Thémis: a database programming language with integrity constraints

Véronique Benzaken
Université de Paris I - Sorbonne
12 place du Panthéon 75005 Paris, France email: benzaken@lri.lri.fr

Anne Doucet
Université de Paris XI - Orsay (LRI, Bat 490)
91405 Orsay, France email: anne@lri.lri.fr

Abstract

This paper is concerned with the problem of efficiently checking of integrity constraints in data base programming languages supporting subtyping and class hierarchies. More specifically, we consider two different problems: (1) statically reduce the number of constraints to be checked, and (2) generate an efficient run time checker. Using simple strategies, one can significantly improve the efficiency of the verification. We show how to reduce the number of constraints to be checked by characterising the portions of the database that are concerned by the constraints and involved in a transaction. We also show how to generate efficient algorithms for checking a large class of constraints. We show how all the techniques presented took great advantage of the underlying type system which provide a significant help both in solving (1) and (2). Last the current status of the Thémis prototype is presented.

1 Introduction

Efforts on database programming languages have mainly been devoted to the definition of elaborated type systems and persistence mechanisms for those languages. In particular, the problems of polymorphism, static typing and inference, and object identity were the main topics of [HMS89]. On the other hand, database programming languages are in general not able to express integrity constraints in a global and declarative way although some interesting works are done in the context of object-oriented databases [Mar91].

On the contrary, extended relational systems take integrity constraints and views into consideration, mainly in the relational way [Sto75, GM79, WSK83]. Those systems are mainly relational systems in which relations attributes domains are not necessarily atomic but can be constructed using abstract types. The associated query language can also be extended to manipulate these user defined types instances. However, extended relational systems are not integrated in the sense of database programming languages. In these systems, relations are a very special kind of data type that cannot be used orthogonally to the others. In most systems, sets cannot be constructed independently of relations, and the query languages are not integrated within the language used to define the new attributes domains.

In the deductive database field, the problem of integrity constraint checking has been fully investigated [HI85, BM86, KSS87, BDM88]. A first technique consists in determining without accessing the database whether a transaction may violate an integrity constraint. Such a technique relies on theorem proving mechanism [SS89]. A second approach assumes transactions to be provided with the *atomicity* property and consists in restricting the constraints to be enforced and in avoiding to retest the portion of the database that was consistent before the transaction [Sto75, Nic79].

Although the methods proposed in this paper rely on the same ideas, we would like to emphasize the benefits gained when relying on a type system in order to cope with object-oriented complexity.

We consider two different problems: (1) statically reduce the number of constraints to be checked, and (2) generate an efficient run time checker. Of course, in the general case the problem is very complicate and finding an optimal solution to (1), for instance, is undecidable. What we want to show is that, using simple strategies, we can significantly improve the efficiency of the verification. In this paper, we shall suppose that transactions can neither be nested nor call other transactions or functions, and that constraint specification does not involve method calls, in order to avoid interprocedural analysis. This general case will be the topic of a forthcoming study.

Our main goal is to fully exploit the type information in order to simplify constraint violation detection and to speed up constraint checking. Not only classes are partially ordered according to an inheritance hierachy but we also have to face the problem of constraint checking in an environment that allows updates to be propagated among several distinct paths among objects. A first part of the paper consists in using simple compilation techniques to statically determine which constraints might be violated by a transaction. The originality of this static analysis is that it captures the notion of inheritance and of subtyping. A second contribution consists, given a transaction and a constraint, in generating a checking algorithm which will operate on the smallest portion of the database involved by the transaction. Unlike in deductive databases, objects are much more structured and they can be accessed through several different paths in the database. We show how to significantly reduce the number of checking operations to be performed, relying on the underlying typing information.

The paper is organized as follows. In Section 2, we give an overview of the Thémis schema definition language and show how to define integrity constraints in this context. In section 3, we use simple compilation techniques to statically determine which constraints might be violated by a transaction, thus reducing the number of constraints to be checked. We propose in Section 4 to generate integrity constraints checking algorithms. These algorithms are shown to significantly improve naive checking methods. In Section 5, we describe the current implementation of the Thémis prototype. It allows to validate our work. Section 6 contains some concluding remarks.

2 Overview of the Thémis Schema Definition Language

We describe here the Thémis language and give several examples that will be used in the remainder of the paper.

2.1 Types

We consider a framework in which all database manipulations are strongly and statically typed. Types can be atomic types like **integer**, **string** or **boolean**. Types can also be concrete structures like tuple and set types:

[`name`: **string**, `children`: { **string** }]
{ Person }, { **integer** }

Subtyping of concrete types is structural and inferred following the classical rules of Cardelli [Car84]. We have, for instance:

[`name`: **string**, `age`: **integer**] $\prec$ [`name`: **string**]
{ [`name`: **string**, `age`: **integer**] } $\prec$ { [`name`: **string**] }

Concrete type instances are *non shared*, *non mutable* values. Abstract types have a name and a list of features which model attributes and operations, as follows:

```
type Person is abstract [
    name: string,
    age: integer,
    children: { Person },
    spouse: Person,
    licences: { string } ]
    changeAge(integer), /* an operation */
end

type Vehicle is abstract [
    type: string,
    agev: integer,
    colour: string,
    number: integer,
    owner: Person ]
end
```

The definition of the Person type states that a person has a name, an age, a set of children, a spouse and is equipped with an operation to update the age feature. Instances of an abstract type are "objects" in the sense that they have an identity which is independent of the value of their features (name, age, etc.). Instances of an abstract type are *mutable* and *shared* values. Subtyping of abstract types is explicit as shown in the following example:

```
type Plane subtype of Vehicle is abstract [
    capacity: integer ]
end
```

This definition states that the abstract type Plane is a subtype the abstract type Vehicle and the capacity feature is specific to planes.

2.2 Names and Classes

Types are used to describe the components of a database. The database can be seen as a graph of interconnected objets and values. The roots of this graph are names defined at the schema level:

let aSetofInts : { **integer** } := { 1, 2, 3 }
let Licences : { **string** } := { "truck", "car", "bike" }

The **let** keyword defines a new name with an associated type and binds an initial value to it. The value can be updated as for any variable.

The notion of class is an extensional notion which is not captured by the type definitions. A class represents a collection of objects of one type (abstract or not) and is characterised by a name and the type of its elements as follows:

class Persons **of type** Person
class Vehicles **of type** Vehicle
class Drivers **of type** [`driver`: Person, `car`: Vehicle]

Class names are special kinds of identifiers as they can be considered as set value names but classes can be organised in a subclass hierarchy. We can write, for example:

class Planes **subclass of** Vehicles **of type** Plane
class BadDrivers **subclass of** Drivers

The class Planes is declared as a subclass of the class Vehicles. Its associated type is Plane. The semantics of the subclass relationship is the set inclusion [LR89]. For instance, the class Planes is a subset of the class Vehicles.

2.3 Transactions

In our framework, transactions are provided with the atomicity property. In this paper, we suppose that transactions do not call other transactions.

We list the operations performed by a transaction. The set $\mathcal{B}$ of elementary statements is defined as follows:

$e_1 := e_2$
$e_1.\mathtt{a} := e_2$
insert e_1 **into** e_2 where e_2 denotes a class name
drop e_1 **from** e_2 where e_2 denotes a class name

The set $\mathcal{I}$ of transactions is recursively defined as follows:

$\forall$ s $\in \mathcal{B}$, s$\in \mathcal{I}$
$\forall$ s_1, $s_2 \in \mathcal{I}$, s_1; $s_2 \in \mathcal{I}$
$\forall$ s_1, $s_2 \in \mathcal{I}$, **if** (b) s_1 **else** $s_2 \in \mathcal{I}$
$\forall$ s $\in \mathcal{I}$, **for** (o $\in$ x) **when** (b) s $\in \mathcal{I}$

The following are examples of transactions:

```
let T1 = trans(t, c: string, p: Person, a, n: integer)
             insert Vehicle[type: t, agev: a, colour: c, number: n, owner: p]
             into Vehicles
/* this transaction inserts a new vehicle in the class Vehicles */

let T2 = trans(p1, p2: Person):
             p1.spouse := p2;
             p2.spouse := p1;
/* this transaction performs a marriage between two persons */
let T3 = trans()
             for p in Persons when (today = p.birthday)
                 print("Happy Birthday ", p.name);
                 p.age := p.age + 1
/* this transaction updates the age of all Persons born on the current day */
```

2.4 Integrity Constraints

In our framework, integrity constraints are well-typed boolean expressions built using the names and classes of the schema and general operators. More formally, terms are defined as follows:

- Constants (true, false, nil) are terms.
- Each variable x is a term.
- Let t be a term, let a be an attribute (and not an operation), t.a is a term.
- Let t_1 and t_2 be two terms, let θ be an arithmetical operator (+, -, *, $\div$), $t_1 \ \theta \ t_2$ is a term.

An integrity constraint, A, is an expression of the form:

$$A = Q_{x_1 \in S_1}, .., Q_{x_k \in S_k} M(x_1, .., x_k)$$

where Q $\in \{\forall, \exists\}$, S_j is a set-structured expression and where $M(x_1, .., x_k)$ is a quantifier free formula. More precisely, formulas M are defined as follows:

- Let θ be a comparator $(=, \neq, <, >, \leq, \geq)$, let x and y be two terms, x θ y is an atomic formula,
- Each atomic formula is a formula,
- Let M and M' be two formulas, M $\wedge$ M', M $\vee$ M', $\neg$ M et (M) are formulas.

Expression $Q_{x_1 \in S_1}, .., Q_{x_k \in S_k}$ is usually referred as the constraint *prefix* while M denotes the constraint *matrix*.

The equality operator can be applied on values of any type. The other comparators can be applied to numbers and sets.

While such expressions could also be built over tuple methods extraction, we first restrict our analysis to simple (non computable) field extraction.

The following are examples of constraints that can be constructed in our framework. They will be used as running examples in Sections 3 and 4.

(A_1) $\forall$ p $\in$ Persons, $\forall$ v $\in$ Vehicles,
(v.`name` $\neq$ "Ferrari" $\vee$ v.`owner` $\neq$ p) $\vee$ p.`age` ≥ 40

This constraint expresses that every Ferrari is owned by an instance of Persons older than 40.

(A_2) $\forall$ p $\in$ Persons, $\forall$ c $\in$ p.`children`, p.`age` $>$ c.`age`

This constraint expresses that every children must be younger than his (her) parents.

(A_3) $\forall$ d $\in$ Drivers, d.`driver.age` ≥ 18
(A_4) $\forall$ p $\in$ Persons, p.`age` $\leq 130 \wedge$ p.`age` ≥ 0;

Constraints A_3 and A_4 are range constraints. The reader should notice that it is different from a domain restriction in a type specification. A domain restriction is valid for every instance of a type and does not depend on the collection to which this instance belongs. A range constraint is local to a class (here Drivers and Persons) and does not mean for instance that *every* instance of the type [`driver`: Person, `car`: Vehicle] has to satisfy this restriction. This is one example of what cannot be expressed only by means of type systems.

(A_5) $\forall$ d $\in$ Drivers,
d.`car.type` $\neq$ "truck" $\vee$ "truck-licence" $\in$ d.`driver.licences`
(A_6) $\forall$ c $\in$ Vehicles, $\exists$ d $\in$ Drivers, d.`car` $=$ c $\wedge$ c.`agev` $=$ d.`driver.age`
(A_7) $\forall$ p $\in$ Persons, p.`spouse.spouse` $=$ p $\vee$ p.`spouse` $=$ **nil**;

Constraint A_5 expresses that every truck driver must have the corresponding licence. The dummy constraint A_6 expresses that for every vehicle there always exists a driver having the same age than the vehicle he drives. Finally the last constraint expresses that every person is either the spouse of his/her spouse or is not married.

In the Thémis language, those constraints are expressed in the following way:

let (A_1) := **constraint forall** p **in** Persons, **forall** v **in** Vehicles,
(v.`name` $\neq$ "Ferrari" **or** v.`owner` $\neq$ p) **or** p.`age` ≥ 40
let (A_2) := **constraint forall** p **in** Persons, **forall** c **in** p.`children`,
p.`age` $>$ c.`age`
let (A_3) := **constraint forall** d **in** Drivers, d.`driver.age` ≥ 18
let (A_4) := **constraint forall** p **in** Persons, p.`age` ≤ 130 **and** p.`age` ≥ 0;
let (A_5) := **constraint forall** d **in** Drivers,
d.`car.type` $\neq$ "truck"
or "truck-licence" **in** d.`driver.licences`
let (A_6) := **constraint forall** c **in** Vehicles, **exists** d **in** Drivers,
d.`car` $=$ c **and** c.`agev` $=$ d.`driver.age`
let (A_7) := **constraint forall** p **in** Persons,
p.`spouse.spouse` $=$ p **or** p.`spouse` $=$ **nil**;

3 Static Anaysis of a Thémis Schema

In order to avoid checking unnecessary constraints, we want to be able to statically characterise the integrity constraints that *may be* violated by a given transaction. As the problem of determining if a transaction will certainly violate a constraint is undecidable (because this depends on input data), we are only looking for the set of constraints which might be violated, and we give a necessary condition to detect this situation.

In order to characterise this superset of constraints, for a given transaction, we shall consider the parts of the database that are dealt with in a given constraint and/or involved in a given transaction. In the following subsections, we thus define *syntactical analysis* of constraints and transactions.

Such analysis consist, informally, in a set of *paths* in the database. A path starts from a schema name or a type name, which can be followed by one or several attribute names. For example, if the path "Person.age" is in the analysis of a constraint, this means that the constraint refers to the age of an object of type Person. If the same path is in the analysis of a transaction, this means that this transaction may update the age of a Person. Therefore, the transaction may violate the given constraint, and it will be checked at the end of the transaction.

One of the characteristics of this static analysis is the use we make of the underlying type system. As we said before, abstract types instances are mutable and shared values, whereas concrete types instances are non shared non mutable values. This concept of mutability and sharing is crucial when dealing with constraints checking. Indeed, an object may be referred to by several other objets and, as a consequence, updated through different database paths. The only information that can be statically manipulated for those objects is their type.

3.1 Static analysis of constraints

A constraint has the following generic form:

$A = Q\ x_1 \in S_1, \ldots, Q\ x_k \in S_k\ M(x_1, \ldots, x_k)$

where Q denotes a quantifier ($\forall$ or $\exists$), the S_j's are set expressions, and $M(x_1, \ldots, x_k)$ is a boolean expression with the x_j's as free variables.

As we said before, the analysis of a constraint is a set of paths in the database. For a constraint A, we shall note $\Upsilon(A)$ its analysis. The set of paths $\Upsilon(A)$ is recursively constructed as follows:

$\Upsilon(\text{exp1}\ \theta\ \text{exp2}) = \Upsilon(\text{exp1}) \cup \Upsilon(\text{exp2})$,
where θ denotes any comparator;
$\Upsilon(\text{<S>}) = \text{<S>}$, if S is a schema name and not a class name;
$= \bigcup_{C_i \leq S} \{C_i\}$ (the set of all subclasses of <S>, including itself,
if <S> is a class name.)
$\Upsilon(\text{exp.}\mathsf{a}) = \bigcup_{t.\mathsf{a} \leq type(exp)} \{C_i\} \cup \Upsilon(\text{exp})$, if type(exp) is an abstract type
(type() is a function which given an expression
returns its corresponding type);
$= \Upsilon(\text{exp})$ otherwise.
$\Upsilon(x) = \Upsilon(S)$, where x represents a quantified variable in the constraint
ranging over the set expression S;

$\Upsilon(\text{M} \wedge \text{M'}) = \Upsilon(\text{M}) \cup \Upsilon(\text{M'})$, where M, M' are two quantifier free formulas;
$\Upsilon(\text{M} \vee \text{M'}) = \Upsilon(\text{M}) \cup \Upsilon(\text{M'})$, where M, M' are two quantifier free formulas;
$\Upsilon(\neg\, \text{M}) = \Upsilon(\text{M})$;
$\Upsilon((\text{M})) = \Upsilon(\text{M})$;

The construction of "Υ(exp.`a`)" deserves some comments. Such an expression can be either the extraction of a *tuple field (non mutable non shared value)*, or the extraction of an *object feature (mutable and shared value)*. If the type of the expression is an *abstract type*, this means that the corresponding value is an object that can be shared. Every update of an object of the same type may violate the constraint. For tuples that are not shared nor mutable, the expression "exp.`a`" may change only if the expression "exp" changes.

The analysis for the constraint examples are listed in Figure 1. BadDrivers appears in the analysis of constraints A_3, A_5, and A_6 because it is a *subclass* of Drivers. The same is true for Planes.

Constraint	**Analysis**
A_1	Person.`age`, Persons, Vehicle.`name`, Vehicle.`owner`, Vehicles, Planes, Plane.`name`, Plane.`owner`
A_2	Person.`age`, Person.`children`, Persons
A_3	Drivers, BadDrivers, Person.`age`
A_4	Persons, Person.`age`
A_5	Drivers, BadDrivers, Person.`licences`, Vehicle.`type`, Plane.`type`
A_6	Vehicles, Planes, Drivers, BadDrivers, Person.`age`, Vehicle.`agev`, Plane.`agev`
A_7	Persons, Person.`spouse`

Figure 1: Static analysis of constraints

The analysis of the constraint A_2 is made of three paths. This means that there are three ways to violate this constraint: we can modify the class Persons (add a new person in it for example) or we can either modify the age or the set of children of a given person.

For the constraint A_5, we obtain five paths. The first two paths express that a modification of the class Drivers (or BadDrivers) may violate the constraint. The following path expresses that the constraint may also be violated if the feature `licenses` of a Person is updated. The last two paths express that a constraint be be violated if the feature `type` of both Vehicle or Plane is modified. This takes the subtyping into account.

3.2 Static Analysis of Transactions

In this subsection, we propose a similar analysis of the transactions that can be run on the database. Our goal is to build, for every transaction T, the set ψ(T) of all paths involved in the transaction. Such paths lead to data that the transaction might modify.

In this paper, we suppose that transactions do not call other transactions or functions, so we shall do no interprocedural analysis. The analysis ψ of a transaction is then defined as follows:

$\psi(s_1 \; ; s_2) = \psi(s_1) \cup \psi(s_2)$;
$\psi($**if** (b) s_1 **else** $s_2) = \psi(s_1) \cup \psi(s_2)$;
$\psi($**for** $(o$ **in** $x)$ **when** (b) $s) = \psi(s)$;
$\psi(e_1 := e_2) = \Upsilon(e_1)$;
$\psi($<S>$)$ = <S>, if S is a schema name and not a class name;
= the set of all subclasses of <S>, including itself,
if <S> is a class name.
$\psi($**insert** e_1 **into** $e_2) = \Upsilon(e_2)$;
$\psi($**drop** e_1 **from** $e_2) = \Upsilon(e_2)$;

This definition means that the analysis of a transaction is the union of the analysis of all the elementary statements involved in it. For an assignment operation like "$e_1 := en_2$", the data that can be modified is the data corresponding to the expression e_1, and it is analysed by $\Upsilon(e_1)$. We examplify this definition with the examples of transactions above.

Transaction	**Analysis**
T_1	Vehicles, Planes
T_2	Person.`spouse`
T_3	Persons, Person.`age`

Figure 2: Analysis of transactions

We have analysed both transactions and constraints as a set of paths into the database that are used by the constraint or involved in the transaction. The detection of the constraints which may be violated by a given transaction is then very simple, as stated by the following property:

Property 1 Let T be a transaction and A be a constraint; the transaction T may violate the constraint A only if $\Upsilon(A) \cap \psi(T)$ is non empty.

If we use this property on the running examples, we obtain the table shown in Figure 3.

Transaction	**Constraints**
T_1	A_1, A_6
T_2	A_7
T_3	A_1, A_2, A_3, A_4, A_6, A_7

Figure 3: Constraints hit by a transaction

4 Enforcement Tests Generation

4.1 Restriction of the constraint language

Incremental checking is not possible for every constraint. Existential quantifiers, for example, cannot be simply incrementally checked as illustrated by the following example:

exist x **in** aSet, Pred(x)

If a transaction removes an element α from aSet and Pred(α) is true, then there is no simple and cheap way to ensure that there is still another element that satisfies the predicate. In the remaining of this section, we shall thus consider only universally quantified and conjunctive constraints.

4.2 Constraint checking

The problem of efficiently checking a constraint at the end of a transaction consists in finding the minimal set of objects involved in the process of checking. Then, the constraint will be checked only on this set which guarantees that data consistency is ensured at the end of checking. However, this set is unfortunately not always reachable at run time. In order to illustrate this we use the following three constraints A_6, A_7 and A_2:

(A_6) $\forall$ p $\in$ Persons, p.`age` $\leq 130 \wedge$ p.`age` ≥ 0;
(A_7) $\forall$ p $\in$ Persons, p.`spouse.spouse` = p $\vee$ p.`spouse` = **nil**;
(A_2) $\forall$ p $\in$ Persons, $\forall$ c $\in$ p.`children`, p.`age` > c.`age`

together with the two transactions T_2 and T_3.

let T_2 = **trans**(p_1, p_2: Person):
    p_1.`spouse` := p_2;
    p_2.`spouse` := p_1;
    /* this transaction performs a marriage between two persons */
let T_3 = **trans**()
    for p **in** Persons **when** (today = p.`birthday`)
        print("Happy Birthday ", p.`name`);
        p.`age` := p.`age` + 1
    /* this transaction updates the age of all Persons born
    on the current day */

If we consider T_3, for the first constraint, we just have to collect the identifiers of every person whose age is modified. The objects collected by this process correspond to the ideal relevant set of objects on which A_6 has to be checked.

For the second constraint, when executing transaction T_2, the ideal relevant set is not so easy to obtain. This set consists of the identifiers of p_1 and p_2 as well as the identifiers of p_1.`spouse` and p_2.`spouse` *before* the assignment. Indeed, we need to know the former spouses of p_1 and p_2, because the constraint A_7 will certainly be violated for them. Of course, collecting those identifiers requires the checker manager to be provided with some kind of "intelligence".

For the third constraint, when executing T_3, we have no means to collect the parents of a child whose age has been modified, because we don't have backward pointers nor indices.

As a consequence, we will not attempt to obtain the ideal set of relevant objects. At the same time, we will not assume the existence of special access structures like indexes or backward pointers. We rather address the problem of finding an efficient checking algorithm which can be applied to all constraints. For constraints such as A_6, the algorithm will operate on the ideal relevant set of objects; for other constraints, we shall show that the checking algorithm improves the trivial approach which consists in performing a whole scan on the populations involved in the constraints.

Let T be a transaction, let A be a constraint. We are looking for an algorithm satisfying the following properties:

- The evaluation of this algorithm at the end of the transaction ensures that the constraint A is still satisfied.
- The evaluation of this algorithm is more efficient than the direct evaluation of A.

At execution time, the only objects which can be collected are those instances of abstract data types whose attributes relevant with respect to the constraints have been modified. It may be the case that such a set of objects exactly matches the ideal relevant set, as for constraint A_6. But in general, the set obtained at execution time only intersects the relevant set, as for constraint A_7 et A_2. Therefore, we propose to generate checking algorithms which allow to test the constraint on the whole set of relevant objects, thus ensuring the database consistency. As a consequence, we will have to perform some additional work, in order to get these objects.

In order to define these algorithms, let us introduce the following definitions.

Definition 1 $\Delta^{\mathtt{C}}$
Given a class C, we posit $\Delta^{\mathtt{C}}$ the set of instances of class C which have been created by a transaction.
□

Definition 2 $\Gamma_{\mathtt{a}}^{\tau(x)}$
Given an iteration variable x of type τ(x) and an attribute $\mathtt{a}$ of x, we posit $\Gamma_{\mathtt{a}}^{\tau(x)}$ the set of instances of the abstract type τ(x), which attribute $\mathtt{a}$ has been modified by a given transaction.
□

This set represents information on the updates that a transaction has made on the database. The constraints considered here have the following generic form:

$$\forall x_1 \in \mathtt{C}_1, \forall x_{1,1} \in x_1.\mathtt{p}_{1,1}, \ldots, \forall x_{1,n_1} \in x_1.\mathtt{p}_{1,n_1},$$

$$\forall x_2 \in \mathtt{C}_2, \forall x_{2,1} \in x_2.\mathtt{p}_{2,1}, \ldots, \forall x_{2,n_2} \in x_2.\mathtt{p}_{2,n_2}, \ldots,$$

$$\forall x_k \in \mathtt{C}_k, \ldots, \forall x_{k,n_k} \in x_k.\mathtt{p}_{k,n_k}, \mathtt{M}(x_1, x_{1,1}, \ldots, x_k, \ldots, x_{k,n_k})$$

where the $\mathrm{p}_{i,j}$ denote prefix paths.

Let x be a variable ranging over class C_i, and let y_1, …, y_n be variables ranging respectively over $x.p_1$, …, $x.p_n$, where p_i denotes a prefix path leading to a set structured component of x. In the following, we show how to generate enforcement tests.

For each class C_i, we generate the following enforcement test:

$$\begin{aligned}
&\forall x \in \Delta^{\mathtt{C}_i} \\
&\quad \text{check } [\ \forall\ x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots, \\
&\quad M(x_1, \ldots, x, \ldots, x_{i+1}, \ldots)]
\end{aligned}$$

For each path $x.a_1 \ldots a_k$ (either in the prefix or in the matrix), we generate:

$$\begin{aligned}
&\forall x \in C_i \\
&\quad \forall y \in \Gamma^{\tau(x)}_{\mathtt{a_1}}, \text{ if } y = x, \\
&\quad \text{check } [\ \forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots, \\
&\quad M(x_1, \ldots, x, \ldots, x_{i+1}, \ldots)] \\
&\quad \ldots \\
&\quad \forall y \in \Gamma^{\tau(x.a_1 \ldots a_{k-1})}_{\mathtt{a_k}}, \text{ if } y = x \ldots a_{k-1}, \\
&\quad \text{check } [\ \forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots, \\
&\quad M(x_1, \ldots, x, \ldots, x_{i+1}, \ldots)]
\end{aligned}$$

For each path $y_i.b_1 \ldots b_l$ in the matrix (where y_i ranges in $x.p_i$), we generate

$$\begin{aligned}
&\forall x \in C_i \\
&\quad \forall y \in x.p_i \\
&\qquad \forall z \in \Gamma^{\tau(y)}_{\mathtt{b_1}}, \text{ if } z = y, \\
&\qquad \text{check } [\ \forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots, \\
&\qquad M(x_1, \ldots, x, \ldots, y, \ldots, x_{i+1}, \ldots)] \\
&\qquad \ldots \\
&\qquad \forall z \in \Gamma^{\tau(y \ldots b_{l-1})}_{\mathtt{b_l}}, \text{ if } z = y \ldots b_{l-1}, \\
&\qquad \text{check } [\ \forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots, \\
&\qquad M(x_1, \ldots, x, \ldots, y, \ldots, x_{i+1}, \ldots)]
\end{aligned}$$

For a given constraint A, the enforcement test generation consists in generating the above tests for each class C_i involved in the constraint prefix. Let us illustrate this on the following constraints.

(A_6) $\forall$ p $\in$ Persons, p.`age` $\leq 130 \wedge$ p.`age` ≥ 0;
(A_7) $\forall$ p $\in$ Persons, p.`spouse.spouse` = p $\vee$ p.`spouse` = **nil**;
(A_2) $\forall$ p $\in$ Persons, $\forall$ c $\in$ p.`children`, p.`age` > c.`age`
(A_1) $\forall$ p $\in$ Persons, $\forall$ v $\in$ Vehicles, (v.`name` $\neq$ "Ferrari" $\vee$ v.`owner` $\neq$ p) $\vee$
p.`age` ≥ 40

For the constraint A_6, the checking algorithm is

$\forall$ x $\in \Delta^{\texttt{Persons}}$, check ($A_6$(x))
$\forall$ x $\in$ Persons
    $\forall$ y $\in \Gamma_{\texttt{age}}^{\texttt{Person}}$, if y = x, check($A_6$(x))

This can be rewritten as following:

$\forall$ x $\in \Delta^{\texttt{Persons}}$, check ($A_6$(x))
$\forall$ x $\in$ Persons $\cap\ \Gamma_{\texttt{age}}^{\texttt{Person}}$, check ($A_6$(x))

In this case, the set $\Gamma_{\texttt{age}}^{\texttt{Person}}$ actually represents the relevant set of objects on which the constraint has to be checked. Indeed, this algorithm leads to check the constraint on the set $\Gamma_{\texttt{age}}^{\texttt{Person}}$ testing for each element if it belongs to the class Persons. We thus perform as many check operations as the minimal algorithm does. Note that the trivial algorithm would have performed as many checks as the number of elements in the class Persons.

For the constraint A_7, the checking algorithm is

$\forall$ x $\in \Delta^{\texttt{Persons}}$, check ($A_7$(x))
$\forall$ x $\in$ Persons
    $\forall$ y $\in \Gamma_{\texttt{spouse}}^{\texttt{Person}}$,
        if y = x, check(A_7(y))
    $\forall$ y $\in \Gamma_{\texttt{spouse}}^{\texttt{Person}}$,
        if y = x.spouse, check(A_7(y))

For this algorithm, we have to scan the whole class Persons and test whether an element of $\Gamma_{\texttt{spouse}}^{\texttt{Person}}$ corresponds to the spouse attribute of a given instance of Persons.

For the constraint A_2, the checking algorithm is

$\forall$ x $\in \Delta^{\texttt{Persons}}$, check ($A_2$(x))
$\forall$ x $\in$ Persons
    $\forall$ y $\in \Gamma_{\texttt{age}}^{\texttt{Person}}$,
        if y = x, check(A_2(y))
    $\forall$ y $\in \Gamma_{\texttt{children}}^{\texttt{Person}}$,
        if y = x, check(A_2(x,y))
    $\forall$ y $\in$ x.children
        $\forall$ z $\in \Gamma_{\texttt{age}}^{\texttt{Person}}$,
        if z = y, check(A_2(x,y))

This algorithm iterates over three sets: Persons, $\Gamma_{\texttt{age}}^{\texttt{Person}}$ and $\Gamma_{\texttt{children}}^{\texttt{Person}}$. For each element x of Persons whose age has been modified, we have to check the constraint. For each element x of Persons, if the age of one of his/her children has been modified, we have to check whether the constraint is still valid. Last, for each element of Persons whose set of children has been modified, we also have to check the constraint.

Finally, for the constraint A_1, the checking algorithm is

$\forall$ x $\in \Delta^{\texttt{Persons}}$, check $(A_1(x))$
$\forall$ x $\in$ Persons
    $\forall$ y $\in \Gamma^{\texttt{Person}}_{\texttt{age}}$,
        if y = x, check$(A_1(y))$
$\forall$ x $\in \Delta^{\texttt{Vehicles}}$, check $(A_1(x))$
$\forall$ x $\in$ Vehicles
    $\forall$ y $\in \Gamma^{\texttt{Vehicle}}_{\texttt{name}}$,
        if y = x, check$(A_1(y))$
    $\forall$ y $\in \Gamma^{\texttt{Vehicle}}_{\texttt{owner}}$,
        if y = x check$(A_1(y))$

This algorithm can be rewritten as follows:

$\forall$ x $\in \Delta^{\texttt{Persons}}$, check $(A_1(x))$
$\forall$ x $\in$ Persons $\cap \, \Gamma^{\texttt{Person}}_{\texttt{age}}$,
        check$(A_1(x))$
$\forall$ x $\in \Delta^{\texttt{Vehicles}}$, check $(A_1(x))$
$\forall$ x $\in$ Vehicles $\cap \, (\Gamma^{\texttt{Vehicle}}_{\texttt{name}} \cup \Gamma^{\texttt{Vehicle}}_{\texttt{owner}}$
        check$(A_1(x))$

This last example deserves some comments: checking A_1 means that we check A_1 with respect to all the elements in either Vehicles or Persons. Therefore some tests are redundant. When checking the set of Persons whose age has been modified, we will consider all Vehicles and particularly those Vehicles whose name or owner attribute has been updated. In the second phase of the algorithm, we shall test the constraint for all updated Vehicles with respect to all Persons including those whose age has been modified. In order to avoid such redundant tests, we refine this algorithm in the following way. We saw on the previous examples (for the constraints A_6 and A_1) that the checking algorithms could be rewritten in an optimized form. Such an optimisation can only take place for the algorithms containing no navigation in the type structures. For example, it is not possible to optimise in this way the algorithm generated for the constraint A_2, because y has to range over x.`children`.

We now give a general optimized version of this class of algorithms.

$\forall x \in C_1 \cap (\cup\Gamma_1)$
    check $[\, \forall x_2 \in C_2, \ldots, \forall x_k \in C_k, \ldots,$
        $M(x, \ldots, x_2, \ldots, x_k, \ldots)]$
    ...
$\forall x \in C_i \cap (\cup\Gamma_i)$
    check $[\, \forall x_1 \in C_1 - (C_1 \cap (\cup\Gamma_1)), \ldots,$
        $\forall x_{i-1} \in C_{i-1} - (C_{i-1} \cap (\cup\Gamma_{i-1})), \ldots,$
        $\forall x_{i+1} \in C_{i+1}, \ldots,$
        $M(x_1, \ldots, x_2, \ldots, x, \ldots, x_k, \ldots)]$
    ...
$\forall x \in C_k \cap (\cup\Gamma_k)$
    check $[\, \forall x_1 \in C_1 - (C_1 \cap (\cup\Gamma_1)), \ldots,$
        $\forall x_i \in C_i - (C_i \cap (\cup\Gamma_i)), \ldots,$
        $\forall x_{k-1} \in C_{k-1} - (C_{k-1} \cap (\cup\Gamma_{k-1})),$
        $M(x_1, \ldots, x_2, \ldots, x, \ldots)]$

The union of Γ_i denotes the set of all instances of an abstract type whose attributes relevant to a given constraint have been updated. Such an optimised version prevents from testing more than one time the same constraint on the same objects.

5 Implementation

This section describes the implementation of a first prototype for the Thémis language. Thémis is implemented on top of the O_2 system, using a preprocessing approach. The preprocessor O_2integrity takes a schema written in Thémis and produces an O_2 schema and a set of O_2 executables programs which allow to instanciate the constraints while preserving the inclusion semantics. O_2integrity is written in C^{++} and uses *lex* and *yacc*.

5.1 Mapping between Thémis and O_2

In this section, we describe the mapping between the Thémis language and O_2.

5.1.1 Atomic types

Thémis	O_2
int	integer
string	string
boolean	boolean
-	real
-	bits

5.1.2 Type Constructors

In the O_2 language, it is possible to define complex objects and values by using various constructors, as in Thémis.

Thémis	O_2
$[a_1{:}t_1,..,a_n{:}t_n]$	tuple($a_1{:}t_1,..,a_n{:}t_n$)
$\{t_1\}$	set(t_1)
(t_1)	list(t_1)

5.1.3 Types

In the O_2 language, the instances of a type are values and the instances of a class are objects. These properties are offered in Thémis through concrete and abstract types.

Thémis	O_2
Concrete type	type
Abstract type	class

5.1.4 Subtyping and inheritance

O_2 and Thémis follow the same subtyping rules:

- An explicit subtyping for abstract types (Thémis) and the classes (O_2).
- An implicit subtyping for concrete types (Thémis) and types (O_2).

5.2 Constraints and transactions

The constraints defined in Thémis are instances of a predefined class "Constraint" in O_2. The transactions are translated into O_2 transactions and compiled by the O_2C compiler.

5.3 Example

In this section, we describe the fonctionalities provided by the O_2integrity precompiler.

The following figure presents our schema in Thémis together with the constraints and the transactions.

Figure 4 : Schema in Thémis

The various components generated by the precompiler, namely, the constraints, the transactions and the classes are displayed in the Figures 5, 6, 7 and 8.

Figure 5 : O_2integrity components

Figure 6: Examples of constraints

Figure 7: O_2 Classes

Figure 8: O_2 Transactions

Finally, the set of constraints which might be violated by the transactions of the current schema is represented in Figure 9.

Figure 9: Constraints hit by transactions

6 Conclusion

In this paper, we proposed to extend relational simplification methods in the context of complex objects with identity in a statically typed environment. Using simple compilation techniques, we have shown how to statically reduce the number of constraints to be checked at the end of a given transaction. While adapting the techniques, focus has been devoted to set-structured data types. Unlike other techniques which only consider the constraint prefix, we consider as well the matrix thus refining the relevant sets of objects that could be involved in the process of checking. Finally, we have presented enforcement tests generation techniques.

This work is a preliminary work on the specification of a compiler for the language proposed in [LR90]. In order to get complete specifications, we will extend our techniques in the following directions.

In Sections 2 and 3, we did not consider methods (that is n-ary features). We will extend our work on static characterisation in order to take into account methods as well.

For the optimisation of constraint checking, we shall work in three different directions. First, we will try to extend our result in the case of quantifiers over general set expressions (not only class names). In the present work, we do not consider such complex iterations. Then, we will consider the case of constraints with negation and/or existential quantifiers. Third, we shall refine the checking algorithms in order to avoid performing redundant checks.

In order to be able to estimate the impact of such extensions, we will try, at the same time, to formalise the cost (in time and space) of constraints verifications. Such a formalisation would allow, for example, to make some trade-off between time (verification at the end of a transaction) and space (amount of data collected during the transaction).

Finally, in order to get more results on constraint optimisation, we will continue to compare our work with the more general domain of logic programs compilation. We think that, although the underlying data models and type systems are very different, there are enough commonalities between these two problems to get some more results.

Acknowledgements

This work was initiated when the authors were in the Altaïr group at INRIA together with C. Lécluse and P. Richard. We are thankful to them for helpful advices and for stimulating discussions and ideas. We also would like to thank P.Y. Policella and P. Tronowski who greatly contributed in the implementation of the first Thémis prototype.

References

[BDM88] F. Bry, H. Decker, and R. Manthey. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *EDBT International Conference*, 1988.

[BM86] F. Bry and R. Manthey. Checking Consistency of Database Constraints: A Logical Basis. In *VLDB International Conference*, 1986.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types*. Springer-Verlag, 1984.

[GM79] G. Gardarin and M. Melkanoff. Proving the Consistency of Database Transactions. In *VLDB International Conference*, Rio, Brasil, October 1979.

[HI85] A. Hsu and T. Imielinski. Integrity Checking for Multiple Updates. In *ACM SIGMOD International Conference*, 1985.

[HMS89] R. Hull, R. Morrison, and D. Stemple, editors. *International Workshop on Database Programming Languages*. Morgan Kaufmann, 1989.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In *VLDB International Conference*, 1987.

[LR89] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *ACM PODS International Conference*, March 1989.

[LR90] C. Lécluse and P. Richard. Data Base Schemas and Types Systems for DBPLs, a Definition and its Applications. Technical report, GIP Altaïr, June 1990.

[Mar91] H. Martin. *Contrôle de la cohérence dans les bases objets : Une approche par le comportement*. PhD thesis, Université Joseph-Fourier - Grenoble I, 1991.

[Nic79] J.M. Nicolas. Logic for Improving Integrity Checking in Relational Databases. Technical report, ONERA-CERT, 1979.

[SS89] T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. *ACM Transactions on Database Systems*, 14(3), September 1989.

[Sto75] M. Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In *ACM SIGMOD International Conference*, San Jose, California, May 1975.

[WSK83] W. Weber, W. Stugky, and J. Karzt. Integrity Checking in database systems. *Information Systems*, 8(2), 1983.