In European Conf. on Object-Oriented Progr. (ECOOP'97), Lecture Notes in Computer Science to appear

Static Integrity Constraint Management in Object-Oriented Database Programming Languages via Predicate Transformers

Véronique Benzaken[1 2] and Xavier Schaefer[1 2]

[1] C.R.I., Université de Paris I - Panthéon - Sorbonne,
12 Pl. du Panthéon, 75005 Paris, France
[2] L.R.I., Université de Paris XI - Orsay, Bat. 490, 91405 Orsay Cedex, France

Abstract. In this paper, we propose an efficient technique to statically manage integrity constraints in object-oriented database programming languages. We place ourselves in the context of an extended version of the O_2 database programming language, and we assume that updates are undertaken by means of methods. An important issue when dealing with constraints is that of efficiency. A naive management of such constraints can cause a severe floundering of the overall system. Our basic assumption is that the run-time checking of constraints is too costly to be undertaken systematically. Therefore, methods that are always safe with respect to integrity constraints should be proven so *at compile time*. The run-time checks should only concern the remaining methods. To that purpose, we propose a new approach, based on *abstract interpretation*, to prove the *invariance* of *integrity constraints* under complex *methods*. We then describe the current implementation of our prototype, reporting many experiments that have been performed with it on non trivial examples.
Though our method is developed in the context of object-oriented database programming languages, it can easily be applied to the problem of static verification of object-oriented languages providing pre and post-conditions such as Eiffel.

1 Introduction

Efforts on object-oriented database programming languages have mainly been devoted to the integration of elaborated type systems and persistence mechanisms in a uniform way.

However, many specific database functionalities have not been taken into account by those works. In particular, functionalities such as view management, role definition and integrity constraints are either absent from, or only roughly supported by most (object-oriented) database programming languages. For the last five years, efforts in the database programming language community have been devoted to the definition of languages integrating such functionalities [AB91, SDA94, ABGO93, ABGO95].

Nevertheless, most object-oriented database programming languages are not able to take into account integrity constraints in a global and declarative way.

A great concern for database users is that the information stored should represent the real world faithfully. The data should respect at any moment a set of conditions, called *semantic integrity constraints*. For example, assume that our database stores persons who have a spouse, a reasonable integrity constraint imposed on these data could be:

No person can be married to itself

Integrity constraint management has always been presented as one of the great promises of database systems. To date, that *basic* facility should be provided by any object-oriented database system. However, no real database system has kept that promise in a satisfactory way.

One of the assumptions that has discouraged people from using integrity constraints is that constraints should be checked *systematically* at run-time, after each update. Despite the several optimisation techniques that have been proposed ([Nic79, LT85, HI85, BM86, KSS87, HCN84, WSK83, BDM88, BD95, Law95]) in order to improve dynamic checking, system performances are still greatly affected. Considering the efficiency problems that most database systems already have without having to deal with integrity constraints, one understands why integrity has more or less been left aside.

The situation can be significantly improved by detecting, at *compile time*, which updates preserve integrity. As in our framework such updates are undertaken by means of methods, a method that will never violate an integrity constraint should be proven safe at compile time. Indeed, the run-time checking of this constraint can be entirely avoided after the execution of this method.

To illustrate our motivation let us consider the simple example suggested previously. We consider a database that stores some information about people: their name and their spouse. The O_2 database schema of such a database could be as follows:

```
class Person inherit Object public type tuple (
                            name: string,
                            spouse: Person
                            )
method marry(q: Person)
end;
```

Consider the following constraint:

"No person can be married to itself."

Such a constraint will be written at the schema level in our extension of O_2:

```
forall x in Persons: x->spouse!=x;
```

where the notation `x->spouse` denotes the extraction of attribute `spouse` of object `x` and where `Persons` is defined in O_2 as the following persistent root:

```
name Persons: set (Person);
```

This command declares an entry point in the database as well as a container for objects which are instances of class `Person`. Let us now consider the following implementation of method `marry`

```
method marry(q:Person) in class Person{
  self->spouse=q;        <set the spouse attribute of the receiver to q>
  q->spouse=self; }      <set the spouse attribute of parameter to self>
```

That update can clearly violate the integrity constraint given previously, in particular when `self` is equal to `q`. The integrity constraint manager will therefore have to check the constraint after each execution of method `marry`. That can be done in several ways. It can either check the constraint as it is, i.e. check that `(x->spouse!=x)` for any `Person x` in the set `Persons`. It can also undertake an optimised run-time check. This test can be obtained with the techniques mentioned previously. In our case, such techniques can reduce the test to just testing that `(self->spouse!=self)` and `(q->spouse!=q)`. An important point is that such an optimisation is possible only for very simple constraints. Most of the time, the run-time checking overhead remains. Moreover, such optimisations are unable to produce the "minimal" run-time test. In our case, the minimal test is just to check that `(self!=q)`.

As far as the programmer is concerned, the best solution, in terms of efficiency, is not to let the system generate the run-time test. The programmer that wants to tune his database should introduce some "minimal" run-time test directly into the implementation of methods. Method `marry` becomes:

```
method marry(q:Person) in class Person {
  if (self!=q){
   self->spouse=q;
   q->spouse=self;}}
```

In that case, the overhead of run-time integrity checking can be avoided, as it is redundant with the precautions that have already been taken by the programmer inside the method. Also, the user does not have to suffer from mysterious program failures (due to constraint violations) anymore. However, we first have to make sure that those precautions are indeed sufficient, i.e. that this method will *never* violate the integrity constraint. That must be proven formally, automatically and at compile-time. The main contribution of this paper is to provide a technique for doing so as well as a prototype implementing it.

However, this technique should *not* be taken as an encouragement for the programmers to take specific precautions with respect to integrity constraints while writing programs. It just allows to *detect* such precautions, as far as they are taken.

Our approach undertakes a very detailed analysis of methods and provides some precise information on the impact they have upon constraints. Partial but reliable information concerning methods is obtained by means of a *predicate transformer*. A predicate transformer is a function that, given a method `m` and a constraint `C` satisfied by the input data of `m`, returns a formula $\overline{\mathtt{m}}(\mathtt{C})$ that is satisfied by the output data of m. In other words, provided C is satisfied before an execution of m (which is the usual assumption as methods are run on a consistent state[3]), $\overline{\mathtt{m}}(\mathtt{C})$ is satisfied after an execution of m. A method m is then safe with respect to constraint C if $\overline{\mathtt{m}}(\mathtt{C}) \Rightarrow \mathtt{C}$ (C is a consequence of $\overline{\mathtt{m}}(\mathtt{C})$). To prove the implication, we use a classical theorem proving technique based on the tableaux method (this method is shortly explained in Section 5.2).

Our approach reduces significantly the number of integrity constraints that have to be checked at run-time. It has been implemented and provides some positive results on some real, non-trivial examples. We believe it can greatly improve the overall efficiency of integrity management. We insist that it is *fully automatic*, in the sense that this technique is embedded in the compiler technology thus no human intervention is required at any stage.

Although our method is developed in the context of object-oriented database programming languages, it can also be applied to the problem of static analysis of object-oriented languages providing pre and post-conditions à-la Eiffel [Mey88, Mey91].

The paper is organised as follows. In Section 2 we informally present our framework: an object-oriented database programming language allowing the definition of integrity constraints in a global and declarative way. In Section 3 we describe an intermediate language in which complex instructions will be translated and give the reasons for such a translation. In Section 4, we define our abstract predicate transformer. We show how to handle simple methods, loops and recursion, and overriding. We describe in Section 5 the implementation of our current prototype and give results of experiments. In Section 6 we compare ourselves with different approaches that have been proposed in the database field. We finally conclude in Section 7.

2 An Object-Oriented Database Programming Language with Constraints

Our framework consists in a very simple extension of the O_2 programming language. The programmer can define, together with the usual classes and methods, some *semantic integrity constraints* which are well formed formulas on a specific language. However, we point out that using the O_2 language is not essential to our approach. We present, through examples, the ingredients of our language: classes and persistent roots, integrity constraints and methods. To focus on important issues, we will reduce the use of concepts that are not essential to our approach.

2.1 Classes and Persistent Roots

In order to illustrate the main concepts, let us start with the simple schema given in Figure 1.

[3] This corresponds to the classical notion of *consistency* in databases.

```
 class Person inherit Object public type tuple(
          name: string,
          spouse: Person,
          bestfriend: Person,
          money: integer )
   method marry(q: Person);
          separate();
          spend(amount: integer);
          setbestfriend(p: Person);
          marry-and-separate(q: Person);
end;

class Parent inherit Person public type tuple(
          children: set (Person))
   method marry(q: Person);
          separate();
end;

class Employee inherit Person  public type tuple(
          boss: Person)
   method separate();
end;

name Persons: set (Person);
name Parents: set (Parent);
name Employees: set (Employee);
```

Fig. 1. A schema

We define three classes (`Person`, `Parent` and `Employee`). Both of them have tuple-structured types. A `Person` has a `name`, a `spouse`, a `bestfriend` and some `money`. The classes `Parent` and `Employee` inherit from `Person`, their respective types are subtypes of the one of class `Person`. Thus attributes `name, spouse, bestfriend` and `money` are inherited in classes `Parent` and `Employee`. Methods `marry` and `separate` have been redefined in class `Parent` and `Employee`. Such a redefinition obeys the covariant rule (see [BC96] for an enlightening discussion on the consequences in O_2 and [Cas96] for a more general framework). It differs from method overloading in languages such as C++ or Java in which methods are redefined on an invariant basis. The semantics of inheritance is the usual inclusion semantics. We also define three persistent names `Persons, Parents` and `Employees`, as a set of `Persons` (respectively `Parents, Employees`). Persistence is achieved by reachability from those persistent names. This means that every object reachable from those names is automatically stored on disk at the end of programs. Unreachable objects are garbage-collected.

2.2 Integrity Constraints

That schema is quite simple but many constraints can already be stated; some examples are:

- There must be at least one `Person` who is not married.
- A `Person` cannot be married to itself.
- No `Parent` can be married to one of its children.
- An `Employee` who is not married must have a boss.
- A `Person` must have some money.
- A `Parent` who is not married cannot have any children.
- There must be at least one `Person` who has no best friend.

All those constraints are given in Figure 2.

```
C1: exists x in Persons: x->spouse==nil;
C2: forall x in Persons: x->spouse!=x;
C3: forall x in Parents: !(x->spouse in x->children);
C4: forall x in Employees: x->spouse==nil ==> x->boss!=nil;
C5: forall x in Persons: x->money > 0;
C6: forall x in Parents: x->spouse==nil ==>!(exists y in x->children);
C7: exists x in Persons: x->bestfriend==nil;
```

Fig. 2. Integrity constraints

We have there a great variety of constraints. The constraints can be structural or on domains, they can involve very simple arithmetic (`C5`), tuples or sets. Most constraints that have several atoms are implication constraints (`C4` and `C6`). We can briefly give a general syntax for constraints. It is simple and close to first order logic. We first define terms, that is, variables like `p` and `q`, constants like `nil` and more complex terms like `p->spouse->spouse`. Atomic formulas can be defined with standard predicates as `==`, `!=`...and complex formulas can be built with the usual connectives "`&&`" (and), "`||`" (or), "`!`" (not), "`==>`" (implies). Quantification is also necessary. So, if "`F`" is a formula, then "`forall p in S: F`" and "`exists p in S: F`" are also formulas. We call that *range restricted quantification*, because the range on which the variable `p` varies is the set `S`. The semantics, i.e. the truth or falsity of those constraints with respect to a specific database should be clear. As usual, a database can be seen as a structure, that is a domain together with a set of relationships between elements of that domain. A database is coherent with respect to an integrity constraint `C` if it is a model of `C`.

2.3 Methods

```
method separate()
in class Person {
    self->spouse=nil;}
```

```
method spend(amount:integer)
in class Person {
  if (self->money-amount>0)
    self->money=self->money-amount;}
```

```
method marry(q:Person)
in class Person{
  if (self!=q){
    self->spouse=q;
    q->spouse=self;}}
```

```
method marry-and-separate(q:Person)
in class Person{
  if (self!=q){
    self->spouse=q
    q->spouse=self;}
  for (u in Persons){
    if ((u!=self) && (u !=q))
      u->spouse=nil;}}
```

```
method setbestfriend(q: Person) in class Person{
    if (exists u in Persons: u!=self && u->bestfriend==nil)
      self->bestfriend=q;}
```

Fig. 3. Implementation of methods of class `Person`

Methods are building blocks which allow to update the persistent data stored in the database. If no precautions are taken by the application programmer, such updates could lead to inconsistent database states.

We give in Figure 3 the implementation of the methods defined on our schema. Those methods are safe with respect to at least one of the constraints previously declared.

Method `separate` in Figure 3 is obviously safe with respect to constraint `C1`: after the execution of `separate`, at least a `Person` `x` is such that `x->spouse==nil`, namely `self`.

As mentioned previously, method `marry` is safe with respect to `C2`. Method `spend` checks prior performing a withdrawal of money that the credit is still positive. Therefore it does not violate constraint `C5`. Method `marry-and-separate` first marries two persons `self` and `q` testing if `self` is different from `q` and then sets the `spouse` attribute of any person different from `self` and `q` to `nil`. Notice that this method performs complex statements such as an `if` embedded in a `loop`. This method is safe with respect to `C2`.

The previous methods are short (as methods often are), but non-trivial. However, the run-time tests that are undertaken are quite simple. Very often, the run-time tests that are necessary to write safe methods are more complicated and involve the use of *boolean queries*. Method `setbestfriend` for example makes use of those queries. Indeed, consider constraint `C7` that says that at least one

Person has no best friend together with the following implementation for method setbestfriend:

```
method setbestfriend(q: Person) in class Person{
     self->bestfriend=q;}
```

That method can clearly violate constraint C7, namely when the receiver is the only person who has no best friend. To make that method safe, we have to test that there is somebody else who has no best friend. This can be done in O_2 by using a classical O_2 boolean query. The previous method made safe becomes:

```
method  setbestfriend(q: Person) in class Person{
  o2 boolean b;
  o2query(b,"exists u in Persons: u !=$1 && u->bestfriend==nil", self);
  if (b) self->bestfriend=q;}
```

That method is now safe with respect to constraint C7. To simplify the notations, we generalize in the following the use of boolean queries and allow them to occur directly inside if's, without using o2query and the working variable b, as shown in Figure 3.

The following methods are defined or overriden for the subclasses Parent and Employee. Method marry in class Parent has been overloaded, it is still safe with respect to C2 but also with respect to constraint C3. Method separate has been redefined in both classes Parent and Employee such as not violating constraints C6 and C4 respectively.

```
method separate()
in class Employee{
  if (self->boss!=nil)
    self->spouse=nil;}
```

```
method separate
in class Parent{
  if (!(exist u in self->children))
    self->spouse=nil;}
```

```
method marry(q:Person) in class Parent{
 if (self != q && !(q in self->children) && !(self in q->children)){
  self->spouse=q;
  q->spouse=self;}}
```

Fig. 4. Implementation of methods for class Parent and Employee

When methods are overriden (as it is the case for methods marry and separate) we shall adopt the following notation in the sequel. We note for any overriden method m, $m = \{m_1, \ldots, m_n\}$ where $m_1, \ldots, m_n$ are all the different implementations of method m in the system. For example, separate is formed by three different implementations those in the classes Person, Employee and Parent.

3 Translation of Methods into an Intermediate Form

The task of analysing computer languages can be difficult, particularly when dealing with object-oriented programming languages. Indeed, in such langages, objects that are involved in an update are reached in a *navigational* way. Defining an abstract interpretation directly on the source language can be a painstaking task, and proving that the abstract interpretation defined is *semantically founded* (which must always be done) can be very difficult. In order to compensate for that extra complexity, our technique is to:

- First, translate the source language into an intermediate form.
- Then, undertake the abstract interpretation on that intermediate form.

We give in this section a short illustration of the complexity problem that occurs in object-oriented langages. This illustration will help justify the intermediate language we consider in the next section. The complexity of instructions in object-oriented languages is due to the way the objects are referenced. Let us see a simple O_2 assignment instruction:

```
p->spouse->spouse=q->spouse
```

That update is not about `p` or `q` but about objects that can be *reached* from `p` and `q`. That instruction groups in fact two radically different operations:

- First, it determines the objects that are used for the update. To that purpose, it *navigates* along the paths defined by the attributes. In our case, the objects that are used in the update are denoted by `p->spouse` and `q->spouse`. Let us call them o_1 and o_2.
- Then, it undertakes the update. The update has become: o_1`->spouse=`o_2.

That decomposition has to be done formally and systematically. To that purpose we will use a simple new instruction:

`forone` *object* `where` *condition* `do` *instruction*.

That instruction executes *instruction*, where *object* denotes the only object that satisfies *condition*. With that new construct, the original instruction:

```
p->spouse->spouse=q->spouse
```

will be translated into:

```
forone o1 where p->spouse=o1
do forone o2 where q->spouse=o2
   do o1->spouse=o2
```

That translation simplifies considerably the matter of defining an abstract interpretation for object-oriented programming languages. Indeed, with such a translation we can avoid defining a complex abstract interpretation for the assignment. We just have to define a simple abstract interpretation of the `forone` and we can consider without loss of generality assignments of the form: $v \to a = v$ instead of the more general form $v \to a = e$ (where e denotes any expression).

The translation of database programming languages into that intermediate language is straightforward. We give in the following definition the syntax of the whole intermediate language we shall use.

Definition 1. *A method is a program* $\mathtt{m}$ = *instruction where* $\mathtt{m}$ *is the* name *of the method and instruction is the* body *of* $\mathtt{m}$ *that is syntactically defined as follows:*

instruction ::= *variable* `->` *attribute* = *variable*
| instruction ; instruction
| { instruction }
| `if` condition `then` instruction
| `forone` variable `where` condition `do` instruction
| `forall` variable `where` condition `do` instruction

This definition deserves the following comments. We only consider instructions there and not methods returning expressions as we are only interested in updates (side effects) which can alter the integrity of data. Therefore, we do not have parameters in our syntax because we do not make any differences between methods parameters and global variables. Following [Cas96], `self` is just the first parameter of methods.

4 Safe Database Schemas via Predicate Transformers

We describe in this section our technique to prove automatically whether methods are consistent with respect to integrity constraints.

Our approach is an instance of abstract interpretation [CC77, CC79, JN95]. The aim of abstract interpretation is to provide, automatically and at compile time, some information on the possible run-time behaviour of a program. This amounts to doing, instead of a real execution of a program on a real environment, an *abstract execution* of that program on an *abstract environment*.

The program analysis framework we develop has been greatly inspired by the works of Dijkstra on predicate transformers [Dij76, DS90]. A (forward) predicate transformer is a function that given a program and a formula describing its input data, produces a formula describing its output data.

The application of predicate transformers to our problem is natural. The input data of a method $\mathtt{m}$ is described by an integrity constraint $\mathtt{C}$. We apply the predicate transformer to $\mathtt{m}$ and $\mathtt{C}$ and get another formula (a post-condition) $\overline{\mathtt{m}}(\mathtt{C})$ describing the output data of $\mathtt{m}$. Formula $\overline{\mathtt{m}}(\mathtt{C})$ is satisfied by any database resulting from an execution of $\mathtt{m}$, provided $\mathtt{C}$ is satisfied before the execution. The main result of this section is that $\mathtt{m}$ cannot violate $\mathtt{C}$ if $\overline{\mathtt{m}}(\mathtt{C}) \Rightarrow \mathtt{C}$ ($\mathtt{C}$ is a consequence of $\overline{\mathtt{m}}(\mathtt{C})$).

Our technique undertakes a very detailed analysis of methods and provides some precise information on the impact they have upon constraints. Hoare and Dijkstra's main interest was to compute a so-called *weakest-precondition* in terms of a *post-condition* and a program. Hoare used some deductive rules to yield that pre-condition, whereas Dijkstra defined a backward predicate transformer[4]. An

[4] As opposed to forward predicate transformers, a backward predicate transformer is a function that, given a program and a formula describing its output data, produces a formula describing its input data

important result of their analyses is that they produce, for simple languages, the most precise information about program behaviour that can be obtained (i.e. strongest post-conditions or weakest pre-conditions). That implies that every property that is true of a program is *provable* (i.e. their method is complete). However, for some complex languages with some sophisticated features such as aliasing, Clarke [Cla79] has shown that completeness cannot be obtained. That has lead us to define an *abstract* forward predicate transformer, as it cannot claim to produce the most-precise information about program behaviour (i.e. strongest post-conditions). Still, it manages to produce some partial, but very useful information (as it is always the case in abstract interpretation). We will see later that this information yielded is sufficient to undertake program verification in even complex cases.

In the rest of this section, we shall first give the formal syntax of our formulas. We will assume without loss of generality that negation can only be applied to atoms[5].

We shall then define the abstract interpretation by induction, both on the structure of the method and the structure of the initial formula. For the sake of clarity, we shall first describe the abstract interpretation of simple instructions. To this end, we first define the predicate transformer for simple methods which do not contain loop instructions and are not overriden. We shall show how it is used to prove method safety. Then, we shall describe the predicate transformer for loops. This also suggests how recursion can be dealt with. Finally, we present the treatment of overriden methods.

4.1 Integrity Constraints

Integrity constraints are well-formed formulas on a first-order language. We use a to range over *attributes* and u, v to range over *variables*. Terms are defined by:

Definition 2. Let V be a set of variables, let C be a set of constants (`0`, `1`, `nil`, etc), the set of terms is inductively defined by:

- For every c in C, c is a term,
- For every $v \in$ V, v is a term
- Let t be a term and let a be an attribute name then $\mathrm{t}{\rightarrow}a$ is a term.

Formulas are constructed as usual by means of comparators ($<$, $>$, $=$, $\neq$) using the connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\forall$ and $\exists$.

4.2 Simple Methods

Definition 3. Let `m` be a method (see Definition 1). We define the *predicate transformer* $\overline{\mathtt{m}}$ of `m`, by induction, as follows ($\overline{\mathtt{m}}$ denotes a formula). Let ϕ and φ be two formulae:

[5] We can use the following rules to ensure that: $\neg(\phi\wedge\varphi) = \neg\phi\vee\neg\varphi$, $\neg(\phi\vee\varphi) = \neg\phi\wedge\neg\varphi$, $\neg\forall x\phi = \exists x\neg\phi$, $\neg\exists x\phi = \forall x\neg\phi$, $\neg\neg\phi = \phi$.

– Formula parse:

1. $\overline{\mathtt{m}}((\phi)) \equiv (\overline{\mathtt{m}}(\phi))$
2. $\overline{\mathtt{m}}(\phi \wedge \varphi) \equiv \overline{\mathtt{m}}(\phi) \wedge \overline{\mathtt{m}}(\varphi)$
3. $\overline{\mathtt{m}}(\phi \vee \varphi) \equiv \overline{\mathtt{m}}(\phi) \vee \overline{\mathtt{m}}(\varphi)$
4. $\overline{\mathtt{m}}(\forall x \phi) \equiv \forall x\ \overline{\mathtt{m}}(\phi)$
5. $\overline{\mathtt{m}}(\exists x \phi) \equiv \exists x\ \overline{\mathtt{m}}(\phi)$

– Method parse:

6. If: $\mathtt{m}\ \equiv u\texttt{->}a\texttt{=}v$ then:
   - If: $\phi \equiv (x \rightarrow a = y)$
     then: $\overline{\mathtt{m}}(\phi) \equiv (u = x \wedge u \rightarrow a = v) \vee (u \neq x \wedge u \rightarrow a = v \wedge x \rightarrow a = y)$
   - If: $\phi \equiv (x \rightarrow a \neq y)$
     then: $\overline{\mathtt{m}}(\phi) \equiv (u = x \wedge u \rightarrow a = v) \vee (u \neq x \wedge u \rightarrow a = v \wedge x \rightarrow a \neq y)$
   - Else: $\overline{\mathtt{m}}(\phi) \equiv \phi \wedge u \rightarrow a = v$

   where ϕ is a literal. If ϕ is not a literal[6], $\overline{\mathtt{m}}(\phi)$ can be computed by parsing ϕ first using (1-5).
7. If: $\mathtt{m} \equiv i_1;i_2$ then: $\overline{\mathtt{m}}(\phi) \equiv \overline{i_2}(\overline{i_1}(\phi))$
8. If: $\mathtt{m} \equiv \texttt{\{}i\texttt{\}}$ then: $\overline{\mathtt{m}}(\phi) \equiv \overline{i}(\phi)$
9. If: $\mathtt{m} \equiv \texttt{if}\ \psi\ \texttt{then}\ i$ then: $\overline{\mathtt{m}}(\phi) \equiv \overline{i}(\psi \wedge \phi) \vee (\neg\psi \wedge \phi)$
10. If: $\mathtt{m} \equiv \texttt{for one}\ v\ \texttt{where}\ \psi(v)\ \texttt{do}\ i$ then: $\overline{\mathtt{m}}(\phi) \equiv \exists v\ \overline{i}(\psi(v) \wedge \phi)$

We assume that there is no ambiguity between symbols. If necessary, we can always rename all the variables and parameters that are used in constraints and methods.

Clause 6 deserves an explanation. Let us assume that the update "$u\texttt{->}a\texttt{=}v$" is undertaken and that the formula "$(x \rightarrow a = y)$" is true before the update. Then, we either have $(u = x)$ or $(u \neq x)$, and a formula that is true after the update is respectively $(u = x \wedge u \rightarrow a = v)$ or $(u \neq x \wedge u \rightarrow a = v \wedge x \rightarrow a = y)$. The result follows: it is the disjunction of those two formulae. The other cases are similar.

We now have to prove formally that the abstract interpretation of a method gives some reliable information about the real behaviour of that method, i.e. that it is *correct*. In the following, we will have to interpret formulae with free variables. To that purpose, we will write $\mathcal{B}_\sigma$ the interpretation $\mathcal{B}$ provided with the assignment of free variables σ.

Theorem 4. Let $\mathcal{B}_{\mathtt{old}}$ be a database, $\mathtt{m}$ be a method, and $\mathcal{B}_{\mathtt{new}}$ be the same database after an execution of $\mathtt{m}$. Let ϕ be a formula, and σ be an assignment for the free variables of ϕ. We have:

$$\mathcal{B}_{\mathtt{old}\sigma} \models \phi \implies \mathcal{B}_{\mathtt{new}\sigma} \models \overline{\mathtt{m}}(\phi)$$

that is, the abstract interpretation $\overline{\mathtt{m}}$ of $\mathtt{m}$ is correct

[6] A *literal* is either an atom or the negation of an atom. An *atom* is an expression $a(t_1, ..., t_n)$, where a is a relation and t_1,...,t_n are terms.

$$
\begin{tikzcd}
\phi \arrow[rr, "\overline{\mathtt{m}}"] & & \overline{\mathtt{m}}(\phi) \\
\mathcal{B}_{\mathtt{old}\,\sigma} \arrow[u, dash, "\models"] \arrow[rr, "\mathtt{m}"] & & \mathcal{B}_{\mathtt{new}\,\sigma} \arrow[u, dash, "\models"']
\end{tikzcd}
$$

Fig. 5. Predicate transformer diagram

This relationship is illustrated by the diagram in Figure 5. That theorem is proven by induction on both formulas and instructions. The proof can be found in [BS95]. We have already said that we need some correct information about the run-time behaviour of programs. Moreover, that information must be precise enough to be useful. After all, we could define a predicate transformer that is correct, but does not provide any useful information. For instance: $\overline{\mathtt{m}}(\phi) = \mathtt{true}$ for any method $\mathtt{m}$ and any formula ϕ. As it will be shown in Section 5, our predicate transformer is precise enough to allow us to prove method correctness for many non trivial examples.

We can now use that predicate transformer to tell whether a method might violate an integrity constraint. The following theorem is a consequence of Theorem 4.

Theorem 5. Let C be a constraint and $\mathtt{m}$ be a method. $\mathtt{m}$ is safe with respect to C if $\overline{\mathtt{m}}(C) \Rightarrow C$.

The proof of this theorem can also be found in [BS95].

To establish that method $\mathtt{m}$ is safe with respect to C, we have to prove that $\overline{\mathtt{m}}(C) \Rightarrow C$. We point out that powerful methods have been developed in the field of automated theorem proving in first-order logic: resolution, paramodulation, tableaux method... We use the tableaux method exposed in [Fit90] because it does not impose any special normal form for the formula being processed. A property of all those proof techniques is that they are *complete*: if a formula is true, then those proof techniques find a proof for the formula in a finite time. The real problem occurs when the formula is *not* true. In that case, the theorem prover keeps on looking for a proof whereas none exists! Practically, theorem provers always look for a proof in a limited search space. So, our technique will:

– detect as being safe some safe methods (depending on the search space devoted to the computation).
– detect as being unsafe some safe methods and all unsafe methods.

4.3 Methods with Loops or Recursion and Overridden Methods

We show in this section how to deal with loops and recursion. For the sake of brevity, we shall focus on the loop. The generalisation to recursion should be clear. Database programming languages allow the use of a specific loop. In our kernel language, it is defined as:

$$\texttt{forall}\ \textit{variable}\ \texttt{where}\ \textit{condition}\ \texttt{do}\ \textit{instruction}$$

The "`forall`" instruction executes "*instruction*" for every element, denoted by "*variable*", that satisfies "*condition*". A first approach is to look for the solution of the usual equation for loops[7]:

$$\overline{\texttt{forall}}(\phi) = \overline{\textit{instruction}}(\overline{\texttt{forall}}(\phi))$$

The solution of that equation is the usual least fix-point of function $\overline{\texttt{forall}}$. However, in general, that fix-point cannot be computed (the computation may not terminate). We therefore have to undertake an analysis that is less precise, but which terminates, and yields some useful information. We use a technique, that allows to reach an approximation of the fix-point in a finite time. Intuitively, the idea is that if "*instruction*" is safe with respect to a constraint, then the overall "`forall`" instruction will also be safe provided that safety was guaranteed prior entering the loop (i.e., the constraint is a logical consequence of the current value of the predicate transformer). Suppose we are proving the safety of a method `m` with respect to a constraint C, and that `m` contains a loop. We define the predicate transformer of that loop as follows:

$$\overline{\texttt{forall}}(\phi) = \begin{cases} C & \text{if } \phi \Rightarrow C \text{ and } \overline{\textit{instruction}}(C) \Rightarrow C \\ \texttt{true} & \text{otherwise.} \end{cases}$$

Each time we encounter a loop with a formula ϕ as input, we either replace it by C if ϕ implies C and the instructions contained in the loop are safe with respect to C, or by `true` otherwise. The abstract interpretation is still correct. Therefore the following holds:

Theorem 6. Let $\mathcal{B}_{\texttt{old}}$ be a database, `m` be a method with loop statements, and $\mathcal{B}_{\texttt{new}}$ be the database obtained after an execution of `m`. Let ϕ be a formula, and σ be an assignment for the free variables of ϕ. We have:

$$\mathcal{B}_{\texttt{old}\sigma} \models \phi \implies \mathcal{B}_{\texttt{new}\sigma} \models \overline{\texttt{m}}(\phi)$$

that is, the abstract interpretation $\overline{\texttt{m}}$ of `m` is correct.

The proof of this theorem relies on the fact that the composition of correct abstract interpretations is still correct. Again we have:

[7] We give a simplified version of $\overline{\texttt{forall}}(\phi)$. In the exact version, we have to take into account "*variable*" and "*condition*".

Theorem 7. Let C be a constraint and `m` be a method with loops statements. `m` is safe with respect to C if $\overline{\mathtt{m}}(C) \Rightarrow C$.

In order to deal with loops, we have to include in the process of building our predicate transformer some theorem proving steps. Again for the same reasons that have been exposed in the previous section, for some safe methods with loops, safety cannot be proven. Safe methods can be detected as such if loops start from a coherent state (i.e. $\phi \Rightarrow C$ can be proven in the search space), and each iteration map a coherent state to a coherent state (i.e. $\overline{instruction}(C) \Rightarrow C$ can be proven in the search space). So, even when dealing with loops, we still have an efficient method for proving method safety. A similar method can be defined with respect to recursion.

We have described in this section a *program analysis framework* for the language defined in the previous section. We also have defined a very precise criterion to tell if a method can violate an integrity constraint. Our machinery can fail in proving the safety of some safe methods, but it will never manage to prove the safety of a method which is not safe. It errs on the side of caution (our technique is sound but not complete). On the basis of first experiments performed on our current prototype and reported in Section 5, we are confident that the proofs we are dealing with are simple and that many safe methods will be detected as such.

The application of our machinery to overridden methods is straightforward. Intuitively, an overridden method is safe if every branch defined for it is safe.

Theorem 8. Let $\mathtt{m} = \{\mathtt{m}_1, \ldots, \mathtt{m}_n\}$ be an overloaded method. Let C be an integrity constraint, `m` is safe with respect to C if the following holds

$$\forall i \in \{1, \ldots, n\}, \overline{\mathtt{m}_i}(C) \Rightarrow C.$$

5 Implementation

We present in this section the prototype that has been implemented. We are currently experimenting its behaviour on some real examples. We will give the results that we have obtained on the examples provided earlier. The prototype is in fact a pre-processor for O_2. It provides O_2 with an integrity constraint management facility. The integrity manager works with:

- an O_2 database schema together with a set of integrity constraints.
- a set of methods written in O_2.

For the sake of simplicity, the prototype only considers a significant subset of O_2 (namely, the instructions for which we have defined a predicate transformer). However, it can be extended to accept the whole O_2 language. We show in Figure 6 the general structure of the integrity constraint manager, and explain in the following the role of each part.

- First, a simple syntactical technique is used to show that some methods are safe with respect to some constraints. That component is a preliminary filter

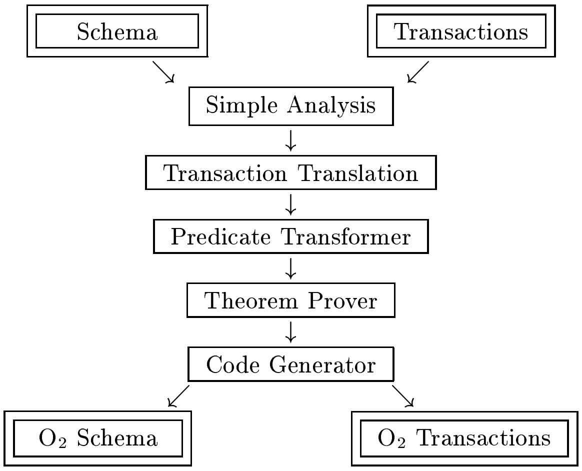

Fig. 6. structure of the integrity constraint manager

that only considers type information from the constraints and the methods. Roughly, it compares the types of the objects that are updated in each method with the types of the objects that are "constrained" by each constraint. This method has been presented with full details in [BD93, BD95]. It reduces the task of the next steps of the analysis in a significant way, and at a very low cost.

- After that, for methods that have not been proven safe, the more sophisticated method of *predicate transformers* is applied in different steps:
  1. Methods and constraints are decomposed. In particular, methods are translated into the intermediate form to make the task of abstract interpretation easier.
  2. For each method and each constraint, the predicate transformer is applied. Which method has to be checked with which constraint is determined by the first step mentioned previously.
  3. An automatic first-order theorem prover based on the tableaux method is applied to the result provided by the predicate transformer.
  4. The code generator finally takes into account the results of the analysis to generate the O_2 schema (if needed) and the O_2 methods. Checking algorithms are automatically generated if the safety proof for a method has failed. Those can be found in [BD93, BD95].

We will develop in the following several important points about the whole process. The first show how the predicate transformer is put into practice in the prototype. This will provide an explanation as to why the tableaux method has been chosen. We will then present briefly that method and give the results of our experiment.

5.1 Implementing the Predicate Transformer: from Theory to Practice

The predicate transformer as described in Section 4 is not directly suited to automation. For the sake of brevity, we have not given any details concerning the size of the formulas generated. Our experiment has shown that those formulas can be quite big, but that also the situation can be greatly improved by the use of appropriate heuristics. We use a very simple and pragmatic method to reduce the size of the output formulas. The first obvious way to do that is to simplify the input formula. The input formula can be simplified when it includes redundancies or contradictions. Another improvement, which is less trivial, is to include explicitly in the formula some facts that are implicit in that formula. This increases the size of the input formula but has the consequence of decreasing the size of the output formula. We insist that all those manipulations preserve the semantics of formulas.

Normalisation First, we systematically put the input formula into disjunctive normal form. It is first put into prenex form (i.e. quantifiers come first), and the matrix (i.e. the part of the formula that is not quantified) is put into disjunctive normal form. Therefore, each formula is of the form:

$$Q_1 x_1, ... Q_n x_n, E_1 \vee ... \vee E_m$$

where the Q_i's are quantifiers and the E_i's, which we shall call *environments*, are a conjunction of literals ($L_1 \wedge ... \wedge L_k$). The cost of normalisation is not too expensive, as the initial formula (the constraint) is usually small, and each step of the analysis keeps the formula normalised.

Dealing with Contradictions and Redundancies Once the formula has been normalised, we make a simple analysis of each environment to get rid of contradictions and redundancies. The search for those is very limited (we would otherwise need a theorem prover). The redundancies we deal with are very simple: when a literal occurs twice in an environment, one occurrence can obviously be removed. Moreover, when a contradiction is detected in an environment E, E can be removed from the disjunction of environments. We point out that such a removal should be done *before* applying the predicate transformer. The reason for that is illustrated below. Consider the following environment:

$$E \;\equiv\; (\mathtt{p}{\rightarrow}\mathtt{spouse}{=}\mathtt{q} \;\wedge\; \mathtt{p}{\rightarrow}\mathtt{spouse}{\neq}\mathtt{q})$$

That environment is contradictory ($E \Leftrightarrow \mathtt{false}$). It should therefore be discarded in the list of environments (because $A \vee \mathtt{false} \Leftrightarrow A$). Let us see what would happen if it was not discarded. Consider the following instruction i:

$$i \;\equiv\; \mathtt{r\text{-}\!>spouse=s}$$

The result of the predicate transformer on i and E, $\bar{i}(E)$, is given below:

$$((\texttt{r=p} \land \texttt{p}\rightarrow\texttt{spouse=s}) \lor (\texttt{r}\neq\texttt{p} \land \texttt{p}\rightarrow\texttt{spouse=q} \land \texttt{r}\rightarrow\texttt{spouse=s}))$$
$$\land$$
$$((\texttt{r=p} \land \texttt{p}\rightarrow\texttt{spouse=s}) \lor (\texttt{r}\neq\texttt{p} \land \texttt{p}\rightarrow\texttt{spouse}\neq\texttt{q} \land \texttt{r}\rightarrow\texttt{spouse=s}))$$

It is not a contradiction, and cannot be removed from the list of resulting environments. However, it results from an initial situation which was impossible, and should therefore not have been considered. Much efficiency can therefore be gained when reducing the initial formula. The rules that are used to detect contradictions are the axioms of equality (reflexivity, symmetry, transitivity) and functions ($x = y \Rightarrow f(x) = f(y)$). For the sake of efficiency, the search space for contradictions is voluntarily limited.

Making the Input Formula more Explicit Another improvement can be obtained by including in the input formula some information that is implicit in that formula. For instance, consider the following environment:

$$E \equiv (\texttt{q}\rightarrow\texttt{spouse=p} \land \texttt{q}\rightarrow\texttt{spouse=q}),$$

and the instruction i:

$$i \equiv \texttt{p->spouse=q}$$

The result of the predicate transformer is given below:

$$((\texttt{p=q} \land \texttt{p}\rightarrow\texttt{spouse=q}) \lor (\texttt{p}\neq\texttt{q} \land \texttt{p}\rightarrow\texttt{spouse=q} \land \texttt{q}\rightarrow\texttt{spouse=p}))$$
$$\land$$
$$((\texttt{p=q} \land \texttt{p}\rightarrow\texttt{spouse=q}) \lor (\texttt{p}\neq\texttt{q} \land \texttt{p}\rightarrow\texttt{spouse=q} \land \texttt{q}\rightarrow\texttt{spouse=q}))$$

After simplification (applying normalisation and discarding contradictions and redundancies), this leads to:

$$((\texttt{p=q} \land \texttt{p}\rightarrow\texttt{spouse=q})$$
$$\lor$$
$$(\texttt{p}\neq\texttt{q} \land \texttt{p}\rightarrow\texttt{spouse=q} \land \texttt{q}\rightarrow\texttt{spouse=p} \land \texttt{q}\rightarrow\texttt{spouse=q}))$$

However, a smaller formula can be obtained if we include in the input formula some of its consequences. For instance we can deduce (by transitivity of equality) from the previous environment E the following one, which is equivalent:

$$E' \equiv (\texttt{q}\rightarrow\texttt{spouse=p} \land \texttt{q}\rightarrow\texttt{spouse=q} \land \texttt{p=q}),$$

The result of the predicate transformer for E' and i is:

$$((\texttt{p=q} \land \texttt{p}\rightarrow\texttt{spouse=q}) \lor (\texttt{p}\neq\texttt{q} \land \texttt{p}\rightarrow\texttt{spouse=q} \land \texttt{q}\rightarrow\texttt{spouse=p}))$$
$$\land$$
$$((\texttt{p=q} \land \texttt{p}\rightarrow\texttt{spouse=q}) \lor (\texttt{p}\neq\texttt{q} \land \texttt{p}\rightarrow\texttt{spouse=q} \land \texttt{q}\rightarrow\texttt{spouse=q}))$$
$$\land$$
$$(\texttt{p=q} \land \texttt{p}\rightarrow\texttt{spouse=q})$$

Again after simplification this reduces to:

`(p=q ∧ p→spouse=q)`

This shows how one can greatly improve the process of generating our post-condition, yielding a formula which is much more compact and can be dealt with more efficiently. Our experiment has shown that in some cases, it has allowed to reduce and simplify the output formula in a drastic way (from a formula with hundreds of literals to a formula with just one literal). Notice that all the optimisation techniques presented here are obviously valid and are automatically undertaken by the prototype.

5.2 The Tableaux Method

This section provides the main reason why the automatic theorem proving technique chosen is the tableaux method. The predicate transformer produces a formula which is (nearly) in disjunctive normal form. The resolution method requires that the formulas be in conjunctive normal form (after skolemisation). The cost of going from one normal form to another is so high that it cannot be done in a reasonable time. That is why we need to use a method that does not require the formulas to be in clausal form. The tableaux method is a refutation method. In order to prove that $F \Rightarrow G$, that method proves in fact that $F \wedge \neg G$ is unsatisfiable (contradictory). In order to do that, it develops a tree, called a tableau, which has several branches. A tableau is of the form:

$$[\text{branch}_1, \text{branch}_2, ..., \text{branch}_n]$$

Each branch is of the form:

$$[\text{formula}_1, \text{formula}_2, ..., \text{formula}_m]$$

In fact, each branch is a conjunction of formulas, and a tableau is a disjunction of branches. Note that any formula can appear on branches (they don't have to be normalised). A tableau is developed with *tableau expansion rules*. The tree is expanded only up to a certain "depth". Then, a "test for closure" is done on that tree, which amounts to an instantiation of free variables that reduces the tableau to a contradiction. The reader can find more details about automatic theorem proving techniques in [Fit90].

5.3 Practical Experiments

The prototype that is implemented consists of approximately 4000 lines of C code. We have chosen C for the sake of efficiency, although Prolog or Lisp might have been better languages for formula manipulation. The syntactic analysis of the input files is done with Lex and Yacc. The actual code is not very complicated as the abstract predicate transformer approach is well suited to implementation. It is very modular, and much of it consists of well-known procedures such a basic formula manipulation (normalisation, substitutions...). We give below the post-condition that is generated by the predicate transformer for method `marry` in class `Person`:

```
method marry(q:Person) in class Person{
  if (self!=q){
    self->spouse=q;
    q->spouse=self;}}
```

The pre-condition is constraint `C2`:

```
C2: forall x in Persons: x->spouse!=x;
```

The prototype uses a Lisp-like structure for formulas. For instance, constraint `C2` is written:

```
(forall x (|| (notin x Persons) (!= (spouse x) x)))
```

Terms like `x->spouse` are written `(spouse x)`, formulas like `(x==y)` are written `(== x y)`, etc. The post-condition is:

```
(forall v0 (forall v1 (||
(&& (== (spouse q) self) (== (spouse self) q) (notin v0 Persons)
(!= self q) (!= self q))
(&& (== (spouse q) self) (== (spouse self) q) (!= (spouse v0) v0)
(!= self q) (== v0 self) (!= v0 q) (!= self q) (!= v0 q))
(&& (== (spouse q) self) (== (spouse self) q) (!= (spouse v0) v0)
(!= self q) (!= v0 self) (!= self q) (== v0 q) (!= v0 self))
(&& (== (spouse q) self) (== (spouse self) q) (!= (spouse v0) v0)
(!= self q) (!= v0 self) (!= self q) (!= v0 q))
(&& (notin v1 Persons) (== self q))
(&& (!= (spouse v1) v1) (== self q)))))
```

The tableaux method is then applied to prove that the post-condition implies the constraint. Our experiment has been done on a Sun sparc (sun4m) server running Solaris 2.3 OS. The complete safety proof takes less than 1 second. The integrity constraint manager has been run on all the examples given above. The results of that experiment are given Figure 7.
The times given are only a rough guide of the efficiency of the method. They show that, in many cases, a proof can be obtained quite fast (see method `separate` with constraint `C1`). In some other cases, although quite simple, the proof might need some extra time (for instance, method `spend` with constraint `C5`).The reasons for such differences between proofs are not clear. A further study should focus on the assessment of the cost of the analysis. However, we insist that our method is run at *compile-time* and once and for all. Therefore, such costs can be afforded by the system.

6 Related Work

The study of integrity constraints in database systems has been a topic of large interest [Sto75, Nic79, GM79, CB80, LT85, HI85, BM86, KSS87, HCN84, WSK83, BDM88, SS89, Law95].

method	class	constraint	time
`separate`	`Person`	`C1`	0.18s.
`separate`	`Employee`	`C1`	0.27s.
`separate`	`Parent`	`C1`	0.29s.
`separate`	`Parent`	`C6`	1.23s.
`separate`	`Employee`	`C4`	1.10s.
`marry`	`Person`	`C2`	0.46s.
`marry`	`Parent`	`C2`	1.38s.
`marry`	`Parent`	`C3`	4s.
`spend`	`Person`	`C5`	1.32s.
`marry-and-separate`	`Person`	`C2`	0.65s.
`setfriend`	`Person`	`C7`	0.24s.

Fig. 7. Results.

Several works concern the optimisation of checking: [Nic79, HI85] for relational databases, [BM86] for deductive databases, [BD95] for object-oriented databases, and [Law95] for deductive object-oriented databases.

In the context of active database systems, triggers are used to express integrity constraints [CW90, SKdM92, CFPT94, CCS94]. A trigger is composed by an event e, a condition c and an action a. If e occurs and if c is satisfied then the action a is executed. Depending from the context, the language used for expressing actions is more or less powerful (in [CCS94], message passing, for example, is possible).

However, triggers are far from the simple and declarative tool that integrity constraints should have been. Indeed, the definition and verification of constraints are merged in a single unit (the trigger). Integrity is spread out among several triggers and therefore the global vision of the application semantics is lost.

The subject of finding consistency proofs for transactions is not new: [GM79] gives some indications on how Hoare's logic could be used for that purpose. The invariance of integrity constraints is proven, by hand, for several examples. However, the ideas exposed are not formalised and no systematic procedure for obtaining such proofs is given.

Besides, [Cla79] has spotlighted the deficiencies of Hoare's logic for languages with complex features such as aliasing, side effects... that are shared by most database programming languages.

[CB80] attempt to give a logical method to assess database transaction semantics in general, with a particular interest for integrity constraints. Their work is closely related to [GM79], but uses dynamic logic. However, the problems about the formalisation and automation of the process remain. The work of [Qia90] is in the line of [GM79, CB80], and presents an axiom system for transaction analysis. However, the applicability of such an axiom system is unclear. [Qia93] asserts that conventional methods are not suited to automatic transaction safety verification and takes the opposite direction of *transaction synthesis*. [Qia93] justifies that new direction in the following way: "The generation of unnecessary constraint checking and enforcement code should be avoided. It should be noticed that such optimisation is out of reach for conventional program optimisation techniques, because conventional program optimisers do not have the ability to reason about integrity constraint." We show in this article that such is not the case. Conventional program optimisation techniques such as *abstract interpretation* can deal with that problem in an efficient way.

More recently, [Law95] has defined a way to obtain some *weakest-preconditions* for a very simple language, in the context of deductive and object oriented databases. The choice of that language is somewhat unrealistic as it does not allow basic constructs such as `if`'s. His work focuses on checking the constraints at run-time, rather than proving at compile-time that constraints will be preserved by transactions. Instead of checking the constraint after the execution of the transaction, he checks the pre-condition before the execution, which avoids subsequent roll-backs.

[SS89] is, to our knowledge, the most in-depth application of the axiomatic method to our problem. It uses a complex theorem prover that uses Boyer-Moore's computational logic, based on higher-order recursive functions. Their theorem prover is in fact an expert system which uses a knowledge base where some properties of database constructs are kept. Some theorems are derived and the knowledge base grows. A complex strategy, with many heuristics, is used to improve the search process. Their knowledge base is first filled with axioms about basic language constructs. The main problem with this approach is that the consistency of those axioms is not guaranteed, particularly in the case of classical set theory (as mentioned by the authors). If those axioms are not consistent, then the expert system could prove just anything. Their approach is based on the confidence that no "surprising" properties can be derived from the axioms and is therefore not formally founded.

Our work differs from the other works on the subject of consistency proofs for transactions, in that we use the technique of *abstract interpretation*. The main advantage of this technique is that it focuses on operational issues rather than on mathematical relationships. Abstract interpretation is an elegant and unified framework to describe program semantics with algorithms that are very well suited to automation. Most other works (except that of [SS89]) attempt to clarify the semantics of updates in databases by linking it to logic. However, they do not suggest an effective way of undertaking program verification. Our method is, to our knowledge, the first approach based on first-order logic that has been implemented.

7 Conclusions

We have presented an efficient technique to deal with integrity constraints in object-oriented database programming languages that provide integrated facilities, as the extended version of O_2. This technique focuses on a compile-time checking of constraints. It first attempts to determine formally whether methods are consistent with integrity constraints. If such is the case, the cost of unnecessary run-time checking can be avoided. For the remaining methods (those who might violate integrity constraints), appropriate run-time tests are automatically inserted according to [BD95]. Our approach uses combined abstract interpretation and automated theorem proving techniques. Abstract interpretation provides, via an *abstract predicate transformer*, some partial information about the possible run-time behaviour of methods. This information is then dealt with by the theorem prover. No human intervention is needed, therefore the whole process is fully automatic.

A prototype which undertakes this method has already been implemented and provides some good results on some non-trivial examples. Ongoing experimentation focuses on the refinement and optimisation of the method. It should also give more information about the cost of this method.

Our approach is not restricted to the problem of statically handling integrity constraints in object-oriented database programming languages. It can also be adapted to static analysis of object-oriented languages providing pre and post-conditions such as Eiffel [Mey88, Mey91].

In our future research, we plan to develop some other abstract interpretation techniques. Particularly, we are currently developing a backward predicate transformer which, as opposed to the one presented in this article, will generate a pre-condition (rather than a post-condition). We hope that this latter analysis will allow us to detect some safe methods which are not handled by the forward predicate transformer described in this paper. Combining both analysis should yield some useful results. In particular, we believe that the backward analysis can be used for automatic repair of unsafe methods.

In the long term we plan to apply our method to dynamic constraints and to this end explore the domain of modal theorem proving techniques.

Acknowledgements

We would like to thank Giorgio Ghelli and Atsushi Ohori who first recommended us to study abstract interpretation theory, Giuseppe Castagna for his suggestions and many valuable comments for improving this paper, and Serenella Cerrito, for reading an early version of this paper. We would also like to thank the reviewers for their precious remarks.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *ACM International Conference on Management of Data (SIGMOD)*, pages 238–248, Denver, Colorado, USA, May 1991.

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *International Conference on Very Large Databases*, pages 39–52, 1993.

[ABGO95] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. Fibonacci: a programming language for object databases. *VLDB Journal*, 4(3):403–444, July 95.

[BC96] John Boyland and Giuseppe Castagna. Type-safe compilation of covariant specialization: a practical case. In *ECOOP'96*, number 1008 in Lecture Notes in Computer Science, pages 3–25. Springer, 1996.

[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.

[BD95] V. Benzaken and A. Doucet. Thémis: A Database Programming Language Handling Integrity Constraints. *VLDB Journal*, 4(3):493–518, 1995.

[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.

[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.

[BS95] V. Benzaken and X. Schaefer. Abstract interpretation and predicate transformers: an application to integrity constraints management. Technical Report 1002, L.R.I, Université de Paris XI, 1995.

[BS96] V. Benzaken and X. Schaefer. Ensuring efficiently the integrity of persistent object systems via abstract interpretation. In S. Nettles and R. Connor, editors, *Seventh International Workshop on Persistent Object Systems*, Cape May, U.S.A, June 1996. Morgan-Kaufmann. To appear.

[Cas96] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkäuser, Boston, 1996. ISBN 3-7643-3905-5.

[CB80] M. A. Casanova and P. A. Bernstein. A formal system for reasonning about programs accessing a relational database. *ACM Transactions on Database Systems*, 2(3):386–414, July 80.

[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.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th popl, San Antonio, Texas*, pages 269–282, January 1979.

[CCS94] C. Collet, T. Coupaye, and T. Svenson. Naos - efficient and modular reactive capabilities in an oodbms. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB)*, pages 132–144, Santiago, Chile, September 1994.

[CFPT94] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active rule management in chimera. Morgan Kaufmann, 1994.

[Cla79] E. M. Clarke. Programming languages constructs for which it is impossible to obtain good hoare axiom systems. *Journal of the ACM*, 26(1):129–147, January 79.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the VLDB International Conference*, pages 566–577, Brisbane, Australia, August 90.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Fit90] Melvin Fitting. *First-order logic and automated theorem proving*. Texts and monographs in computer science. Springer-Verlag, 1990.

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

[HCN84] L. Henschen, W. Mc Cune, and S. Naqvi. Compiling constraint checking programs from first order formulas. In H. Gallaire, J. Minker, and J.M. Nicolas, editors, *Advances in Database Theory*, volume 2. Plenum, 1984.

[HI85] A. Hsu and T. Imielinski. Integrity Checking for Multiple Updates. In *Proceedings of the ACM SIGMOD International Conference*, pages 152–168, 1985.

[JN95] N. D. Jones and F. Nielson. Abstract interpretation. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, chapter 5, pages 527–636. Oxford Science Publication, 1995.

[KSS87] R. Kowalski, F. Sadri, and P. Soper. Integrity Checking in Deductive Databases. In *Proceedings of the VLDB International Conference*, pages 61–70, 1987.

[Law95] Michael Lawley. Transaction safety in deductive object-oriented databases. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases, Singapore*, number 1013 in LNCS, pages 395–410. Springer-Verlag, Dec 1995.

[LT85] J. W. Lloyd and R. W. Topor. A basis for deductive database systems. *Journal of Logic Programming*, 2(2), 1985.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall International Series, 1988.

[Mey91] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

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

[Qia90] Xiaolei Qian. An axiom system for database transactions. *Information Processing letters*, 36:183–189, November 1990.

[Qia93] Xiaolei Qian. The deductive synthesis of database transactions. *ACM Transactions on Database Systems*, 18(4):626–677, December 1993.

[SDA94] C. Sousa, C. Delobel, and S. Abiteboul. Virtual schemas and bases. In M. Jarke, J. Bubenko, and K Jeffery, editors, *International Conference on Extending Database Technology*, number 779 in LNCS, pages 81–95. Springer-Verlag, 1994.

[SKdM92] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational dbms. In *Proceedings of the Eighteenth International Conference on Very Large Database (VLDB)*, pages 315–326, Vancouver, British Columbia, August 1992.

[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.

This article was processed using the LaTeX macro package with LLNCS style