Finite Satisfiability of Integrity Constraints in Object-Oriented Database Schemas

Anna Formica
IASI-CNR, Rome, Italy
formica@iasi.rm.cnr.it

Abstract

Checking satisfiability of database constraints is a fundamental problem in database design. In addition, database constraints have to be not only satisfiable but also *finitely* satisfiable. This problem is generally addressed by using theorem provers that, being developed for first order logic formulas, are based on semidecidable procedures. Furthermore, even in simple cases, theorem provers are quite inefficient in dealing with comparison operators such as, for instance, the equality. In this paper, a decidable, sound, and complete method for checking finite satisfiability of a specific class of integrity constraints for Object-Oriented databases, including the equality constraints, is presented. The method, that is based on a graph-theoretic approach, has exponential complexity in the worst case, rarely occurring in practice.

Index Terms: *Object-Oriented databases*, *constraint satisfiability* (*consistency*), *recursive schemas*, *axioms of infinity*, *equality constraints*.

1 Introduction

In database area great attention has been devoted to the problem of *constraint satisfaction* rather than *constraint satisfiability*. The former consists in verifying if a *given* database satisfies the set of integrity constraints specified in the schema (i.e., if the database is a *model* of the constraint set), whereas the latter (also referred to as *consistency*) addresses the absence of contradictions within the set of integrity constraints itself, independently of any given database (i.e., the existence of at least one *model* of the constraint set). Furthermore, since a database is finite by definition, database constraints have to be not only satisfiable but also *finitely* satisfiable, that is, the existence of at least one *finite* model is required. In this sense, we say that database constraints have to be free of the so-called *axioms of infinity*, i.e., sets of constraints that admit *only infinite* models [7, 16].

For instance, consider the following set of database constraints (*):

- every *person* has one *salary*, that is an *integer*, and drives one *vehicle*, that is a *car*;
- every *car* has one *price*, that is an *integer*, and one *owner*, that is a *person*;
- the *salary* of a *person* has to be strictly greater than the *price* of the *vehicle* he/she drives, whereas the *price* of a *car* has to be strictly greater than the *salary* of his/her *owner*.

This set of integrity constraints does not contain any contradiction, i.e., it is satisfiable. However, it is easy to verify that only infinite databases may satisfy it, i.e., it is an axiom of infinity (or, in terms of Object-Oriented databases [3, 25], an infinite number of objects is required to define a model for it). Now, consider the further integrity constraint (**):

- every *person* has to be the *owner* of the *vehicle* he/she drives.

The above constraint essentially states that the person driving the vehicle and the owner of that vehicle must be the same, i.e., it involves an equality constraint. Now, it is easy to verify that the set of constraints (*), together with (**), becomes unsatisfiable, i.e., no database (finite or infinite) can satisfy it. Therefore, the above two sets of integrity constraints are both inadmissible sets of database constraints.

In general, the problem of whether a set of first order logic formulas is unsatisfiable is semidecidable, i.e., it is possible to define procedures that report unsatisfiability in a finite time if the set is indeed unsatisfiable, whereas for satisfiable sets, termination is not guaranteed (and semidecidability holds also for finite satisfiability). Therefore, in order to check finite satisfiability of database constraints, two possible directions can be followed: (i) to support the database designer with theorem provers that, being developed for first order logic formulas, are based on semidecidable procedures; (ii) to define decidable methods for database constraint specification languages with a restricted expressive power.

For instance, following the first direction, in the case of the set (*), that is an axiom of infinity, a theorem prover does not provide any answer because the computation does not terminate. Whereas, in the case of the set (*) enriched with the constraint (**), the inconsistency is detected, even if, in general, equality constraints are not handled by theorem provers in an efficient way (this issue will be better illustrated in the related work).

In this paper, the second direction has been followed, and a database constraint language sufficiently expressive to specify, for instance, the set of constraints (*) and (**) has been considered. In particular, this work focuses on finite satisfiability of Object-Oriented database schemas, where a schema is defined by a set of *types*, possibly recursive. The data model employed in this paper is based on the language TQL^+, aimed at modeling the structural aspects and integrity constraints of an Object-Oriented database application [19, 30]. In particular, in this work a decidable, sound, and complete method for checking finite satisfiability of a specific class of integrity constraints, namely the *θ-constraints*, is presented. θ-constraints, where θ stands for a comparison operator such as $>$, $\geq$, $=$, 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. The idea of *paths* appeared in [32], and their importance was also emphasized in [27], for expressing queries in Object-Oriented databases.

From a theoretical point of view, the method is exponential in the worst case, even if, as we will see in the paper, the schemas which exhibit type patterns leading to such a worst case have a low practical impact.

1.1 Related work

In the past, the consistency of a schema was not considered a problem because the use of data models with a limited expressive power (such as the relational one) prevented from writing inconsistent schemas [35, 37]. With the advent of more expressive data models, checking the consistency of a database schema, independently of any database state, becomes a fundamental and critical problem, and the need for automatic tools that support schema design becomes mandatory. However, in the literature, less attention has been paid to the problem of constraint satisfiability [21], whereas a different, but strictly related, issue has been widely addressed: the constraint satisfaction problem, i.e., the problem of verifying the validity of a database state with respect to a set of integrity constraints (see, for instance, [23, 28, 36]). In the following, some of the existing works related to constraint satisfiability are recalled.

In [1], the consistency of a set of types related by means of ISA relationships and disjointness constraints, in a network data model, has been investigated. A characterization of consistent and acyclic schemas with respect to ISA hierarchies (*non-redundant* schemas) has been given by using a graph-theoretic approach. However, in the related data model types are not structured, i.e., no properties

(typed or untyped) are associated with them.

Some results, originally obtained in the field of Entity Relationship conceptual modelling [29, 15], migrated to Object-Oriented databases [12]. In particular, in [29] the consistency problem for a specific class of database integrity constraints, namely the *cardinality ratio* constraints, has been analyzed. Informally, these constraints are used to specify restrictions on the mappings allowed between entities and relationships. In [15] an Entity Relationship oriented model, called *Entity Relationship Language* (*ERL*), including ISA relationships, typing, and disjointness constraints, has been presented. In particular, a sound and complete inference algorithm, allowing the satisfiability checking of ERL schemas, has been shown. Finally, in [12], a technique has been presented for checking the consistency of class definitions in an extended Object-Oriented data model, offering the possibility of specifying ISA relationships, disjointness constraints, inverse attributes (properties), and cardinality constraints. However, the data models proposed in all these papers do not allow the modeling of integrity constraints involving comparison operators as, for instance, θ-constraints.

In [5], the consistency of Object-Oriented database schemas enriched with integrity constraints has been addressed. In particular, two alternative formalisms have been presented: one including *path relations* (very similar to θ-constraints), but disallowing recursive schemas, the other one with the capability of expressing recursive schemas, but disallowing path relations. Being the common kernel of the two formalisms very expressive (it includes union, complement, quantified sets, value types etc..), the consistency checking of recursive schemas enriched with path relations is undecidable. As a result, in the mentioned paper, two specialized reasoners for consistency checking, based on the *tableaux calculus*, have been defined, one for each proposed formalism (that, therefore, do not allow the consistency checking, for instance, of the sets (*) or (**) mentioned at the beginning of this section, for which recursive schemas and path relations are required).

In [39], the problem of reasoning with *implication* and *referential* constraints has been addressed. The former are a generalization of *functional dependencies*, whereas the latter correspond to *inclusion dependencies*. In particular, since the *implication* problem for functional and inclusion dependencies is undecidable, in the mentioned paper acyclic referential constraints have been addressed. Therefore,

among the main results, a novel characterization for the implication and referential constraints (IRC)-refuting problem has been proved. Furthermore, the relationships among the IRC-refuting problem with the query containment problem and the redundant implication constraint problem have been analyzed.

The class of integrity constraints addressed in this paper can be seen as a restricted form of cyclic referential constraints as defined in [39], where in each formula: (i) the antecedent is defined by one unary predicate, and the consequent contains unary and binary predicates at most, (ii) constants are not allowed, (iii) the $\neq$ operator is not allowed, and (iv) variables appearing in the predicates of the consequent obey to a sort of "concatenation" law, for which every variable belongs to a chain originating from the variable defined in the antecedent (this is clearly stated in formal terms in Subsection 2.2, where the formal semantics of the language is presented).

Furthermore, in [13] *path equation* constraints on complex objects have been addressed, that can be compared with the equality constraints mentioned in this paper. However, in that paper such constraints are analysed in combination with functional dependencies and specialization constraints, while integrity constraints defined with comparison operators are not addressed.

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, 26]. Inheritance conflict resolution consists in determining the types to be inherited in the case of common properties in the definitions of the supertypes. Of course, inheritance conflicts may arise depending on the expressive power of the language for specifying the structural component of types. For instance, in [38] an algorithm that allows the construction of the ISA hierarchy of an Object-Oriented database schema has been presented, where the structural component of a type is defined by a set of properties. However, properties are not typed, therefore, inheritance conflicts cannot arise.

The problem of checking the consistency of Object-Oriented database schemas organized according to ISA hierarchies has been analyzed in [4, 17, 18]. In particular, in [4] a method for treating and solving different kinds of inheritance conflicts has been defined, allowing maximum flexibility to the designer. In [17, 18], different methods for the consistency checking of a set of types defined according to the type theory of [9] have been presented, where structural recursion is allowed. In particular, in [18] a decidable, sound, and complete method has been proposed that allows consistency to be checked in polynomial time

in the size of the input schema. However, in these papers explicit integrity constraints are not addressed.

In the context of deductive databases, schema consistency checking has been widely investigated by relying on theorem prover techniques. For instance in [7, 8] a schema is a set of first order logic formulas, and schema consistency checking is performed by using the theorem prover *Satchmo*, successively refined in *Sic* [6]. In these papers the authors propose methods for checking inconsistency and finite consistency that are semidecidable.

However, besides semidecidability, it is well-known that, in the presence of equality constraints, theorem provers do not handle equality in an efficient way. Below, two main techniques on which theorem provers are based to deal with equality are very briefly recalled (a deeper analysis is beyond the scope of this paper, and the reader may refer to [11, 22]).

In order to have more efficient theorem proving procedures, an inference rule that is a refinement of the *resolution* inference rule is employed, namely the *hyperresolution*. Furthermore, two possible techniques are adopted to deal with equality: (a) all equality axioms are added to the schema, or (b) one more inference rule, namely the *paramodulation* rule, is added to the schema, together with and a reduced set of equality axioms. In [20], we analyzed both these solutions and we also proposed a new inference rule, referred to as *HaRT* (*Hyperparamodulation and Reflex on Terms*), that allows one to handle equality without requiring the specification of any equality axioms. In particular, in [20] the results about some experiments on these three possible approaches have been presented, using the theorem prover *Satchmo*. For instance, for checking the unsatisfiability of the set of integrity constraints (*) and (**) mentioned at the beginning of this section, according to the solutions (a) or (b), we had to stop *Satchmo*'s elaboration when more than two thousands of predicates were produced, while *Satchmo* enriched with the *HaRT* ad-hoc inference rule allowed the detection of the inconsistency after the generation of about one hundred of these predicates. Nevertheless, even if the absence of equality axioms greatly improves the efficiency of the proofs, all the three mentioned approaches are unsatisfactory being semidecidable.

Finite satisfiability checking becomes decidable if the problem is to find at least one finite model of (or not exceeding) *a given cardinality*. For instance, *Sem* [40] is a system for enumerating finite models of first order theories, in which finite satisfiability checking is treated as a constraint satisfaction problem. In terms of Object-Oriented databases, given a set of constraints and, for each type of the database

schema, the cardinality of the set of objects (*instances*) allowed for that type, the system always returns the existence or non-existence of a model verifying such cardinality requirements. Similarly, *Leibniz* is a system for logic programming, based on *logic decomposition* techniques [34]. It compiles fast solution algorithms for checking the satisfiability of a given set of boolean formulas in conjunctive normal form, in which the variables range on finite domains.

As already mentioned, in this paper, a decidable method for checking finite satisfiability of recursive schemas enriched with θ-constraints is presented. The method is based on a graph-theoretic approach, where θ operators are modeled by labeled arcs. In particular, in the presence of equality constraints, in place of the proliferation of equality predicates, typical of the elaboration of a theorem prover, nodes connected through arcs labeled with equality are simply collapsed.

The paper is organized as follows. In Section 2, the syntax and the formal semantics of the database constraint language employed in this paper is presented. In Section 3, the proposed method is introduced by means of examples and, in particular, the constraint sets (*) and (**), informally discussed in this section, are formally tackled. In Section 4, the definitions necessary to formally introduce the proposed approach are given and, in Section 5, the soundness and the completeness of the method (i.e., a formal characterization of finitely satisfiable schemas) is shown. Then, in Section 6, a few more examples are illustrated and, in Section 7, the conclusion and a few indications about future work follow. Finally, in the Appendix, the formal syntax of the constraint language used in this paper and the proof of the main theorem of Section 5 are presented.

2 Formal Basis

In this section the Object-Oriented database (ODB) analysis model, employed in this paper, is briefly presented. It is based on the language TQL^{++} for conceptual modeling of ODB applications [30, 31]. In particular, this paper focuses on the sublanguage TQL^{+} [19], that is the component of TQL^{++} aimed at modeling the structural aspects and integrity constraints of an ODB application. Such an ODB model is compliant with ODMG [10], a standard for ODBs that is gaining a wide consensus within the database community.

2.1 TQL^+ Syntax

In the ODB model of TQL^+, the *intensional* component (also called *conceptual model*) of a database application is represented by a TQL^+ *schema*, which is a collection of *types*. Types contain the description of the structure of the objects that will populate the type *extension* (also called *class*), and the integrity constraints these objects have to satisfy. An example of a set of types forming a TQL^+ schema is shown below.

Example 2.1
$person := [name : string, age : integer, phone : integer, child : person]$
$employee :=$ **ISA** $person\ [salary : integer, vehicle : car, boss : employee]$,
$\quad ic1 : this.salary \leq this.boss.salary$
$car := [maker : string, color : string]$

□

The formal syntax of TQL^+ is presented in Appendix A.1. In TQL^+ a type has a *label* (*t_term*) and a *tuple*. A tuple is defined by a set of *typed properties* (*tp*), that is a set of properties each of which is associated with a type. Types are organized according to a generalization hierarchy, declared by means of the **ISA** construct. A property is identified by a label (*p_term*). It can be either an *attribute* or a *relationship* if typed by using respectively: (i) an *atomic type* (*a_type*), e.g., *integer* or *string* (for instance, in the example, *person.name*); (ii) a *t_term* (as, in the example, *person.child*), establishing an explicit link (or association) between two types.

Relationships can form cycles, hence resulting in recursive types. In a tuple, multiple occurrences of the same property labels are not allowed and properties are assumed to be single-valued.

TQL^+ explicit integrity constraints are *θ-constraints*, where θ stands for a comparison operator, such as "=", "≥, ">". A θ-constraint has a left and a right hand sides, 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 example, in the above schema, one θ-constraint is given stating that an *employee* has a salary not exceeding the one of his/her *boss*.

In order to present the notion of a TQL^+ schema, the $\mathcal{T}$ function is introduced below. Such a function is applied to a *t_term* followed by a *path*, and returns the type of the first property of the *path* (as defined in the tuple associated with the *t_term*), followed by the remainder sequence of properties of that *path*.

Definition 2.1 [**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:

$\mathcal{T}(employee.boss.salary) = employee.salary$

and:

$\mathcal{T}^2(employee.boss.salary) = integer$

while:

$\mathcal{T}(employee.maker)$ is undefined.

The formal definition of a TQL^+ schema follows.

Definition 2.2 [TQL^+ **schema**] A finite set of types is a TQL^+ *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:

  $label$: $this.p_1...p_n$ θ $this.q_1...q_v$

  $\mathcal{T}^k(\tau.p_1...p_n)$ and $\mathcal{T}^h(\tau.q_1...q_v)$ are defined, for k = 1...n, and h = 1...v, and, furthermore:

  $\mathcal{T}^n(\tau.p_1...p_n) = \mathcal{T}^v(\tau.q_1...q_v)$ if $v > 0$

  $\mathcal{T}^n(\tau.p_1...p_n) = \tau$ if $v = 0$.

□

The first two requirements, in the above definition, are quite obvious and the acyclicity requirement for inheritance is well-known. The last requirement concerns integrity constraints *paths*: each property of a *path* must be present in the tuple of the "traversed" type and, furthermore, the types associated with the last properties of the *path* must coincide.

Notice that in a schema the integrity constraints are supposed to be correctly typed, e.g., in Example 2.1, the integrity constraint:

$ic2 : this.boss.boss > this.boss$

associated with *employee* would be rejected at a pre-processing stage, by using a type-checker.

With regard to inheritance hierarchies, it is well-known that, in order to obtain all the typed properties of a type defined in terms of a set of supertypes, the **ISA** construct can be removed by applying the inheritance process [9]. The inheritance process in the structural component of an ODB schema has been extensively investigated in [17, 18]. 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.

TQL^+ is endowed with a denotational semantics [19], inspired by the *descriptive* semantics presented in [33], that will be recalled in the next section.

2.2 Semantics of TQL^+

Definition 2.3 [**Extension function**] Let $\mathcal{D}$ be a (possibly infinite) set of oids [24] representing a given state of the Application Domain, T the set of TQL^+ 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 $n \geq 2$.

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)

- $\mathcal{E}([tp]) = \mathcal{E}([p\_term{:}body]) = \{x \in \mathcal{D} \mid \|S_{p\_term,x}\| = 1$, and if $y \in S_{p\_term,x} \Rightarrow y \in \mathcal{E}(body)\}$ where $\|S_{p\_term,x}\|$ stands for the cardinality of the set $S_{p\_term,x}$.

Integrity Constraint Extension:

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

□

Definition 2.4 [**Interpretation of a TQL^+ schema**] An *interpretation* of a TQL^+ 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^+ schema**] A *model* of a TQL^+ 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^+ schema**] A TQL^+ 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^+ schema**] A TQL^+ 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 A Graph-theoretic Approach to Finite Satisfiability Checking

In this section, the method proposed in this paper for checking finite satisfiability of a TQL^+ schema is informally presented via examples. In particular, first, two simple non-recursive schemas are illustrated and, then, the database constraints informally discussed in the Introduction are formally tackled.

3.1 Non-recursive Schemas: Examples

In the proposed approach each type of the schema is associated with a graph. The nodes of this graph, referred to as *schema* graph, are labeled with *t_terms* or atomic types, while the arcs are labeled with

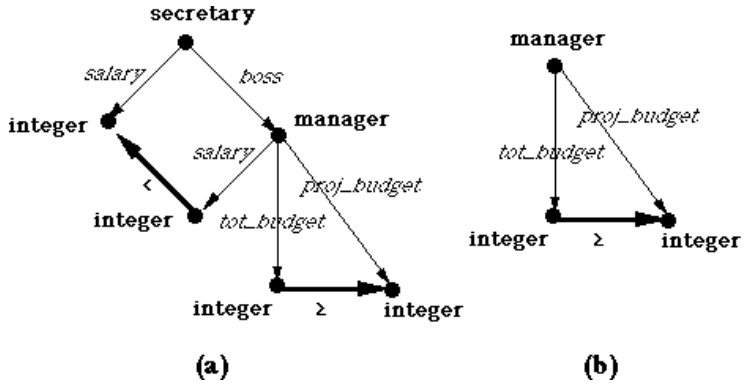

Figure 1: **(a)** $G_{Sat}(secretary)$ and **(b)** $G_{Sat}(manager)$ graphs - Example 3.1

properties or comparison operators. In particular, in a schema graph different nodes with the same type label may be present and, in addition, for each node, outgoing arcs with the same property label and different destination nodes are not allowed. Given a type, the graph on which the method is based is constructed starting from a particular schema graph, referred to as the G_{Sat} graph (where Sat stands for satisfiability). In the G_{Sat} graph associated with a given type, each integrity constraint of the type is modeled by a pair of paths. In particular, for each of the sequences of properties defining the right and the left hand sides (i.e., the *paths*) of the integrity constraint, a path exists in the G_{Sat} graph whose ordered sequence of property labels coincides with the one defined in the constraint. For instance, consider the following schema.

Example 3.1

$secretary := [name : string, salary : integer, boss : manager],$
$\quad ic1 : this.salary < this.boss.salary$
$manager := [name : string, salary : integer, proj\_budget : integer, tot\_budget : integer],$
$\quad ic2 : this.proj\_budget \leq this.tot\_budget$

It is easy to verify that the above schema is finitely satisfiable: a finite and non-empty model can be defined, for instance, by simply introducing one single oid in the *secretary* extension, and one single oid in the *manager* extension, with the related values, one for each typed property.

The G_{Sat} graph associated with the *secretary* type of the above schema is shown in Fig.1 (a), and is indicated as $G_{Sat}(secretary)$. In this graph, we have two nodes labeled with *secretary* and *manager*, respectively, and four nodes labeled with *integer*. Furthermore, since the type *secretary* contains the integrity constraint $ic1$, two paths starting from the node labeled with *secretary* are present in the graph, one labeled with the *salary* property (corresponding to the left hand side of $ic1$), the other one labeled with the sequence *boss.salary* (corresponding to the right hand side of $ic1$). Notice that, for

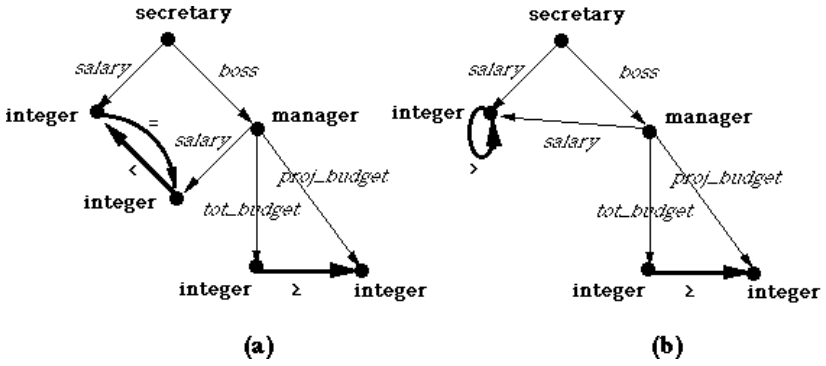

Figure 2: **(a)** $G_{Sat}(secretary)$ and **(b)** $G^{eq}_{Sat}(secretary)$ graphs - Example 3.2

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 *boss* is labeled with *manager*, as defined by the tuple of *secretary*). Furthermore, an arc between the final nodes of the paths modeling $ic1$ is present, labeled with the related comparison operator (we assumed that arcs labeled with comparison operators are directed according to the $>$ or $\geq$ operators, but the opposite choice could be adopted as well).

Similarly, in Fig.1 (a) the outgoing arcs of the node labeled with *manager* model the *paths* of the constraint $ic2$, and a further arc has been drawn modeling the related comparison operator $\leq$. Finally, the $G_{Sat}(manager)$ graph, shown in Fig.1 (b), is a subgraph of $G_{Sat}(secretary)$. □

Before showing how finite satisfiability can be checked for this schema on the basis of the G_{Sat} graphs, consider the previous schema enriched with the constraint $ic3$, as shown by the following example.

Example 3.2

$secretary := [name : string, salary : integer, boss : manager]$,
$\quad ic1 : this.salary < this.boss.salary$,
$\quad ic3 : this.salary = this.boss.salary$
$manager := [name : string, salary : integer, proj\_budget : integer, tot\_budget : integer]$,
$\quad ic2 : this.proj\_budget \leq this.tot\_budget$

With the addition of the constraint $ic3$, this schema obviously becomes unsatisfiable. In this case, the $G_{Sat}(secretary)$ graph is the one shown in Fig.1 (a) with one additional arc, introduced by the constraint $ic3$ (see Fig.2 (a)).

In the proposed method, the presence of equality constraints requires a further step, that is, all the nodes connected through arcs labeled with the equality operator have to collapse. Therefore, the $G_{Sat}(secretary)$ graph of Fig.2 (a) is transformed into the one shown in Fig.2 (b), that will be referred

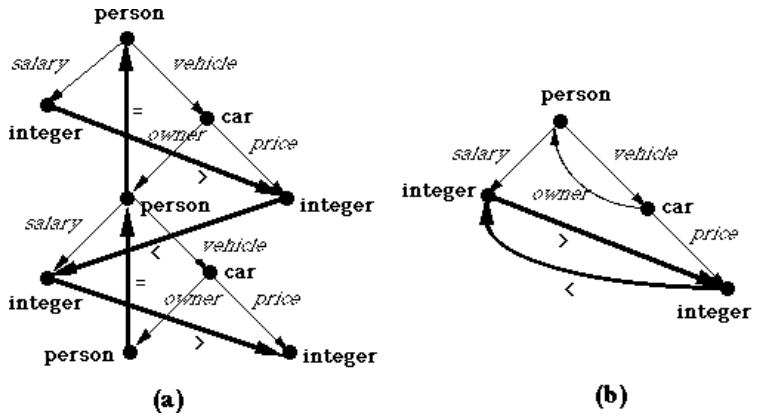

Figure 3: **(a)** $G_{Sat}(person)$ and **(b)** $G^{eq}_{Sat}(person)$ graphs - Example 3.3

to as the $G^{eq}_{Sat}(secretary)$ graph (eq stands for equality). □

Checking finite satisfiability of the above schemas according to the proposed method consists in verifying that, in each of the G^{eq}_{Sat} graphs associated with the types of the schema, no cycles labeled with strict comparison operators are present (notice that, in the absence of equality constraints, G_{Sat} and G^{eq}_{Sat} coincide). Therefore, the schema of Example 3.1 is finitely satisfiable, while the schema of Example 3.2 is unsatisfiable, because the graph of Fig.2 (b) contains a loop labeled with the $>$ operator (notice that also in the case of Example 3.2, the $G^{eq}_{Sat}(manager)$ graph is a subgraph of the one defined for $secretary$).

3.2 Recursive Schemas: Examples

The schemas of the examples presented in the previous subsection are very simple. Consider now the database constraints that have been informally discussed in the Introduction. Let us start with the set (*) enriched with (**) (i.e., the unsatisfiable constraint set), that is formalized in the following example.

Example 3.3

$person := [salary : integer, vehicle : car],$
$\quad ic1 : this.salary > this.vehicle.price,$
$\quad ic2 : this.vehicle.owner = this$
$car := [price : integer, owner : person],$
$\quad ic3 : this.price > this.owner.salary$

The graph shown in Fig.3 (a) represents the $G_{Sat}(person)$ graph associated with the type $person$ of Example 3.3. Notice that the node labeled with $person$, on the top of the graph, has an outgoing arc labeled with $salary$, modeling the left hand side of the constraint $ic1$. While the paths starting from that

node, one labeled with the sequence *vehicle.price* and the other one with *vehicle.owner*, model the right hand side of $ic1$ and the left hand side of $ic2$, respectively (see also Fig.5 (a) in Section 4, where these paths are formally introduced). Furthermore, an arc labeled with the comparison operator $>$ has been set between the final nodes of the paths modeling $ic1$, while an arc labeled with the equality operator has been drawn between the final node of the path *vehicle.owner* and the node *person* itself (being the right hand side of $ic2$ simply defined by *this*).

Similarly to the previous examples, all the nodes of the paths modeling $ic1$ and $ic2$ (i.e., the integrity constraints associated with *person*) have been "expanded", in the sense that the paths modeling the integrity constraints associated with their type labels have been drawn. In particular, in Fig.3 (a) the nodes labeled with the first occurrence of *car* (being traversed by the paths modeling both $ic1$ and $ic2$) and the second occurrence of *person* (being traversed by the paths modeling $ic2$), starting from the node on the top of the graph, have been "expanded".

Since an equality constraint is present in the schema, the graph of Fig.3 (a) is successively transformed into the $G^{eq}_{Sat}(person)$ graph shown in Fig.3 (b), by collapsing the nodes connected through the equality operator (notice that, the G^{eq}_{Sat} graph must be a schema graph, i.e., for each node, the outgoing arcs with the same property labels must have the same destination nodes).

In the schema graph shown in Fig.3 (b), a cycle is present, labeled with strict comparison operators. The presence of such a cycle in at least one of the G^{eq}_{Sat} schema graphs associated with the types of the schema allows us to state that the schema is unsatisfiable. □

As already mentioned in the Introduction, in the absence of the constraint $ic2$, the previous schema contains an axiom of infinity, as illustrated by the following example.

Example 3.4

$person := [salary : integer, vehicle : car],$
$\quad ic1 : this.salary > this.vehicle.price$
$car := [price : integer, owner : person],$
$\quad ic3 : this.price > this.owner.salary$

In this case, the G^{eq}_{Sat} graph is a subgraph of the one depicted in Fig.3 (a), and is shown in Fig.4 (a). In particular, with respect to the graph of Fig.3 (a), the paths modeling $ic2$ have been omitted (e.g., the arcs labeled with the equality operator). Furthermore, in this case the node labeled with the second occurrence of *person* has not been "expanded" because such a node is not traversed by any path modeling the integrity constraints associated with *person*, i.e., $ic1$ (see also Section 4, where the construction of

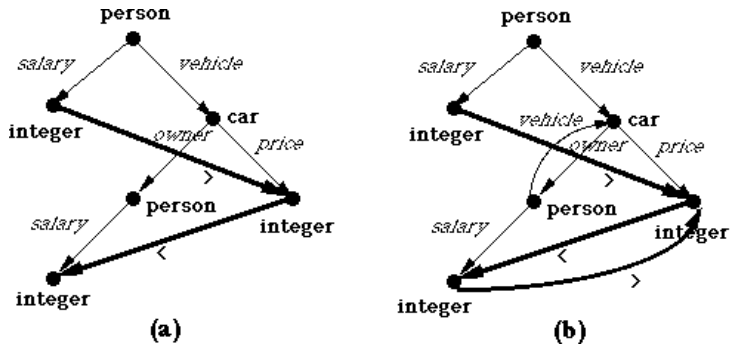

Figure 4: **(a)** $G^{eq}_{Sat}(person)$ and **(b)** $\mathcal{F}(G^{eq}_{Sat}(person))$ graphs - Example 3.4

the $G^{eq}_{Sat}(person)$ graphs of Examples 3.3 and 3.4 is formally illustrated step by step).

In this case, being the $G^{eq}_{Sat}(person)$ graph free of cycles labeled with strict comparison operators, a supergraph, indicated as $\mathcal{F}(G^{eq}_{Sat}(person))$, must be analyzed, that is constructed by suitably adding labeled arcs to the $G^{eq}_{Sat}(person)$ graph of Fig.4 (a). The reason for the construction of such a graph is due to the fact that the paths modeling the integrity constraints of a type have to be present for each of the nodes of the G^{eq}_{Sat} graph labeled with that type. For instance, the node labeled with *person* in the lower part of the graph of Fig.4 (a) has to exhibit the same labeled outgoing paths of the node labeled with *person* on the top of the graph. In this case, the $\mathcal{F}(G^{eq}_{Sat}(person))$ graph is shown in Fig.4 (b). Since this graph contains a cycle labeled with strict comparison operators, the schema of Example 3.4 is not finitely satisfiable. □

In the remainder of the paper, the method is formally presented. In particular, we will see that in the above examples no further "expansions" are required for finite satisfiability checking. In fact, the "expansion" mechanism above mentioned via examples has to be recursively applied until a termination condition holds (that allows us to deal with recursive schemas). Informally, we can anticipate that such a condition consists in checking if the nodes to be "expanded" already *occur* in the graph. This issue will be formally tackled in Subsection 4.1. Furthermore, in Section 5, we will see that, in the case of recursive schemas, the $\mathcal{F}(G^{eq}_{Sat})$ graph is, in general, not connected, and finite satisfiability holds if and only if, for each type of the schema, the related $\mathcal{F}(G^{eq}_{Sat})$ graph has at least one connected component that is free of cycles labeled with strict comparison operators.

4 Formal Definitions

In this section, the notions that are on the basis of the method informally presented above are formally introduced, starting from the notion of a *schema* graph. 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.

Definition 4.1 [**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 4.2 [**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*. □

4.1 The T_{Path} Tree

The notion of a G_{Sat} graph, informally introduced in Section 3, is based on the notion of a T_{Path} tree. In order to formally define it, 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.

In the remainder of the paper, given a schema Σ and a type τ of Σ, $I(\tau)$ indicates the set of θ-constraints associated with τ in Σ.

Definition 4.3 [**The** $left_r$ **and** $right_r$ **sets**] Consider a schema Σ, a type τ of Σ such that $I(\tau) \neq \emptyset$, and an 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 4.4 [**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, being $T_{Path,r}(\tau)$ 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, having a schema, in the above definition the labels of the nodes m_1 and m_2 are singletons that coincide.

For instance, the $T_{Path}(person)$ trees of Examples 3.3 and 3.4 are depicted in Fig.5 (a) and Fig.6 (a), respectively, while the $T_{Path}(car)$ trees related to the same examples coincide (see Fig.5 (b) and Fig.6 (b)).

In Section 3, the notion of "expansion" has been informally introduced by examples. This notion is now formalized by the *Expand* function presented below. To this end, the $\bigsqcup$ operator between schema trees is first introduced.

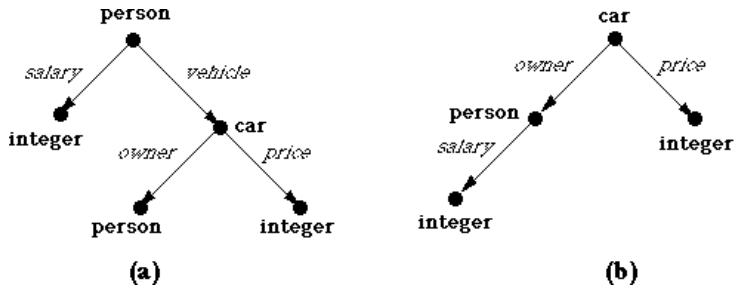

Figure 5: **(a)** $T_{Path}(person)$ and **(b)** $T_{Path}(car)$ trees - Example 3.3

Definition 4.5 [**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 4.6 [**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:

$e(m)$=$e(m_1) \cup e(m_2)$. □

For instance, if the *Expand* function is applied to the $T_{Path}(person)$ trees of Fig.5 (a) and Fig.6 (a), the trees shown in Fig.7 (a) and Fig.7 (b) are obtained, respectively. However, as already mentioned in Section 3, in the case of recursive schemas, the expansion mechanism is more elaborated. In particular, the nodes of the T_{Path} tree must be recursively expanded, until a termination condition holds. Such a condition is formalized by the notion of a T^{rec}_{Path} tree, that is introduced below.

Definition 4.7 [**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

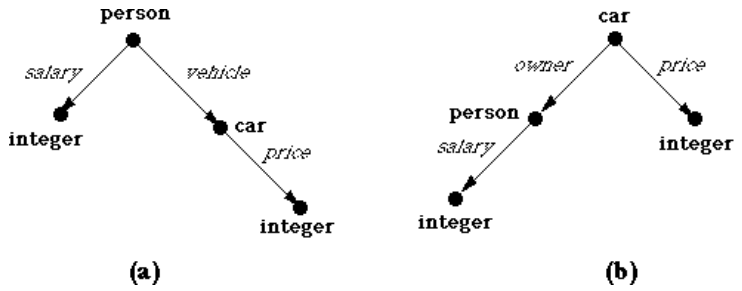

Figure 6: **(a)** $T_{Path}(person)$ and **(b)** $T_{Path}(car)$ trees - Example 3.4

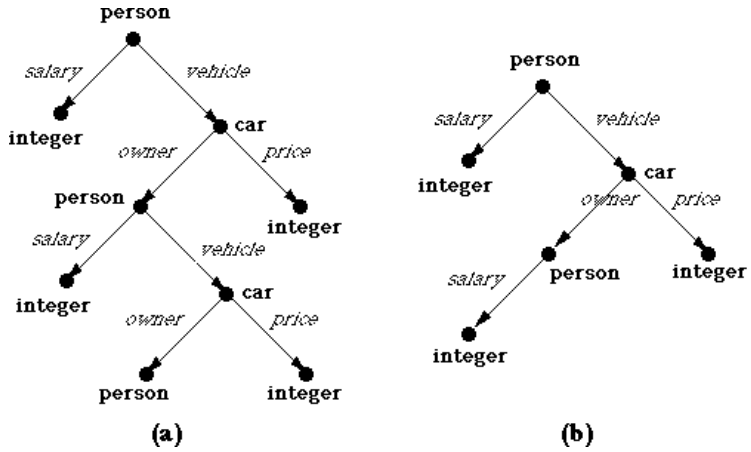

Figure 7: $Expand(T_{Path}(person))$ trees - **(a)** Example 3.3 and **(b)** Example 3.4

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. □

Notice that, in the above definition, at least one expansion is required. This issue will be better clarified in Section 5. However, we can anticipate that the first expansion often allows us to distinguish unsatisfiability from axioms of infinity (as, for instance, in the case of the Example 3.3).

Furthermore, in the above definition, being the *Expand* function applied to T_{Path} trees, in Def.4.6

the labels of the nodes m_1 and m_2 are singletons that coincide.

For instance, in the cases of Examples 3.3 and 3.4, the trees represented in Fig.7 (a) and Fig.7 (b) are the $T^{rec}_{Path}(person)$ trees, respectively, because in Def.4.7, $k = 0$. In fact, in both these cases, the *Expand* function is applied only once because, for each path originating from the root of the tree, *car* and *person* nodes are generated having no new outgoing *property-arcs* with respect to the nodes already occurring in the path. In Section 6, an example where more than one expansion is required will be shown.

Proposition 4.1 [**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. Assume that the schema Σ has n types, m is the maximum of the lengths of the *paths* defining the integrity constraints of Σ, and r is number of properties defined in Σ. Given a type τ of Σ, by construction, from Def.4.6, and Def.4.7, it follows that the $T^{rec}_{Path}(\tau)$ tree is defined and has height at most rmn. □

In the next subsection, the notions of a G_{Sat} graph and a G^{eq}_{Sat} graph are formally presented.

4.2 The G_{Sat} Graph

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 4.8 [**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.4.3, 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:

$\Theta_r(\tau, ic) = \emptyset$. □

As already mentioned, different choices in directing the θ-*arcs*, in the above definition, could be adopted as well.

The G_{Sat} graph is formally introduced below.

Definition 4.9 [**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.4.6, 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. □

For instance, the $G_{Sat}(person)$ graph of Fig.3 (a) has been obtained from the $T^{rec}_{Path}(person)$ tree of Fig.7 (a), according to the definition above.

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

Definition 4.10 [**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.4.6 is applied. □

Definition 4.11 [**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, being the *Collapse* 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 4.2 [**The Collapse** lfp] The *Collapse* has a lfp.

Proof. Given a type τ of a schema Σ, suppose to recursively apply the *Collapse* relation defined above to the schema graph $G_{Sat}(\tau)$. Then, if A_k is the number of nodes of such a graph at the step k, it results:

$A_1 \geq A_2 \geq ... \geq A_n \geq A_{n+1}$ ...

Since the set of positive integers is well-ordered, there exists a minimal element of the sequence that can be obtained in a finite number of steps. This element does not depend on the order the nodes to be collapsed are chosen. □

5 Finitely Satisfiable Schemas

In this section the soundness and the completeness of the proposed method is presented, i.e., a formal characterization of finitely satisfiable TQL^+ schemas. To this end, the $\mathcal{F}(G^{eq}_{Sat})$ graph is formally introduced. In Subsection 3.2, we mentioned that in such a graph the paths modeling the integrity constraints of a type are present for each of the nodes labeled with that type. Roughly, the main idea behind the $\mathcal{F}(G^{eq}_{Sat})$ graph can be summarized as follows: (i) each node of the G^{eq}_{Sat} graph stands for an *instance* (i.e., element of the extension) of the type labeling that node; (ii) according to Def.4.7, further expansions do not add anything "new" to the T^{rec}_{Path} tree. Therefore, once the nodes standing for the instances that have to coincide have been collapsed, the schema is finitely satisfiable if and only if one non-empty and finite model can be defined by "using" the nodes of the G^{eq}_{Sat} graphs associated with the types of the schema. This can be checked through the $\mathcal{F}(G^{eq}_{Sat})$ graph, where each node of the related G^{eq}_{Sat} graph standing for an instance of the type labeling it, stands also for an instance of the integrity constraints associated with that type. To this end, the definitions of *pairs of equivalent paths* and *induced owner* node are introduced below, that allow us to identify the arcs that need to be added to the G^{eq}_{Sat} graph to define the related $\mathcal{F}(G^{eq}_{Sat})$ graph.

Definition 5.1 [**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 5.2 [**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. □

Of course, the owner of a graph is also an induced owner. Given a graph G, the notion of a $\mathcal{F}(G)$ graph can now be formally presented.

Definition 5.3 [**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 5.1 [**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. According to Def.4.7, any node n generated by further steps of the *Expand* function (with respect to the ones required to define the T^{rec}_{Path} tree) already occurs (together with its labeled outgoing arcs) in the path connecting n to the owner of $G^{eq}_{Sat}(\tau)$. Therefore, it is always possible to define at least one connected component in which any node of $G^{eq}_{Sat}(\tau)$ is an induced owner, by adding arcs among the already existing nodes of $G^{eq}_{Sat}(\tau)$. □

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

For instance, in the case of the Example 3.4, the graph shown in Fig.4 (b) is the only connected component that can be defined starting from the graph of Fig.4 (a), having all induced owner nodes (i.e., the $\mathcal{F}(G^{eq}_{Sat}(person))$ graph is connected).

Finally, the notion of a *monotonic* cycle follows and, then, the soundness and completeness of the method is presented.

Definition 5.4 [**Monotonic cycle**] Given a schema Σ, a *monotonic* cycle of a schema graph is a cycle of *θ-arcs* labeled with at least one strict comparison operator. □

Theorem 5.2 [**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 $\mathcal{F}(G^{eq}_{Sat}(\tau))$ has at least one connected component free of monotonic cycles.

Proof. See Appendix A.2. □

Corollary 5.3 [**Unsatisfiable schemas**] If in a schema Σ there exists at least one type γ, $I(\gamma) \neq \emptyset$, such that the schema graph $G^{eq}_{Sat}(\gamma)$ contains monotonic cycles, then the schema Σ is *unsatisfiable*.

Proof. It directly follows from the proof of Theorem 5.2. □

Therefore, the presence of monotonic cycles in one of the G^{eq}_{Sat} graphs associated with the types of the schema is sufficient to state that the schema is unsatisfiable. However, in Section 6, we will see that this condition is not necessary, because there exist unsatisfiable schemas such that, for each type of the schema, the related G^{eq}_{Sat} graph is free of monotonic cycles.

With regard to the complexity of the method, consider a schema Σ having n types and r property labels (p_terms). Furthermore, let c be the number of explicit integrity constraints defined in the schema, m the maximum of the lengths of the *paths* defining the integrity constraints, and $s = min\{r, 2c\}$. As already mentioned in Prop.4.1, the height of the T^{rec}_{Path} tree associated with a type of Σ is at most rmn. In particular, an upper bound for it is the *complete* s-ary tree of height rmn [14] (i.e., the tree in which all leaves have the same depth and all internal nodes have degree s). Indeed, it is easy to verify that, according to Def.4.7, any T^{rec}_{Path} of height rmn, necessarily, is not a complete s-ary tree. Vice versa, any complete s-ary T^{rec}_{Path} tree has nodes of depth at most mn. In other words, it is not possible to define TQL^+ schemas with types whose associated T^{rec}_{Path} trees are complete s-ary trees of height rmn. However, the complexity of the method remains exponential, because the size of the T^{rec}_{Path} tree may be exponential in the size of the schema, as shown by the example below. Consider the following schema, where $n = 4$, $m = 5$, $r = 5$, $c = 8$:

$t\_term_i := [p1 : t\_term_i, p2 : t\_term_i, p3 : t\_term_i, p4 : t\_term_i, p5 : t\_term_{i+1}]$,
$\quad ic1_i : this.p1.p1.p1.p1.p5\ \theta\ this.p2.p2.p2.p2.p5$,
$\quad ic2_i : this.p3.p3.p3.p3.p5\ \theta\ this.p4.p4.p4.p4.p5$
for i = 1...3

$t\_term_4 := [p1 : t\_term_4, p2 : t\_term_4, p3 : t\_term_4, p4 : t\_term_4, p5 : t\_term_1]$,
$\quad ic1_4 : this.p1.p1.p1.p1.p5\ \theta\ this.p2.p2.p2.p2.p5$,
$\quad ic2_4 : this.p3.p3.p3.p3.p5\ \theta\ this.p4.p4.p4.p4.p5$

In this schema, the number of nodes of the $T^{rec}_{Path}(t\_term_1)$ tree exceeds the number of nodes of the complete $(s-1)$-ary tree of height mn. However, as already mentioned in the Introduction, in general, the schemas containing integrity constraints somehow similar to the ones above seem to have a low practical impact. In fact, they model application domains rather complex with respect to the application domains usually addressed in the database field. For instance, it is quite unusual to find a database application domain described by a schema where *all* the types have the *same set* of integrity constraints and, in addition, properties seem to obey to a mathematical law (see also [17], where this issue has been addressed

with regard to inheritance hierarchies).

Then, the G_{Sat} graph can be obtained by visiting the T^{rec}_{Path} tree at most c times. Successively, if e is the number of equality θ-constraints that are present in Σ, since the *Collapse* is applied at most e times, the G^{eq}_{Sat} graph can be constructed by visiting the G_{Sat} graph at most e times. Furthermore, if the G^{eq}_{Sat} graph has q nodes, the number of connected components of $\mathcal{F}(G^{eq}_{Sat})$ corresponds at most to the cardinality of the powerset of the possible sq labeled arcs that can be defined in it (however, this is a theoretical upper bound because each connected component is obtained by adding arcs for a proper subset of nodes of the G^{eq}_{Sat} graph).

Finally, the absence of monotonic cycles in a schema graph can be verified according to well-known algorithms for checking negative-weight cycles in weighted, directed graphs [14], that are polynomial in the size of the input graph.

6 Further Examples

Below, a schema containing types for which the *Expand* function is applied more than once is illustrated.

Example 6.1 Consider the following schema.

(a) $t\_term1 := [p1 : integer, p2 : t\_term2],$
$ic_1 : this.p1 \leq this.p2.p3,$
$ic_2 : this.p1 > this.p2.p4$
$t\_term2 := [p3 : integer, p4 : integer, p5 : t\_term3,$
$p6 : integer, p7 : integer, p8 : integer],$
$ic_3 : this.p3 < this.p5.p9,$
$ic_4 : this.p4 \geq this.p5.p10$
$t\_term3 := [p9 : integer, p10 : integer, p11 : t\_term2],$
$ic_5 : this.p9 < this.p11.p6,$
$ic_6 : this.p10 \geq this.p11.p8,$
$ic_7 : this.p11.p8 > this.p11.p7,$
$ic_8 : this.p11.p7 > this.p11.p6$

The $G_{Sat}(t\_term1)$ graph of this schema, shown in Fig.8, contains a monotonic cycle, therefore, according to Corollary 5.3, the schema is unsatisfiable (for sake of simplicity, the labels of the *θ-arcs* have been omitted). In this case, the *Expand* function has been applied three times, i.e., in Def.4.7, $k = 2$. (However, with respect to the schema of Example 3.3, this is a case where the further expansion due to $k + 1$ in Def.4.7 is not necessary to derive unsatisfiability.)

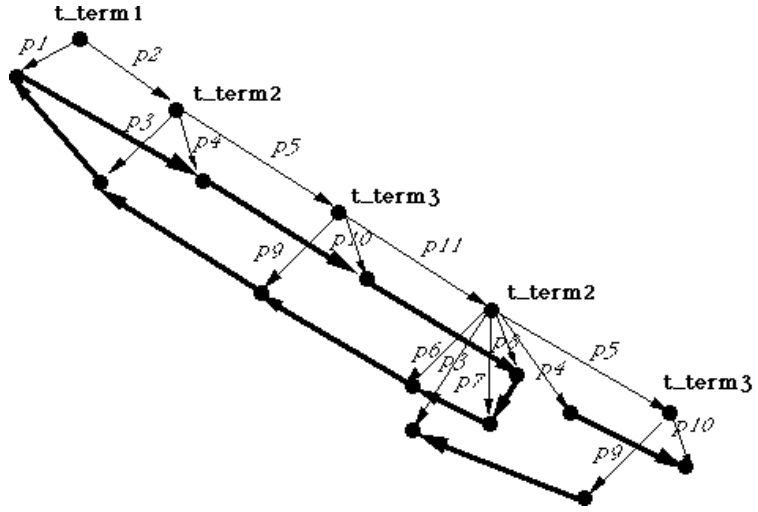

Figure 8: $G_{Sat}(t\_term1)$ graph - Example 6.1 (a)

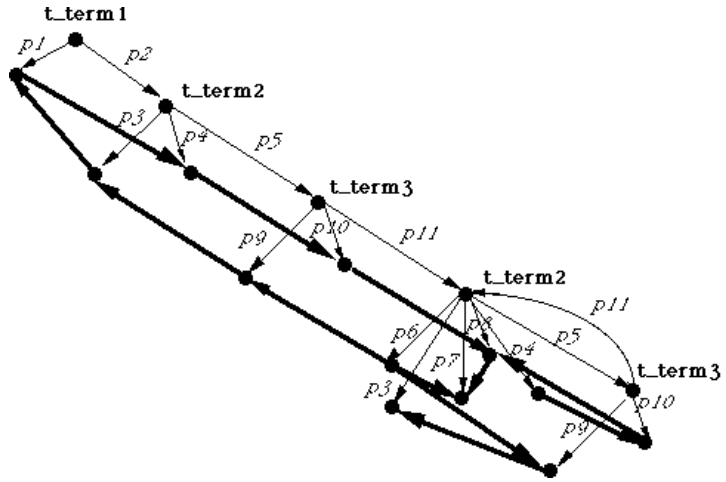

Figure 9: A connected component of $\mathcal{F}(G_{Sat}(t\_term1))$ - Example 6.1 (b)

Now, consider a schema, say (b), obtained from the schema (a), where the integrity constraint ic_8 is replaced by the integrity constraint ic'_8, defined as follows:

ic'_8: $this.p11.p7 < this.p11.p6$.

The schema (b) is finitely satisfiable (for instance, a connected component of $\mathcal{F}(G_{Sat}(t\_term1))$ is given in Fig.9). Notice that, had we applied the *Expand* function only once (i.e., k=0), we would have reported a finitely satisfiable schema as a not finitely satisfiable one (a connected component of $\mathcal{F}(G_{Sat}(t\_term1))$ constructed starting from the $Expand(T_{Path}(t\_term1))$ tree is shown in Fig.10). □

Finally, in the previous section, we mentioned that the presence of monotonic cycles in the G^{eq}_{Sat} graphs is a sufficient but not a necessary condition for unsatisfiability. For instance, consider the following

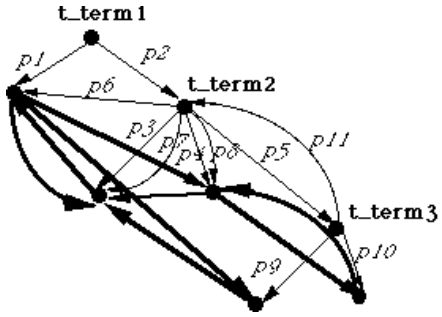

Figure 10: A connected component constructed from the

$Expand(T_{Path}(t\_term1))$ tree - Example 6.1 (b)

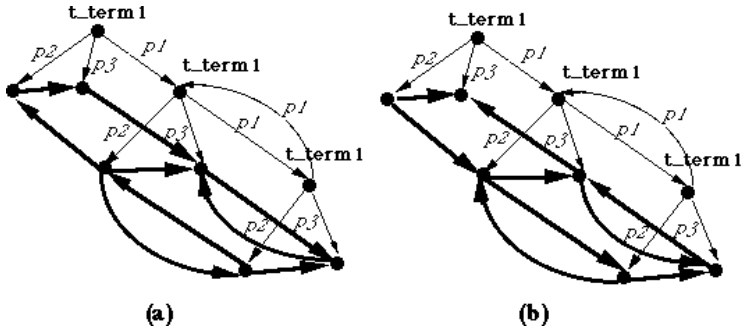

Figure 11: A connected component of $\mathcal{F}(G_{Sat}(t\_term1))$ -

(a) Example 6.2 (a) and **(b)** Example 6.2 (b)

example, regarding two schemas that are not finitely satisfiable.

Example 6.2 Consider first the following schema.

(a) $t\_term1 := [p1 : t\_term1, p2 : integer, p3 : integer]$,
$ic_1 : this.p2 > this.p3$,
$ic_2 : this.p1.p3 < this.p3$,
$ic_3 : this.p1.p2 > this.p2$

The $G_{Sat}(t\_term1)$ graph of this schema does not contain any monotonic cycle, whereas this does not hold for the graph $\mathcal{F}(G_{Sat}(t\_term1))$ (one of its connected components is represented in Fig.11 (a)).

Now, suppose to modify the previous schema by interchanging the comparison operators of the integrity constraints ic_2 and ic_3, as follows:

(b) $t\_term1 := [p1 : t\_term1, p2 : integer, p3 : integer]$,
$ic_1 : this.p2 > this.p3$,
$ic'_2 : this.p1.p3 > this.p3$,
$ic'_3 : this.p1.p2 < this.p2$

This schema behaves similarly to the previous one (a connected component of $\mathcal{F}(G_{Sat}(t\_term1))$ is shown in Fig.11 (b)). According to Theorem 5.2, both the schemas (a) and (b) are not finitely satisfiable, and Corollary 5.3 is not applicable. Indeed, it is possible to see that the schema (a) contains an axiom of infinity, whereas the schema (b) is unsatisfiable. In fact, roughly, the cardinality of the intervals defined by the integer numbers that must be instantiated in correspondence to the properties $p2$ and $p3$, for each of the instances of t_term1, is strictly increasing in the former case, and strictly decreasing in the latter case. □

It is interesting to observe that if we replace the atomic type *integer* with *real*, also the schema (b) contains an axiom of infinity. In other words, if a schema is not finitely satisfiable and the associated G^{eq}_{Sat} graphs are free of monotonic cycles, this method does not allow us to distinguish unsatisfiability from infinite satisfiability. However, this does not seem to be a drawback because, as already mentioned, unsatisfiable constraints and axioms of infinity are both inadmissible database constraints.

7 Conclusion and Future work

In this paper, the problem of checking finite satisfiability of database constraints has been investigated. In particular, this work focuses on a specific class of database constraints, namely the θ-constraints. θ-constraints, that include equality constraints, have been specified by using the language TQL^+, aimed at modeling the structural aspects and integrity constraints of ODB applications [19, 30]. In particular, in this paper, a decidable method for checking finite satisfiability of θ-constraints has been presented, and the proof of its soundness and completeness has been given.

As a future work, there seems to be no obstacles in extending the proposed method to a more expressive data model and, in particular, to a wider class of θ-constraints also including multi-valued properties, and null-values. In particular, the second direction seems to be quite easy to realize, because the presence of properties with null-values simply inhibits the expansion of the schema trees in correspondence to the arcs labeled with that properties.

Finally, the definition of a complexity class for the problem of finite satisfiability of the class of integrity constraints addressed in this paper represents an interesting and worthwhile issue to investigate, on which only preliminary results are available at the moment.

A Appendix

A.1 Formal Syntax of TQL^+

In the following definition, the formal syntax of TQL^+ is presented: 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 are optional (the underscore character stands for iteration).

Definition A.1 [**Syntax of** TQL^+]

⟨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⟩
⟨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 ⟩ ::= $\leq$ | < | > | $\geq$ | =

□

A.2 Characterization of finitely satisfiable schemas

Below the proof of the Theorem 5.2 presented in Section 5 is given.

Theorem 5.2 [**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 $\mathcal{F}(G^{eq}_{Sat}(\tau))$ has at least one connected component free of monotonic cycles.

Proof. $\Rightarrow$ By contradiction. Suppose that there exists one type γ of Σ, $I(\gamma) \neq \emptyset$, such that any connected component of $\mathcal{F}(G^{eq}_{Sat}(\gamma))$ contains at least one monotonic cycle. We have to distinguish the following two cases.

1. The graph $G^{eq}_{Sat}(\gamma)$ contains at least one monotonic cycle. According to the definition of a G^{eq}_{Sat} graph and the formal semantics of TQL^+, each arc corresponds to an association or a θ-constraint of the schema, requiring the existence of a pair of oids or values that are instances of the types labeling the connected nodes and, in the case of a θ-arc, also satisfying the θ inequality. Since the *integers* and *reals* are both totally ordered sets, any restriction of the $\leq$ (or $\geq$) relation must be a partially ordered set (i.e., the *reflexive*, *antisymmetric* and *transitive* properties must hold). In

particular, in the case of strict comparison operators, we have a strict partially ordered set (i.e., the *irreflexive*, *asymmentric*[1] and *transitive* properties must hold). Therefore, in the presence of monotonic cycles the schema is unsatisfiable: contradiction.

2. The graph $G^{eq}_{Sat}(\gamma)$ does not contain monotonic cycles.

   Assume that m is the nonnegative integer such that:

   $T^{rec}_{Path}(\gamma) = Expand^m(T_{Path}(\gamma))$.

   Given $n \geq 0$, consider the tree:

   $Expand^{m+n}(T_{Path}(\gamma))$

   and let $G^{eq,n}_{Sat}(\gamma)$ be the $G^{eq}_{Sat}(\gamma)$ graph constructed starting from the tree $Expand^{m+n}(T_{Path}(\gamma))$.

   Below the following statement will be proved:

   (*) $\forall\ n \geq 0$, any connected component of $\mathcal{F}(G^{eq,n}_{Sat}(\gamma))$ contains at least one monotonic cycle.

   By induction on n.

   - $n = 0$. This step directly follows from the hypothesis, i.e., any connected component of $\mathcal{F}(G^{eq,0}_{Sat}(\gamma))$ contains at least one monotonic cycle.
   - Suppose now that the statement (*) holds for any k, where $0 \leq k < n$. We have to prove that (*) holds for n, i.e., that any connected component of $\mathcal{F}(G^{eq,n}_{Sat}(\gamma))$ has at least one monotonic cycle. By contradiction. Suppose that there exists one connected component G'_γ of $\mathcal{F}(G^{eq,n}_{Sat}(\gamma))$ free of monotonic cycles. Consider the tree $Expand^{m+n}(T_{Path}(\gamma))$ from which $G^{eq,n}_{Sat}(\gamma)$ has been constructed. Consider a node n_i, where $e(n_i) = \{\sigma\}$, that belongs to the tree $Expand^{m+n}(T_{Path}(\gamma))$ and does not belong to the tree $Expand^{m+k}(T_{Path}(\gamma))$, $0 \leq k < n$. By definition of a T^{rec}_{Path} tree, there exists at least one node $\tilde{n}_i$ in the tree $Expand^{m+k}(T_{Path}(\gamma))$ that belongs to the path connecting the root of the tree to the node n_i, such that $e(\tilde{n}_i) = \{\sigma\}$, and $\tilde{n}_i$ has all the outgoing *property-arcs* of n_i. Therefore, if there exists one connected component G'_γ of $\mathcal{F}(G^{eq,n}_{Sat}(\gamma))$ free of monotonic cycles, it is also possible to find one connected component G''_γ of $\mathcal{F}(G^{eq,k}_{Sat}(\gamma))$, $0 \leq k < n$, free of monotonic cycles: contradiction.

     Then, the statement (*) holds for any $n \geq 0$.

   Therefore, according to the formal semantics of TQL^+, the schema is not finitely satisfiable: contradiction.

$\Leftarrow$ By construction. Given a type γ of Σ, $I(\gamma) \neq \emptyset$, let G_γ be one connected component of $\mathcal{F}(G^{eq}_{Sat}(\gamma))$ free of monotonic cycles. Let us distinguish the following two cases.

1. $G\gamma$ does not contain cycles of *θ-arcs*. For each node n_j of G_γ, let x_{n_j} be a variable standing for an instance of the type in $e(n_j)$, and let D_γ be the set of all the variables associated with the nodes of

[1] Notice that, if R is a binary relation, R is *antisymmetric* iff $(a,b) \in R$ and $(b,a) \in R \Rightarrow a = b$, whereas R is *asymmetric* iff $(a,b) \in R \Rightarrow (b,a) \notin R$.

G_γ. Furthermore, let P_γ be a function defined on the set P of *p_terms* of the schema Σ such that, if applied to a *p_term*, say p, $P_\gamma(p)$ returns the set of all the ordered pairs of variables (x_{n_i}, x_{n_j}), $x_{n_i}, x_{n_j} \in D_\gamma$, such that there exists one arc $< n_i, n_j >_p$ in G_γ. Finally, let E_γ be a function defined on the set T of TQL^+ sentences, such that:

- E_γ is an extension function over D_γ with respect to each type of Σ;
- $E_\gamma(\sigma) = \{x_{n_k} \in D_\gamma\colon\ \sigma \in e(n_k)\}$, for each type σ of Σ.

In particular, for each variable $x_{n_k} \in E_\gamma(\sigma)$, and for each property p of the type σ that is not modeled by an outgoing arc of n_k, add in D_γ one more variable, say x_h, such that, if δ is the type of the property p in σ, then $x_h \in E_\gamma(\delta)$ and $(x_{n_k}, x_h) \in P_\gamma(p)$.

Then, the triple $< D_\gamma, E_\gamma, P_\gamma >$ is an interpretation of Σ. By construction, this interpretation is also a model, i.e., for each type σ of Σ, if σ is the label of a node in G_γ, it results $E_\gamma(\sigma) \neq \emptyset$ (in fact, in G_γ any node n is an owner or an induced owner of a G^{eq}_{Sat} graph, and there are no cycles of *θ-arcs*).

2. G_γ contains a cycle of *θ-arcs*. Then, from the hypothesis, this cycle is not monotonic, i.e., all the *θ-arcs* are labeled with the $\leq$ (or $\geq$) operator. Then, suppose to collapse all the nodes of such a cycle in one single node (that is labeled with a singleton, having a schema). Similarly to the previous case, we can define a triple $< D_\gamma, E_\gamma, P_\gamma >$ that is a model of Σ.

Suppose to iterate the above construction for each *t_term* of the schema, with the assumption that, for each pair σ,δ of *t_terms* of Σ, $\sigma \neq \delta$, $D_\sigma \cap D_\delta = \emptyset$. In particular, in the case of *t_terms*, say ρ, such that $I(\rho) = \emptyset$, the construction of the triple $< D_\rho, E_\rho, P_\rho >$ is limited to the introduction of the set of variables related to the typed properties of ρ. Let L ($\subseteq T$) be the set of all the *t_terms* of the schema Σ. Then, consider the triple $< \mathcal{D}, \mathcal{E}, \mathcal{P} >$ defined as follows:

$\mathcal{D} = \bigcup_{\delta \in L} D_\delta$

$\mathcal{E}$ is an extension function, $\mathcal{E} : T \rightarrow \wp(\mathcal{D})$, over $\mathcal{D}$ with respect to each type of the schema Σ and, such that, for each type γ of Σ:

$\mathcal{E}(\gamma) = \bigcup_{\delta \in L} E_\delta(\gamma)$

$\mathcal{P}$ is a function, $\mathcal{P} : P \rightarrow \wp(\mathcal{D} \times \mathcal{D})$, such that for each $p \in P$:

$\mathcal{P}(p) = \bigcup_{\delta \in L} P_\delta(p)$.

Then, the triple $< \mathcal{D}, \mathcal{E}, \mathcal{P} >$ is an interpretation that is also a non-empty and finite model of the schema Σ: the schema is finitely satisfiable.

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

[10] 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.

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

[12] 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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

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

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

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

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.