Logics for OO information systems: a semantic study of object orientation from a categorial substructural perspective

de Haas, E.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

A generalized language for object oriented information systems

"Again we face most basic questions like what is the right logic and even what are the right structures"

Yuri Gurevich ([Gurevich88])

The large popularity and numerous occurrences of modeling and database languages using graphical syntax and object notions suggest that it underlies an important intuition on how to model parts of the (real) world. We will present a language that bears these notions. In [Adriaans92] Pieter Adriaans proposed graphical structures called 'categorial graphs' that in a very general way describe most of the structures used when modeling for analysis and design.\(^1\) We will, in this thesis, take the notion of 'categorial graph' as a starting point for our theory on modeling with object oriented information systems. We will 'mathematize' (i.e., turn into mathematics) the notion of 'categorial graph' to obtain a model for object oriented information systems.

In this chapter we present a language, called categorial graph language, that combines both graphical and textual phrases and is tailored to define information systems. The categorial graph language is modeled after the real life practical languages of modeling and design, and bears the features of object orientation we described in the previous chapters. We also present a language of object graphs that enables us to write down particular models of object oriented information systems that satisfy the categorial description in the categorial graph language.

In the framework of categorial graphs we will make a distinction between the way in which we write down matters of signature, and matters of constraints. The matters of signature are denoted graphically, while for the constraints we assume

\(^{1}\)There also exists a modeling tool for these graphs (see [Adriaans92]).
the presence of a constraint language in which we write down the restrictions. From a type-theoretic viewpoint this distinction may seem artificial, but in the eyes of an information system designer, it is the way it is traditionally done. Our aim is to stay as close as possible to the practitioners intuition as possible.

In the next section we will start with an example to illustrate the language and notions for object oriented information systems in an informal presentation of a case. This example will emphasize important features for object oriented information system languages. We will use this example in the whole chapter to solidify the abstract definitions of the syntactical theory.

In the subsequent section we will display the syntax of a class of languages, called the categorial graph languages. These languages contain a graphical and a textual part. The graphical part is built from so called edge graphs. The edge graphs form the core of the class of languages. The textual part is a language that is used to write down constraints. Next to presenting the syntactical building blocks of the language we will discuss syntactic matters like writing down schemas and (typed) instances for information systems. Furthermore we will tackle a common problem of writing down large schemas for information systems using a good way of imploding and exploding the syntactic structure (the edge graphs), without losing means to interpret the syntax in a proper way.

### 3.1 A Case for Object Oriented Information Systems

In the example below we will speak about objects with a complex signature: the have attributes and aggregations. We write down both the schema and an instance of an information system. Furthermore we extend the schema by adding constraints to the signature.

#### 3.1.1 Example. Consider a working roster of an aviation pilot. As a simple example, his roster will look like a sequence of flights, time-offs, and possibly some obligatory training courses. His roster then could be modeled like an aggregation of objects that model flights, time-offs and courses. These flight, time-off and course objects then, would typically have attributes (adjacents) like start-date-and-time, end-date-and-time. A flight also would have as an attribute the departure airport, and the arrival airport. In a picture:
3.1. A Case for Object Oriented Information Systems

The picture also contains cardinality constraints. The '*' denotes the adjacency of zero or more tasks to the pilot. It would be convenient to have the ability to denote that we have an object of type roster that is an arbitrary long aggregation of edges of some other type (i.e. task). The roster is an aggregated object, and can be addressed (as a whole) just like any other object.

- A roster ISA aggregation of zero or more tasks

Furthermore we add a constraint that touches inheritance. For a clear division between object-structure and object-constraints we do not denote it graphically like in most OO design languages (this will become clear later on in this chapter).

- A pilot ISA person

An instance of the above schema could look as follows:
For the safety of the passengers, the rosters of pilots should respect certain rules. In this example we state two rules:

- A flying duty should be rostered to a pilot and not to some other type of person
- A flying duty should be preceded by a rest of at least 11 hours

### 3.2 The Syntactic Theory

The language of categorial graphs combines both graphical and textual phrases. For a proper presentation of this language we need a nice syntactic theory like the theory of formal languages (in a sense we are leaving the good developed theory of words as described in [Davis58], [HopcroftUllman79], [LewisPapadimitriou81]). Note that it is not problematic to consider graphical structures in the syntax, as long as there exists a well defined mathematical formulation for these syntactic structures. One should provide a simple syntactic theory for the graphical structures in the same style as done for conventional textual structures. Observe that graphical structures like graphs are simple mathematical entities, just as strings, words and sentences are.

The language of categorial graphs is tailored to defining and writing down information models and database schemas. It consists of a graphical part, in which one can specify the signatorial matters, and a textual part in which one can state constraints on the specified signature. This is exactly as most information system languages, especially those in analysis and design, are structured. The most commonly known are UML, NIAM and (E)ER.

The syntax for the graphical part of the language of categorial graphs is built from edge graphs, which are presented below. The edge graphs define a signature for an information system like a theory of categorial grammars Using a sufficiently rich language, one can put (additional) constraints on these categories (types).

#### 3.2.1 Edge Graphs

When we draw pictures of the objects we are modeling, we in a way 'talk' about these objects in terms of boxes and lines and other graphical constructs. Defining the graphical syntax more formally we could say we talk about the objects in terms of edges and vertexes. In our language we want to talk about objects as if they were structured entities. We could, of course, encode the structure of an object using vertexes and edges. However, when one starts modeling some part of the world, the structure and complexity of the objects are not known in their full extent. One will not know 'a priori' which possible attributes of an object
3.2. The Syntactic Theory

to take into the model or which relations to other objects exist. In order to talk about the objects one would then need a way to address the objects without being specific right away (or to a full extent) about the structure of an object. Moreover, it would be very natural if one did not need to change the structure of all the sentences when in the process of modeling the world one needs to add parts of an object to the model. This is only possible if one has a direct way of denoting complex objects and furthermore that the interpretation of these objects are incomplete or even non-wellfounded in the sense that parts of the object can be left 'unspecified' and filled in later.

A direct way to represent a structured object can be achieved by having it denoted by a structured graphical entity which is a basic building block of the graphical syntax. Moreover, we need a special 'structured object' that is empty, so we will have ways to extend the complex structure of the objects in our syntax without having to change all of the syntax.

The structured entity we will use as a basic building block is mathematically equivalent to a hyper edge. An edge has a structure, namely it has adjacents. We will talk about structured objects using these edges. This is a powerful generalization of using graphical entities for describing objects. We will be able to denote all objects, whether simple or complex, using a mathematical structure that is formally an edge. This way, all objects, both simple and complex, can be treated uniformly, because they are all denoted by an edge. We will also introduce an 'empty edge' that marks the end of the structure of an object². This way additions in the structure of an object will only affect the denotation of the object itself, not of its context.

An edge graph is a simple generalization of an ordinary graph. Consider a collection of nodes. In between these nodes we can draw edges. Such an edge has as its source and as its target a node. Having drawn edges, we can imagine we can draw edges between edges; i.e. we can draw an edge which has an edge as its source and an edge as its target. The same way we can draw edges between a node and an edge, or between an edge and a node. Let us now assume that we can imagine a node to be an edge as well. We define node as being an edge with as its source and as its target some distinguished abstract edge, called the empty edge, denoted by 1. The structure we then obtain contains edges only. We will call such a structure an 'edge graph'.

3.2.1. EXAMPLE. Look at figure 3.1. Here we have drawn two edges a and b that both have the empty edge as their source and as their target. These edges are in a sense basic edges (or nodes) because their source and target are 'empty'. Note that in the figure we 'copied' the empty edge 1 a couple of times for drawing purposes. Edge c is a complex edge, having two other edges as its source and as

²e.g. An object will always have an empty adjacent
its target. We can also make very weird edges like edge \( d \), which has the empty edge as its source and itself as its target.

We can generalize the above concept of edge graph to a concept of hyper edge graph, in the same manner as graphs are generalized to hyper graphs. Instead of letting an edge have exactly two adjacents, its source and its target, we can allow an edge to have an arbitrary number of adjacents. Such an edge is called a hyper edge. From hyper edges we can build hyper edge graphs.

When modeling part of the real world, there are several ways to talk about the structure of an object. One can be interested solely in the fact that an object has a certain attribute or adjacent, i.e. in graphical denotation we are interested whether there is or is not a line between one object and another. For example, if we are interested whether an object of type \textit{man} has an \textit{arm} adjacent. In this case the adjacency structure can be described by a \textit{set}. An object has a set of adjacents, and membership to that set determines whether an object is adjacent to another object.

One could also be interested in \textit{counting} the adjacents of an object. For example, one can be interested in how many adjacents of a certain type an object has. In this case the adjacency structure of an object is resource conscious, and can be described by a \textit{bag} or \textit{multiset}. In this case we may draw more than one line between two objects. E.g. one can then express that an object of type \textit{man} has two \textit{arm} adjacents.

One could even more specifically be interested in the fact that the first or second adjacent of an object is of a certain kind. As another example one could want to say that a certain adjacent of an object is the the adjacent labeled by 'child' and another adjacent is labeled by 'parent'. To be able to express these things the adjacency structure needs to be ordered. This can be expressed using an \textit{ordered list} for describing the adjacency structure of an object. For example
3.2. The Syntactic Theory

one could say of an object of type man that it has an arm adjacent labeled by 'left' and an arm adjacent labeled by 'right'.

It is important to observe that if we want to have some specific abilities to talk about the objects one is modeling, this has direct consequences for the theory on it. Not only does the language need a way to denote the things one wants to say, moreover the semantic domain in which this language is interpreted needs to model the desired behavior. In other words if, we want to talk resource consciously about objects, we need a syntactic graphical entity that can denote a multiset structure, and moreover, one needs to be able to count the adjacents of an object in the semantic domain in which we interpret the syntactic constructions.

In order to be able to talk about objects in the different ways indicated above, the syntactic entity of a hyper edge comes in different flavors. Starting with the most expressive one, there are the so called 'directed' hyper edges. For these hyper edges the order of the adjacents is important. For a directed hyper edge we have a first adjacent, a second adjacent, a third adjacent and so forth. The adjacents of a directed hyper edge form an ordered list. We will also consider 'undirected' hyper edges. For these edges the order of their adjacents is irrelevant. The adjacents of an undirected hyper edge constitutes a multiset. The most abstract flavor of hyper edges we will consider are the so called 'set' hyper edges. With a set hyper edge we abstract over the multiplicity of its adjacents. The adjacency structure of such an edge can be given by an ordinary set\(^3\).

3.2.2. Definition. (universes of edges and the empty edge) We will consider three universes of edges, one for set hyper edges denoted by Edge\(_\text{set}\), one for undirected hyper edges denoted by Edge\(_\text{undirected}\) and one for directed hyper edges denoted by Edge\(_\text{directed}\). All three universes contain a designated edge called the empty edge which is respectively denoted by 1\(_\text{set}\), 1\(_\text{undirected}\) and 1\(_\text{directed}\).

On the edges of the three universes, so called adjacency functions will be defined that map these edges to their adjacents. For edges in Edge\(_\text{set}\) these adjacency functions map edges to a set of edges, for edges in Edge\(_\text{undirected}\) to a multiset of edges and for edges in Edge\(_\text{directed}\) to a list of edges.

We also introduce the empty edge for the hyper edge case. Even though it is not necessary to have the empty edge in the hyper edge case, because we have genuine empty sets, empty bags and empty lists in the definition of a hyper edge (this was not the case with the binary edges). We introduce it here because it enables us later on, in an algebraic setting, to conveniently talk about (being or having) an empty structure. To ensure that 1\(_\text{set}\), 1\(_\text{undirected}\) and 1\(_\text{directed}\) really behave like the empty edge, we will require the following:

For all adjacency functions Adj we put

\(^3\)Note that we do not demand the adjacency structure (list, multiset or set) to be finite, although in almost all practical cases it will be finite.
Chapter 3. A generalized language for object oriented information systems

- $Adj(1_{\text{set}}) = \emptyset$ (empty set)
- $Adj(1_{\text{undirected}}) = \emptyset$ (empty multiset)
- $Adj(1_{\text{directed}}) = []$ (empty list)

To enforce the neutral behavior of the empty edge (1) we will identify (in the interpretation) the adjacency-sets $\{a, b, c\} \cup \{1_{\text{set}}\}$ and $\{a, b, c\}$. In other words, $\{1_{\text{set}}\}$ will behave like the empty set. Similarly $\{1_{\text{undirected}}\}$ will behave like the empty multiset and $[1_{\text{directed}}]$ will behave like the empty list.

Note that the presentation of an edge graph relates to the second most popular presentation of conventional graphs, where a graph is given by a set of nodes $V$ and a set of edges $E$ together with two functions $\text{source} : E \to V$ and $\text{target} : E \to V$ mapping the edges respectively to their source and their target.

3.2.3. Definition. (set edge graph) A set edge graph is given by a pair $(G, Adj_{\text{set}})$ where

- $G \subseteq \text{Edge}_{\text{set}}$ is a set of objects called set hyper edges,
- $Adj_{\text{set}} : G \to \mathcal{P}(G)$ is an adjacency function mapping the edges of $G$ to their adjacents (which form a set).

We will usually identify a set edge graph with its set of edges, and assume the adjacency function exists and is called $Adj_{\text{set}}$ or even plainly $Adj$, provided this does not lead to confusion.

The definitions for undirected edge graph and directed edge graph are similar to the definition of the set edge graphs. For the sake of completeness we will give these definitions anyway.

3.2.4. Definition. (undirected edge graph) An undirected edge graph is given by a pair $(G, Adj_{\text{undirected}})$ where

- $G \subseteq \text{Edge}_{\text{undirected}}$ is a set of objects called undirected hyper edges,
- $Adj_{\text{undirected}} : G \to \mathcal{P}(G)$ is an adjacency function mapping the edges of $G$ to their adjacents (which form a multiset).

---

4 Admittedly these identifications look a little peculiar in this set-theoretic presentation. Note however that in algebraic formalisms we have no problems at all with these kind of neutral elements.

5 The most popular presentation, of course, consists of two sets $V$ and $E$ where $E \subseteq V \times V$.

6 The notation $\mathcal{P}(G)$ denotes the set of all possible multisets constructed with elements from $G$, similar to the powerset set construction for ordinary sets.
3.2. The Syntactic Theory

We will usually identify a set edge graph with its set of edges, and assume the adjacency function exists and is called $Adj_{\text{undirected}}$ or even plainly $Adj$, provided this does not lead to confusion.

3.2.5. Definition. (Directed edge graph) A set edge graph is given by a pair $(G, Adj_{\text{directed}})$ where

- $G \subseteq \text{Edge}_{\text{directed}}$ is a set of objects called directed hyper edges,

- $Adj_{\text{directed}} : G \rightarrow \text{list}(G)$ is an adjacency function mapping the edges of $G$ to their adjacents (which form a list)\(^7\).

We will usually identify a set edge graph with its set of edges, and assume the adjacency function exists and is called $Adj_{\text{directed}}$ or even plainly $Adj$, if this does not lead to confusion.

3.2.6. Example. Below we have drawn 3 edge graphs, one of each flavor. The first one is a set edge graph, the second an undirected edge graph, and the third a directed edge graph. We have denoted the graphs graphically and have given their mathematical description.

\(^7\)The notation list$(G)$ denotes the set of all possible lists constructed with elements from $G$, similar to the powerset set construction for ordinary sets.
Note that in theory it is possible that an edge has itself as its source or its target. It is even possible that an edge has itself as the target of the source of its target. We will call this kind of edge 'cyclic edges', and graphs containing such edges 'cyclic edge graphs'.

We can also imagine that it is possible that a given edge has the property that one can infinitely many times 'descent' to its adjacents (source or target). Note that, among others, a cyclic edge has this property. If an edge has this property, we will call it an unfounded edge. On the other hand we will say that an edge is 'founded' if there is no infinite chain in its adjacency structure. This means that at a certain point in descending along the adjacencies, one should encounter an edge that has an empty adjacency structure. For this purpose we
introduced the *empty edge*, denoted by \( 1 \), which behaves similar to the empty word in standard formal language theory. The empty edge enables us to denote the fact that an edge has no source or target, by saying that the source or target is the empty edge. Of course the empty edge itself also has no source or target, meaning the same as saying that it has only itself as its source and target\(^8\). We will call an edge graph that contains, apart from the empty edge, only founded edges a *founded edge graph*.

3.2.7. **Definition.** (founded edge graphs) A set edge graph \( G \) is called *founded* if

1. \( 1 \in G \) (\( G \) contains the empty edge) \(^9\),
2. all edges are *founded on one* (FOO) where

\[
\text{FOO}(a) \Leftrightarrow a = 1 \text{ or } (\text{Adj}(a) = A \text{ and } \forall b \in A[\text{FOO}(b)])
\]
3. acyclism\(^{10}\)
4. nothing else but implied by 1,2,3 and 4.

The definitions for founded undirected and founded directed edge graphs are very similar. We leave these definitions as an exercise for the reader\(^{11}\).

As an aside, we note that we can *encode* an edge graph with a normal (conventional) directed graph with nodes *and* binary edges. In the representation using a conventional graph, an edge graph is simply a graph in which the nodes denote the names of the 'edges' of the edge graph and the directed edges of the conventional graph point to the nodes that denote the names of the adjacent 'edges' of the edge graph.

3.2.8. **Example.** Consider the following (set) edge graph \( G \):

\[
G = \{a, b, c, 1\}
\]
\[
\text{Adj} = \{(a, \{1\}), (b, \{1\}), (c, \{a, b\}), (1, 1)\}
\]

This graph is drawn in figure 3.2 in three ways, the upper two different edge graph notations for graph \( G \), and below these two, a conventional graph \( G' \) is drawn

---

\(^8\)This infinity in the possible denotation of the empty edge motivated us to use the term *founded* instead of *wellfounded*, because the latter term in set theory surely forbids the empty edge.

\(^9\)Redundant.

\(^{10}\)implied by 2

\(^{11}\)Hint: Only one symbol in the second clause needs to be changed!
encoding the edge graph $G$. The mathematical description of the conventional graph is as follows:

$$G' = (V, E)$$

where

$$V = \{a, b, c, 1\}$$

$$E = \{(a, 1), (b, 1), (c, a), (c, b), (1, 1)\}$$

In such a representation, a founded edge graph is nothing more than a directed acyclic graph (with a sink\(^\text{12}\) if we have the empty arrow having itself as its only adjacent). We note however that the nodes and edges in the conventional graph

\(^{12}\text{the empty arrow 1 is the sink, i.e. has a loop.}
model will not correspond directly to a (semantic) concept in which we commonly reason about information systems; In particular they do not directly denote a type or object in a traditional object oriented diagram. This flaw will make it necessary to use concepts in the semantics, corresponding to these nodes and directed edges, which are not common in information systems. The nice thing about the edge graphs, is that the syntactic primitives here (complex edges) do have a natural semantic interpretation in terms of common concepts of information systems. We will take advantage of this phenomenon in the following chapters.

As an aside we want to remark that for the syntax of the categorial graph language we have no problems with different graphical representations of the same graph, because we have taken the mathematical notion of edge graph as a syntactic entity. In fact we choose the mathematical structure of an edge graph as a syntactic entity with the intention that we then will not have to worry about trivial identity; i.e. when in many other computer languages one has language expressions that very clearly denote the same thing, in the edge graph language one has actually either the same syntactic expression, or else there are important structural differences in the expressions, and these differences are not trivially amounting to the same thing. By considering the mathematical structure of the edge

\[ < a, \{(a, b_1), (a, b_2)\} > \]

we avoided the problem of identifying this syntactic structure with the structure

\[ < a, \{(a, b_2), (a, b_1)\} > \]

where the order of the symbols differs, or even

\[ ^{13}\text{Using the terminology of information systems: The complex edges are first class citizens of the system.} \]
which have different geometric properties. These denotations are all the same syntactic structure in our language, so we do not have to worry whether or not they are interpreted (semantics!) as the same object.

3.2.2 Operations

Above we defined edge graphs, the basic syntactic entities of the language(s) of categorial graphs. As in most syntactic theories there are some basic operations or constructions defined on the syntactic entities. For example the most basic and common of these operations in textual languages is the concatenating of two strings, forming a new string. In traditional programming languages even much more complex operations are common, for example in procedural programming languages there normally exists an 'if-then-else-operator' that takes a Boolean expression\(^\text{14}\) and two statements\(^\text{15}\) and returns a statement. Complexity of these formalisms can reach even to undecidable systems like the two level grammars of ALGOL 68 ([WijngaardenEtAlii76]). The point we want to make with the above examples is that a syntactic theory can be quite complex, and that the operations we will introduce below for the syntactic entities of the categorial graph language are quite modest with respect to their complexity.

We already saw one operation for edges: the forming of an edge graph (see definitions 3.2.3, 3.2.4 and 3.2.5). Given a token and a set (multiset) [list] of edge denotations, it produces an edge.

\(^{14}\)A Boolean expression is a syntactic category.

\(^{15}\)A statement is a syntactic category.
3.2. The Syntactic Theory

One important operation used in modeling the world is taking objects together. Such an operation is used to tie objects together and to look at the tied-up objects as one player in the model. Both in syntax and in semantics we will present an aggregation operation that can be used for this purpose. For example one can think of a 'Pilot' object, having 'a number of tasks' as its roster (see example 3.1.1).

3.2.9. DEFINITION. (Aggregation) Consider a universe of edges Edge. Given two edges \( a \) and \( b \) then \( a \cdot b \) denote the aggregation of \( a \) and \( b \). Given two sets of edges \( A, B \), then \( A \cdot B \) denotes all the products \( a \cdot b \) where \( a \in A \) and \( b \in B \).

Now let \( \text{Edge}^* \) denote the set of all products of edges and products of products of edges etc. etc. (strings over \( \text{Edge} \)); i.e.

\[
\text{Edge}^* = \text{Edge} \cup \text{Edge} \cdot \text{Edge}^*
\]

Now we will define all products in \( \text{Edge}^* \) being edges as follows:

- set edges: \( \text{Adj}(a \cdot b) := \text{Adj}(a) \cup \text{Adj}(b) \)
- undirected edges: \( \text{Adj}(a \cdot b) := \text{Adj}(a) \cup \text{Adj}(b) \)
- directed edges: \( \text{Adj}(a \cdot b) := \text{append}(\text{Adj}(a), \text{Adj}(b)) \)

Note that \( \cup \) is the set-union (for set edges), and \( \cup \) is multiset-union (for undirected edges), while \( \text{append} \) is list-union (for directed edges).

For the edges in \( \text{Edge}^* \) we will have to make some non-trivial identifications:

- concatenation for set edges must obey the axioms of set union;
- concatenation for undirected edges must obey the axioms of multiset union\(^{15}\);
- concatenation for directed edges must obey the axioms of list union\(^{15}\).

\[\triangle\]

3.2.10. EXAMPLE. In the running example the roster of a pilot is modeled as an aggregation of tasks. The roster is an ordered sequence of tasks:

\[
\text{TASK} \quad \bullet \quad \text{TASK} \quad \bullet \quad \text{TASK} \quad \bullet \quad \text{TASK}
\]

An other example of aggregation would be the model a finite set. Suppose you have an object 'cabin-crew' that is an attribute of a object 'flight' and models a set of stewards or stewardesses (let's say they are modeled by 'cabin-personnel' objects having attributes like name, age etc.). Then this object 'cabin crew' is an aggregation of a couple of 'cabin-personnel' objects. In a picture:
The attributes of the aggregated objects like 'roster' or 'cabin-crew' will be the attributes of the individual parts of the aggregation. For example a roster will have several arrival station attributes, each belonging to another flight. Also a cabin-crew would have several name-attributes, one for each cabin personnel in the aggregation, and similarly, several 'age' attributes. One could think of having operation on these attributes like SUM, AVERAGE and COUNT resulting in attributes of the aggregation object ('roster' or 'cabin-crew'). These well known operations are called aggregates in relational database languages like SQL.

To avoid confusion we note that in this section we talk about static operations, in the sense that we talk about operations on the structures that we will interpret in a static semantic model. Objects may have both attributes and abilities. Although abilities have a dynamic content, dynamics itself is not covered in this thesis. The model represents an information system instance, without a past or a future. In the paper [Haas01] we embryonically discuss dynamics of information systems based on the language presented here. There we look at models that have, in a formal sense, a past and a future. The subject matter of dynamic operation is orthogonal on the matter of static operations as discussed above.

### 3.2.3 Types, Objects and Constraints

We will use edge graphs to denote a signature that specifies a type\(^\text{16}\). For example we can denote with an edge graph that an object of the type MARRIAGE involves

\(^{16}\)We will give a precise definition of the notion of type when we present the semantics of our language. Here it will remain a 'vague', but well known semantic concept that will enable us to talk about the language without having to stay totally abstract
two objects of the type PERSON and one object of the type DATE; i.e.

```
PERSON
    - MARRIAGE
    - PERSON

DATE
```

In object oriented modeling and database languages we most often talk about types of objects; i.e. the most commonly used modeling documents are database class diagrams. However we also need to talk about instances. For example the modeling language UML has a syntax for describing instances which are used in several diagramming techniques such as sequence diagrams and collaboration diagrams ([FowlerScott00] note that these diagrams also cover operational aspects that are not covered by the syntax described here). The language to talk about instances is very strong related to the language that talks about objects. Even more strictly, one can not talk meaningfully about instances if one does not have a notion of the types of these instances, because we need to know what kind of objects we are talking about (described by the types) before we can interpret the denoted instance.

For example, an object or instance of the type MARRIAGE should typically be of the same 'structure' as the edge for the type MARRIAGE itself; i.e. in mathematical terms there should exist a homomorphism between the object and the edge denoting the MARRIAGE type. We can fairly write down the structure of such an object using, again, an edge graph\textsuperscript{17}; i.e.

```
AL
  - marriage act no. 97634593
  - Peg

aug. 21 1967
```

The homomorphism between the edge graph of objects (from now onwards called \textit{object graph}) and the edge graph denoting types (from now onwards called \textit{type graph}) will be called a \textit{typing functor}; i.e. it assigns a type in the type graph to the edges in the object graph. In the above example the typing homomorphism assigns the upper PERSON edge to the AL edge, the other PERSON edge to the PEG edge, the MARRIAGE edge to an actual marriage act, and finally the DATE edge to August 21 1967. We will consider an edge graph with a typing functor

\textsuperscript{17}Note that this is not uncommon, see for example IFO of Abiteboul [AbiteboulHull87]
as a basic building block of the information in an information system; i.e. basic entities of an instance of an information system are typed complex objects. We can also use the typing functor to validate a typed object graph. If the typing functor is not a homomorphism to a given type graph, then we will say that the typed object graph is syntactically not valid with respect to that type graph.

Note that the restriction we put on the typing functor (i.e. being a homomorphism, or in other words structure preserving) is a syntactical restriction on the denotation of an instance of an information system. The purpose of this restriction is that if we have a schema and a typed instance, we are certain that all the things we wrote down in our instance can be talked about using the definitions in our schema\textsuperscript{18}.

As an aside we note that this typing homomorphism is in a sense an extension to traditional syntactic systems, that do not have complex objects but only atomic ones and aggregates. For example suppose you have a type graph modeling an English sentence as follows:

\textbf{NOUN \cdot VERB}

Then "\textit{John \cdot walks}" would be a syntactic proper instance of the graph modeling a sentence if the typing functor would map "John" to the type "NOUN" and "walks" to the type "VERB". On the other hand the sentence "\textit{John \cdot loves \cdot Mary}" does not have a sensible typing function to the above type graph. In order to construct a homomorphic typing function we have several possibilities:

1. We can map "\textit{John}" to "\textit{NOUN}" and "\textit{loves \cdot Mary}" to "\textit{VERB}"

2. or alternatively "\textit{John \cdot loves}" to "\textit{NOUN}" and "\textit{Mary}" to "\textit{VERB}"

3. or even map "\textit{John \cdot loves \cdot Mary}" to "\textit{NOUN}" and the empty word "" to "\textit{VERB}"

\textsuperscript{18}In other words this means that if we type an object in some instance we should assign to it the initial (most specific) type, because only then can we speak about all the information of the object we wrote down in the instance. If we would have typed an object with a less specific type (i.e. a supertype of the specific type), we will have no means to talk about the object in the way we wrote it down (i.e. as being of a more specific subtype of the type assigned to it). The object then would have adjacents we cannot classify being of that object. If an object is typed properly, then when we interpret the object (semantics!) we can, of course, infer the object being of this less specific type. But this is on the semantic level. On the syntactic level we are only interested in writing things down properly such that when we interpret the things written down, we get a proper meaning for the things written down. In other words the syntactic rules should prevent us writing down things we cannot give a proper meaning to. Chomski said: Colorless green dreams sleep furiously.
3.2. The Syntactic Theory

4. or map the empty word "" to "NOUN" and "John • loves • Mary" to "VERB".

5. More peculiarly we can map "John" to "NOUN" and "loves" to "VERB" and "Mary" to the empty type "1"

None of these alternatives are sensible typing functions for the object graph "John • loves • Mary". The first 4 examples clearly give undesirable types to the objects (it is especially undesirable that an empty word is a NOUN or a VERB). The last example, though, needs some more attention. The last example shows the strength of the empty edge as a type. In the last example the objects "John" and "loves" have a proper type and object "Mary" is postulated to be of the empty type. In a sense this means that we ignore the object "Mary" in our analysis, using the 'datamodel' NOUN • VERB. The empty type enables one to write syntactically correct type graphs and object graphs without the necessity to be fully specific in our analysis. In other words we give the data analyst a chance to (syntactically correct) write down data models and instances during the process of his modeling task, when he does not know the full complexity of the object in the universe of discourse he is modeling. To continue with the above example, only when the analyst has gained the insight that an English sentence could alternatively be of type NOUN • VERB • NOUN, he can fully specific type the instance "John • loves • Mary".

Note that for graphs with complex objects, the empty object plays the same role as the empty or 'unspecified' type. A trivial example for this is a type graph consisting of one atomic edge A and an object graph with an object a and an adjacent b that in turn has an adjacent c; this object graph can be typed as follows: a : A, b : 1, c : 1.

Let us consider two disjoint universes, T and O; and let us reserve Edge(T) for denoting edge graphs for types and Edge(O) for denoting edge graphs for information objects. An edge graph G over T (i.e. a type graph), determines a set containing all typed edge graphs over O (i.e. object graphs) H for which the typing function is a homomorphism from H to G. These typed object graphs are members of this set because they have the signature defined by the type graph G. In information systems we often want to classify objects, not only by their signature, but also by some other properties concerning the whole graph. In information systems these properties are usually called constraints. Examples of constraints in information systems are constraints that are common in relational databases like functional dependencies, join dependencies, inclusion dependencies, primary keys, foreign keys, or constraints that are common in object oriented databases like subtyping constraints, etc.. We want in our language for information systems the ability to express a restriction on a set of object graphs that have the 'proper' signature, that is satisfied by those object graphs of this
signature that also satisfy some additional constraints\textsuperscript{19}. The language of categorial graphs we present in the next section will be able to talk about objects by drawing a type edge graph and adding textual constraints. These categorial graphs will denote data or information models. Instances of these data models can be denoted by object edge graphs.

\textbf{3.2.11. Example.} Let us again take the example of a roster of a pilot (see example 3.1.1). All the entries in the 'roster' object\textsuperscript{20}, i.e. the flights, the time-offs, and the courses, will certainly have things in common, because they are all entries in a roster. Suppose we want to model that by saying that these three kinds of objects are things we call 'tasks'. The attributes of a task are typically the attributes that flights, time-offs and courses have in common, for example start-date-and-time and end-date-and-time. Furthermore, we can say now that a roster is an aggregation of tasks, and that the types 'flight', 'time-off' and 'course' are subtypes of the type 'task'. Moreover, we can force that all tasks in a roster do not overlap in date-and-time. To assert that some type is a subtype of another type, e.g. 'flight' is a subtype of 'task', amounts to putting a constraint on the models that have objects of these types. It says that all objects of type 'flight' should also be objects of type 'task'\textsuperscript{21}. And, obviously, saying that tasks in a roster should not overlap is evidently a constraint on the models that have objects of type 'roster'.

Note that in many formalisms, among which UML is one, the subtype constraint is drawn graphically with an arrow. As subtyping is a matter of constraint and not a matter of signature (structure) we choose to keep it a textual constraint in this formal presentation of the language of categorial graphs, so it will not cause any confusion with the edges that denote matters of signature.

We can draw categorial graphs in many ways. In fact, because we take a very abstract point of view, there are many existing formalisms that might be seen as a drawing of categorial graphs. We want the categorial graphs to be general enough to capture most object oriented information system formalisms. Because the categorial graph language will be given a proper semantics, it provides a vehicle to give a proper semantics to these formalisms

\textsuperscript{19}Note that we are talking about static constraints; i.e. properties that should be satisfied regardless of the past or future of the models (if you let a model be dynamic of course). We will consider dynamic constraints when we look at dynamic models of categorial graphs.

\textsuperscript{20}i.e. an object of type 'roster'.

\textsuperscript{21}Note that for the syntax it suffices for a typing function to map an edge that will be interpreted as an object to only one other edge that will be interpreted as a type. In the semantics however an object of type 'flight' should be both of type 'flight' and type 'task'.

### 3.2.4 Categorial Graphs

A categorial graph will be the syntactic vehicle to define a database schema. A categorial graph consists of an edge graph and a set of phrases. Each edge in categorial graph will represent a type\(^{22}\). The structure of the edge will determine the signature of the type it represents. The phrases will define constraints on the types.

**3.2.12. Example.** Consider figure 3.3. It contains an edge graph \(G\). The figure shows that \(G\) consists of the basic edges\(^{23}\) HUSBAND, WIFE, NAME, SEX and YEAR, four complex edges: one edge MARRIAGE with adjacents HUSBAND, WIFE and DATE, one complex edge BIRTH with adjacents MARRIAGE DATE and PERSON, one complex edge PERSON with adjacents NAME and SEX, and finally one complex edge DATE with adjacent YEAR. Note that we have also written down some constraints:

- a HUSBAND ISA PERSON
- a WIFE ISA PERSON
- the year in the DATE of a MARRIAGE should be after 1848
- the date of a MARRIAGE that is accounted in a BIRTH should be before the DATE of a BIRTH

It is easy to see that the typed object edge graph \(H\) of figure 3.3 is an instance\(^{24}\) of \(G\).

**3.2.13. Definition.** (constraint phrases) Given a universe of edges \(\text{Edge}(T)\) and an edge graph \(G\) over this universe; let \(\text{Prop}\) be the set of propositional variables ranging over \(\text{Edge}(T)\), and let \(\text{Prop}(G) \subseteq \text{Prop}\) be the set of propositional variables ranging over the edges in \(G\). Let \(\text{Con}\) be a set of constants. Furthermore let \(\text{Op}\) be a finite set of operators of arity \(i\). Let also \(\text{Fun}\) be a countable set of function symbols. Then the language \(L\) of constraint phrases is defined as follows:

\[
\begin{align*}
L & = \{ P | \text{(propositional variables)} \\
& \quad \cup \text{Con} | \text{(constants)} \\
& \quad \cup \text{Op}^i(L) | \text{LoOp}^2L | \text{Op}^3(L, L, L) | \ldots \text{(operations)} \\
& \quad \cup \text{Fun}^0(L) | \text{Fun}^1(L) | \text{Fun}^2(L, L, L) | \ldots \text{(function symbols)}
\end{align*}
\]

The set \(\text{phrases}_{L}(G)\) will be defined as the set of all phrases of the form \(L \Rightarrow L\) for which all propositional variables occur in \(G\). The intended meaning for a phrase \(A \Rightarrow B\) is "all objects satisfying \(A\) should satisfy \(B\)".

---

\(^{22}\)Semantics will be given in the next section

\(^{23}\)We consider an edge to be simple (opposed to complex), if it has only the empty arrow 1 as adjacent. We abbreviate its denotation by omitting its adjacents because \(\{1\} = 0\)

\(^{24}\)In the picture of \(H\) we wrote the types after the denotation of the object, e.g. we wrote i:marriage for an object i that is typed to be a marriage
A type graph with constraints

```
marriage => marriage.date.year > 1848
birth => birth.marriage.date < birth.date
```

and an instance

```
Figure 3.3: Example of a categorial graph: Married with children
```
3.2. The Syntactic Theory

We defined very generically the syntax of the phrases (i.e. the constraint language), because there are many possibilities, flavors, operators, constants, functions etc. to choose from. For each different feature one wants to be able to talk about, one needs a hook in the language that enables one to talk about it. Below we will introduce a number of basic constants and operators that are important for expressing constraints.

3.2.14. Definition. (constraint operators) Constants for talking about structure:

1 (representing the empty edge)

Unary operators for talking about structure

\[ \Diamond \] (for talking about adjacency; \( \Diamond A \) means "having an adjacent of type \( A \)"")

Binary operators for talking about structure:

\[ \ast \] (for aggregation; \( A \ast B \) means "being an aggregate of two objects, one of type \( A \) and one of type \( B \)"")

Constants for reasoning:

\[ \bot \] (representing falsehood)

\[ \top \] (representing truth)

Unary operators for reasoning:

\[ \neg \] (for negation: \( \neg A \) means "being not of type \( A \)"")

Binary operators for reasoning:

\[ \sqcap \] (for conjunction: \( A \sqcap B \) means "being both of type \( A \) and of type \( B \)"")

\[ \sqcup \] (for disjunction: \( A \sqcup B \) means "being either of type \( A \) or of type \( B \)"")

Constants to talk about identity

\[ \text{self} \] (representing the object itself; i.e. to talk about the object in whose scope we write down the constraint)

\[ \triangle \]

We also have function symbols in the constraint language. The function symbols enable one to constrain objects within the domain of the type of the object. The most common examples of such functions are arithmetic functions for objects that are natural numbers, or for objects that are strings one can have string operations like alphabetic-ordering or pattern-matching\(^{25}\).

\(^{25}\)comparable with the SQL operator 'like \( \text{pattern} \)',
3.2.15. Definition. (categorical graph) Given a universe of edges $\text{Edge}$ and a language $L$ as above. A categorical graph is a pair $(G, S)$ where

- $G$ is an edge graph over $\text{Edge}$,
- $S \subseteq \text{phrases}_L(G)$ is a set of phrases over the tokens of $G$

3.2.16. Example. Consider again the type graph with constraints of figure 3.3 above. The constraints ornamenting the graph can be formulated as follows in the constraint language:

- \( \text{husband} \Rightarrow \text{person} \)
- \( \text{wife} \Rightarrow \text{person} \)
- \( \text{marriage} \Rightarrow (\Diamond \text{date} \land \Diamond f_{\geq 1848}(\text{year})) \)
- \( \text{birth} \Rightarrow f_{\text{before-in-time}}(f_{\text{get-date}}(f_{\text{get-marriage}}(\text{self})), f_{\text{get-date}}(\text{self})) \)

3.2.5 Imploding and Exploding of categorical graphs

Categorical graphs are structures that have both a graphical and a textual part. The edge graphs are mathematical structures that can be classified as graphical, because they can be represented conveniently with 'pictures' (graphs). We mentioned this before. The advantages of graphical representation of information is stressed a lot of times in many fields of science, especially in the field of artificial intelligence (psychology and graphical knowledge representation) ([Dastani98]). To (bluntly) summarize these advantages it has often been stressed that a picture says more than a thousand words. There exists a limit, though, in using pictures: if the amount of graphical elements in the picture becomes very large, the meaning of the picture becomes incomprehensible\(^{26}\). This phenomenon is thoroughly observed and studied ([Adriaans90]). From a practical point of view it forces graphical syntaxes to provide some mechanism for reducing the number of graphical objects in a picture. In this area also there are some proposals. Most of these proposals, however, are very ad hoc, and imprecise in defining the meaning of the syntactic elements of a reduced picture. Evidently this is problematic. For example, suppose you have a graphical language in which the basic syntactic elements are nodes and edges\(^{27}\). Let us assume that reducing the number of syntactic elements in a picture (text) amounts to imploding a subgraph into a new node. The problem now arises to define a proper meaning for this new node of the imploded graph. This is not trivial, even though we already have a

\(^{26}\) A picture with a thousand objects says totally nothing.

\(^{27}\) i.e. conventional graphs.
3.2. The Syntactic Theory

proper meaning for normal nodes and edges. These new nodes that represent an (imploded) subgraph can hide very complex structure, which becomes apparent only if we explode them. For example the connectivity to the context is lost in the following implosion of a traditional graph:

In edge graphs this matter is solved in a natural manner. Instead of coding the complex structure of an object in atomic graphical elements, each graphical element in an edge graph (i.e. each edge) can be of arbitrary complexity. This means that if we draw an edge like this:

\[ \square \]

it can be the picture of a very simple edge (for example the empty edge 1) or a very complex one. The denotation of an edge contains its whole complex structure.
We may draw it as one simple edge and still interpret it as the complex edge it is. The reason is that we do not encode a complex structure using atomic ones, but give a complex edge the same status as a simple one. They are both edges. We may also choose to reveal something of its complexity by drawing some of its structure, i.e.

All in all we can implode an edge as one syntactic element, and explode it again revealing all of its structure, without running into problems of interpretation. Actually the reason for this freedom is inherent in the difference between the mathematical nature of an edge graph (which we view as syntax) and a drawing of it in a two dimensional space (a denotation we see with our eyes). Just like you can draw a conventional graph in arbitrary many ways, you can vary the drawing of an edge graph. The holistic nature of an edge then enables one to abbreviate the drawing of an edge. In the following picture the effect of the implosion is perfectly clear (objects just hide their structure); i.e.

---

28 An edge contains its own structure.
A little more realistically, recall the pilot example of example 3.1.1. For instance, if one would display the full roster of a pilot, and take along all its details like the flight concerned with the task, the plane that will be flown in this flight, the cabin crew that will be on the particular flight together with their hobbies etc. (note that in reality a task in a roster carries a lot of complex objects as adjacents), the view of the roster would be seriously obscured. Note that in the 'real' world the information of complex objects like a roster is enormous. For example the roster information of the cabin crew of a real airline is typically stored in more then 40 different tables.

3.3 Summary

In this chapter we presented a language for object oriented information systems. This language has both graphical and textual ingredients and a formal syntactic
theory. In the next chapter we will give a proper mathematical semantics to this language. This semantics will reveal the meaning of the constructs in a conceptual object oriented world.