ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

A. Formica

FINITE SATISFIABILITY OF OBJECT-ORIENTED DATABASE INTEGRITY CONSTRAINTS WITH INEQUALITY AND NULL-VALUES

R. 538 Dicembre 2000

Anna Formica – Istituto di Analisi dei Sistemi ed Informatica del CNR, viale Manzoni 30 - 00185 Roma, Italy. Email: {formica}@iasi.rm.cnr.it.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica, CNR

viale Manzoni 30, 00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.rm.cnr.it
URL: http://www.iasi.rm.cnr.it

Abstract

In this paper, a method for checking finite satisfiability of a specific class of database integrity constraints is presented. In particular, this work starts from a previous result of the author concerning a decidable, sound, and complete method for checking finite satisfiability of *θ-constraints*. θ-constraints are a sort of *path* constraints, where θ stands for one of the comparison operators $>$, $\geq$, $<$, $\leq$, $=$. In this paper such a method is extended to deal with a more expressive constraint language including the inequality operator and null-values. The proposed extension is formally characterized by providing the necessary and sufficient conditions that allow finite satisfiability of this wider class of integrity constraints to be checked.

1. Introduction

Database constraints satisfiability and, in particular, finite satisfiability are key problems in database design. Checking satisfiability of database constraints consists in verifying the existence of at least one database that is a *model* of the constraint set, i.e., the absence of contradictions within the set of integrity constraints independently of any given database. This problem differs from the constraint satisfaction problem, whose goal is to verify whether or not a *given* database is a model of, or satisfies, a set of integrity constraints.

Furthermore, database constraints have to be *finitely* satisfiable, that is, the existence of at least one *finite* model is required. In other words, database constraints have to be free of *axioms of infinity*, i.e., sets of constraints that admit *only infinite* models [7, 17]. For instance, consider the following database constraints:

- every *person* has one *age*, that is an *integer*, and one *child*, that is a *person*;
- the *age* of a *person* has to be strictly greater than the *age* of his/her *child*.

This is an axiom of infinity. In fact, every database that is a model of it is necessarily infinite.

In this regard, in [18] a decidable, sound, and complete method for checking finite satisfiability of a specific class of integrity constraints, namely the *θ-constraints*, has been presented. θ-constraints, where θ stands for one of the comparison operators $>$, $\geq$, $<$, $\leq$, and $=$, are an integral part of the database schema. They are defined by *paths*, that are dot-separated sequences of properties that allow navigation through the types of the schema.
In this paper, an extension of the method presented in [18] has been proposed. In particular, such a method has been revisited in order to check finite satisfiability of a wider set of integrity constraints containing also the inequality operator and null-values. Null-values have been extensively addressed in the literature, with several different semantics, such us, for instance, the *unknown* [14], or the *does not exist* [31, 42] meanings. In this paper null-values are addressed with the *does not exist* meaning only.

Of course, due to the richer expressive power, the extended constraint language allows the designer to describe the application domain at a finer level of detail. For instance, in the presence of null-values the first integrity constraint of the previous example can be modified as follows:

- every *person* has one *age*, that is an *integer*, and at most one *child*, that is a *person*.

In this case, the description of a person is closer to the reality, i.e., for a person there is the possibility, not the need, of having one child. Notice that the presence of null-values has an important impact on the expressive power of the language and on satisfiability. For instance, by allowing *child* to be a null-value, the above set of integrity constraints becomes finitely satisfiable.

Let us now consider the inequality operator. Also in the restrictive form of θ-constraints, the inequality constraint ($\neq$*-constraint*) allows finer details of the application domain to be described as illustrated in the following example:

- every *person* has one *father*, that is a *person*, and drives one *vehicle*, that is a *car*;
- every *car* has one *owner*, that is a *person*;
- the *owner* of the *vehicle* the *person* drives is his/her father.

- a *person* is the *owner* of the *vehicle* he/she drives.

This is a finitely satisfiable set of integrity constraints (it will be described in Subsection 3.2). However, in every model that can be defined for it, a person necessarily will coincide with his/her father. The possibility of expressing $\neq$-constraints allows us to specify the further constraint:

- a *person* does not coincide with his/her *father*.

The addition of this constraint impacts on the satisfiability of the set of statements. In fact, the set of constraints, from finitely satisfiable, becomes unsatisfiable.

In this paper an extension of the language TQL^+, on which the method presented in [18] is based, has been presented. Such a language has been referred to as TQL^+_{in} (where i and n stand for *i*nequality and *n*ull-values, respectively). In order to extend the mentioned method to database constraints including the inequality operator and null-values, the theory presented in [18] has been revisited. In particular, a decidable, sound and complete method for finite satisfiability checking of TQL^+_{in} schemas has been defined.

The paper is organized as follows. In Section 2, the syntax and the semantics of the language TQL^+_{in} are given. In Section 3, the extended method is informally presented via examples, by focusing first on null-values and then on inequality. Finally in Section 4, the characterization of the extended method is formally introduced. In particular, in that section, the key definitions that differ from the ones given in [18] are given, whereas in the Appendix all the remaining definitions are recalled for reader's convenience. In Section 5, the conclusion and future work are briefly presented. Below the related work follows.

1.1. Related work

The satisfiability checking of a set of integrity constraints has not been widely addressed in the literature. There are some results concerning this problem in different data model as, for instance, in the network data model [1], in the Entity Relationship model [32, 16], or in the Object-Oriented data model [9]. However, the data models proposed in these papers do not allow the modeling of explicit integrity constraints involving comparison operators similar to θ-constraints. Such constraints have been addressed for instance in [5], in regard to Object-Oriented database schemas. In particular, in such a paper *path relations* have been addressed that are very similar to the θ-constraints, including the $\neq$ operator and the possibility of specifying null-values. However, the results provided in that paper do not allow finite satisfiability to be checked for schemas containing both recursive definitions and path relations, that are required to express, for instance, the examples illustrated in the Introduction.

The class of integrity constraints addressed in this paper can be also seen as a restricted form of cyclic referential constraints as defined in [43] (i.e., corresponding to *inclusion dependencies*). In particular, a θ-constraint can be seen as a cyclic referential constraint where the antecedent is defined by one unary predicate, the consequent contains unary and binary predicates at most, constants are not allowed, and the variables obey to a sort of concatenation law as defined by the formal semantic of the language TQL^+_{in}. In [43] (where the $\neq$ operator and null-values are also allowed), the problem of reasoning with *implication* constraints (that are a generalization of *functional dependencies*) and *referential* constraints has been addressed. Since the *implication*

problem for functional and inclusion dependencies is undecidable, in the mentioned paper acyclic referential constraints have been addressed, and a novel characterization for the implication and referential constraints-refuting problem has been proved.

In the context of deductive databases, schema consistency checking has been widely investigated by relying on theorem prover techniques. For instance in *Sic* [6], formerly known as *Satchmo* [7, 8], a schema is a set of first order logic formulas, and schema consistency checking is performed by using semidecidable procedures. Furthermore, it is well-known that such procedures are rather inefficient in the presence of the equality operator. The problem of dealing with equality is one of the main points addressed by the method proposed in [18], where a decidable method for checking finite satisfiability of recursive schemas enriched with θ-constraints has been presented. Such a method is based on a graph-theoretic approach, where θ operators are modeled by labeled arcs. In particular, the nodes connected trough arcs labeled with the equality operator are collapsed, therefore avoiding the proliferation of equality predicates typical of the theorem prover approach. This method will be briefly recalled in the following sections in order to extend it to null-values and inequality.

Finally, it is worth recalling that the presence of ISA hierarchies in the data model, independently of any other kind of integrity constraints, may be source of contradictions in the schema, due to the well-known *inheritance conflicts* [2, 28]. The problem of checking the consistency of Object-Oriented database schemas organized according to ISA hierarchies has been analyzed in [4, 19, 20] and goes beyond the scope of this paper.

2. Formal Basis

In this section the language TQL^{+}_{in} is presented. Such a language is a fragment of TQL^{++} [33], aimed at modeling the structural aspects and the integrity constraints of an Object-Oriented database (ODB) application.

2.1. TQL^{+}_{in} Syntax

In TQL^{+}_{in}, a *schema* is defined by a set of *types*. A type describes the structure of the objects that populate the related type *extension* (also called *class*), and the integrity constraints these objects have to satisfy. For instance the first example informally illustrated in the Introduction, in TQL^{+}_{in} is expressed as follows:

Example 2.1.
$person := [name : string, age : integer, child : \{person\}]$
$\quad ic : this.age > this.child.age$

□

In TQL^{+}_{in}, a type has a label (t_term) and a *tuple* that is defined by a set of *typed properties* (tp). In the above example, only one type is present, whose type label is *person*. In a tuple, a property is identified by a label (p_term), as for instance *name* or *child* in the example above, and is associated with a type. In particular, a property can be either an *attribute* or a *relationship* if the associated type is respectively: (i) an *atomic type* (a_type), e.g., *integer* or *string*, as for instance, in the example, *name*; (ii) a t_term as, in the example, *child*, establishing an explicit

link (or association) between two types. Recursive types are allowed (in this case, *person* is a recursive type due to the self recursive relationship *child*). In TQL^+_{in}, properties may be *single-valued* or *null-valued*. With respect to single-valued properties, null-valued properties are denoted by curly braces. For instance *child*, in the example, is a null-valued property, whereas *name* and *age* are single-valued. Notice that in a tuple, multiple occurrences of the same property labels are not allowed. Types can also be organized according to a generalization hierarchy, declared by means of the **ISA** construct [10].

As already mentioned in the Introduction, TQL^+_{in} explicit integrity constraints are *θ-constraints*, where θ stands for a comparison operator, such as "$=$", "$\geq$, "$>$", and, in addition to TQL^+, also "$\neq$". In the previous example, we have a simple θ-constraint stating that the *age* of a *person* must be strictly greater than the age of his/her *child*. Notice that in a θ-constraint a left and a right hand sides are present, each of which is specified by the keyword *this* followed by a *path*. The keyword *this* refers to a single object in the extension of the type the integrity constraint is associated with. A *path* is a sequence of *p_terms*, expressed according to the dot-notation formalism, that allows navigation through types. In a *path*, the same property labels may occur more than once. For instance, the second example discussed in the Introduction in TQL^+_{in} can be represented as follows:

Example 2.2.
$person := [name : string, father : person, vehicle : car]$
$\quad ic1 : this.father = this.vehicle.owner$
$\quad ic2 : this.vehicle.owner = this$
$\quad ic3 : this.father \neq this$
$car := [owner : person, maker : string, color : string]$

□

The formal syntax of TQL^+_{in} is presented below: non-terminal symbols are enclosed between angle brackets, terminal symbols are in bold, symbols in italics represent user-defined strings, whereas symbols enclosed between curly braces followed by "*" are optional (the underscore character stands for iteration).

Definition 2.1. [Syntax of TQL^+_{in}]

⟨type⟩ ::= *t_term* **:=** ⟨type-definition⟩ {, ⟨c_expr$_1$⟩, _ ⟨c_expr$_n$⟩ }*
⟨type-definition⟩ ::= **ISA** *t_term*$_1$ _ *t_term*$_k$ { ⟨tuple⟩ }*
| ⟨tuple⟩
⟨tuple⟩ ::= [⟨tp$_1$⟩, _ ⟨tp$_m$⟩]
⟨tp⟩ ::= *p_term* **:** ⟨body⟩
| *p_term* **:** {⟨body⟩}
⟨body⟩ ::= *t_term*
| ⟨a_type⟩
⟨a_type⟩ ::= **integer**
| **real** | **boolean** | **string**
⟨c_expr⟩ ::= *label*: **this**.⟨path⟩ ⟨ θ ⟩ **this**{.⟨path⟩}
⟨path⟩ ::= *p_term*$_1$. _ *p_term*$_h$
⟨ θ ⟩ ::= $\leq$ | $<$ | $>$ | $\geq$ | $=$ | $\neq$

□

The previous examples are both TQL^+_{in} *schemas*. The notion of a TQL^+_{in} schema coincides with the one given in [18], that is informally recalled below. Essentially, it concerns the uniqueness of type labels, the absence of dangling type labels, the acyclicity of inheritance hierarchies and, furthermore, the definedness of integrity constraints.

Definition 2.2. [TQL^+_{in} **schema**] A finite set of types is a TQL^+_{in} *schema* (*schema*, for short) iff:

- every type label is uniquely defined (i.e., the same *t_term* is not associated with more than one type-definition);
- there are no dangling type labels (i.e., every *t_term* declared in the schema is defined);
- inheritance is acyclic (i.e., a type has not itself as supertype, up in the hierarchy);
- for each type τ and each associated integrity constraint, every property of a *path* has to be present in the tuple of the traversed type. Furthermore, in the case of equality θ-constraints, the two *paths* of each integrity constraint have to lead to the same type.

□

Notice that integrity constraints of the form:
$ic : this.child.age > this.child$
for instance associated with *person* of the Example 2.1 would be rejected at a pre-processing stage, by using a type-checker.

Structural inheritance hierarchies in OBD schemas have been extensively investigated, see for instance [19, 20, 4]. Therefore, in this paper, types are supposed to have all their typed properties explicitly given, and types defined with the **ISA** construct will be not addressed.

2.2. Semantics of TQL^+_{in}

In this section the formal semantics of TQL^+_{in} is given. It is inspired by the *descriptive* semantics presented in [36]. Notice that, with respect to TQL^+, below the point related to null-valued properties and, in the constraint extension, the $\neq$ operator have been added.

Definition 2.3. **[Extension function]** Let $\mathcal{D}$ be a (possibly infinite) set of oids [26] representing a given state of the Application Domain, T the set of TQL^+_{in} sentences, and P ($\subset T$) the set of *p_terms*. Consider a function:
$\mathcal{E} : T \rightarrow \wp(\mathcal{D})$
where $\wp$ is the powerset, and a function:
$\mathcal{P} : P \rightarrow \wp(\mathcal{D} \times \mathcal{D})$.
Then, $\mathcal{E}$ is an *extension function* over $\mathcal{D}$ with respect to the type:
t_term**:=** *type-definition*, $c\_expr_1, ..., c\_expr_n$
iff the values of $\mathcal{E}$ on *type-definition* and c_expr_j, j = 1...n, are constructed starting from the values of their components as follows.

Given a $path = p\_term_1 ... p\_term_n$, let $S_{path,x}$ be defined as follows:

$S_{path,x} = \{x\}$ if $n = 0$;
$S_{path,x} = \{y \in \mathcal{D} \mid < x, y > \in \mathcal{P}(p\_term_1)\}$ if $n = 1$;
$S_{path,x} = \{y \in \mathcal{D} \mid \exists (y_1, ..., y_{n-1}) : < x, y_1 > \in \mathcal{P}(p\_term_1), < y_1, y_2 > \in \mathcal{P}(p\_term_2),$
$< y_{n-1}, y > \in \mathcal{P}(p\_term_n)\}$ if $n \geq 2$.

8.

Type Extension:

- $\mathcal{E}(t\_term) \subseteq \mathcal{D}$
- $\mathcal{E}(type\text{-}definition) = \mathcal{E}\ ([tp, ..., tp]) = \bigcap_j \mathcal{E}([tp_j])$
- $\mathcal{E}(a\_type) =$
  - $\mathcal{E}(integer) = \mathbf{Z} \cap \mathcal{D}$
  - $\mathcal{E}(real) = \mathbf{R} \cap \mathcal{D}$
  - $\mathcal{E}(boolean) = \{true, false\} \cap \mathcal{D}$
  - $\mathcal{E}(string) = \mathbf{S} \cap \mathcal{D}$ (where $\mathbf{S}$ is the set of all the possible strings)
- $\mathcal{E}([tp]) = \mathcal{E}([p\_term{:}body]) =$
  $\{x \in \mathcal{D} \mid \|S_{p\_term,x}\| = 1, \forall\ y \in S_{p\_term,x} \Rightarrow y \in \mathcal{E}(body)\}$
- $\mathcal{E}([tp]) = \mathcal{E}([p\_term{:}\{body\}]) =$
  $\{x \in \mathcal{D} \mid \|S_{p\_term,x}\| \leq 1, \forall\ y \in S_{p\_term,x} \Rightarrow y \in \mathcal{E}(body)\}$

where $\|S_{p\_term,x}\|$ stands for the cardinality of the set $S_{p\_term,x}$.

Integrity Constraint Extension:

- $\mathcal{E}(c\_expr) = \mathcal{E}(label : this.path_i\ \theta\ this.path_j) =$
  $\{x \in \mathcal{D} \mid \forall\ y, z,\ y \in S_{path_i,x}, z \in S_{path_j,x} \Rightarrow y\ \theta\ z\}$
  where, according to the syntax, θ is one of the comparison operators: $>$, $\geq$, $<$, $\leq$, $=$, $\neq$.

□

Below, the notions of *interpretation*, *model*, *satisfiability*, and *finite satisfiability* of a TQL^+_{in} schema (that coincide with the ones of TQL^+), are recalled for reader's convenience.

Definition 2.4. [Interpretation of a TQL^+_{in} schema] An *interpretation* of a TQL^+_{in} schema is a triple $\mathcal{I} =< \mathcal{D}, \mathcal{E}, \mathcal{P} >$ where $\mathcal{D}$ is a set representing the Application Domain, $\mathcal{P}$ is a function as defined above, and $\mathcal{E}$ is an extension function over $\mathcal{D}$ with respect to *each* type of the schema. □

Definition 2.5. [Model of a TQL^+_{in} schema] A *model* of a TQL^+_{in} schema is an interpretation $\mathcal{I} =< \mathcal{D}, \mathcal{E}, \mathcal{P} >$ such that, for *each* type:

t_term **:=** $type\text{-}definition$, $c\_expr_1, ..., c\_expr_n$

of the schema, we have:

$\mathcal{E}(t\_term) \subseteq \mathcal{E}(type\text{-}definition) \cap (\bigcap_j \mathcal{E}(c\_expr_j))$. □

Definition 2.6. [Satisfiable TQL^+_{in} schema] A TQL^+_{in} schema is *satisfiable* iff there exists at least one *non-empty* model, i.e., one model $\mathcal{I} =< \mathcal{D}, \mathcal{E}, \mathcal{P} >$ such that for *each* t_term of the schema, we have:

$\mathcal{E}(t\_term) \neq \emptyset$. □

Definition 2.7. [Finitely Satisfiable TQL^+_{in} schema] A TQL^+_{in} schema is *finitely satisfiable* iff there exists at least one non-empty model that is $finite$, i.e., one model $\mathcal{I} =< \mathcal{D}, \mathcal{E}, \mathcal{P} >$ such that $\mathcal{D}$ is finite. □

3. The Extended Finite Satisfiability Checking Method

In this section, the method for checking finite satisfiability of a TQL^+ schema presented in [18] is informally recalled, and its extension to the language TQL^+_{in} is introduced via the examples informally discussed in the Introduction. In particular, in the first subsection the extension concerning null-values will be addressed, whereas in the second one, the extension related to the inequality operator will be illustrated. Before introducing the examples, below an informal description of the graph on which the method is based is briefly given.

The approach proposed in [18] is based on the construction of a graph, one for each type of the schema. Such a graph is referred to as $\mathcal{F}(G^{eq}_{Sat})$ (where Sat stands for satisfiability, and eq stands for equality). In particular, $\mathcal{F}(G^{eq}_{Sat})$ is a *schema* graph, i.e., it is a graph having the nodes labeled with t_terms or atomic types, and the arcs labeled with properties or comparison operators. Furthermore, in a schema graph different nodes with the same type label may be present and, for each node, outgoing arcs with the same property label and different destination nodes are not allowed. Roughly, the main idea behind the $\mathcal{F}(G^{eq}_{Sat})$ graph is that each node of the graph stands for an *instance* (i.e., an oid or a value) of the type labeling that node, and each arc establishes an association between type instances, according to the structure of the types, or a relation according to the integrity constraints associated with the types. The $\mathcal{F}(G^{eq}_{Sat})$ graph associated with a type is in general not connected. In particular, in such a graph, each node labeled with a t_term is the origin of a pair of paths, one for each integrity constraint associated with the type labeled with the t_term. These paths are labeled according to the sequences of properties defining the right and the left hand sides (i.e., the *paths*) of the related integrity constraint. Notice that the presence of equality constraints plays a special role. In fact, since the nodes connected through arcs labeled with the equality operator denote the same oids, they are collapsed into a single node.

3.1. Null-valued properties

Consider the following schema, slightly different with respect to the one of the Example 2.1:

Example 3.1.
$person := [name : string, age : integer, child : person]$
$\quad ic : this.age > this.child.age$

□

In this case the *child* property is single-valued, that is, we are addressing the language component that does not include null-values. As already mentioned in the Introduction, in this case we have an axiom of infinity: only databases with an infinite number of *person* oids can satisfy it. According to [18], the $\mathcal{F}(G^{eq}_{Sat})$ graph associated with this type is the one shown in Figure 1.

In this case, the $\mathcal{F}(G^{eq}_{Sat}(person))$ graph is not connected. In both the connected components we can see that starting from each of the nodes labeled with *person*, at least two paths originate: one labeled with the *age* property, corresponding to the left hand side of the integrity constraint *ic*, the other one labeled with the sequence *child.age* corresponding to the right hand side of *ic*. Notice that, for each arc, the destination node is labeled with the type of the property labeling the arc, as defined in the schema (for instance, the destination node of the arc labeled with *child* is labeled with *person*, as defined by the tuple of *person*).

Furthermore, an arc between the final nodes of the paths modeling the integrity constraint is present, labeled with the related comparison operator (we assumed that arcs labeled with

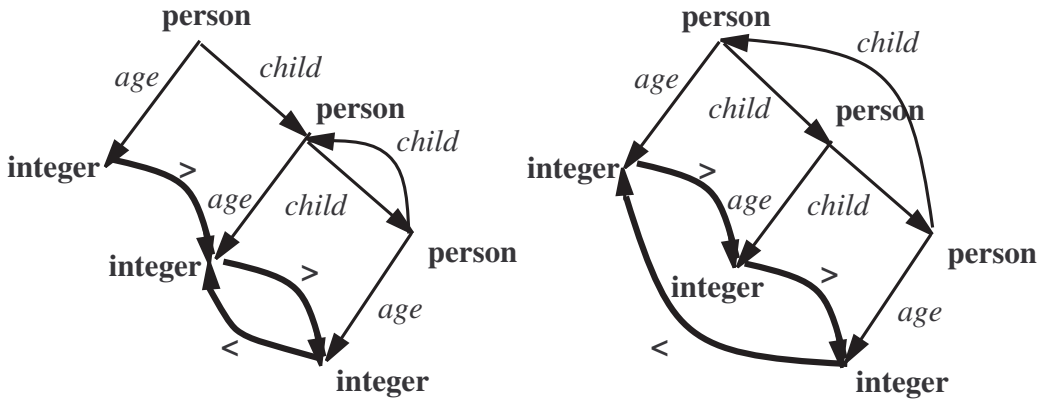

Figure 1: $\mathcal{F}(G^{eq}_{Sat}(person))$ graph - Example 3.1

comparison operators are directed according to the ">" operator, but the opposite choice could be adopted as well).

According to the theory presented in [18], the above schema is finitely satisfiable if and only if in the $\mathcal{F}(G^{eq}_{Sat}(person))$ graph there exists at least one connected component free of cycles labeled with strict comparison operators (that are referred to as *monotonic* cycles). Therefore, in this case, since both the connected components of $\mathcal{F}(G^{eq}_{Sat}(person))$ contain such cycles, the schema is not finitely satisfiable.[1]

Suppose now to deal with the language TQL^{+}_{in}, i.e, to have the possibility of expressing null-values, and consider the Example 2.1 where the *child* property is null-valued. The extension of the mentioned method to null-values is quite immediate. In fact, it is possible to deal with them by simply ignoring the paths of the graph in correspondence to the null-valued properties. More in general, in the construction of the graph, only the integrity constraints do not containing null-valued properties have to be considered. Therefore, in the example, rather than the $\mathcal{F}(G^{eq}_{Sat}(person))$ graph of Figure 1, one single node labeled with *person* will be considered. As a result, the schema is finitely satisfiable, trivially.

3.2. The inequality operator

The extension of the method to the inequality θ-constraints is less immediate with respect to the previous one. Consider the second example mentioned in the Introduction, without the inequality constraint.

Example 3.2.
$person := [name : string, father : person, vehicle : car]$
$\quad ic1 : this.father = this.vehicle.owner$
$\quad ic2 : this.vehicle.owner = this$
$car := [owner : person, maker : string, color : string]$

□

[1] Notice that the presence of three nodes labeled with *person* is due to a sort of "expansion" mechanism on which the construction of the $\mathcal{F}(G^{eq}_{Sat})$ graph is based. See the Appendix, where the basic definitions given in [18] have been recalled for reader's convenience.

One of the key points of the $\mathcal{F}(G^{eq}_{Sat})$ graph is that the arcs labeled with equality operators do not appear. In fact, since they connect nodes that have to coincide (according to the formal semantics of the language), they are always removed and the connected nodes collapsed. In the above example, two integrity constraints with the equality operator are present. It is easy to see that after the removal of the equality arcs, the resulting graph is the one shown in Figure 2. Since in such a graph there are no monotonic cycles, this schema is finitely satisfiable.

As already mentioned in the Introduction, any model that satisfies this schema necessarily requires that a *person* coincides with his/her $father$. Therefore, the addition of the further constraint $ic3$ defined in the Example 2.2 makes the schema unsatisfiable. In the extended method, inequality constraints are initially treated similarly to the other θ-constraints. Therefore, rather than the graph of Figure 2, a graph with an additional arc labeled with the $\neq$ operator will be considered, as shown in Figure 3. Such a graph is referred to as $\mathcal{F}(G^{eq,in}_{Sat})$.

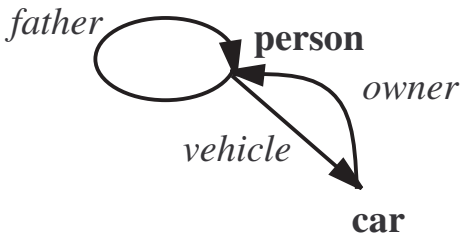

Figure 2: $\mathcal{F}(G^{eq}_{Sat}(person))$ graph - Example 3.2

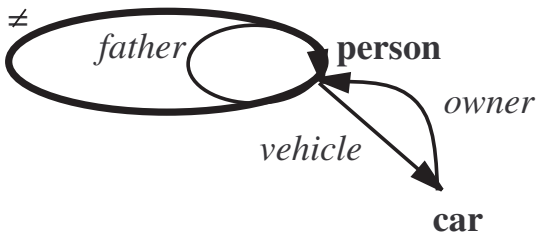

Figure 3: $\mathcal{F}(G^{eq,in}_{Sat}(person))$ graph - Example 3.2

In the extended method, a loop labeled with the $\neq$ operator ($\neq$-loop) in all the connected components of the $\mathcal{F}(G^{eq,in}_{Sat})$ (in this case it is only one) is a sufficient condition to derive that the schema is not finitely satisfiable.

In the general case, the method requires a further step. It concerns the removal from the graph of all the $\neq$-arcs labeling cycles that are not loops. This is due to the fact that, in the absence of the $\neq$ operator in the data model, a cycle of θ-arcs labeled with strict comparison operators (once the equality arcs have been removed) denotes a contradiction, whereas in the presence of $\neq$-arcs, such a cycle does not denote any contradiction. Just to show an example about this, consider the following schema:

12.

Example 3.3.
$person := [salary : integer, vehicle : car]$,
$\quad ic1 : this.salary > this.vehicle.price$
$car := [price : integer, owner : person]$,
$\quad ic2 : this.price > this.owner.salary$

□

This is an axiom of infinity. In fact, it requires an infinite chain of *person* and *car* oids to be defined in the database. The $\mathcal{F}(G^{eq}_{Sat}(person))$ graph associated with *person* in this case is shown in Figure 4.

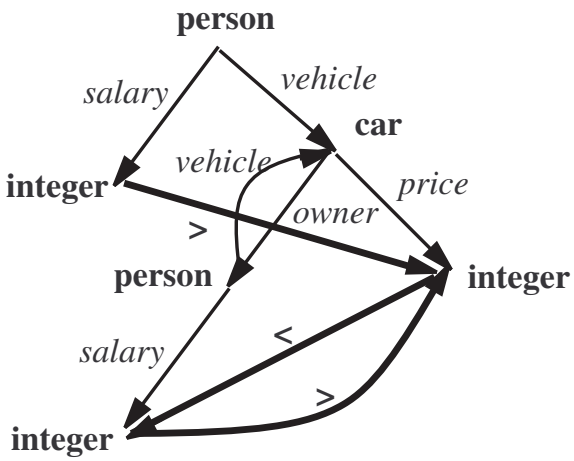

Figure 4: $\mathcal{F}(G^{eq}_{Sat}(person))$ graph - Example 3.3

Such a schema is not finitely satisfiable due to the presence of a monotonic cycle in the $\mathcal{F}(G^{eq}_{Sat}(person))$ graph (we have considered *person*, but the type *car* could be considered as well). Now, suppose to modify such a schema by simply replacing the $>$ operator of the integrity constraint $ic2$ with the $\neq$ operator, as follows:

Example 3.4.
$person := [salary : integer, vehicle : car]$,
$\quad ic1 : this.salary > this.vehicle.price$
$car := [price : integer, owner : person]$,
$\quad ic2 : this.price \neq this.owner.salary$

□

In this case, the schema is finitely satisfiable. The $\mathcal{F}(G^{eq,in}_{Sat}(person))$ graph is shown in Figure 5, where the $\neq$-arc has been removed and no monotonic cycle is present.

In the next section, the method proposed in [18] will be revisited in order to deal with null-valued properties and inequality θ-constraints.

4. Characterization of Finitely Satisfiable Schemas

In this section, the extension of the method is formally addressed. Below, we start by introducing the notion of a *non-null* integrity constraint. Such a notion will be used in order to consider,

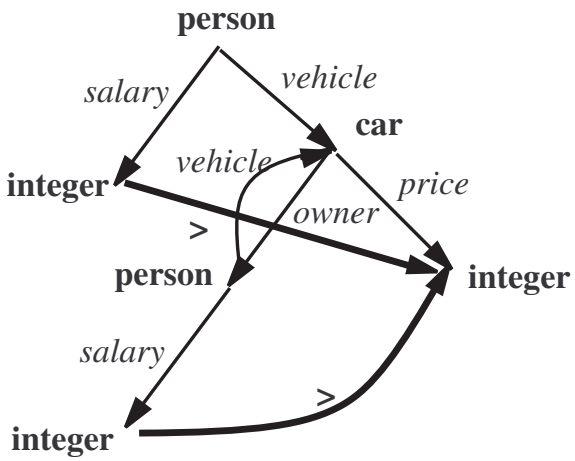

Figure 5: $\mathcal{F}(G^{eq,in}_{Sat}(person)$ graph - Example 3.4

in the construction of the graph, only the integrity constraints do not containing null-valued properties.

Notationally, given a type τ, $I(\tau)$ denotes the set of explicit integrity constraints of τ.

Definition 4.1. [Non-null integrity constraint] Given a schema Σ, a type τ of Σ such that $I(\tau) \neq \emptyset$, and an integrity constraint $ic \in I(\tau)$. Then, ic is *non-null* iff all the properties defining it are not null-valued properties. □

Staring from this notion, all the formal definitions that are at the basis of the construction of the $\mathcal{F}(G^{eq}_{Sat})$ graph can be reformulated very similarly to the definitions given in [18]. For this reason, and also in order to better address the contribution of this paper with respect to [18], the revisitation of the construction of the $\mathcal{F}(G^{eq}_{Sat})$ graph is given in the Appendix (where essentially only the Definition A.4 has been modified in order to deal with null-valued properties).

Therefore, below we assume that the reader is familiar with the definitions of a schema graph (see Definition A.2), and a $\mathcal{F}(G^{eq}_{Sat})$ graph (see Definition A.15). Furthermore, it is worth recalling below the notion of *monotonic* cycle, that is the same of the paper [18], although now strict comparison operators include also $\neq$.

Definition 4.2. [Monotonic cycle] Given a schema Σ, a *monotonic* cycle of a schema graph is a loop or a cycle of θ-*arcs* labeled with at least one strict comparison operator (i.e., $\neq$, $>$, or $<$). □

Now, we are able to give the definition of a $\mathcal{F}(G^{eq,in}_{Sat})$ graph.

Definition 4.3. [The $\mathcal{F}(G^{eq,in}_{Sat})$ graph] Given a schema Σ, the $\mathcal{F}(G^{eq,in}_{Sat})$ graph if the $\mathcal{F}(G^{eq}_{Sat})$ graph where all the $\neq$-arcs labeling cycles that are not loops have been removed. □

This definition allows an immediate extension of the formal characterization of finitely satisfiable schemas in the presence of inequality and null-values.

Theorem 4.1. [Characterization of finitely satisfiable schemas] A schema Σ is *finitely satisfiable* iff for each type τ of Σ such that $I(\tau) \neq \emptyset$ the schema graph $\mathcal{F}(G^{eq,in}_{Sat}(\tau))$ has at least one connected component free of monotonic cycles.

Proof. It is similar to the one given in [18], taking into account the following points.

Given a type τ, the $\mathcal{F}(G^{eq}_{Sat}(\tau))$ graph is constructed by recursively expanding some of the paths of a tree modeling the integrity constraints of the type τ, that is referred to as $T_{Path}(\tau)$ tree (see in the Appendix, the Definition A.8). In particular, in [18] it has been proved that any further expansion of such a tree is not relevant to the presence of monotonic cycle in the connected components of the $\mathcal{F}(G^{eq}_{Sat}(\tau))$. This statement also holds in the case of the $\mathcal{F}(G^{eq,in}_{Sat}(\tau))$ graph because its construction derives from an additional step performed on the $\mathcal{F}(G^{eq}_{Sat}(\tau))$.

Furthermore, any cycle of θ-arcs labeled with $>$, $\geq$, and at least one $\neq$ operator, does not denote any contradiction because it can always be "instantiated" with a set of values satisfying the related inequalities. □

With regard to the complexity of the method, the proposed extension does not affect the analysis performed in [18]. In fact, the presence of null-values simply inhibits the construction of some of the paths defined in the $\mathcal{F}(G_{Sat})$ graph. Furthermore, with respect to the previous approach, only one further visit of the graph is required, in order to remove the $\neq$-arcs that are not loops. Therefore, also the extended method has an exponential worst case. However, we recall that such a worst case rarely occurs when dealing with the application domains usually addressed in the database field.

5. Conclusion and Future work

In this paper a method for checking finite satisfiability of a specific class of database constraints has been presented. In future work, possible extensions of the expressive power of the language will be analyzed, for instance including constraint expressions comparing attribute values with constants. However, since the more expressive the language the harder the reasoning with the language expressions, a deep preliminary analysis about the trade-off between the expressive power of the language and the possibility of reasoning with it is required. Such an activity, i.e., the identification of fragments of formal logic that allow decidable reasoning methods to be defined, is one of the main challenges of conceptual modeling.

A. Appendix

In this section, the notions that are on the basis of the method presented in [18] are revisited according to the proposed extension TQL^{+}_{in}. In the remainder of the paper, for sake of simplicity, a *type* will indicate either a type label, i.e., a *t_term*, or an atomic type.

Below, we start with the notion of the $\mathcal{T}$ function.

Definition A.1. [The $\mathcal{T}$ function] Given a schema Σ, the $\mathcal{T}$ function is defined on a *t_term* τ followed by a sequence of n (≥ 1) *p_terms* of Σ, as follows:

- $\mathcal{T}(\tau.p_1...p_n) = \sigma.p_2...p_n$ if τ has the typed property "$p_1 : \sigma$", i.e., $\tau := [..., p_1 : \sigma, ...]$

- $\mathcal{T}(\tau.p_1...p_n)$ is undefined otherwise.

For $1 \leq k \leq n$, $\mathcal{T}^k$ is the composition of the $\mathcal{T}$ function k times, i.e.:
$\mathcal{T}^k(\tau.p_1...p_n) = \mathcal{T}(\mathcal{T}(...(\mathcal{T}(\tau.p_1...p_n))...))$ □

For instance, in Example 2.1:
$\mathcal{T}(person.child.age) = person.age$
and:
$\mathcal{T}^2(person.child.age) = integer$.

Definition A.2. **[The Schema graph]** Given a schema Σ, a *schema graph associated with* Σ (*schema graph*, for short) is a directed labeled graph whose sets N and A of nodes and arcs, respectively, are labeled as follows:

- a node $n \in N$ is labeled with a set of types of Σ, that is indicated as $e(n)$. In general, different nodes may have the same label;
- an arc $a \in A$ is an ordered pair of nodes, $n, m \in N$, labeled with a:
  1. *p_term*, say p, defined in Σ. It is indicated as:
     $< n, m >_p$,
     and is referred to as a *property-arc*;
  2. θ operator defined in Σ. It is indicated as:
     $< n, m >_\theta$,
     and is referred to as a θ-*arc*;
- for each node $n \in N$, if there exist two *property-arcs* such that:
  $< n, m >_p$ and $< n, q >_p$
  then: $m \equiv q$,
  i.e., for each node, the outgoing arcs with the same property labels have the same destination nodes.

□

Based on the definition of a schema graph, the notion of a *schema tree* is given below.

Definition A.3. **[The Schema tree]** Given a schema Σ, a *schema tree* is a schema graph that is a tree, directed from the root to the leaves, whose arcs are all *property-arcs*. □

In order to formally define the notion of G_{Sat} graph tree, below the $left_r$ and $right_r$ sets are presented, that are related to the modeling of the left and the right hand sides of an integrity constraint, respectively.

Given a schema Σ and a type τ of Σ, we recall that $I(\tau)$ indicates the set of θ-constraints associated with τ in Σ.

Below, the definition of $left_r$ and $right_r$ sets is given. With respect to the one given in [18], here only the non-null integrity constraints are considered (see Definition 4.1).

Definition A.4. **[The $left_r$ and $right_r$ sets]** Consider a schema Σ, a type τ of Σ such that $I(\tau) \neq \emptyset$, and a non-null integrity constraint $ic \in I(\tau)$, defined as follows:

$ic = label : this.p_1...p_n\ \theta\ this.q_1...q_v$

Then, the $left_r(\tau, ic)$ and $right_r(\tau, ic)$ are two sets of *property-arcs* of a schema graph originating from the node r and defined, respectively, as follows:

$left_r(\tau, ic) = \{< r, r_2 >_{p_1}, < r_2, r_3 >_{p_2}, ..., < r_n, r_{n+1} >_{p_n}\}$

$right_r(\tau, ic) = \{< r, s_2 >_{q_1}, < s_2, s_3 >_{q_2}, ..., < s_v, s_{v+1} >_{q_v}\}$

where:

- $e(r) = \{\tau\}$;
- $e(r_{k+1}) = \{\mathcal{T}^k(\tau.p_1...p_k)\}$, for k = 1...n;
- $e(s_{h+1}) = \{\mathcal{T}^h(\tau.q_1...q_h)\}$, for h = 1...v.

(In the case of $v = 0$, $right_r(\tau, ic)$ is formed by the r node only.) □

The T_{Path} tree is introduced below. It models, essentially, the component of the G_{Sat} graph containing only *property-arcs*.

Definition A.5. **[The T_{Path} tree]** Given a schema Σ, consider a type τ of Σ whose associated set of integrity constraints $I(\tau)$ is non-empty.

Then, the T_{Path} tree of root r associated with τ, indicated as $T_{Path,r}(\tau)$, is the schema tree whose set of arcs, say A_r, is defined as follows:

$A_r = \bigcup_{ic \in I(\tau)} (left_r(\tau, ic) \cup right_r(\tau, ic))$

and whose set of nodes is completely characterized by the set A_r.

Notice that, since $T_{Path,r}(\tau)$ is a schema tree, if there exist two *property-arcs* such that:

$< n, m_1 >_p$ and $< n, m_2 >_p$

then the nodes m_1 and m_2 are replaced by a single node m, such that:

$e(m)$=$e(m_1) \cup e(m_2)$.

The short form $T_{Path}(\tau)$ indicates the $T_{Path,r}(\tau)$ tree, where r is any root. □

Notice that, since we have a schema, in the above definition the labels of the nodes m_1 and m_2 are singletons that coincide.

Definition A.6. **[The $\bigsqcup$ operator]** Given a schema Σ, let $\mathcal{S}$ be the forest of all its possible schema trees. Then, the $\bigsqcup$ operation between two schema trees T,$T' \in \mathcal{S}$, $T = (N, A)$ and $T' = (N', A')$, returns the forest, say T'', of schema trees defined as follows:

$T'' = T \bigsqcup T' = (N'', A'')$

where:

$N'' = N \cup N'$, $A'' = A \cup A'$. □

Definition A.7. **[The Expand function]** Consider a schema Σ, and the forest $\mathcal{S}$ of all its possible schema trees. Given a schema tree $T = (N, A)$, let N^- be the set:

$N^- = \{m \in N \mid e(m) = \{\gamma\}, \gamma$ is a *t_term* of Σ, and $I(\gamma) \neq \emptyset\ \}$.

Then, the *Expand* function is defined as follows:

$Expand : \mathcal{S} \rightarrow \mathcal{S}$

and, when applied to T, returns the schema tree:

$Expand(T) = \bigsqcup_{n \in N^-} (T_{Path,n}(\delta)) \bigsqcup T$

where $e(n) = \{\delta\}$.

Notice that, since the *Expand* function returns a schema tree, if there exist two *property-arcs* such that:

$< n, m_1 >_p$ and $< n, m_2 >_p$

then the nodes m_1 and m_2 are replaced by a single node m, such that:

$e(m)=e(m_1) \cup e(m_2)$. □

Definition A.8. **[The T^{rec}_{Path} tree]** Given a schema Σ and a type τ of Σ such that $I(\tau) \neq \emptyset$, the $T^{rec}_{Path}(\tau)$ tree is a schema tree, that is a supergraph of $T_{Path}(\tau)$, defined as follows. Let $Expand^h(T)$ be the composition of the *Expand* function h times. Furthermore, assume that:

$T_{Path}(\tau) = Expand^0(T_{Path}(\tau))$

and let N^h be the set of the nodes of the $Expand^h(T_{Path}(\tau))$ tree, for $h \geq 0$. Then:

$T^{rec}_{Path}(\tau) = Expand^{k+1}(T_{Path}(\tau))$

where $k \geq 0$ is the smallest nonnegative integer such that the following condition holds:
$\forall\ n \in (N^{k+1} \setminus N^k)$ whose label does not contain atomic types, if q is the path of the tree:

$Expand^{k+1}(T_{Path}(\tau))$

connecting the root to the node n, and q^- is the subpath of q belonging to the tree:

$Expand^k(T_{Path}(\tau))$

then, there exists a node m in the path q^- such that $e(n) = e(m)$ and, furthermore, m has at least all the outgoing *property-arcs* of n. □

Proposition A.1. **[The T^{rec}_{Path} tree size]** Given a schema Σ, for any type τ of Σ such that $I(\tau) \neq \emptyset$, the $T^{rec}_{Path}(\tau)$ tree is defined and has a finite number of nodes.

Proof. See [18]. □

The Θ_r set, introduced below, identifies the set of *θ-arcs* that need to be added to the T^{rec}_{Path} tree to obtain the G_{Sat} graph.

Definition A.9. **[The Θ_r set]** Given a schema Σ, a type τ of Σ such that $I(\tau) \neq \emptyset$, and a schema graph G. Consider an integrity constraint associated with τ, say ic, defined as follows:

$ic = label : this.p_1...p_n\ \theta\ this.q_1...q_v$

If in G there exist the paths:

$left_r(\tau, ic) = \{< r, r_2 >_{p_1}, < r_2, r_3 >_{p_2}, ..., < r_n, r_{n+1} >_{p_n}\}$

$right_r(\tau, ic) = \{< r, s_2 >_{q_1}, < s_2, s_3 >_{q_2}, ..., < s_v, s_{v+1} >_{q_v}\}$

defined according to Def.A.4, then:

$\Theta_r(\tau, ic) = \{< r_{n+1}, s_{v+1} >_\theta\}$ if θ is $>$, $\geq$, $=$;

$\Theta_r(\tau, ic) = \{< s_{v+1}, r_{n+1} >_\theta\}$ if θ is $<$, $\leq$.

In all the other cases:

$\Theta_r(\tau, ic) = \emptyset$. □

Definition A.10. **[The G_{Sat} graph]** Consider a type τ of a schema Σ such that $I(\tau) \neq \emptyset$, and the associated tree $T^{rec}_{Path}(\tau) = (N, A)$. Let N^- be the set defined as in Def.A.7, i.e.:

$N^- = \{n \in N \mid e(n) = \{\gamma\}, \gamma$ is a *t_term* of Σ, and $I(\gamma) \neq \emptyset\ \}$.

Then, $G_{Sat}(\tau) = (N_{Sat}, A_{Sat})$ is the schema graph defined as follows:

- $N_{Sat} = N$
- $A_{Sat} = \bigcup_{m \in N^-} \bigcup_{ic \in I(\delta)} (\Theta_m(\delta, ic)) \cup A$

where $e(m) = \{\delta\}$.

The root of the tree $T^{rec}_{Path}(\tau)$ will be referred to as the *owner* of the $G_{Sat}(\tau)$ graph. □

The following relation, namely the *Collapse*, is now introduced in order to present the G^{eq}_{Sat} graph.

Definition A.11. [The Collapse relation] Given a schema Σ, let $\mathcal{G}$ be the set of all its possible schema graphs. Then, let *Collapse* be the relation:

$Collapse : \mathcal{G} \rightarrow \mathcal{G}$

such that, when applied to a schema graph $G \in \mathcal{G}$, returns a schema graph $G^{-} \in \mathcal{G}$, defined as follows.

Let $< n_i, n_j >_\theta$ be a *θ-arc* of G where θ is the "=" operator. Then, in G^{-} $< n_i, n_j >_\theta$ is removed, and the nodes n_i and n_j are replaced by a node n_k such that:

$e(n_k) = e(n_j) \cup e(n_i)$.

Notice that, since G^{-} must be a schema graph, in the case of outgoing *property-arcs* with the same labels, the same of Def.A.7 is applied. □

Definition A.12. [The G^{eq}_{Sat} graph] Given a type τ of a schema Σ, $I(\tau) \neq \emptyset$, consider the $G_{Sat}(\tau)$ graph. Then, $G^{eq}_{Sat}(\tau)$ is a schema graph of the same owner of $G_{Sat}(\tau)$, that is the *least fixed point* (lfp) of the *Collapse* applied to the $G_{Sat}(\tau)$ graph, i.e.:

$G^{eq}_{Sat}(\tau) = lfp(Collapse(G_{Sat}(\tau)))$ □

Notice that, in general, G^{eq}_{Sat} is a multi-graph, i.e., a pair of nodes may be connected through more arcs (differently labeled). Furthermore, since the *Collapse* is applied to a G_{Sat} graph, the labels of the nodes to be collapsed are singletons that coincide (see Def.2.2 of a TQL^{+} schema).

Proposition A.2. [The Collapse lfp] The *Collapse* has a lfp.

Proof. See [18]. □

Definition A.13. [Pairs of equivalent paths] Given a schema Σ, consider a pair of paths, q_1,q_2 of a schema graph, defined as follows:

$q_1 = \{< r_1, r_2 >_{p_1}, < r_2, r_3 >_{p_2}, ..., < r_m, r_{m+1} >_{p_m}\}$

$q_2 = \{< r'_1, r'_2 >_{p'_1}, < r'_2, r'_3 >_{p'_2}, ..., < r'_{m'}, r'_{m'+1} >_{p'_{m'}}\}$

where each p_h, p'_k, $h = 1...m$, $k = 1...m'$, is a property label or a θ operator. Then the pair of paths s_1,s_2:

$s_1 = \{< g_1, g_2 >_{l_1}, < g_2, g_3 >_{l_2}, ..., < g_n, g_{n+1} >_{l_n}\}$

$s_2 = \{< g'_1, g'_2 >_{l'_1}, < g'_2, g'_3 >_{l'_2}, ..., < g'_{n'}, g'_{n'+1} >_{l'_{n'}}\}$

(where, again, each l_q, l'_v, $q = 1...n$, $v = 1...n'$, is a property label or a θ operator) is *equivalent* to the pair q_1,q_2 iff:

- $m = n$ and $m' = n'$;
- $p_h = l_h$, for $h = 1...m$, and $p'_k = l'_k$, for $k = 1...m'$;
- $e(r_h) = e(g_h)$, for $h = 1...m+1$, and $e(r'_k) = e(g'_k)$, for $k = 1...m'+1$
- if in q_1,q_2 there exist, respectively, two nodes r_i,r'_j, $1 \leq i \leq m+1$, $1 \leq j \leq m'+1$, such that $r_i \equiv r'_j$ (i.e., r_i and r'_j coincide) then, in s_1,s_2: $g_i \equiv g'_j$. □

Definition A.14. [Induced owner] Given a schema Σ, consider a schema graph $G = (N, A)$ and a node $n \in N$. Then, n is an *induced owner* in G iff for each pair of paths starting from the owner of a $G^{eq}_{Sat}(\gamma)$ graph, where $\gamma \in e(n)$ and $I(\gamma) \neq \emptyset$, there exists an equivalent pair of paths starting from n in G. □

The notion of a $\mathcal{F}(G)$ graph can now be formally given.

Definition A.15. [The $\mathcal{F}(G)$ graph] Given a schema Σ, let $G{=}(N, A)$ be a schema graph. Then, $\mathcal{F}(G)$ is the schema graph whose connected components, say $G_k = (N_k, A_k)$, $k = 1...s$, verify the following conditions:

- $N = N_k$,
- $A \subseteq A_k$,
- $\forall\ n \in N$, n is an induced owner in G_k. □

Proposition A.3. [The $\mathcal{F}(G^{eq}_{Sat})$ graph] Given a schema Σ, for any type τ of Σ, $I(\tau) \neq \emptyset$, the graph $\mathcal{F}(G^{eq}_{Sat}(\tau))$ has at least one (non-empty) connected component.

Proof. See [18]. □

Notice that in the case of non-recursive schemas, $\mathcal{F}(G^{eq}_{Sat})$ coincides with G^{eq}_{Sat}.

References

[1] P.Atzeni, D.S.Parker Jr; *Formal properties of net-based knowledge representation schemes*; Data & Knowledge Engineering 3 (1988), pp. 137-147, 1988.

[2] J.Banerjee, H.Chou, J.F.Garza, W.Kim, D.Woelk, N.Ballou, H.J.Kim; *Data Model Issues for Object-Oriented Applications*; in "Readings in Object-Oriented Database Systems", S.B.Zdonik and D.Maier (Eds.), pp.197-208, Morgan Kaufmann; San Mateo, CA, 1990.

[3] C.Beeri; *A formal approach to object-oriented databases*; Data & Knowledge Engineering 5; 353-382; North-Holland, 1990.

[4] C.Beeri, A. Formica, M. Missikoff; *Inheritance Hierarchy Design in Object-Oriented Databases*. Data & Knowledge Engineering (DKE), Vol.30, No.3, pp.191-216, July 1999.

[5] D.Beneventano, S.Bergamaschi, S.Lodi, C.Sartori; *Consistency Checking in Complex Object Database Schemata with Integrity Constraints*; IEEE Transactions on Knowledge and Data Engineering, Vol.10, No.4, July/August 1998.

[6] F.Bry, N.Eisinger, H.Schutz, S.Torge; *SIC: Satisfiability Checking for Integrity Constraints*; Proc. of Deductive Databases and Logic Programming (DDLP'98), workshop at JICSLP, 1998.

[7] F.Bry, R.Manthey; *Checking Consistency of Database Constraints: a Logical Basis*; Proc. of the 12th Int. Conf. on Very Large Data Bases '86 (VLDB'86); Kyoto, Japan, August 1986.

[8] F.Bry, R.Manthey; *Proving Finite Satisfiability of Deductive Databases*; Proc. of the Conf. on Logic and Computer Science; Karlsruhe, West Germany, October 1987.

[9] D.Calvanese, M.Lenzerini; *Making Object-Oriented Schemes More Expressive*; Proc. of the 13th Int. Symp. on Principles of Database Systems '94 (PODS'94); Minneapolis, USA, May 1994.

[10] L.Cardelli; *A Semantics of Multiple Inheritance*; Info.& Comp., 76, pp. 138-164; 1988 (Preliminary version in LNCS 173, Springer-Verlag, 1984).

[11] R.G.G.Cattel, D.Barry; *ODMG-97: The Object Database Standard*; Release 2.0, Morgan Kaufmann Series in Data Management Systems, Jim Gray Series Editor, 1997.

[12] C.Chang, R.Lee; *Symbolic Logic and Mechanical Theorem Proving*; Academic Press, London 1987.

[13] N.Coburn, G.E.Weddel; *Path Constraints for Graph-Based Data Models: Towards a Unified Theory of Typing Constraints, Equations, and Functional Dependencies*; Proc. of Int. Conf. on Deductive and Object-Oriented Databases '91 (DOOD'91), Munich, Germany, 1991; Lecture Notes in Computer Science (LNCS) 566, Springer-Verlag.

[14] E.F.Codd; *Extending the database relational model to capture more meaning*; ACM TODS, Vol.4, No.4, 1979.

[15] T.H.Cormen, C.E.Leiserson, R.L.Rivest; *Introduction to Algorithms*; The MIT Press and McGraw-Hill Book Company; 1990.

[16] G.Di Battista, M.Lenzerini; *Deductive Entity Relationship Modeling*; IEEE Transactions on Knowledge and Data Engineering, Vol.5, No.3, June 1993.

[17] R.Fagin, M.Y.Vardi; *The Theory of Data Dependency - An Overview*; Proc. of ICALP; 1984.

[18] A.Formica; *Finite Satisfiability of Integrity Constraints in Object-Oriented Database Schemas*; Technical Report IASI R.469, and accepted to be published in IEEE Transactions on Knowledge and Data Engineering, 2000.

[19] A.Formica, H.D.Groger, M.Missikoff; *Object-Oriented Database Schema Analysis and Inheritance Processing: a Graph-Theoretic Approach*; Data & Knowledge Engineering, Vol.24, No.2, pp. 157-181, North-Holland, 1997.

[20] A.Formica, H.D.Groger, M.Missikoff; *An Efficient Method For Checking Object-Oriented Database Schema Correctness*; ACM Transactions on Database Systems (TODS), Vol.23, No.3, pp. 333-369, September 1998.

[21] A.Formica, M.Missikoff; *Integrity Constraints Representation in Object-Oriented Databases*; in "Information and Knowledge Management", T.W.Finin, C.K.Nicholas, Y.Yesha (Eds.), Lecture Notes in Computer Science (LNCS) 752, pp. 69-85, Springer-Verlag, 1993.

[22] A.Formica, M.Missikoff, R.Terenzi; *Constraint Satisfiability in Object-Oriented Databases*; East-West Database Workshop, Klagenfurt 1994; Workshops in Computing, J.Eder and L.A.Kalinichenko (Eds.), pp.48-60, London, Springer-Verlag, 1995.

[23] H.Gallaire, J.M.Nicholas; *Logic and Databases: An assessment*; Proc. of Third Int. Conf. on Database Theory'90 (ICDT'90), S.Abiteboul, P.Kanellakis (Eds.), Lecture Notes in Computer Science (LNCS) 470, Springer-Verlag, 1990.

[24] M.R.Genesereth, N.J.Nilsson; *Logical Foundations of Artificial Intelligence*; Morgan Kaufmann; Los Altos, CA, 1987.

[25] M.Hammer, D.J.McLeod; *A framework for data base semantic integrity*; Proc. of 2nd Int. Conf. Software Engineering"; pp.498-504, San Francisco, October, 1976.

[26] S.N.Khoshafian, G.P.Copeland; *Object Identity*; Proc. of Int. Conf. on Object-Oriented Programming Systems Languages and Applications '86 (OOPSLA'86), September 1986.

[27] S.Khoshafian, R.Abnous; *Object-Orientation - Concepts, Languages, Databases, User-Interfaces*; Wiley, New York, 1990.

[28] W.Kim; *Object-Oriented Databases: Definition and Research Directions*; IEEE Transactions on Knowledge and Data Engineering, Vol.2, No.3, September 1990.

[29] M.Kifer, W.Kim, Y.Sagiv; *Querying Object-Oriented Databases*; Proc. of ACM SIGMOD'92 Conference, San Diego, pp.393-402, June 1992.

[30] R.Kowalski, F.Sadri, P.Soper; *Integrity Checking in Deductive Databases*; Proc. of the 13th Int. Conf. on Very Large Data Bases '87 (VLDB'87); Brighton, 1987.

[31] Y.E.Lien; *Multivalued Dependecies with Null Values in Relational Databases*; Proc. 5th VLDB, Rio de Janeiro, 1979.

[32] M.Lenzerini, P.Nobili; *On the Satisfiability of Dependency Constraints in Entity-Relationship Schemata*; Information Systems, Vol.15, N.4, pp. 453-461, 1990.

[33] M.Missikoff, M.Toiati; *MOSAICO - A System for Conceptual Modeling and Rapid Prototyping of Object-Oriented Database Applications*; Proc. of ACM SIGMOD'94 Conference, Minneapolis, p.508, 24-27 May, 1994.

[34] M.Missikoff, M.Toiati; *Safe Rapid Prototyping of Object-Oriented Database Applications*; Proc. of IEEE Int'l Workshop on Rapid System Prototyping, Grenoble, June, 1994.

[35] J.Mylopoulos, P.A.Bernstein, H.K.T.Wong; *A Language Facility for Designing Database-Intensive Applications*; ACM Transactions on Database Systems (TODS), Vol.5, No.2, June 1980.

[36] B.Nebel; *Terminological cycles: Semantics and Computational properties*; in "Principles of Semantic Networks", J.Sowa (Ed.), Morgan Kaufmann, 1991.

[37] K.Truemper; *Effective Logic Computation*; Wiley-Interscience Pub., New York, 1998.

[38] D.C. Tsichritzis, F.H. Lochovsky; *Data Models*; Prentice-Hall, Englewood Cliffs, N.J., 1982.

[39] S.D.Urban, L.M.L.Delcambre; *Constraint Analysis for Specifying Perspectives of Class Objects*; Proc. of the 5th Int. Conf. on Data Engineering; Los Angeles, February 1989.

[40] J.D.Ullman; *Principles of Database and Knowledge-Base Systems*; vol.I; Computer Science Press; 1988.

[41] A.Yahia, L.Lakhal, R.Cicchetti, J.P.Bordat; *iO2 An Algorithmic Method for Building Inheritance Graphs in Object Database Design*; Proc. of ER'96, Cottbus, Germany, October 1996.

[42] C.Zaniolo; *Database Relations with Null Values*; Journal of Computer and System Science 28, 1984.

[43] X.Zhang, Z.M.Ozsoyoglu; *Implication and Referential Constraints: A New Formal Reasoning*; IEEE Transactions on Knowledge and Data Engineering (TKDE), V.9, No.6, 1997.

[44] J.Zhang, H.Zhang; *System Description: Generating Models by Sem*; in "Automated Deduction, CADE-13" M.A.McRobbie, J.K.Slaney (Eds.), Lecture Notes in Artificial Intelligence (LNAI) 1104, 1996.