Nested Concept Graphs: Applications for Databases and Mathematical Foundations

Frithjof Dau and Joachim Hereth Correia

Darmstadt University of Technology, Department of Mathematics,
Schlossgartenstr. 7, D–64289 Darmstadt, Germany,
{dau,hereth}@mathematik.tu-darmstadt.de

Abstract. While the basic idea of using Conceptual Graphs as query interface to relational databases has already been stated very early in [Sow84], no approach so far has covered the full expressiveness of modern database query languages. Especially negation and the so-called aggregating functions have not been treated.
In this paper, we present *Nested Concept Graphs with Cuts* which extend the syntactical Concept Graph (which mathematize Conceptual Graphs) to treat simultaneously negation and nesting. With these extensions they have the expressiveness of database query languages (e. g. SQL), which will be exemplified by selected queries.

1 Introduction

The goal of Conceptual Graphs has always been to provide a graphical representation for logic which is able to support human reasoning. Many of the possible applications of such a logical representation system have been described in [Sow84, Sow92, Sow00], but one of the maybe most interesting ones has not been completely worked out until today: a consistent, graphical interface for database interaction.

In [Dau03] one of the authors studied in depth a calculus for the mathematization of Conceptual Graphs and their equivalence to first order predicate logic. The relational algebra is also known to be equivalent to first order predicate logic. However, modern database languages extend this expressivity. Considering especially *aggregate functions*, which operate on tuples instead of relations, we developed *Nested Concept Graphs with Cuts* (in the following shortly called NCGwC) which include *hypostatic abstractions* (cf. [Bur91]) as an important extension.

We believe that the graphical language of Conceptual Graphs can provide a simple, but universal interface to relational databases. In the following section, we will illustrate the properties of NCGwCs via *Nested Query Concept Graphs*, which may be used to visualize queries to relational databases.

The general idea of combining Conceptual Graphs and relational databases has already been approached in [BCHL93, CH94, EGSW00, GE01] from various angles, but *aggregate functions* and *negation* are not considered. NCGwCs represent thus a first approach to represent the full expressivity of modern database languages, which exceeds the expressivity of first order predicate logic.

2 Simple Queries

For relational databases, the standard query language today is SQL (*Structured Query Language*), which has been released in several major versions. The most wide-spread version today is SQL2 from 1992, but the SQL3 standard from 1999 is gaining momentum with the more recent releases of the major database systems.

For the purpose of explaining the crucial elements of Nested (Query) Concept Graphs and their expressivity, we will translate some typical SQL queries into the graphical representation of the corresponding Nested Query Concept Graph.[1] Syntactically, the only difference between Nested Concept Graphs with Cuts and the Nested Query Concept Graphs is the availability of *query marks* (written $?i$) as object name. Those query marks only need special treatment in the evaluation of the graph.

Our examples are based on a small database, consisting of the tables `Person` (with the columns Lastname, Firstname, and Age) and `ResearchArea` (with the columns Lastname and Topic). The first example shows a rather simple query, asking for the Lastname of people in the table `Person` whose age is below 30. In SQL this might be expressed as follows:

```
SELECT Lastname FROM Person WHERE Age <= 30
```

The representation of the corresponding Nested Query Concept Graph is as follows:

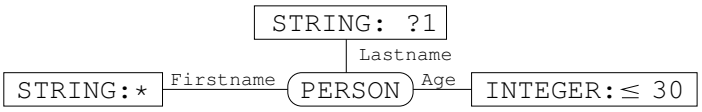


Fig. 1. A simple query graph

Please note that we have used a simple graphical shortcut. The right-hand node [INTEGER:≤ 30] is a short form for [INTEGER:*]–(≤)–[INTEGER:30]. Every incoming arc

[1] A more detailed version can be found in [DH03a].

is to be read to belong to the node with the asterisk. The same short form may be used for the other comparison relations ($<$, $\leq$, $>$, $\geq$, $=$).

This simple example only uses the most simple primitives of graphs, no nesting and no negation. Therefore, it is syntactically covered by the definitions of [Dau03], if we add the query marks $?i$ to the object names. Likewise, the example is comparable to the corresponding graphs in [BCHL93].

Of course, queries may also draw data from more than one table. For getting a list of topics of the researchers below 30, a query in SQL might look like this:

```
SELECT Topic FROM Person, ResearchArea
  WHERE Person.Lastname=ResearchArea.Lastname  AND Age <= 30
```

This query builds the cross product of the tables `Person` and `ResearchArea` and selects the entries in the column `Topic`, provided that the Lastnames in both tables are the same and the Age is below 30. The condition of equal values in certain columns is a standard condition for joining tables. For this reason, the SQL standard provides the JOIN operation,[2] with which we may reformulate the SQL query as follows:

```
SELECT Topic FROM Person JOIN ResearchArea USING (Lastname)
  WHERE Age <= 30
```

These two versions also represent different approaches one could take if one wants to get the answer by constructing intermediate tables by hand. However, most modern database systems will optimize the queries anyway, so the actual process in the system will probably be the same for both versions. Interestingly, the corresponding query graph looks also almost the same for both versions, as shown in Figure 2.

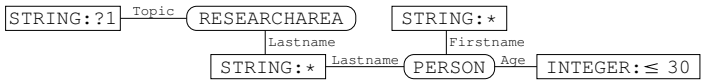


Fig. 2. A query graph joining two relations

The syntactical difference between the two formulations of the SQL query is not reflected in the graph. Of course, we can interpret the construction of the two SQL queries in graphical terms. The cross product of two relations means

[2] JOIN is not considered in [BCHL93, CH94], but JOIN operations are equivalent to operations using cross products and selects, thus are in the expressivity of the SQL fragment used in those works.

simply juxtaposing the two corresponding graphs. The equality condition then additionally joins the two concept boxes on the Lastname arcs with the equality relation. The graph shown in Figure 2 then is a simple equivalent transformation, merging these two nodes together. Interpreting the join of two graphs is even simpler; when joining graphs you either have to join the two boxes that belong to arcs with the same name (which has to be unique according to the SQL standard), or you have to denote explicitly over which arcs the relations should be joined (as it is done in our example). Then, you have the resulting joined relation. Of course, the result is equivalent, and which graphical representation is preferable depends on the taste of the user.

3 Negation in Queries

The examples we presented in the previous section were simple in the sense that to evaluate them we need only positive knowledge about the data. But often arise questions not about what is, but about what is not. Visualizing queries with negation is rarely seen in commercial tools, probably because the visualization of the negation seems to be difficult. However, Charles Sanders Peirce found a way to do exactly this in his Existential Graphs (cf. [Pei92, PS00]). He used ovals to mark the part of the graph which should be negated and developed a calculus for this diagrammatic logic system. In [Dau03], the notion of *cuts* (as Peirce called these ovals) has been adopted for the system of Concept Graphs. Continuing this idea for our query graphs, we have thus a mechanism to make negation available as a graphical primitive.

Our next example selects those people that are not focused on researching CGs, but also have other research areas:

```
SELECT Person.Lastname,Firstname FROM Person,ResearchArea
  WHERE Person.Lastname=ResearchArea.Lastname AND Topic <> 'CG'
```

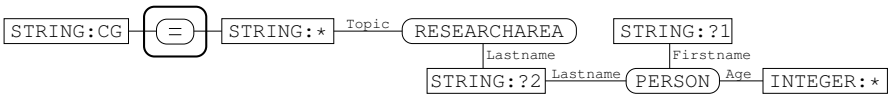


Fig. 3. A query graph with cut (negation)

A similar question is: What researchers are not interested in CGs? Semantically, there is only a small difference: in one case the question is if there are other research areas, in the second if there are *only* other research areas. Hence, it is intuitive to expect a similar graph. In the graphical version, this means the same structure, but a different part of the graph is negated, as we can see in Figures 3 and 4.

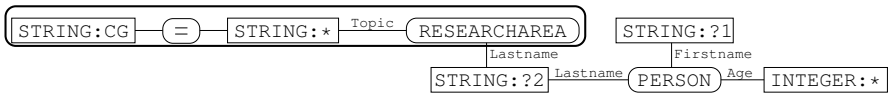


Fig. 4. Similar graph with different meaning

However, in SQL the second query is rather different:

```
SELECT Lastname,Firstname FROM Person WHERE Lastname NOT IN
  ( SELECT Lastname FROM ResearchArea
      WHERE Topic='CG')
```

We can easily see that this query has the same result as the query graph in Figure 4. However, the graph is *not* the canonical translation of this SQL query (this will be discussed at the end of the next section). Graphs are parallel in nature, therefore we can write both queries ("Who works on CGs?" and "What are the peoples names?") side by side and make the necessary (negated) connection. In SQL we could write a simple query if we had not the negation in the question. Because of the negation, we get a completely different structure and have to declare a subquery. This is part of the reason why SQL looks so unapproachable for beginners; to solve related problems you have to write very different SQL statements, and sometimes even learn new syntactical elements.

As elaborated in [Dau03], the tools we have presented so far provide the expressiveness of first order predicate logic. More specifically this means that we have the usual set operations like intersection, union, and complement available. The meaning of the latter naturally depends on the method used for the evaluation. This topic is treated in the discussion on safety of queries in [AHV95], and the approaches discussed there translate directly to approaches we can take for our database application. An important point is to assume that the universe contains only the objects known to the database (the closed-world-assumption), the so-called *domain* of the database. In [Dau03], this happens naturally by using the object set of the underlying power context family as domain.

4 Subqueries and Nested Graphs

So far, we have only considered relations between elements of our object set. The examples presented so far are all covered by the work presented in [Dau03], but in practice we need more expressivity. Not reasoning with, but reasoning about relations is a topic that has not been addressed by Codd in [Cod70]. However, queries aiming at summarizing or *aggregating* sets are very important in daily (database) life. Therefore, Codds relational calculus had to be adapted on the theoretical side (cf.[Klu88]). In the SQL standard, we find for this the term *set function*. A set function does not relate objects to objects, but objects to

relations, thus set functions break the idea we have followed so far. The SQL standard provides five such set functions. They are `AVG` (average), `MAX`, `MIN`, `SUM`, and `COUNT`. All of them basically do one thing: They take a set as input and return a simple element as output.

In his book 'A Peircean Reduction Thesis' [Bur91], Robert Burch provides an algebraization of Peirces logic system and analyzes Peirces thesis that any relation may be constructed out of ternary relations. For his solution, he has to abstract from the relations and use them as objects in other relations, he calls those abstractions *hypostatic abstractions*. While a theory involving this concept may look confusing at first, the concept itself is very natural. In reasoning, humans sometimes consider the elements of a set and their properties, and sometimes the set itself as an element on its own having other properties. This shift in perspective transforms a relation into an object which then may be attached to relations again. In visualization, we use nested concept boxes as known from [Sow84, Sow92, Sow00].

To exemplify hypostatic abstractions, we consider the question about the average age of the researchers. In SQL, we need for this the set function `AVG` and write:

```
SELECT AVG(Age) FROM Person
```

The difference to the previous queries is obvious: Before, we were interested in what is in the relation `Person`, now we are talking about a summarizing property of the relation itself. While the information we were looking for was contained in the tuples of the relation before, no single tuple can provide the answer now, only all of them together. Using the idea of hypostatic abstraction, we get the query graph depicted in Figure 5. Intuitively, by drawing the box around the description of a relation (in the example, of the plain relation `Person`), we transform it into an object, where we may attach relations like `AVG_2`, which means building the average on the second column. Of course, the two query marks ?1 that appear in the picture are not related, i. e. the query marks are only valid in their respective contexts (see [DH03b] for details). Modeling the set functions this way, we follow the presentation of the relational calculus with aggregate functions in [AHV95, Chap. 20].

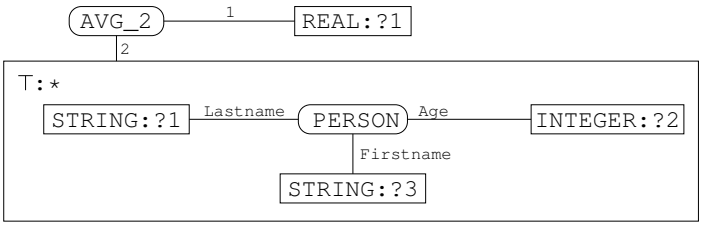


Fig. 5. A query graph with *hypostatic abstraction*

From the user point of view, an extension from this query to the question of average age per research topic seems rather simple. Using the graphical metaphor, it is: simply add the `ResearchArea` relation and add a query mark for the Topic string (cf. Figure 6). While this may look simple at first glance, it has also an effect on the relation defined by the hypostatic abstraction – after all, we draw a connection from the outer context into the box defining the relation! Connecting a part of it to a node outside results either in a selection (if the node refers to a constant) or in a grouping, where the tuples that have the connecting property in common are grouped together.

In SQL we find the same idea, but we have to use a new syntactical construction, the `GROUP BY`. Reading the standard, you find them writing about the result of a `GROUP BY` being a set of groups. But because relations as elements of the resulting relation are not allowed, it has to be assured that the result becomes flat again by adding set functions. As we know from our own experience, this too confuses beginners. The full SQL statement for the last question looks like this:

```
SELECT AVG(Age),Topic FROM Person, ResearchArea
  WHERE Person.Lastname=ResearchArea.Lastname
    GROUP BY Topic
```

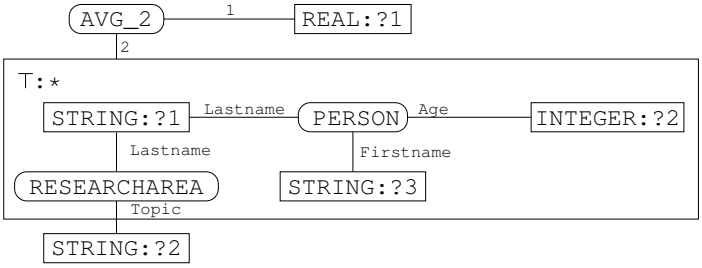


Fig. 6. A query graph extending the previous query

The last complicated part of the query specification in the SQL standard is the `<having clause>`. Again, this has to do with the fact that SQL has to take different syntactical elements depending if they want to talk about elements or groups. Roughly said, the `<having clause>` is analogue of the `<where clause>` for groups. As our groups appear in the form of (complex) vertices in the graphical representation, the translation of the `<having clause>` does not introduce new graphical elements. Let us consider the `GROUP BY` query from the last example. Now, we want to put an additional restriction on the groups, the average age should be below 30, but we are not interested in the exact average age. So we ask, for which research topics is the average age below 30? In the graphical notation, the graph looks almost exactly as the former one, with a small change in the obvious position (Figure 7).

In SQL, we use the keyword `HAVING` and get:

```
SELECT Topic FROM Person,ResearchArea
  WHERE Topic.Lastname=ResearchArea.Lastname
  GROUP BY Topic
    HAVING AVG(Age) < 30
```

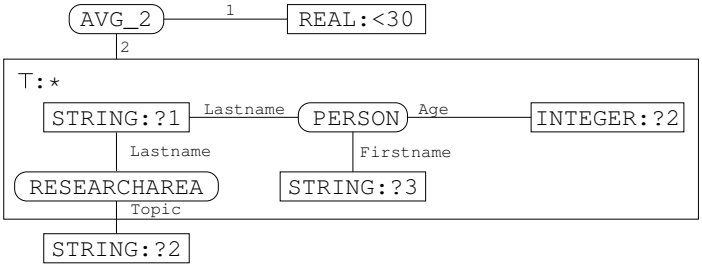


Fig. 7. A query graph corresponding to a SQL query with `HAVING`

At the end of the discussion how to represent SQL queries in a graphical form, we return to the query of the previous section, where we presented the graph in Figure 4 which was equivalent but not the canonical translation of the given SQL query. After having introduced the hypostatic abstraction, we can now also represent the more direct representation as graph, as is shown in Figure 8.

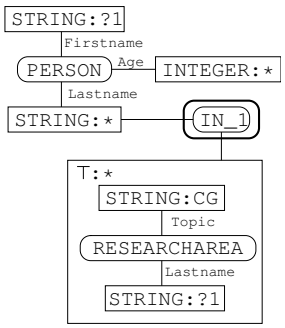


Fig. 8. A query graph for a SQL subquery

Subqueries are indeed a delicate part of SQL. Some common databases (e. g. MySQL, on which we tested all example queries except the subquery one) do not provide this functionality. The documentation proposes to use temporary tables, which is conceptually the same while being more effort to the user. For our concerns, we have to investigate the `IN_`i relation shown in the picture. It

means, that the given element is an element of the first column of the connected relation. It is easily comprehensible that those relations are not essential for the diagrammatic representation of queries. To be or not to be in the set of elements having some property is of course equivalent to having this property or not. This is the reason why we did not need the concept of hypostatical abstraction to represent the query. However, for didactical reasons it might be preferable to have the `IN_`i relations available in a real system.

5 The Structure of Nested Concept Graphs with Cuts

In this section, we define the underlying structure of NCGwCs. As can be seen in the last sections, we will have two different kinds of vertices: simple vertices, and complex vertices (named *hypostatic abstractions*, HAs), which contain further elements, like other vertices or edges, of the graph. A vertex is considered to be an HA if and only if it contains other elements of the graph, i. e. we can already see from the underlying structure, not from the (later introduced) labelling of vertices and edges, whether a vertex is a simple vertex or a HA.

Definition 1. *A structure* $(V, E, \nu, \top, Cut, area)$ *is called a* Nested Relational Graph with Cuts *if*

- V, E *and* Cut *are pairwise disjoint, finite sets whose elements are called* vertices, edges *and* cuts, *respectively,*
- $\nu : E \to \bigcup_{k\in\mathbb{N}} V^k$ *is a mapping,*
- $\top$ *is a single element, called the* sheet of assertion, *and*
- $area : V \dot\cup Cut \dot\cup \{\top\} \to \mathfrak{P}(V \dot\cup E \dot\cup Cut)$ *is a mapping such that*
  - *a)* $area(k_1) \cap area(k_2) \neq \emptyset \Rightarrow k_1 = k_2$ *for* $k_1, k_2 \in V \dot\cup Cut \dot\cup \{\top\}$
  - *b)* $V \cup E \cup Cut = \bigcup\{area(k) \mid k \in V \dot\cup Cut \dot\cup \{\top\}\}$,
  - *c)* $x \notin area^n(x)$ *for each* $x \in V \cup \{\top\} \cup Cut$ *and* $n \in \mathbb{N}$ *(with* $area^0(x) := \{x\}$ *and* $area^{n+1}(x) := \bigcup\{area(y) \mid y \in area^n(x)\}$*).*

For an edge $e \in E$ *with* $\nu(e) = (v_1, \ldots, v_k)$ *we define* $|e| := k$ *and* $\nu(e)\big|_i := v_i$. *Sometimes, we will write* $e\big|_i$ *instead of* $\nu(e)\big|_i$, *and we will write* $e = (v_1, \ldots, v_k)$ *instead of* $\nu(e) = (v_1, \ldots, v_k)$. *We set* $E^{(k)} := \{e \in E \mid |e| = k\}$.

For $v \in V$ *we set* $E_v := \{e \in E \mid \exists\, i.\nu(e)\big|_i = v\}$. *Analogously, for* $e \in E$ *we set* $V_e := \{v \in V \mid \exists\, i.\nu(e)\big|_i = v\}$.

We set $HA := \{v \in V \mid area(v) \neq \emptyset\}$. *The elements of* HA *are called* hypostatic abstractions.[3] *The elements of* $V \setminus HA$ *are called* simple vertices.

The elements of $Cut \dot\cup HA \dot\cup \{\top\}$ *are called* contexts. *As every* $x \in V \cup E \cup Cut$ *is directly enclosed by exactly one context* $k \in Cut \dot\cup HA \dot\cup \{\top\}$, *we can write* $k = area^{-1}(x)$ *for every* $x \in area(c)$, *or more simple and suggestive:* $k = ctx(x)$.

[3] In [Pre98], the are called *complex vertices.*

NCGwCs will be constructed from Nested Relational Graph with Cuts by additionally labelling the vertices and edges with names. There is a crucial difference between Concept Graphs and most other languages of logic: Usually, the well-formed formulas of a language are built up inductively. In contrast to that, NCGwCs are defined in one step. The structure of a formula in an inductively defined language is given by its inductive construction. Although definded in one step, NCGwCs bear a structure as well: Similar to simple Relational Graphs with Cuts, a context k of a Nested Relational Graph with Cuts may contain other contexts l in its area (i. e. $l \in area(k)$), which in turn may contain further contexts, etc. Is has to be expected that this idea induces an order $\leq$ on the contexts which should be a tree, having the sheet of assertion $\top$ as greatest element. This order reflects the structure of the graph. The naïve understanding of $\leq$ is to set $l < k$ iff l is 'deeper nested' than k. The next definition is the mathematical implementation of this naïve idea. Furthermore the definition extends this idea to the set of vertices and edges. This order will be important in the evaluation of contexts (cf. [DH03b]).

Definition 2. *Let $(V, E, \nu, \top, Cut, area)$ be a Relational Graph with Cuts. Define $\beta : V \cup E \cup Cut\dot\cup\{\top\} \to Cut\dot\cup HA\dot\cup\{\top\}$ by*

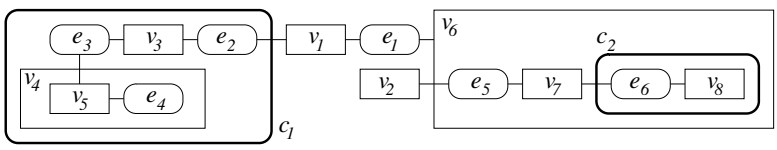
$$\beta(x) := \begin{cases} x \text{ for } x \in Cut\dot\cup HA\dot\cup\{\top\} \\ ctx(x) \text{ for } x \in (V \backslash HA) \cup E \end{cases}.$$

We set $x_1 \leq x_2 :\Leftrightarrow \exists n \in \mathbb{N}_0 . \beta(x_1) \in area^n(\beta(x_2))$ *for* $x_1, x_2 \in V\dot\cup E\dot\cup Cut\dot\cup\{\top\}$.

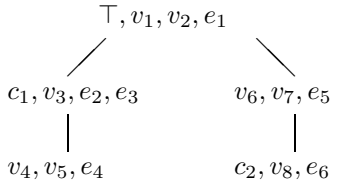
In order to avoid misunderstandings, let $x < y :\Longleftrightarrow x \leq y \wedge y \not\leq x$, $x \lneq y :\Longleftrightarrow x \leq y \wedge y \neq x$, and $x \sim y :\Longleftrightarrow x \leq y \wedge y \leq x$. Moreover, for $c \in Cut\dot\cup\{\top\}$, we define shortcuts for the ideals $\leq[c] := \{x \in V \cup E \cup Cut\dot\cup\{\top\} \mid x \leq c\}$ and $\lneq[c] := \{x \in V \cup E \cup Cut\dot\cup\{\top\} \mid x \lneq c\}$.

The following is an example of a well-formed Nested Relational Graph with Cuts, its diagram (we have written the names for the vertices end edges inside them), its mapping β and the induced relation $\leq$. We see that $\leq$ is a quasiorder and that each equivalence class of the induced equivalence relation $\sim$ contains exactly one context. This will be elaborated after the example.

$$\begin{aligned}\mathfrak{G} := (&\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}, \{e_1, e_2, e_3, e_4, e_5, e_6\}, \\ &\{(e_1, (v_1, v_6)), (e_2, (v_3, v_1)), (e_3, (v_5, v_3)), (e_4, (v_5)), (e_5, (v_2, v_7), (e_6, (v_7, v_8))\}, \\ &\top, \{c_1, c_2\}, \{(\top, \{v_1, v_2, v_6, e_1, c_1\}), (c_1, \{v_3, v_4, e_2, e_3\}), \\ &\quad (v_4, \{v_5, e_4\}), (v_6, \{v_7, e_5, c_2\}), (c_2, \{v_8, e_6\})\})\end{aligned}$$



x	$\top$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	e_1	e_2	e_3	e_4	e_5	e_6	c_1	c_2
$\beta(x)$	$\top$	$\top$	$\top$	c_1	v_4	v_4	v_6	v_6	c_2	$\top$	c_1	c_1	v_4	v_6	c_2	c_1	c_2



As already mentioned, it is conjecturable that $\leq$ is a quasiorder. This is not immediately clear from the definitions, but it can be proven. Analogously to [Dau03], we get the following lemma:

Lemma 1. *Let $(V, E, \nu, \top, Cut, area)$ be a Nested Relational Graph with Cuts. We have $\leq[k] = \bigcup\{area^n(k) \mid n \in \mathbb{N}_0\}$ and $\lneq[k] = \bigcup\{area^n(k) \mid n \in \mathbb{N}\}$ for each $k \in Cut \dot\cup HA \dot\cup \{\top\}$. Moreover, $\leq$ is a quasiorder such that the restritction $\leq|_{Cut \dot\cup HA \dot\cup \{\top\}}$ is an order on $Cut \dot\cup HA \dot\cup \{\top\}$ which is a tree with the sheet of assertion $\top$ as greatest element.*

Now it is easy to describe the mapping ctx in a purely order-theoretic way. It can be shown that we have $ctx(x) = \min\{k \in Cut \dot\cup HA \dot\cup \{\top\} \mid x \lneq k\}$ for each element $x \in V \dot\cup E \dot\cup Cut \dot\cup \{\top\}$ of a Nested Relational Graph with Cuts $(V, E, \nu, \top, Cut, area)$. This characterization of the mapping ctx is adopted for hypostatic abstractions. Sometimes we have to consider all elements which are enclosed by an hypostatic abstraction, even if they are deeper nested in some cuts, but *not* if they are contained by a deeper nested hypostatic abstraction. These are the elements x with $ha(x) = h$. In order to to this, it is evident to define a mapping ha (which corresponds to ctx for contexts) and $encl$ (which corresponds to $area$ for contexts) as follows:

Definition 3. *Let $(V, E, \nu, \top, Cut, area)$ be a Nested Relational Graph with Cuts. We set*

$$ha : \begin{cases} V \dot\cup E \dot\cup Cut \to HA \dot\cup \{\top\} \\ \qquad\qquad x \mapsto min\{h \in HA \dot\cup \{\top\} \mid x \lneq h\} \end{cases}$$

Now, we define a mapping $encl : HA \to V \dot\cup E \dot\cup Cut$ by: $encl^1(h) := area(h)$, $encl^{n+1}(h) := \bigcup\{area(c) \mid c \in encl^n(h) \cap Cut\}$, and finally $encl(h) := \bigcup\{encl^n(h) \mid n \in \mathbb{N}\}$.

Similar to the order-theoretic description of ctx, it can be shown that we have $encl(h) = \{x \in V \dot\cup E \dot\cup Cut \mid ha(x) = h\}$ for a HA $h \in HA$ in a graph $\mathfrak{G} := (V, E, \nu, \top, Cut, area)$.

We will only consider graphs in which vertices must not be deeper nested than any relation they are incident with. This applies to all contexts, that is to cuts as well as to hypostatic abstractions.

Definition 4. *Let* $\mathfrak{G} := (V, E, \nu, \top, Cut, area)$ *be a Nested Relational Graph with Cuts. If* $ctx(e) \leq ctx(v)$ *for every* $e \in E$ *and* $v \in V_e$*, then* $\mathfrak{G}$ *is said to have* dominating nodes.

6 Syntax for Nested Concept Graphs with Cuts

6.1 Types and Names

With simple Concept Graphs, we can denote relations between the objects of a given set of objects with relation names. Relations always have a fixed arity. For this reason, we assign in the syntax of simple Concept Graphs (and other logic languages like first order logic as well) to each relation name an arity.

The set functions like `COUNT` or `SUM` of SQL can be understood as relations between numbers and relations. For example, our understanding of `COUNT` is that `COUNT` is the following dyadic relation: A number n stands in relation `COUNT` to a relation R if and only if R contains exactly n tuples. So `COUNT` is not a classical relation on the set of objects, but a relation between objects and relations. To describe the structure of such relations, we will use *signatures* instead of arities. A signature will not only give the arity of a relation, but they will furthermore describe for each place of a tuple the structure of the entries in that place, e. g. whether the entry is an object of our ground set or a relation itself. We will use the sign $\star$ to denote an arbitrary object of our ground set. A 'classical' k-ary relation on our ground set will be therefore described by the signature $(\star, \ldots, \star)$ (with k stars '$\star$'). In particular, signatures are a generalization of arities.

Definition 5 (Signatures).

Let '$\star$*' be a sign. We set it to be a signature, and if* $S_1, \ldots, S_n$*,* $n \in \mathbb{N}_0$ *are signatures, then* $(S_1, \ldots, S_n)$ *is a signature. The set of all signatures is denoted by* Sig*. We partition* Sig *into the following types of signatures: The sign* $\star$ *is called* object signature*, the signature* $()$ *is called* boolean signature*, every signature* $(\star, \ldots, \star)$ *is called* simple relation signature*, and all remaining signatures are called* nested relation signatures*. We set furthermore* $Sig^0 := Sig \backslash \{\star\}$.

Next we have to define an alphabet for NCGwCs. In the case of simple Concept Graphs, we had relation names and assigned to each relation name its arity. Thus, for NCGwCs, the approach which suggest itself is to assign signatures to relation names. But this approach is problematic, which can be again explained with the relation `COUNT` in SQL. We have in fact an infinite number of `COUNT`-relations, namely a relation `COUNT` between natural numbers and unary relations, a relation `COUNT` between natural numbers and dyadic relations and so on. But when `COUNT` is applied in a SQL-statement, it is applied to a relation where its signature is known, so we know which of the different `COUNT`-relations should be used. Thus we define names as pairs of so-called plain names and their signatures, i. e. a plain name can be used together with different signatures (in the terminology of object-oriented languages: We *overload* names). This yields the following definition.

Definition 6 (Alphabet).

An alphabet *is a pair* $(\mathcal{C}, \mathcal{N})$*, where*

- $(\mathcal{C}, \leq_{\mathcal{C}})$ *is a finite ordered set whose elements are called* concept names*, and*
- $(\mathcal{N}, \leq_{\mathcal{N}})$ *is a finite ordered set which consists of pairs* (N, S) *with* $S \in Sig$*. Each element* $(N, S) \in \mathcal{N}$ *is called* signed name*. For the signed name* (N, S)*, we call* N *the* (plain) name *of* (N, S) *and* S *the* signature *of* (N, S)*. We demand that only names with the same signature can be compared, that is we demand* $(N_1, S_1) \leq (N_2, S_2) \Longrightarrow S_1 = S_2$ *.*

Let $(N, S) \in \mathcal{N}$ *be a signed name. If* $S = \star$*, then* (N, S) *is called* object name*. The set of all object names is denoted by* $\mathcal{N}_{\mathcal{G}}$*. If* $S = ()$*, then* (N, S) *is called* boolean name*. The set of all boolean names is denoted by* $\mathcal{N}_{\mathcal{B}}$*. If* $S = (\star, \ldots, \star)$*, then* (N, S) *is called* simple relation name*. The set of all simple relation names is denoted by* $\mathcal{N}_{\mathcal{R},s}$*. All remaining signed names* (N, S) *are called* nested relation name*. The set of all nested relation names is denoted by* $\mathcal{N}_{\mathcal{R},n}$*.*

Simple relation names and nested relation names are both called relation names*. The set of all relation names (i. e.* $\mathcal{N}_{\mathcal{R},s} \dot{\cup} \mathcal{N}_{\mathcal{R},n}$*) is denoted by* $\mathcal{N}_{\mathcal{R}}$*.*

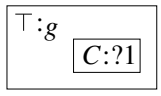
We demand that we have a plain name $\doteq$ *with* $(\doteq, (S, S)) \in \mathcal{N}$ *for each* $S \in Sig$*.*

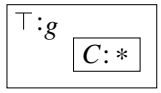
6.2 Nested Concept Graphs with Cuts

NCGwCs are built from nested relational graphs with Cuts by additionally labelling the vertices and edges with names. There are two important points we have to meet:

First of all, when writing the diagram of a NCGwC, it is desirable that it is sufficient to use plain names.[4] That is, *if we use a plain name in the diagram of a Concept Graph, it has to be clear from the context which* signed *name is denoted by the plain name.*

Secondly, it is important that the labelling is (syntactically) reasonable, i. e. that the graph can be interpreted in a model. To get an impression of this reflections, consider the two diagrams on the right: (where C is the plain part of a concept name and g is a plain name):

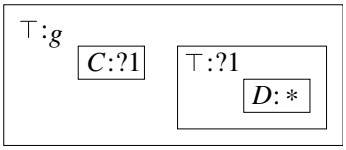
$\mathfrak{G}_1 :=$ 

$\mathfrak{G}_2 :=$ 

The hypostatic abstraction (that is the outer vertex) in $\mathfrak{G}_1$ shall describe a (unary) simple relation, which can be seen from the inner vertex, which is labelled with a query marker $?i$. So $\mathfrak{G}_1$ can only be the diagram of a NCGwC if g is the plain part of the simple relation name $(g, (*))$. Analogously, $\mathfrak{G}_2$ can only be the diagram of a NCGwC if g is the plain part of the boolean name $(g, ())$.

[4] This again corresponds the the overloading of function names in object oriented languages. When we have a function-call in the program code, it has to be clear from the context which implementation of the function is meant.

Furthermore, we have another problem which may occur. Consider the graph on the right. Again we have the question which signature g should have.

$\mathfrak{G}_3 :=$ 

From the first vertex in the outer hypostatic abstraction we conclude that g is a unary *simple* relation name, but from the second vertex in the outer hypostatic abstraction we conclude that g is a unary *nested* relation name. This is contradictory, so $\mathfrak{G}_3$ should not be the diagram of a well-formed NCGwC.

For the graphs $\mathfrak{G}_1$, $\mathfrak{G}_2$, $\mathfrak{G}_3$, the discussions of the signature of the name g had to refer not only to the underlying structure (the Nested Relational Graph with Cuts) of the graph, but to other other names in the graph as well. For this reason, the definition of NCGwCs is done in two steps. In the first step, we label the vertices and edges of a Relational Graph with Cuts with signed names. This will yield *quasi nested concept graphs with Cuts*. Based on the labelling, we can assign to each hypostatic abstraction h (and to other vertices and to the edges) a set of signatures which is derived from the subgraph enclosed by h ($\mathfrak{G}_3$ shows that we sometimes can not derive a unary signature for a given hypostatic abstraction). After that, we only consider quasi nested Concept Graphs with Cuts where each vertex and edge has a uniquely determined signature and where all vertices and edges are labelled accordingly to these signatures.

Definition 7 (Quasi Nested Concept Graph with Cuts).

A quasi nested Concept Graph with Cuts over the alphabet $(\mathcal{C}, \mathcal{N})$ *is a structure* $\mathfrak{G} := (V, E, \nu, \top, Cut, area, \kappa, \rho)$ *where*

- $(V, E, \nu, \top, Cut, area)$ *is a Nested Relational Graph with Cuts that has dominating nodes,*
- $\kappa : V \cup E \to \mathcal{C} \dot\cup \{\top\} \dot\cup \mathcal{N}$ *is a mapping such that* $\kappa(V \backslash HA) \subseteq \mathcal{C} \dot\cup \{\top\}$, $\kappa(HA) = \{\top\}$ *and* $\kappa(E) \subseteq \mathcal{N}^{\mathcal{R}}$, *and*
- $\rho : V \to \mathcal{N} \dot\cup \{*\} \dot\cup \{?_i \mid i \in \mathbb{N}\}$ *is a mapping such that for all* $h \in HA$ *exists a natural number* $ar(h) \in \mathbb{N}_0$ *with*

$$\{i \mid \exists v \in encl(h) \text{ with } \rho(v) = ?i\} = \{1, \ldots, ar(h)\}\,.$$

Moreover we set $V^* := \{v \in V \mid \rho(v) = *\}$, $V^? := \{v \in V \mid \rho(v) = ?i,\ i \in \mathbb{N}\}$, *and* $V^{\mathcal{G}} := \{v \in V \mid \rho(v) \in \mathcal{N}\}$. *The nodes in* V^* *are called* generic nodes, *the nodes in* $V^?$ *are called* query nodes, *and the nodes in* $V^{\mathcal{G}}$ *are called* object nodes.

If a quasi nested Concept Graph with Cuts is given, we can assign to all vertices and edges a set of signatures.

Definition 8 (Signatures of Vertices, Edges, Hypostatic Abstractions).

Let $\mathfrak{G} := (V, E, \nu, \top, Cut, area, \kappa, \rho)$ be a quasi nested Concept Graph with Cuts over the alphabet $(\mathcal{C}, \mathcal{N})$. We assign sets of signatures to vertices, edges and hypostatic abstractions as follows:

1. *For $v \in V \backslash HA$, we set $sig(v) := \{\star\}$.*
2. *Let $h \in HA$. We set $sig(h)|_i := \{sig(v) \,|\, v \in V$ with $ha(v) = h$ and $\rho(v) = ?i\}$ for $1 \leq i \leq ar(h)$, and $sig(h) := sig(h)|_1 \times \ldots \times sig(h)|_{ar(h)}$.*
3. *For $e \in E$ with $|e| = k$, we set $sig(e) = sig(e|_1) \times \ldots \times sig(e|_k)$.*

We will only consider graphs where the signatures of hypostatic abstractions (and hence for edges, too) are uniquely determined. Furthermore, we demand that the signed names we assign to vertices and edges match the signatures of the vertices and edges, respectively. This yields the following definition:

Definition 9 (Nested Concept Graphs with Cuts).

A nested Concept Graph with Cuts over the alphabet $(\mathcal{C}, \mathcal{N})$ *is a quasi nested Concept Graph with Cuts $\mathfrak{G} := (V, E, \nu, \top, Cut, area, \kappa, \rho)$ over the alphabet $(\mathcal{C}, \mathcal{N})$, where:*

- *For each $h \in HA$, we have $|sig(h)| = 1$. For $x \in V \dot\cup E$, we write $sig(x) = S$ instead of $sig(x) = \{S\}$. Moreover, for $1 \leq i \leq ar(h)$, we have $sig(h)|_i = \{S\}$ for an $S \in Sig$, and we write $sig(h)|_i = S$ instead of $sig(h)|_i = \{S\}$.*
- *If $v \in V^{\mathcal{G}} \backslash HA$ and $\rho(v) = (N, S)$, then $S = \star$ (i. e. (N, S) is an object name).*
- *If $h \in V^{\mathcal{G}} \cap HA$ and $\rho(h) = (N, S)$, then $S = sig(h)$.*
- *If $e \in E$ and $\kappa(e) = (N, S)$, then $S = sig(e)$.*

If we have $HA = \emptyset$ and if $\rho : V \to \mathcal{N} \dot\cup \{*\}$ holds, then $\mathfrak{G}$ is called simple Concept Graph with Cuts.

For the set E of edges, let $E^{id} := \{e \in E \,|\, \kappa(e) = \doteq\}$ and analogously $E^{nonid} := \{e \in E \,|\, \kappa(e) \neq \doteq\}$. The elements of E^{id} are called identity-links.

For each vertex and each edge, we can read the signature of the vertex and edge from the graph, thus we can reconstruct the signature of the names which are assigned to the vertices and edges. Furthermore, we know that all vertices and edges are labelled with names which match their signatures. For this reason, in the diagrams of the graphs it is sufficient to label edges and vertices with plain names instead of signed names.

7 Outlook

As said in the introduction, this is only the beginning of a mathematical foundation for NCGwCs, and there is more work to do. A first step towards extensional

semantics of NCGwCs can be found in [DH03b], where nested power context families are introduced as models, and where it is shown how NCGwCs are evaluated in these models. These semantics have to be further adapted for databases, where some relations are stored in tables (and these relations are always simple, i. e. they are relations on the elements of the domain), while all other relations (esp. the set-functions) are computed on the fly by the database engine.

Incorporating the fact that the ground domain of a database is typed (by built-in types like `INT`, `REAL` or `STRING` will yield a reasonable syntactical and semantical extension of [DH03b]. Therefore, a theory with typed signatures is desirable. First attempts have shown that this approach leads to surprising difficulties.

Moreover, it has to be investigated whether the expressiveness of the graphs is restricted enough such that a sound and complete calculus is possible, and a caluculus should be developed, if this is the case.

Finally, it is desirable that there will be implementations of these graphs to query existing databases: Only human interaction can show whether the considerations of [DH03a] and this paper yield working and comprehensible query graphs for databases.

References

[AHV95] Abiteboul, Serge, Richard Hull, and Victor Vianu: *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.

[BCHL93] Boksenbaum, Claude, Boris Carbonneill, Ollivier Haemmerlé, and Thérèse Libourel: *Conceptual graphs for relational databases*. In Mineau, Guy, Bernard Moulin, and John F. Sowa (editors): *Conceptual Structures: Conceptual Graphs for Knowledge Representation. First International Conference on Conceptual Structures, ICCS 1993*, number 699 in *LNAI*, pages 142–161, Quebec City, Canada, August 1993. Springer, Berlin – Heidelberg – New York.

[Bur91] Burch, Robert W.: *A Peircean Reduction Thesis: The Foundation of Topological Logic*. Texas Tech. University Press, Lubbock, 1991.

[CH94] Carboneill, Boris and Ollivier Haemmerlé: *Standardizing and interfacing relational databases using conceptual graphs*. In Tepfenhart, William M., Judith P. Dick, and John F. Sowa (editors): *Conceptual Structures: Current Practices. Second International Conference on Conceptual Structures, ICCS 1994*, volume 835 of *LNAI*, pages 311–330, College Park, USA, August 1994. Springer, Berlin – Heidelberg – New York.

[Cod70] Codd, E. F.: *A relational model of data for large shared data banks*. Communications of the ACM, 13(6):377–387, 1970.

[Dau03] Dau, Frithjof: *The Logic System of Concept Graphs with Negations and its Relationship to Predicate Logic*. PhD thesis, Darmstadt University of Technology, 2003. (to appear in Lecture Notes in Computer Science, Springer).

[DH03a] Dau, Frithjof and Joachim Hereth Correia: *Nested concept graphs: Applications for databases*. Available at: `http://www.mathematik.tu-darmstadt.de/~dau/DauHereth03a.pdf`, 2003.

[DH03b] Dau, Frithjof and Joachim Hereth Correia: *Nested concept graphs: Mathematical foundations.* Available at: http://www.mathematik.tu-darmstadt.de/~dau/DauHereth03b.pdf, 2003.

[EGSW00] Eklund, Peter, Bernd Groh, Gerd Stumme, and Rudolf Wille: *A contextual-logic extension of toscana.* In Ganter, Bernhard and Guy W. Mineau (editors): *Conceptual Structures: Logical, Linguistic and Computational Issues. 8th International Conference on Conceptual Structures, ICCS 2000*, number 1867 in *LNAI*, pages 453–467, Darmstadt, Germany, 2000. Springer, Berlin – Heidelberg – New York.

[GE01] Groh, Bernd and Peter Eklund: *A cg-query engine based on relational power context families.* In Delugach, Harry S. and Gerd Stumme (editors): *Conceptual Structures: Broadening the Base. 9th International Conference on Conceptual Structures, ICCS 2001*, number 2120 in *LNAI*, Stanford, USA, July 2001. Springer, Berlin – Heidelberg – New York.

[Klu88] Klug, Anthony: *Equivalence of relational algebra and relational calculus query languages having aggregate functions.* Journal of the ACM, 29(3):699–717, 1988.

[Pei92] Peirce, Charles Sanders: *Reasoning and the logic of things.* In Kremer, K. L. (editor): *The Cambridge Conferences Lectures of 1898.* Harvard Univ. Press, Cambridge, 1992.

[Pre98] Prediger, Susanne: *Kontextuelle Urteilslogik mit Begriffsgraphen – Ein Beitrag zur Restrukturierung der Mathematischen Logik.* Shaker Verlag, Aachen, 1998. Dissertation, Technische Universität Darmstadt.

[PS00] Peirce, Charles Sanders and John F. Sowa: *Existential Graphs: MS 514 by Charles Sanders Peirce with commentary by John Sowa*, 1908, 2000. http://users.bestweb.net/~sowa/peirce/ms514.htm.

[Sow84] Sowa, John F.: *Conceptual structures: information processing in mind and machine.* Addison-Wesley, 1984.

[Sow92] Sowa, John F.: *Conceptual graphs summary.* In Nagle, Timothy E., Janice A. Nagle, Lauries L. Gerholz, and Peter W. Eklund (editors): *Conceptual Structures: Current Research and Practice*, pages 3–51. Ellis Horwood, 1992.

[Sow00] Sowa, John F.: *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Brooks Cole, Pacific Grove, CA, 2000.