ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

Paolo Atzeni and Val Tannen (Eds)

Database Programming Languages (DBPL-5)

Proceedings of the Fifth International Workshop on Database
Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995

Paper:

Consistency Checking in Complex Object Database Schemata with Integrity Constraints

D. Beneventano, S. Bergamaschi, S. Lodi and C. Sartori

Published in collaboration with the
British Computer Society

Consistency Checking in Complex Object Database Schemata with Integrity Constraints

D. Beneventano, S. Bergamaschi

Dipartimento di Scienze dell'Ingegneria, Università di Modena - CSITE-CNR

S. Lodi, C. Sartori

Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna - CSITE-CNR

Abstract

Integrity constraints are rules which should guarantee the integrity of a database. Provided that an adequate mechanism to express them is available, the following question arises: is there any way to populate a database which satisfies the constraints supplied by a designer? i.e., does the database schema, including constraints, admit at least one model in which all classes are non-empty? This work gives an answer to the above question in an OODB environment, providing a Data Definition Language (DDL) able to express the semantics of a relevant set of state constraints and a specialized reasoner able to check the consistency of a schema with such constraints. The choice of the set of constraints expressed in the DDL is motivated by decidability issues.

1 Introduction

Formalisms supporting classification abstraction and organization in multiple inheritance hierarchies are a milestone for database systems and knowledge representation systems, developed in Artificial Intelligence. Nevertheless, inheritance is exploited in different fashions in the two areas. OODB models deal with taxonomies of classes, described by data structures and procedures (methods), and populate them with objects, but consistency checking is done only when static type checking is supported [24]. Knowledge representation formalisms, and in particular those known as Description Logics [28], derived from the KL-ONE model [12], deal with structured classes, called *concepts*, organized in multiple inheritance taxonomies and perform consistency checking and other taxonomic inferences, but this is obtained at the cost of limiting expressiveness and disregarding any procedural aspect.

Description Logics present similarities with Complex Object Data Models (*CODM*), recently proposed in the area of Deductive Databases [2, 3] and OODBs [23, 24]. *CODMs* are, to some extent, more expressive than Description Logics of running Knowledge Representation Systems. In *CODMs* value-types and class-objects are differentiated, different type constructors are expressible (set, sequence) and cyclic descriptions are allowed. In contrast, in Description Logics the emphasis is on logical inferences, rather than on structuring the elements of the universe internally. More precisely, the central ideas of DL are the distinction between *primitive* and *defined* concepts, representing, respectively, necessary conditions and necessary and sufficient conditions,¹ computing the *subsumption* relation (the inverse of the *IS-A* relation implied by concept definitions), and checking the consistency of concept definitions. The complexity of the mentioned inferences (called *taxonomic reasoning*) was thoroughly studied for a large number of languages of the DL family (see e.g., [18]).

Recently, description logic-based formalisms for databases have been proposed in the literature to support database schema acquisition and organization, preserving consistency and minimality with respect to inheritance [6, 8, 10, 11, 14, 15]. These works aim at providing a database designer with a tool which can help him/her in the non-trivial task of complex applications design.

¹Thus, primitive concepts correspond to classes, whereas defined concepts correspond to views, in a database environment.

The design task is even more difficult if we want to specify *integrity constraints*. Most database models and systems include support for integrity constraints: in RDBMSs and OODBMSs some type constraints can be directly represented at schema definition time using the DDL; others are expressed and enforced by mechanisms such as *check conditions* and *assertions* in RDBMSs or specific *methods* in OODBMSs. However, as pointed out in [19], little attention has been paid to the problem of determining whether a set of constraints (or/and rules) is consistent, i.e., non-contradictory. Deductive databases are a notable exception: they offer a uniform solution to this problem for the wide class of integrity constraints which can be expressed as *deduction rules*, through the availability of general theorem provers, which can be used both to check the consistency of a set of data definitions and constraints and to check database states with respect to constraints. Significant contributions on these topics are found in [13, 22, 25, 26]. Our approach differs from the perspective taken in deductive database theory: the latter relies on highly general inference mechanisms, operating on a logical model, while we propose a relatively simple and complete consistency checking procedure operating on types populated by complex values. In this environment, the choice of type constructors determines the computational properties of the consistency problem.

Work on the inclusion of constraints in the schema description has been done also in [17, 30, 29] where the notion of declarative integrity constraints is introduced in Object Oriented Data models and the inference problem (i.e., whether a constraint is implied by a given set of constraints) is solved. The constraints they can declare are different from ours, since they do not include union and complement, but can express functional dependencies.

In this paper we extend the expressiveness of complex object data models, in order to capture the semantics of a considerable set of state constraints; then, we provide a decision procedure to check the consistency of a database schema. The formalism we propose allows the inclusion in class descriptions of state integrity constraints by the expression of: typing constraints, equation constraints, intersection, union and complement operators. The formalism is derived from the one proposed in [6, 9, 10]: the expressiveness has been, on the one hand, extended to include state integrity constraints, and on the other limited, disallowing cycles at schema level, due to computational problems, as will be discussed in the following.

The specialized reasoner is based on the *tableaux calculus* [27] and extends the theoretical results obtained in [4, 20]. Its behavior is the following: given a schema and one of its class descriptions (eventually including integrity constraints), it returns true if, and only if, the class is *consistent* with respect to the schema, i.e., if at least a legal instance of the database schema exists such that the class interpretation is non-empty. The reasoner can also use consistency checking to perform subsumption computation, i.e., to compute implicit *isa* relations hidden in class descriptions when views are part of a schema.

In the literature we find works related to constraint satisfiability in the areas of deductive databases and conceptual modeling.

Paper [22] is a forerunner in using the tableaux calculus technique to check constraints satisfiability in logic databases. Its static constraints are first order multityped logic sentences, restricted to finite worlds to achieve decidability.

The works descending from the SATCHMO project [13, 25, 26] face the problem of constraints satisfiability in a deductive database environment. They consider a greatly extended set of constraints, namely closed first order formulas. This set allows an inference procedure which is claimed to be satisfactorily efficient. The solution proposed in [25] follows a model-generation paradigm, since a set of constraints is satisfiable if a model, possibly empty, can be built for it; additional controls are included to find regular patterns that could give rise to infinite models. Unlike these works, we consider to be contradictory a constraint set which either does not allow any model or which forces some classes to be empty, while the algorithms in [25] should be slightly extended to individuate empty models.

On the conceptual modeling side, [5] deals with consistency and minimality of Entity/Relationship schemata, by transforming the consistency problem into a graph problem. The Entity/Relationship model is less expressive than our model, since it does not include path constraints.

Paper [15] presents a universal language which can be used to model various class-based representation formalisms, including object models, and proves that satisfiability checking is decidable. With respect to our model, the language on the one hand offers the inverse of the function represented by an attribute, number restrictions and cyclic definitions, but on the other does not represent type and path constraints, which are essential for the representation of integrity constraints. A similar comment can be made with respect to the work in [14] where the authors present a new data model which extends object oriented models in various directions, including n-ary relations, inverse of an

attribute and cardinality constraints and provide schema satisfiability procedures. It can also be observed that our tuple constructor can be used to represent n-ary relations.

The structure of the paper is the following: section 2 illustrates our approach with some informal examples. In section 3 the formalism is presented and consistency is formally defined. Section 4 gives the definitions necessary to develop the calculus, and an application example of the consistency checking algorithm, while the appendix describes the calculus in more details.

2 Some informal examples

This subsection illustrates, with some examples, the types of integrity constraints which can be expressed with the proposed formalism and the contradictions that can be found.

Let us introduce some descriptions relative to a part of the organizational structure of a Company: “Employees have a name and earn a salary. Managers are employees and have a level composed of a title and a seniority. Repositories have a denomination, which can be either a string or a structure composed by a repository name and an address; a repository stocks a set of items which are all materials. Materials are described by a name and a risk. Departments have a denomination (string), and are managed by a manager. Warehouses have both the properties of departments and repositories. Fluids are materials and have a viscosity property. Solids are materials disjointed from fluids”. Using OODB-like syntax, the Company domain example can be described as in Table 1.

type Level	=	[title: String, seniority: Integer]
class Employee	=	[name: String, salary: Integer]
class Manager	=	isa Employee and [level: Level]
class Repository	=	[denomination: String or [rname: String, address: String], stock: {Material}]
class Department	=	[denomination: String, manager: Manager]
class Warehouse	=	isa Department and Repository
class Material	=	[name: String, risk: Integer]
class Fluid	=	isa Material and [viscosity: Integer]
class Solid	=	isa Material and not Fluid

Table 1: The Company domain schema in OODB-like syntax

Note that the conjunction operator (**and**) allows the expression of *multiple inheritance* (see class Warehouse). In particular, a subclass can redefine a field inherited from one (or more) its superclass. In this case, due to our semantics, we compute the resulting type of the field as the conjunction of the given types. For example, in the class Manager we could redefine the field salary as a subrange of Integer. The complement operator (**not**) allows *exclusive* specialization hierarchies to be expressed (see classes Fluid and Solid). Furthermore, in class Repository the disjunction operator (**or**) describes two alternative structures for the attribute denomination.

A remarkable capability of our formalism is attribute path support. Attribute paths are dot-separated sequences of attributes expressing navigation through classes and types of a complex object data schema. They are necessary for query languages in an OODBMS environment (see the examples in [21]) and were introduced to express constraints in [17]. We express integrity constraints by comparing an attribute path with a base value or with another attribute path. For example, let us add to the schema the following class descriptions: “Dangerous materials (DangerMaterial) are materials with a risk greater than 5. Top managers (TopManager) are managers with a parameter greater than 10. Technicians are employees who work in a department; they earn less than the manager of the department where they work”.

```

class DangerMaterial = isa Material and (risk > 5)
class TopManager    = isa Manager and (level.seniority > 10)
class Technician    = isa Employee and [works: Department]
                        and (salary < works.manager.salary)
    
```

The logical operators and, or and not allow the expression of notable integrity constraints, that is, logical implications. For example, let us introduce the class Shipment:

```

class Shipment = [urgency: Integer, item: Material]
    
```

and specify for it the following integrity constraint: “**for all** shipments it must hold that **if** (`item.risk > 3`), i.e., the risk of the material is greater than 3, **then** (`urgency > 10`), i.e., its urgency must be greater than 10”. By translating implication, we can integrate the constraint in the class description, obtaining the following description for Shipment:

```

class Shipment = [urgency: Integer, item: Material]
                  and (not(item.risk > 3) or (urgency > 10))
    
```

The above examples show how class structures and integrity constraints can be represented in a unified framework, by embedding integrity constraints in class descriptions. Consequently, by assigning a precise semantics to the formalism constructors, this framework can be the basis to perform a global consistency checking.

Let us examine some examples of inconsistency. The following three classes show inconsistency due to type conflicts between inherited attributes and locally redefined attributes:

```

class Storage = isa Warehouse
                and [denomination: [ rname: String,
                                       address: String]]
class AWarehouse = isa Warehouse and [stock: {Fluid}]
class BWarehouse = isa AWarehouse and [stock: {exists Solid}]
    
```

Class `Storage` inherits from `Warehouse` a string attribute `denomination` and redefines it as a tuple. Class `AWarehouse` stocks a set of objects which belong all to the class `Fluid`. Class `BWarehouse` stocks at least a `Solid` (this is expressed by the existentially quantified set operator exists), but this is in conflict with the definition of its superclass `AWarehouse`, which requires `Fluid` only in its stock (remember that `Fluid` and `Solid` are disjointed).

The following examples show how less intuitive contradictions can occur:

```

class BossTechnician = isa Technician
                      and (self = works.manager)
class SlowDangerShipment = isa Shipment
                          and [item: DangerMaterial]
                          and (urgency < 5)
    
```

The class `BossTechnician` exploits paths and the *self* reference, and describes technicians who work in the departments they manage. Note that, due to the equality predicate (self = `works.manager`), we have that each object of the class `BossTechnician` must be an object of the class `Manager`. Furthermore, on the basis of the constraint expressed on the superclass `Technician`, we obtain that a `BossTechnician` earn less than himself. On the other hand, if we modify the `BossTechnician` description as follows:

```

class BossTechnician = isa Technician
                      and (self.name = works.manager.name)
    
```

we describe technicians who work in the departments managed by manager with the same name and no contradiction occurs.

Class `SlowDangerShipment` is inconsistent since `DangerMaterial` is constrained to a risk value greater than 5 and therefore its shipment requires an urgency value greater than 10.

Finally, we can express a set of database *views* by introducing the semantics of *virtual classes*, that is classes whose descriptions represent a set of necessary and sufficient conditions [1, 11], and we can check their consistency. For example:

$$\begin{aligned} \text{virtual class SuperFluid} &= \text{isa Fluid and (viscosity} < 10) \\ \text{virtual class TopFluid} &= \text{isa Fluid and not SuperFluid} \\ &\quad \text{and (viscosity} < 3) \end{aligned}$$

the virtual class TopFluid is detected as inconsistent; note that if SuperFluid were a class, instead of a virtual class, no inconsistency would be detected but an exclusive specialization hierarchy would be expressed.

3 A formalism for Complex Object Databases including integrity constraints

This section introduces the syntax and interpretation of complex objects and constraints. The formalism is made up of a *base type system*, *type names* and *type descriptions*, *path specifications* and *comparison operators* for the description of *predicates*. Then, the notion of consistent type and subsumption relation are introduced.

Values and Objects

We assume the union of the integers, the strings, the booleans, and the reals as the set $\mathcal{D}$ of *base values*. To build *complex values*, we further assume two disjoint countable sets of symbols $\mathbf{A}$ and $\mathcal{O}$, respectively called the *attributes* (denoted by $a_1, a_2, \dots$), and the *object identifiers* (denoted by $o, o', \dots$), both disjoint from $\mathcal{D}$. The set $\mathcal{V}(\mathcal{O})$ of all *values over* $\mathcal{O}$ is defined as the smallest set containing $\mathcal{D}$ and $\mathcal{O}$, such that, if $v_1, \dots, v_p$ are values over $\mathcal{O}$, then the set $\{v_1, \dots, v_p\}$ is a value over $\mathcal{O}$, and a partial function $t: \mathbf{A} \rightarrow \{v_1, \dots, v_p\}$ is a value over $\mathcal{O}$. The standard notation $[a_1: v_1, \dots, a_p: v_p]$ will be henceforth used. Object identifiers are assigned values by a *total value function* δ from $\mathcal{O}$ to $\mathcal{V}(\mathcal{O})$.

Let $=, \neq, >, <, \geq, \leq$ be the equality, inequality and total order relations, denoted by θ , defined as usual on $\mathcal{D}$. Equality and inequality can be extended from $\mathcal{D}$ to all $\mathcal{V}(\mathcal{O})$: the equality operator ($=$) has the meaning of *identity*, i.e., two objects are equal if they have the same identifier; two sets are equal iff they equal elements; two tuples, say $v_a = [a_1: v_1, \dots, a_p: v_p]$ and $v_b = [a'_1: v'_1, \dots, a'_q: v'_q]$, are equal if they have equal attribute labels and equal attribute labels are mapped to equal values.

Paths

A *path* p is either the symbol ϵ , or a dot-separated sequence of elements $e_1 . e_2 . \dots . e_n$, where $e_i \in \mathbf{A} \cup \{\Delta\}$ ($i = 1, \dots, n$). As usual, we use the dot notation to indicate single attributes, but, in addition, the sequence of attributes can contain the symbol Δ . We need this symbol to distinguish between an attribute of the tuple value associated to an object from an attribute of a tuple value. As an example, in $\Delta.\text{level}.\text{seniority}$, $\Delta.\text{level}$ refers to the attribute `level` of an object having a tuple value, while `seniority` refers to an attribute of a tuple. By ϵ we denote the unique path of length 0 (the *empty path*), corresponding to the *self* reference, and by $\mathbf{W}$ we denote the set of all paths. Given a set of object identifiers $\mathcal{O}$ and a value function δ , we introduce the function $\mathcal{J}: \mathbf{W} \rightarrow 2^{\mathcal{V} \times \mathcal{V}}$, associating a function on $\mathcal{V}$ to every path:

- empty path: $\mathcal{J}[\epsilon] = \{(v, v) \in \mathcal{V} \times \mathcal{V}\}$
- single element path: $\mathcal{J}[a] = \{(v_1, v_2) \in \mathcal{V} \times \mathcal{V} \mid v_1 = [\dots, a: v_2, \dots]\}$
 $\mathcal{J}[\Delta] = \{(o, v) \in \mathcal{O} \times \mathcal{V} \mid \delta(o) = v\}$
- multiple element path: $\mathcal{J}[e_1 . e_2 . \dots . e_{n-1} . e_n] = \mathcal{J}[e_1] \circ \mathcal{J}[e_2] \circ \dots \circ \mathcal{J}[e_{n-1}] \circ \mathcal{J}[e_n]$, where $\circ$ is the symbol of function composition.

Let v be a value and p be a path. By $\mathcal{J}[p](v)$ we mean the set of values reachable from v following p , that is $\mathcal{J}[p](v) = \{v' \mid (v, v') \in \mathcal{J}[p]\}$; note that $\mathcal{J}[p](v)$ has cardinality ≤ 1 .

Types and Schemata

We assume a countable set $\mathbf{N}$ of type names (denoted by N, N'), which includes the set $\mathbf{B} = \{\text{Integer}, \text{String}, \text{Bool}, \text{Real}\}$ of base-type designators (denoted by $B, B', \dots$) and the symbols $\top, \perp$.

The set of all *finite type descriptions* $(S, S', \dots)$, also briefly called *types*, over given $\mathbf{A}, \mathbf{N}$ is denoted by $\mathbf{S}(\mathbf{A}, \mathbf{N})$ and is obtained according to the following abstract syntax rule ²:

$$\begin{aligned} S \rightarrow & N \mid S \sqcup S' \mid S \sqcap S' \mid \neg S \\ & \mid \{S\}_{\forall} \mid \{S\}_{\exists} \mid [a_1 : S_1, \dots, a_k : S_k] \mid \Delta S \\ & \mid p\theta p' \mid p\theta d \mid p\uparrow \end{aligned}$$

$\top$ denotes the *top type*, $\perp$ denotes the *empty type*, $\{\}_{\forall}$ and $\{\}_{\exists}$ denote the usual type constructors of set and tuple, respectively. The $\{S\}_{\exists}$ construct is an existential set specification, where at least one element of the set must be of type S . The construct $\sqcap$ holds for *conjunction*, the construct $\sqcup$ holds for *union*, the construct $\neg$ holds for *complement*, whereas Δ is an object set forming constructor. $p\theta p'$, $p\theta d$, $p\uparrow$ represent *atomic predicates*: $p\theta p'$ is a *path equation*, $p\theta d$ is a *range restriction* and $p\uparrow$ expresses *path undefinedness*.

Given a set of type descriptions $\mathbf{S}(\mathbf{A}, \mathbf{N})$, a *schema* σ over $\mathbf{S}(\mathbf{A}, \mathbf{N})$ is a total function $\sigma: \mathbf{N} \setminus (\mathbf{B} \cup \{\top, \perp\}) \rightarrow \mathbf{S}$, which associates type names to descriptions. σ is partitioned into two functions: σ_P , which introduces the description of *primitive types*, and σ_V , which introduces the description of *virtual types*. Some of the types introduced in the examples are mapped below into the formalism:

$$\begin{aligned} & \vdots \\ \sigma_V(\text{Level}) &= [\text{title: String, seniority: Integer}] \\ \sigma_P(\text{Material}) &= \Delta[\text{name: String, risk: Integer}] \\ \sigma_P(\text{Fluid}) &= \text{Material} \sqcap \Delta[\text{viscosity: Integer}] \\ \sigma_P(\text{Solid}) &= \text{Material} \sqcap \neg \text{Fluid} \\ \sigma_P(\text{TopManager}) &= \text{Manager} \sqcap (\Delta.\text{level}.\text{seniority} > 10) \\ \sigma_P(\text{BossTechnician}) &= \text{Technician} \sqcap (\epsilon = \Delta.\text{works}.\Delta.\text{manager}) \\ \sigma_P(\text{AWarehouse}) &= \text{Warehouse} \sqcap \Delta[\text{stock: \{Fluid\}_{\forall}}] \\ \sigma_P(\text{BWarehouse}) &= \text{AWarehouse} \sqcap \Delta[\text{stock: \{Solid\}_{\exists}}] \\ \sigma_V(\text{SuperFluid}) &= \text{Fluid} \sqcap (\Delta.\text{viscosity} < 10) \end{aligned}$$

Observe that the Δ operator allows the distinction of object classes from value types: *class descriptions* are preceded by Δ (as in the case of `Material`), while descriptions without Δ are related to value-type names (as in the case of `Level`).

We require that a schema be *acyclic*: a type name which is defined using other type names must describe finitely nested values. Formally, cycles are recognized by means of the notion of *dependence*: N *depends on* N' , where $N, N' \in \mathbf{N}$ if N' is contained in the expression defining N , $\sigma(N)$; a schema is *acyclic* iff the transitive closure of *depends on* is a strict partial order.

Inheritance is expressed by conjunction ($\sqcap$) in a type or class definition. We say that N *inherits directly from* N' , written $N \dashv N'$, iff $\sigma(N) = S_1 \sqcap \dots \sqcap S_n$, $n > 0$, and $N' = S_i$ for some i , $1 \leq i \leq n$. As a matter of fact, an acyclic schema is *inheritance well-founded*, i.e., the transitive closure of $\dashv$ is a strict partial order.

Interpretations and Database Instances

Let $\mathcal{I}_{\mathbf{B}}$ be the (fixed) standard interpretation function from $\mathbf{B}$ to $2^{\mathcal{D}}$. For a given object assignment δ , each type expression S is mapped to a set of values (its interpretation). An *interpretation function* is a function $\mathcal{I}$ from $\mathbf{S}$ to $2^{\mathcal{V}(\mathcal{O})}$ satisfying the following equations:

²In the following, we will write $\mathbf{S}$ instead of $\mathbf{S}(\mathbf{A}, \mathbf{N})$ when the components are obvious from the context.

$$\begin{aligned}
 \mathcal{I}[\top] &= \mathcal{V}(\mathcal{O}), \quad \mathcal{I}[\perp] = \emptyset, \quad \mathcal{I}[B] = \mathcal{I}_B[B], \\
 \mathcal{I}[S \sqcap S'] &= \mathcal{I}[S] \cap \mathcal{I}[S'], \quad \mathcal{I}[S \sqcup S'] = \mathcal{I}[S] \cup \mathcal{I}[S'], \quad \mathcal{I}[\neg S] = \mathcal{V}(\mathcal{O}) \setminus \mathcal{I}[S], \\
 \mathcal{I}[\{S\}_{\vee}] &= \{M \mid M \subseteq \mathcal{I}[S]\} \\
 \mathcal{I}[\{S\}_{\exists}] &= \{M \mid M \cap \mathcal{I}[S] \neq \emptyset\} \\
 \mathcal{I}[[a_1 : S_1, \dots, a_p : S_p]] &= \{t : \mathbf{A} \rightarrow \mathcal{V}(\mathcal{O}) \mid t(a_i) \in \mathcal{I}[S_i], 1 \leq i \leq p\} \\
 \mathcal{I}[\Delta S] &= \{o \in \mathcal{O} \mid \delta(o) \in \mathcal{I}[S]\} \\
 \mathcal{I}[(p\theta d)] &= \{v \in \mathcal{V}(\mathcal{O}) \mid \mathcal{J}[p](v) \cap \{d' \mid d'\theta d\} \neq \emptyset\} \\
 \mathcal{I}[(p_1\theta p_2)] &= \{v \in \mathcal{V}(\mathcal{O}) \mid (\mathcal{J}[p_1](v) \times \mathcal{J}[p_2](v)) \cap \\
 &\quad \{(d_1, d_2) \mid d_1\theta d_2\} \neq \emptyset\} \\
 \mathcal{I}[(p\uparrow)] &= \{v \in \mathcal{V}(\mathcal{O}) \mid \mathcal{J}[p](v) = \emptyset\}
 \end{aligned}$$

We say that $\mathcal{I}$ is a *legal instance* of a schema σ iff the set $\mathcal{O}$ is *finite*, and for all $N \in \mathbf{N}$:

$$\mathcal{I}[N] \subseteq \mathcal{I}[\sigma_P(N)] \text{ if } N \in \text{dom } \sigma_P, \quad \mathcal{I}[N] = \mathcal{I}[\sigma_V(N)] \text{ if } N \in \text{dom } \sigma_V$$

From this definition, we see that the interpretation of a primitive type *is included* in the interpretation of its description, i.e., has to be provided by the user, according to the given description, while the interpretation of a virtual type *is* the interpretation of its description, i.e., every object whose description fits those of virtual types is automatically recognized as an element of these virtual types. Since the schema is acyclic, the legal instance is uniquely determined by the interpretations of primitive types [6].

As usual, the notion of consistency is embodied by the possibility of filling a type with at least one value for some legal instance of the schema. In terms of the interpretation function, given a schema σ over $\mathbf{S}(\mathbf{A}, \mathbf{N})$, a type $S \in \mathbf{S}$ is *consistent* if and only if there is a legal instance $\mathcal{I}$ of σ , such that $\mathcal{I}[S] \neq \emptyset$. A schema σ over $\mathbf{S}(\mathbf{A}, \mathbf{N})$ is called *consistent* iff for all $N \in \mathbf{N}$, N is consistent.

Finally, given two types S, S' of a schema σ , we say that S *subsumes* S' iff $\mathcal{I}[S] \supseteq \mathcal{I}[S']$ for all legal instances $\mathcal{I}$ of σ . Consistency and subsumption can be reduced to each other, according to the following rules: S is subsumed by S' iff $S \sqcap \neg S'$ is inconsistent, and S is consistent iff S is not subsumed by $\perp$. Therefore, given an algorithm for checking the consistency of a type, it can also be used for subsumption computation.

4 Checking consistency

Our algorithm for consistency checking is inspired by the rules known as *tableaux calculus* for first order logic. This technique has already been successfully applied to similar problems in *DLs* [20]. The algorithm performs three steps:

1. Transformation of a schema into a *negation normal form*, that is a form where negations of complex structures are eliminated. The transformation results from the application of a set of type rewriting rules which preserve the interpretation of type descriptions. This also implies that the consistency property of the original schema is preserved.
2. Generation for each type of the negation normal schema of a *constraint system*³, which is a set of associations between variables and types; the constraint system is generated by a set of terminating *completion rules*.
3. Examination of the constraint system to determine if it contains a *clash*, which is a set of clearly unsatisfiable constraints (it can be proved that the presence of a clash is equivalent to the inconsistency of the type).

In the following we give the definitions necessary to develop the calculus, and an application example of the consistency checking algorithm. The complete set of rules used in steps 1, 2 is given in the appendix; in this section, we need only present the rules which are relevant to the example.

We assume that an alphabet of variable symbols exists, which will be denoted by the letters x, y and z . A *constraint* is a syntactic object of the form $x/S, xpy, x\exists y, x\theta y, x\theta d$, where S is a negation normal type, p is a path, $\theta \in \{=, \neq, <, >\}$ and d is a base value. Let $\mathcal{I}$ be a legal instance. An *$\mathcal{I}$ -assignment* is a function α that maps every

³Not to be confused with the set of integrity constraints which are included in the schema definition. In the remainder of this section the term *constraint* will indicate an element of a constraint system.

variable to an element of $\mathcal{V}(\mathcal{O})$. We say that: α satisfies x/S , if $\alpha(x) \in \mathcal{I}[S]$; α satisfies xpy , if $(\alpha(x), \alpha(y)) \in \mathcal{J}[p]$; α satisfies $x\exists y$, if $\alpha(x) = \{\dots, v, \dots\}$, and $\alpha(y) = v$; α satisfies $x\theta y$, if $\alpha(x)\theta\alpha(y)$ is true; α satisfies $x\theta d$, if $\alpha(x)\theta d$ is true. A constraint c is *satisfiable* if there is a legal instance $\mathcal{I}$ and an $\mathcal{I}$ -assignment α such that α satisfies c . A constraint system $\mathbf{K}$ is a finite set of constraints. An $\mathcal{I}$ -assignment α satisfies a constraint system $\mathbf{K}$ if α satisfies every constraint in $\mathbf{K}$. A constraint system $\mathbf{K}$ is *satisfiable* if there is a legal instance $\mathcal{I}$ and an $\mathcal{I}$ -assignment α such that α satisfies $\mathbf{K}$.

Let us now consider the satisfiability for the type associated to the class `BossTechnician`:

$$\begin{aligned}\bar{S} &= (\epsilon = \Delta.\text{works}.\Delta.\text{manager}) \sqcap \\ &\quad (\Delta.\text{salary} < \Delta.\text{works}.\Delta.\text{manager}.\Delta.\text{salary})\end{aligned}$$

The calculus starts with the constraint system $\mathbf{K} := \{x/\bar{S}\}$ and adds constraints to $\mathbf{K}$ by means of the completion rules, until it is *complete*, i.e., until no rule is applicable. The resulting constraint system is called a *completion* of $\mathbf{K}$. The completion rules needed to compute the satisfiability of $\bar{S}$ are the following (where $\mathbf{K}_y^z$ denotes the constraint system obtained from $\mathbf{K}$ by replacing each occurrence of y by z):

$$\begin{aligned}\mathbf{K} &\rightarrow_{\sqcap} \{x/S_1, x/S_2\} \cup \mathbf{K} \quad \text{if } x/S_1 \sqcap S_2 \in \mathbf{K}, \text{ and } x/S_1 \text{ and } x/S_2 \text{ are} \\ &\quad \text{not both in } \mathbf{K} \\ \mathbf{K} &\rightarrow_{\theta} \{xp_1y, xp_2z, (y\theta z)\} \cup \mathbf{K} \quad \text{if } x/(p_1\theta p_2) \in \mathbf{K}, \text{ and there are no} \\ &\quad \text{variables } y', z' \text{ such that } xp_1y', \\ &\quad xp_2z' \text{ and } (y'\theta z') \text{ are in } \mathbf{K}, \text{ and } y \\ &\quad \text{and } z \text{ are new variables} \\ \mathbf{K} &\rightarrow_{\text{path}} \{xez, zpy\} \cup \mathbf{K} \quad \text{if } xe.py \in \mathbf{K}, \text{ there is no variable } z' \text{ such} \\ &\quad \text{that } xez' \text{ and } z'py \text{ are in } \mathbf{K}, \text{ and } z \text{ is a new} \\ &\quad \text{variable} \\ \mathbf{K} &\rightarrow_{\text{replace}} \mathbf{K}_y^z \quad \text{if } y = z, \text{ or } y\epsilon z, \text{ or } xey \text{ and } xez, \text{ with } e \in \mathbf{A} \cup \{\Delta\}, \\ &\quad \text{are in } \mathbf{K}, \text{ and } y \neq z.\end{aligned}$$

Let $\mathbf{K} = \{x/\bar{S}\}$ be the initial constraint system; a sequence of applications of the completion rules to $\mathbf{K}$ is as follows:

$$\begin{aligned}\mathbf{K} &:= \mathbf{K} \cup \{x/(\Delta.\text{salary} < \Delta.\text{works}.\Delta.\text{manager}.\Delta.\text{salary}) \\ &\quad x/(\epsilon = \Delta.\text{works}.\Delta.\text{manager}))\} \quad (\rightarrow_{\sqcap}\text{-rule}) \\ \mathbf{K} &:= \mathbf{K} \cup \{x\epsilon x_1, x\Delta.\text{works}.\Delta.\text{manager}x_2, x_1 = x_2\} \quad (\rightarrow_{\theta}\text{-rule}) \\ \mathbf{K} &:= \mathbf{K} \cup \{x\Delta.\text{salary}x_3, x\Delta.\text{works}.\Delta.\text{manager}.\Delta.\text{salary}x_4, \\ &\quad x_3 < x_4\} \quad (\rightarrow_{\theta}\text{-rule}) \\ \mathbf{K} &:= \mathbf{K} \cup \{x\Delta x_5, x_5\text{works}x_6, x_6\Delta x_7, x_7\text{manager}x_2, x\Delta x_8, x_8\text{salary}x_3, \\ &\quad x\Delta x_9, x_9\text{works}x_{10}, x_{10}\Delta x_{11}, x_{11}\text{manager}x_{12}, x_{12}\Delta x_{13}, \\ &\quad x_{13}\text{salary}x_4\} \quad (\text{recursive application of } \rightarrow_{\text{path}}\text{-rule}) \\ \mathbf{K} &:= \mathbf{K}_{x_1}^x = \{x\epsilon x, x = x_2, \dots\} \quad (\rightarrow_{\text{replace}}\text{-rule due to } x\epsilon x_1) \\ \mathbf{K} &:= \mathbf{K}_{x_2}^x = \{x = x, x_7\text{manager}x, \dots\} \quad (\rightarrow_{\text{replace}}\text{-rule due to } x = x_2) \\ \mathbf{K} &:= \mathbf{K}_{x_8}^{x_5} = \{x\Delta x_5, x_5\text{salary}x_3, \dots\} \quad (\rightarrow_{\text{replace}}\text{-rule due to } x\Delta x_8, x\Delta x_5) \\ \mathbf{K} &:= \mathbf{K}_{x_9}^{x_5} = \{x\Delta x_5, x_5\text{works}x_{10}, \dots\} \quad (\rightarrow_{\text{replace}}\text{-rule due to } x\Delta x_9, x\Delta x_5) \\ \mathbf{K} &:= \mathbf{K}_{x_{10}}^{x_5} = \{x_5\text{works}x_6, x_6\Delta x_{11}, \dots\} \\ &\quad (\rightarrow_{\text{replace}}\text{-rule due to } x_5\text{works}x_6, x_5\text{works}x_{10}) \\ \mathbf{K} &:= \mathbf{K}_{x_{11}}^{x_7} = \{x_6\Delta x_7, x_7\text{manager}x_{12}, \dots\} \\ &\quad (\rightarrow_{\text{replace}}\text{-rule due to } x_6\Delta x_{11}, x_6\Delta x_7) \\ \mathbf{K} &:= \mathbf{K}_{x_{12}}^x = \{x_7\text{manager}x, x\Delta x_{13}, \dots\} \\ &\quad (\rightarrow_{\text{replace}}\text{-rule due to } x_7\text{manager}x_{12}, x_7\text{manager}x) \\ \mathbf{K} &:= \mathbf{K}_{x_{13}}^{x_5} = \{x\Delta x_5, x_5\text{salary}x_4, \dots\} \quad (\rightarrow_{\text{replace}}\text{-rule due to } x\Delta x_{13}, x\Delta x_5) \\ \mathbf{K} &:= \mathbf{K}_{x_4}^{x_3} = \{x_5\text{salary}x_3, x_3 < x_3, \dots\} \\ &\quad (\rightarrow_{\text{replace}}\text{-rule due to } x_5\text{salary}x_4, x_5\text{salary}x_3)\end{aligned}$$

One can verify that no rule can be applied to $\mathbf{K}_{x_4}^{x_3}$; since it contains the constraint $x_3 < x_3$, which is obviously unsatisfiable (namely it is a clash), the type $\bar{S}$ is also unsatisfiable.

The calculus (whose complete description can be found in the appendix) has been proved to be terminating, sound and complete in [7].

Theorem 1 (Termination) *There is no infinite sequence of constraint systems $K_1 = \{x/S\}$, $K_2, \dots$, such that $K_i \rightarrow_* K_{i+1}$, for $i \geq 1$, is an instance of a completion rule.*

Theorem 2 *S is consistent iff some completion K of $\{x/S\}$ contains no clash.*

5 Final remarks

The guideline of the present work was to develop a complete consistency checking algorithm, thus we had to limit the formalism expressiveness. In fact, we proved in [7] that two significant extensions, namely, allowing cyclic type descriptions and paths including multi-valued attributes, lead to undecidability. In order to extend the formalism expressiveness we plan to allow some kinds of incompleteness. Future work will be devoted to explore the best tradeoff between expressiveness and incompleteness.

References

- [1] S. Abiteboul and A. Bonner. Objects and views. In *SIGMOD*, pages 238–247. ACM Press, 1991.
- [2] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *SIGMOD*, pages 159–173. ACM Press, 1989.
- [3] P. Atzeni, editor. *LOGIDATA⁺: Deductive Databases with Complex Objects*. Springer-Verlag: LNCS n. 701, Heidelberg - Germany, 1993.
- [4] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *12th International Joint Conference on Artificial Intelligence.*, Sydney, Australia, 1991.
- [5] G. Di Battista and M. Lenzerini. Deductive entity relationship modeling. *IEEE Trans. on Knowledge and Data Engineering*, 5(3):439–450, June 1993.
- [6] D. Beneventano and S. Bergamaschi. Subsumption for complex object data models. In J. Biskup and R. Hull, editors, *4th Int. Conf. on Database Theory - Berlin*, pages 357–375, Heidelberg, Germany, October 1992. Springer-Verlag.
- [7] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Consistency checking in complex object database schemata with integrity constraints. Technical Report 103, CIOC–CNR, Viale Risorgimento, 2 Bologna, Italia, September 1994.
- [8] D. Beneventano, S. Bergamaschi, and C. Sartori. Taxonomic reasoning with cycles in LOGIDATA⁺. In P. Atzeni, editor, *LOGIDATA⁺: Deductive Databases with Complex Objects*, page 105:128. Springer-Verlag: LNCS n. 701, Heidelberg - Germany, 1993.
- [9] S. Bergamaschi and B. Nebel. Theoretical foundations of complex object data models. Technical Report 5/91, CNR, Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Roma, January 1991.
- [10] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.
- [11] S. Bergamaschi and C. Sartori. On taxonomic reasoning in conceptual design. *ACM Transactions on Database Systems*, 17(3):385–422, September 1992.
- [12] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.
- [13] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In H.W. Schmidt, S. Ceri, and M. Missikoff, editors, *EDBT '88 - Advances in Database Technology - Venezia, Italy*, pages 488–505, Heidelberg, Germany, March 1988. Springer-Verlag.
- [14] D. Calvanese and M. Lenzerini. Making object-oriented schemas more expressive. In *PODS - Principles of Database Systems*. ACM Press, 1994.

- [15] D. Calvanese, M. Lenzerini, and D. Nardi. A unified framework for class-based formalisms. In *KR '94 - Int. Conf on Principles of Knowledge Representation and Reasoning*, pages 109–120, Cambridge - MA, 1994. Morgan Kaufmann Publishers, Inc.
- [16] T.Y. Clement and W. Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):362–375, September 1989.
- [17] N. Coburn and G.E. Weddel. Path constraints for graph-based data models: Towards a unified theory of typing constraints, equations and functional dependencies. In *2nd Int. Conf. on Deductive and Object-Oriented Databases*, pages 312–331, Heidelberg, Germany, December 1991. Springer-Verlag.
- [18] F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR '91 - 2nd Int. Conf on Principles of Knowledge Representation and Reasoning*, pages 151–162, Cambridge - MA, April 1991. Morgan Kaufmann Publishers, Inc.
- [19] H. Gallaire and J.M. Nicholas. Logic and databases: An assessment. In S. Abiteboul and P. Kanellakis, editors, *ICDT 90*, pages 177–186, Heidelberg, Germany, December 1990. Springer-Verlag.
- [20] B. Hollunder, W. Nutt, and M. Schmidt-Schauss. Subsumption algorithms for concept description languages. In *9th ECAI*, pages 348–353, Stockholm, Sweden, 1990.
- [21] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *SIGMOD '92*, pages 393–402. ACM, June 1992.
- [22] C. Kung. A tableaux approach for consistency checking. In *IFIP WG 8.1 Conf. on Theoretical and Formal Aspects of Information Systems*, Sitges, Spain, April 1985.
- [23] C. Lecluse and P. Richard. Modelling complex structures in object-oriented databases. In *Symp. on Principles of Database Systems*, pages 362–369, Philadelphia, PA, 1989.
- [24] C. Lecluse and P. Richard. The O₂ database programming language. In *15th Int. Conf. on Very Large Databases*, pages 411–422, Amsterdam, February 1989.
- [25] R. Manthey. Satisfiability of integrity constraints: Reflections on a neglected problem. In *Int. Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1990.
- [26] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in prolog. In *CADE 88 - 9th Conference on Automated Deduction*. Springer-Verlag - LNCS, 1988. ECRC Tech. Rep. KB-21, Nov. 87.
- [27] R.M. Smullyan. *First-Order Logic*. Springer-Verlag, Berlin, Germany, 1986.
- [28] J. Sowa, editor. *Principles of Semantic Networks*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.
- [29] M.F. van Bommel and G.E. Weddel. Reasoning about equations and functional dependencies on complex objects. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):455–469, June 1994.
- [30] Grant E. Weddel. Reasoning about functional dependencies generalized for semantic data models. *ACM Transactions on Database Systems*, 17(1):32–64, March 1992.

Appendix: Normalization and completion rules

We have succinctly given the structure of the consistency checking algorithm in section 4. Here we present all the rules of the calculus. The normalization rewriting rules are the following:

$$\neg \top \longrightarrow \perp \quad (1)$$

$$\neg \perp \longrightarrow \top \quad (2)$$

$$\neg(S \sqcap S') \longrightarrow \neg S \sqcup \neg S' \quad (3)$$

$$\neg(S \sqcup S') \longrightarrow \neg S \sqcap \neg S' \quad (4)$$

$$\neg \neg S \longrightarrow S \quad (5)$$

$$[a_1 : S_1, \dots, a_p : S_p] \longrightarrow [a_1 : S_1] \sqcap \dots \sqcap [a_p : S_p] \quad (6)$$

$$\neg[a_1 : S_1, \dots, a_p : S_p] \longrightarrow a_1 \uparrow \sqcup \dots \sqcup a_p \uparrow \sqcup [a_1 : \neg S_1] \sqcup \dots \sqcup [a_p : \neg S_p] \quad (7)$$

$$\neg \Delta S \longrightarrow (\Delta \uparrow) \sqcup \Delta \neg S \quad (8)$$

$$\neg \{S\}_{\exists} \longrightarrow \neg \{\top\}_{\forall} \sqcup \{\neg S\}_{\forall} \quad (9)$$

$$\neg \{S\}_{\forall} \longrightarrow \neg \{\top\}_{\forall} \sqcup \{\neg S\}_{\exists} \text{ if } S \neq \top \quad (10)$$

$$\neg(p_1 \theta p_2) \longrightarrow (p_1 \uparrow) \sqcup (p_2 \uparrow) \sqcup (p_1 \bar{\theta} p_2) \quad (11)$$

$$\neg(p \theta d) \longrightarrow p \uparrow \sqcup (p \bar{\theta} d) \quad (12)$$

$$(e \cdot p \uparrow) \longrightarrow (e \uparrow) \sqcup \begin{cases} [a : (p \uparrow)], & \text{if } e = a \in \mathbf{A} \\ \Delta(p \uparrow), & \text{if } e = \Delta \end{cases} \quad (13)$$

$$\neg(e \uparrow) \longrightarrow \begin{cases} [a : \top], & \text{if } e = a \in \mathbf{A} \\ \Delta \top, & \text{if } e = \Delta \end{cases} \quad (14)$$

$$(\epsilon \uparrow) \longrightarrow \perp \quad (15)$$

$$p_1 \leq p_2 \longrightarrow p_1 < p_2 \sqcup p_1 = p_2 \quad (16)$$

$$p_1 \geq p_2 \longrightarrow p_1 > p_2 \sqcup p_1 = p_2 \quad (17)$$

$$\{S\}_{\forall} \longrightarrow \{\perp\}_{\forall} \sqcup (\{S\}_{\forall} \sqcap \{\top\}_{\exists}) \quad (18)$$

The schema resulting from the application of the above transformations to σ will be denoted by $\bar{\sigma}$. It is easy to see that the transformations above preserve the interpretation of type descriptions. This implies that also the consistency property on the original schema is preserved. The basic definitions (constraint system and satisfiability of a constraint system) have already been given in section 4. We add here the definition of *simple constraint system*:

Definition 1 A constraint system is called *simple* if it contains only constraints of the form

1. x/B
2. $x\theta y, \theta \in \{<, =, >\}$
3. $x\theta d, \theta \in \{<, =, >\}$
4. $x/\neg B$, where x occurs in at least one constraint of the forms 1, 2, 3.
5. $x \neq y$, where x and y occur in at least one constraint of the forms 1, 2, 3.

The satisfiability of a simple constraint system has already been dealt with in the literature (see, for example, [16]).

We consider now the satisfiability for a generic type S . The calculus starts with a constraint system $\mathbf{K} = \{x/S\}$ and adds constraints to $\mathbf{K}$ until either a contradiction is generated or an interpretation satisfying S can be obtained from the resulting system. With $\mathbf{K}_y^z$ we denote the constraint system obtained from $\mathbf{K}$ by replacing each occurrence of y by z . The calculus is given by the following *completion rules*:

1. $\mathbf{K} \longrightarrow_{\sqcup} \{x/S_3\} \cup \mathbf{K}$ if $x/S_1 \sqcup S_2 \in \mathbf{K}$, neither x/S_1 nor x/S_2 is in $\mathbf{K}$, and $S_3 = S_1$ or $S_3 = S_2$
2. $\mathbf{K} \longrightarrow_{\sqcap} \{x/S_1, x/S_2\} \cup \mathbf{K}$ if $x/S_1 \sqcap S_2 \in \mathbf{K}$, and x/S_1 and x/S_2 are not both in $\mathbf{K}$
3. $\mathbf{K} \longrightarrow_{\exists} \{x\exists y, y/S\} \cup \mathbf{K}$ if $x/\{S\}_{\exists} \in \mathbf{K}$, and there is no variable z such that $x\exists z, z/S$ are in $\mathbf{K}$, and y is a new variable
4. $\mathbf{K} \longrightarrow_{\forall} \{y/S\} \cup \mathbf{K}$ if $x\exists y$ and $x/\{S\}_{\forall}$ is in $\mathbf{K}$, and y/S is not in $\mathbf{K}$
5. $\mathbf{K} \longrightarrow_{[]} \{xay, y/S\} \cup \mathbf{K}$ if $x/[a:S] \in \mathbf{K}$, and there is no variable z such that $xaz, z/S$ are in $\mathbf{K}$, and y is a new variable
6. $\mathbf{K} \longrightarrow_{\Delta} \{x \Delta y, y/S\} \cup \mathbf{K}$ if $x/\Delta S \in \mathbf{K}$, and there is no variable z such that $x \Delta y, z/S$ is in $\mathbf{K}$, and y is a new variable
7. $\mathbf{K} \longrightarrow_{\theta} \{xp_1y, xp_2z, (y\theta z)\} \cup \mathbf{K}$ if $x/(p_1\theta p_2) \in \mathbf{K}$, and there are no variables y', z' such that xp_1y' , xp_2z' and $(y'\theta z')$ are in $\mathbf{K}$, and y and z are new variables
8. $\mathbf{K} \longrightarrow_{\theta_d} \{xpy, (y\theta d)\} \cup \mathbf{K}$ if $x/(p\theta d) \in \mathbf{K}$, and there is no variable z such that xpz and $(z\theta d)$ are in $\mathbf{K}$, and y is a new variable
9. $\mathbf{K} \longrightarrow_{path} \{xez, zpy\} \cup \mathbf{K}$ if $xe.py \in \mathbf{K}$, there is no variable z' such that xez' and $z'py$ are in $\mathbf{K}$, and z is a new variable
10. $\mathbf{K} \longrightarrow_{replace} \mathbf{K}_y^z$ if
 - (a) $y = z$, or
 - (b) $y\epsilon z$, or
 - (c) xey and xez , with $e \in \mathbf{A} \cup \{\Delta\}$
 are in $\mathbf{K}$, and $y \neq z$.
11. $\mathbf{K} \longrightarrow_{\sigma} \{x/\sigma(N)\} \cup \mathbf{K}$ if $x/N \in \mathbf{K}$ and $x/\sigma(N) \notin \mathbf{K}$.
12. $\mathbf{K} \longrightarrow_{\neg} \{x/\neg\sigma_V(N)\} \cup \mathbf{K}$ if $x/\neg N \in \mathbf{K}$ and $x/\neg\sigma_V(N) \notin \mathbf{K}$.
13. $\mathbf{K} \longrightarrow_{\neq} \{x \neq y\} \cup \mathbf{K}$ if $x/N \in \mathbf{K}$ and $y/\neg N \in \mathbf{K}$ and none of $x \neq y, y \neq x$ is in $\mathbf{K}$.

A constraint system is *complete* if none of the rules applies to it.

We enumerate all clashes, that is all possible subsets of a complete constraint system $\mathbf{K}$ derived from $\{x/S\}$ which reveal that the initial type S was inconsistent. Let $e \in \mathbf{A} \cup \{\Delta, \exists\}$ and κ_s denote simple constraints. A *clash* is a set of constraints of one of the following forms:

1. a simple constraint system that is not satisfiable;
2. $\{\kappa_s, xey\}$;
3. $\{x\theta x\}$ where $\theta \in \{>, <, \neq\}$;

4. $\{x/\perp\}$;
5. $\{x/N, x/\neg N\}$;
6. $\{xey, x/(e\uparrow)\}$;
7. $\{xey, xe'y'\}$, where $(e, e') \notin \mathbf{A}^2 \cup \{\Delta\}^2 \cup \{\exists\}^2$;
8. $\{x/\{S\}_*, x/\neg\{\top\}_\forall\}$, where $\{S\}_*$ denotes both $\{S\}_\forall$ and $\{S\}_\exists$;
9. $\{x/\{\perp\}_\forall, y/\{\perp\}_\forall, x \neq y\}$;
10. $\{x/\{\perp\}_\forall, xey\}$;
11. $\{x/\{\perp\}_\forall, \kappa_s\}$;
12. $\{x_i e_i x_{i+1} \mid 1 \leq i \leq n-1\}$, where $x_1 = x_n$ and $e_i \neq \Delta$, for $1 \leq i \leq n$.

It should be observed that rule 1 is non-deterministic, since one of the two alternatives must be chosen. Therefore, according to theorem 2, a decision procedure based on this calculus must generate all the complete constraint systems which can be derived from $\{x/S\}$ (which are, up to variable renaming, finitely many), where S is the type tested for consistency. If all these systems contain a clash, then S is unsatisfiable, otherwise it is satisfiable.