}\{t : \{p : a_1, q : b_1\}, u : d_1\},\\ \{t : \{p : a_1, q : b_2\}, u : d_1\}\end{matrix}\right\}$$

Summarizing, in order to make $\mathsf{Facts}(\tau)$ joinable, we first had to look for the *highest common descendants* of the involved fact types $\mathsf{Fact}(p) = f$, $\mathsf{Fact}(s) = g$ and $\mathsf{Fact}(u) = h$, which is fact type f. Then we had

to flatten (unnest) the ancestor fact types g and h, until we arrive at the situation of *joinable via common object types* discussed in section section 4.2.2. It should be remarked that the highest common descendant of a uniqueness constraint is not necessarily a fact type part of that constraint.

4.2.6 A complex uniqueness constraint over objectification

As an example of a less trivial uniqueness constraint over objectification, consider figure 4.13. The Object Interpretation Graph of the associated relevant Object Interpretation Structure, is depicted in figure 4.14.

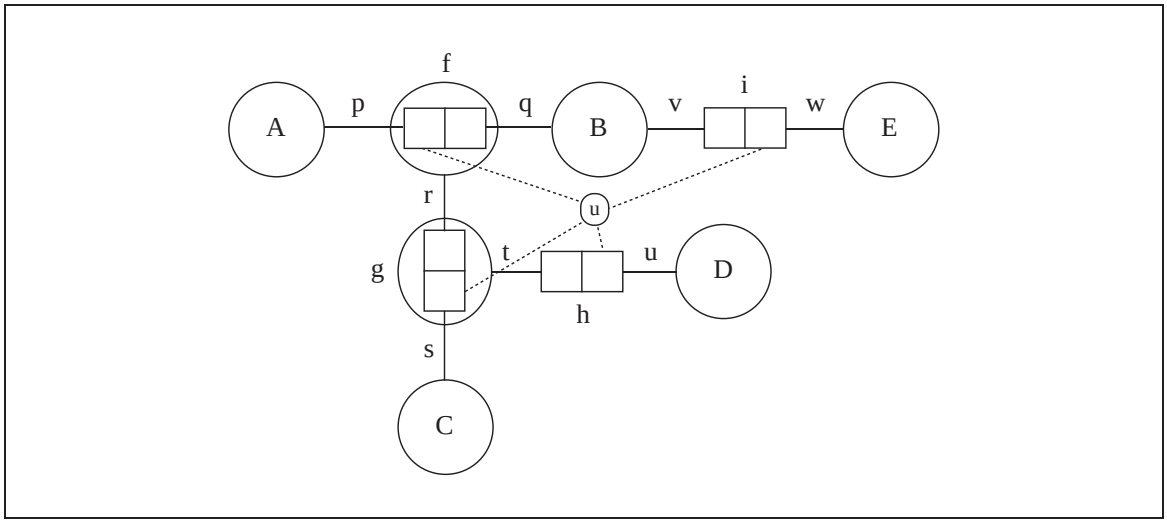

Figure 4.13: Example of a complex uniqueness constraint

The first step of the Uniquest Algorithm is to unnest top h via predicator t. This results in the Object Interpretation Graph of figure 4.15.

In this Object Interpretation Graph, node $\eta^t(h)$ has to be unnested via predicator r, which results in the Object Interpretation Graph of figure 4.16. Now the condition *joinable via common object types* is satisfied. The semantics of this uniqueness constraint therefore is:

$$\mathsf{identifier}\left(\sigma_{v=q}(i \bowtie \eta^r(\eta^t(h))), \{w, p, s, u\}\right)$$

A population, of the fact types, f, g, h and i, that is excluded by this uniqueness constraint is:

$$\begin{aligned}
\mathsf{Pop}(f) &= \left\{\begin{matrix}\{p : a_1, q : b_1\},\\ \{p : a_1, q : b_2\}\end{matrix}\right\}\\
\mathsf{Pop}(g) &= \left\{\begin{matrix}\{r : \{p : a_1, q : b_1\}, s : c_1\},\\ \{r : \{p : a_1, q : b_2\}, s : c_1\}\end{matrix}\right\}\\
\mathsf{Pop}(h) &= \left\{\begin{matrix}\{t : \{r : \{p : a_1, q : b_1\}, s : c_1\}, u : d_1\},\\ \{t : \{r : \{p : a_1, q : b_2\}, s : c_1\}, u : d_1\}\end{matrix}\right\}\\
\mathsf{Pop}(i) &= \left\{\begin{matrix}\{v : b_1, w : e_1\},\\ \{v : b_2, w : e_1\}\end{matrix}\right\}
\end{aligned}$$

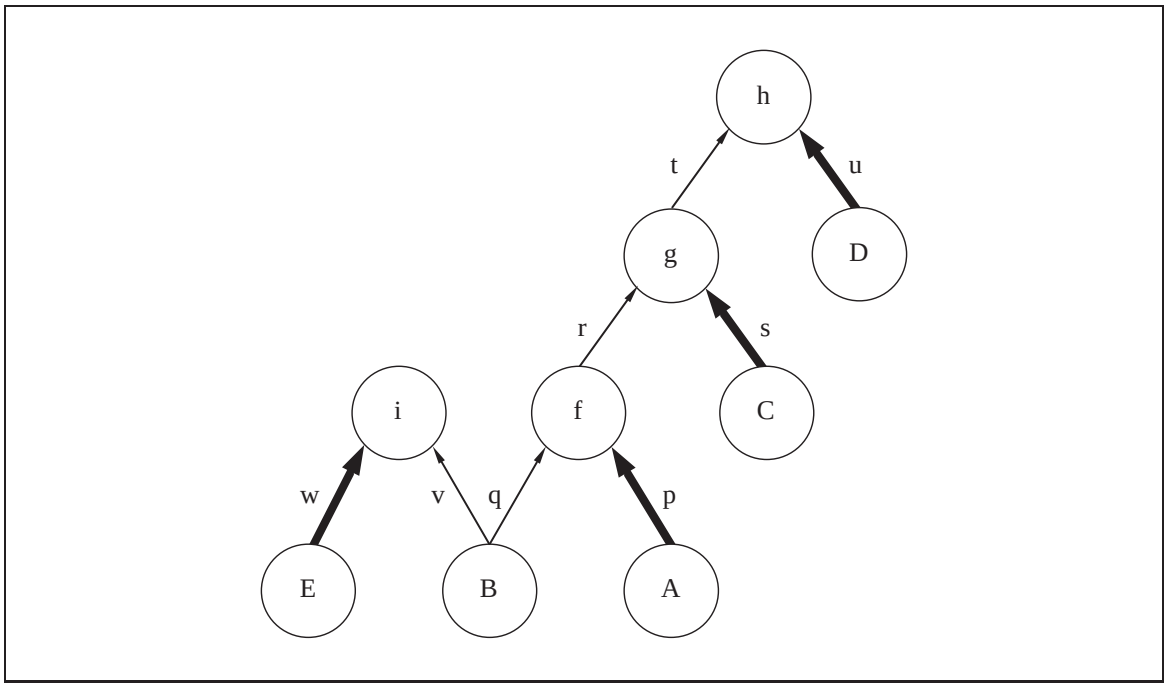

Figure 4.14: The associated Object Interpretation Graph

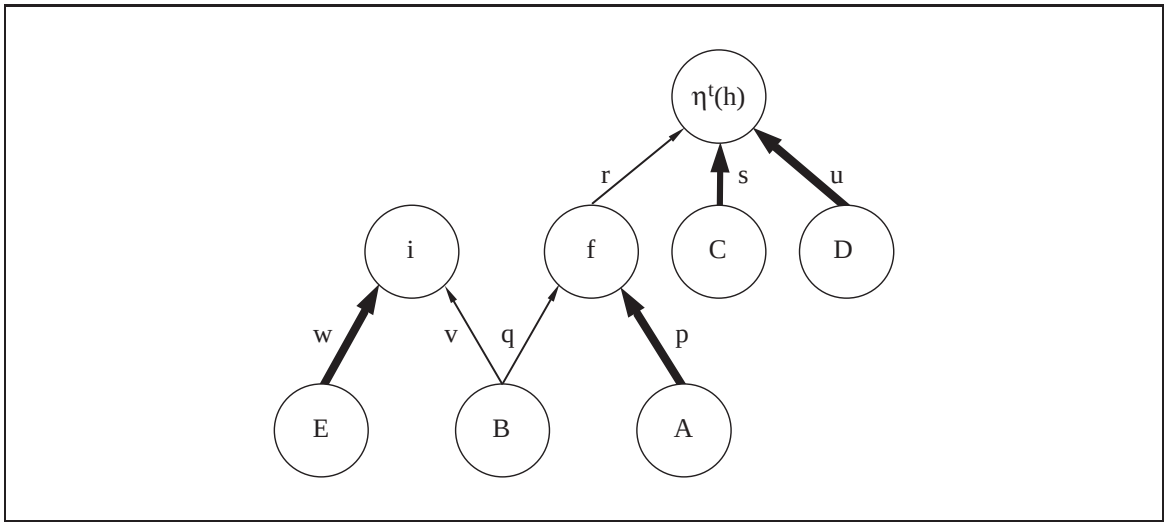

Figure 4.15: Object Interpretation Graph after the first pass

4.2.7 A uniqueness constraint requiring strong unnesting

Suppose that in the information structure of figure 4.43 the uniqueness constraint $\tau = \{p, t\}$ is specified. The corresponding Object Interpretation Graph is depicted in figure 4.17. The Uniquest Algorithm now has to choose whether fact type g has to be unnested via predicator r or via predicator s. Suppose that predicator r is chosen, then unnesting of fact type g via r results in the Object Interpretation Graph of figure 4.18.

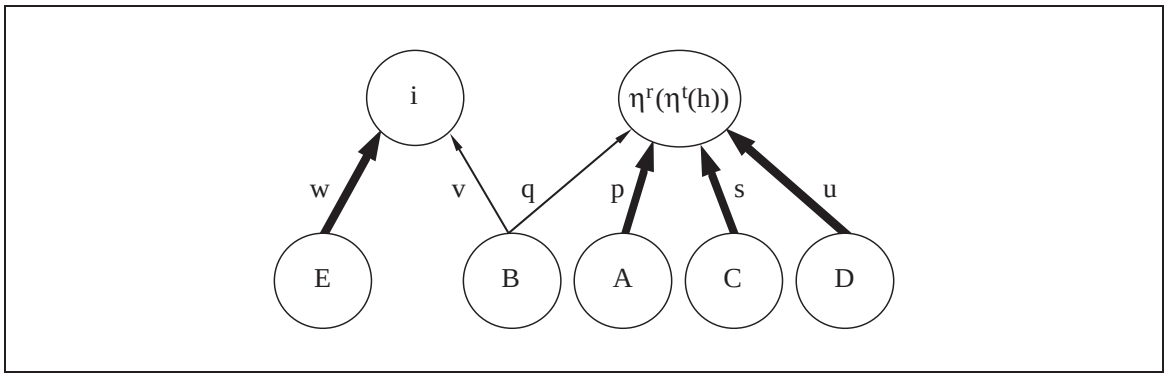

Figure 4.16: Object Interpretation Graph after the second pass

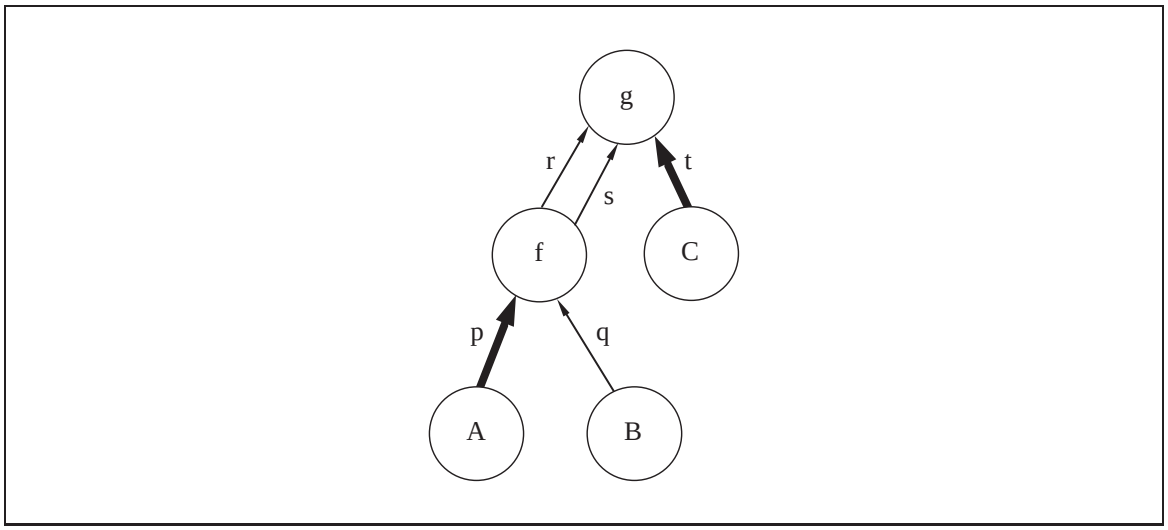

Figure 4.17: The associated Object Interpretation Graph

From the Object Interpretation Graph of figure 4.18 it is clear that top $\eta^r(g)$ should be unnested via predicator s. This results in the Object Interpretation Graph of figure 4.19. Now the first test of the Uniquest Algorithm succeeds. Consequently, the semantics of this constraint is:

$$\mathsf{identifier}\,(\eta^s(\eta^r(g)), \{p, t\})$$

As explained in example 4.15, this situation requires strong unnesting instead of plain unnesting. A population, of fact types f and g, excluded by this constraint is:

$$\begin{aligned} \mathsf{Pop}(f) &= \left\{ \begin{matrix} \{p : a_1, q : b_1\}, \\ \{p : a_1, q : b_2\} \end{matrix} \right\} \\ \mathsf{Pop}(g) &= \left\{ \begin{matrix} \{r : \{p : a_1, q : b_1\}, s : \{p : a_1, q : b_1\}, t : c_1\}, \\ \{r : \{p : a_1, q : b_2\}, s : \{p : a_1, q : b_2\}, t : c_1\} \end{matrix} \right\} \end{aligned}$$

4.2.8 Two erroneous uniqueness constraints

A uniqueness constraint over objectification is invalid if it is *ambiguous with respect to unnesting*. This means that for a certain fact type that has to be unnested via one of its predicators, several fact types exist

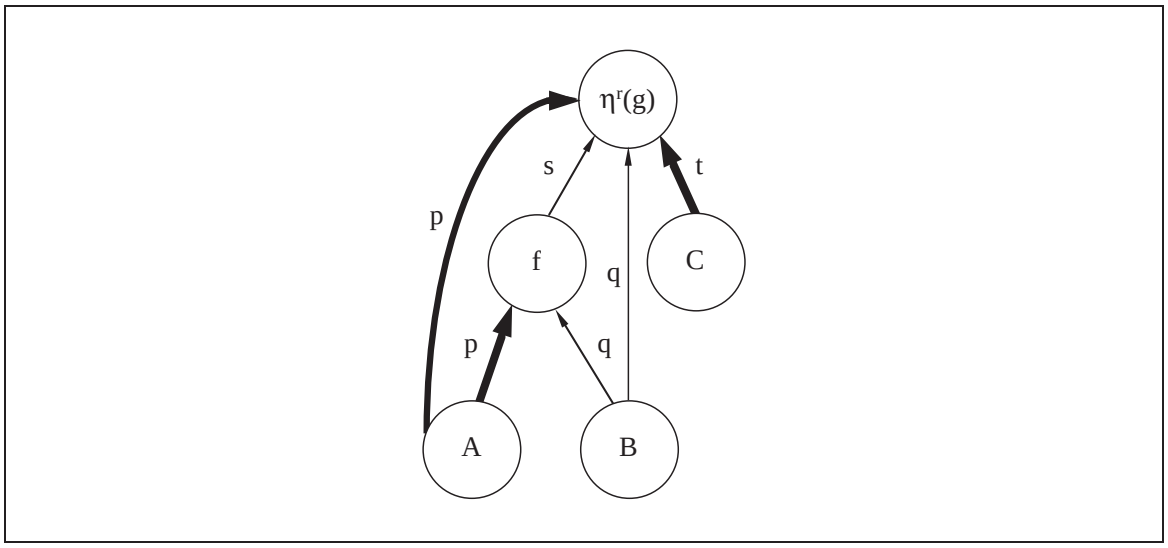

Figure 4.18: Object Interpretation Graph after the first pass

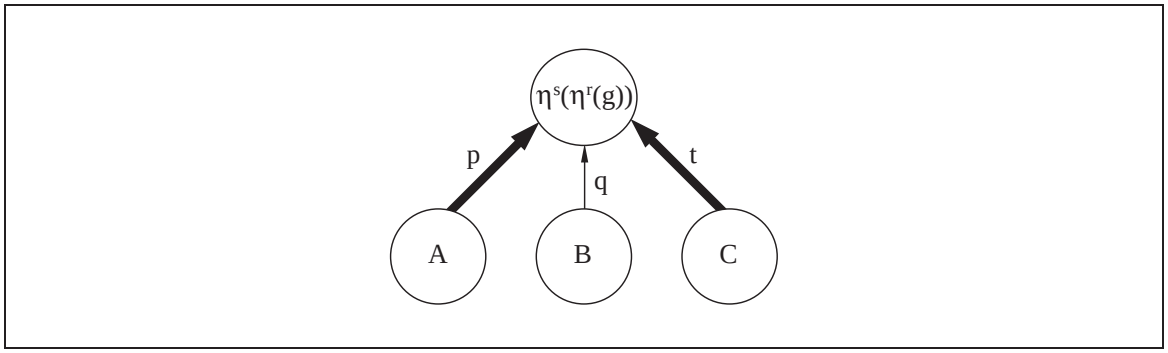

Figure 4.19: Object Interpretation Graph after the second pass

that are type related with the base of this predicator. Therefore, for unnesting, an arbitrary choice for one of these fact types has to be made. Consequently, this type of uniqueness constraint is not meaningful. An example of an ambiguous uniqueness constraint is shown in figure 4.20.

In the uniqueness constraint in figure 4.20, unnesting can be performed via either fact type f or fact type g. Therefore, this constraint is ambiguous with respect to unnesting, caused by generalisation.

In figure 4.21, a uniqueness constraint over objectification is shown and in figure 4.22, the associated Object Interpretation Graph. Fact type g has to be unnested via predicator r. This results in the Object Interpretation Graph of figure 4.23.

From this resulting Object Interpretation Graph it is clear that node $\eta^r(g)$ is a top node, with depth greater than one. However, the second condition of the Uniquest Algorithm does not apply as unnesting via predicator s is not allowed due to the fact that s is part of the uniqueness constraint. The Uniquest Algorithm therefore yields the error *no joinable descendants*.

As discussed in section 4.2.3, the uniqueness constraint of figure 4.20 is ambiguous with respect to unnesting. The second condition of the Uniquest Algorithm succeeds, as fact type h can be unnested via predicator t. However, due to the fact that both $g \sim \mathsf{Base}(t)$ and $f \sim \mathsf{Base}(t)$, the error *ambiguous with respect to unnesting* is generated. In this figure, the uniqueness constraints $\tau = \{q, u\}$ and $\nu = \{s, u\}$ would be

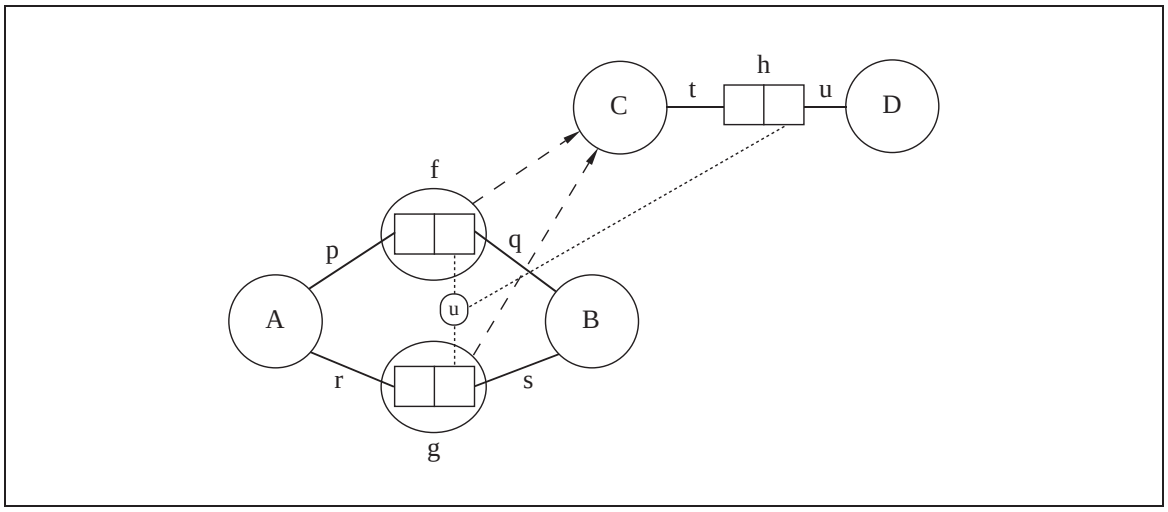

Figure 4.20: Invalid uniqueness constraint (ambiguity)

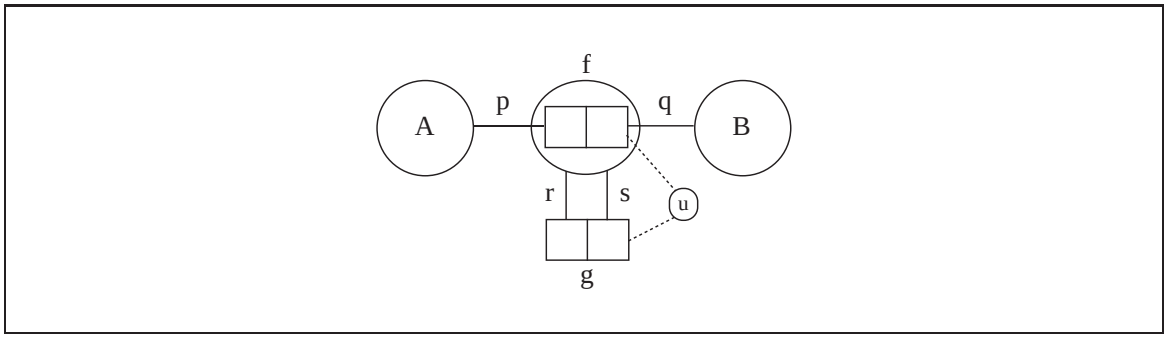

Figure 4.21: Example uniqueness constraint

correct. The semantics of the uniqueness constraint $\{q, u\}$, for example, is:

$$\mathsf{identifier}\left(\eta^t(h), \{q, u\}\right)$$

4.2.9 A uniqueness constraint with specialisation

As a final example of a uniqueness constraint, consider figure 4.24. This constraint requires unnesting of g via predicator r and fact type f (as $\mathsf{Base}(r) \sim f$). The semantics of this constraint is:

$$\mathsf{identifier}\left(\eta^r(g), \{s, q\}\right)$$

4.3 Occurrence frequency constraints

Uniqueness constraints are used to express the fact that instances of object types may play a certain combination of roles at most once. This restriction can be generalised as follows: if a certain combination of object type instances occurs in a set σ of predicators, then this combination should occur at least n and at most m times in this set. This is denoted as:

$$\mathsf{frequency}(\sigma, n, m)$$

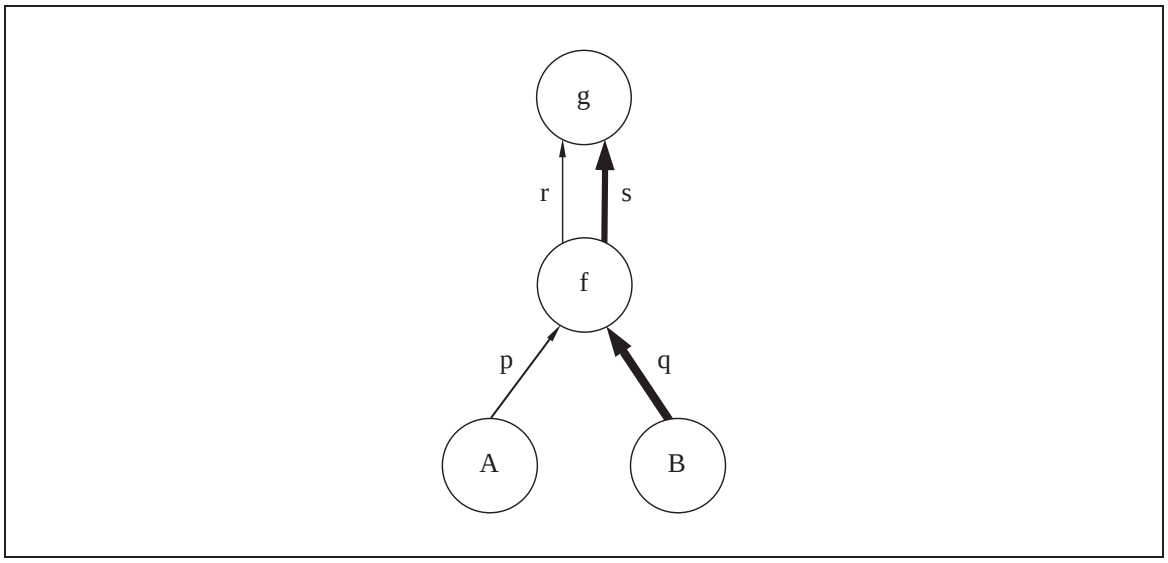

Figure 4.22: The associated Object Interpretation Graph

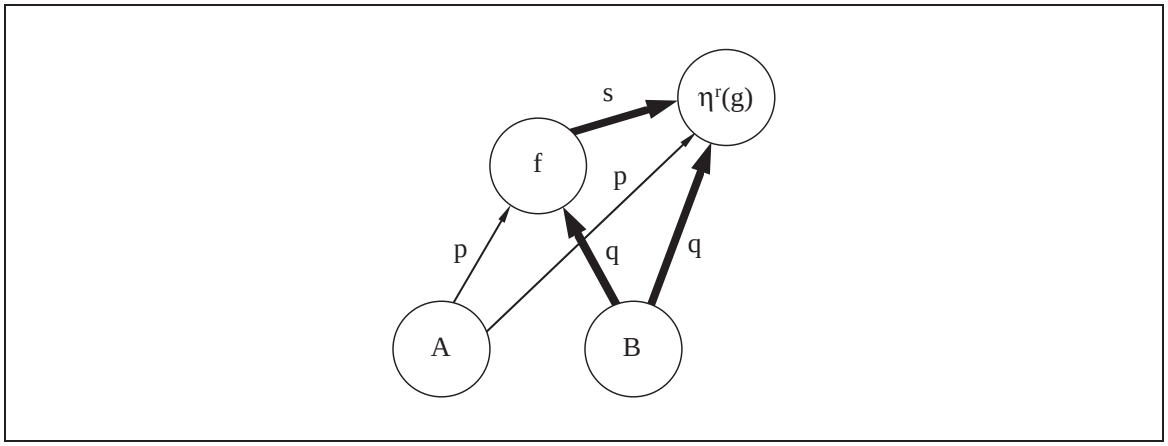

Figure 4.23: Object Interpretation Graph after the first pass

Formally, $\mathsf{Pop} \models \mathsf{frequency}(\sigma, n, m)$ iff:

$$\mathsf{Pop} \not\models \mathsf{IsEmpty}(\xi(\sigma)) \Rightarrow \begin{cases} \mathsf{Val}[\![\mathsf{min}(\chi(\xi(\sigma), \sigma, a), a)]\!]\,(\mathsf{Pop}) & \geq & n \\ \mathsf{Val}[\![\mathsf{max}(\chi(\xi(\sigma), \sigma, a), a)]\!]\,(\mathsf{Pop}) & \leq & m \end{cases}$$

Of course this implies:

$$\begin{aligned} \mathsf{Pop} \models \mathsf{unique}(\sigma) \quad &\Longleftrightarrow \quad \mathsf{Pop} \models \mathsf{frequency}(\sigma, 0, 1) \\ &\Longleftrightarrow \quad \mathsf{Pop} \models \mathsf{frequency}(\sigma, 1, 1) \end{aligned}$$

Note that $\mathsf{frequency}(\sigma, 0, 0)$ is a very strong condition, as it constrains $\xi(\sigma)$ to the empty population.

Example 4.1

The occurrence frequency constraint $\mathsf{frequency}(\{q, s\}, 0, 0)$ *in figure 4.25 ensures:*

$$\mathsf{Val}[\![\theta_p(\pi_p(f))]\!]\,(\mathsf{Pop}) \cap \mathsf{Val}[\![\theta_r(\pi_r(g))]\!]\,(\mathsf{Pop}) = \varnothing$$

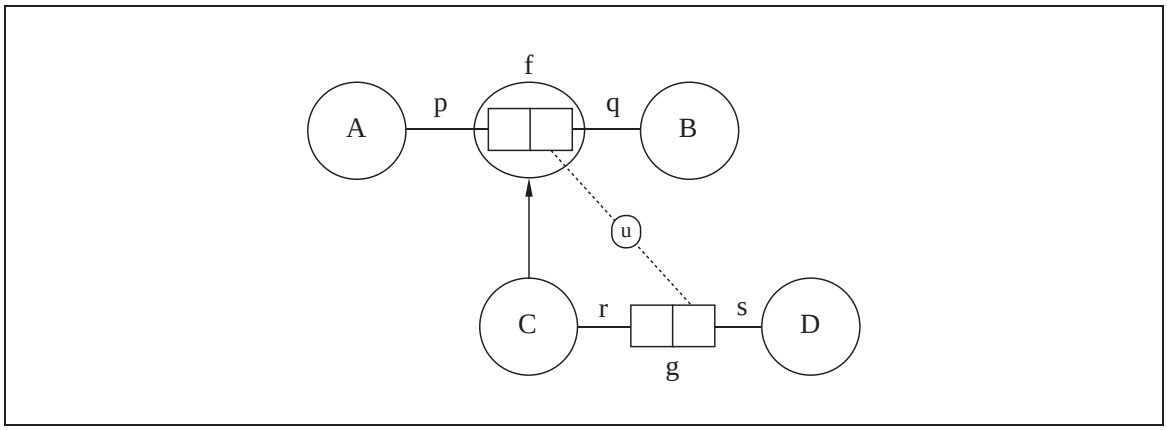

Figure 4.24: Uniqueness constraint involving subtyping

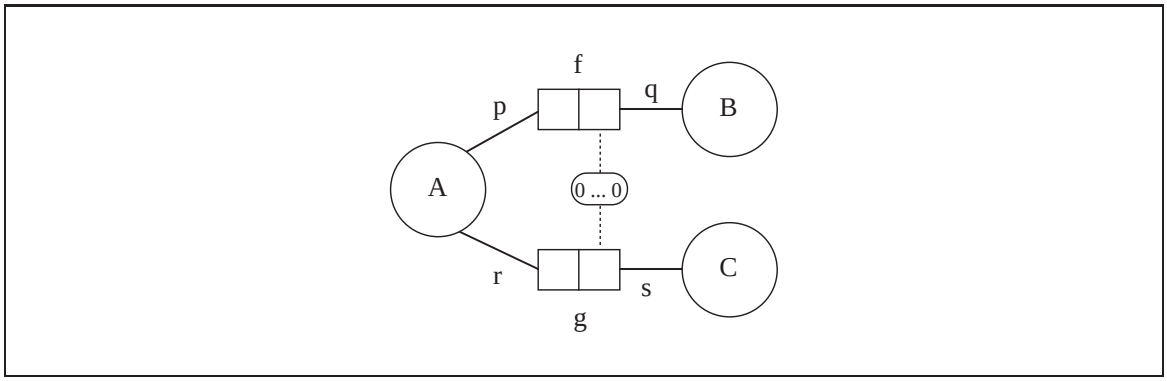

Figure 4.25: Example of an occurrence frequency constraint

This can be alternatively formulated as:

$$\mathsf{Pop} \models \pi_p(f) \otimes_{\{p\,:\,r\}} \pi_r(g)$$

We will come back to this example in the next section. □

4.4 Total role constraints

An object type may have the property that in any population, all its instances must be involved in some set of predicators. This property is called a total role constraint.

Syntactically, a total role constraint τ is a nonempty set of predicators, i.e. $\tau \neq \varnothing \wedge \tau \subseteq \mathcal{P}$. A total role constraint τ only makes sense if all predicators in τ are type related:

$$\forall_{p,q \in \tau}\, [p \sim q]$$

A population Pop satisfies the total role constraint τ, denoted as $\mathsf{Pop} \models \mathsf{total}(\tau)$, iff the following equation holds:

$$\bigcup_{q \in \tau} \mathsf{Pop}(\mathsf{Base}(q)) = \bigcup_{q \in \tau} \mathsf{Val}[\![\theta_q(\pi_q(\mathsf{Fact}(q)))]\!]\,(\mathsf{Pop})$$

If for all populations Pop of schema Σ, that is $\mathsf{IsPop}(\Sigma, \mathsf{Pop})$, we have $\mathsf{Pop} \models \mathsf{total}(\tau)$, the assertion $\mathsf{total}(\tau)$ is true. We will write $\mathsf{total}(\tau) \in \mathcal{R}$, if and only if τ is a specified total role constraint in schema $\Sigma = \langle \mathcal{I}, \mathcal{R} \rangle$. Obviously, $\mathsf{total}(\tau) \in \mathcal{R} \Rightarrow \mathsf{total}(\tau)$. The reverse however is not true: $\mathsf{total}(\tau) \not\Rightarrow \mathsf{total}(\tau) \in \mathcal{R}$.

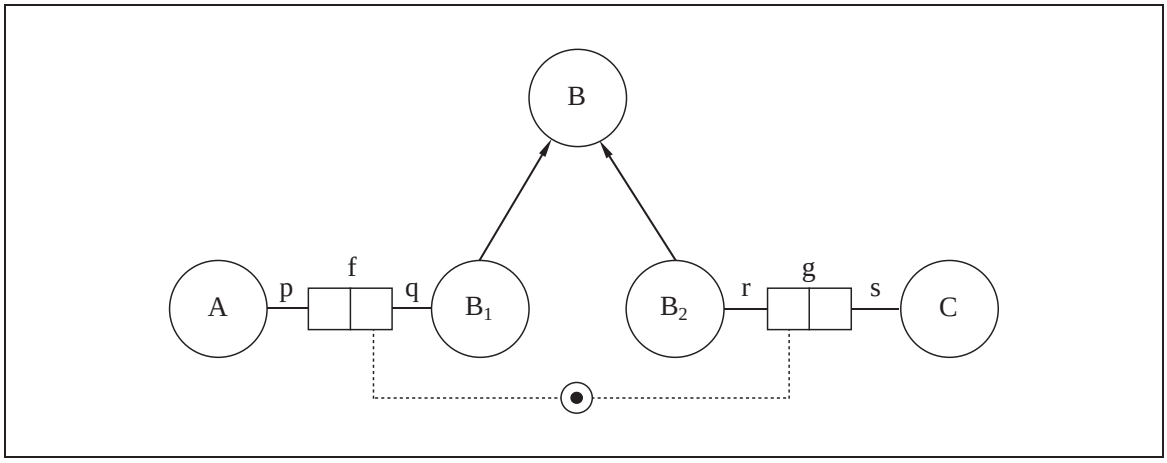

Figure 4.26: Example of a total role constraint

Example 4.2

Consider the total role constraint $\{q, r\}$ in the information structure of figure 4.26. The meaning of this constraint is:

$$\mathsf{Pop}(B_1) \cup \mathsf{Pop}(B_2) = \mathsf{Val}[\![\theta_q(\pi_q(f))]\!]\,(\mathsf{Pop}) \cup \mathsf{Val}[\![\theta_r(\pi_r(g))]\!]\,(\mathsf{Pop})$$

A population that is excluded by this constraint is:

$$\begin{array}{lcllcl} \mathsf{Pop}(B_1) & = & \{b_1, b_2\} & \mathsf{Pop}(f) & = & \{\{q : b_2, p : a_1\}\} \\ \mathsf{Pop}(B_2) & = & \{b_2, b_3\} & \mathsf{Pop}(g) & = & \{\{r : b_3, s : c_1\}\} \\ \mathsf{Pop}(A) & = & \{a_1\} & \mathsf{Pop}(C) & = & \{c_1\} \\ \mathsf{Pop}(B) & = & \{b_1, b_2, b_3, b_4\} & & & \end{array}$$

□

4.5 Set constraints

Other frequently occurring types of constraints are the so-called set constraints. Let σ and τ be nonempty sets of predicators, and ϕ a match between σ and τ, then:

1. $\mathsf{Pop} \models \mathsf{subset}_\phi(\sigma, \tau)$ if and only if $\mathsf{Pop} \models \pi_\sigma\,(\xi(\sigma)) \subseteq_\phi \pi_\tau\,(\xi(\tau))$
2. $\mathsf{Pop} \models \mathsf{equal}_\phi(\sigma, \tau)$ if and only if $\mathsf{Pop} \models \pi_\sigma\,(\xi(\sigma)) =_\phi \pi_\tau\,(\xi(\tau))$
3. $\mathsf{Pop} \models \mathsf{exclusion}_\phi(\sigma, \tau)$ if and only if $\mathsf{Pop} \models \pi_\sigma\,(\xi(\sigma)) \otimes_\phi \pi_\tau\,(\xi(\tau))$

A matching ϕ is a bijection between σ and τ where each $\phi(x)$ is type related with x (see also section 4.11.2). Usually, the matching ϕ is immediate from the context, and is omitted. In figure 4.27 however, an example of an exclusion constraint is shown where the matching is not clear from the context. The matching could be either $\{p : r, q : s\}$ or $\{p : s, q : r\}$.

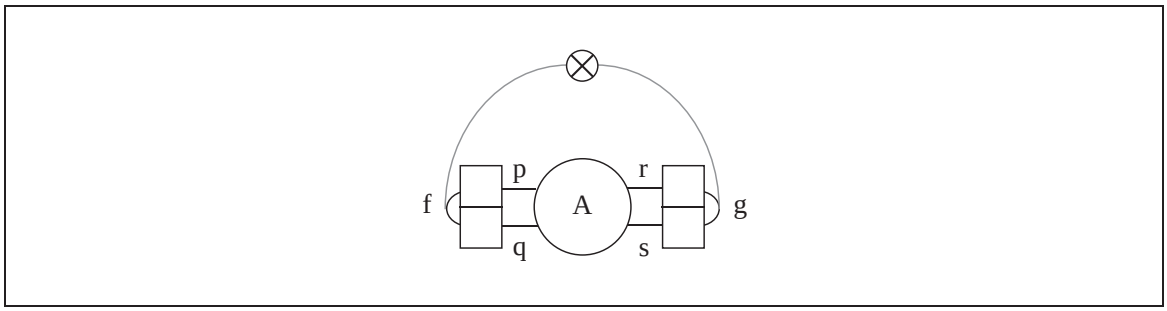

Figure 4.27: An example exclusion constraint

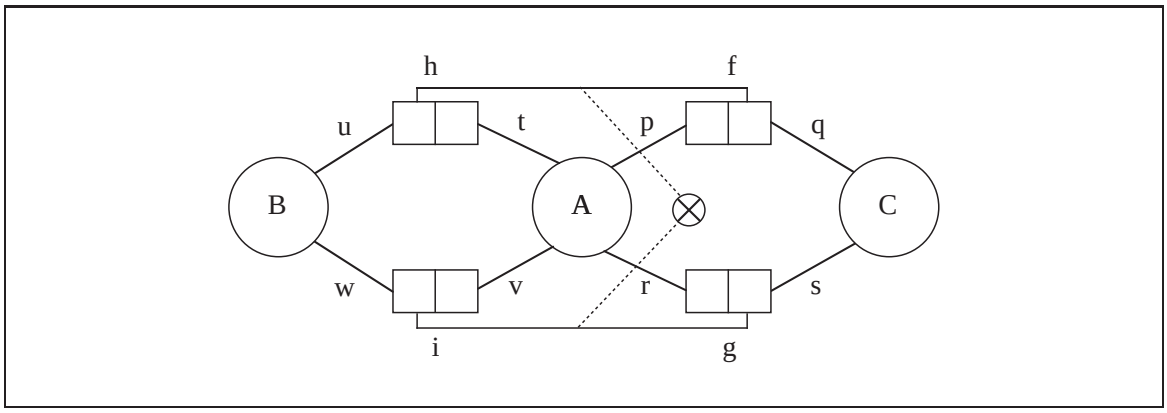

Figure 4.28: Example of a complex exclusion constraint

Example 4.3

The semantics of the exclusion constraint of figure 4.28 is:

$$\pi_{u,q}\left(\sigma_{t=p}(h \bowtie f)\right) \otimes \pi_{w,s}\left(\sigma_{v=r}(i \bowtie g)\right)$$

The matching $\{u : w, q : s\}$ is omitted, because it can be derived from the schema. A sample population of the fact types f, g, h and i, excluded by this constraint is:

$$\begin{array}{lcllcl} \mathsf{Pop}(h) & = & \{\{u : b_1, t : a_1\}\} & \mathsf{Pop}(f) & = & \{\{p : a_1, q : c_1\}\} \\ \mathsf{Pop}(i) & = & \{\{w : b_1, v : a_2\}\} & \mathsf{Pop}(g) & = & \{\{r : a_2, s : c_1\}\} \end{array}$$

□

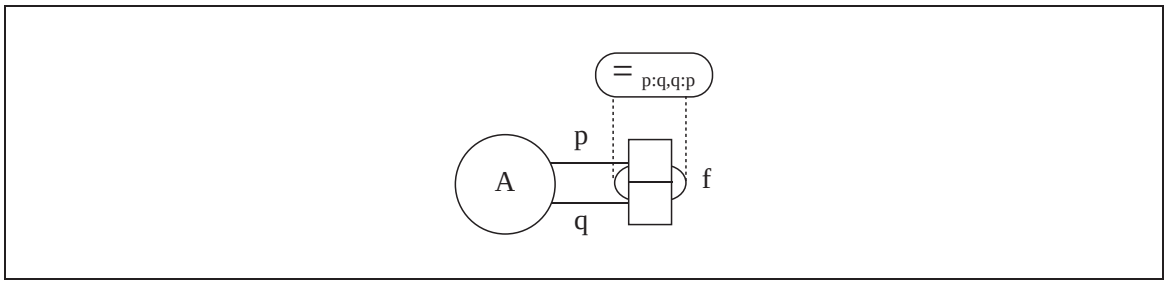

Figure 4.29: A symmetric homogeneous relation

Example 4.4

Figure 4.29 shows a constraint ensuring symmetry for a homogeneous binary fact type. □

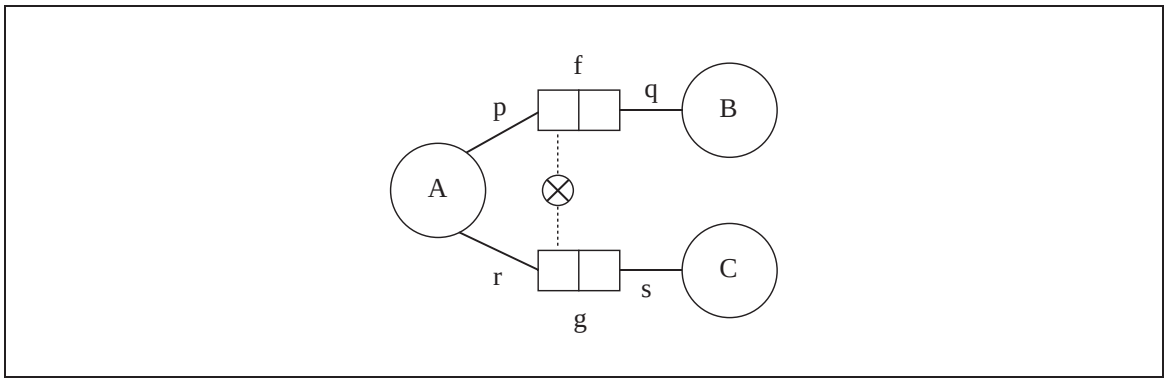

Figure 4.30: Occurrence frequency constraint as exclusion constraint

Example 4.5

The occurrence frequency constraint of figure 4.25 is specified as an exclusion constraint in figure 4.30. □

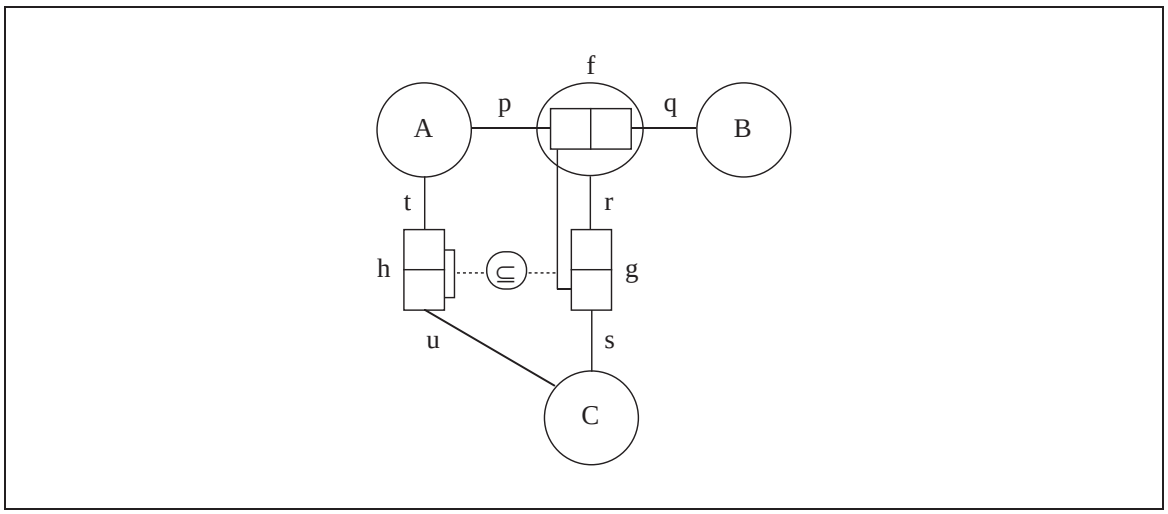

Figure 4.31: Subset constraint over objectification

Example 4.6

As the set constraints make use of the ξ operator, these types of constraints can also be specified for objectified structures. An example is shown in figure 4.31. The semantics of this constraint is:

$$h \subseteq \pi_{p,s}(\eta^r(g))$$

Again, the matching $\{t : p, u : s\}$ is clear from the schema. A population of the fact types f, g and h excluded by this constraint is:

$$\begin{array}{lcllcl} \mathsf{Pop}(f) & = & \{\{p : a_1, q : b_1\}\} & \mathsf{Pop}(g) & = & \{r : \{p : a_1, q : b_1\}, s : c_1\} \\ \mathsf{Pop}(h) & = & \{\{t : a_1, u : c_2\}\} & & & \end{array}$$

This population would be correct if $\mathsf{Pop}(h) = \{\{t : a_1, u : c_1\}\}$. □

4.6 Enumeration constraint

An enumeration constraint is used to explicitly restrict the values of a label type to an enumerated domain. If l is a label type and V a set of values, then $\mathsf{Pop} \models \mathsf{enumeration}(l, V)$ iff:

$$\mathsf{Pop}(l) \subseteq V$$

An alternative formulation is:

$$\mathsf{Dom}(l) = V$$

An example of an enumeration constraint can be found in figure 2.24, where the population of label type *PT-code* is restricted to the values *'House'* and *'Car'*.

4.7 Power type constraints

In this section, some typical constraint types for power types are defined. Similar constraint types can be devised (and formally defined) for sequence types and schema types. This will not be done however.

4.7.1 Power exclusion constraint

A power type may have the property that in any population all its instances are disjunct. This property can be expressed using a *power exclusion constraint*. Graphically, this constraint has the same representation as the exclusion constraint for fact types, but it is connected to the power type involved. In figure 4.32 an example of a power exclusion constraint is shown. This power exclusion constraint expresses that a *Student* should not be a member of several *Tutorial-groups*.

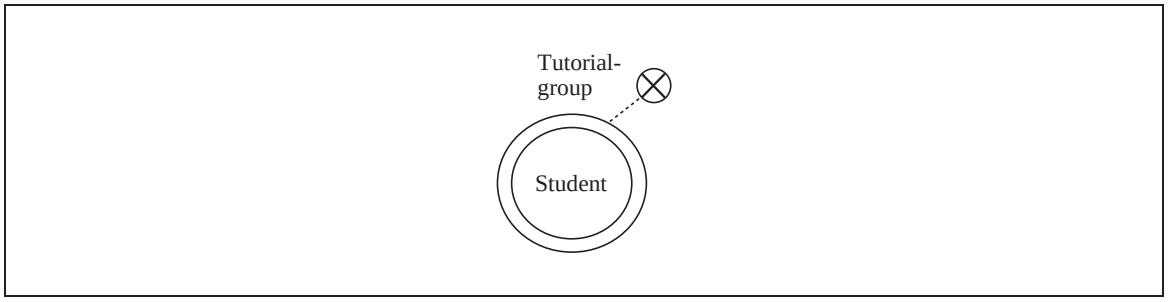

Figure 4.32: Example of power exclusion constraint

The power exclusion constraint of figure 4.32 can be expressed by means of a key on the implicit fact type associated with the power type *Tutorial-group*, as shown in figure 4.33. This is not always possible as power exclusion constraints may also be specified for subtypes of power types. These subtypes do not have an associated implicit fact type.

Formally, a power exclusion constraint $\mathsf{pow\_exclusion}(g)$ refers to an object type g with $\sqcap(g) \in \mathcal{G}$. A population Pop satisfies a power exclusion constraint $\mathsf{pow\_exclusion}(g)$, denoted as $\mathsf{Pop} \models \mathsf{pow\_exclusion}(g)$, iff:

$$\forall_{x,y \in \mathsf{Pop}(g)} [x \cap y \neq \varnothing \Rightarrow x = y]$$

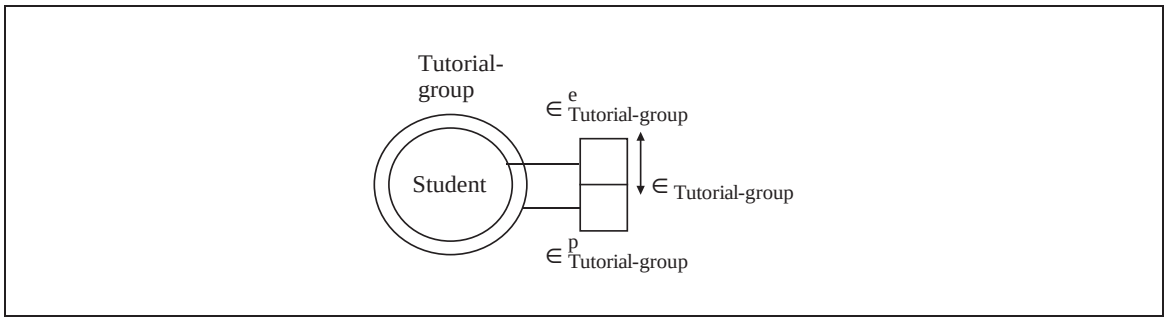

Figure 4.33: Power exclusion constraint of figure 4.32 as key on implicit fact type

4.7.2 Cover constraint

The requirement that every instance of a certain object type has to occur in an instance of a power type can be expressed by a *cover constraint*. Graphically, this constraint is depicted as a total role constraint connected to the power type involved. An example is shown in figure 4.34: each *Student* should be a member of at least one *Tutorial-group*.

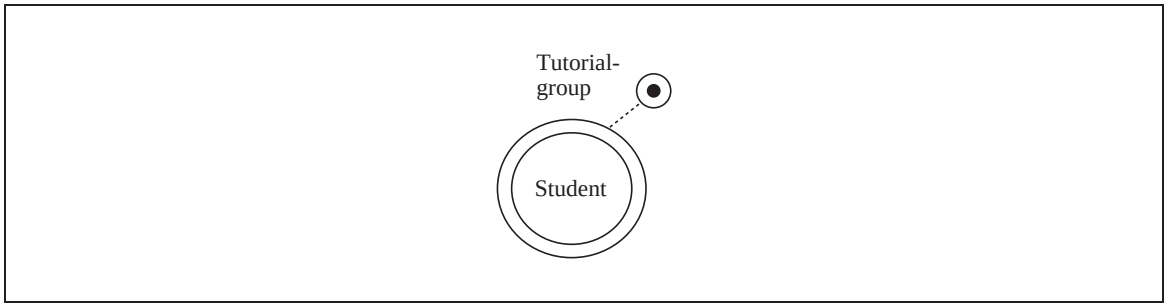

Figure 4.34: Example of cover constraint

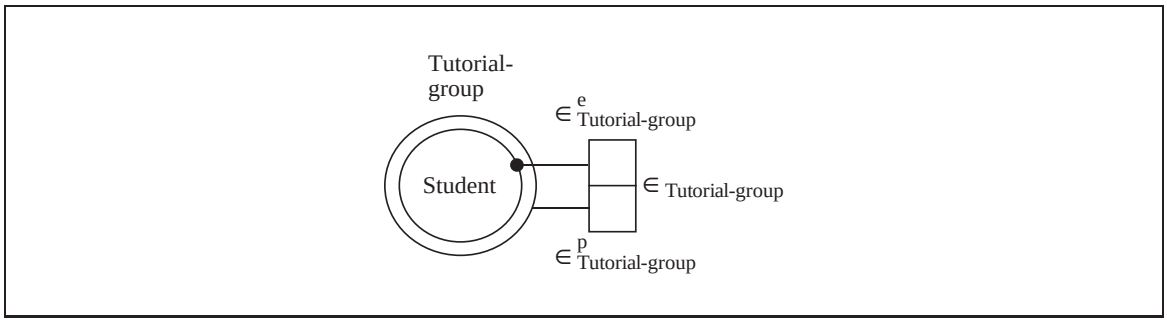

Figure 4.35: Cover constraint of figure 4.34 as total role on implicit fact type

The cover constraint of figure 4.34 can be expressed by means of a total role constraint on the implicit fact type associated with the power type *Tutorial-group*, as shown in figure 4.35. Again, a cover constraint cannot always be expressed as a total role constraint on an implicit fact type, as a cover constraint may also be assigned to a subtype of a power type. The population of the subtype should then cover the population of the element type of its pater familias.

Formally, a cover constraint $\mathsf{cover}(g)$ refers to an object type g with $\sqcap(g) \in \mathcal{G}$. A population Pop satisfies a cover constraint $\mathsf{cover}(g)$, $\mathsf{Pop} \models \mathsf{cover}(g)$, iff:

$$\bigcup \mathsf{Pop}(g) = \mathsf{Pop}(\mathsf{Elt}(\sqcap(g)))$$

Naturally, it is possible to combine the power exclusion constraint and the cover constraint. In this way it can be expressed that the population of an object type is a partition of the population of the element type of its pater familias. An example is shown in figure 4.36: each *Student* should be a member of precisely one *Tutorial-group*.

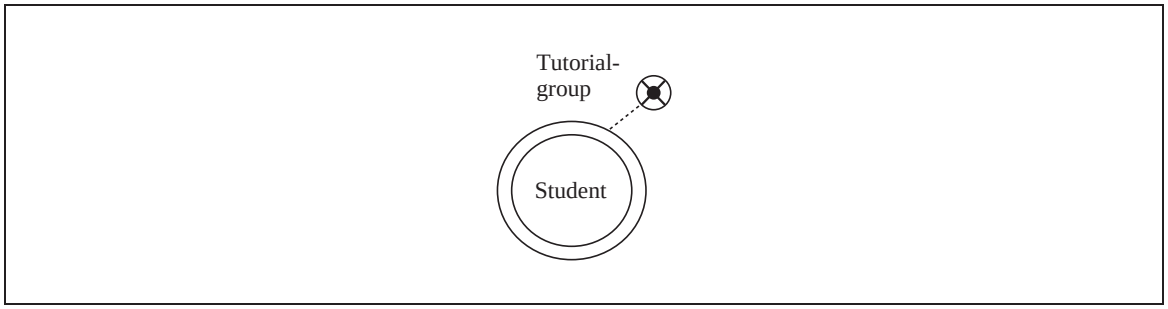

Figure 4.36: Example of partition constraint

4.7.3 Set cardinality constraint

It may be the case, that instances of power types have a minimum or maximum number of elements. A *set cardinality constraint* ensures that the instances of a power type (or a subtype of a power type) consist of at least n and at most m elements (n and m are arbitrary natural numbers). Graphically, a set cardinality constraint is depicted as an occurrence frequency constraint connected to the object type involved. For an example consider figure 4.37: *Tutorial-groups* should consist of at least 2 and at most 3 *Students*.

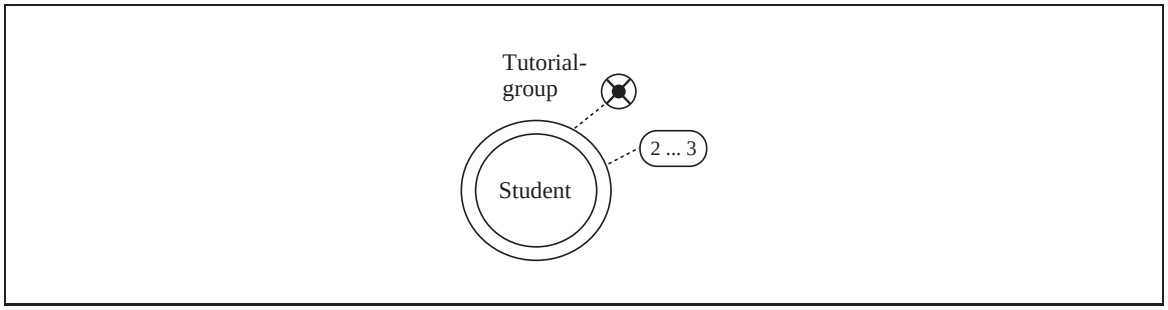

Figure 4.37: Example of set cardinality constraint

Formally, a set cardinality constraint $\mathsf{cardinality}(g, n, m)$ consists of an object type g with $\sqcap(g) \in \mathcal{G}$ and two natural numbers ($n, m \in \mathbb{N}$). A population Pop satisfies a set cardinality constraint, $\mathsf{Pop} \models \mathsf{cardinality}(g, n, m)$, iff:

$$\forall_{x \in \mathsf{Pop}(g)} [n \leq |x| \leq m]$$

4.7.4 Membership constraint

A *membership constraint* is used to express that instances should be element of other instances. Consider for example figure 4.38, which is an extension of the convoy example of section 2.1.4. A convoy can have a

flagship. The membership constraint expresses that the flagship of a convoy should be part of that convoy.

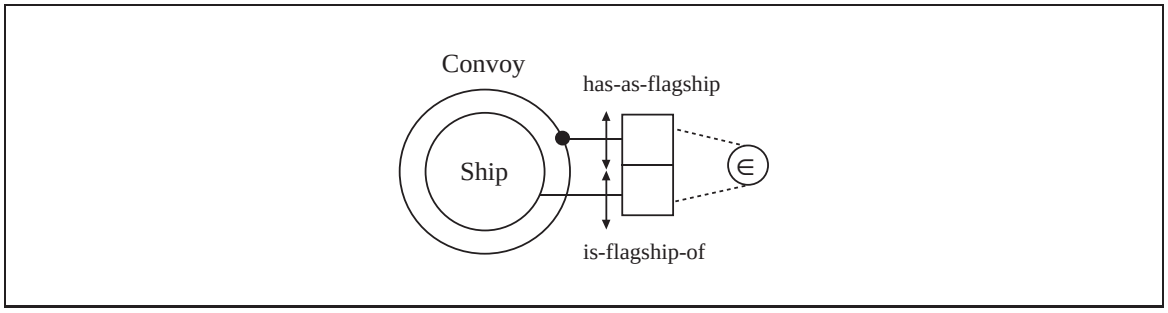

Figure 4.38: Example of membership constraint

A population satisfies a membership constraint, $\mathsf{Pop} \models \mathsf{member}(p, q)$, where $p, q \in \mathcal{P}$, iff:

$$t \in \mathsf{Val}[\![\xi(\{p, q\})]\!]\,(\mathsf{Pop}) \Rightarrow t(p) \in t(q)$$

4.8 Specialisation constraints

In this section two constraint types for subtypes are defined.

Let $\phi \subseteq \mathcal{E} \wedge \phi \neq \varnothing$ be a *family* of entity types, i.e.:

$$\forall_{x,y \in \phi}\,[\sqcap(x) = \sqcap(y)]$$

The lowest common ancestor of ϕ is denoted as $\sqcap(\phi)$, and is defined by:

1. $\forall_{x \in \phi}\left[x\,\mathsf{Spec}^{+}\,\sqcap(\phi)\right]$
2. $\forall_{x \in \phi}\left[x\,\mathsf{Spec}^{+}\,y\right] \Rightarrow \sqcap(\phi)\,\mathsf{Spec}^{*}\,y$

A *subtype exclusion constraint* over a family of entity types ϕ expresses that the populations of the involved entity types are disjunct. A population Pop satisfies the subtype exclusion constraint ϕ, $\mathsf{Pop} \models \mathsf{sub\_exclusion}(\phi)$, iff:

$$\forall_{x,y \in \phi}\,[x \neq y \Rightarrow \mathsf{Pop}(x) \cap \mathsf{Pop}(y) = \varnothing]$$

The subtype exclusion constraint has the same graphical representation as the exclusion constraint for fact types. It is connected to the specialisation arrows that have an entity type from ϕ as source.

A *total subtype constraint* over a family of entity types ϕ expresses that the union of the populations of the involved entity types should be equal to the population of the lowest common ancestor of ϕ. Naturally, this requires that the lowest common ancestor is defined for ϕ. A population Pop satisfies the total subtype constraint ϕ, $\mathsf{Pop} \models \mathsf{sub\_total}(\phi)$, iff:

$$\mathsf{Pop}(\sqcap(\phi)) = \bigcup_{x \in \phi} \mathsf{Pop}(x)$$

The total subtype constraint has the same graphical representation as the total role constraint. It is connected to the specialisation arrows that have an entity type from ϕ as source.

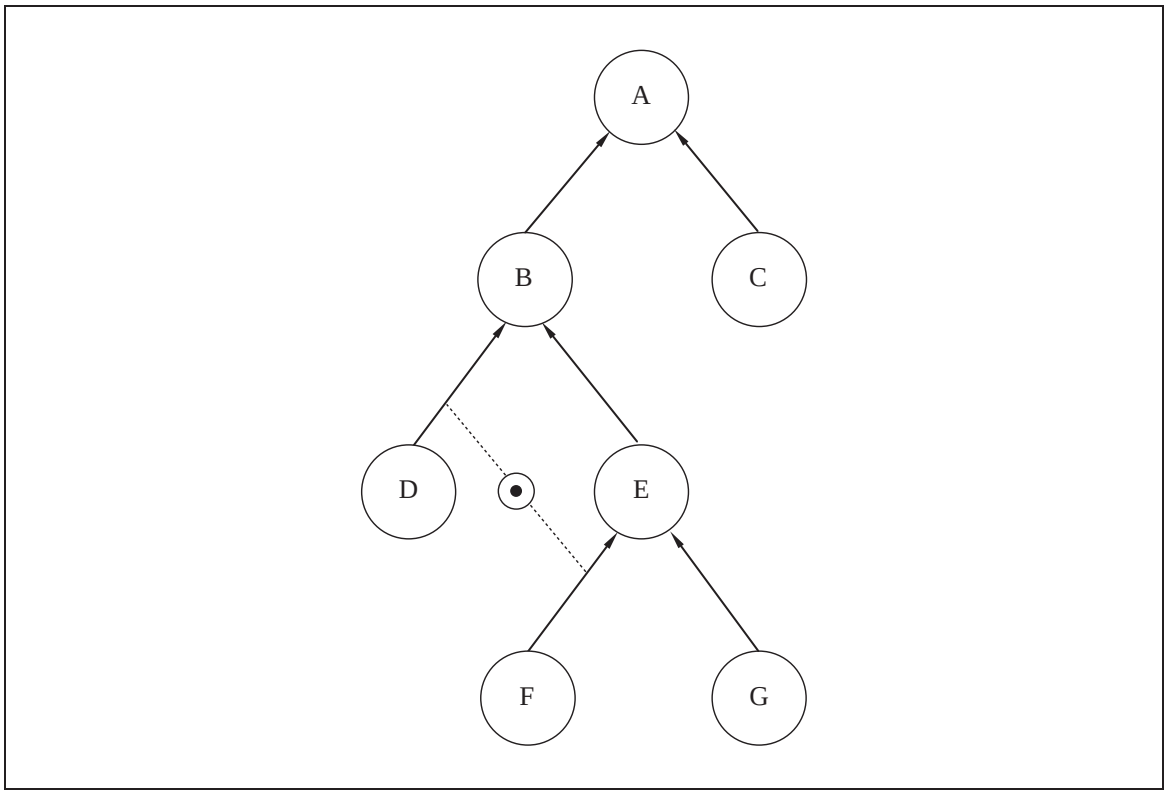

Figure 4.39: Example of a total subtype constraint

Example 4.7

In figure 4.39, the total subtype constraint $\{D, F\}$ is specified. The lowest common ancestor of this family of entity types is B. Consequently, the semantics of this constraint is:

$$\mathsf{Pop}(B) = \mathsf{Pop}(D) \cup \mathsf{Pop}(F)$$

□

Subtype exclusion constraints and total subtype constraints may be combined to express subtype partition constraints. An example of a subtype partition constraint can be found in figure 2.24.

4.9 Subtype defining rules

In this section, subtype defining rules are considered. Informally, a subtype defining rule, for a specific subtype, is a decision criterion that can be used to determine whether an instance of its pater familias is also an instance of that subtype. Subtype defining rules are expressed in the relational algebra of section 4.11. Although this algebra is not very powerful, many frequently occurring kinds of subtype defining rules can be expressed. A more powerful and comprehensible language for the expression of subtype defining rules can be defined using LISA-D ([39], [35]).

A subtype defining rule is considered to be a constraint, $\mathsf{SubRule}(s, r)$, where s is a subtype ($\mathsf{spec}(s)$), and r a relational expression having a singleton schema $\{p\}$ with $s\,\mathsf{Spec}^{+}\,\mathsf{Base}(p)$. This subtype defining rule

is satisfied iff:

$$\mathsf{Pop}(s) = \bigcap_{t,s\ \mathsf{Spec}\ t} \mathsf{Pop}(t) \cap \mathsf{Val}[\![\theta_p(r)]\!]\,(\mathsf{Pop})$$

For each subtype a subtype defining rule is required. More than one subtype defining rule is not allowed, as this may lead to contradictions. The unique subtype defining rule for subtype s is denoted as $\mathsf{SubRule}(s)$.

Cyclic subtype defining rules are not allowed, for example, a subtype defining rule for a subtype s that depends on a subtype of s. This can be formalised by introducing $\mathsf{Objects}(r)$ as the set of object types needed to evaluate r and a dependency notion $\rhd$.

Informally, $a \rhd b$ is to be interpreted as "a depends on b", i.e. object type a can only be populated if object type b can be populated.

In these rules, a predicate error indicates whether an object type depends on itself or on an object type that depends on itself. To avoid cyclic subtype dependencies, subtypes should not satisfy this predicate:

$$\forall_{x \in \mathcal{E}, \mathsf{spec}(x)}\,[\neg\, \mathsf{error}(x)]$$

In section 5.4, it will be proved that object types (not only subtypes) that satisfy the predicate error are not structurally identifiable.

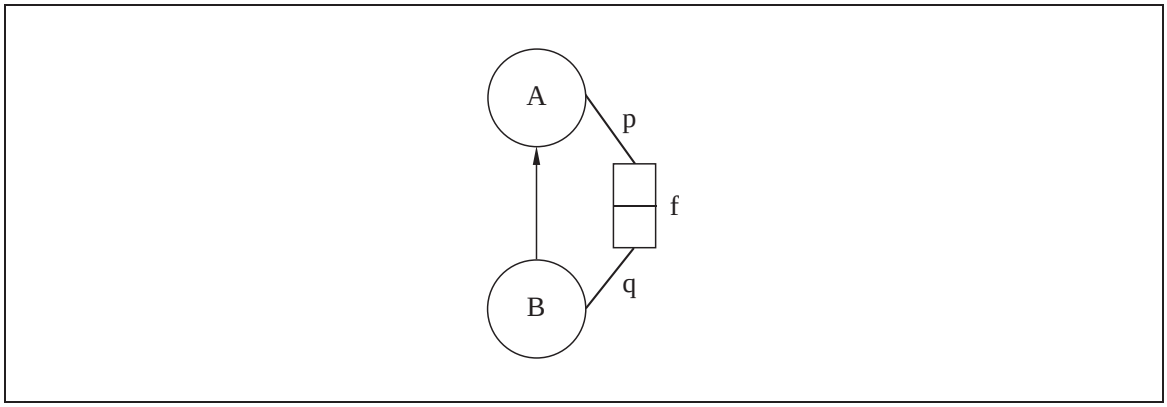

Figure 4.40: A simple subtype hierarchy

Example 4.8

Consider the subtype hierarchy in figure 4.40. The subtype defining rule $\pi_p(f)$ is not valid for subtype B, as $B \in \mathsf{Objects}(\pi_p(f))$ and therefore, $B \rhd B$ and $\mathsf{error}(B)$. □

Example 4.9

Consider the schema of figure 4.41. In this figure the subtype defining rule $\pi_t(h)$ for B would not be valid. As $E \in \mathsf{Objects}(\pi_t(h))$, $B \rhd E$ (rule D5*). According to rule* D6*, $E \rhd B$. Application of the rules* D1 *and* D2*, then yields $\mathsf{error}(B)$.* □

4.10 Schema type constraints

So far, constraints specified for object types in the decomposition of a schema type, had to be satisfied by the global population of those object types. It is possible to restrict these constraints to the instances of the schema type as these instances are populations themselves. This means that each instance of the

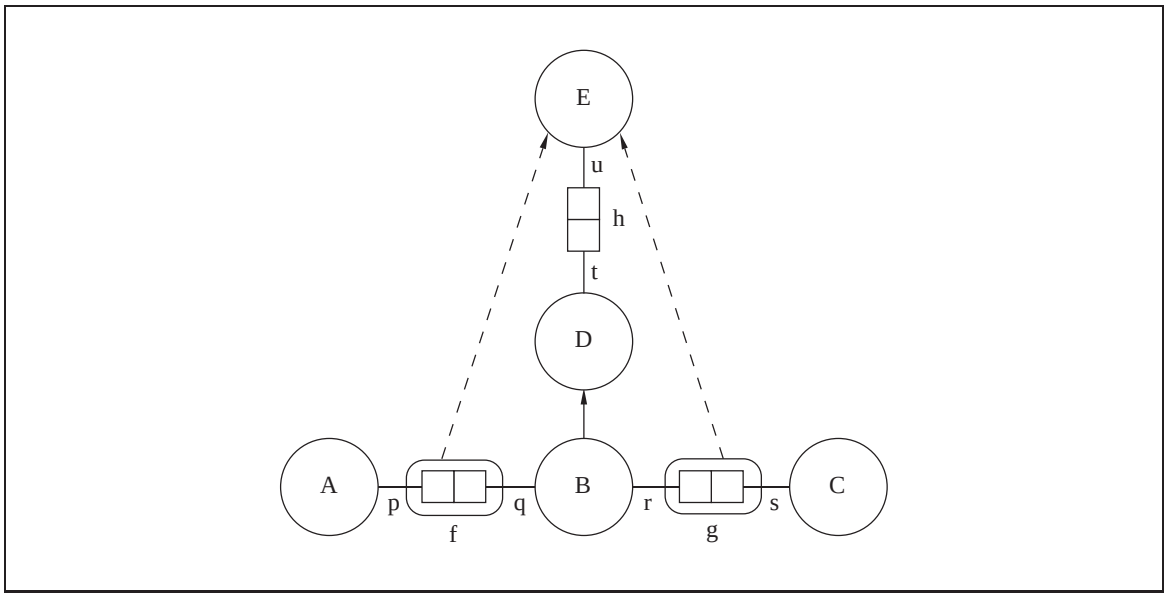

Figure 4.41: Another subtype hierarchy

involved schema type should satisfy the constraint. Formally, these types of constraints consist of a schema type $c \in \mathcal{C}$ and a constraint $r \in \mathbf{\Gamma}(\mathcal{I}_c)$, specified for the information structure associated with schema type c. A population Pop satisfies such a constraint, $\mathsf{Pop} \models \mathsf{schema\_constr}(c, r)$, iff:

$$\forall_{x \in \mathsf{Pop}(c)} [x \models r]$$

4.11 Relational algebra

The relational algebra has been introduced as a retrieval language for the Relational Model. In this section, a relational algebra is introduced, as a means for describing so-called derived fact types. Derived fact types are described by a relational expression, from which the type can be derived, and the population, on the basis of a population of the schema. The type of a derived fact type is referred to as its *schema*, which is a set of predicators.

The set of relational expressions that can be constructed for an information structure $\mathcal{I}$ is denoted as $\boldsymbol{\mathcal{R}}(\mathcal{I})$. This set is inductively defined, giving the opportunity to use structural induction to prove a property for all its members.

4.11.1 Relational expressions

A relational expression is either a fact type or a relational operator applied to one (or more) relational expression(s). If it is a fact type, say f, the schema of this expression is the set of predicators in f ($\mathsf{Schema}(f) = f$). Otherwise, the schema can be derived from the schemata of its constituing relational expressions. The population of expression r is given by the operator Val, which operates on a population and yields a set of tuples, each functions from $\mathsf{Schema}(r)$ to Ω. Naturally, for each fact type f: $\mathsf{Val}[\![f]\!](\mathsf{Pop}) = \mathsf{Pop}(f)$. In each definition of $\mathsf{Val}[\![r]\!](\mathsf{Pop})$, with r a relational expression, it is required, but not explicitly stated, that its instances are functions from its schema to values from Ω:

$$t \in \mathsf{Val}[\![r]\!](\mathsf{Pop}) \Rightarrow t : \mathsf{Schema}(r) \rightarrow \Omega$$

The basic relational operators are union, difference, join, projection, selection, extension, unnest and strong unnest. First, union and difference are defined. Suppose r and s are compatible relational expressions, i.e. $\mathsf{Schema}(r) = \mathsf{Schema}(s)$, then the union $r \cup s$ and the difference $r \setminus s$ both are relational expressions, having schema $\mathsf{Schema}(r)$, and having the following populations:

- $\mathsf{Val}[\![r \cup s]\!]\,(\mathsf{Pop}) = \mathsf{Val}[\![r]\!]\,(\mathsf{Pop}) \cup \mathsf{Val}[\![s]\!]\,(\mathsf{Pop})$
- $\mathsf{Val}[\![r \setminus s]\!]\,(\mathsf{Pop}) = \mathsf{Val}[\![r]\!]\,(\mathsf{Pop}) \setminus \mathsf{Val}[\![s]\!]\,(\mathsf{Pop})$

If r and s are relational expressions, then the join $r \bowtie s$ is a relational expression, defined by:

1. $\mathsf{Schema}(r \bowtie s) = \mathsf{Schema}(r) \cup \mathsf{Schema}(s)$
2. $\mathsf{Val}[\![r \bowtie s]\!]\,(\mathsf{Pop}) = \left\{ t \;\middle|\; \mathsf{Restrict}\, t \mathsf{Schema}(r) \in \mathsf{Val}[\![r]\!]\,(\mathsf{Pop}) \wedge \mathsf{Restrict}\, t \mathsf{Schema}(s) \in \mathsf{Val}[\![s]\!]\,(\mathsf{Pop}) \right\}$

From this definition, it is clear that if the intersection between $\mathsf{Schema}(r)$ and $\mathsf{Schema}(s)$ is empty, the join behaves like the cartesian product. The function $\mathsf{Restrict}\, f A'$ is the function f restricted to a subdomain $A' \subseteq \mathsf{dom}(f)$. This function is defined by:

$$\mathsf{Restrict}\, f A' = \left\{ \langle a, b \rangle \in f \;\middle|\; a \in A' \right\}$$

Example 4.10

Consider the information structure of figure 2.1 and its sample population presented in section 2.2.3. For the relational expression $f \bowtie i$ we have:

$$\begin{array}{rcl} \mathsf{Schema}(f \bowtie i) & = & \{p, q, w, x\} \\ \mathsf{Val}[\![f \bowtie i]\!]\,(\mathsf{Pop}) & = & \{\{p : b_1, q : a_1, w : b_1, x : 17\}, \{p : b_1, q : a_2, w : b_1, x : 17\}\} \end{array}$$

□

Suppose r is a relational expression, $p_1, \ldots, p_n$ and $q_1, \ldots, q_n$ are predicators, all $q_1, \ldots, q_n$ different and $p_1, \ldots, p_n \in \mathsf{Schema}(r)$, then the projection $\pi_{q_1:p_1,\ldots,q_n:p_n}(r)$ is a relational expression, defined by:

1. $\mathsf{Schema}(\pi_{q_1:p_1,\ldots,q_n:p_n}(r)) = \{q_1, \ldots, q_n\}$
2. $\mathsf{Val}[\![\pi_{q_1:p_1,\ldots,q_n:p_n}(r)]\!]\,(\mathsf{Pop}) = \left\{ t \;\middle|\; \exists_{s \in \mathsf{Val}[\![r]\!]\,(\mathsf{Pop})} \forall_{1 \leq i \leq n} \left[t(q_i) = s(p_i) \right] \right\}$

This definition is in line with [61]. Note that the projection can be used to "rename" a predicator. Furthermore, the projection operator can be used to extend relations with new predicators. We will use $\pi_{p_1,\ldots,p_n}(r)$ as a shorthand for $\pi_{p_1:p_1,\ldots,p_n:p_n}(r)$. If τ is a set of predicators, then the notation $\pi_\tau(r)$ is also used. This notation is correct, since the order of the predicators used for a projection is not important (a result of the mapping oriented approach).

Example 4.11

In the context of the same information structure and population of the previous example, the relational expression $\pi_{y:q,z:q}(f)$ has the following schema and population:

$$\begin{array}{rcl} \mathsf{Schema}(\pi_{y:q,z:q}(f)) & = & \{y, z\} \\ \mathsf{Val}[\![\pi_{y:q,z:q}(f)]\!]\,(\mathsf{Pop}) & = & \{\{y : a_1, z : a_1\}, \{y : a_2, z : a_2\}\} \end{array}$$

□

For the selection operator, the syntax and semantics of selection formulas need to be defined.

Definition 4.1

The set Ψ containing the selection formulas is inductively defined as the smallest set satisfying:

1. $p, q \in \mathcal{P}' \Rightarrow p = q \in \Psi$.
2. $p \in \mathcal{P}'$, $c \in \bigcup D \Rightarrow p = c \in \Psi$.
3. $f_1, f_2 \in \Psi \Rightarrow f_1 \wedge f_2 \in \Psi$.
4. $f_1 \in \Psi \Rightarrow \neg f_1 \in \Psi$.

Recall that $\mathcal{P}'$ is the set of predicators, which can occur in information structures. □

The following definition defines when a tuple occurring in the population of a relational expression satisfies a selection formula. This definition makes use of the inductive definition of Ψ.

Definition 4.2

If t is a partial function from $\mathcal{P}'$ to Ω, then

1. $t \models p = q$ *if and only if* $t(p) = t(q)$ *and* $t(p)\downarrow$ *and* $t(q)\downarrow$.
2. $t \models p = c$ *if and only if* $t(p) = c$ *and* $t(p)\downarrow$.
3. $t \models f_1 \wedge f_2$ *if and only if* $t \models f_1$ *and* $t \models f_2$.
4. $t \models \neg f_1$ *if and only if not* $t \models f_1$.

□

If r is a relational expression, and F a selection formula ($F \in \Psi$), then the selection $\sigma_F(r)$ is also a relational expression, according to:

1. $\mathsf{Schema}(\sigma_F(r)) = \mathsf{Schema}(r)$
2. $\mathsf{Val}[\![\sigma_F(r)]\!]\,(\mathsf{Pop}) = \left\{ t \in \mathsf{Val}[\![r]\!]\,(\mathsf{Pop}) \;\middle|\; t \models F \right\}$

Example 4.12

In the context of the information structure and the population of the previous examples, the relational expression $\sigma_{t=v}(\pi_{s,t}(g) \bowtie \pi_v(h))$ has the following schema and population:

$$\begin{aligned} \mathsf{Schema}(\sigma_{t=v}(\pi_{s,t}(g) \bowtie \pi_v(h))) &= \{s, t, v\} \\ \mathsf{Val}[\![\sigma_{t=v}(\pi_{s,t}(g) \bowtie \pi_v(h))]\!]\,(\mathsf{Pop}) &= \{\{s : b_1, t : g_1, v : g_1\}\} \end{aligned}$$

Note that this relational expression is equivalent to the relational expression

$$\pi_{s:s,t:t,v:t}(g) \bowtie \pi_v(h)$$

□

The extension operator χ extends a relational expression r with a new predicator a ($a \notin \mathsf{Schema}(r)$), counting the number of tuples with an equal τ-value (where $\tau \subseteq \mathsf{Schema}(r)$), in the following way:

1. $\mathsf{Schema}(\chi(r, \tau, a)) = \mathsf{Schema}(r) \cup \{a\}$
2. $\mathsf{Val}[\![\chi(r, \tau, a)]\!]\,(\mathsf{Pop}) =$

   $\left\{ t \;\middle|\; \mathsf{Restrict}\, t \mathsf{Schema}(r) \in \mathsf{Val}[\![r]\!]\,(\mathsf{Pop}) \wedge t(a) = \left| \left\{ s \in \mathsf{Val}[\![r]\!]\,(\mathsf{Pop}) \;\middle|\; \mathsf{Restrict}\, t\tau = \mathsf{Restrict}\, s\tau \right\} \right| \right\}$

This operator is a restricted version of the extension operator introduced in [11], which was introduced to handle GROUP BY queries in SQL.

Example 4.13

Suppose that in the information structure of figure 2.1:

$$\mathsf{Pop}(f) = \{\{p : b_1, q : a_1\}, \{p : b_1, q : a_2\}, \{p : b_2, q : a_1\}\}$$

The relational expression $\chi(f, \{p\}, x)$ then has the following schema and population:

$$\begin{aligned} \mathsf{Schema}(\chi(f, \{p\}, x)) &= \{p, q, x\} \\ \mathsf{Val}[\![\chi(f, \{p\}, x)]\!](\mathsf{Pop}) &= \left\{ \begin{array}{l} \{p : b_1, q : a_1, x : 2\}, \\ \{p : b_1, q : a_2, x : 2\}, \\ \{p : b_2, q : a_1, x : 1\} \end{array} \right\} \end{aligned}$$

□

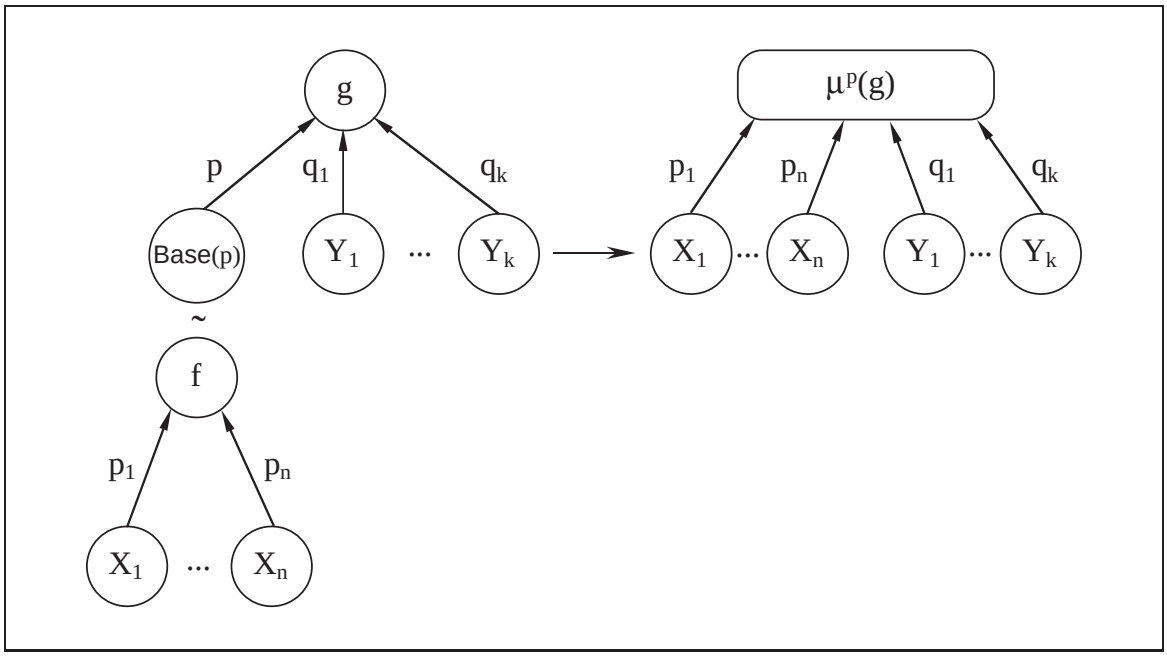

Figure 4.42: A functional view on unnesting

The unnest operator is used to flatten nested fact types, i.e. fact types that contain predicators having a base which is type related with a fact type. This operator has been defined for the NF^2 data model in [61]. For our purposes, an alternative, simplified, definition is needed. Let g be a relational expression, $p \in \mathsf{Schema}(g)$, and f a fact type such that $\mathsf{Base}(p) \sim f$. (see figure 4.42). Let $S = \mathsf{Schema}(g) \backslash \{p\}$. Then $\mu^p_f(g)$ is a relational expression, defined by:

1. $\mathsf{Schema}(\mu^p_f(g)) = f \cup S$
2. $\mathsf{Val}[\![\mu^p_f(g)]\!](\mathsf{Pop}) = \left\{ t \cup \mathsf{Restrict}\, sS \;\middle|\; t \in \mathsf{Val}[\![f]\!](\mathsf{Pop}) \wedge s \in \mathsf{Val}[\![g]\!](\mathsf{Pop}) \wedge s(p) = t \right\}$

In this definition the expression $t \cup \mathsf{Restrict}\, sS$ relies on the introduction of a tuple as a mapping, while a mapping is mathematically defined as a set of pairs. Note that if also another object type x exists, which is type related with $\mathsf{Base}(p)$, its values are simply filtered out. If fact type f is uniquely determined, one can write $\mu^p(g)$ instead of $\mu^p_f(g)$.

Example 4.14

Suppose that in the information structure of figure 2.1:

$$\mathsf{Pop}(g) = \{\{r : \{p : b_1, q : a_1\}, s : b_2, t : b_3\}, \{r : \{p : b_2, q : a_2\}, s : b_3, t : b_2\}\},$$

then the relational expression $\mu^r_f(g)$ has the following schema and population:

$$\begin{array}{rcl} \mathsf{Schema}(\mu^r_f(g)) & = & \{p, q, s, t\} \\ \mathsf{Val}[\![\mu^r_f(g)]\!]\,(\mathsf{Pop}) & = & \{\{p : b_1, q : a_1, s : b_2, t : b_3\}, \{p : b_2, q : a_2, s : b_3, t : b_2\}\} \end{array}$$

□

The relational expression $\mu^p_f(g)$ is not defined, if $S \cap \mathsf{Schema}(f) \neq \varnothing$. In this case, it cannot be guaranteed that the instances of $\mathsf{Val}[\![\mu^p_f(g)]\!]\,(\mathsf{Pop})$ are functions, they may be relations. To avoid this, the strong unnest operator has to be applied. This operator ensures that the instances in the resulting population are functions.

Let g be a relational expression, $p \in \mathsf{Schema}(g)$, and f a fact type such that $\mathsf{Base}(p) \sim f$. Let $S = \mathsf{Schema}(g) \backslash \{p\}$ and $T = f \cap S$, then $\eta^p_f(g)$ is a relational expression defined by:

1. $\mathsf{Schema}(\eta^p_f(g)) = f \cup S$
2. $\mathsf{Val}[\![\eta^p_f(g)]\!]\,(\mathsf{Pop}) =$

$$\left\{ t \cup \mathsf{Restrict}\, sS \;\middle|\; t \in \mathsf{Val}[\![f]\!]\,(\mathsf{Pop}) \wedge s \in \mathsf{Val}[\![g]\!]\,(\mathsf{Pop}) \wedge s(p) = t \wedge \mathsf{Restrict}\, sT = \mathsf{Restrict}\, s(p)T \right\}$$

Note that in the case that $T = \varnothing$, the strong unnest operator behaves like the unnest operator. For this operator, we will also omit fact type f if this fact type can be determined uniquely.

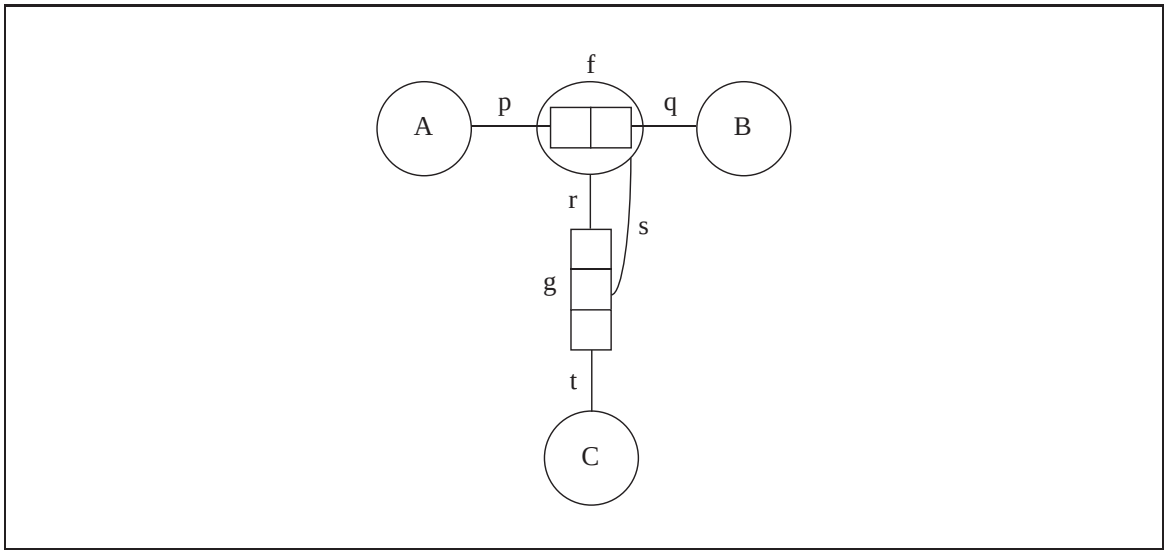

Figure 4.43: Example information structure

Example 4.15

Consider the information structure of figure 4.43. Suppose that we want to unnest g via r, and subsequently via s. In this case the values in the population of $\mu^s(\mu^r(g))$ are in general, depending

on the population of g, not functions, but relations. This is a result of the fact that p and q are part of $\mathsf{Schema}(\mu^r(g))$. *Strong unnesting however, avoids this problem. Suppose that the schema of figure 4.43 has as population of fact types f and g:*

$$\begin{aligned}\mathsf{Pop}(f) &= \{\{p:a_1, q:b_1\}, \{p:a_1, q:b_2\}\}\\ \mathsf{Pop}(g) &= \left\{\begin{matrix}\{r:\{p:a_1,q:b_1\}, s:\{p:a_1,q:b_1\}, t:c_1\},\\ \{r:\{p:a_1,q:b_1\}, s:\{p:a_1,q:b_2\}, t:c_2\},\\ \{r:\{p:a_1,q:b_2\}, s:\{p:a_1,q:b_2\}, t:c_3\}\end{matrix}\right\},\end{aligned}$$

then $\mathsf{Val}[\![\eta^s(\eta^r(g)))]\!](\mathsf{Pop}) = \big\{\{p:a_1, q:b_1, t:c_1\}, \{p:a_1, q:b_2, t:c_3\}\big\}$. □

4.11.2 Boolean operations

This section is concerned with boolean expressions. If b is a boolean expression, we will define when a population Pop satisfies this expression, notation: $\mathsf{Pop} \models b$.

The test $\mathsf{IsEmpty}$ indicates whether a relational expression has an empty population:

$$\mathsf{Pop} \models \mathsf{IsEmpty}(r) \text{ if and only if } \mathsf{Val}[\![r]\!](\mathsf{Pop}) = \varnothing$$

A very important notion in relational algebra theory is that of functional dependency. Let r be a relational expression and let $\sigma, \tau \subseteq \mathsf{Schema}(r)$. Then $\sigma \xrightarrow{r} \tau$ is a boolean expression which is true in a population Pop ($\mathsf{Pop} \models \sigma \xrightarrow{r} \tau$) iff in this population, the σ-part of each instance in $\mathsf{Val}[\![r]\!](\mathsf{Pop})$ uniquely determines its τ-part. In that case, τ is called functionally dependent on σ in r. Formally, $\mathsf{Pop} \models \sigma \xrightarrow{r} \tau$ iff:

$$\forall_{x,y \in \mathsf{Val}[\![r]\!](\mathsf{Pop})} [\mathsf{Restrict}\, x\sigma = \mathsf{Restrict}\, y\sigma \Rightarrow \mathsf{Restrict}\, x\tau = \mathsf{Restrict}\, y\tau]$$

Example 4.16

Suppose that in the information structure of figure 2.1, the population of fact type g is given by:

$$\mathsf{Pop}(g) = \left\{\begin{matrix}\{r:\{q:a_1,p:b_1\}, s:b_1, t:g_1\},\\ \{r:\{q:a_1,p:b_1\}, s:b_2, t:g_1\},\\ \{r:\{q:a_2,p:b_2\}, s:b_1, t:g_3\},\\ \{r:\{q:a_2,p:b_3\}, s:b_2, t:g_3\}\end{matrix}\right\}$$

In this population the boolean expression $\{r\} \xrightarrow{g} \{t\}$ is true. The boolean expression $\{s\} \xrightarrow{g} \{t\}$ however, is not true. □

Comparing relational expressions is only useful in the case of relational expressions with different schemata, and particularly if the bases of the predicators involved can be matched. First we consider the subset operator. Let r and s be relational expressions. A function ϕ between $\mathsf{Schema}(r)$ and $\mathsf{Schema}(s)$ is called a *match*, if ϕ is a bijection between $\mathsf{Schema}(r)$ and $\mathsf{Schema}(s)$, such that $\forall_{p \in \mathsf{Schema}(r)} [p \sim \phi(p)]$. In that case, $\mathsf{Pop} \models r \subseteq_\phi s$ if and only if:

$$\forall_{t \in \mathsf{Val}[\![r]\!](\mathsf{Pop})} \exists_{u \in \mathsf{Val}[\![s]\!](\mathsf{Pop})} \forall_{p \in \mathsf{Schema}(r)} [t(p) = u(\phi(p))]$$

Example 4.17

In the population of example 4.16, the boolean expression $\pi_s(g) \subseteq_{\{s:p\}} \pi_p(\mu^r(g))$ is true, while its reverse, $\pi_p(\mu^r(g)) \subseteq_{\{p:s\}} \pi_s(g)$ is not true. The bijections $\{s:p\}$ and $\{p:s\}$ are matches as $s \sim p$ in the information structure of figure 2.1. □

The equality operator can be defined using the subset operator:

$$r =_\phi s = (r \subseteq_\phi s \wedge s \subseteq_{\phi^{-1}} r)$$

The exclusion test $\otimes_\phi$ is defined by:

$$\forall_{t \in \mathsf{Val}[\![r]\!](\mathsf{Pop})} \neg \exists_{u \in \mathsf{Val}[\![s]\!](\mathsf{Pop})} \forall_{p \in \mathsf{Schema}(r)} [t(p) = u(\phi(p))]$$

Example 4.18
In the context of the information structure of figure 2.1 and its sample population of section 2.2.3, the boolean expression $\pi_v(h) \otimes_{\{v\,:\,t\}} \pi_t(g)$ *is not true.* □

4.11.3 Special operations

The range operator θ_p, where p is a predicator, coerces the population of a relational expression r, with $\mathsf{Schema}(r) = \{p\}$, to the set of values that are taken by this single predicator:

$$\mathsf{Val}[\![\theta_p(r)]\!](\mathsf{Pop}) = \{ t(p) \mid t \in \mathsf{Val}[\![r]\!](\mathsf{Pop}) \}$$

Example 4.19
In the sample population of section 2.2.3: $\mathsf{Val}[\![\theta_q(\pi_q(f))]\!](\mathsf{Pop}) = \{a_1, a_2\}$ □

The range operator is not an algebraic operator, since it might yield values from entity types or label types, which are not mappings with as domain a set of predicators (see e.g. example 4.19). The lift operator, which can be considered to be the reverse of the range operator, on the other hand, *is* an algebraic operator. This operator has as arguments a predicator and either an object type or a relational expression. The values in its result are mappings from this predicator to values occurring in the population of the involved object type or relational expression. Therefore, application of the lift operator yields a relational expression. Formally:

1. $\mathsf{Schema}(\zeta_p(r)) = \{p\}$
2. $\mathsf{Val}[\![\zeta_p(r)]\!](\mathsf{Pop}) = \{ t \mid t(p) \in \mathsf{Val}[\![r]\!](\mathsf{Pop}) \}$

Example 4.20
In the sample population of section 2.2.3: $\mathsf{Val}[\![\zeta_q(A)]\!](\mathsf{Pop}) = \{\{q : a_1\}, \{q : a_2\}\}$ □

If the instances occurring in the population of a predicator p, part of the schema of a relational expression r, can be ordered, then the extreme values can be calculated by:

$$\begin{aligned} \mathsf{Val}[\![\mathsf{min}(r,p)]\!](\mathsf{Pop}) &= \min(\mathsf{Val}[\![\theta_p(\pi_p(r))]\!](\mathsf{Pop})) \\ \mathsf{Val}[\![\mathsf{max}(r,p)]\!](\mathsf{Pop}) &= \max(\mathsf{Val}[\![\theta_p(\pi_p(r))]\!](\mathsf{Pop})) \end{aligned}$$

4.11.4 Object Interpretation Structures

Complex operations on an information structure, as needed in section 4.2 dealing with the definition of uniqueness constraints, do not always yield another information structure. In the resulting structures, roles may participate in more than one (possibly derived) fact type. *Object Interpretation Structures* can be used to visualise such complex operations on information structures. Object Interpretation Structures are sets of object types from a particular information structure and relational expressions associated with that information structure. Formally, an Object Interpretation Structure in an information structure $\mathcal{I}$ is a set $N \subseteq \mathcal{O} \cup \boldsymbol{\mathcal{R}}(\mathcal{I})$.

An Object Interpretation Structure N can be represented as a labeled directed graph, the *Object Interpretation Graph*, by choosing as nodes the elements from N and drawing edges from x to y with label p iff $p \in \mathcal{P}$ such that $p \in \mathsf{Schema}(y)$ and $\mathsf{Base}(p) = x$. This corresponds to the functional drawing style introduced for information structures in section 2.1.3. Note that this definition implies that object types, which are not fact types, are never the destination of an edge and that relational expressions, which are not fact types, are never the source of an edge.

Remark 4.1

The Object Interpretation Graph of an Object Interpretation Structure that contains relational expressions with predicators that do not occur in the information structure, e.g. predicators introduced for renaming purposes, cannot be drawn. The reason for this is that for such predicators the function Base *is undefined. For our purposes however, this does not pose any problems.* □

We will now define a number of auxiliary functions and relations for Object Interpretation Structures. For these definitions it is assumed that N is an Object Interpretation Structure in an information structure $\mathcal{I}$ and $n \in N$.

The function children applied on a node n yields the nodes that are type related with a node which is source of an edge with destination n:

$$\mathsf{children}(N, n) = \{x \in N \mid \exists_{p \in \mathsf{Schema}(n)} [\mathsf{Base}(p) \sim x]\}$$

On the basis of this function a graph can be defined with as nodes the elements from N and arrows from x to y iff $x \in \mathsf{children}(N, y)$. The Object Interpretation Structure N is termed *acyclic* if this graph does not contain directed cycles.

Leaves are nodes without children: $\mathsf{leaf}(N, n)$ if and only if $\mathsf{children}(N, n) = \varnothing$. The depth of a node is given by the function depth defined by:

$$\mathsf{depth}(N, n) = \textbf{if } \mathsf{leaf}(N, n) \textbf{ then } 0 \textbf{ else } 1 + \max_{x \in \mathsf{children}(N,n)} \mathsf{depth}(N, x) \textbf{ fi}$$

This function is only defined when N is acyclic. Tops are nodes that do not have parents: $\mathsf{top}(N, n) = \neg\exists_{x \in N} [n \in \mathsf{children}(N, x)]$. Superfluous nodes are top nodes that have a schema that is a proper subset of the schema of another node or top nodes that are also leaf nodes:

$$\mathsf{superfluous}(N, n) = \mathsf{top}(N, n) \wedge (\exists_{x \in N} [\mathsf{Schema}(n) \subset \mathsf{Schema}(x)] \vee \mathsf{leaf}(N, n))$$

The superfluous nodes in an Object Interpretation Structure can be removed by the operation reduce:

$$\mathsf{reduce}(N) = N \backslash \{n \in N \mid \mathsf{superfluous}(N, n)\}$$

A node n in an Object Interpretation Structure N can be replaced by a node x as follows:

$$N\,[x \leftarrow n] = \{x\} \cup N \backslash \{n\}$$

Let $\mathcal{I}$ be an information structure and τ be a set of predicators from this information structure ($\tau \subseteq \mathcal{P}$). The *relevant* Object Interpretation Structure with respect to $\mathcal{I}$ and τ, $\mathsf{OIS}(\mathcal{I}, \tau)$, is inductively defined as the smallest set satisfying:

1. $p \in \tau \Rightarrow \mathsf{Base}(p) \in \mathsf{OIS}(\mathcal{I}, \tau) \wedge \mathsf{Fact}(p) \in \mathsf{OIS}(\mathcal{I}, \tau)$
2. $\mathsf{Fact}(p) \in \mathsf{OIS}(\mathcal{I}, \tau) \wedge \exists_{q \in \mathsf{Fact}(p)} [\mathsf{Base}(q) \in \mathsf{OIS}(\mathcal{I}, \tau)] \Rightarrow \mathsf{Base}(p) \in \mathsf{OIS}(\mathcal{I}, \tau)$
3. $x, y \in \mathsf{OIS}(\mathcal{I}, \tau) \wedge \mathsf{path}(x, z) \wedge \mathsf{path}(z, y) \Rightarrow z \in \mathsf{OIS}(\mathcal{I}, \tau)$

where $\mathsf{path}(x, y) = x \sim y \vee \exists_{p \in \mathcal{P}} [\mathsf{Base}(p) \sim x \wedge \mathsf{path}(\mathsf{Fact}(p), y)]$. Therefore, a relevant Object Interpretation Structure does not contain relational expressions other than fact types.

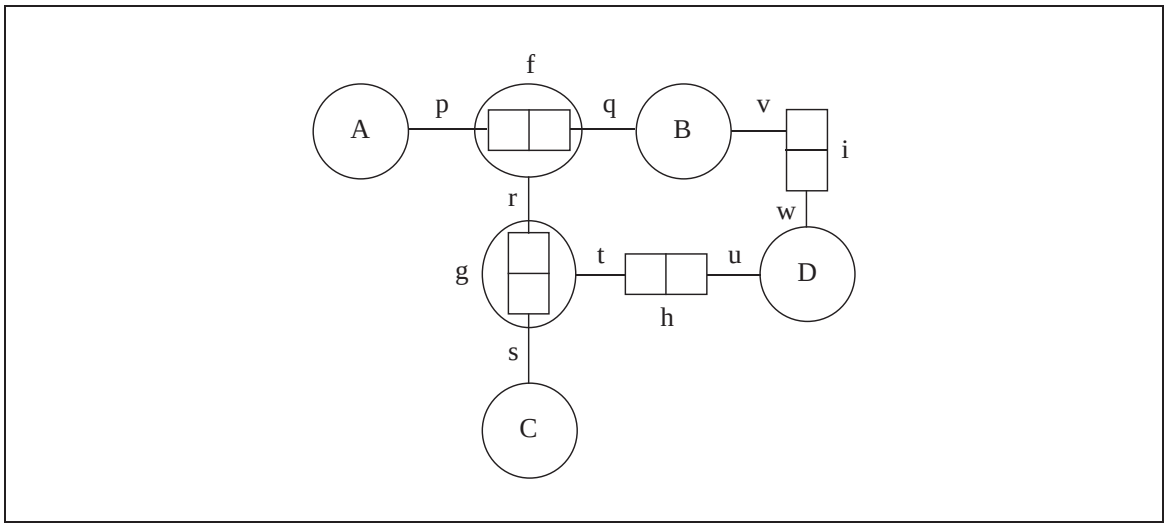

Figure 4.44: Example information structure

Example 4.21

The relevant Object Interpretation Structure with respect to the information structure of figure 4.44 and the set of predicators $\{u, q\}$ is:

$$\mathsf{OIS}(\mathcal{I}, \{u, q\}) = \{A, B, C, D, f, g, h\}$$

□

Chapter 5

Identification and verification

5.1 Introduction

This chapter focuses on the issue of identification and presents basic theoretical results regarding complexity of verification and expressive power.

On the one hand *identification* involves determining whether each entity can be represented in terms of labels, on the other hand identification involves determining whether substructures exist that can only be populated with certain peculiar populations (for example, using instances that do not have a finite representation). Sections 5.2 and 5.3 address the issue of identification.

While identification deals with the populatability of information structures, *verification* deals with the populatability of schemata (section 5.5). As a consequence of constraint contradictions, schemata may be only partly populatable. Various types of constraint contradictions exist. These various types can be characterised by schema properties. Section 5.6 deals with the complexity of the verification of two important schema properties.

Sections 5.7, 5.8 and 5.9 each deal with *expressive power*. In section 5.7 we consider the problem of (meta)modelling a data modelling technique in terms of that same technique. Section 5.8 is concerned with the relation between the axioms of set theory and data models. In section 5.9 a translation of context-free grammars to data models is given. This demonstrates that the expressive power of context-free grammars can be embedded in data models. From a practical point of view this is relevant in the context of multimedia and office automation where document structures are usually described by means of context-free grammars (in the style of SGML [64], [45] or ODA [46], [44]).

5.2 Weak identification

5.2.1 Introduction to weak identification

Labels can be represented directly, while entities depend for their representation on labels. If two different entities have exactly the same properties, and are therefore connected (directly or indirectly) to the same labels, they cannot be distinguished. A population in which entities with the same properties are equal is called *weakly identified*. Weak identification is the topic of section 5.2.2.

Weak identification is a property of populations. *Structural identification* or *strong identification* is a property of schemata and guarantees that *each* population is weakly identified. Furthermore, an information structure that is structurally identifiable does not have substructures that can only be populated with certain "peculiar" populations (a precise characterisation is provided in section 5.4). The population of

such substructures could, for example, require instances that do not have a finite representation. These instances have been excluded from populations in section 2.2.3. Structural identification therefore acts as a well-formedness rule for data schemata. In section 5.3, structural identification is defined, while in section 5.4 theoretical results with respect to structural identification are formulated and proved.

In fact-oriented data models, identification of objects is based on the properties of those objects. In object-oriented data models, identification is often based on artificial identifiers, such as system generated id-codes. In our view, identification in terms of artificial id-codes can not be considered conceptual.

5.2.2 Definition of weak identification

Labels are considered to be representable directly. Consequently, labels are identified by themselves. Entities on the other hand can only be represented by their properties. Therefore, entities with the same properties are not distinguishable. The properties of entities are recorded by the facts in which they participate.

A population Pop of an information structure $\mathcal{I}$ is called weakly identified, $\mathsf{WeakId}(\mathcal{I}, \mathsf{Pop})$, if and only if all entities in Pop which have the same properties are equal:

$$\forall_{e \in \mathcal{Q}} \forall_{x,y \in \mathsf{Pop}(e)} \left[\mathsf{IdProp}(x, y) \Rightarrow x = y \right]$$

where $\mathsf{IdProp}(x, y)$ is defined as:

$$\forall_{f \in \mathcal{F}} \forall_{t \in \mathsf{Pop}(f)} \left[t\,[x := y] \in \mathsf{Pop}(f) \wedge t\,[y := x] \in \mathsf{Pop}(f) \right]$$

where $t\,[x := y]$ is tuple t where each occurrence of x is substituted by an occurrence of y:

$$t\,[x := y] = \lambda p \in f \textbf{.if } t(p) = x \textbf{ then } y \textbf{ else } t(p) \textbf{ fi}$$

The lambda notation used in this definition is borrowed from the lambda calculus, see e.g. [3].

Weak identification is typical for systems that deal with complete knowledge only. Weak identification guarantees that no naming conflicts for objects can occur. It does not ensure, however, that every object has a name.

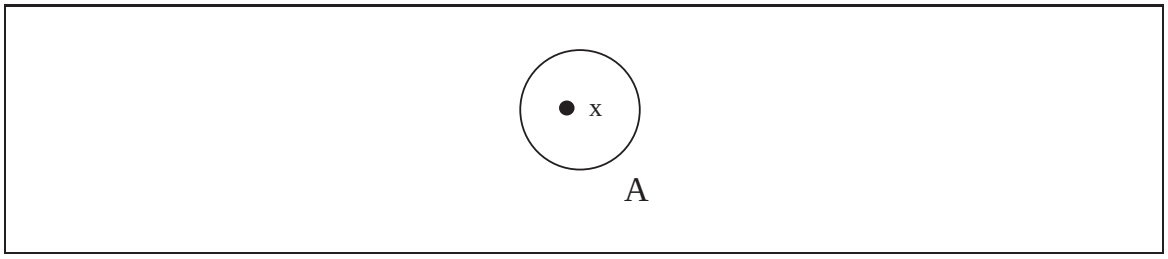

Figure 5.1: Population which is weakly identified

Example 5.1

The information structure diagram of figure 5.1 consists of only one entity type. Its population consists of one entity x. Consequently, this population is weakly identified. Note that x is an anonymous object, it does not have a name. □

Example 5.2

The population in figure 5.2 is not weakly identified. Entities x and y have exactly the same properties (in fact they do not have any properties). □

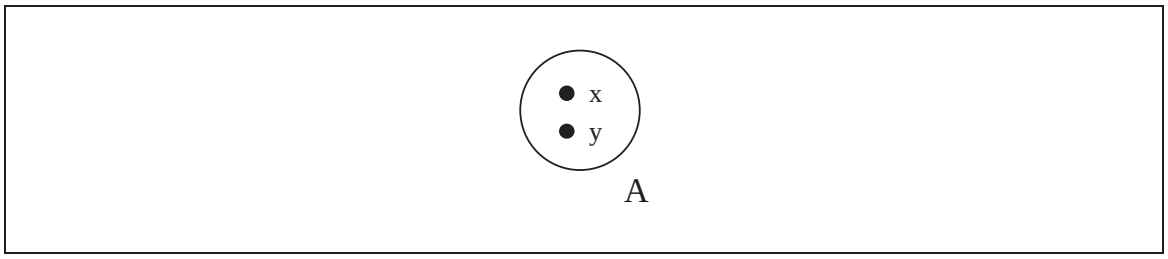

Figure 5.2: Population which is not weakly identified

5.3 Structural identification

Weak identification is an important property, as it ensures that all objects can be addressed uniquely. In this section, it is considered how this property can be guaranteed from constraints in a schema.

A schema Σ is called *structurally identifiable*, $\mathsf{StructId}(\Sigma)$, iff:

1. Each label type occurs in some total role constraint:

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

   The motivation behind this rule is to ensure the absence of unused labels.

2. All object types can be identified:

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

The predicate $\mathsf{Identifiable}$ is defined using derivation rules. An object type is identifiable if and only if this can be proved from these derivation rules.

5.3.1 Identification of label types

Label types (except the index type I) are structurally identifiable.

[IDT1] $x \in \mathcal{L} \backslash \{\mathrm{I}\} \vdash \mathsf{Identifiable}(x)$

The index type I (only present if there are sequence types) is structurally identifiable if a sequence type exists that is structurally identifiable.

[IDT2] $\exists_{x \in \mathcal{S}} \left[\mathsf{Identifiable}(x) \right] \vdash \mathsf{Identifiable}(\mathrm{I})$

The reason for this exception is the fact that the index type is the only label type that has a derivable population. The index type depends for its population on the population of sequence types.

5.3.2 Identification of root entity types

A root entity type is an entity type that is neither generalised, nor a subtype. We have the following rule:

[IDT3] $\mathsf{Root}(e) \wedge \exists_{\tau \subseteq \mathcal{P}} \left[\mathsf{Identification}(e, \tau) \right] \vdash \mathsf{Identifiable}(e)$

In this derivation rule $\mathsf{Identification}(e, \tau)$ is defined as follows:

1. $\mathsf{unique}(\tau) \in \mathcal{R}$

   The set of predicators τ is to serve as an identification for root entity type e. Instances of e can then be denoted by means of their specific combination of τ values. To this end, the combination of τ values should be unique for every instance of e. This can be guaranteed if τ is a uniqueness constraint.

2. $\forall_{p \in \tau} \exists!_{q \in \mathsf{Fact}(p)} [q \notin \tau]$

   For each predicator p in τ there should be exactly one predicator q in the same fact type not in τ. This unique predicator is denoted as $\mathsf{co}(\tau, p)$. The base of this predicator should be root entity type e: $\mathsf{Base}(\mathsf{co}(\tau, p)) = e$.

3. $\forall_{p \in \tau} [\mathsf{unique}(\{\mathsf{co}(\tau, p)\}) \in \mathcal{R} \wedge \mathsf{total}(\{\mathsf{co}(\tau, p)\}) \in \mathcal{R}]$

   Each instance of e should have precisely one p-part in τ.

4. $\forall_{p \in \tau} [\mathsf{Identifiable}(\mathsf{Base}(p)]$

   The bases of the predicators in τ have to be identifiable.

Conceivably, a particular root entity type e could have several sets τ of predicators such that $\mathsf{Identification}(\tau, e)$. To facilitate the denotation of entities, the choice for one of these sets should be explicitly recorded in a schema. To further facilitate these denotations, the order of the predicators in such a set τ also has to be recorded. This way the use of predicators in entity denotations can be avoided, as the position of a particular part of a denotation determines the corresponding predicator from τ. Therefore, for each entity type, τ should be an *ordered* set of predicators.

The function $\mathsf{Ident} : \mathcal{Q} \to \mathcal{P}^+$, from now on part of each structurally identifiable schema Σ, records the choice of τ for each root entity type e and the order of the predicators in τ. This function should be such that for the identification of each root entity type e, $\mathsf{set}(\mathsf{Ident}(e))$ can be chosen for τ in rule IDT3. The auxiliary partial function $\mathsf{Copred} : \mathcal{P} \mapsto \mathcal{P}$ yields the unique copredicator of each predicator involved in structural identification via Ident:

$$\mathsf{Copred}(p) = \mathsf{co}(\mathsf{set}(\mathsf{Ident}(e)), p)$$

for that unique (!) root entity type e for which $p \in \mathsf{set}(\mathsf{Ident}(e))$.

Finally, it should be noted that a far less restrictive identification scheme for root entity types is possible that still guarantees weak identification. The interested reader is referred to [40].

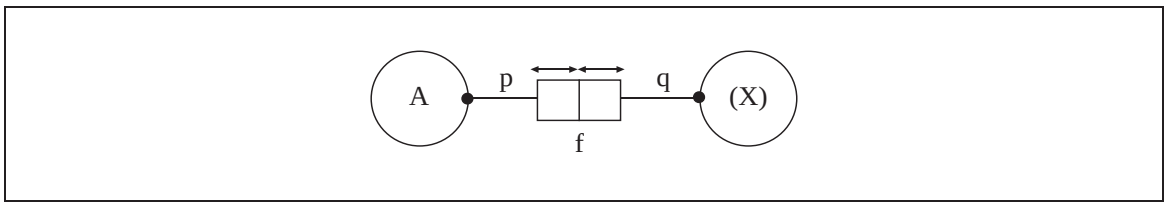

Figure 5.3: Simple identification

Example 5.3

Consider the schema of figure 5.3. In this schema X is identifiable, as it is a label type. Entity type A can be identified by choosing $\tau = \{q\}$. In that case, $\mathsf{unique}(\tau)$ *and* $\mathsf{Identifiable}(\mathsf{Base}(q))$. *Furthermore,* $\mathsf{co}(\tau, q) = p$, $\mathsf{Base}(p) = A$ *and* $\mathsf{total}(\{p\})$ *and* $\mathsf{unique}(\{p\})$. *Usually, the situation of*

figure 5.3 is abbreviated, the name of the label type is then simply put below the name of the entity type. This convention has often been applied in chapter 2.

Assuming that fact type f is identifiable (the identification of fact types is addressed in section 5.3.4), the whole schema is structurally identifiable as all its object types are identifiable and its only label type is involved in a total role constraint. For this schema, the functions Ident *and* Copred *are trivial:* $\mathsf{Ident}(A) = \langle q \rangle$ *and* $\mathsf{Copred}(q) = p$. □

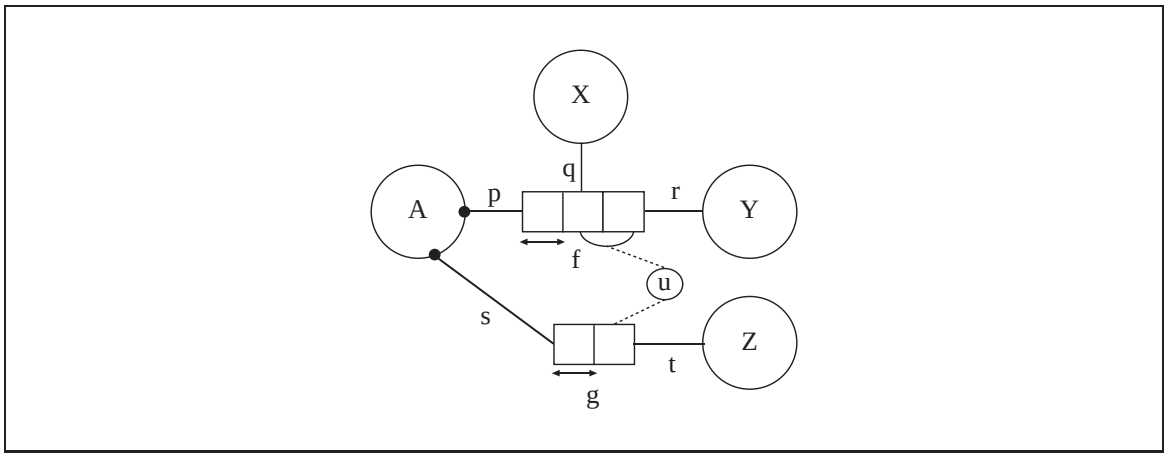

Figure 5.4: Abstract example of complex identification

Example 5.4

Suppose that in the schema of figure 5.4, the entity types X, Y and Z are identifiable. For the identification of A, $\tau = \{q, r, t\}$ has to be chosen. In this case the function Ident *cannot be determined on the basis of the schema alone, two possible choices for* $\mathsf{Ident}(A)$ *are for example,* $\langle q, r, t \rangle$ *and* $\langle t, r, q \rangle$. *The function* Copred *however, can be determined uniquely and is given by:*

$$\mathsf{Copred}(q) = p \qquad \mathsf{Copred}(r) = p \qquad \mathsf{Copred}(t) = s$$

□

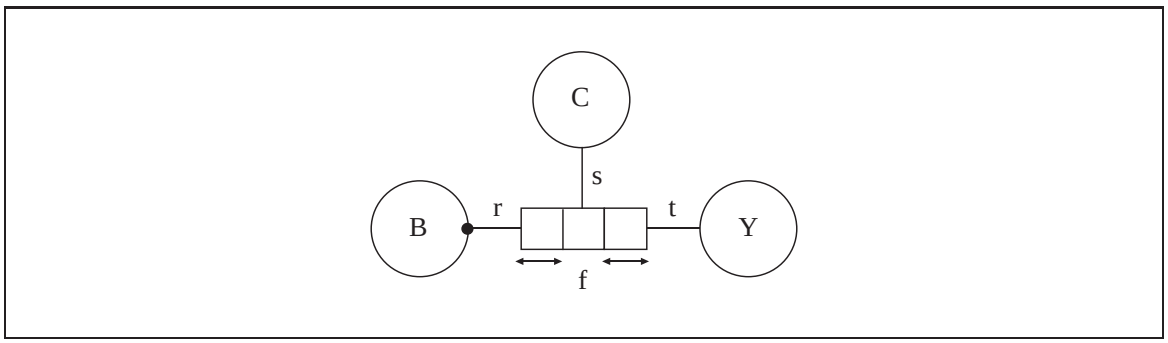

Figure 5.5: Unique reference but not identifiable

Example 5.5

Consider the schema of figure 5.5. Even if it is assumed that C and Y are identifiable B cannot be

identified. The only choice for τ is $\{t\}$. The problem with this choice, from a formal point of view, is that t does not have a unique copredicator.

Informally, it is clear that in this schema every instance of B is associated with a unique instance from Y. Therefore, no problems exist from a static *point of view. There are, however, problems from a* dynamic *point of view. If an instance is to be added to the population of B its denotation has to be specified. This denotation has to be the denotation of an instance of Y. The new instance of B has to be associated to that instance of Y in the population of fact type f. This, however, requires the participation of an* arbitrary *instance of C, as well.* □

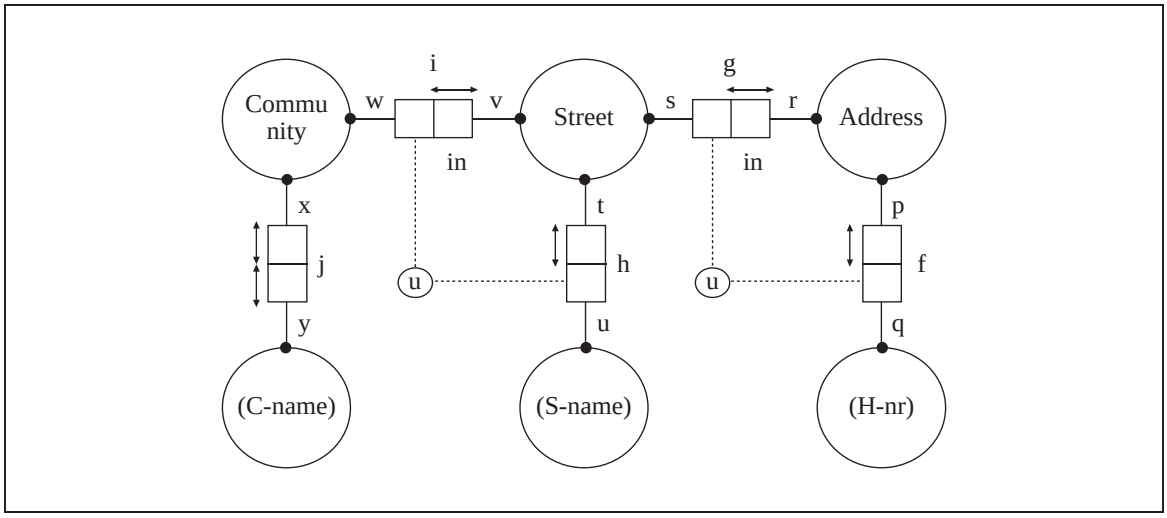

Figure 5.6: Concrete example of complex identification

Example 5.6

Figure 5.6 contains a concrete example of complex identification. Each label type is involved in a total role constraint. Entity type Community *can be identified by choosing $\tau = \{y\}$ (compare example 5.3). Entity type* Street *can then be identified by choosing $\tau = \{w, u\}$. Finally, entity type* Address *can be identified by the set of predicators $\{s, q\}$.*

For this schema the function Ident *could be defined as follows:*

$$\begin{array}{lcllcl} \mathsf{Ident}(\text{Address}) & = & \langle s, q\rangle & \mathsf{Ident}(\text{Street}) & = & \langle w, u\rangle \\ \mathsf{Ident}(\text{Community}) & = & \langle y\rangle & & & \end{array}$$

This would allow a denotation of addresses in the form $\langle$C-name, S-name, H-nr$\rangle$. *The function* Copred *is uniquely determined:*

$$\begin{array}{lcllcl} \mathsf{Copred}(q) & = & p & \mathsf{Copred}(s) & = & r \\ \mathsf{Copred}(u) & = & t & \mathsf{Copred}(w) & = & v \\ \mathsf{Copred}(y) & = & x & & & \end{array}$$

□

5.3.3 Identification of non-root entity types

A subtype is structurally identifiable if and only if all the object types needed for the evaluation of its subtype defining rule as well as all its supertypes are structurally identifiable:

[IDT4] $\mathsf{spec}(x) \wedge \forall_{y \in \mathsf{Objects}(\mathsf{SubRule}(x)) \cup \left\{ z \in \mathcal{O} \;\middle|\; x \,\mathsf{Spec}\, z \right\}} [\mathsf{Identifiable}(y)] \vdash \mathsf{Identifiable}(x)$

A generalised entity type inherits its identification from *some* of its specifiers. In some cases the identification of a specifier may depend on the identification of its generalised type. Therefore, it is not required that a generalised type is identifiable if and only if *all* its specifiers are identifiable. Formally:

[IDT5] $\mathsf{gen}(x) \wedge \exists_{y \in \mathcal{O}} [x \,\mathsf{Gen}\, y \wedge \mathsf{Identifiable}(y)] \vdash \mathsf{Identifiable}(x)$

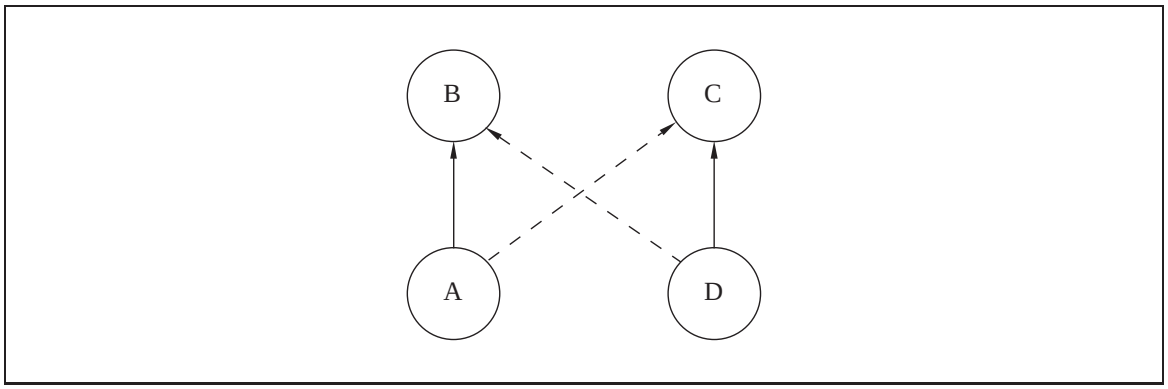

Figure 5.7: Non-identifiable information structure with specialisation and generalisation

Example 5.7

The information structure of figure 5.7 consists of four non-root entity types and is not identifiable, even if one assumes the existence of correct subtype defining rules. Entity type B can only be identified if its only specifier D can be identified (rule IDT5*). D is a subtype and can only be identified if its direct supertype C can be identified (rule* IDT4*). Identification of entity type C requires the identification of entity type A, which identification depends on the identification of B. Therefore, in order to prove that B is identifiable, one has to prove that B is identifiable. Evidently, this is impossible.* □

5.3.4 Identification of fact types

Fact types depend for their identification on the bases of their constituent predicators:

[IDT6] $f \in \mathcal{F} \wedge \forall_{p \in f} [\mathsf{Identifiable}(\mathsf{Base}(p))] \vdash \mathsf{Identifiable}(f)$

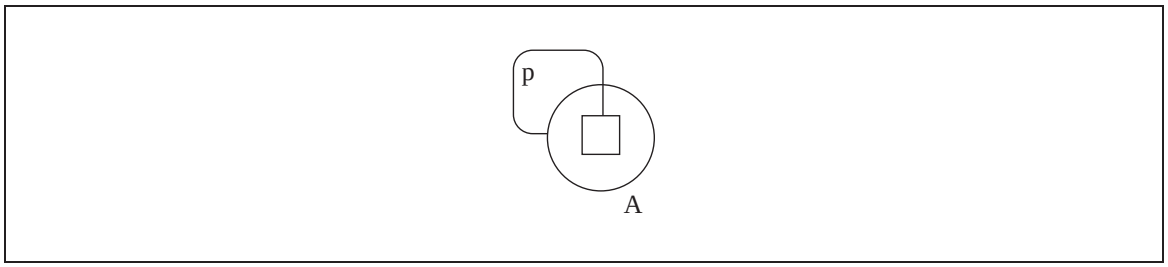

Figure 5.8: Cyclic unary fact type

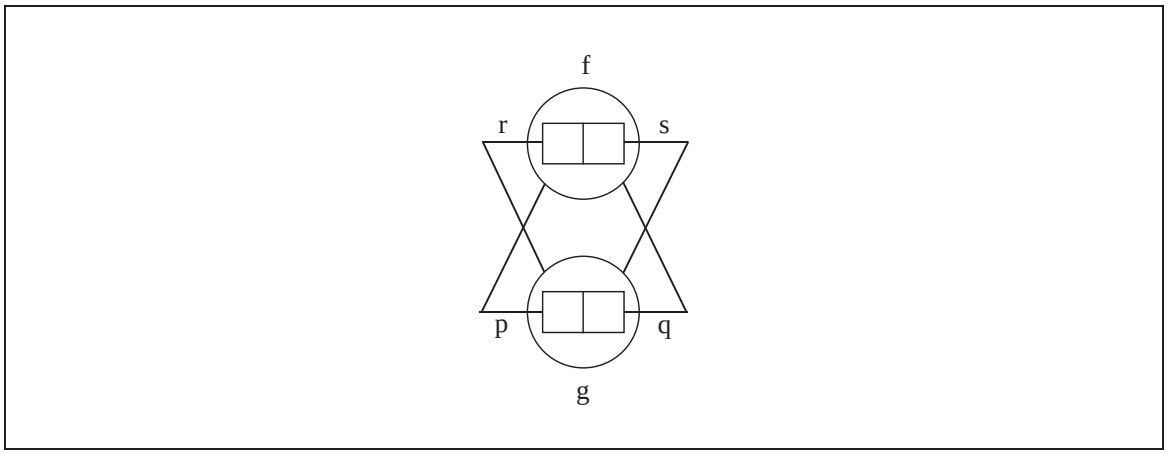

Figure 5.9: Two fact types contained in a cycle

Example 5.8

The information structure depicted in figure 5.8 consists of a single fact type $A = \{p\}$ *where* $\mathsf{Base}(p) = \mathsf{Fact}(p) = A$. *Therefore, this fact type is not identifiable. The fact types in figure 5.9 are not identifiable either. Proof that* f *can be identified requires proof that* g *can be identified. However, proof that* g *can be identified requires proof that* f *can be identified. Note that these three fact types cannot be populated with instances from* Ω. *In section 5.4 the relation between structural identification and populatability will be formalised.* □

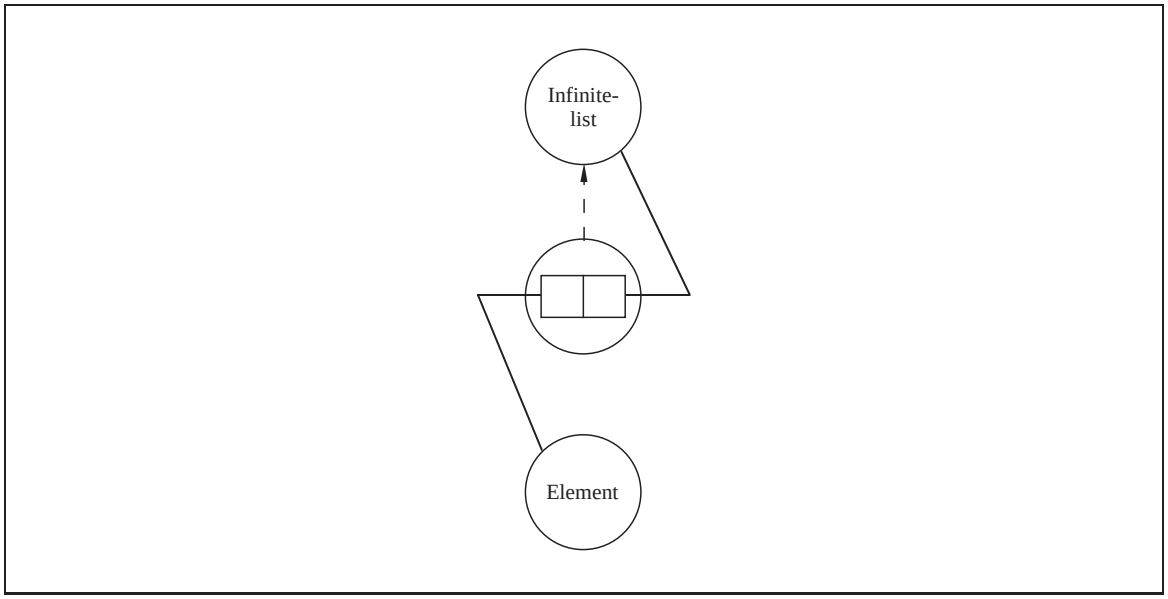

Figure 5.10: Schema of an infinite list

Example 5.9

As stated before, the identification of a specifier may depend on the identification of its generalised type. The formula example of figure 2.20 serves as an example. In this case, the identification of

Formula *depends on* Variable. *Fact type* f*, a specifier of* Formula*, depends for its identification on* Formula.

An example of an information structure combining fact objectification and generalisation that is not identifiable is shown in figure 5.10. The problem is that object type Infinite-list *depends for its identification on its only specifier, a fact type, which on its turn depends for its identification on* Infinite-list. □

From the definition of structural identification it is clear that the presence of unique names for facts does not guarantee that the involved fact type can be identified. In figure 5.11 for example, each fact in the population of fact type f is associated with a unique label from label type L via fact type g. The identification of fact type f however, depends on the identification of entity types A and B.

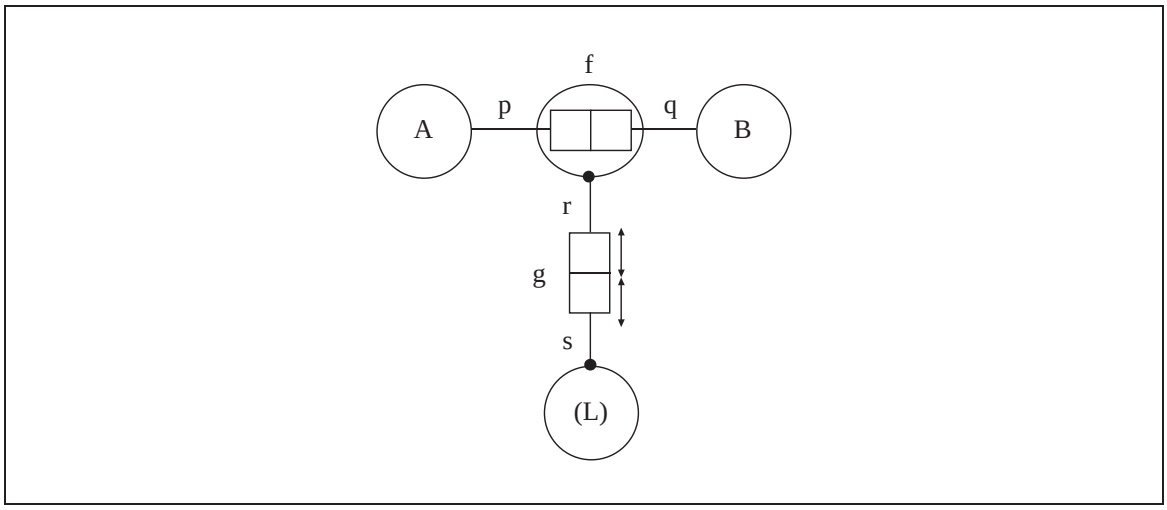

Figure 5.11: Standard names but not identifiable

In order to facilitate the textual denotation of facts, the function Ident is augmented with the fact types. For each fact type f, $\mathsf{set}(\mathsf{Ident}(f)) = f$. In that case, facts can be denoted without mentioning their predicators, as they are clear from the textual ordering.

5.3.5 Identification of power types and sequence types

Sets and sequences depend for their denotation on their constituent elements, therefore, power types and sequence types depend for their identification on their element types.

[IDT7] $x \in \mathcal{G} \cup \mathcal{S} \wedge \mathsf{Identifiable}(\mathsf{Elt}(x)) \vdash \mathsf{Identifiable}(x)$

Example 5.10

Assuming that entity type C *is identifiable in figure 5.12, power type* B *is identifiable, since its element type* A *may derive its identification from* C. □

Example 5.11

In the schema of figure 2.12, the entity type Function *may derive its identification from* Function-name *and* Parameter-type-sequence*. Sequence type* Parameter-type-sequence *derives its identification from its element type* Parameter-type*, which on its turn is identified by label type* Pt-name. □

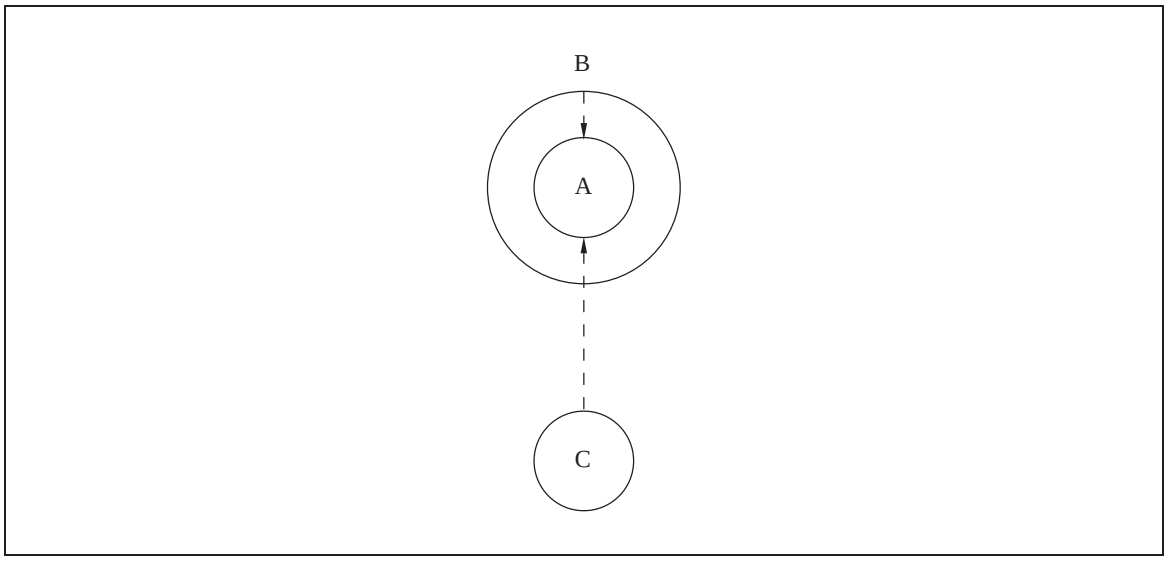

Figure 5.12: Identifiable generalised object type

5.3.6 Identification of schema types

Schema types can only be identified if all object types in their decomposition can be identified.

[IDT8] $x \in \mathcal{C} \wedge \forall_{y \in \mathcal{O}, x \prec y} [\mathsf{Identifiable}(y)] \vdash \mathsf{Identifiable}(x)$

5.4 Theoretical results

In this section, focus is on the theoretical consequences of the definition of structural identification. Let $\mathcal{A}$ be the set of atomic object types, consisting of the label types and the root entity types: $\mathcal{A} = \mathcal{L} \cup \mathcal{Q}$. Now, the following property is trivial:

$$\mathsf{StructId}(\Sigma) \Rightarrow \forall_{x \in \mathcal{A}} \exists_{p \in \mathcal{P}} [\mathsf{Base}(p) = x]$$

This is called the Atomic Anchorage Rule. For root entity types this can be sharpened using total role constraints:

$$\mathsf{StructId}(\Sigma) \Rightarrow \forall_{x \in \mathcal{Q}} \exists_{p \in \mathcal{P}} [\mathsf{Base}(p) = x \wedge \mathsf{total}(\{p\}) \in \mathcal{R}]$$

Each population of a schema of which all root entity types are identifiable is weakly identified:

Theorem 5.1 Let Σ be a schema.

$$\forall_{e \in \mathcal{Q}} [\mathsf{Identifiable}(e)] \Rightarrow \forall_{\mathsf{Pop} \in \mathsf{POP}_\Sigma} [\mathsf{WeakId}(\mathcal{I}, \mathsf{Pop})]$$

Another important consequence of structural identification is that it guarantees that each population is connected:

Theorem 5.2 Let Σ be a schema.

$$\forall_{e \in \mathcal{Q}} [\mathsf{Identifiable}(e)] \Rightarrow \forall_{\mathsf{Pop} \in \mathsf{POP}_\Sigma} [\mathsf{Connected}(\mathsf{Pop})]$$

Connectivity in NIAM is in fact a stronger notion than connectivity as defined in section 2.2. NIAM stipulates that entities should not only occur in the population of some predicator, but in the population of some predicator not involved in the identification of the root entity type involved, as well. This requirement can be formalised as:

$$\forall_{e \in \mathcal{Q}} \forall_{x \in \mathsf{Pop}(e)} \exists_{p \in \mathcal{P} \setminus \mathsf{set}(\mathsf{Ident}(e))} \exists_{t \in \mathsf{Pop}(\mathsf{Fact}(p))} [t(p) = x]$$

In [31], Halpin argues that sometimes it is necessary to allow the recording of the mere existence of an entity. Such entities are termed *lazy entities*. Lazy entities do not fulfil the aforementioned strong connectivity requirement. Along with Halpin we think that it is not always desirable to exclude lazy entities. Therefore, we do not require populations to fulfil the strong connectivity requirement.

Finally we consider the following property:

Theorem 5.3 If $\Sigma = \langle \mathcal{I}, \mathcal{R} \rangle$ is a schema and $x \in \mathcal{O}$, then:

$$\mathsf{Identifiable}(x) \Rightarrow \exists_{\mathsf{Pop} \in \mathsf{POP}_\mathcal{I}} [\mathsf{Pop}(x) \neq \varnothing]$$

The proofs of these properties are found in [35].

5.5 Verification

It is possible, that as a result of constraint contradictions, an object type can be instantiated by a population of the information structure, but not by a valid population of the schema. In section 5.5.2, various kinds of constraint contradictions are identified. These constraint contradictions can be characterized by schema properties. Section 5.6 deals with the complexity of verifying two important schema properties. We start with a general description of verification in section 5.5.1.

5.5.1 Background

The need for effective verification, validation and integrity (VVI) technologies is well established in both the Database (DB) and Expert Systems (ES) communities ([79]). VVI requirements continue to grow in both fields. This is a consequence of:

- The exponential growth in potential users made possible by network technology and the Web in particular.
- The potential for distributed multi-component systems, such as multidatabase and intelligent multi-agent systems.
- The expanding range of applications to which ES and DB technology can be applied.

There is much overlap between the Verification, Validation and Integrity issues that the ES and DB communities seek to address. Examples include:

1. The verification of conceptual schema in DB and the ontologies concept in ES.

2. Automated and semi-automated schema refinement in DB, and knowledge refinement in ES.
3. Conflict resolution in the event of updating replicated data through DB integrity control, and truth maintenance and reconciliation in ES.
4. Rule checking in Active DB and the classic V&V of rule bases.
5. Common approaches applicable to both DB and ES V&V.
6. Shared tools, e.g. the use of neural net works for integrity checking.

5.5.2 Basic schema properties

The empty population $\varnothing_\Sigma$ of a schema $\Sigma = \langle \mathcal{I}, \mathcal{R} \rangle$, is defined by:

$$\mathsf{IsPop}(\mathcal{I}, \varnothing_\Sigma) \wedge \forall_{x \in \mathcal{O}} \left[\varnothing_\Sigma(x) = \varnothing \right]$$

The *law of the excluded miracle* states that no information can be derived from the empty population.

$$\forall_{r \in \boldsymbol{\mathcal{R}}(\mathcal{I})} \left[\mathsf{Val}[\![r]\!] \left(\varnothing_\Sigma \right) = \varnothing \right]$$

The validity of this law can be shown, using structural induction on the construction of relational expressions. Atomic relational expressions, i.e. fact types, obviously have an empty population. To prove the induction step, one has to assume that r and s are relational expressions with an empty population. Application of the various relational operators then again yields a relational expression with an empty population.

Although we have introduced several powerful integrity constraints, these constraints are not able to exclude empty populations. In other words, the empty population fulfils all constraints:

$$\mathsf{IsPop}(\Sigma, \varnothing_\Sigma)$$

The validity of this property can be shown as follows. The definition of the empty population ensures that $\mathsf{IsPop}(\mathcal{I}, \varnothing_\Sigma)$. Most types of constraints ensure certain relationships between (populations of) relational expressions. According to the law of the excluded miracle, all relational expressions on $\mathcal{I}$ have an empty population, and therefore these types of constraints are satisfied. This is easily checked for each type of constraint. Other types of constraints, e.g. the constraints for power types, have not been defined using relational expressions. It is however easily verified that these types of constraints are satisfied in the empty population. As a result $\mathsf{IsPop}(\Sigma, \varnothing_\Sigma)$.

5.5.3 Local populatability

Evidently, a schema which only allows the empty population is not desirable. But there are other forms of "emptyness" that are not desirable. For example, one would at least require that every atomic object type of a schema can be populated. A schema where all atomic object types can be populated is called *local atomic populatable*. Formally:

$$\mathsf{LocAtomPop}(\Sigma) = \forall_{a \in \mathcal{A}} \exists_{\mathsf{Pop} \in \mathsf{POP}_\Sigma} \left[\mathsf{Pop}(a) \neq \varnothing \right]$$

As an example, the schema of figure 5.13 is not local atomic populatable, due to the fact that entity type A cannot be populated. The occurrence frequency constraint implies that f cannot be populated. As the total role constraint states that each instance of A has to occur in f, the population of A has to be empty.

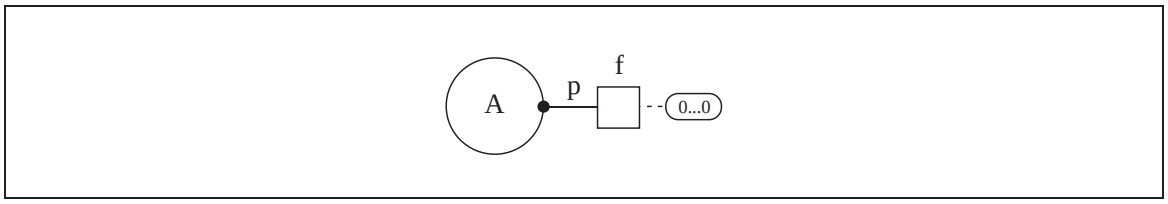

Figure 5.13: A non-local atomic populatable schema

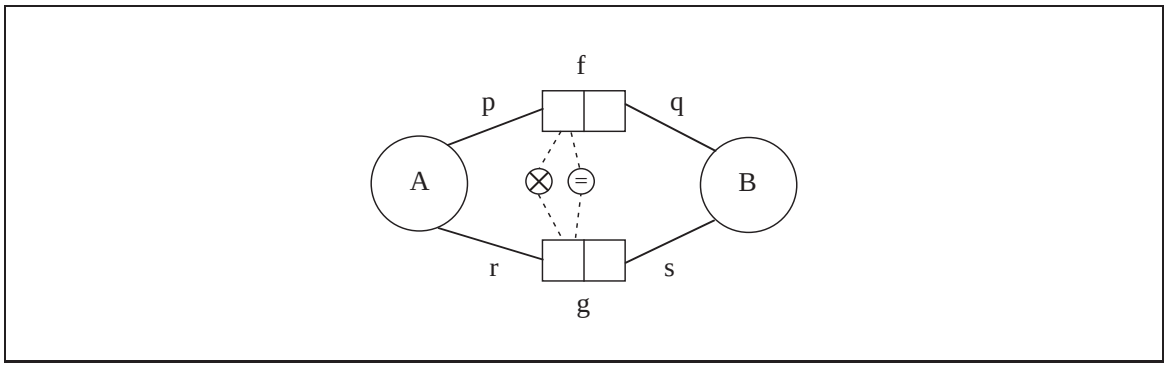

Figure 5.14: A non-local object populatable schema

An analogous property can be defined for all object types. A schema is called *local object populatable*, if and only if each object type can be populated:

$$\mathsf{LocObjPop}(\Sigma) = \forall_{x \in \mathcal{O}} \exists_{\mathsf{Pop} \in \mathsf{POP}_\Sigma} [\mathsf{Pop}(x) \neq \varnothing]$$

Naturally, local object populatability is a stronger notion than local atomic populatability, because $\mathcal{A} \subseteq \mathcal{O}$.

As an example, let Pop be a population of the schema of figure 5.14. The equality constraint states that

$$\mathsf{Val}[\![\theta_p(\pi_p(f))]\!](\mathsf{Pop}) = \mathsf{Val}[\![\theta_r(\pi_r(g))]\!](\mathsf{Pop}),$$

while the exclusion constraint states that

$$\mathsf{Val}[\![\theta_p(\pi_p(f))]\!](\mathsf{Pop}) \cap \mathsf{Val}[\![\theta_r(\pi_r(g))]\!](\mathsf{Pop}) = \varnothing.$$

Therefore, $\mathsf{Pop}(f) = \varnothing$ and $\mathsf{Pop}(g) = \varnothing$. This implies that the schema of figure 5.14 is not local object populatable.

5.5.4 Global populatability

The previous properties require that a population exists for each object type. This does not mean that a population exists, that populates each object type. A schema is called *global atomic populatable* if and only if a population exists, that populates each atomic object type:

$$\mathsf{GlobAtomPop}(\Sigma) = \exists_{\mathsf{Pop} \in \mathsf{POP}_\Sigma} \forall_{a \in \mathcal{A}} [\mathsf{Pop}(a) \neq \varnothing]$$

Naturally, global atomic populatability implies local atomic populatability.

We consider the following example. The schema of figure 5.15 is local object populatable, but not global atomic populatable. A population Pop cannot instantiate atomic object types A, B and C at the same time. Due to the total role constraints on $\{q\}$ and on $\{s\}$, the fact types f and g would have to have nonempty

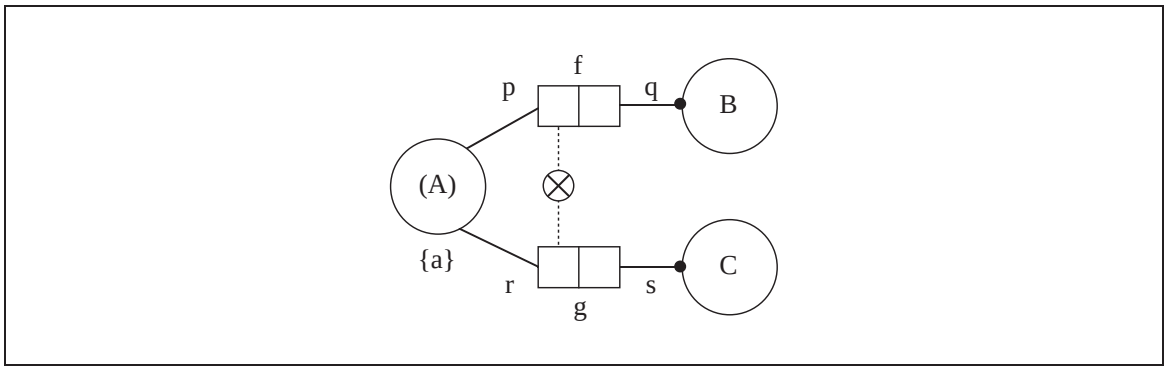

Figure 5.15: A local object populatable schema which is not global atomic populatable

populations, which is impossible as in that case instance a would have to occur in the populations of predicators p and r, contradicting the exclusion constraint.

Next we consider *global object populatability*. This is defined as:

$$\mathsf{GlobObjPop}(\Sigma) = \exists_{\mathsf{Pop} \in \mathsf{POP}_\Sigma} \forall_{x \in \mathcal{O}} [\mathsf{Pop}(x) \neq \varnothing]$$

This definition implies that a global object populatable schema is also global atomic populatable.

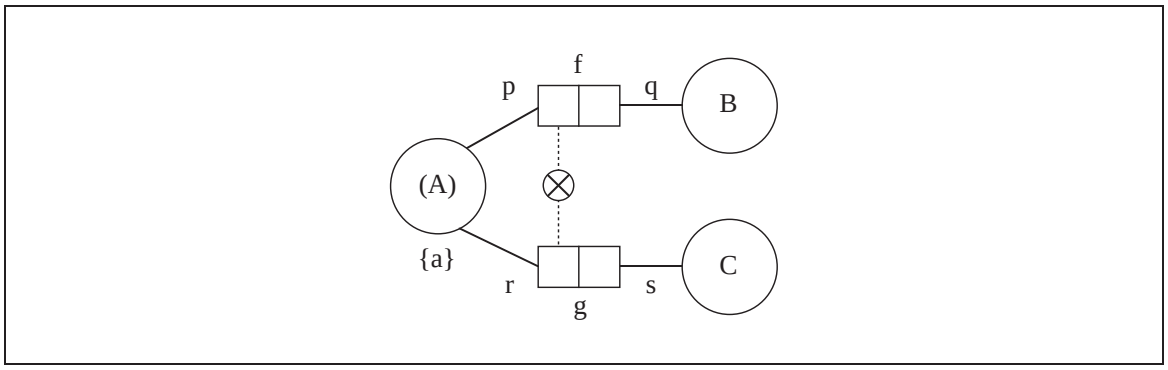

Figure 5.16: A non-global object populatable schema

As an example, the schema of figure 5.16 is not global object populatable, since this would require a population where both f and g have nonempty instantiations. This is impossible, as explained in the previous example.

5.5.5 Significant populatability

A schema can be global object populatable, but still exhibit constraint peculiarities. In order to deal with these peculiarities, we introduce *significantly populatable* schemata. A schema is called significantly populatable if and only if **it does not contain constraints which can be proved to be too weak**.

Formally, let $\mathcal{I}$ be an information structure and c and c' be sets of constraints in $\mathbf{\Gamma}(\mathcal{I})$. Constraint set c is at least as restrictive as constraint set c' in information structure $\mathcal{I}$, $c \vDash_{\mathcal{I}} c'$, iff:

$$\forall_{\mathsf{Pop} \in \mathsf{POP}_\mathcal{I}} [\mathsf{Pop} \models c \Rightarrow \mathsf{Pop} \models c']$$

A schema $\Sigma = \langle \mathcal{I}, \mathcal{R} \rangle$ is called significantly populatable iff:

$$\mathsf{Signif}(\Sigma) = \neg\exists_{c \subseteq \mathcal{R}, c \neq \varnothing} \exists_{c' \subseteq \mathbf{\Gamma}(\mathcal{I})} \left[\forall_{\mathsf{Pop} \in \mathsf{POP}_\Sigma} \left[\mathsf{Pop} \models c' \right] \wedge c' \Vdash_{\mathcal{I}} c \wedge c \nVdash_{\mathcal{I}} c' \right]$$

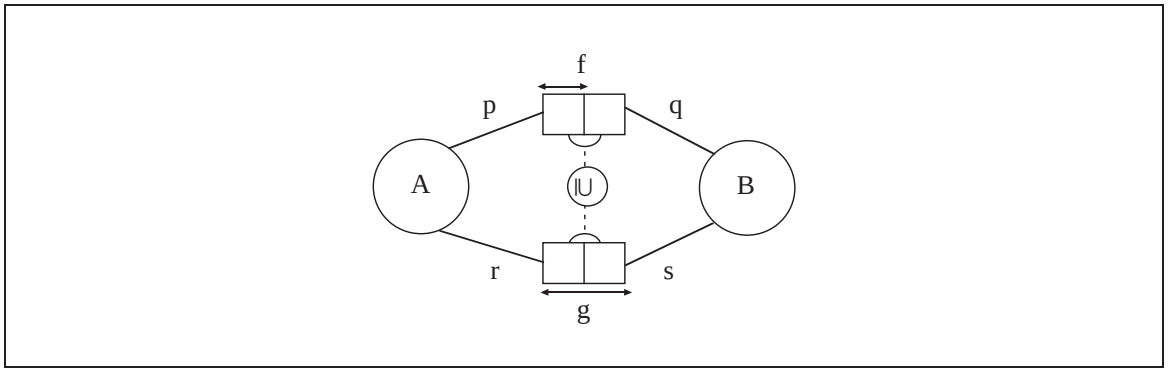

Figure 5.17: A non-significantly populatable schema: $\mathsf{unique}(\{r, s\})$ is too weak

As an example, figure 5.17 shows a schema which is easily seen to be global object populatable. However, it is clear that the uniqueness constraints specified are not in harmony with the subset constraint. We will show that this schema, which we will refer to as $\Sigma = \langle \mathcal{I}, \mathcal{R} \rangle$, is not significantly populatable. Suppose that a population $\mathsf{Pop} \in \mathsf{POP}_\Sigma$ exists, such that $\mathsf{Pop} \not\models \{\mathsf{unique}(\{r\})\}$. In that case, fact type g contains two instances with the same r-component. Due to the subset constraint however, these instances would also have to occur in the population of fact type f contradicting its uniqueness constraint. Therefore, $\forall_{\mathsf{Pop} \in \mathsf{POP}_\Sigma} [\mathsf{Pop} \models \{\mathsf{unique}(\{r\})\}]$. Furthermore, $\{\mathsf{unique}(\{r\})\} \Vdash_{\mathcal{I}} \{\mathsf{unique}(\{r, s\})\}$ but not $\{\mathsf{unique}(\{r, s\})\} \Vdash_{\mathcal{I}} \{\mathsf{unique}(\{r\})\}$.

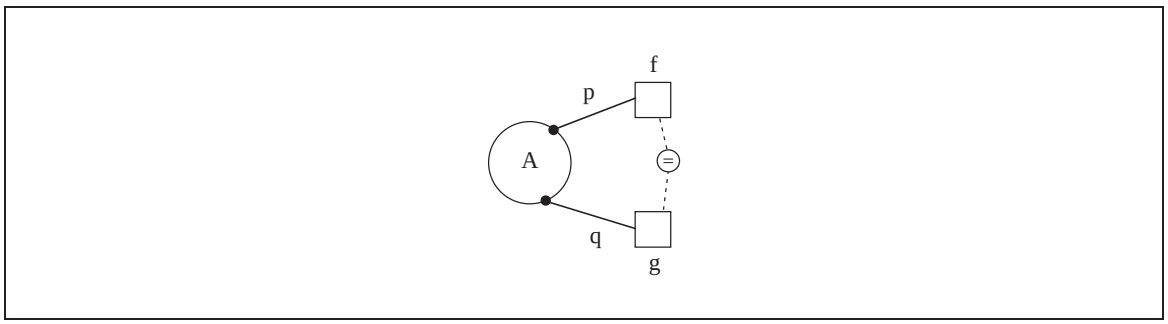

Figure 5.18: Another non-significantly populatable schema: $\mathsf{equal}(\{p\}, \{q\})$ is too weak

As another example, in the schema of figure 5.18, we have

$$\{\mathsf{total}(\{p\}), \mathsf{total}(\{q\})\} \Vdash_{\mathcal{I}} \{\mathsf{equal}(\{p\}, \{q\})\}$$

but not vice versa (the reader should verify this!). Hence, this schema is not significantly populatable.

5.6 Complexity of verification

In this section, the complexity of the verification of schema properties is considered. Focus is on the two most important schema properties, global atomic populatability (section 5.6.2), and global object populatability (section 5.6.3). For a schema $\Sigma = \langle \mathcal{I}, \mathcal{R} \rangle$, $|\mathcal{R}|$ is the size of the problem. We will prove that

determining whether a schema is global atomic populatable and determining whether a schema is global object populatable are both NP-complete problems. To this end, we will use the 3-D matching problem (section 5.6.1).

5.6.1 The 3-D Matching Problem

The 3-D matching problem is often used for proving NP-completeness results and is a generalisation of the well-known marriage problem. The 3-D matching problem is formulated as follows ([28]):

> Let $M \subseteq W \times X \times Y$, where W, X and Y are disjoint sets having the same number of elements q. The problem is to determine whether M contains a so-called matching, i.e. a subset $M' \subseteq M$ with q elements, such that no two elements of M' agree in any coordinate.

This problem is known to be NP-complete (see [28]). Without loss of generality, we assume that every element of W, X and Y occurs in some element of M (which is easily checked). If this is not the case, no matching is possible. Note that due to this assumption a matching problem is completely determined by the set M.

As an example, we have $W = \{w_1, w_2\}$, $X = \{x_1, x_2\}$, $Y = \{y_1, y_2\}$, and

$$M = \{\langle w_1, x_2, y_2\rangle, \langle w_2, x_1, y_1\rangle, \langle w_1, x_1, y_1\rangle\}$$

Now it is evident that $M' = \{\langle w_1, x_2, y_2\rangle, \langle w_2, x_1, y_1\rangle\}$ is a matching of M.

5.6.2 Complexity of global atomic populatability

In this section, the translation of a 3-D matching problem M to a PM schema $\Sigma(M)$ is described. The translation is such, that M has a matching if and only if $\Sigma(M)$ is global atomic populatable. Consequently, determining whether $\Sigma(M)$ is global atomic populatable is an NP-complete problem.

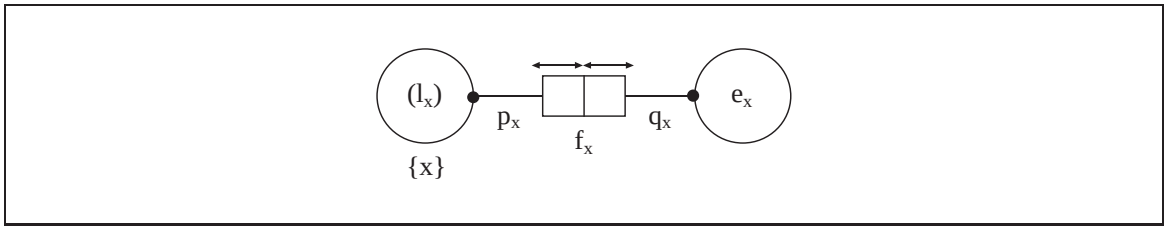

Figure 5.19: A fact type between l_x and e_x

For each element $x \in W \cup X \cup Y$, a fact type $f_x = \{p_x, q_x\}$ is introduced (see figure 5.19), with $\mathsf{Base}(p_x) = l_x \in \mathcal{L}$ and $\mathsf{Base}(q_x) = e_x \in \mathcal{E}$. The enumeration constraint ensures that label type l_x can only contain x. The uniqueness and total role constraints allow e_x to be identified by label type l_x.

For each element $m \in M$ two fact types are introduced. For $m = \langle w, x, y\rangle$ the fact types $f_m = \{r_m, s_m, t_m\}$ and $f'_m = \{u_m, v_m, z_m\}$ are as shown in figure 5.20. Fact type f_m is intended to be populated with an instance corresponding to m. A nonempty population of f'_m will correspond to $m \in M'$.

The matching conditions (each element of W, X and Y should occur in the matching, and no two elements of M' may agree in any coordinate) are transformed into total role and exclusion constraints (respectively) for each element $x \in W \cup X \cup Y$ (see figure 5.21).

Obviously, this transformation takes not more than polynomial time. The correctness of this transformation is expressed by the following theorem:

$$M \text{ has a matching} \iff \mathsf{GlobAtomPop}(\Sigma(M))$$

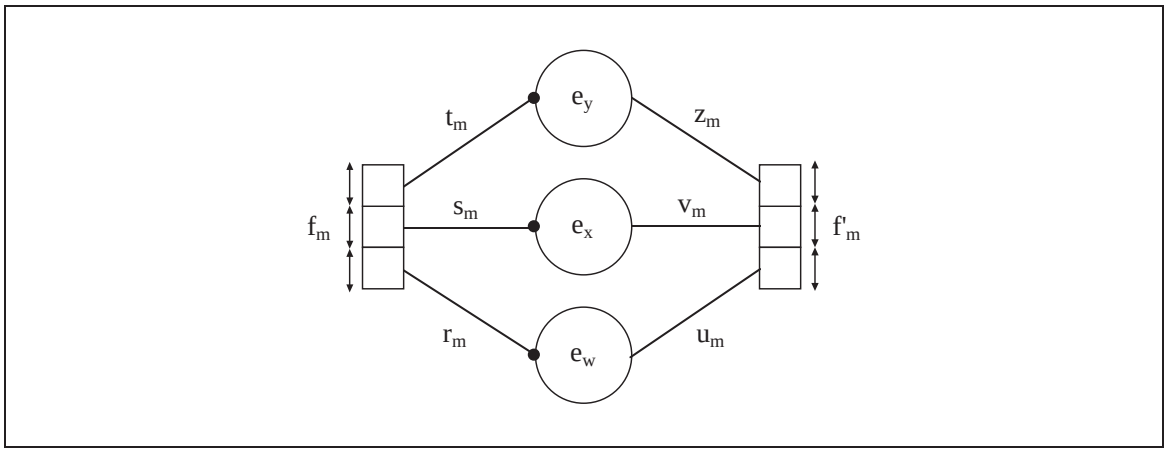

Figure 5.20: Two fact types for each $m \in M$

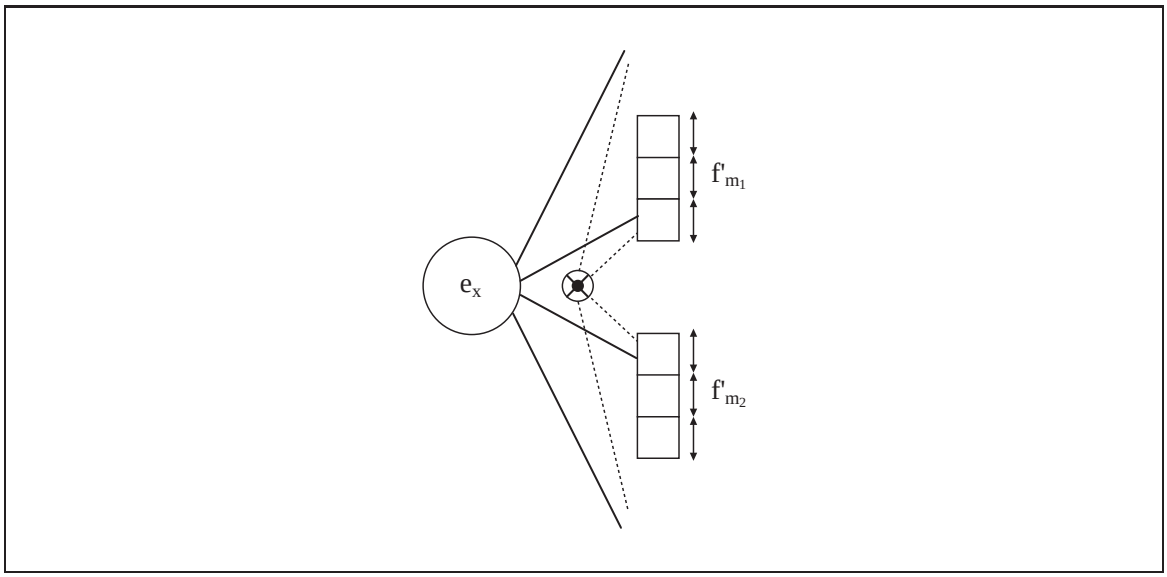

Figure 5.21: Matching conditions via constraints

Proof:

$\Longrightarrow$ Let $M' \subseteq M$ be a matching. From M' we construct a population Pop, such that every atomic object type is populated. For each label type l_x, $\mathsf{Pop}(l_x) = \{x\}$. For each entity type e_x, $\mathsf{Pop}(e_x) = \{\theta_x\}$, where $\theta_x \in \Theta$. Each θ_x is linked to x via bridge type f_x. Let $m = \langle w, x, y\rangle$ be an element of M. The population of each fact type f_m then consists of the single fact $\{r_m : \theta_w, s_m : \theta_x, t_m : \theta_y\}$. If $m \in M'$, then $\mathsf{Pop}(f'_m) = \{\{u_m : \theta_w, v_m : \theta_x, z_m : \theta_y\}\}$, otherwise, $\mathsf{Pop}(f'_m) = \varnothing$. This population satisfies all constraints and populates all atomic object types.

$\Longleftarrow$ Assume $\mathsf{GlobAtomPop}(\Sigma(M))$ and let Pop be a global atomic population of $\Sigma(M)$. A matching can now be defined as:

$$M' = \big\{m \in M \;\big|\; \mathsf{Pop}(f'_m) \neq \varnothing\big\},$$

because exclusion and total role constraints ensure the matching conditions. □

The previous theorem implies that determining whether a schema is global atomic populatable is an NP-hard problem. To prove that this problem is NP-complete, one has to show that it is in NP. This can be accomplished by proving that checking whether a specific population fulfils all constraints can be performed in polynomial time.

As an example, we consider the following sets:

$$\begin{aligned} W &= \{w_1, w_2, w_3\} \\ X &= \{x_1, x_2, x_3\} \\ Y &= \{y_1, y_2, y_3\} \\ M &= \{(w_1, x_1, y_1), (w_2, x_3, y_2), (w_3, x_2, y_3), \\ &\quad (w_2, x_2, y_2), (w_3, x_3, y_3)\} \end{aligned}$$

After checking whether M has a matching, we may give an example of such a matching. Also, we may give the GlobAtomPop schema for M and give a population for this schema, if possible.

5.6.3 Complexity of global object populatability

A transformation of a matching problem M to a schema $\Sigma(M)$ such that M has a matching if and only if $\Sigma(M)$ is global object populatable is analogous with the previous section. This transformation is described in [35].

5.7 Self-meta-modelling

To provide a first indication of expressive power, this section investigates the (meta)modelling of a technique in terms of itself. Self-meta-modelling is used in other contexts as well, for example:

Relational databases: A relational database consists of tables. Information about the structure of the database (e.g. the names of the columns of the tables) may be stored in tables as well. The advantage of this approach is that a single language (e.g. SQL) can be used for database retrieval as well as for database maintenance.

Compiler construction: In case syntax and semantics of a programming language (e.g. C) is expressed in the same language, we have a good starting-point for the construction of a compiler. This is particularly the case if only a subset (e.g. a subset C' of C) is needed to express the language. We then only have to map this subset to a lower level language, such as assembler.

5.7.1 Meta model for information structures

Figure 5.22 shows a schema, modelling part of the syntax of schemata. In this schema, power type $\mathcal{F}$ has as element type $\mathcal{P}$, modelling fact types being sets of predicators. The partition constraint models the fact that every predicator occurs in precisely one fact type. Every predicator is related to precisely one instance from $\mathcal{O}$, its base, via fact type Base. The fact types Spec and Gen model the Spec and Gen relations. The exclusion constraints ensure that if a Spec b, then neither a Gen b nor b Gen a. The object types $\mathcal{G}$ and $\mathcal{S}$ are generalised into object type $\mathcal{G} \cup \mathcal{S}$. Each instance from this object type is related via fact type Elt to precisely one instance of object type $\mathcal{O} \backslash \mathcal{L}$, its element type. The decomposition of schema types is captured by fact type $\prec$.

The object type *Subtype* is a subtype of object type $\mathcal{E}$ and is to contain those entity types that are subtypes (see figure 5.23). Therefore, its subtype defining rule is: π_q(Spec). Each subtype has a subtype defining rule, which is a relational expression. This is modelled by the fact type between the object types *Subtype* and $\boldsymbol{\mathcal{R}}(\mathcal{I})$.

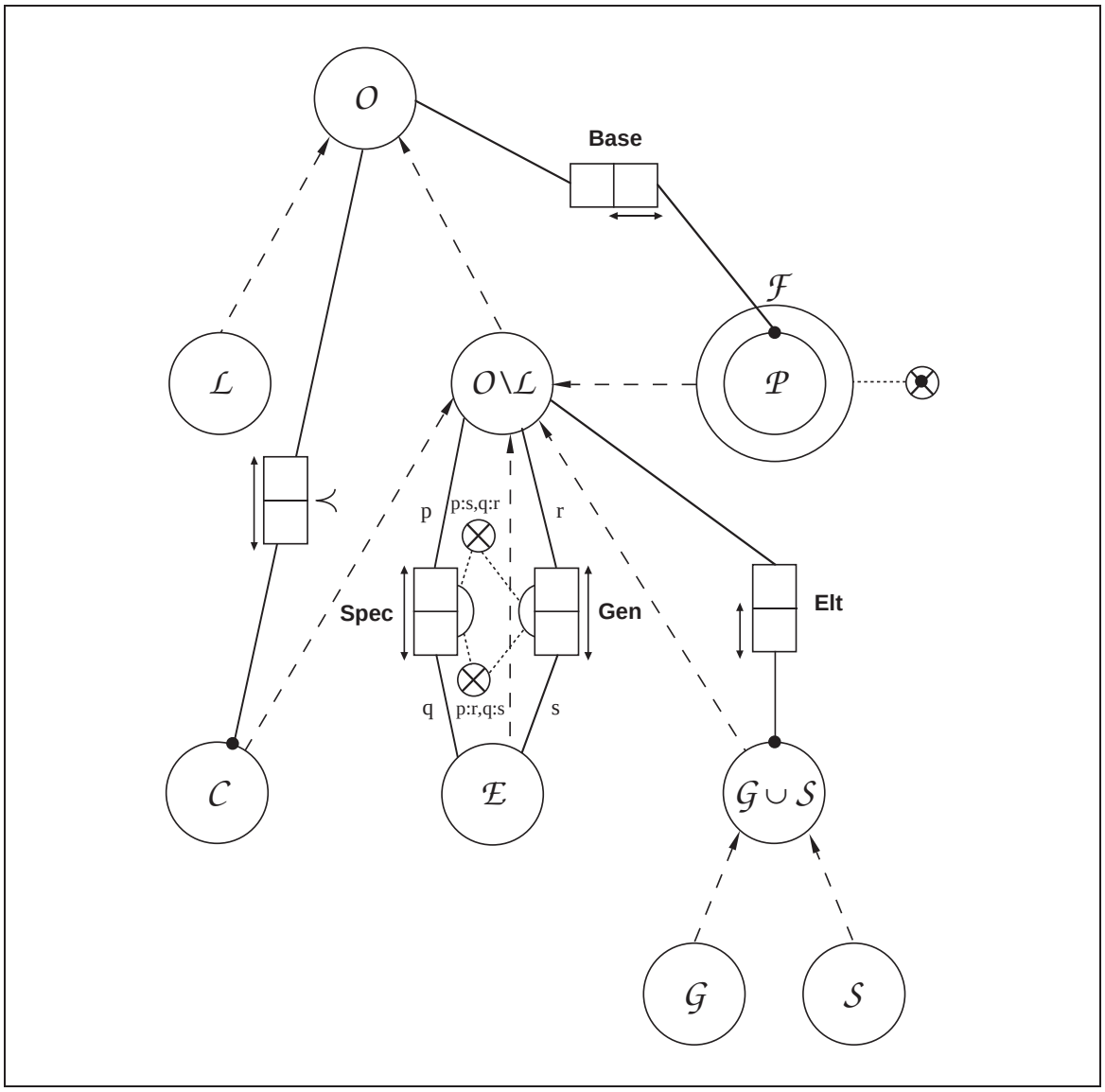

Figure 5.22: Meta model for information structures

5.7.2 Meta model for relational algebra

Object type $\mathcal{R}(\mathcal{I})$ is recursively defined. A small part of its definition is shown in figure 5.24. This schema models that each fact type is a relational expression, that the union of two relational expressions is also a relational expression and that a projection is syntactically determined by a relational expression and two ordered sequences of predicators. The other relational operators can be modelled analogously. It should be noted that many of the restrictions that have to be imposed on the construction of relational expressions, for example that the two ordered sequences of predicators involved in a projection should have the same length, cannot be expressed using only the graphical constraints. To capture the semantics of relational expressions as well, a powerful constraint language is required.

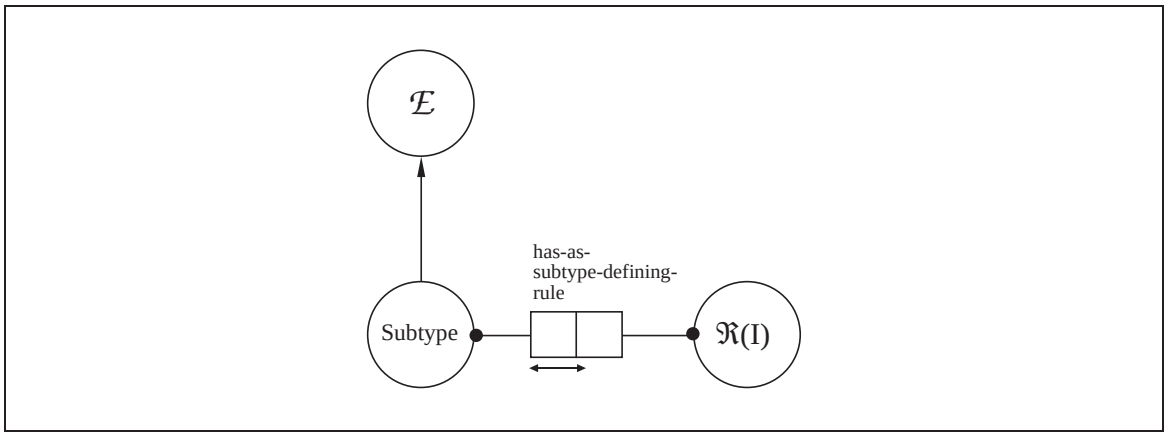

Figure 5.23: Subtypes

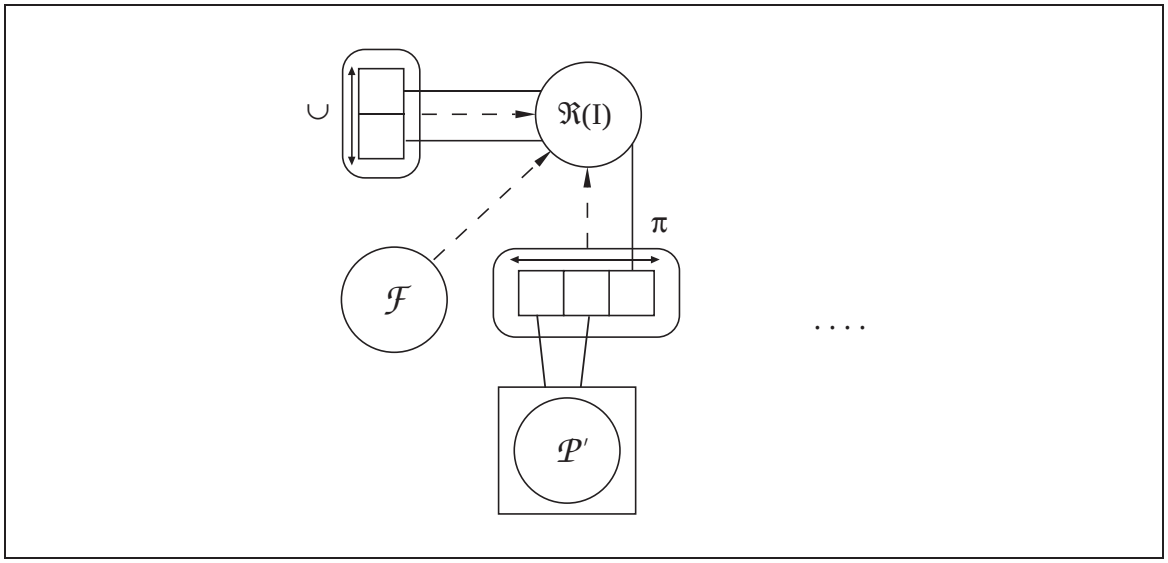

Figure 5.24: Relational operators

5.7.3 Meta model for integrity constraints

In schema 5.25, the total role constraint and the occurrence frequency constraint are modelled. A total role constraint is a set of predicators, therefore object type *Total-role-constraint* is a power type with as element type $\mathcal{P}$. An occurrence frequency constraint is also a set of predicators in combination with a lower bound and upper bound. These lower and upper bounds come from a label type *Range* which has as domain the set of natural numbers united with infinity (∞).

In the schema of figure 5.26, the set constraints are modelled. The three types of set constraints can be modelled as a combination of two ordered sets of predicators.

Figure 5.27 shows that an enumeration constraint can be modelled by a relation between object type $\mathcal{L}$ and object type Ω. It is possible to model the precise construction of object type Ω. This will not be done however, as this construction is analogous to the construction of $\boldsymbol{\mathcal{R}}(\mathcal{I})$.

Power type constraints are modelled in figure 5.28. The cover constraint and the power exclusion constraints

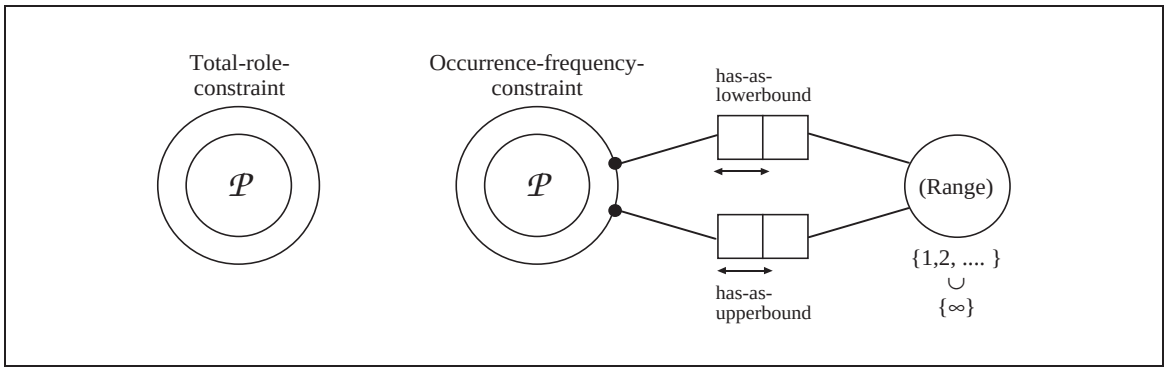

Figure 5.25: Total role and occurrence frequency constraints

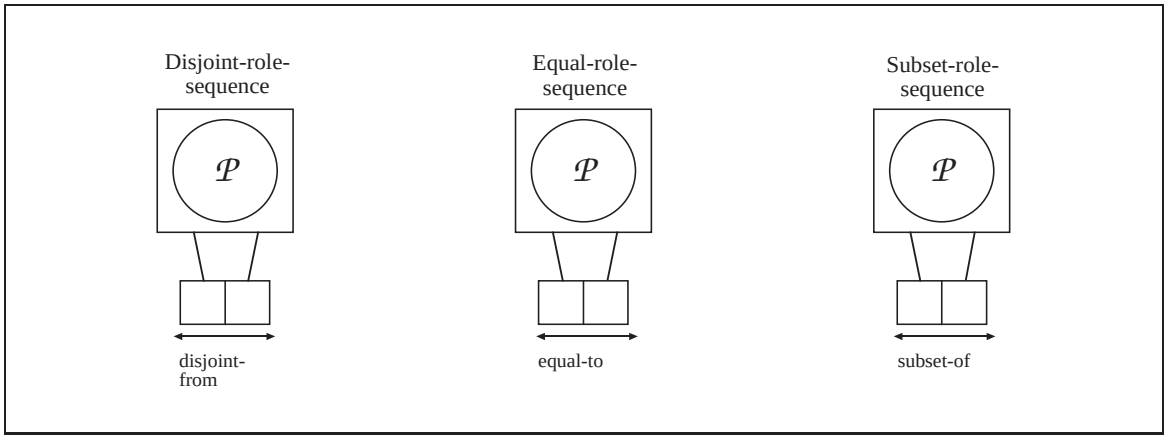

Figure 5.26: Set constraints

can be modelled by means of unary fact types associated to object type $\mathcal{O}\backslash\mathcal{L}$. They are not associated to object type $\mathcal{G}$, since they may also be specified for subtypes of power types. If for an object type a set cardinality constraint is specified, then both an upper bound and a lower bound must be specified. This is ensured by the equality constraint. The membership constraint is captured by a binary fact type on $\mathcal{P}$, as syntactically it consists of two predicators.

The specialisation constraints are modelled in the schema of figure 5.29. Both the subtype exclusion constraint and the total subtype constraint can be modelled by means of power types having as element type *Subtype*. Again, the constraint stating that the subtypes involved in these constraints have to come from the same specialisation hierarchy, cannot be expressed.

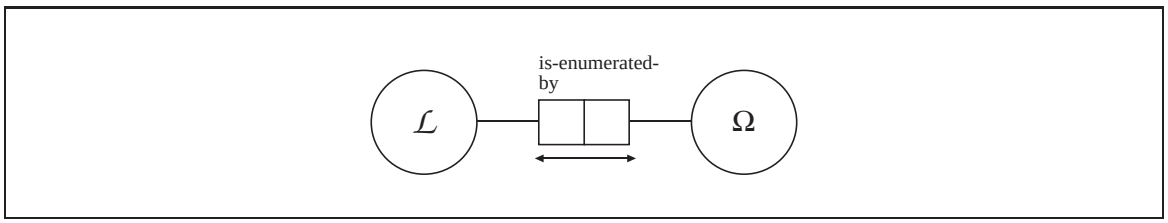

Figure 5.27: Enumeration constraints

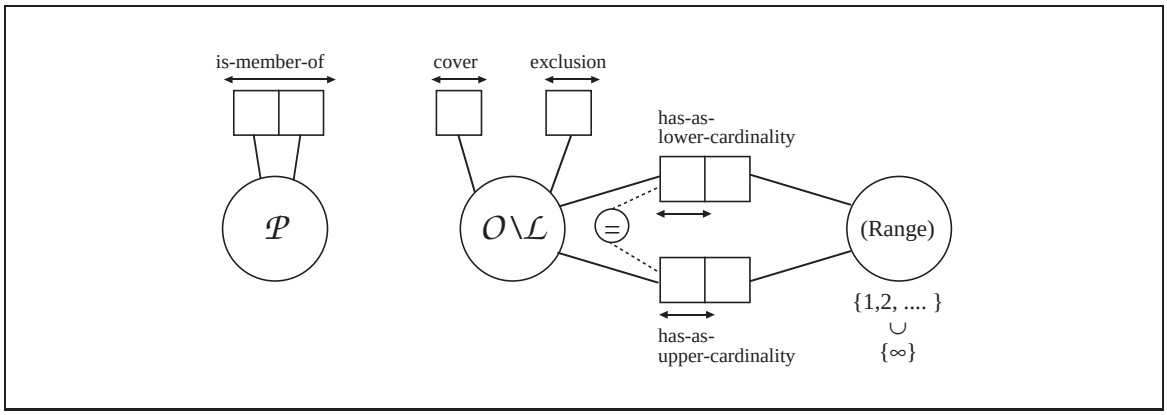

Figure 5.28: Power type constraints

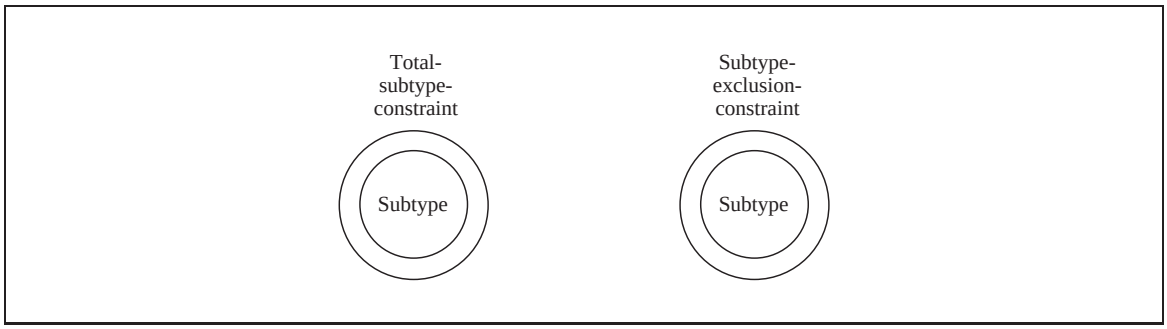

Figure 5.29: Specialisation constraints

Object types can be populated with instances from Ω, see figure 5.30. The many rules that populations must fulfil cannot be graphically expressed.

5.8 Motivation from the axioms of set theory

In this section, set theory (see e.g. [16]) is compared with the type construction mechanisms in data models. The purpose of this comparison is to further investigate expressive power and to motivate the type construction mechanisms.

There are two fundamental differences between data models and set theory:

1. In set theory every object is a set, elements of sets are also sets. Data models on the other hand are

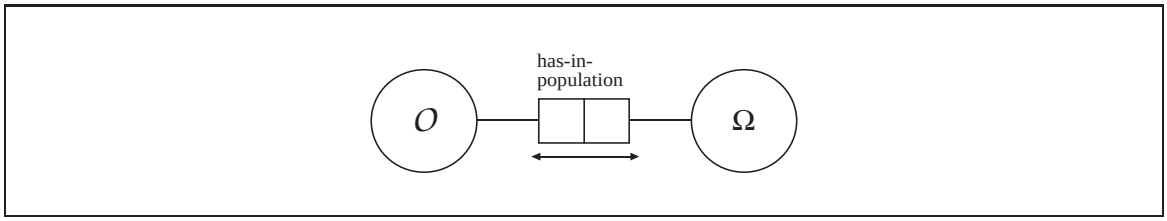

Figure 5.30: Populations

based on the type-instance dichotomy.

2. The Axiom of Extensionality,

$$\forall_z [z \in x \iff z \in y] \Rightarrow x = y,$$

states that if two sets contain exactly the same elements, they are equal. A set is therefore completely characterised by its *extension*, i.e. its elements. In data models however, object types are not characterised by their instances. Two different object types may have the same instantiation in some population. An object type is characterised by its *intension*, which is reflected in its name.

The relevant axioms of set theory (see e.g. [16]) can be compared with the type construction mechanisms in data models. The results are found in [35].

5.9 Data models for context-free grammars

In this section, a translation of context-free grammars to data schemata is described. This demonstrates that the data models we use are at least as expressive as context-free grammars. Since context-free grammars are often employed for describing hypertext information structures (see for example [9], [66]), the translation also shows the suitability of data models for describing hypertext information structures.

5.9.1 Definition of context-free grammars

A context-free grammar G is a tuple $\langle N, \Sigma, \Pi, S \rangle$, where N is a finite set of nonterminal symbols, Σ is a finite set of terminal symbols, $S \in N$ is the initial symbol and Π is a set of production rules of the form $A \to \omega$ where $A \in N$ and $\omega \in (N \cup \Sigma)^*$.

In the remainder, only production rules with a nonempty right-hand side are considered. This restriction is not very severe, as only the possibility of generating the empty string is lost (see for example [34]). The empty string would correspond to an information structure without object types.

5.9.2 Translating a grammar to a data model

We describe the translation Δ of a context-free grammar G to a schema $\Delta(G)$. Each symbol of G is interpreted as an entity type. Each terminal symbol is interpreted as an entity type, which can be directly identified by a label type. Let x be a terminal symbol, then its associated entity type x is identified by label type l_x (see figure 5.31).

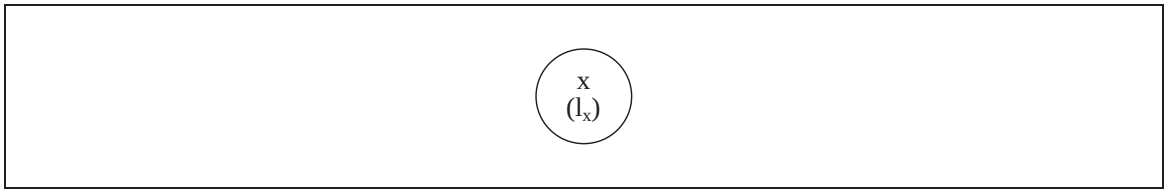

Figure 5.31: Translation of terminals

Let $P \in \Pi$ be a production rule, and let P be of the form $t \to s_1, \ldots, s_n$. This rule results in a generalisation relation in $\Delta(G)$, where entity type t is a generalised object type having as specifier an objectified fact type F_P. This fact type consists of predicators $P^1_{s_1}, \ldots, P^n_{s_n}$, where for each $1 \leq i \leq n$: $\mathsf{Base}(P^i_{s_i}) = s_i$. In figure 5.32, the translation of production rule P is depicted graphically. This concludes the description of translation Δ. Note that power types do not result from this translation.

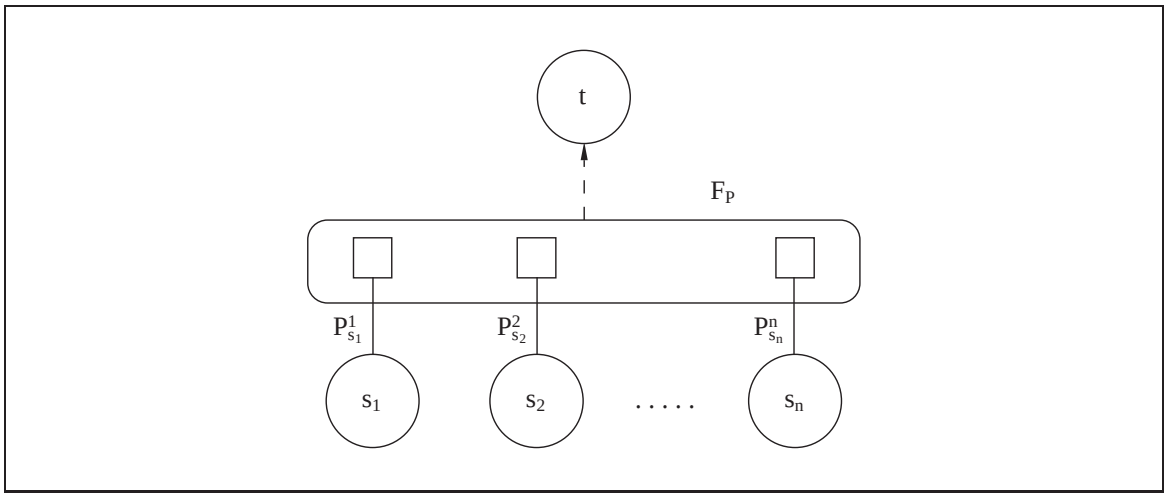

Figure 5.32: Translation of production rules of the form $t \rightarrow s_1, \ldots, s_n$

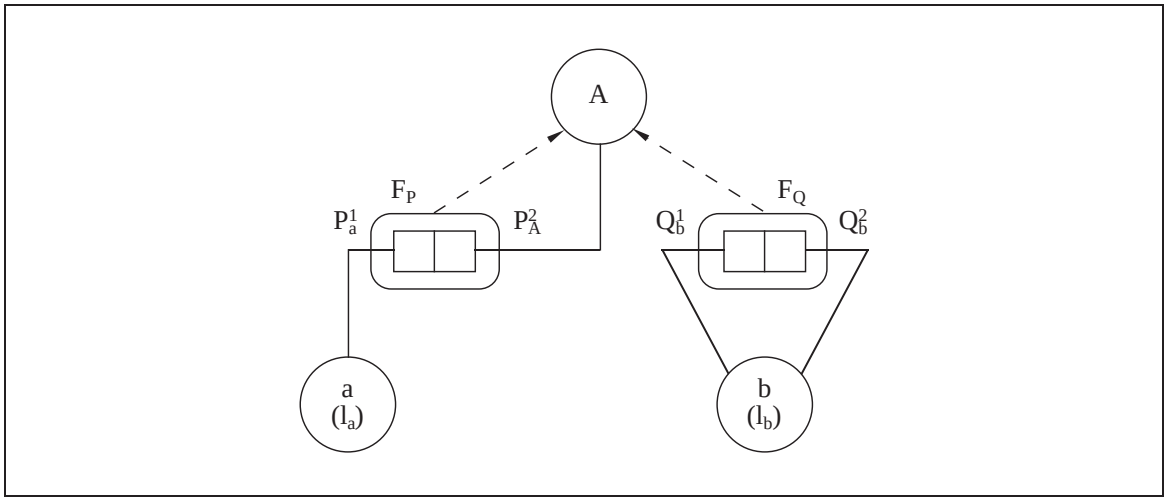

Figure 5.33: Example of translation of context-free grammar

As an example, in figure 5.33 the resulting schema $\Delta(G)$ is shown for the context-free grammar G with the following production rules:

$$\begin{aligned} P : A &\rightarrow aA \\ Q : A &\rightarrow bb \end{aligned}$$

It is important to note that the schema resulting from the translation of a context-free grammar does not explicitly show the order of the symbols in the right-hand side of production rules. This corresponds to a *mapping oriented* view to the right-hand side of a production rule, rather than the usual tuple oriented view.

5.9.3 Sequence types in translated grammars

In some cases it is convenient to allow grammar rules of the form $t \rightarrow s^+$. A rule $t \rightarrow s^+$ is a shorthand for the rules $t \rightarrow s$ and $t \rightarrow st$. When such rules are employed, sequence types can be used in the translation

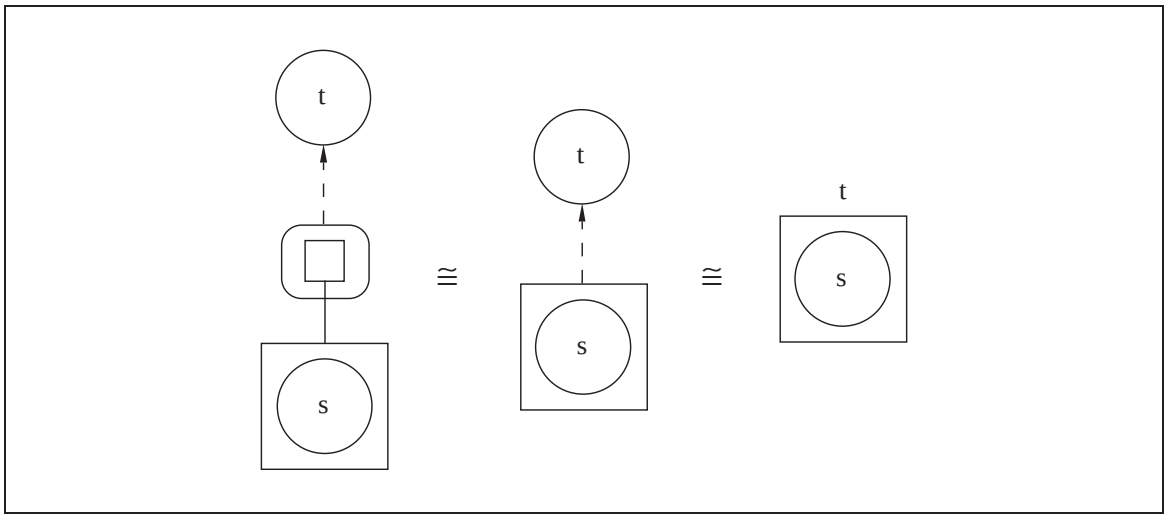

Figure 5.34: A simplified translation of $t \to s^+$

of a grammar. In figure 5.34 a simplified translation of the production rule $t \to s^+$ is shown, in case this rule is the only production rule for nonterminal t.

The following example makes full use of this simplified translation. The schema of figure 5.35 is the result of the translation of the grammar in the style of SGML ([45]) for describing the structure of a book:

$$\begin{aligned} \text{book} &\to \text{title contents} \\ \text{contents} &\to \text{chapter}^+ \\ \text{chapter} &\to \text{title sections} \\ \text{sections} &\to \text{section}^+ \\ \text{section} &\to \text{string} \\ \text{title} &\to \text{string} \\ \text{string} &\to \text{char}^+ \end{aligned}$$

Grammars can be incorporated more concisely in a data schema via the grammar box. An example of a grammar box is shown in figure 5.36. The grammar box takes as inputs the object types that correspond to terminal symbols. The output of the grammar box is the object type corresponding to the start symbol.

In the schema in figure 5.35, a data model for a book is shown. In case more semantic properties of books (or documents in general) have to be incorporated, it is possible to apply XML. An example of this approach is found in e.g. [25].

5.9.4 Productive grammars and object identification

Grammars can have nonterminals that are not productive. A nonterminal x is called not productive if it cannot generate a sequence of terminals (i.e. $L(x) = \varnothing$). A grammar with only productive nonterminal symbols is called universally productive. The test of this property can be done in linear time [34].

The following theorem shows that a schema resulting from the translation of a context-free grammar is structurally identifiable if and only if that context-free grammar is universally productive. Let $G = \langle N, \Sigma, \Pi, S \rangle$ be a context-free grammar and let $\Delta(G)$ be the corresponding schema, then:

$$\forall_{T \in N} \exists_{\omega \in \Sigma^*} \left[T \overset{*}{\to} \omega \right] \iff \forall_{x \in N} \left[\mathsf{Identifiable}(x) \right]$$

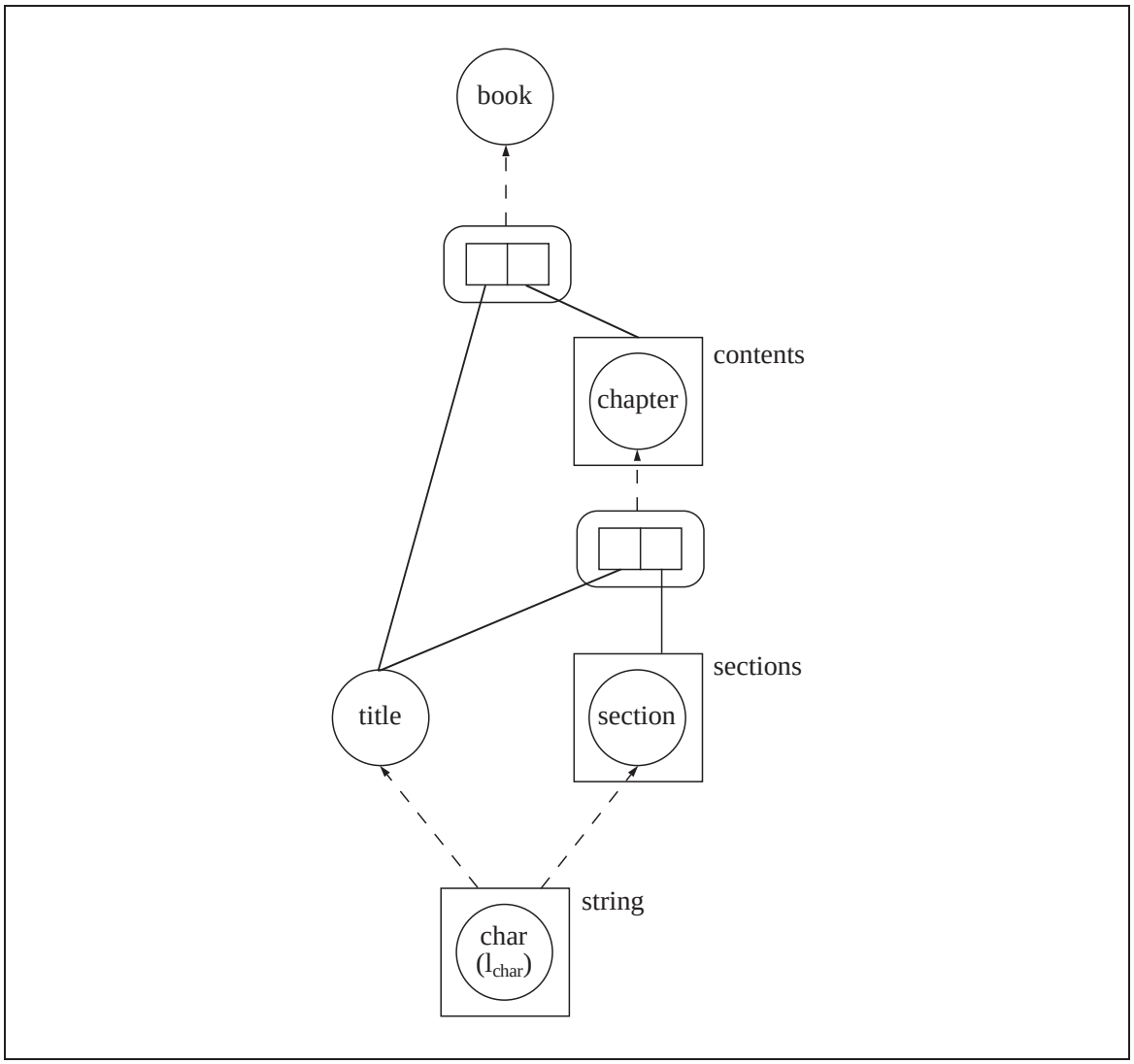

Figure 5.35: Example of translation of SGML structure

5.9.5 Context-sensitive grammars and data models

Having embedded context-free grammars in data models, it is interesting to note that these data models can handle some forms of context sensitivity as well. To fully deal with context sensitivity however, a powerful constraint language is needed.

As an example, consider a small fragment of the syntax of a programming language, the part that deals with the declaration of variables and the assignment of expressions to variables. This fragment is described by the schema of figure 5.37. A declaration statement defines the data type of a variable. An assignment statement assigns the value of an expression to a variable. The subset constraint expresses that a variable has to be declared when it acts as a source of an assignment statement. This obviously is a property that can be handled by a context-sensitive grammar, but not by a context-free grammar. On the other hand, the requirement that the type of an expression should correspond to the type of the variable in an assignment statement, cannot be expressed in our data models (in general).

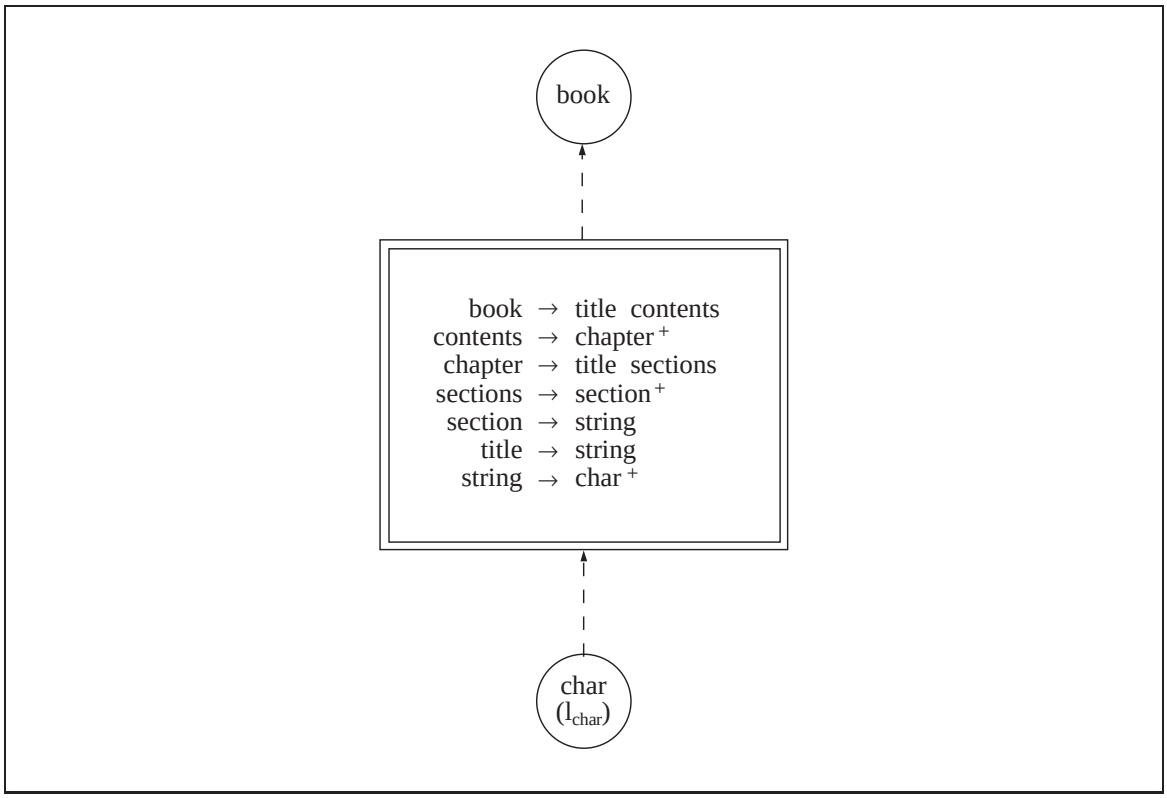

Figure 5.36: A grammar box

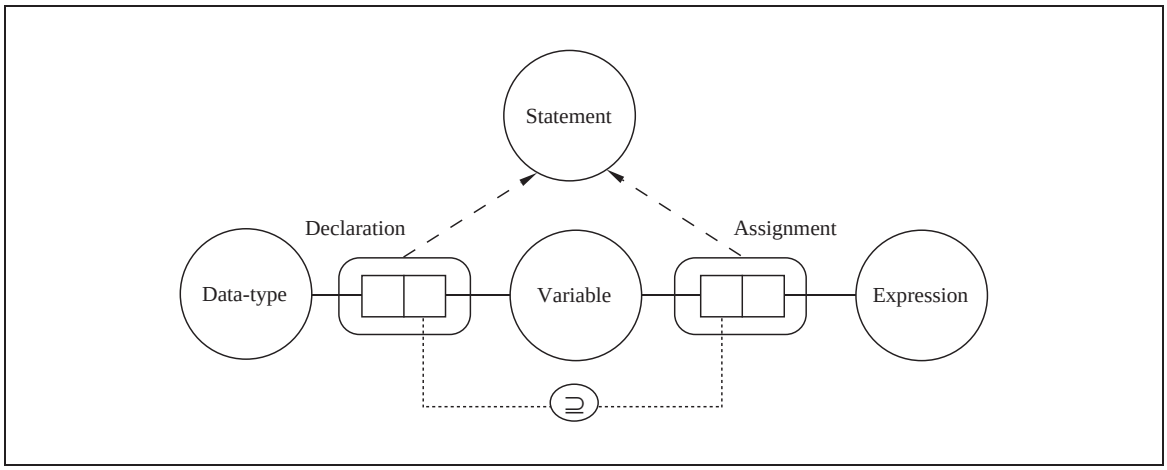

Figure 5.37: A fragment of the syntax of a programming language

Chapter 6

Path expressions

6.1 Introduction

The process of information analysis starts with number of sentences capturing the structure of the Universe of Discourse. The names of object types and roles originate from these sentences. Consequently, using these names in the formulation of queries and constraints yields (generally speaking) results very close to the original formulation in natural language of these queries and constraints.

The first language to pursue this idea was introduced in [18] and [55]. This language, however, never received much acceptance as it did not have a formal syntax and semantics and was based on the restricted binary version of object-role models (see e.g. [72]).

In this chapter, we introduce a language for path expressions through information structures. This language can be considered as a redesign of the original language mentioned above, with a functionality far exceeding that of the original. The language of path expressions is quite primitive, yet relatively powerful (more powerful than the relational algebra defined in section 4.11). As path expressions are based on multisets, this chapter starts with a formal definition of multisets and operations on multisets (section 6.2). Section 6.3 then presents the formal definition of path expressions. The set of path expressions for a given information structure $\mathcal{I}$, is denoted as $\boldsymbol{\mathcal{PE}}(\mathcal{I})$.

6.2 Multisets

Multisets [52], also known as *multiple membership sets* [51], or *bags* [60], differ from ordinary sets in that a multiset may contain one element more than once. A multiset can be denoted by an enumeration of its elements, e.g. $\{\!\{a, a, b, c, d, c, a\}\!\}$. The empty multiset then simply corresponds to the empty set $\varnothing$.

6.2.1 Basic multiset axioms

Other than through enumeration, multisets can be constructed by application of the union or difference on other multisets. If X is a multiset, then $\#(a, X)$ denotes the number of occurrences of a in X. This operator is subject to the following axioms:

[MS1] $\#(a, \varnothing) = 0$

[MS2] $\#(a, \{\!\{b\}\!\}) = \textbf{if } a = b \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi}$

[MS3] $\#(a, X \cup Y) = \#(a, X) + \#(a, Y)$

[MS4] $\#(a, X \backslash Y) = \max(0, \#(a, X) - \#(a, Y))$

Two multisets are equal if they contain the same elements the same number of times:

[MS5] $\forall_a [\#(a, X) = \#(a, Y)] \Rightarrow X = Y$

6.2.2 Basic multiset operators

The membership operator for multisets takes the occurrence frequency of elements in a multiset into account:

$$a \in^n X \iff \#(a, X) = n$$

In the remainder of this chapter, $a \in X$ is used as a shorthand for $\#(a, X) > 0$. As a result, $a \notin X$ is equivalent to $a \in^0 X$ (or $\#(a, X) = 0$). Multiset X is a subset of multiset Y, $X \subseteq Y$, if and only if:

$$\forall_a [\#(a, X) \leq \#(a, Y)]$$

Multiset X is a proper subset of multiset Y, $X \subset Y$, if and only if:

$$X \subseteq Y \wedge X \neq Y$$

The intersection of two multisets can be defined using the difference operator:

$$X \cap Y = X \backslash\!\backslash (X \backslash Y)$$

Lemma 6.1 $\#(a, X \cap Y) = \min(\#(a, X), \#(a, Y))$

Proof:

$$\begin{aligned} \#(a, X \cap Y) &= \max(0, \#(a, X) - \#(X \backslash Y)) \\ &= \max(0, \#(a, X) - \max(0, \#(a, X) - \#(a, Y))) \\ &= \textbf{if } \#(a, X) \geq \#(a, Y) \textbf{ then } \#(a, Y) \textbf{ else } \#(a, X) \textbf{ fi} \\ &= \min(\#(a, X), \#(a, Y)) \end{aligned}$$

□

6.2.3 Denotation

Nonempty multisets can be denoted in two different ways. The extensional denotation simply enumerates the elements of the multiset:

$$\{\!|a_1, \ldots, a_n|\!\} = \{\!|a_1|\!\} \cup \ldots \cup \{\!|a_n|\!\}$$

The second kind of denotation is based on bag comprehension. Let $C(a, n)$ be a predicate such that for each a exactly one n exists, such that $C(a, n)$. A multiset can then be denoted by means of the bag comprehension schema [5]:

$$\{\!|a\uparrow^n \mid C(a, n)|\!\}$$

This set is an intensional denotation of the multiset X that is determined by:

$$C(a, n) \iff a \in^n X$$

6.2.4 Further operators

Doubling a multiset is defined as follows:

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

The projection of a multiset M of pairs on its first component is defined as:

$$\pi_1(M) = \bigcup_y \{\!\!\{ x\uparrow^n \mid \langle x, y\rangle \in^n M \}\!\!\}$$

The projection on the second component, $\pi_2(M)$, can be defined analogously.

The operator Lin converts a tuple to the corresponding multiset:

$$\mathsf{Lin}(x) = \bigcup_{1\leq i\leq |x|} \{\!\!\{ x_{<i>} \}\!\!\}$$

Example 6.1
$\mathsf{Lin}(\langle a, b, c, d, a\rangle) = \{\!\!\{ a, a, b, c, d\}\!\!\}$ □

A special version of the union-operator, operating on a multiset of sets, is:

$$\biguplus M = \left\{\!\!\left\{ x\uparrow^n \;\middle|\; n = \sum_{A\in^i M\wedge x\in A} i \right\}\!\!\right\}$$

Example 6.2
$\biguplus \{\!\!\{\{a, b\}, \{a, b\}, \{b, c\}\}\!\!\} = \{\!\!\{ a, a, b, b, b, c\}\!\!\}$ □

A set can be coerced to a multiset by means of the Multi operator:

$$\mathsf{Multi}(X) = \{\!\!\{ a\uparrow^1 \mid a \in X \}\!\!\}$$

Conversely, a multiset can be coerced to a set by means of the Set operator:

$$\mathsf{Set}(X) = \{ a \mid a \in X \}$$

The size of a multiset X is defined as:

$$|X| = \sum_a \#(a, X)$$

Example 6.3
$|\{\!\!\{ a, a, a, b, b\}\!\!\}| = 5$ □

6.3 Syntax and semantics of path expressions

Path expressions describe derived fact types in a style 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 the rest of this chapter we assume the existence of a fixed information structure $\mathcal{I}$). 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, although needed for the evaluation of path expressions, are discarded in their final result. The advantage of this approach is uniformity, as it 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 are, in general, *multisets* of tuples. More complex forms of path expressions may be inhomogeneous, i.e. their evaluation may lead to tuples from different domains. Path expressions are therefore interpreted as inhomogeneous binary multiset relations.

6.3.1 Overview and motivation

The syntax of path expressions is presented as an abstract syntax. In [56] the motivation for the use of an abstract syntax is stated as follows:

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

The semantics of path expressions is defined using denotational semantics (see e.g. [68]). The semantics of each syntactical construct is 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, which represents the state of a program. In the case of path expressions, the environment is the population of the information structure.

As a simple path expression corresponds to a (directed) path through the information structure, a simple path expression is interpreted as the description of a relation between the object type at its begin and the object type at its end point. As a result of uniting simple path expressions with different begin and end points, path expressions, however, may be inhomogeneous. Such path expressions lead to inhomogeneous binary relations. Consequently, the semantics of a path expression is defined as a binary relation over (multiple) object types. We found it convenient to treat these binary relations tuple oriented, as opposed to the mapping oriented approach to tuples in the population of fact types. As a result, the domain for these inhomogeneous binary multiset relations is derived from Ω in the following way:

$$\Omega_{\boldsymbol{\mathcal{PE}}} = \left\{ X \mid \mathsf{Set}(X) \subseteq \Omega^2 \right\}$$

The definition of path expressions uses the following syntactical categories: constant, multiset, object type ($\mathcal{O}$), predicator ($\mathcal{P}$) and path expression ($\boldsymbol{\mathcal{PE}}(\mathcal{I})$). The naming conventions are: c for constants, M for multisets, X, $X_1, \ldots, X_n$ for object types, p for predicators and P, $P_1, \ldots, P_n$, G, S and Q for path expressions. The function

$$\mu : \boldsymbol{\mathcal{PE}} \times \mathsf{POP}_{\mathcal{I}} \rightarrow \Omega_{\boldsymbol{\mathcal{PE}}}$$

is used to define the semantics of path expressions.

6.3.2 Atomic path expressions

First we introduce the atomic path expressions. The empty path expression ($\varnothing_{\boldsymbol{\mathcal{PE}}}$) and the neutral path expression ($1_{\boldsymbol{\mathcal{PE}}}$) are atomic path expressions. Constants, multisets, object types and predicators can also be interpreted as atomic path expressions. In the following definition, the operator $\cdot$ represents functional composition.

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

Example 6.4

Let Pop *be the sample population, defined in section 2.2.3, of the information structure diagram of figure 2.1, then:*

$$\mu[\![16]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 16 & 16 \\ \hline \end{array}$$

$$\mu[\![\{\!|5,5,7|\!\}]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 5 & 5 \\ 5 & 5 \\ 7 & 7 \\ \hline \end{array}$$

$$\mu[\![f]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline \{p:b_1, q:a_1\} & \{p:b_1, q:a_1\} \\ \{p:b_1, q:a_2\} & \{p:b_1, q:a_2\} \\ \hline \end{array}$$

$$\mu[\![u]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline \{a_1\} & \{u:\{a_1\}, v:b_1\} \\ \{a_1, a_2\} & \{u:\{a_1, a_2\}, v:g_1\} \\ \hline \end{array}$$

□

6.3.3 Unary operators

A number of operators and functions are available for the construction of composed path expressions. First we introduce the unary operators. They allow for the reversal of a path ($^{\leftarrow}$), the isolation of the front elements of a path ($\not\mathcal{F}$), the removal of multiple occurrences (ds), the determination of the number of elements in a path expression (Cnt), the addition of the elements in a path expression (Sum) and the determination of the minimum or maximum element in a path expression (Min and Max). The power set of a path expression P, $\wp P$, yields a path expression with all sets of instances occurring in the first component of P. The unary operators are defined in the following table:

name	expr	$\mu[\![\mathsf{expr}]\!]\,(\mathsf{Pop})$
reverse	$P^{\leftarrow}$	$\{\!\{\langle q,p\rangle\!\uparrow^n \mid \langle p,q\rangle \in^n \mu[\![P]\!]\,(\mathsf{Pop})\}\!\}$
front	$\not\mathcal{F} P$	$\mathsf{Sqr}\cdot\pi_1\cdot\mu[\![P]\!]\,(\mathsf{Pop})$
distinct	$\mathsf{ds}\,P$	$\mathsf{Multi}\cdot\mathsf{Set}\cdot\mu[\![P]\!]\,(\mathsf{Pop})$
count	$\mathsf{Cnt}\,P$	$\mathsf{Sqr}\,(\{\!\{\lvert\mu[\![P]\!]\,(\mathsf{Pop})\rvert\}\!\})$
sum	$\mathsf{Sum}\,P$	$\mathsf{Sqr}\,(\ \{\!\{\sum_{x\in^n \pi_1\cdot\mu[\![P]\!]\,(\mathsf{Pop})} n\times x\}\!\}\ )$
minimum	$\mathsf{Min}\,P$	$\mathsf{Sqr}\,(\{\!\{\min\cdot\pi_1\cdot\mu[\![P]\!]\,(\mathsf{Pop})\}\!\})$
maximum	$\mathsf{Max}\,P$	$\mathsf{Sqr}\,(\{\!\{\max\cdot\pi_1\cdot\mu[\![P]\!]\,(\mathsf{Pop})\}\!\})$
power set	$\wp P$	$\mathsf{Sqr}\,(\{\!\{i\!\uparrow^1 \mid i\subseteq\pi_1\cdot\mu[\![P]\!]\,(\mathsf{Pop})\}\!\})$

Example 6.5

In the situation of the previous example:

$$\mu[\![u^{\leftarrow}]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline \{u:\{a_1\}, v:b_1\} & \{a_1\} \\ \{u:\{a_1, a_2\}, v:g_1\} & \{a_1, a_2\} \\ \hline \end{array}$$

$$\mu[\![\not\mathcal{F}\, p]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline b_1 & b_1 \\ b_1 & b_1 \\ \hline\end{array}$$

$$\mu[\![\mathsf{ds}(\not\mathcal{F}\, p)]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline b_1 & b_1 \\ \hline\end{array}$$

$$\mu[\![\mathsf{Cnt}(\not\mathcal{F}\, p)]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 2 & 2 \\ \hline\end{array}$$

$$\mu[\![\mathsf{Cnt}(\mathsf{ds}(\not\mathcal{F}\, p))]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 1 & 1 \\ \hline\end{array}$$

The last two examples show that not always $\mathsf{Cnt}(P) = \mathsf{Cnt}(\mathsf{ds}(P))$. *In the case that* $\mathsf{Pop}(i) = \{\{w : b_1, x : 17\}, \{w : b_1, x : 18\}\}$ *instead of* $\{\{w : b_1, x : 17\}\}$*:*

$$\mu[\![\mathsf{Sum}\, x]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 35 & 35 \\ \hline\end{array}$$

$$\mu[\![\mathsf{Min}\, x]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 17 & 17 \\ \hline\end{array}$$

$$\mu[\![\mathsf{Max}\, x]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 18 & 18 \\ \hline\end{array}$$

$$\mu[\![\wp(v)]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline \varnothing & \varnothing \\ \{b_1\} & \{b_1\} \\ \{g_1\} & \{g_1\} \\ \{b_1, g_1\} & \{b_1, g_1\} \\ \hline\end{array}$$

□

6.3.4 Extension of paths

Paths can be extended in several ways. Concatenation ($\circ$) is the most elementary operator for extending paths. The Cartesian product ($\times$) uses the begin values of both path expressions involved. Furthermore, the usual set operators ($\cap$, $\cup$ and $-$) are available. These operators are formally described in the following table:

name	expr	$\mu[\![\mathsf{expr}]\!]\,(\mathsf{Pop})$
concatenate	$P \circ Q$	$\bigcup_r \{\!\{\langle p, q\rangle\uparrow^{n\times m} \mid \langle p, r\rangle \in^n \mu[\![P]\!]\,(\mathsf{Pop}) \wedge \langle r, q\rangle \in^m \mu[\![Q]\!]\,(\mathsf{Pop})\}\!\}$
product	$P \times Q$	$\bigcup_{r,s} \{\!\{\langle p, q\rangle\uparrow^{n\times m} \mid \langle p, r\rangle \in^n \mu[\![P]\!]\,(\mathsf{Pop}) \wedge \langle q, s\rangle \in^m \mu[\![Q]\!]\,(\mathsf{Pop})\}\!\}$
intersection	$P \cap Q$	$\mu[\![P]\!]\,(\mathsf{Pop}) \;\cap\; \mu[\![Q]\!]\,(\mathsf{Pop})$
union	$P \cup Q$	$\mu[\![P]\!]\,(\mathsf{Pop}) \;\cup\; \mu[\![Q]\!]\,(\mathsf{Pop})$
minus	$P - Q$	$\mu[\![P]\!]\,(\mathsf{Pop}) \;\setminus\; \mu[\![Q]\!]\,(\mathsf{Pop})$

Example 6.6

If Pop *is the population of section 2.2.3, then:*

$$\mu[\![p \circ q^{\leftarrow}]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline b_1 & a_1 \\ b_1 & a_2 \\ \hline\end{array}$$

$$\mu[\![v \circ u^{\leftarrow} \circ \in_E^p \circ \in_E^{e\,\leftarrow}]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline b_1 & a_1 \\ g_1 & a_1 \\ g_1 & a_2 \\ \hline\end{array}$$

$$\mu[\![A \times (\not\mathcal{F}\, p)]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline a_1 & b_1 \\ a_1 & b_1 \\ a_2 & b_1 \\ a_2 & b_1 \\ \hline\end{array}$$

$$\mu[\![C \cap D]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline b_1 & b_1 \\ \hline \end{array}$$

$$\mu[\![p \cup G]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline b_1 & \{p : b_1, q : a_1\} \\ \hline b_1 & \{p : b_1, q : a_2\} \\ \hline g_1 & g_1 \\ \hline \end{array}$$

$$\mu[\![C - D]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline g_1 & g_1 \\ \hline \end{array}$$

□

6.3.5 Many-sorted algebra

Functions and binary relations from the many-sorted algebra $\mathcal{D} = \langle D, F\rangle$ can be used in path expressions.

name	expr	$\mu[\![\mathsf{expr}]\!]\,(\mathsf{Pop})$
function application	$f(P_1, \ldots, P_n)$	see below
binary relation	$P\,\mathrm{R}\,Q$	see below

If f is an n-ary function, then function application of f on n path expressions yields a path expression where the front elements result from function application on all possible front element combinations of the path expressions involved.

$$\mu[\![f(P_1, \ldots, P_n)]\!]\,(\mathsf{Pop}) = \\ \left\{\!\!\left[\langle f(x_1, \ldots, x_n), x\rangle\!\uparrow^k \;\middle|\; \forall_{1\le i\le n}\,[x_i \in \pi_1 \cdot \mu[\![P_i]\!]\,(\mathsf{Pop})] \wedge \langle x_n, x\rangle \in^k \mu[\![P_n]\!]\,(\mathsf{Pop})\right]\!\!\right\}$$

Example 6.7

For any population Pop*:*

$$\mu[\![45 + 25]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 70 & 25 \\ \hline \end{array}$$

□

If R is a binary relation, then path expression $P\,\mathrm{R}\,Q$ is a concatenation of P and Q via elements of R.

$$\mu[\![P\,\mathrm{R}\,Q]\!]\,(\mathsf{Pop}) = \\ \bigcup_{x,y} \left\{\!\!\left[\langle v, w\rangle\!\uparrow^{n\times m} \;\middle|\; \langle v, x\rangle \in^n \mu[\![P]\!]\,(\mathsf{Pop}) \wedge \langle y, w\rangle \in^m \mu[\![Q]\!]\,(\mathsf{Pop}) \wedge \langle x, y\rangle \in \mathrm{R}\right]\!\!\right\}$$

Example 6.8

Let P and Q be path expressions and Pop *a population such that:*

$$\mu[\![P]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline a_1 & 17 \\ \hline a_2 & 19 \\ \hline \end{array}$$

$$\mu[\![Q]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline 16 & b_1 \\ \hline 20 & b_2 \\ \hline 18 & b_3 \\ \hline 18 & b_3 \\ \hline \end{array}$$

Then:

$$\mu[\![P < Q]\!]\,(\mathsf{Pop}) = \begin{array}{|c|c|}\hline a_1 & b_2 \\ \hline a_1 & b_3 \\ \hline a_1 & b_3 \\ \hline a_2 & b_2 \\ \hline \end{array}$$

□

6.3.6 Type conversions

Special constructs are available for data type conversions. Elements in a path expression can be grouped according to a grouping criterion, they can be ungrouped and they can be ordered according to an ordering criterion.

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

Grouping path expression P, according to a grouping criterion G, is performed by application of the function φ on P and G. The elements to be grouped are part of the first component of path expression P. Path expression G specifies a grouping criterion for these elements. Suppose $g \in \pi_2 \cdot \mu[\![G]\!](\mathsf{Pop})$, then K_g is defined by:

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

The result of grouping P according to G can then be defined as:

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

Example 6.9

Suppose that P is a path expression that evaluates, in the context of a population Pop*, to:*

$$\mu[\![P]\!](\mathsf{Pop}) = \begin{array}{|l|l|}\hline \textit{alice} & * \\ \textit{john} & * \\ \textit{mary} & * \\ \textit{peter} & * \\ \textit{jane} & * \\ \hline \end{array}$$

The first component of P contains a number of persons, while the second component is ignored as it is of no interest for this example. Path expression G records the programming languages persons are familiar with:

$$\mu[\![G]\!](\mathsf{Pop}) = \begin{array}{|l|l|}\hline \textit{alice} & \textit{cobol} \\ \textit{alice} & \textit{pascal} \\ \textit{john} & \textit{cobol} \\ \textit{mary} & \textit{pascal} \\ \textit{peter} & \textit{fortran} \\ \textit{bert} & \textit{basic} \\ \hline \end{array}$$

Grouping P according to G leads to:

$$\mu[\![\varphi(P,G)]\!](\mathsf{Pop}) = \begin{array}{|l|l|}\hline \{\textit{alice, john}\} & \textit{cobol} \\ \{\textit{alice, mary}\} & \textit{pascal} \\ \{\textit{peter}\} & \textit{fortran} \\ \hline \end{array}$$

Note the absence of both jane *and* bert *in the grouping. Ungrouping the result of $\varphi(P,G)$ yields:*

$$\mu[\![\Upsilon(\varphi(P,G))]\!](\mathsf{Pop}) = \begin{array}{|l|l|}\hline \textit{alice} & \textit{alice} \\ \textit{alice} & \textit{alice} \\ \textit{john} & \textit{john} \\ \textit{mary} & \textit{mary} \\ \textit{peter} & \textit{peter} \\ \hline \end{array}$$

□

Example 6.10
A partition of singletons is obtained by $\varphi(P, \not\digamma\, P)$. □

Example 6.11
Path expression $\varphi(P, P \times c)$*, where* c *is an arbitrary constant, consists of one instance having as first component a set containing all instances occurring in the first component of path expression* P *and as second component constant* c. □

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

1. s contains all elements of $\pi_1 \cdot \mu[\![P]\!]\,(\mathsf{Pop})$ in the same frequency:
$$\mathsf{Lin}(s) = \pi_1 \cdot \mu[\![P]\!]\,(\mathsf{Pop}),$$
2. the order of elements in s does not conflict with the ordering rules from S, i.e. for each i, j such that $0 \leq i < j < |s|$:
$$\exists_{y_1,y_2}\,[\langle s_{<i>}, y_1\rangle \in \mu[\![P]\!]\,(\mathsf{Pop}) \wedge \langle s_{<j>}, y_2\rangle \in \mu[\![P]\!]\,(\mathsf{Pop}) \wedge \langle y_2, y_1\rangle \notin \mu[\![S]\!]\,(\mathsf{Pop})]$$

The result of sorting now is defined as:

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

Example 6.12
Suppose that P *is a path expression that evaluates, in the context of a population* Pop*, to the following set of persons with their age:*

$$\mu[\![P]\!]\,(\mathsf{Pop}) = \begin{array}{|l|l|}\hline \textit{alice} & \textit{21} \\ \textit{john} & \textit{18} \\ \textit{mary} & \textit{39} \\ \textit{peter} & \textit{42} \\ \textit{jane} & \textit{27} \\ \hline \end{array}$$

These persons can be sorted on their age as follows:

$$\mu[\![\psi(P, Q < Q)]\!]\,(\mathsf{Pop}) = \mathsf{Sqr}\left(\boxed{\langle john, alice, jane, mary, peter\rangle}\right)$$

where $Q = \not\digamma(P^{\leftarrow})$. □

Example 6.13
If no sorting criterion ($S = \varnothing_{\mathcal{PE}}$) *is imposed on* P*, then* $\mu[\![\psi(P, \varnothing_{\mathcal{PE}})]\!]\,(\mathsf{Pop})$ *contains all orderings of elements in the first component of* $\mu[\![P]\!]\,(\mathsf{Pop})$. □

6.3.7 Transitive closure

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

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

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

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

Example 6.14
Let P be a path expression and Pop *a population such that:*

$$\mu[\![P]\!](\mathsf{Pop}) = \begin{array}{|c|c|}\hline a_1 & a_1 \\ a_1 & a_1 \\ a_1 & a_2 \\ a_2 & a_3 \\ \hline \end{array}$$

In that case:

$$\mu[\![P^+]\!](\mathsf{Pop}) = \begin{array}{|c|c|}\hline a_1 & a_1 \\ a_1 & a_2 \\ a_1 & a_3 \\ a_2 & a_3 \\ \hline \end{array}$$

□

6.3.8 Confluence

Typically, the confluence operator is used when different sorts of information are to be integrated. For example, name, day of birth, salary and address of each employee. The confluence operator can be used for the definition of *queries*.

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

If $P_1, \ldots, P_n$ and Q are path expressions then $[P_1, \ldots, P_n \mid Q]$ is a path expression, referred to as the confluence of $P_1, \ldots, P_n$ under Q. Informally, evaluation of the first component of this expression leads to sequences of elements, occurring in the first components of the P_i (where the i should correspond to the position of the element in the sequence), each related to the same element occurring in the second components of the P_i and also occurring in the first component of Q, while the second component then contains this shared element. Formally:

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

The effect of the condition can be neutralised by choosing $Q = 1_{\boldsymbol{\mathcal{PE}}}$. As a shorthand, we define: $[P_1, \ldots, P_n] = [P_1, \ldots, P_n \mid 1_{\boldsymbol{\mathcal{PE}}}]$.

Example 6.15
Suppose that P_1, P_2 and Q are path expressions and that Pop *is a population such that:*

$$\mu[\![P_1]\!](\mathsf{Pop}) = \begin{array}{|c|c|}\hline a_1 & x_1 \\ a_2 & x_1 \\ a_3 & x_2 \\ a_4 & x_3 \\ \hline \end{array}$$

$$\mu[\![P_2]\!](\mathsf{Pop}) = \begin{array}{|c|c|}\hline b_1 & x_1 \\ b_1 & x_1 \\ b_2 & x_2 \\ b_3 & x_2 \\ \hline\end{array}$$

$$\mu[\![Q]\!](\mathsf{Pop}) = \begin{array}{|c|c|}\hline x_1 & x_1 \\ x_2 & x_2 \\ \hline\end{array}$$

Then:

$$\mu[\![[P_1, P_2 \mid Q]]\!](\mathsf{Pop}) = \begin{array}{|c|c|}\hline \langle a_1, b_1\rangle & x_1 \\ \langle a_1, b_1\rangle & x_1 \\ \langle a_2, b_1\rangle & x_1 \\ \langle a_2, b_1\rangle & x_1 \\ \langle a_3, b_2\rangle & x_2 \\ \langle a_3, b_3\rangle & x_2 \\ \hline\end{array}$$

□

6.3.9 Additional operators

The elements occurring in a population are collected by the following path expression:

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

The definition of the active complement $\neg$ (see [54]) uses this path expression. Let P be a path expression. We then have:

$$\neg P \;=\; \mathsf{ActVals} - \not\exists P$$

A set (sequence) of path expressions can be converted to a path expression consisting of sets (sequences) constructed from the front elements, by using the set (sequence) constructor. The set and the sequence constructor are used for the denotation of instances from power and sequence types respectively.

name	expr	$\mu[\![\mathsf{expr}]\!](\mathsf{Pop})$
set construction	$\{P_1, \ldots, P_n\}$	$\mathsf{Double} \cdot \mathsf{Set}(\bigcup_{1 \le i \le n} \pi_1 \cdot \mu[\![P_i]\!](\mathsf{Pop}))$
sequence construction	$\langle P_1, \ldots, P_n\rangle$	$\mathsf{Sqr} \cdot \mathsf{Multi}(\{\langle x_1, \ldots, x_n\rangle \mid \forall_{1 \le i \le n}[x_i \in \pi_1 \cdot \mu[\![P_i]\!](\mathsf{Pop})]\})$

The function Double is defined as follows:

$$\mathsf{Double}(x) = \{\!\{\langle x, x\rangle\}\!\}$$

Usually the path expressions $P_1, \ldots, P_n$ used in a set or sequence construction contain only one value. It should be noted that we allow the number of path expressions involved in set or sequence construction to be zero.

Example 6.16

Let P, Q be path expressions and Pop *a population such that:*

$$\mu[\![P]\!](\mathsf{Pop}) = \begin{array}{|c|c|}\hline a_1 & b_1 \\ \hline\end{array}$$

$$\mu[\![Q]\!](\mathsf{Pop}) = \begin{array}{|c|c|}\hline c_1 & b_2 \\ \hline\end{array}$$

Then:

$$\mu[\![\{P,Q\}]\!]\,(\mathsf{Pop}) \;=\; \boxed{\{a_1,c_1\}}\boxed{\{a_1,c_1\}}$$

$$\mu[\![\langle P,Q\rangle]\!]\,(\mathsf{Pop}) \;=\; \boxed{\langle a_1,c_1\rangle}\boxed{\langle a_1,c_1\rangle}$$

□

The schema constructor allows for the definition of a path expression which is a denotation of a function of object types to path expressions. This constructor is used for the denotation of instances from schema types. In those cases, the path expressions involved (possibly zero) contain at most one element.

name	expr	$\mu[\![\mathsf{expr}]\!]\,(\mathsf{Pop})$
schema construction	$\mathsf{sc}\,(X_1 : P_1, \ldots, X_n : P_n)$	$\mathsf{Double}\,(\{\langle X_1, y_1\rangle, \ldots, \langle X_n, y_n\rangle\})$ with $y_i \in \pi_1 \cdot \mu[\![P_i]\!]\,(\mathsf{Pop})$ for each $1 \leq i \leq n$

In some cases, the evaluation of a path expression needs to remain interpretable as a path expression. For this purpose, we introduce the *freeze* operator †, which operates on elements of $\Omega_{\mathcal{PE}}$ (in the definition represented by $\omega_{\mathcal{PE}}$). Evaluation of this operator applied to an element of $\Omega_{\mathcal{PE}}$ simply yields this same element:

name	expr	$\mu[\![\mathsf{expr}]\!]\,(\mathsf{Pop})$
freeze	$\dagger\,\omega_{\mathcal{PE}}$	$\omega_{\mathcal{PE}}$

6.4 Information descriptors

In this section, we consider so-called *information descriptors*. These can be used for the specification of constraints, updates, and queries. Most examples in this section are taken from the schema of figure 6.1, which captures information about presidents of the USA. This Universe of Discourse was also used in e.g. [27] and was first enunciated in [77]. It should be noted that the schema of figure 6.1 only captures part of this Universe of Discourse. Furthermore, in this schema only keys are present, as other constraints are not important here.

One of the most important design criteria of information descriptors is readability. These descriptors should look natural (as much as possible) and their formal meaning should be close to their intuitive meaning. Furthermore, they should allow for elegant descriptions of information needs. Note that this does not imply the exclusion of "unelegant" descriptions. In fact, the language of information descriptors is a very liberal language, as not many expressions are syntactically excluded.

An important step in the definition of information descriptors is the introduction of names for the various mathematical objects occurring in the information structure at hand (e.g. object types). This makes the actual use of these abstract objects in information descriptors possible.

6.4.1 The assignment of names

Object types are assigned a name by the function

$$\mathsf{ONm} : \mathcal{O} \to \mathcal{N},$$

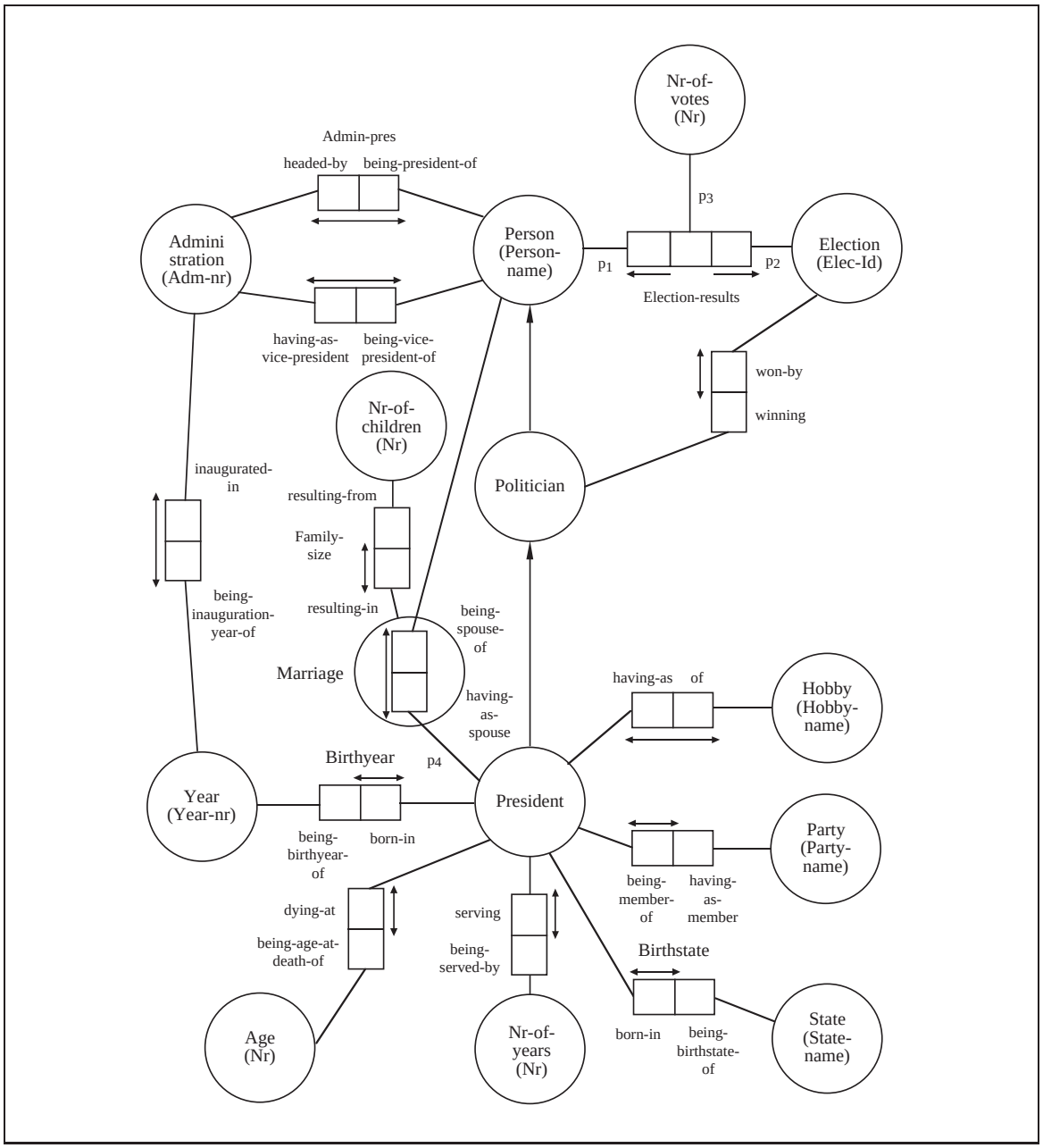

Figure 6.1: Part of an information structure dealing with American presidents

where $\mathcal{N}$ is a set of names. Different object types have different names:

$$\mathsf{ONm}(x_1) = \mathsf{ONm}(x_2) \Rightarrow x_1 = x_2$$

The partial function $\mathsf{Object} : \mathcal{N} \mapsto \mathcal{O}$ is the left-inverse of ONm, and relates object type names to their corresponding object type:

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

In order to improve readability, x rather than $\mathsf{Object}(x)$ is written, if no confusion can occur. The context should clarify whether x is used as an information descriptor, or as a shorthand for $\mathsf{Object}(x)$.

To predicators, an *identification name* may be assigned via the partial function

$$\mathsf{PNm} : \mathcal{P} \mapsto \mathcal{N}$$

Identification names should be unique for predicators belonging to the same fact type:

$$\mathsf{PNm}(p) = \mathsf{PNm}(q) \Rightarrow p = q \vee \mathsf{Fact}(p) \neq \mathsf{Fact}(q)$$

Predicators of different fact types may have identical identification names. The operator $. : \mathcal{N} \times \mathcal{N} \mapsto \mathcal{P}$ retrieves the predicator that is associated with a given identification name within a fact type (if existing):

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

For unique identification names, the fact type name may be omitted. Object type names and identification names should be different:

$$\mathsf{ran}(\mathsf{ONm}) \cap \mathsf{ran}(\mathsf{PNm}) = \varnothing$$

In binary versions of NIAM (e.g. [78] or [72]), special names are introduced for predicators, in order 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. To avoid confusion, we refer to role names as *connector names* as they correspond to special connections (which can be described by means of path expressions) through (binary) fact types. Connector names are recorded in the partial function

$$\mathsf{RNm} : \mathcal{P} \mapsto \mathcal{N}$$

Object type names and connector names should be different:

$$\mathsf{ran}(\mathsf{ONm}) \cap \mathsf{ran}(\mathsf{RNm}) = \varnothing$$

In figure 6.1 connector names are assigned to all predicators of binary fact types that are not a bridge type. Generally speaking, the connector name of a predicator is not identical to its identification name as it serves a different purpose.

As examples of the use of connector names, 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 procedure, and act as a natural language intermediate between application domain expert(s) and system analyst(s). Deep structure sentences can be interpreted uniquely if each valid combination Object-Name Role-Name Object-Name has a unique interpretation in the information structure. This is called the *Role Identification Rule* (see [78]). A combination nx np ny is valid if a predicator p exists such that:

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

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

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

The combination nx np ny has a unique interpretation in the information structure, if predicators p and q are unique. In binary NIAM, the restrictions that have to be imposed on the assignment of connector names to guarantee uniqueness of p and q are not very severe, though it should be noted that in the case of homogeneous symmetric binary fact types it is not always natural to be forced to assign different connector names to the predicators involved. In non-binary NIAM, however, the situation is different. Consider for example figure 4.3. The combination $\mathsf{ONm}(B)\,\mathsf{RNm}(r)\,\mathsf{ONm}(A)$ cannot be uniquely interpreted in this information structure, as it cannot be resolved whether one has to go from object type B to object type A via predicators r and p or via predicators r and q. Note that it does not matter which connector name is assigned to predicator r. We therefore choose a different approach for the interpretation of combinations nx np ny. In this approach, we allow arbitrary path expressions to be named. Connector names then simply correspond to path expressions of a specific form. Furthermore, ambiguity is resolved, by taking the union of all possible interpretations (see section 6.4.4), instead of by imposing restrictions on the assignment of names.

As a simple example consider the (ternary) election relation in figure 6.1. To find all persons contesting in an election, it would be natural to be able to formulate

Person contesting-in Election.

The name contesting-in then is used to denote the path expression $p_1 \circ {p_2}^{\leftarrow}$. Another example is

Nr-of-votes of Person.

In this expression, the name of is to be interpreted, in the context of Nr-of-votes and Person as the path expression $p_3 \circ {p_1}^{\leftarrow}$.

6.4.2 Assigning a path expression to a name

The partial function

$$\mathsf{Path} : \mathcal{O} \times \mathcal{O} \times \mathcal{N} \mapsto \boldsymbol{\mathcal{PE}}$$

assigns, in a given context (two object types), a path expression to a name. The name n then can be used as a denotation for a path connecting two object types. In this case, name n is said to be a *defined name*. The definition of the Path function is spread across this section. At this point, it can be stated that this function contains at least:

1. Names of object types. The name $\mathsf{ONm}(x)$ of object type x represents path expression x:

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

2. Identification names. If p is a predicator having an identification name, then this name, $\mathsf{PNm}(p)$, describes a path from the base of p to its corresponding fact type:

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

3. Connector names. If predicator p of binary fact type $f = \{p, q\}$ has a connector name, then this name is interpreted as in RIDL:

$$\mathsf{Path}(\mathsf{Base}(p), \mathsf{Base}(q), \mathsf{RNm}(p)) = p \circ q^{\leftarrow}$$

provided f is not a homogeneous fact type with ambiguous connector names (i.e. $\mathsf{Base}(p) = \mathsf{Base}(q) \wedge \mathsf{RNm}(p) = \mathsf{RNm}(q)$), in which case:

$$\mathsf{Path}(\mathsf{Base}(p), \mathsf{Base}(q), \mathsf{RNm}(p)) = p \circ q^{\leftarrow} \cup q \circ p^{\leftarrow}$$

6.4.3 Overview of information descriptors

As stated before, information descriptors have a very liberal syntax. 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 detect such flaws in information descriptors. In this book, however, we will pay only marginal attention to static semantic checks.

The syntax of information decriptors is based on a number of syntactical categories. In this section, the category *Information Descriptor* is introduced. The underlying elementary syntactical categories are: *Var* for variables and $\mathcal{N}$ for names. The syntactical category *Var* is the disjoint union of the syntactical categories *IVar* and *RVar*, which are explained later. The naming conventions for instances of these syntactical categories are as follows: P, P', P_1, P_2, O, Q are elements of the syntactical category *Information Descriptor*, v is an element of the syntactical category *IVar*, w an element of the syntactical category *RVar*, and n an element of the syntactical category $\mathcal{N}$.

The semantics of the syntactical category *Information Descriptor* is given by the valuation function

$$\mathbb{D} : \mathit{Information\ Descriptor} \times \mathsf{ENV} \rightarrow \boldsymbol{\mathcal{PE}}$$

that maps information descriptors on path expressions. This valuation function is defined inductively on the structure of information descriptors, a recurrence rule is associated to each syntactical construct of the syntactical category *Information Descriptor*. An information descriptor has to be evaluated in the context of an environment. This environment contains the current values of the variables, and is a partial function from *Var* to *VarRange*. The set *VarRange* is defined as

$$\Omega_{\boldsymbol{\mathcal{PE}}} \cup \mathit{Constant\ Denotation}$$

Variables from *IVar* can only have values from $\Omega_{\boldsymbol{\mathcal{PE}}}$ and variables from *RVar* only from *Constant Denotation*. This has been formalised in the definition of assignments in [35]. ENV is the set of all possible environments, therefore, $e \in \mathsf{ENV}$ implies $e : \mathit{Var} \mapsto \mathit{VarRange}$.

6.4.4 Atomic information descriptors

The basic building blocks for information descriptors are the defined names of $\mathcal{N}$, as introduced in the previous section. The meaning of a name is obtained as the sum of all possible interpretations as recorded by the Path function. Variables are another elementary construct for information descriptors, as they are used to store intermediate results. The meaning of a variable (from *IVar*) is its current value in the environment preceded by the freeze operator † as its meaning should be a path expression. Variables from *RVar* can be interpreted as information descriptors if their value in the enviroment is a constant. Formally, the meaning of the elementary constructs is:

$$\begin{aligned}
\mathbb{D}[\![n]\!](e) &= \bigcup_{\mathsf{Path}\,(x,y,n)\downarrow} \mathsf{Path}\,(x,y,n) \\
\mathbb{D}[\![v]\!](e) &= \begin{cases} \dagger\, e(v) & \text{if } e(v)\downarrow \\ \varnothing_{\boldsymbol{\mathcal{PE}}} & \text{otherwise} \end{cases} \\
\mathbb{D}[\![w]\!](e) &= \begin{cases} e(w) & \text{if } e(w)\downarrow \text{ and a constant} \\ \varnothing_{\boldsymbol{\mathcal{PE}}} & \text{otherwise} \end{cases}
\end{aligned}$$

Some examples of atomic information descriptors are names for object types (e.g. Year), and connector names (e.g. winning). As a more complex example, consider the information descriptor born-in. Assuming that in schema 6.1, for each predicator the identification name is identical to the connector name (if present), the meaning of this information descriptor is:

$$\mathbb{D}[\![\mathsf{born\text{-}in}]\!](e) = \mathsf{Birthyear.born\text{-}in}$$

$$\cup\ \mathsf{Birthyear.born\text{-}in} \circ \mathsf{being\text{-}birthyear\text{-}of}^{\leftarrow}$$
$$\cup\ \mathsf{Birthstate.born\text{-}in}$$
$$\cup\ \mathsf{Birthstate.born\text{-}in} \circ \mathsf{being\text{-}birthstate\text{-}of}^{\leftarrow}$$

6.4.5 Concatenation of information descriptors

Atomic information descriptors as such are rather limited. The atomic information descriptor born-in, for instance, has a very general meaning. More useful information descriptors can be constructed by concatenation:

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

A crucial effect of concatenation is that it significantly reduces the number of possible interpretations of the names involved. Both information descriptors P_1 and P_2 may have a large number of interpretations, if they are used in the context of each other, many of these interpretations do not apply anymore. An extreme example would be that of both information descriptors having no meaning in each other's context due to the fact that there is no connection between the object types involved. The information descriptor born-in Hobby serves as an example:

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

Note that it can be statically decided whether a connection exists between two information descriptors, which are both defined names. This fact is captured by the *first filter property*:

Lemma 6.2 Suppose n_1 and n_2 are names, then:

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

where $\mathsf{cond}(x, y, z_1, z_2) = z_1 \sim z_2 \wedge \mathsf{Path}(x, z_1, n_1)\!\downarrow \wedge \mathsf{Path}(z_2, y, n_2)\!\downarrow$.

Proof:

Suppose $z_1 \not\sim z_2$, then in each population Pop of information structure $\mathcal{I}$, z_1 and z_2 have no values in common as a consequence of the Strong Typing Rule, so $\mathsf{Pop}(z_1) \cap \mathsf{Pop}(z_2) = \varnothing$. Consequently, there is no contribution from $\mathsf{Path}(x, z_1, n_1) \circ \mathsf{Path}(z_2, y, n_2)$ to the result of $n_1\ n_2$, for any x and y. □

As another example of concatenation, consider the information descriptor born-in State, which is composed of two atomic information descriptors.

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

Sometimes, an information descriptor is extended to improve readability only. The information descriptor born-in State for example, has the same meaning as the information descriptor President born-in State. Despite their semantic equivalence, one may find this latter information descriptor more readable.

Information descriptors corresponding to object types can also be concatenated, e.g.:

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

This demonstrates that sometimes a shorter formulation is preferable. The next lemma, the *second filter property*, identifies situations where one can reduce an information descriptor without losing information.

Lemma 6.3 Suppose n_1 and n_2 are names of object types X_1 and X_2 respectively, then:

$$X_1 \,\mathsf{Spec}\, X_2 \vee X_2 \,\mathsf{Gen}\, X_1 \Rightarrow \mathbb{D}[\![n_1\ n_2]\!]\,(e) = \mathbb{D}[\![n_1]\!]\,(e)$$

Proof:

Suppose that n_1 and n_2 are names and X_1 and X_2 are object types, such that $X_1 = \mathsf{Object}(n_1)$, $X_2 = \mathsf{Object}(n_2)$, and $X_1 \,\mathsf{Spec}\, X_2 \vee X_2 \,\mathsf{Gen}\, X_1$. Then, in each population Pop of information structure $\mathcal{I}$: $\mathsf{Pop}(X_1) \subseteq \mathsf{Pop}(X_2)$. As a result, $\mathbb{D}[\![n_1\ n_2]\!]\,(e) = \mathbb{D}[\![n_1]\!]\,(e)$ in each environment e. □

The identification name of a predicator can be employed in the formulation of information descriptors describing paths through objectified fact types. To allow for a natural formulation of the information descriptor one usually has to choose the identification name different from the connector name. As an example, suppose that $\mathsf{PNm}(p_4) = \mathsf{in}$, instead of $\mathsf{PNm}(p_4) = \mathsf{RNm}(p_4) = \mathsf{having\text{-}as\text{-}spouse}$, in figure 6.1. The information descriptor President in Marriage resulting-in Nr-of-children then evaluates to a path expression connecting presidents with their number of children:

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

6.4.6 Keywords

The use of special keywords is illustrated in figure 6.2.

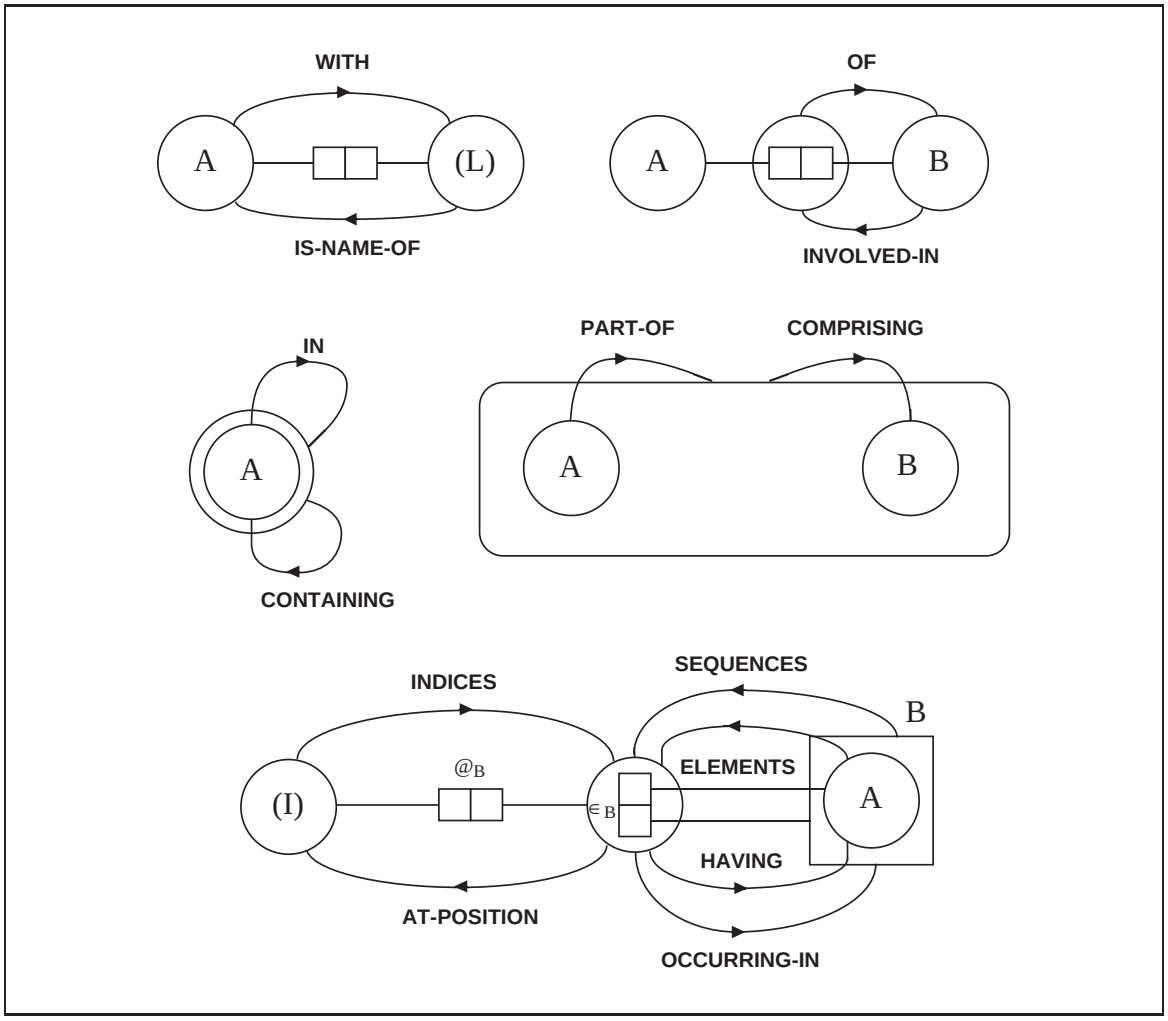

Figure 6.2: Illustration of the meaning of keywords

Appendix A

Mathematical Notation

In this appendix the mathematical notation used in this thesis, as far as it is non-standard, is explained briefly.

A.1 Functions

A partial function f from A to B is defined by $f : A \mapsto B$. Formally, $f \subseteq A \times B$ such that $\langle a, b\rangle \in f \wedge \langle a, c\rangle \in f \Rightarrow b = c$. This property allows one to write $f(a) = b$ instead of $\langle a, b\rangle \in f$. The following abbreviations are used:

$$\begin{array}{rcl} f(a)\downarrow & = & \exists_{b \in B}\,[f(a) = b] \\ f(a)\uparrow & = & \neg f(a)\downarrow \\ \mathsf{dom}(f) & = & \left\{ a \in A \;\middle|\; f(a)\downarrow \right\} \\ \mathsf{ran}(f) & = & \left\{ b \in B \;\middle|\; \exists_{a \in A}\,[f(a) = b] \right\} \end{array}$$

If f is a partial function, then $f \oplus \{a : b\}$ is also a partial function defined by:

$$f \oplus \{a : b\} = \{\langle a, b\rangle\} \cup \left\{ \langle x, y\rangle \in f \;\middle|\; x \neq a \right\}$$

The function $f \oplus \{a : b\}$ therefore behaves the same as function f except that its value in a is b.

A total function f from A to B is defined by $f : A \to B$. Formally, f is a partial function from A to B, i.e. $f : A \mapsto B$, such that f is defined for each element of A, $\mathsf{dom}(f) = A$.

To avoid having to use many parenthesis as a result of repetitive function applications, the function composition notation may be applied. The function $f \cdot g$ is defined by:

$$f \cdot g(x) = f(g(x))$$

Naturally, it is required that $\mathsf{ran}(g) \subseteq \mathsf{dom}(f)$.

The function $\mathsf{Restrict}\, f A'$ is the function f restricted to a subdomain $A' \subseteq \mathsf{dom}(f)$. This function is defined by:

$$\mathsf{Restrict}\, f A' = \left\{ \langle a, b\rangle \in f \;\middle|\; a \in A' \right\}$$

To avoid the frequent use of tuple brackets ($\langle\rangle$), a shorthand for denoting functions is used. For example, the function f, defined by $f = \{(p, a), (q, b)\}$, is denoted as $\{p : a, q : b\}$.

A.2 Sets and tuples

The power set of a set A, i.e. the set of all subsets of A, is denoted as $\wp(A)$. The set of all *finite* subsets of A is given by $\wp^{\text{fin}}(A)$. The set of all finite tuples with elements from A is denoted by A^*, while the set of all *nonempty* finite tuples with elements from A is denoted as A^+.

The i-th element of a tuple $\langle a_1, \ldots, a_i, \ldots, a_n \rangle$, i.e. a_i, can be found by projection:

$$\langle a_1, \ldots, a_i, \ldots, a_n \rangle_{<i>} = a_i$$

The length of a tuple, i.e. its number of elements, can be found as follows:

$$|\langle a_1, \ldots, a_i, \ldots, a_n \rangle| = n$$

The head of a nonempty tuple $\langle a_1, \ldots, a_i, \ldots, a_n \rangle$, $\mathsf{head}(\langle a_1, \ldots, a_i, \ldots, a_n \rangle)$, is its first element, i.e. a_1. The tail of a nonempty tuple $\langle a_1, \ldots, a_i, \ldots, a_n \rangle$, $\mathsf{tail}(\langle a_1, \ldots, a_i, \ldots, a_n \rangle)$, is the tuple without its head, i.e. $\langle a_2, \ldots, a_i, \ldots, a_n \rangle$. A tuple can be extended with a new last element by the operator $*$:

$$\langle a_1, \ldots, a_i, \ldots, a_{n-1} \rangle * \langle a_n \rangle = \langle a_1, \ldots, a_i, \ldots, a_{n-1}, a_n \rangle$$

A tuple can be converted to a set by the function set:

$$\mathsf{set}\, \langle a_1, \ldots, a_i, \ldots, a_n \rangle = \{a_1, \ldots, a_i, \ldots, a_n\}$$

The set B^A denotes the set of all (total) functions from A to B, i.e.:

$$B^A = \left\{ f \in \wp(A \times B) \;\middle|\; f : A \to B \right\}$$

A is a proper subset of B, iff A is different from B and A is a subset of B:

$$A \subset B = A \subseteq B \wedge A \neq B$$

The difference between sets A and B, $A \backslash B$, is a set containing those elements from A that do not occur in B:

$$A \backslash B = \left\{ a \in A \;\middle|\; a \notin B \right\}$$

Appendix B

Graphical Notation

This appendix contains an overview of the graphical conventions of the various techniques defined in this thesis.

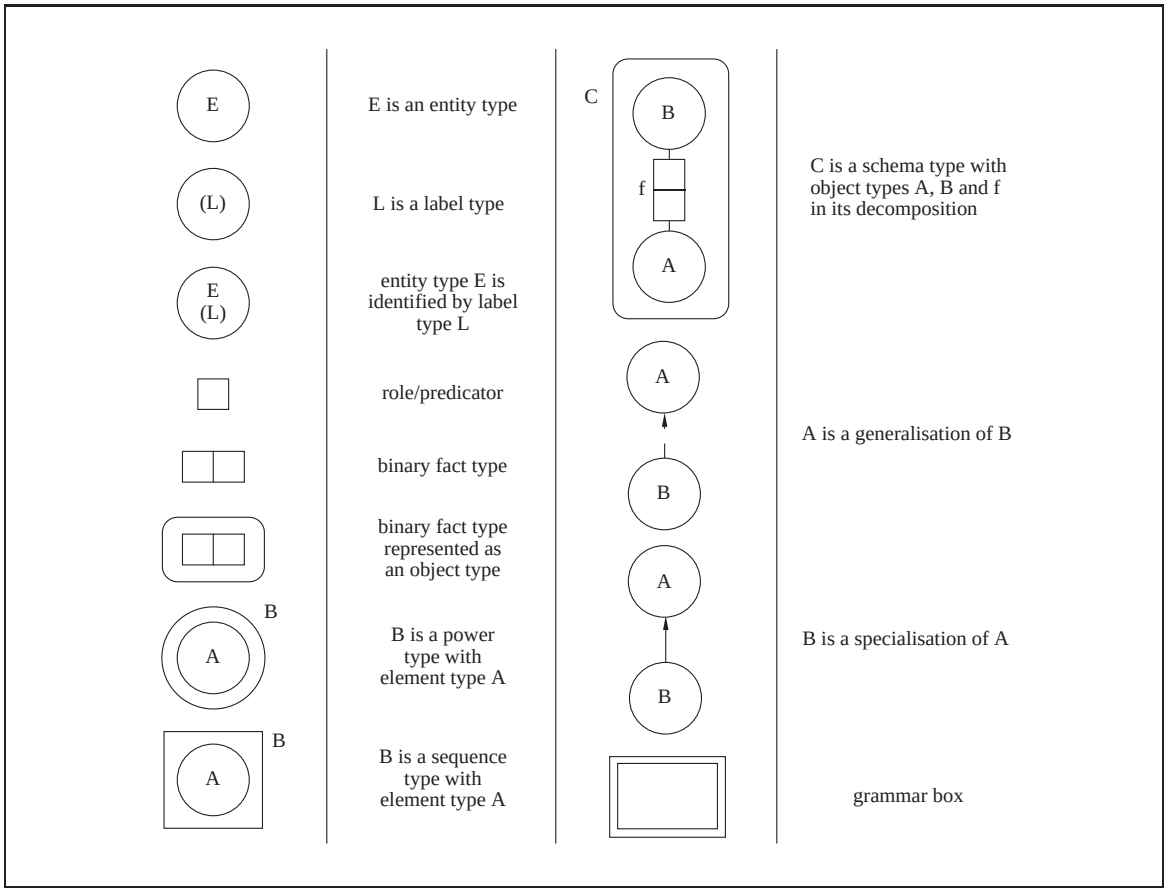

Figure B.1: Construction mechanisms of PSM

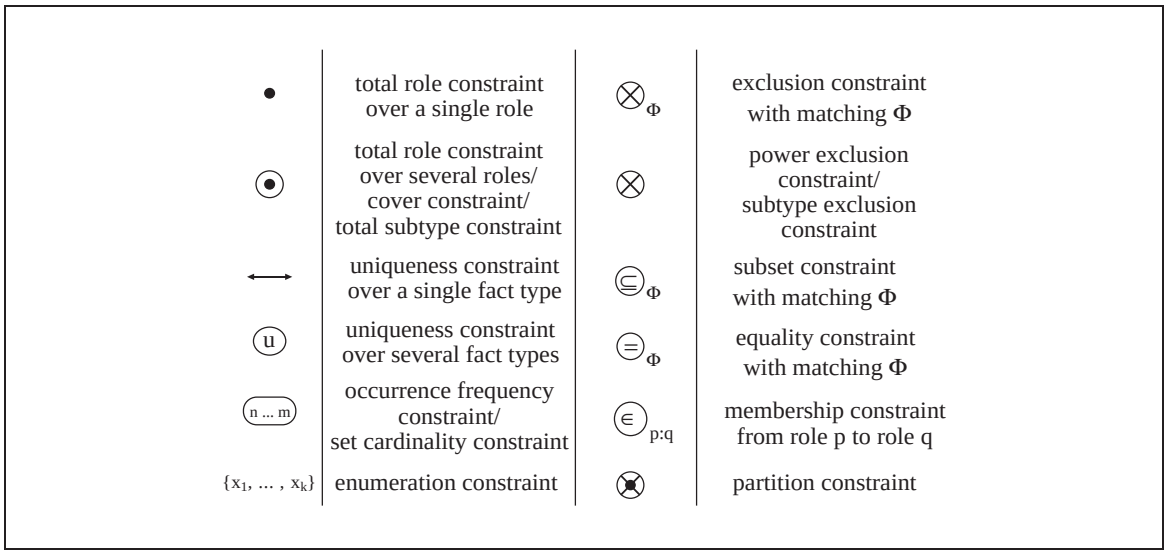

Figure B.2: Graphical constraints in PSM

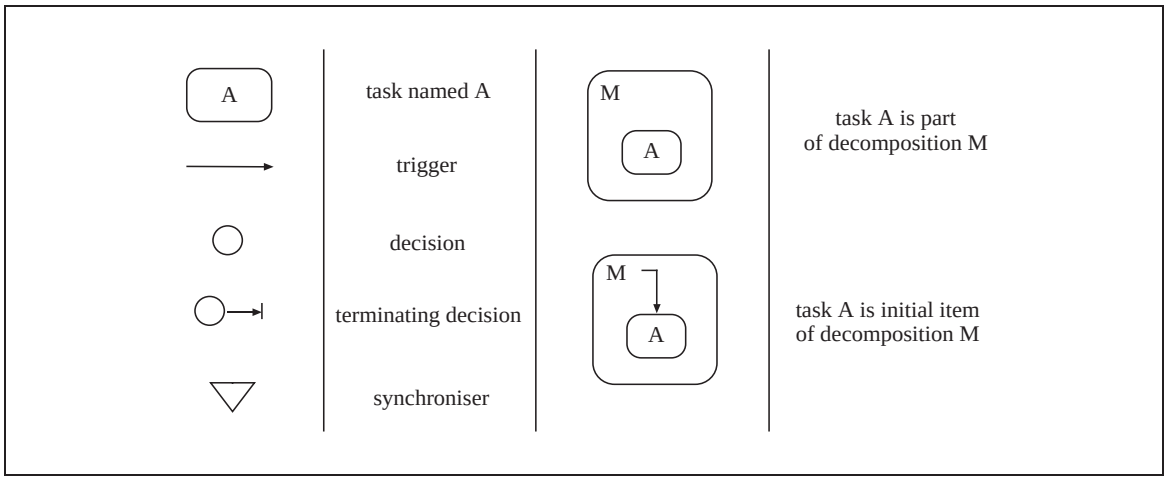

Figure B.3: Task structure concepts

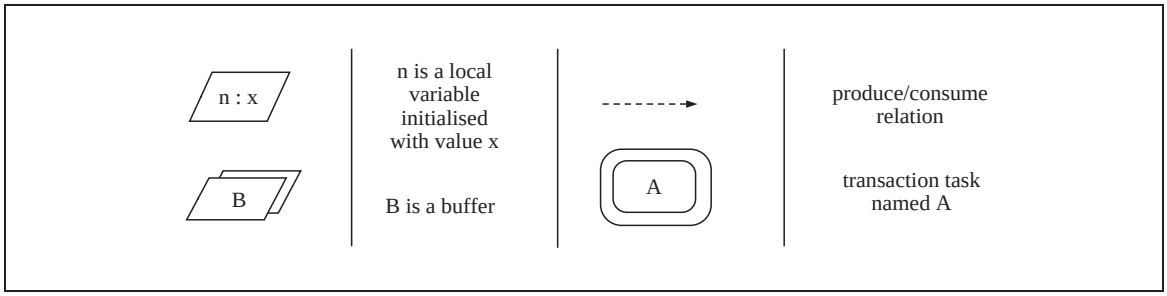

Figure B.4: Transaction concepts

Appendix C

Further formalization

C.1 Properties of specialisation

$\forall_{a \in \mathcal{O}} \exists_{b \in \mathcal{O}} [\sqcap(a, b)]$

Proof:

Due to the fact that $\mathcal{O}$ is finite and specialisation networks are acyclic, we can define a function $\mathsf{depth} : \mathcal{O} \to \mathbb{N}$ as follows:

$$\mathsf{depth}(x) = \begin{cases} 0 & \text{if } \neg\, \mathsf{spec}(x) \\ 1 + \min\limits_{x\, \mathsf{Spec}\, y} \mathsf{depth}(y) & \text{otherwise} \end{cases}$$

The proof now follows by induction on the depth of an object type x. If $\mathsf{depth}(x) = 0$, then $\neg\, \mathsf{spec}(x)$ and therefore $\sqcap(x, x)$ since $x\, \mathsf{Spec}^*\, x$. If $\mathsf{depth}(x) = n + 1$, then an object type y exists, such that $x\, \mathsf{Spec}\, y$ and $\mathsf{depth}(y) = n$. Applying the induction hypothesis then yields that an object type z exists, such that $\sqcap(y, z)$. Therefore, $\neg\, \mathsf{spec}(z)$ and $y\, \mathsf{Spec}^*\, z$. This implies that $x\, \mathsf{Spec}^*\, z$ and consequently, $\sqcap(x, z)$. □

Idempotency of $\sqcap$: $\sqcap(\sqcap(a)) = \sqcap(a)$

Proof:

Assume $\sqcap(a) \neq \sqcap(\sqcap(a))$. From $\sqcap(a)\, \mathsf{Spec}^*\, \sqcap(\sqcap(a))$ it then follows that a $b \in \mathcal{O}$ exists such that $\sqcap(a)\, \mathsf{Spec}\, b$ and $b\, \mathsf{Spec}^*\, \sqcap(\sqcap(a))$; $\sqcap(a)\, \mathsf{Spec}\, b$, however, implies that $\mathsf{spec}(\sqcap(a))$, which contradicts the definition of $\sqcap$. □

A subtype has the same pater familias as its supertype(s):

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

Proof:

Assume $a\, \mathsf{Spec}\, b$. From $b\, \mathsf{Spec}^*\, \sqcap(b)$ it then follows that $a\, \mathsf{Spec}^*\, \sqcap(b)$. Also, $a\, \mathsf{Spec}^*\, \sqcap(a)$. From the requirement of having a unique pater familias, it then follows that $\sqcap(a) = \sqcap(b)$. □

C.2 Population rules

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

[P1] $\quad x \in \mathcal{G} \wedge y \in \mathsf{Pop}(x) \Rightarrow y \in \wp(\mathsf{Pop}(\mathsf{Elt}(x))) \backslash \{\varnothing\}$

The implicit fact type $\in_x$ that is provided for each power type x, relates sets in the population of power type x, to their elements in the population of element type $\mathsf{Elt}(x)$. This is described in the *Power Base Rule*:

[P2] $\quad x \in \mathcal{G} \Rightarrow \mathsf{Pop}(\in_x) = \left\{ \left\{ \in_x^p : u, \in_x^e : v \right\} \;\middle|\; u \in \mathsf{Pop}(x) \wedge v \in u \right\}$

The Power Base Rule is a *derivation rule* for the population of fact type $\in_x$. Note that, in the formulation of this rule, it is not necessary to state that $v \in \mathsf{Pop}(\mathsf{Elt}(x))$, since this follows from the Conformity Rule.

The population of a sequence type consists of nonempty sequences of instances of the corresponding element type. This is called the *Sequence Type Rule*:

[P3] $x \in \mathcal{S} \wedge y \in \mathsf{Pop}(x) \Rightarrow y \in \mathsf{Pop}(\mathsf{Elt}(x))^+$

The population of the index type I is the set of possible indices. This set can be derived from the sequences occurring in the populations of sequence types (*Active Index Rule*).

[P4] $\mathsf{Pop}(\mathrm{I}) = \left\{ n \in \mathbb{N} \setminus \{0\} \;\middle|\; \exists_{s \in \mathcal{S}} \exists_{u \in \mathsf{Pop}(s)} \left[|u| \geq n \right] \right\}$

The populations of the implicit fact types $\in_x$ and $@_x$, provided for each sequence type x, are given by the *Sequence Decomposition Rules*.

[P5] $x \in \mathcal{S} \Rightarrow \mathsf{Pop}(\in_x) = \left\{ \{ \in^s_x : u, \in^e_x : v \} \;\middle|\; u \in \mathsf{Pop}(x) \wedge \exists_{i \in \mathsf{Pop}(\mathrm{I})} \left[u_{<i>} = v \right] \right\}$

[P6] $x \in \mathcal{S} \Rightarrow \mathsf{Pop}(@_x) = \left\{ \left\{ @^s_x : u, @^i_x : v \right\} \;\middle|\; u \in \mathsf{Pop}(\in_x) \wedge u(\in^s_x)_{<v>} = u(\in^e_x) \right\}$

These rules act as derivation rules for $\in_x$ and $@_x$.

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

[P7] $x \,\mathsf{Spec}\, y \Rightarrow \mathsf{Pop}(x) \subseteq \mathsf{Pop}(y)$

This rule does not require that instances of subtypes have to fulfil the subtype defining rule associated to the involved subtype. To express this requirement, a language for formulating subtype defining rules is necessary. Subtype defining rules are addressed in section 4.9. It should be noted at this point however, that subtype defining rules act as population derivation rules for subtypes.

Respecting the generalisation hierarchy is captured by the the *Generalisation Rule*:

[P8] $\mathsf{gen}(x) \Rightarrow \mathsf{Pop}(x) = \bigcup_{x \,\mathsf{Gen}\, y} \mathsf{Pop}(y)$

The *Generalisation Rule*, which clearly is a derivation rule, requires that the population of a generalised object type is the union of the populations of its specifiers.

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

[P9] $x \in \mathcal{C} \wedge y \in \mathsf{Pop}(x) \Rightarrow \mathsf{IsPop}(\mathcal{I}_x, y)$

The relations between instances occurring in the population of a schema type and instances occurring in the population of its constituing object types are recorded in the implicit fact types $\in_{c,d}$. Their population is prescribed by the *Decompositor Rule*, which is a derivation rule:

[P10] $x \prec y \Rightarrow \mathsf{Pop}(\in_{x,y}) = \left\{ \left\{ \in^c_{x,y} : u, \in^d_{x,y} : v \right\} \;\middle|\; u \in \mathsf{Pop}(x) \wedge v \in u(y) \right\}$

The following lemma states that instances occurring in an instance of a schema type, which is, as stated before, a population, are also instances of an object type part of the decomposition of that schema type.

Lemma C.1 $x \prec y \Rightarrow \forall_{u \in \mathsf{Pop}(x)} \left[u(y) \subseteq \mathsf{Pop}(y) \right]$

Proof:

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

C.3 Object types in relational expressions

$\mathsf{Objects}(r)$ is the set of object types needed to evaluate relational expression r:

$$
\begin{aligned}
\mathsf{Objects}(f) &= \left\{ \mathsf{Base}(p) \;\middle|\; p \in f \right\} \\
\mathsf{Objects}(r \cup s) &= \mathsf{Objects}(r) \cup \mathsf{Objects}(s) \\
\mathsf{Objects}(r \setminus s) &= \mathsf{Objects}(r) \\
\mathsf{Objects}(r \bowtie s) &= \mathsf{Objects}(r) \cup \mathsf{Objects}(s) \\
\mathsf{Objects}(\pi_{q_1:p_1,\ldots,q_n:p_n}(r)) &= \mathsf{Objects}(r) \\
\mathsf{Objects}(\sigma_F(r)) &= \mathsf{Objects}(r) \\
\mathsf{Objects}(\chi(r,\tau,a)) &= \mathsf{Objects}(r) \\
\mathsf{Objects}(\mu_f^p(g)) &= \mathsf{Objects}(g) \cup \mathsf{Objects}(f) \\
\mathsf{Objects}(\eta_f^p(g)) &= \mathsf{Objects}(g) \cup \mathsf{Objects}(f) \\
\mathsf{Objects}(\zeta_p(x)) &= \textbf{if } x \in \mathcal{O}\backslash\mathcal{F} \textbf{ then } x \textbf{ else } \mathsf{Objects}(x) \textbf{ fi}
\end{aligned}
$$

The operator $\mathsf{Objects}$ is used in section 4.9 for subtype defining rules.

C.4 Derivation rules for dependency of object types

$a \rhd b$ is to be interpreted as "a depends on b", i.e. object type a can only be populated if object type b can be populated:

[D1] $a \rhd b \wedge b \rhd c \vdash a \rhd c$

[D2] $a \rhd a \vdash \mathsf{error}(a)$

[D3] $a \rhd b \wedge \mathsf{error}(b) \vdash \mathsf{error}(a)$

[D4] $a\,\mathsf{Spec}\,b \vdash a \rhd b$

[D5] $a \in \mathsf{Objects}(\mathsf{SubRule}(b)) \vdash b \rhd a$

[D6] $\mathsf{gen}(a) \wedge \forall_{b,a\,\mathsf{Gen}\,b}\,[b \rhd c \vee b = c \vee b \rhd a \vee \mathsf{error}(b)] \vdash a \rhd c$

[D7] $\vdash \mathsf{Fact}(p) \rhd \mathsf{Base}(p)$

[D8] $\vdash a \rhd \mathsf{Elt}(a)$

[D9] $a \prec b \vdash a \rhd b$

These derivation rules are used in section 4.9 for subtype defining rules.

Appendix D

Further application

D.1 Chipkaarten vanuit informatiekundig perspectief

De technologische mogelijkheden van chipkaarten worden steeds verder uitgebreid [1]. Ook de organisatorische inbedding daarvan is in volle gang, hoewel met name op dit laatste punt nog enorm veel moet gebeuren. Echter, om tot een gezonde organisatorische inbedding te komen, is het belangrijk om in de betreffende organisaties een grondige informatie- en procesanalyse uit te voeren. Als dergelijke analyses zijn uitgevoerd, dient de toepassing van chipkaarten gekoppeld te worden aan de opgeleverde informatie- en procesmodellen.

In dit artikel zullen we de toepassing van chipkaarten vanuit informatiekundig perspectief bekijken. Dit is nodig om de organisatorische inbedding (beter) mogelijk te maken, zeker wanneer het om *echt* multifunctionele chipkaarten gaat. In het bijzonder is dit nodig om koppelingen met databases te kunnen maken.

Bij informatie-analyse speelt het begrip 'type' een belangrijke rol. Er zijn verschillende typen van informatie, bijvoorbeeld namen, bedragen, adressen, telefoonnummers, etcetera. Bovendien is er het onderscheid tussen typen en hun instanties. Zo kan het type 'naam' als instanties hebben 'Piet', 'Henk', 'Truus'. Interessant daarbij is de verhouding tussen het aantal typen en het aantal instanties. Er kunnen veel typen zijn met relatief weinig instanties, maar ook weinig typen met relatief veel instanties. In het extreme geval waarin elk type slechts een enkel instantie heeft, is typering niet zinvol meer.

Hoe zit het met de typering van informatie op chipkaarten? We kunnen er allerlei typen op tegenkomen, varierend van eenvoudige tot meer complexe typen. Voorbeelden van eenvoudige typen zijn codes, naam van eigenaar, naam van huisarts, telefoonnummers, en bedragen (prijzen en beursinhoud). Meer complexe typen kunnen bijvoorbeeld medische gegevens, lichamelijke conditie, kenmerken van lichaamsdelen (bijvoorbeeld vingerafdrukken), data en tijdstippen, spaarakties, zegeltjes, Air Miles, en vervoersbewijzen betreffen. Verder kunnen aan een elektronische portemonnee verschillende betalingsprotocollen worden gekoppeld, bijvoorbeeld om te betalen via Internet. De eerste indruk bij dit alles is dat typering op chipkaarten wel zin heeft, hoewel het aantal instanties per type relatief laag is (zeker vergeleken met wat in databases gebruikelijk is).

Naast typen en hun instanties is ook 'type gerelateerdheid' belangrijk. Twee typen worden gerelateerd genoemd als ze dezelfde instanties kunnen bevatten. Type gerelateerdheid ontstaat met name door specialisatie en generalisatie. Specialisatie wil zeggen dat een type wordt verdeeld in subtypen, terwijl bij generalisatie verschillende typen worden samengevoegd tot een enkel type. Type gerelateerdheid is interessant, omdat het ons in staat stelt om operaties en beperkingsregels op systematische wijze te specificeren.

Hoe zit het met de type gerelateerdheid in de informatie op chipkaarten? Namen zijn type gerelateerd. Data, tijdstippen, en bedragen ook. Alle objecten van het type 'aantal' kunnen type gerelateerd zijn, hoewel dit niet altijd zinvol is. Je zult een hoeveelheid gespaarde zegeltjes zelden willen vergelijken met het salaris of het aantal kinderen. De genoemde operaties en beperkingsregels kunnen bijvoorbeeld worden gebruikt om betalingsprotocollen te ondersteunen.

Naast bovenstaande aspecten van informatie-analyse is het nodig te analyseren wat de relevante relaties zijn tussen de typen op chipkaarten, bij welke processen in de betreffende organisaties de informatie op chipkaarten is betrokken (als input of als output), en op welke wijze deze informatie wordt gemanipuleerd. Ter afsluiting kan (enigszins pessimistisch) worden opgemerkt dat bij de invoering van chipkaarten helaas wel hetzelfde zal gebeuren als bij de invoering van veel andere vormen van Informatie Technologie: informatie- en procesanalyse vinden achteraf plaats i.p.v. vooraf.

[1] Deze sectie is gebaseerd op [6].

D.2 Miracles To Save The Realm

Faustian Bargains Or Noble Pursuits?

The software crisis[2] is very much alive almost 30 years after it was first defined. Only now, it is beyond crisis proportions and is eating into our GNP. With a problem so large and wide-spread, most organizations have not really addressed it. Large organizations have built massive legacy mountains of data and code; 150 terabytes of data and 200 million lines of code is not unusual for a North American telephone company. Over the past decade the information systems community has produced some brilliant ideas, including the following:

> client/server, business processes re-engineering, workflows, distributed object computing, World Wide Web, inheritance, polymorphism, re-use, interoperability, ontologies, legacy systems, wrappers, repositories, cooperative ISs, COTS, class libraries, agents, business objects and rules, object-oriented type systems and languages, electronic commerce.

These revolutionary ideas have been the basis for promises of orders of magnitude improvements in productivity so as to address the software crisis. There have been rare cases in which success has been achieved. In general, however, they have made almost no impact on industrial-strength problems. More widespread success may well be achieved in some years (e.g., 20 for relational DBMSs). They cannot be applied now in vanilla programming shops with vanilla staff.

[2]This section is taken from the keynote speech given by M.L. Brodie at the ER'97 conference.

Bibliography

[1] S. Abiteboul and R. Hull. IFO: A Form Sem Db Model. *TDS*, 12(4), 1987.

[2] D.E. Avison and G. Fitzgerald. *IS Dev.* Blackwell, 1988.

[3] H.P. Barendregt. *Lambda Calc*, volume 103 of *Stud in Log & F of Math.* North-Holl, 1984.

[4] J.A. Bergstra and G.R. Renardel. De plaats van form spec in de softw techn. *Inf*, 31(6), 1989.

[5] E.A. Boiten. *Views of Form Prog Dev.* PhD thesis, Univ of Nijm, 1992.

[6] P. van Bommel. Multifunct chipkaarten vanuit informatiekundig persp. *Chipcards*, (6), 1997.

[7] P. van Bommel (ed). *Inf Model for Internet Appl.* IGP, Hershey PA, USA, 2003.

[8] M.L. Brodie. Dev of data models. In *Conc Model, Persp from AI, Db's and Prog Lang.* Spr, 1984.

[9] P.D. Bruza and Th.P. van der Weide. 2-Lev Hyperm. In *DEXA*. Spr, 1990.

[10] J.A. Bubenko. IS Meth. In *IS Design Meth.* North-Holland, 1986.

[11] S. Ceri and G. Gottlob. Transl SQL Into Rel Alg. *Trans on Softw Eng*, 11(4), 1985.

[12] E.F. Codd. A Rel Model of Data for Large Shared Db. *CACM*, 13(6), 1970.

[13] B. Cohen. Justif of form meth for syst spec. *Softw Eng J*, 4(1), 1989.

[14] Conf on visual database systems. *IFIP 2.6.* 1998.

[15] P. Creasy. ENIAM. In *VLDB*, Amsterdam, 1989.

[16] J.N. Crossley et al. *What is mathematical logic?* Oxford University Press, 1972.

[17] A.M. Davis. *Softw Requirements.* Prentice-Hall, 1990.

[18] O.M.F. De Troyer et al. RIDL User Guide. Technical report, Control Data, Brussels, 1984.

[19] O.M.F. De Troyer et al. RIDL* on CRIS Case. In *Comp Ass dur IS Life Cycle*, 1988.

[20] G. Engels et al. Conc model of db using EER. *DKE*, 9(4), 1992.

[21] E.D. Falkenberg. Db and IS I. Lecture Notes, Univ of Nijm, 1986.

[22] E.D. Falkenberg. Det ER Model. 88-13, Univ of Nijm, 1988.

[23] E.D. Falkenberg and Th.P. van der Weide. Form Descr of TOP Model. 88-01, Univ of Nijm, 1988.

[24] E.D. Falkenberg et al. Underst proc struct diagr. *IS*, 16(4), 1991.

[25] L. Feng et al. Sem Netw-Based Design Meth for XML Docs. *TOIS*, 20(4), 2002.

[26] A. Finkelstein and J. Hagelstein. Form Framew for Underst IS Req Eng. In *IFIP WG 8.1*, 1989.

[27] J.P. Fry and E.H. Sibley. Evolution of DBMS's. *Computing Surveys*, 8(1), 1976.

[28] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman and Company, 1979.

[29] J.J. van Griethuysen, editor. *Conc & Term for Conc Sch & Inf Base.* ISO, 1982.

[30] J. Hagelstein et al. Intro Form Req into Industry. In *Ws on Meth based on Form Spec*, 1989.

[31] T.A. Halpin. *A logical anal of IS.* PhD thesis, University of Queensland, 1989.

[32] M. Hammer and D. McLeod. Db Descr with SDM. *TDS*, 6(3), 1981.

[33] D. Harel. On Visual Form. *CACM*, 31(5), 1988.

[34] M. Harrison. *Introduction to Form Lang Theory.* Addison-Wesley, 1978.

[35] A.H.M. ter Hofstede. *Inf Model in Data Int Dom.* PhD thesis, Univ of Nijm, 1993.

[36] A.H.M. ter Hofstede and Th.P. van der Weide. Form of techn. *IST*, 34(1), 1992.

[37] A.H.M. ter Hofstede et al. Meth and Graph Knowl. In *Conf on Softw and Knowl Eng*, 1992.

[38] A.H.M. ter Hofstede et al. Spec of Graph Conv in Meth. In *Ws on Next Gen CASE Tools*, 1992.

[39] A.H.M. ter Hofstede et al. Conc DML. *IS*, 18(7), 1993.

[40] A.H.M. ter Hofstede et al. Fact Or in Compl ORM Techn. In *Conf on ORM*, 1994.

[41] U. Hohenstein and G. Engels. Form Sem of ER-Based Query Lang. In *ER Conf*, 1990.

[42] I. van Horenbeek and J. Lewi. *Algebr spec in softw eng.* Spr, 1989.

[43] R. Hull and R. King. Sem Db Model. *Computing Surveys*, 19(3), 1987.

[44] R. Hunter et al. ODA. *Comp Comm*, 12(2), 1989.

[45] ISO. *Inf Proc - Text and Office Systems - SGML.* 1986.

[46] ISO. *Inf Proc - Text and Office Systems - ODA.* ISO8613, 1989.

[47] M.A. Jackson. *System Dev.* Prentice-Hall, 1983.

[48] C.B. Jones. *Syst Softw Dev using VDM.* Prentice-Hall, 1986.

[49] W. Kim. On Unifying Rel and OO DB Syst. volume 615 of *LNCS*, 1992.

[50] G.M. Kuper and M. Vardi. Expr Pow of Log Data Model. In *Conf on Man of Data*, 1985.

[51] A. Levy. *Basic Set Theory.* Spr, 1979.

[52] A. Lew. *Comp Sc: Math Intro.* Prentice-Hall, 1985.

[53] H.R. Lewis and C.H. Papadimitriou. *Elem of th of computat.* Prentice-Hall, 1981.

[54] D. Maier. *Th of Rel Db's.* Computer Science Press, 1988.

[55] R. Meersman. The RIDL Conc Lang. Technical report, Control Data, Brussels, 1982.

[56] B. Meyer. *Intr to the Th of Prog Lang.* Prentice-Hall, 1990.

[57] G.A. Miller. Magic Number Seven, Plus or Minus Two. *Psych Rev*, 63, 1956.

[58] G.M. Nijssen and T.A. Halpin. *Conc Sch and Rel Db Design*. Prentice-Hall, 1989.

[59] T.W. Olle et al. *IS Meth*. Addison-Wesley, 1988.

[60] H. Partsch. *Spec and Transf of Prog*. Spr, 1990.

[61] H.J. Schek and M.H. Scholl. The rel mod with rel-val attr. *IS*, 11(2), 1986.

[62] G. Scheschonk. *Auf PetriNetzen bas Meth zur Model IS*. PhD thesis, Berlin Univ of Techn, 1984.

[63] C. Sernadas et al. Proof-th Conc Model. In *IS Concepts*, 1989.

[64] J.M. Smith. SGML and related standards. *Comp Comm*, 12(2), 1989.

[65] H.G. Sol. Kennis en erv rond het ontw van IS. *Inf*, 27(3), 1985.

[66] P.L. van der Spiegel et al. Transact Model for Hypert. In *DEXA*, 1991.

[67] J.M. Spivey. *Understanding Z*. Cambridge Univ Press, 1988.

[68] J.E. Stoy. *Denotational Sem*. MIT Press, 1977.

[69] B. Stroustrup. *The C++ Prog Lang*. Addison-Wesley, 1991.

[70] T.H. Tse and L. Pong. Tow a Form Def for DeMarco DFD. *Comp J*, 32(1), 1989.

[71] T.H. Tse and L. Pong. Exam of Req Spec Lang. *Comp J*, 34(2), 1991.

[72] G. Verheijen and J. van Bekkum. NIAM: an Inf Anal Meth. In *IS Des Meth*. 1982.

[73] T.F. Verhoef. Struct Yourdon's Struct Anal. In *Ws on Next Gen of CASE Tools*, 1991.

[74] A.A. Verrijn-Stuart and G.J. Ramackers. Model Int of IP Tools. volume 593 of *LNCS*, 1992.

[75] R.J. Welke. The CASE Repository. In *INTEC Symp Syst Anal and Des*, 1988.

[76] G.M. Wijers. *Model Support in IS Dev*. PhD thesis, Delft Univ, 1991.

[77] S.E. Willner et al. COMRADE data man syst. In *AFIPS Comp Conf*, 1973.

[78] J.J.V.R. Wintraecken. *The NIAM Inf Anal Meth*. Kluwer, 1990.

[79] Workshop on verification, validation, and integrity. *In conjunction with DEXA'98*. 1998.

[80] E. Yourdon. *Modern Structured Anal*. Prentice-Hall, 1989.