Thémis: a Database Programming Language Handling 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 article presents a data base programming language: Thémis supporting subtyping and class hierarchies and allowing for the definition of integrity constraints in a global and declarative way. We first describe the salient features of the language: types, names, classes, integrity constraints (including methods) and transactions. The inclusion of methods into integrity constraints allows an increased of the declarative power of these constraints. Indeed, the information needed in the definition of a constraint is not always stored in the database through attributes, but is sometimes computed or derived data. Then we address the problem of efficiently checking constraints. More specifically, we consider two different problems: (1) statically reducing the number of constraints to be checked, and (2) generating 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 involved in both the constraints and a transaction. We also show how to generate efficient algorithms for checking a large class of constraints. We show how all the techniques presented exploit the underlying type system which provides significant help 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 [Car84, CW85, Car87, Car88, AB87, HMS89] and more recently of [CGL95, Cas95a, Cas95b].

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 work is done in the context of object-oriented databases [Mar91].

In [BLR92, BD93], a first specification of a language able to express integrity constraints has been proposed. However, the class of constraints expressible by this language is restricted to first-order logic well typed formulas. In fact, some information are derived data (computed attributes, such as the age of a person, given the birth-date, or the computation of the incoming and outgoing degrees of a directed graph). This kind of derived data cannot be defined by first-order logic. In object-oriented languages, the only way to express this kind of information consists of using methods.

In order to precisely capture the semantics of an application, some integrity constraints have to consider this kind of information. It is thus necessary to introduce method calls in the language used to express constraints.

*PhD Université of Paris XI, Maître de conférences, also researcher at L.R.I, Bat 490, Université de Paris XI, 91405 Orsay Cedex, France

†PhD Université of Paris VII, Professor, on leave since october 1994 to L.A.F.O.R.I.A, Institut Blaise Pascal, case 169, Université de Paris VI, 4, Place Jussieu, 75005 Paris, France

Relational and extended relational systems take integrity constraints and views into consideration [Sto75, GM79, WSK83, SS89]. These systems provide models in which relation attribute 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.

A second approach assumes transactions to be provided with the *atomicity* property and consists of restricting the constraints to be enforced and of avoiding to retest the portion of the database that is known to be consistent after the execution of the transaction [Nic79, HI85].

In the deductive database field, the problem of integrity constraint checking has been fully investigated [BM86, KSS87, BDM88]. Most of the techniques proposed are based on the Linear resolution with function Selection on Definite clauses with Negation as Failure (SLD/SLDNF resolutions) and theorem proving.

The work described by Sheard and Stemple in [SS89] consists of proving at compile time that database transactions respect integrity constraints, in order to reduce the overhead of unnecessary runtime tests. Their framework is the relational model. Transactions are complex updates of multiple relations, and constraints can be functional dependencies, inclusion dependencies, aggregate constraints, intersection dependencies and inter-relational redundancies. In [SS89], the axiomatic semantics method is used. In that method, properties about language constructs are defined. These properties are found by using axioms and inference rules. Inference rules are re-write rules on functional expressions of theorems, that allow to reduce these expressions to true by using axioms, function definitions, and previously proven theorems. This leads to a formal proof of the property. The system uses a mechanical theorem prover in higher order computational logic to build a formal theory about database systems. That theory is extended to a specific database by generating specific knowledge from the structures and constraints contained in the schema. It is finally used by a transaction safety verifier.

We first describe the salient features of the Thémis language: types, names, classes, integrity constraints (including methods) and transactions. Then, we consider two different problems: (1) statically reducing the number of constraints to be checked, and (2) generating an efficient run time checker. Of course, in the general case the problem is very complicated 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 article, we shall suppose that transactions can neither be nested nor call other transactions. The 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 are classes partially ordered according to an inheritance hierarchy 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 article consists of 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 subtyping and of late binding.

A second contribution consists of generating from a transaction and a (restricted) constraint a checking algorithm which will operate on the smallest portion of the database involved by the transaction. We show how to significantly reduce the number of checking operations to be performed, relying on the underlying typing information.

The article is organized as follows. In Section 2, we summarize the main techniques which have been developed in the domain, in order to compare our work. We describe in Section 3 the salient features of the Thémis language: types, names, classes, integrity constraints (including methods) and transactions. We also present a detailed example to motivate and illustrate our language and checking techniques. Then, we consider the two steps of the verification process. In Section 4, 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 5 to generate constraint checking algorithms for a special class of constraints (universally quantified formulae). These algorithms are shown to significantly improve naive checking methods. In Section 6, we describe the current implementation of the Thémis prototype. It allows us to validate our work. Section 7 contains some concluding remarks.

2 Related Work

Relational and extended relational systems generally handle integrity by means of triggers. Triggers allow a user-defined procedure to be executed when a predicate is satisfied. Integrity constraints can be seen as rules, but they do not perform database updates. They simply return an error condition when an attribute is incorrectly modified. Although rules (or triggers) allow to integrity constraints to be specified in a declarative way, it is the responsibility of the application programmer to code the procedures which will guaranty database safety. Our approach is different, in the sense that we propose to automatically generate these checking procedures, thus relieving the programmer of such a task, and therefore enhancing his/her productivity. To ensure database integrity, the user only describes the constraints.

Hsu and Imilienski propose in [HI85] another solution, which extends Blaustein's work [Bla81]. The constraints they considered are closed formulæof tuple calculus in prenex normal form. The simplification method consists here in transforming a constraint into an AND-OR tree of constraints which are simpler to evaluate. Simpler means that the checking space is reduced. Indeed, instead of testing the constraint on the whole data, they only consider the data which might affect the database consistency with respect to both constraints and transactions. Interesting data are either inserted tuples or deleted tuples (updates consisting of deletions followed insertions).

Constraint simplification is performed in three steps. The constraint is first transformed into an updated form, involving the updated data. The second step consists of applying decomposition rules to the prefix of the updated constraint. These rules, which take into account only some prefix patterns, are recursively applied to the constraint, and produce either a conjunction or a disjunction of new formulæ. The third step consists of eliminating the subformulæ which are known to be true.

Our work adapts and extends these techniques to the object-oriented framework. More precisely, we use a similar technique in the second phase of our checking process, namely the generation of checking algorithms.

The approach adopted by Sheard and Stemple [SS89] consists of proving at compile time that database transactions respect integrity constraints, in order to reduce the overhead of unnecessary runtime tests. Their framework is the relational model. Transactions are complex updates of multiple relations, and constraints can be functional dependencies, inclusion dependencies, aggregate constraints, intersection dependencies and inter-relational redundancies. In [SS89], the axiomatic semantics method is used. In that method, properties about language constructs are defined. These properties are found by using axioms and inference rules. Inference rules are re-write rules on functional expressions of theorems, that allow the reduction of these expressions to true by using axioms, function definitions, and previously proven theorems. This leads to a formal proof of the property. The system uses a mechanical theorem prover in higher-order computational logic to build a formal theory about database systems. That theory is extended to a specific database by generating specific knowledge from the structures and constraints contained in the schema. It is finally used by a transaction safety verifier.

The deductive framework is well suited to integrity constraint management [BM86, KSS87, BDM88]. Deductive databases are a set of facts associated with a set of rules, which represent derived data. Integrity constraints can be expressed in this formalism as rules. Such rules of course do not perform updates, nor generate new facts. In this context, two problems are addressed: satisfaction and satisfiability. Two kinds of updates are considered for the problem of satisfaction, which are addition of a new fact and deletion of an existing one. According to the update, a first step consists of detecting which integrity constraint might be affected. Then, the checking process operates on the facts contained in the database. Both steps are achieved using SLD/SLDNF resolution.

For the second problem, namely satisfiability, the update considered is the addition of a new constraint. In order to detect whether the constraint is consistent with respect to the existing ones, the method aims at generating a finite model, independent of the existing instance. The method has been shown to be semi-decidable. In both cases, only a restricted set of integrity constraints is handled, all of which are either universally quantified or existentially quantified constraints.

3 Basic concepts of the Thémis language

In this section, we present the basic concepts of the Thémis language. These concepts will be illustrated by detailed examples described in Section 3.5. Thémis is a strongly and statically typed object-oriented database language. In Thémis a schema is defined using abstract and concrete types, classes, integrity constraints and transactions.

3.1 Types

We consider a framework in which all database manipulations are strongly and statically typed. Let us suppose the existence of the set $\mathcal{D}$ of atomic types containing **integer, string** and **boolean**. Types can either be concrete types or abstract types.

3.1.1 Concrete types

The set of expressions of concrete types, denoted $\mathcal{T_C}$ is built by induction in the following way:

- Basic types: $\mathcal{D}$
- If $t_1, .., t_n \in \mathcal{T_C}$ and $\mathtt{a}_1, .., \mathtt{a}_n \in \mathcal{A}$ ($\mathtt{a}_i \neq \mathtt{a}_j$ for $i, j \in 1..n$, $i \neq j$ and $n \geq 1$) then $[\mathtt{a}_1 : t_1, ..., \mathtt{a}_n : t_n] \in \mathcal{T_C}$, $\{t_1\} \in \mathcal{T_C}$, and $(t_1) \in \mathcal{T_C}$
  where [ ], { } and ( ) respectively denote the constructors *tuple, set* and *list*, and $\mathcal{A}$ denotes the set of attribute names.

Concrete type equivalence is structural. Subtyping of concrete types is structural and inferred following the classical rules of Cardelli [Car84]. For instance, we have:
`[num: integer, label: string]` $\prec$ `[num: integer]`
Concrete type instances are *non shared, non mutable* values.

3.1.2 Abstract types

Abstract types have names. An abstract type is composed of a structural part, and a behavioral part. The structural part is similar to concrete types. The behavioral part is described by a set of operations, called *methods*. Methods are defined in the following section. Instances of abstract types are objects, and have an *identity*, which is independent of their value. These instances are *mutable* and may be *shared* values. Equality of instances of abstract types is identity. Equivalence of abstract types is name equivalence. Subtyping of abstract types is explicit. The subtyping relation is declared in the definition of the abstract type.

3.1.3 Methods

Methods describe the behavior of the objects. They are composed of a signature and of an implementation (the body of the method). Methods are not considered here as first class objects, and thus cannot be passed as parameters of other methods. Let m be a method defined for the abstract type T. We denote its signature by $m@T(\tau_1, .., \tau_{n-1}) : \tau_n$, where $\tau_1, .., \tau_{n-1}$ represent its parameter types and where τ_n represents the result type.

Passing a message is denoted by $o \leftarrow m(x_1, .., x_{n-1})$. This means that the method m is sent to the object o, called the receiver of the message, the x_i's denote the actual parameters of method m. We denote $o \leftarrow m()$ when the method m has no parameters.

Message passing can be more complex and may consist of message passing. This is denoted by: $o \leftarrow m_1(x_1^1, .., x_{n_1}^1) \leftarrow ... \leftarrow m_k(x_1^k, .., x_{n_k}^k)$. When a method is redefined in a subtype hierarchy, the corresponding signatures are constrained to be *covariant*.

3.2 Classes

Types are used to describe the components of a database. The database can be seen as a graph of interconnected objects and values. The persistent roots of this graph are classes. Persistence is achieved through reachability.

A class gathers the set of objects having the same characteristics and the same behavior. The notion of class is an extensional notion. It represents a collection of objects of one type (abstract or not) and is characterised by a name and the type of its elements.

Classes are organized in a subclass hierarchy. The semantics of the inheritance relation is inclusion.

3.3 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, etc.) 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 (provided that t is a tuple-structured term with attribute `a`).
- Let t be a term, $x_1, \ldots, x_n$ be variables, let `m` be a method, $t \leftarrow \mathtt{m}(x_1, \ldots, x_n)$ is a term.
- Let t_1 and t_2 be two terms, let θ be an arithmetical operator (+, −, *, ÷), $t_1\ \theta\ t_2$ is a term.

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

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

where Q $\in \{\forall, \exists\}$, S_j is a set-structured expression and $\mathrm{M}(x_1, .., x_k)$ is a quantifier free formula. Expression $\mathrm{Q}x_1 \in S_1$,..., $\mathrm{Q}x_k \in S_k$ is usually referred as the constraint *prefix* while M denotes the *matrix* of the constraint. More precisely, formulæ 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 F and F' be two formulas, F $\wedge$ F', F $\vee$ F', $\neg$ F and (F) are formulæ.

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

3.3.1 Remarks and Restrictions

The introduction of methods into integrity constraints allows us to increase the declarative power of these constraints. Indeed, the information needed in the definition of a constraint is not always stored in the database through attributes, but is sometimes computed or derived data. This happens for information requiring important computations or for derived data structures which cannot be defined with the first-order logic (the transitive closure, for instance).

In order to keep the declarative aspect of a constraint, a method cannot modify the data stored in the database, but it must be allowed to define *virtual* data (methods allowed in the definition of constraints are overloaded queries). This *virtual* data represents the *intensional structures* of the database.

A method can appear both in the prefix and in the matrix of a constraint. The signature of a method appearing in the prefix of the constraint must return a set structured result. However, in a constraint all quantified variables denote persistent data. Therefore, in order to keep this property, a method appearing in the prefix of a constraint must return a set of persistent data. Hence, the body of this method can only contain a set of selections over the classes.

3.4 Transactions

Transactions are provided with the atomicity property: a transaction is either completely executed, or not executed at all. This mechanism allows us to overcome some errors, and to provide consistent executions. For the sake of simplicity, we consider in this paper only simple "flat" transactions (a transaction that does not call other transactions). A transaction is syntactically defined as follows: $T = trans(\tau_1, ..., \tau_n)\ \Gamma$ where

$\tau_i \in \{ \mathcal{T}_\mathcal{C} \cup \mathcal{T}_\mathcal{A} \}$[1] and Γ represents the set of all elementary statements of a transaction. This set is recursively defined as follows:

- **assignment**
  $e_1 := e_2 \in \Gamma$
  $e_1.\mathtt{a} := e_2 \in \Gamma$ if ($\mathtt{a} \in \mathcal{A}$)
  where e_2 represents any expression
- **method call**
  $o \leftarrow \mathtt{m}(x_1, \ldots, x_n) \in \Gamma$
- **sequencement**
  $\forall\, s_1, s_2 \in \Gamma$
  $s_1; s_2 \in \Gamma$
- **conditional test**
  $\forall\, s_1, s_2 \in \Gamma$
  if (b) **then** s_1 **else** $s_2 \in \Gamma$
  where b denotes a boolean expression
- **iteration loop**
  $\forall\, s \in \Gamma$
  for (o **in** x) $s \in \Gamma$
  where x denotes a set expression and o an element of x
- **set operations**
  insert o **into** $x \in \Gamma$
  drop o **from** $x \in \Gamma$
  where x denotes a set expression and o an element of x for the **drop** instruction

3.5 Example

In order to illustrate the concepts of Thémis, we give in this section an example, which will be used in the remainder of the paper. Let us consider the types given in Figure 1. The type Person has five attributes (`name, age, birthday, spouse` and `children`), and three methods. The method `descendants()` computes the graph representing all the descendants of a given person. The method `ancestor()` computes the graph representing the ancestors of a given person. The method `genealogy()` computes both the ancestors and the descendants of a person.

A graph is represented by a pair $< V, E >$, where V is a finite set of vertices, and E is a finite set of edges, each edge being a pair of vertices.

The type Matrix is used as an alternative representation of a graph which simplifies the implementation of some algorithms. The type Matrix is a list of lists of booleans. The `closure()` operation returns a matrix representing the set of all possible paths in the graph. The `connected()` operation indicates if the graph is strongly connected or not. Finally, the `non_circuit()` operation determines if the graph has a circuit or not.

For this schema, we define the classes and constraints described in Figure 2. Constraint A_1 expresses that the descendants of a given person is represented by a directed acyclic graph, while constraint A_2 expresses that the genealogy of a given person is a strongly connected graph.

Constraint A_3 states that the age of a person ranges between 0 and 130. Constraint A_4 expresses that every person is either the spouse of his/her spouse or is not married, and constraint A_5 expresses that every child must be younger than his (her) parents. Finally constraint A_6 expresses that every Ferrari is owned by an instance of Persons older than 40.

Lastly, we define in our schema the transactions given in Figure 3.

[1] $\mathcal{T}_\mathcal{A}$ denotes the set of abstract types

```
type Person is abstract [
    name: string,
    age: integer,
    birthday: integer,
    spouse: Person,
    children: set {Person}]
    descendants(): Graph,
    ancestors(): Graph,
    genealogy(): Graph
end
type Graph is abstract [edges: Edge, vertices: Vertex]
    add_edge(v1, v2: Vertex),
    delete_edge(e: Edge),
    matrix(): Matrix
end
type Matrix is abstract ((boolean))
    closure(): Matrix,
    connected(): boolean,
    non_circuit(): boolean
end
type Vertex is abstract [num: integer, id: Person]
    incoming_degree(g: Graph): integer,
    outgoing_degree(g: Graph): integer
end
type Edge is abstract [ vertex1: Vertex, vertex2: Vertex, weight: integer]
end
type Vehicle is abstract [name: string, owner: Person ]
end
```

Figure 1: A Thémis Schema

4 Static analysis of a Thémis schema

In order to avoid checking unnecessary constraints, we want to be able to statically characterize 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, we are only looking for the set of constraints which might be violated. In order to characterize this superset of constraints, for a given transaction, we consider the parts of the database that are dealt with in a given constraint and/or involved in a given transaction. A syntactic analysis of the constraints and the transactions has thus been defined in [BLR92, BD93]. Such an analysis consists, informally, of a set of *paths* in the database, gathering the set of classes and attributes used in the constraints and the transactions.

The analysis proposed in [BLR92, BD93] only considers the structure of the database, but does not take methods into consideration. The introduction of methods makes the situation much more complex. Indeed, the data structures they manipulate are not always explicitly present in the database, but can be defined only for computing purposes. Thus, a syntactic analysis of the methods will retrieve the set of "paths", which create these "temporary structures".

We propose in this section a structural and behavioral syntactic analysis of the constraints and the transactions.

class Persons of type Person
class Vehicles of type Vehicle

(A_1) $\forall$ p $\in$ Persons, p $\leftarrow$`descendants()`$\leftarrow$`matrix()`$\leftarrow$`closure()`$\leftarrow$`non_circuit()`;
(A_2) $\forall$ p $\in$ Persons, p $\leftarrow$`genealogy()`$\leftarrow$`matrix()`$\leftarrow$`closure()`$\leftarrow$`connected()`;
(A_3) $\forall$ p $\in$ Persons, p.`age` $\leq$ 130 $\wedge$ p.`age` $\geq$ 0;
(A_4) $\forall$ p $\in$ Persons, p.`spouse.spouse` = p $\vee$ p.`spouse` = **nil**;
(A_5) $\forall$ p $\in$ Persons, $\forall$ c $\in$ p.`children`, p.`age` > c.`age`
(A_6) $\forall$ p $\in$ Persons, $\forall$ v $\in$ Vehicles, (v.`name` $\neq$ "Ferrari" $\vee$ v.`owner` $\neq$ p) $\vee$ p.`age` $\geq$ 40

Figure 2: Classes and Integrity Constraints

```
T1 = trans(p1.  p2:  Person) {
        insert p2 in p1.children }
/* this transaction adds a new child to a person */
T2 = trans(p1, p2:  Person) {
        p1.spouse := p2;
        p2.spouse := p1 }
/* this transaction performs a marriage between two persons */
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 */
```

Figure 3: Transactions

4.1 Syntactic analysis of the constraints

4.1.1 Structure

The structural analysis of the constraints is recursively defined as follows :

Υ(exp1 θ exp2) = Υ(exp1) $\cup$ Υ(exp2), where θ denotes any comparator;
$\Upsilon(C) = \bigcup_{C_i \leq C} \{C_i\}$ (the set of all subclasses of C, including itself.)
Υ(exp.`a`) = $\bigcup_{t \leq type(exp)} \{t.\mathtt{a}\} \cup \Upsilon(exp)$, if *type(exp)* is an abstract type
(*type()* is a function which given an expression returns its corresponding type);
= Υ(exp) otherwise.
Υ(x) = Υ(S),
where x represents a quantified variable in the constraint ranging over the set expression S;
Υ(F $\wedge$ F') = Υ(F) $\cup$ Υ(F'), where F and F' are two quantifier free formulæ;
Υ(F $\vee$ F') = Υ(F) $\cup$ Υ(F'), where F and F' are two quantifier free formulæ;
Υ($\neg$ F) = Υ(F), where F is a quantifier free formula;
Υ((F)) = Υ(F), where F is a quantifier free formula;

4.1.2 Behavior

The syntactic analysis of a method requires more details. A method m of signature m@T($\tau_1, .., \tau_{n-1}$):τ_n is applied to an object o of abstract type T in the following way: $o \leftarrow m(x_1, .., x_{n-1})$. The syntactic analysis Φ of the operations Π^m performed in m is defined as follows:

- **Assignment :**
  The syntactic analysis of an assignment "$e_1 := e_2$" is the union of the syntactic analysis of the two expressions e_1 and e_2.
  $\Phi\ (e_1 := e_2) = \Phi(e_1) \cup \Phi(e_2)$
  Indeed, in an assignment "$e_1 := e_2$", the left part (e_1) cannot represent a persistent variable (a variable attached to a persistent root). However, the expression e_1 can be built from a constructor using persistent parameters, we therefore have to analyse the expression e_2.

- **Field extraction :**
  $\Phi(\text{exp.a}) = \Upsilon(\text{exp.a})$ if exp is a variable attached to a persistent root.

  While analysing exp.a, we do not take into account temporary variables (non persistent variables), which appear in the bodies of the methods for computing purposes, nor instances of concrete types, because they are non mutable, non shared. In the case where exp is represented by the key word `self`, which refers to the receiver of the method, *type(*`self`*)* is equal to the abstract type of the receiver.

- **Sequencement :**
  The syntactic analysis of a sequence of elementary statements is the union of the syntactic analysis of all these statements.

  $\Phi(s_1; s_2) = \Phi(s_1) \cup \Phi(s_n)$

- **Iteration loop :**
  The syntactic analysis of an iteration loop is the union of the syntactic analysis of all the statements performed in the loop and of the set on which the iteration holds.
  $\Phi(\text{for(o in x) s}) = \Phi(x) \cup \Phi(\text{s})$

- **Conditional test :**
  The syntactic analysis of a conditional test is the union of the syntactic analysis of the boolean expression b and of the statements s_1 and s_2 which might be performed.
  $\Phi\ (\text{if(b) } s_1 \text{ else } s_2) = \Phi(b) \cup \Phi(s_1) \cup \Phi(s_2)$

  It is necessary to consider the syntactic analysis of the boolean expression (b) for the same reasons as those given above, concerning the iteration set in the case of the syntactic analysis of a conditional test.

- **Set operations :**
  Φ (insert o in x) = Φ(o) $\cup$ Φ(x).
  Φ (drop o from x) = Φ(o) $\cup$ Φ(x).

Φ allows to analyse the body of the methods invoked in the constraints. The methods to execute are dynamically linked to the selector (the name of the method), according to the abstract type of the receiver. For a given selector, it happens that there are several methods belonging to different abstract types, corresponding to different implementations. It is not always possible to determine at compile time which method to link. Thus, the same message passing may have different results, depending on the abstract type of the receiver.
In order to analyse a message passing $o \leftarrow m$, it is necessary to take into account the set of all methods that might be executed, according to the abstract type of the receiver. It is sufficient to consider, for each message passing $o \leftarrow m$, the set of all methods m declared in the subtypes τ_i of the receiver τ_o.

The syntactic analysis of a method call in a constraint corresponds to the syntactic analysis (Φ) of the operations invoked in this method:

$$\Upsilon(o \leftarrow m(x_1, \ldots, x_n)) = \bigcup_{\tau_i \preceq \tau_o} \Phi(\Pi^{m@\tau_i})$$

The syntactic analysis of multiple calls is defined by:

$$\Upsilon(o \leftarrow m_1(x_1^1, \ldots, x_{n_1-1}^1) \leftarrow m_2(x_1^2, \ldots, x_{n_2-1}^2) \leftarrow \ldots \leftarrow m_k(x_1^k, \ldots, x_{n_k-1}^k)) =$$

$$\bigcup_{\tau_i \preceq \tau_o} \Phi(\Pi^{m_1 @ \tau_i}) \cup \bigcup_{\tau_i \preceq \tau^1_{n_1}} \Phi(\Pi^{m_2 @ \tau_i}) \cup ... \cup \bigcup_{\tau_i \preceq \tau^k_{n_{k-1}}} \Phi(\Pi^{m_k @ \tau_i})$$

$\bigcup_{\tau_i \preceq \tau^k_{n_{k-1}}} \Phi(\Pi^{m_k @ \tau_i})$ represents the union of the syntactic analysis of the operations appearing in the methods m_k declared in the type $\tau^k_{n_{k-1}}$ (abstract type of the result of the previous method m_{k-1}) and in the subtypes of $\tau^k_{n_{k-1}}$.

4.1.3 Example

Table 1 represents the results yielded by $\Upsilon()$ for the constraints defined in Figure 2. We detail the syntactic

Constraint	Syntactic Analysis Υ(A)
A_1	`Persons, Person, Person.children`
A_2	`Persons, Person, Person.children`
A_3	`Persons, Person.age`
A_4	`Persons, Person.spouse`
A_5	`Persons, Person.children, Person.age`
A_6	`Persons, Vehicles, Person.age, Vehicle.owner, Vehicle.name`

Table 1: Syntactic analysis of the constraints

analysis of A_1.

(A_1) $\forall$ p $\in$ Persons, p $\leftarrow$`descendants()`$\leftarrow$`matrix()`$\leftarrow$`closure()` $\leftarrow$`non-circuit()`.
We have to analyse the following:

$$\Upsilon(p \leftarrow \texttt{descendants}() \leftarrow \texttt{matrix}() \leftarrow \texttt{closure}() \leftarrow \texttt{non} - \texttt{circuit}())$$

which is:

$$\bigcup_{\tau_i \preceq \texttt{Person}} \Phi(\Pi^{\texttt{descendants}() @ \tau_i}) \bigcup_{\tau_i \preceq \texttt{Graph}} \Phi(\Pi^{\texttt{matrix}() @ \tau_i})$$
$$\bigcup_{\tau_i \preceq \texttt{Matrix}} \Phi(\Pi^{\texttt{closure}() @ \tau_i}) \bigcup_{\tau_i \preceq \texttt{Matrix}} \Phi(\Pi^{\texttt{non-circuit}() @ \tau_i})$$

which yields:

$$\Phi(\Pi^{\texttt{descendants}() @ \texttt{Person}}) \cup \Phi(\Pi^{\texttt{matrix}() @ \texttt{Graph}}) \cup$$
$$\Phi(\Pi^{\texttt{closure}() @ \texttt{Matrix}}) \cup \Phi(\Pi^{\texttt{non-circuit}() @ \texttt{Matrix}})$$

Let us assume that method `descendants()` is implemented as shown in Figure 4. The analysis of the loop

```
{ Graph g;
    Person x;
    g = new (Graph);
    for(x in self.children) {
     insert x <- descendants() in g;
     insert [vertex1:  self, vertex2:  x] in g.edges}
    insert self in g.vertices;
    return(g) }
```

Figure 4: Implementation of method `descendants()`

leads to Υ(`self.children`) which is `Person.children`. The analysis of the code in the loop reduces to the analysis of `g.edges` which is empty as `g` is a temporary variable, and of `[vertex1: self, vertex2: x]` which is `Person`. The analysis of the last assignment is again `Person`.

Furthermore, $\Phi(\Pi^{\texttt{matrix()@Graph}}) \cup \Phi(\Pi^{\texttt{closure()@Matrix}}) \cup \Phi(\Pi^{\texttt{non-circuit()@Matrix}}) = \emptyset$ because no persistent variable (no variable that might be attached to a persistent root) appears in these methods.

The syntactic analysis $\Upsilon(\mathrm{A}_1)$ of the constraint A_1 is thus :

$$\{\texttt{Persons, Person, Person.children}\}.$$

4.2 Syntactic analysis of transactions

The syntactic analysis of a transaction T yields the set $\Psi(T)$ of all paths involved in T. This set represents again "paths" in the database, those paths corresponding to data that might be modified by a transaction.

The syntactic analysis of transactions containing no methods is recursively built as follows :

$\Psi(\mathrm{s}_1;\mathrm{s}_2) = \Phi(\mathrm{s}_1) \cup \Phi(\mathrm{s}_2)$

$\Psi(\text{if(b) s}_1 \text{ else s}_2) = \Phi\ (\text{if(b) s}_1 \text{ else s}_2)$

$\Psi(\text{for(o in x) s}) = \Phi(s)$

$\Psi(\text{exp.a}) = \Upsilon(\text{exp.a})$

In the analysis of Ψ(exp.a), we do not take into account non persistent temporary variables which appear in the body of the methods for computing purpose.

$\Psi\ (\mathrm{e}_1 := \mathrm{e}_2) = \Upsilon(\mathrm{e}_1)$

In an assignment "$\mathrm{e}_1 := \mathrm{e}_2$", the left part ($\mathrm{e}_1$) can represent a variable attached to a persistent root.

$\Psi\ (\text{insert o in x}) = \Phi(\text{x})$

$\Psi\ (\text{drop o from x}) = \Phi(\text{x})$

$\Psi(\ \mathrm{o} \leftarrow \mathrm{m}_1(x_1^1,\ldots,x_{n_1-1}^1) \leftarrow \mathrm{m}_2(x_1^2,\ldots,x_{n_2-1}^2) \leftarrow \ldots \leftarrow m_k(x_1^k,\ldots,x_{n_k-1}^k)) =$
$\Phi(\mathrm{o} \leftarrow \mathrm{m}_1(x_1^1,\ldots,x_{n_1-1}^1) \leftarrow \mathrm{m}_2(x_1^2,\ldots,x_{n_2-1}^2) \leftarrow \ldots \leftarrow m_k(x_1^k,\ldots,x_{n_k-1}^k))$

4.2.1 Example

The analysis of transactions T_1, T_2 and T_3 yields the results given in Table 2.

Transaction	**Syntactic Analysis (Ψ(T))**
T1	`Person, Person.children`
T2	`Person, Person.spouse`
T3	`Persons, Person, Person.age`

Table 2: Syntactic Analysis of Transactions

4.3 Safety detection

While analysing transactions and constraints, we have detected the set of "paths" used in a constraint or invoked by a transaction. This set of paths gathers the various structures that are manipulated by the transactions and the constraints. Intuitively, a transaction T might violate a constraint A if T and A manipulate the same "data structures". This property, presented in [BLR92, BD93] is still valid in this context. It can be expressed the following way :

Property 1 Given a transaction T and a constraint A, the transaction T might violate the constraint A if and only if $\Upsilon(A) \cap \Psi(T) \neq \emptyset$.

Table 3 gathers the set of the constraints which might be violated by a given transaction. Note that T_3 is detected as a transaction that might (potentially) violate the constraint A_6 even if this will never happen since this transaction only increments the attribute `age` of a Person instance. This is, indeed, one limitation of this approach. Such a problem will be solved using more powerful techniques, namely, abstract interpretation of programming languages [CC77].

Transaction	Constraints
T1	A_1, A_2, A_5
T2	A_4
T3	A_3, A_4, A_5, A_6

Table 3: Constraints hit by Transactions

5 Generation of Enforcement Tests

5.1 Restrictions

The problem addressed in this section is to generate, given a constraint, a checking algorithm which guarantees that either the constraint is satisfied at the end of the transaction or the transaction is aborted. In a first approach we restrict our constraints to constraints without methods. We also impose some restrictions on the constraint prefixes allowed.

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

$$\exists \text{ x } \textbf{in} \text{ aSet, M(x)}$$

If a transaction removes an element α from aSet and M(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 remainder of this section, we shall thus consider only universally quantified constraints.

5.2 Constraint checking

The problem of efficiently checking a constraint at the end of a transaction consists of 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 four constraints A_3, A_4, A_5 and A_6, together with transactions T_1, T_2 and T_3 as shown in Figure 5.

(A_3) $\forall$ p $\in$ Persons, p.`age` $\leq$ 130 $\wedge$ p.`age` $\geq$ 0;
(A_4) $\forall$ p $\in$ Persons, p.`spouse.spouse` = p $\vee$ p.`spouse` = **nil**;
(A_5) $\forall$ p $\in$ Persons, $\forall$ c $\in$ p.`children`, p.`age` > c.`age`
(A_6) $\forall$ p $\in$ Persons, $\forall$ v $\in$ Vehicles, (v.`name` $\neq$ “Ferrari” $\vee$ v.`owner` $\neq$ p) $\vee$ p.`age` $\geq$ 40

Figure 5: Integrity Constraints

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_3 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_4 will certainly be violated for them. Of course, collecting those identifiers requires the constraint checking manager to be provided with some kind of “intelligence”. This problem is addressed in [BDS95] and relies on abstract interpretation techniques.

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_3, 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 of 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_3. But in general, the set obtained at execution time only intersects the relevant set, as for constraint A_4. Therefore, we propose to generate checking algorithms which allow us 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 (and inserted in C) 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), in 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 $p_{i,j}$ 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. We show, in Figure 6, how to generate generic checking algorithms.

For each class C_i, we generate the following enforcement test:
$\forall x \in \Delta^{C_i}$
check [$\forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots,$
$M(x_1, \ldots, x, \ldots, x_{i+1}, \ldots)$]
For each path $x.a_1 \ldots a_k$ (either in the prefix or in the matrix),
we generate the following enforcement test:
$\forall x \in C_i$
$\forall y \in \Gamma_{a_1}^{\tau(x)}$, if $y = x$,
check [$\forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots,$
$M(x_1, \ldots, x, \ldots, x_{i+1}, \ldots)$]
...
$\forall y \in \Gamma_{a_k}^{\tau(x.a_1 \ldots a_{k-1})}$, if $y = x \ldots a_{k-1}$,
check [$\forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots,$
$M(x_1, \ldots, x, \ldots, x_{i+1}, \ldots)$]
For each path $y_i.b_1 \ldots b_l$ in the matrix (where y_i ranges in $x.p_i$), we generate
$\forall x \in C_i$
$\forall y \in x.p_i$
$\forall z \in \Gamma_{b_1}^{\tau(y)}$, if $z = y$,
check [$\forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots,$
$M(x_1, \ldots, x, \ldots, y, \ldots, x_{i+1}, \ldots)$]
...
$\forall z \in \Gamma_{b_1}^{\tau(y \ldots b_{l-1})}$, if $z = y \ldots b_{l-1}$,
check [$\forall x_1 \in C_1, \ldots, \forall x_{i+1} \in C_{i+1}, \ldots,$
$M(x_1, \ldots, x, \ldots, y, \ldots, x_{i+1}, \ldots)$]

Figure 6: Generic Checking Algorithm

For a given constraint A, the enforcement test generation consists of generating the above tests for each class C_i involved in the constraint prefix. Let us illustrate this on the constraints, A_3, A_4, A_5 and A_6.
For the constraint A_3,

$$(A_3)\ \forall\ p \in \text{Persons},\ p.\mathtt{age} \leq 130 \wedge p.\mathtt{age} \geq 0;$$

the checking algorithm is

$\forall\ x \in \Delta^{\mathtt{Persons}}$, check $(A_3(x))$
$\forall\ x \in$ Persons
$\forall\ y \in \Gamma_{\mathtt{age}}^{\mathtt{Person}}$, if $y = x$, check$(A_3(x))$

Figure 7: Algorithm for A_3

This can be rewritten as shown in Figure 8.

$\forall\ x \in \Delta^{\mathtt{Persons}}$, check $(A_3(x))$
$\forall\ x \in \text{Persons} \cap \Gamma_{\mathtt{age}}^{\mathtt{Person}}$, check $(A_3(x))$

Figure 8: Optimised Algorithm for A_3

In this case, the set $\Gamma_{\mathtt{age}}^{\mathtt{Person}}$ 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 $\Gamma_{\mathtt{age}}^{\mathtt{Person}}$ 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_4

$$(A_4)\ \forall\ p \in \text{Persons},\ p.\texttt{spouse}.\texttt{spouse} = p \lor p.\texttt{spouse} = \textbf{nil};$$

the checking algorithm is described in Figure 9.

$\forall\ x \in \Delta^{\texttt{Persons}}$, check $(A_4(x))$
$\forall\ x \in$ Persons
    $\forall\ y \in \Gamma^{\texttt{Person}}_{\texttt{spouse}}$,
        if $y = x$, check($A_4(y)$)
    $\forall\ y \in \Gamma^{\texttt{Person}}_{\texttt{spouse}}$,
        if $y = x$.spouse, check($A_4(y)$)

Figure 9: Checking Algorithm for A_4

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

For the constraint A_5

$$(A_5)\ \forall\ p \in \text{Persons},\ \forall\ c \in p.\texttt{children},\ p.\texttt{age} > c.\texttt{age}$$

the checking algorithm is shown in Figure 10.

$\forall\ x \in \Delta^{\texttt{Persons}}$, check $(A_5(x))$
$\forall\ x \in$ Persons
    $\forall\ y \in \Gamma^{\texttt{Person}}_{\texttt{age}}$,
        if $y = x$, check($A_5(y)$)
    $\forall\ y \in \Gamma^{\texttt{Person}}_{\texttt{children}}$,
        if $y = x$, check($A_5(x,y)$)
    $\forall\ y \in$ x.children
        $\forall\ z \in \Gamma^{\texttt{Person}}_{\texttt{age}}$,
        if $z = y$, check($A_5(x,y)$)

Figure 10: Checking Algorithm for A_5

This algorithm iterates over three sets: Persons, $\Gamma^{\texttt{Person}}_{\texttt{age}}$ and $\Gamma^{\texttt{Person}}_{\texttt{children}}$. 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_6,

$$(A_6)\ \forall\ p \in \text{Persons},\ \forall\ v \in \text{Vehicles},\ (v.\texttt{name} \neq \text{“Ferrari”} \lor v.\texttt{owner} \neq p) \lor p.\texttt{age} \geq 40$$

the checking algorithm is illustrated by Figure 11.

$\forall\ x \in \Delta^{\texttt{Persons}}$, check $(A_6(x))$
$\forall\ x \in$ Persons
    $\forall\ y \in \Gamma^{\texttt{Person}}_{\texttt{age}}$,
        if $y = x$, check($A_6(y)$)
$\forall\ x \in \Delta^{\texttt{Vehicles}}$, check $(A_6(x))$
$\forall\ x \in$ Vehicles
    $\forall\ y \in \Gamma^{\texttt{Vehicle}}_{\texttt{name}}$,
        if $y = x$, check($A_6(y)$)
    $\forall\ y \in \Gamma^{\texttt{Vehicle}}_{\texttt{owner}}$,
        if $y = x$ check($A_6(y)$)

Figure 11: Checking Algorithm for A_6

$$
\begin{array}{l}
\forall\ x \in \Delta^{\texttt{Persons}},\ \text{check}\ (A_6(x)) \\
\forall\ x \in \text{Persons} \cap \Gamma^{\texttt{Person}}_{\texttt{age}}, \\
\qquad \text{check}(A_6(x)) \\
\forall\ x \in \Delta^{\texttt{Vehicles}},\ \text{check}\ (A_6(x)) \\
\forall\ x \in \text{Vehicles} \cap (\Gamma^{\texttt{Vehicle}}_{\texttt{name}} \cup \Gamma^{\texttt{Vehicle}}_{\texttt{owner}}) \\
\qquad \text{check}(A_6(x))
\end{array}
$$

Figure 12: Optimised Checking Algorithm for A_6

This algorithm can be rewritten as shown in Figure 12.
This last example deserves some comments: checking A_6 means that we check A_6 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_3 and A_6) that the checking algorithms could be rewritten in an optimised 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_5, because y has to range over `x.children`.

We now give (Figure 13) a general optimised version of this class of algorithms. The union of Γ_i denotes

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

Figure 13: Optimised Algorithms

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

6 Implementation

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 executable programs which allow us to instantiate the constraints while preserving the inclusion semantics. O_2integrity is written in C^{++} and uses *lex* and *yacc*.

6.1 Mapping between Thémis and O_2

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

6.1.1 Atomic types

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

6.1.2 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)

6.1.3 Types and classes

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
Classes	named values

6.1.4 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).

6.2 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. Each time a transaction is compiled, the O_2Integrity preprocessor updates a global table describing which constraints might be violated by the transaction. Meanwhile, the corresponding checking algorithms are generated at the end of the transaction.

The user can visualize the set of constraints defined on the schema and the global table showing the constraints that will be checked for a given transaction. Each time a constraint is actually violated by the execution of a transaction, the user is warned and the transaction is aborted.

7 Conclusion

This work proposes a specification of a database programming language allowing for the definition of integrity constraints in a global and declarative way. The characteristics of the object-oriented data model are taken into account, in particular, inheritance and subtyping. The language used to express the integrity constraints is not limited to first-order logic formulæ, but also includes method calls. This allows an increased declarative power of the constraints.

In order to detect which constraints may be violated by a given transaction, we define a syntactic analysis of both the constraints and the transactions. This analysis takes into consideration the specificities of the object-oriented model, such that inheritance, subtyping, late binding and the persistent nature of the data. It allows us to obtain a necessary and sufficient condition to determine at compile time if a transaction might violate a constraint. A second part of the work presented here, consists in automatically generating constraint checking algorithms at the end of transactions. Those algorithms are generated for a sub-class of formulæ: universally quantified formulæ.

A first prototype of the Thémis language has been implemented. This prototype allows to validate the proposed analysis. We propose to extend our work in the following directions.

The analysis proposed detect transactions as being (potentially) unsafe while they are actually safe. More generally, we would like to refine our static analysis by using abstract interpretation techniques.

In order to be able to generate an efficient constraints checker, we will extend our checking algorithms to constraints including methods and existential quantifiers.

Finally, our last aim is to build a complete compiler for the Thémis language. Such a compiler should be implemented on a persistent object manager (O_2Engine, Napier88 Store).

8 Acknowledgments

We would like to thank P.Y. Policella and P. Tronowski for implementing the first Thémis prototype. We also greatly acknowledge the referees for their enlightening comments and helpful suggestions.

References

[AB87] M. Atkinson and P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, June 1987.

[BD93] V. Benzaken and A. Doucet. Thémis: a database programming language with integrity constraints. In Shasha Beeri, Ohori, editor, *Proceedings of the 4th International Workshop on Database Programming Languages*, Workshop in Computing, pages 243–262, New York City, USA, September 1993. Springer-Verlag.

[BDM88] F. Bry, H. Decker, and R. Manthey. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In Missikoff Schmidt, Ceri, editor, *Proceedings of the EDBT International Conference*, LNCS 303, pages 488–505, 1988.

[BDS95] V. Benzaken, A. Doucet, and X. Schaefer. Integrity Constraint Checking Optimization based on Abstract Databases Generation and Program Analysis. *Journal de l'Ingénierie des Systèmes d'Information*, 1(3):9–29, 1995.

[Bla81] B.T. Blaustein. *Enforcing Database Assertions.* PhD thesis, Harvard University, Computer Science Department, Cambridge, MA, USA, 1981.

[BLR92] V. Benzaken, C. Lécluse, and P. Richard. Enforcing Integrity Constraints in Database Programming Languages. In R. Morrison A. Albano, editor, *Proceedings of the 5th International Workshop on Persistent Object Systems*, Workshop in Computing, pages 282–299, Pisa, Italy, September 1-5 1992. Springer-Verlag.

[BM86] F. Bry and R. Manthey. Checking Consistency of Database Constraints: A Logical Basis. In *Proceedings of the VLDB International Conference*, pages 13–20, August 1986.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types*, LNCS 173, pages 51–67. Springer-Verlag, 1984.

[Car87] L. Cardelli. Basic Polymorphic Type Checking. *Science of Computer Programming*, 1987.

[Car88] L. Cardelli. Structural Subtyping and the Notion of Power Type. In *ACM POPL International Conference*, 1988.

[Cas95a] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3), 1995.

[Cas95b] G. Castagna. A proposal for making O_2 more type safe. Rapport de Recherche liens-95-4, LIENS, March 1995.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th POPL, Los Angeles, CA*, pages 238–252, January 1977.

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[GM79] G. Gardarin and M. Melkanoff. Proving the Consistency of Database Transactions. In *VLDB International Conference*, pages 291–298, Rio, Brasil, October 1979.

[HI85] A. Hsu and T. Imielinski. Integrity Checking for Multiple Updates. In *Proceedings of the ACM SIGMOD International Conference*, pages 152–168, 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 *Proceedings of the VLDB International Conference*, pages 61–70, 1987.

[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 Transaction on Database Systems*, 14(3):322–368, 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):125–136, 1983.