Static Management of Integrity in Object-Oriented Databases: Design and Implementation

Véronique Benzaken[1] and Xavier Schaefer[1]

C.R.I., Université de Paris I - Panthéon - Sorbonne,
90, Rue de Tolbiac, 75013 Paris, France
e-mail {benzaken,schaefer}@univ-paris1.fr.

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 a simplified database programming language, close to O_2, in which we assume that updates are undertaken by means of methods. An important issue when dealing with constraints is that of efficiency. A naïve 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 the use of predicate transformers combined with automatic theorem proving techniques, to prove the *invariance* of *integrity constraints* under complex *methods*. We then describe the current implementation of our prototype, and report some experiments that have been performed with it on non trivial examples.
The counterpart of the problem of program verification is that of program correction. Static analysis techniques can also be applied to solve that problem. We present a systematic approach to undertake the automatic correction of potentially unsafe methods. However, the advantages of the latter technique are not as clear as those of program verification. We will therefore discuss some arguments for and against the use of method correction.

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.

Integrity constraints reflect the user's concern 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 some persons who have a spouse. A reasonable integrity constraint imposed on this 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 (systematically generated by the compiler) 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. A database schema of such a database could be as follows:

```
class Person 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 a first-order like language:

```
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 as the following persistent root:

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

This command declares an entry point in the database, which is a container for objects that 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`$\neq$`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 [Nic79, LT85, HI85, BM86, KSS87, HCN84, WSK83, BDM88, BD95, Law95]. In our case, such techniques can reduce the test to just testing that `(q.spouse`$\neq$`q)` and `(self.spouse`$\neq$`self)`.

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`$\neq$`q)` before performing the assignment.

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;}}
```

Note that taking such precautions can be dangerous because, as constraints can evolve, the programmer might also have to change the appropriate tests he has included in his programs. However, in real systems, this style of programming called *defensive programming* is often used, and therefore has to be taken into account.

If the programmer has already taken appropriate precautions with respect to integrity inside the programs, the overhead of run-time integrity checking can be avoided, as it is redundant with those precautions. 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 first contribution of this paper is to provide a technique for doing so as well as a prototype implementing it.

We insist that our aim is *not* to encourage defensive programming, but as this style of programming is widely used in real systems, it should be detected at compile-time to improve the efficiency of integrity management. Note that if defensive programming is *not* allowed (or at least ill-advised), we provide a tool for detecting the programs where defensive programming is used, to give proper warning to the programmer. However, whatever the programming discipline, understanding the semantics of database updates is important, as well as providing automatic tools for doing so.

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 *predicate transformers* [Dij76,

DS90]. A predicate transformer is a function that takes for input a formula and a program (in our case, a method), and yields another formula. There are two kinds of predicate transformers:

- A *forward* predicate transformer is a function that, given a method m and a constraint C satisfied by the input data of m (which is the usual assumption as methods are run on a consistent state[1]), returns a formula $\overrightarrow{m}\ (C)$ that is necessarily satisfied by the output data of m.
- A *backward* predicate transformer is a function that, given a method m and a constraint C, returns a sufficient condition on the input data of m for C to be satisfied by the output data. This condition is denoted $\overleftarrow{m}\ (C)$.

Both predicate transformers can be used to detect at compile-time that a method m will never violate a constraint C. Using the forward predicate transformer, as done in [BS97b], we have to prove that $\overrightarrow{m}\ (C) \Rightarrow C$. If we use the backward predicate transformer, we have to prove that $C \Rightarrow \overleftarrow{m}\ (C)$. To prove both implications, we use classical theorem proving techniques (the tableau method and resolution).

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 and no human intervention is required at any stage. We will of course compare the results obtained by the two predicate transformers.

Another important question is whether there is a systematic way of automatically correcting the faulty version of `marry`, to yield a safe version of `marry`. The answer to this question is positive, and the *backward* predicate transformer can be used to this purpose. Consider a method m that might violate an integrity constraint C. The backward predicate transformer generates in fact a sufficient condition on the input data of this method for it to be safe with respect to C. The implementation of m can be modified to include that preliminary condition. However, the advantages of the latter technique are not as clear as for the former one. Its assessment is still the subject of ongoing research, so we will give some arguments for and against its use in a real system.

In [BS97b], we presented a preliminary version of the forward predicate transformer. In this paper, we extend this preliminary work in the following way. The forward predicate transformer takes into account recursive methods. We present the backward one which was neither defined nor implemented. We choose to present both transformers as we think that it is interesting to have a complete view of the technique as well as a comparison of the respective merits of both of them.

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 two predicate transformers. We

[1] This corresponds to the classical notion of *coherence* in databases.

shall purposely focus on the *forward* predicate transformer. Most remarks about the forward predicate transformer can be applied to the backward predicate transformer. We describe in Section 5 the implementation of our current prototype and give results of experiments. In particular, we will compare the results provided by the two predicate transformers. In section 6, we present a technique to undertake the automatic correction of unsafe methods. In Section 7 we compare ourselves to different approaches that have been proposed in the database field. We finally conclude in Section 8.

2 An object-oriented database programming language with constraints

Our framework consists in a very simple database model with object-oriented features very close to O_2 [Deu90]. The programmer can define, together with the usual classes and methods, some *semantic integrity constraints* which are well formed formulae on a specific language. However, we point out that the specificities of the data model are 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 of our framework, let us start with the simple schema given in Figure 1. 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). 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.

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

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

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

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

Fig. 1. A schema

- A `Person` must have some money.
- A `Parent` who is not married cannot have any children.
- At least one `Person` has no best friend.

All those constraints are given on Figure 2. Constraints are closed first-order formulae built on the following language:

- The symbols includes constants (`0`, `1`,..., `nil`, etc.), and variables (x, y, etc.).
- Terms are built as follows: if t is a term, and a an attribute, then $t.a$ is a term. It denotes the extraction of field a of object t.
- Formulae are built in the usual way by using connectives ($\wedge$, $\vee$, $\neg$, etc.) and quantifiers (`forall`, `exists`).

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 B is coherent with respect to an integrity constraint C if it is a model of C, written $B \models C$.

```
C1: exists x in Persons: x.spouse=nil;
C2: forall x in Persons: x.spouse≠x;
C3: forall x in Parents: x.spouse ∉ 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 ∈ x.children);
C7: exists x in Persons: x.bestfriend=nil
```

Fig. 2. Integrity constraints

```
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 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`

2.3 Methods

Methods are building blocks which allow to update the 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 in class `Person`. Those methods are safe with respect to at least one of the constraints previously declared. Appart from method `separate` defined in class `Person`, which is "naturally" safe, all those methods use defensive programming.

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 (the object that was denoted by) `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`.

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. In any database programming language, this can be done by using a classical *boolean query*. Boolean queries are simple conditions in which the use of quantifiers is allowed. The safe version of setbestfriend, which includes such a query, is given in Figure 3. Note that the test which is embedded in the safe version of setbestfriend looks like constraint C7. However, it is not precisely the same and the whole problem is to show that this test is sufficient to ensure that the following update will lead to a consistent state. We point out that even if this test was *exactly* the same as constraint C7, it would not be sufficient to ensure that the update preserves C7. What counts is for C7 to be checked *after* the update, and therefore to prove that the boolean query embedded in setbestfriend is sufficient.

The methods of Figure 4 are defined or overridden 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 to violate constraints C6 and C4 respectively. Method spend in class Parent withdraws an amount "amount" from the account of a person, and also for each of its children. The tests undertaken are similar than the one in method spend in class Person. Notice that this method performs complex statements such as an if embedded in a loop. This method is safe with respect to C5.

When methods are overridden (as it is the case for methods marry and separate) we

```
method separate()
in class Employee{
  if (self.boss≠nil)
    self.spouse:=nil;}
```

```
method separate
in class Parent{
  if (¬(exist u ∈ self.children))
    self.spouse:=nil;}
```

```
method marry(q:Person)
in class Parent{
    if (self ≠ q
      ∧ ¬(q ∈
self.children)
      ∧ ¬(self ∈
q.children)){
    self.spouse:=q;
    q.spouse:=self;}}
```

```
method spend (amount:integer)
in class Parent{
  if (self.money-amount>0)
    self.money=self.money-amount;
  forall u where (u in self.children) do {
       if (u.money-amount>0)
         u.money=u.money-amount;}}
```

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

shall adopt the following notation in the sequel. We note for any overridden method m,

$m = \{m_1, ..., m_n\}$ where m_1, ..., m_n are all the different implementations of method m in the system. For example, `separate` has three different implementations, respectively 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 in which objects that are involved in an update are reached in a *navigational* way. Defining predicate transformers directly on the source language can be a painstaking task and proving that they are correctly defined (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 static analysis on that intermediate form.

We give in this section a short illustration of the complexity problem that occurs in object-oriented languages. 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 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` o_1 `where p.spouse=`o_1
`do forone` o_2 `where q.spouse=`o_2
  `do` o_1`.spouse:=`o_2

That translation simplifies considerably the matter of defining a static analysis for object-oriented programming languages. Indeed, with such a translation we can avoid defining a complex analysis for the assignment. We just have to define how to deal with the `forone` and we can consider assignments of the form:

$$\text{“}variable.attribute := variable\text{”}$$

instead of the more general form[2]:

$$\text{“}expression.attribute := expression\text{”}.$$

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* $m(p_1, ..., p_n)$ =instruction *where* m *is the* name *of the method,* p_1*,...,*p_n *are its parameters, and* instruction *is the* body *of* 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

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. Following [Cas96], `self` can be considered as the first parameter of methods. To avoid heavy notation, we have not mentioned the class for which a method is defined. An abuse of notation we will make quite freely is to identify a method with its body, i.e., the list of instructions it executes. This has no consequences when methods are not overloaded. We will deal with overloaded methods in a specific section.

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. The program analysis framework we develop has been greatly inspired by the works of Dijkstra on predicate transformers [Dij76, DS90] and of course the seminal work on Hoare logic [Hoa69]. It can also be seen as 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*. In fact, abstract interpretation, Hoare's logic and predicate transformers are not separate disciplines (although they are often presented as such). They just focus on different aspects of program semantics. The framework of abstract interpretation is interesting, as it allows us to focus on practical issues such as the applicability of our method, rather than on theoretical issues.

The central concept of our approach is that of predicate transformers. We define in the following two predicate transformers and show how to use them to undertake

[2] Handling more complex statements is the subject of ongoing research.

automatic program verification. The forward predicate transformer has for input a pre-condition ϕ and a method m and generates a post-condition $\overrightarrow{m}\ (\phi)$. The backward predicate transformer has for input a post-condition ϕ and a method m and generates a pre-condition $\overleftarrow{m}\ (\phi)$. The correctness of the two predicate transformers can be formulated as follows:

- Provided ϕ is true before the execution of m, $\overrightarrow{m}\ (\phi)$ is true after.
- Provided $\overleftarrow{m}\ (\phi)$ is true before the execution of m, ϕ is true after.

The application of predicate transformers to our problem is natural. A method m cannot violate C if $\overrightarrow{m}\ (C) \Rightarrow C$ or $C \Rightarrow \overleftarrow{m}\ (C)$.

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. An 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 led us to define *abstract* predicate transformers, as they cannot claim to produce the most-precise information about program behaviour (i.e., strongest post-conditions or weakest pre-condition). Still, they manage 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.

We will purposely present the *forward* predicate transformer with more details that the backward predicate transformer. Most remarks that we make about the former can be easily applied to the latter.

4.1 A forward predicate transformer.

In this section, we define a forward predicate transformer, by induction, both on the structure of the methods and the structure of the initial formulae. For the sake of clarity, we shall first describe how to deal with simple instructions. To this end, we first define the predicate transformer for simple methods which do not contain loop instructions and are not overridden. We shall show how it is used to prove method safety. Then, we shall describe the predicate transformer for loops and recursive methods. Finally, we present the treatment of overridden methods. For the time being, we will consider that methods are not overridden. This will allow us to identify a method with its body.

Simple methods As our work involves first-order logic with equality, we will use the "$=$" symbol to denote equality in the formulae we manipulate. We will use the "$\equiv$" symbol to denote equality in the meta-language.

Definition 2. *Let m be a method (see Definition 1). We define the* predicate transformer *$\vec{m}$ of m, by induction, as follows ($\vec{m}$ denotes a formula). Let ϕ and ψ be formulae. Let u, v, x and y be terms and a be an attribute:*

- *Formula parse:*
  1. $\vec{m}\ ((\phi)) \equiv (\vec{m}\ (\phi))$
  2. $\vec{m}\ (\phi \wedge \psi) \equiv \vec{m}\ (\phi) \wedge \vec{m}\ (\psi)$
  3. $\vec{m}\ (\phi \vee \psi) \equiv \vec{m}\ (\phi) \vee \vec{m}\ (\psi)$
  4. $\vec{m}\ (\texttt{forall x}:\ \phi(\texttt{x})) \equiv \texttt{forall x} : \vec{\texttt{m}}\ (\phi(\texttt{x}))$
  5. $\vec{m}\ (\texttt{exists x}:\ \phi(\texttt{x})) \equiv \texttt{exists x} : \vec{\texttt{m}}\ (\phi(\texttt{x}))$
- *Method parse:*
  6. *If:* $m \equiv u.a{:=}v$ *and* ϕ *is a literal*[3] *, then:*
     - *If* $\phi \equiv (x.a = y)$ *then* $\vec{m}\ (\phi) \equiv (u = x \wedge u.a = v) \vee (u \neq x \wedge u.a = v \wedge x.a = y)$
     - *If* $\phi \equiv (x.a \neq y)$ *then* $\vec{m}\ (\phi) \equiv (u = x \wedge u.a = v) \vee (u \neq x \wedge u.a = v \wedge x.a \neq y)$
     - *Else:* $\vec{m}\ (\phi) \equiv \phi \wedge u.a = v$

     With (1-5), we can assume without loss of generality that ϕ is a literal.
  7. *If:* $m \equiv i_1;i_2$ *then:* $\vec{m}\ (\phi) \equiv \vec{i_2}\ (\vec{i_1}\ (\phi))$
  8. *If:* $m \equiv \{i\}$ *then:* $\vec{m}\ (\phi) \equiv \vec{i}\ (\phi)$
  9. *If:* $m \equiv$ "$\texttt{if}\ \psi\ \texttt{then}\ i$" *then:* $\vec{m}\ (\phi) \equiv \vec{i}\ (\psi \wedge \phi) \vee (\neg\psi \wedge \phi)$
  10. *If:* $m \equiv$ "$\texttt{for one}\ v\ \texttt{where}\ \psi(v)\ \texttt{do}\ i$" *then:* $\vec{m}\ (\phi) \equiv \texttt{exists v} : \vec{\texttt{i}}\ (\psi(\texttt{v}) \wedge \phi)$

This definition allows to capture (partially) the semantics of updates. For example, point (6) first item above only states that if the instruction to be executed is an assignement of the form *u.a := v* and if the pre-condition satisfied before the assignement is of the form $x.a = y$ then you are faced with two distinct cases: either x equals u and therefore the effect of executing the assignement will be captured by *u = x* $\wedge$ *u.a = v*, or x is different from u and in this case the effect of the assigment will be $x \neq u \wedge u.a = v \wedge x.a = y$.

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. In the following, we will have to interpret formulae with free variables. To that purpose, we will write $\mathcal{B}_\sigma$ the database $\mathcal{B}$ provided with the assignment of free variables σ.

Theorem 3. *Let $\mathcal{B}_{\texttt{old}}$ a database, m a method, and $\mathcal{B}_{\texttt{new}}$ the result of the execution of m on $\mathcal{B}_{\texttt{old}}$. Let ϕ a formula, and σ an assignment for the free variables of ϕ. We have:*

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

that is, the predicate transformer $\vec{m}$ of m is correct.

[3] 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.

The proof of this theorem is given in [BS97a].
This relationship is illustrated by the diagram in Figure 5.

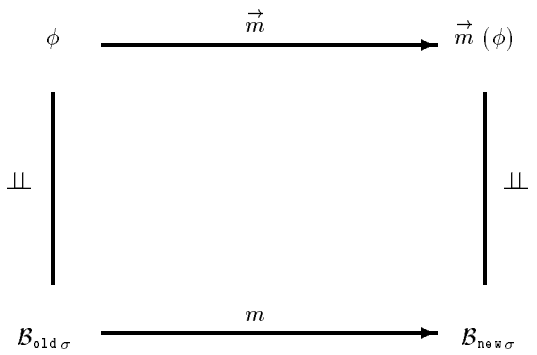

Fig. 5. Predicate transformer diagram

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: $\vec{m}\ (\phi) = \mathtt{true}$ for any method 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 if a method can never violate an integrity constraint. The following corollary is a consequence of Theorem 3.

Corollary 4. Let C be a constraint and m be a method. If $\vec{m}\ (C) \Rightarrow C$ then m is safe with respect to C .

To establish that method m is safe with respect to C, we have to prove that $\vec{m}\ (C) \Rightarrow C$. We point out that powerful methods have been developed in the field of automated theorem proving in first-order logic: resolution, paramodulation, tableaux method...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.

Methods with loops Database programming languages allow the use of a specific loop. In our kernel language, it is defined as:

`forall` *variable* `where` *condition* `do` *instruction*

The "`forall`" instruction executes "*instruction*" for every element, denoted by "*variable*", that satisfies "*condition*". The way we define a predicate transformer is very similar to the way Hoare logic deals with "`while`" loops. It amounts to computing an "invariant" for the loop instruction, that is, a formula which is preserved by the loop. Intuitively, the idea is that if "*instruction*" is safe with respect to a constraint, then the overall "`forall`" instruction will also be safe (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 l defined as $l \equiv$"`forall` x `where` F `do` i". We define the predicate transformer of that loop as follows:

$$\vec{l}\ (\phi) = \begin{cases} C & \textbf{if } \phi \Rightarrow \mathbf{C} \textbf{ and } \vec{\mathbf{i}}\ (\mathbf{C}) \Rightarrow \mathbf{C} \\ \texttt{true} & \textbf{otherwise.} \end{cases}$$

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 predicate transformer is still correct. Therefore the following holds:

Theorem 5. *Let $\mathcal{B}_{\texttt{old}}$ be a database, m be a method with loops statements, and $\mathcal{B}_{\texttt{new}}$ the database obtained after an execution of m. Let ϕ a formula, and σ an assignment for the free variables of ϕ. We have:*

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

that is, the predicate transformer $\vec{m}$ of m is correct.

The proof of this theorem relies on the fact that the composition of correct predicate transformers is still correct and can also be found in [BS97a]. Note that the effect of a loop may depend on the order in which the elements a_1, ..., a_n are chosen. However, the predicate transformer provides some reliable information which is independent from any order. Again we have:

Corollary 6. Let C be a constraint and m be a method with loops statements.
m is safe with respect to C if $\vec{m}\ (C) \Rightarrow C$.

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 maps a coherent state to a coherent state (i.e., $\vec{i}\ (C) \Rightarrow C$ can be proven in the search space). Those restrictions actually correspond to the way database loops are usually used.

Recursion and overridden methods As far as recursion is concerned, we adapt the usual techniques that have been defined in the field of static analysis. For the sake of clarity, we will present the theory on which our approach is based in a less abstract way than what is usually done. The drawback of this description is its lack of generality. However the reader interested in a general description of the static analysis of recursive procedures can consult [Sch86] and [Apt81].

To oversimplify, we consider that any recursive method m can be written in the following way:

$$\begin{array}{l}\texttt{method}\, m\ (\textit{parameters})\ \texttt{in class}\ \kappa \\ \quad \{\texttt{if}\ \textit{base case} \\ \quad\ \texttt{then}\ i \\ \quad\ \texttt{else}\ \textit{recursive call}\}\end{array}$$

Where "*base case*" denotes the test to stop recursion and "*recursive call*" denotes a call to m with an object that can be reached from "`self`" or any parameter of m. We define the predicate transformer of m as follows:

$$\vec{m}\ (\phi) = \begin{cases} C \quad \textbf{if}\ \phi \Rightarrow \mathbf{C}\ \textbf{and}\ \vec{\mathbf{i}}\ (\mathbf{C}) \Rightarrow \mathbf{C} \\ \texttt{true}\ \textbf{otherwise}. \end{cases}$$

The predicate transformer is still correct. Therefore the following holds:

Theorem 7. *Let $\mathcal{B}_{\texttt{old}}$ be a database, m be a recursive method, and $\mathcal{B}_{\texttt{new}}$ the database obtained after an execution of m. Let ϕ a formula, and σ an assignment for the free variables of ϕ. We have:*

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

The proof can be found in [BS97a]. Again we have:

Corollary 8. Let C be a constraint and m be a recursive method.
m is safe with respect to C if $\vec{m}\ (C) \Rightarrow C$.

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

Theorem 9. *Let $m = \{m_1, \ldots, m_n\}$ be an overloaded method. Let C be an integrity constraint, m is safe with respect to C if the following holds*

$$\forall i \in \{1, \ldots, n\}, \vec{m_i}\ (C) \Rightarrow C.$$

We would like the reader to notice that our static analysis ensures modularity. Indeed, if one add a new branch for a method *m*, provided that this new branch is in turn safe, this update will not alter the safety of those programs which were calling *m*. No further verifications are required.

We have described in this section a *program analysis framework* for the language defined in the previous section. We have also 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.

4.2 A backward predicate transformer

In this section, we define a backward predicate transformer. As opposed to the forward predicate transformer, a backward predicate transformer is a function that, given a program m and a formula ϕ, yields a sufficient condition on the input data of m in order for the output data to satisfy ϕ. We will define this predicate transformer and show how it can also be used to prove that methods preserve constraints. We will give less details about the backward predicate transformer than we have done for the forward one. In particular, we will not show how to deal with loops and recursion. The approach used for the forward predicate transformer can be extended quite simply to the backward predicate transformer.

Definition 10. *Let m be a method (see Definition 1). We define the* backward predicate transformer $\overleftarrow{m}$ *of m, by induction, as follows. Let ϕ and ψ formulae. u, v, x and y are terms and a is an attribute:*

- *Formula parse:*
  1. $\overleftarrow{m}\ ((\phi)) \equiv (\overleftarrow{m}\ (\phi))$
  2. $\overleftarrow{m}\ (\phi \wedge \psi) \equiv \overleftarrow{m}\ (\phi) \wedge \overleftarrow{m}\ (\psi)$
  3. $\overleftarrow{m}\ (\phi \vee \psi) \equiv \overleftarrow{m}\ (\phi) \vee \overleftarrow{m}\ (\psi)$
  4. $\overleftarrow{m}\ (\texttt{forall x}:\ \phi(\texttt{x})) \equiv \texttt{forall x} : \overleftarrow{\texttt{m}}\ (\phi(\texttt{x}))$
  5. $\overleftarrow{m}\ (\texttt{exists x}:\ \phi(\texttt{x})) \equiv \texttt{exists x} : \overleftarrow{\texttt{m}}\ (\phi(\texttt{x}))$
- *Method parse:*
  6. *If:* $m \equiv u.a{:=}v$ *and* ϕ *is a literal, then:*
     - *If* $\phi \equiv (x.a = y)$ *then* $\overleftarrow{m}\ (\phi) \equiv (u = x \wedge v = y) \vee (u \neq x \wedge x.a = y)$
     - *If* $\phi \equiv (x.a \neq y)$ *then* $\overleftarrow{m}\ (\phi) \equiv (u = x \wedge v \neq y) \vee (u \neq x \wedge x.a \neq y)$
     - *Else:* $\overleftarrow{m}\ (\phi) \equiv \phi$

     With (1-5), we can assume without loss of generality that ϕ is a literal.
  7. *If:* $m \equiv i_1;i_2$ *then:* $\overleftarrow{m}\ (\phi) \equiv \overleftarrow{i_1}\ (\overleftarrow{i_2}\ (\phi))$
  8. *If:* $m \equiv \{i\}$ *then:* $\overleftarrow{m}\ (\phi) \equiv \overleftarrow{i}\ (\phi)$
  9. *If:* $m \equiv \texttt{if}\ \psi\ \texttt{then}\ i$ *then:* $\overleftarrow{m}\ (\phi) \equiv (\psi \wedge \overleftarrow{i}\ (\phi)) \vee (\neg\psi \wedge \phi)$
  10. *If:* $m \equiv$ "$\texttt{for one}\ v\ \texttt{where}\ \psi(v)\ \texttt{do}\ i$" *then:* $\overleftarrow{m}\ (\phi) \equiv \texttt{exists v} : \psi(\texttt{v})\ \overleftarrow{\texttt{i}} \wedge (\phi)$

The correctness property holds for the backward predicate transformer. It consists in establishing the opposite relationship from the one we showed for the forward predicate transformer.

Theorem 11. *Let $\mathcal{B}_{\mathtt{old}}$ a database, m a method, and $\mathcal{B}_{\mathtt{new}}$ the same database after an execution of m. Let ϕ a formula, and σ an assignment for the free variables of ϕ. We have:*

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

that is, the predicate transformer $\overleftarrow{m}$ of m is correct.

The proof is also given in [BS97a].

We can now state a new safety criterion, which is symmetrical to the one already stated. This criterion gives us potentially a new way of proving that a method is safe with respect to a constraint.

Corollary 12. Let C be a constraint and m be a method. m is safe with respect to C if $C \Rightarrow \overleftarrow{m}\ (C)$.

In the next section, we will compare both approaches (forward and backward) to prove that a method preserves a constraint.

5 Implementation

We present in this section the prototype that has been implemented. In fact, there are two prototypes. The first (resp. second) one uses the forward (resp. backward) predicate transformer. We are currently experimenting their behaviour on some real examples. We will give the results that we have obtained on the examples given earlier. This will provide us with a practical comparison of the forward and backward predicate transformers. The prototypes are in fact pre-processors for O_2. They provide 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_2C.

For the sake of simplicity, the prototypes only consider a significant subset of O_2C (namely, the instructions for which we have defined a predicate transformer). However, they can be extended to accept the whole O_2 language. We show in Figure 6 the general structure of an 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 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 [BD95, BS96]. 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 using *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.

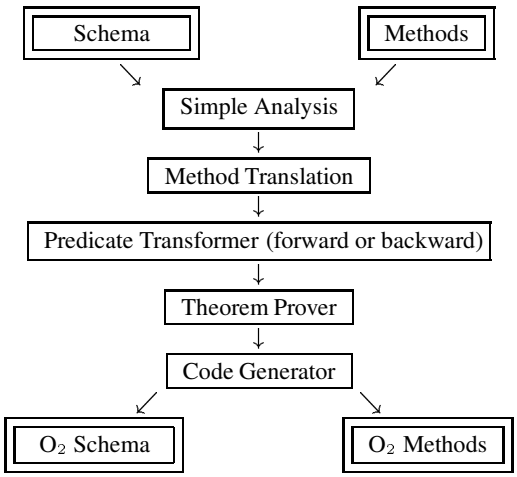

Fig. 6. structure of the integrity constraint manager

2. For each method and each constraint, the predicate transformer (forward or backward) 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 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 [BD95].

We will develop in the following several important points about the whole process. The first show how the predicate transformers are put into practice in the prototypes.

5.1 Implementing the predicate transformers: from theory to practice

The predicate transformers as described in Section 4 are not directly suited to automation. For the sake of brevity, we have not given any details concerning the size of the formulae generated. Our experiment has shown that those formulae 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 formulae. 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 formulae.

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:

$$E \equiv (\texttt{p.spouse=q} \wedge \texttt{p.spouse}\neq\texttt{q})$$

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

$$i \equiv \texttt{r.spouse:=s}$$

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

$$((\texttt{r=p} \wedge \texttt{p.spouse=s}) \vee (\texttt{r}\neq\texttt{p} \wedge \texttt{p.spouse=q} \wedge \texttt{r.spouse=s}))$$
$$\wedge$$
$$((\texttt{r=p} \wedge \texttt{p.spouse=s}) \vee (\texttt{r}\neq\texttt{p} \wedge \texttt{p.spouse}\neq\texttt{q} \wedge \texttt{r.spouse=s}))$$

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. This remark also holds if we use the backward predicate transformer. 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:

$$E \equiv \texttt{(q.spouse=p} \wedge \texttt{q.spouse=q)},$$

and the instruction *i*:

$$i \equiv \texttt{p.spouse:=q}$$

The result of the forward predicate transformer is given below:

```
((p=q ∧ p.spouse=q)∨(p≠q ∧ p.spouse=q ∧ q.spouse=p))
∧
((p=q ∧ p.spouse=q)∨(p≠q ∧ p.spouse=q ∧ q.spouse=q))
```

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

```
((p=q ∧ p.spouse=q)∨(p≠q ∧ p.spouse=q ∧ q.spouse=p ∧ q.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:

$$E' \equiv \texttt{(q.spouse=p} \wedge \texttt{q.spouse=q} \wedge \texttt{p=q)},$$

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

```
((p=q ∧ p.spouse=q)∨(p≠q ∧ p.spouse=q ∧ q.spouse=p))
∧
((p=q ∧ p.spouse=q)∨(p≠q ∧ p.spouse=q ∧ q.spouse=q))
∧
(p=q ∧ p.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. This technique can also be applied to the backward predicate transformer. 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.

Some other optimisations of the same kind have been implemented. They require a redefinition of the predicate transformers so that they can take into account the semantics of the input formulae. A specific ordering has also been defined on the evaluation of the predicate transformers (the definitions of the predicate transformers are essentially non-deterministic, so they can be evaluated in different ways). This ordering is "satisfactory", in the sense that it produces smaller formulae, with less quantifiers. For the sake of brevity, we will not describe these extra improvements, as they follow the spirit of those that have been presented previously. The reader interested could find details in [Sch97].

5.2 Automatic theorem proving techniques

While attempting to prove that a method m is safe w.r.t. a constraint C, we either have to prove that $\overrightarrow{m}\ (C) \Rightarrow C$, or that $C \Rightarrow \overleftarrow{m}\ (C)$. In both cases, we have to use some automated theorem proving techniques. An important result of our experiment is that the theorem prover suited to the backward analysis is not the same as the one suited to the forward one.

As mentioned previously, for optimisation reasons, the predicate transformers have to keep the formulae they manipulate in disjunctive normal form. The result provided by both predicate transformers ($\overrightarrow{m}\ (C)$ and $\overleftarrow{m}\ (C)$) is also (nearly) in disjunctive normal form. Proving that $\overrightarrow{m}\ (C) \Rightarrow C$ or $C \Rightarrow \overleftarrow{m}\ (C)$ is respectively proving that $\overrightarrow{m}\ (C) \wedge \neg C$ or $C \wedge \neg\ \overleftarrow{m}\ (C)$ is unsatisfiable. The first formula is nearly in disjunctive normal form, whereas the second one is nearly in conjunctive normal form. This is a problem as all theorem proving techniques require that the formulae be put into some specific normal form. Resolution accepts clausal form (conjunctive normal form). The tableaux method undertakes a kind of run time normalisation into disjunctive normal form. Also, the cost of changing the normal form of a formula is prohibitive. We therefore have to use two different theorem provers: a resolution based theorem prover for the backward analysis, and a tableaux method theorem prover for the forward analysis. The resolution theorem prover is Otter [McC], and the tableaux theorem prover is taken from [Fit90].

5.3 Practical experiments

The prototype that is implemented consists of approximately 6000 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 formulae. For instance, constraint `C2` is written:

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

Terms like `x.spouse` are written `(spouse x)`, formulae 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 and resolution are then applied to prove that the post-condition (pre-condition) implies (is implied by) the constraint respectively. Our experiment has been done on a Sun sparc (sun4m) server running Solaris 2.3 OS. The integrity constraint manager has been run on all the examples given above. The results of that experiment are given Figure 7.

method	class	constraint	forward	backward
separate	Person	C1	0.21s.	0.12s.
separate	Employee	C1	0.27s.	0.23s.
separate	Parent	C1	0.29s.	0.57s.
separate	Parent	C6	1.21s.	2.48s.
separate	Employee	C4	1.06s.	0.22s.
marry	Person	C2	0.41s.	0.33s.
marry	Parent	C2	1.30s.	9.04s
marry	Parent	C3	3.60s.	1h.06min.35s.
spend	Person	C5	3.68s.	76.42s.
spend	Parent	C5	1.71s.	153.04s
setbestfriend	Person	C7	3.26s.	0.31s.

Fig. 7. Results.

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. Note that for simple examples, the prototype that uses the backward predicate transformer is slightly more efficient than the prototype that uses the forward predicate transformer. On the other

hand, for complex examples, the resolution based theorem prover seems to flounder and the overall backward analysis can take much longer than the forward one. This is rather surprising, as backward predicate transformers have the reputation of being more efficient than their forward counterpart, and resolution has the reputation of being more efficient than the tableaux method.

6 Method correction

The natural counterpart of safety detection is that of method correction. We will describe a technique for doing so, which is based on the backward predicate transformer. The backward predicate transformer is used to produce a condition which, when included into the code of a method, will ensure its safety with respect to a specific constraint. As opposed to the approach we have already presented, the advantages of automatic method correction are unclear, and have to be further assessed.

There are many ways in which one can correct a faulty method. Nevertheless, such a correction should respect the following requirements:

- it should be fully automatic. The user should not have to intervene in any way.
- it should preserve the semantics of the initial method.

To that purpose, the following definition provides a very general method to alter methods in order to make them safe.

Definition 13. *Let C a constraint and m be a method defined as follows:*

$$\begin{array}{l}\texttt{method}\, m \textit{ (parameters)} \texttt{ in class } \kappa \\ \qquad\qquad \textit{body}\end{array}$$

The corrected version of m with respect to C, written m_C, is defined as follows:

$$\begin{array}{l}\texttt{method}\, m_C \textit{ (parameters)} \texttt{ in class } \kappa \\ \qquad\qquad \{\texttt{if } \overleftarrow{m}\ (C) \\ \qquad\qquad\ \texttt{then } \textit{body}\}\end{array}$$

We now have to show have to show that the corrected version of a method is well defined, i.e., that it preserves integrity.

Theorem 14. *Let m be a method and C a constraint. m_C is safe with respect to C.*

This technique of method correction is sound, but many questions put its applicability in doubt. Schematically, methods can be classified in the following way: they are safe or not, and if safe they can be detected as being safe or not. Considering the complexity of the underlying logic, it is clear that safety detection will fail for many safe methods. Unfortunately, this is not a shortcoming that is specific to our approach, but it is a theoretical barrier that no other approach can claim to jump over.

A first problem will occur if we apply method correction to a method which is safe but cannot be proven so. In that case, the correction is unnecessary, and will certainly burden the original code with redundant tests.

A second problem is in assessing the gain that method correction brings in terms of the efficiency of the system. The idea of method correction is just to undertake a preliminary test on the database before undertaking the original update. In any case, this technique prevents the system from having to undertake expensive roll-backs when a constraint is violated. However, this preliminary test can be in itself very expensive, as the predicate transformers generate formulae that are far bigger than their input formulae. So, undertaking systematic method correction might be the best way of ensuring very low system performances.

7 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].

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 database programs 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 precondition before the execution, which avoids subsequent rollbacks.

[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.

8 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 two *abstract predicate transformers*, 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].

We also plan to apply our method to dynamic (temporal) constraints. To this end we shall define temporal-logic based predicate transformers and explore the domain of temporal theorem proving techniques.

Acknowledgements

We are thankful to Giorgio Ghelli and Atsushi Ohori who first recommended us to study abstract interpretation theory. We are very grateful to Giuseppe Castagna for his suggestions and many valuable comments for improving this paper. We also would like to thank Serenella Cerrito for reading an early version of this paper. Patrick Cousot deserves a special acknowldgement there, in assuring us that this work was far from a trivial application of abstract interpretation he gave us a tremendous confidence.

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.

[Apt81] K. R. Apt. Ten years of hoare's logic. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[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.

[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.

[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*, pages 72–88, Cape May, U.S.A, June 1996. Morgan-Kaufmann.

[BS97a] V. Benzaken and X. Schaefer. Forward and backward analysis of object-oriented database programming languages: an application to static integrity management. Technical report, L.R.I, 1997. Available by ftp at `ftp.lri.fr/LRI/articles/benzaken/report.ps.gz`.

[BS97b] V. Benzaken and X. Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In M. Aksit and S. Matsuoka, editors, *European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 60–85. Springer-Verlag, 1997.

[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.

[Deu90] O. Deux. The Story of O_2. *IEEE Transaction on Knowledge and Data Engineering*, 2(1), March 1990.

[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.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12, 1969.

[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.

[McC] William W. McCune. *OTTER 3.0 Reference Manual aud Guide*. Argonne National Laboratory. Peut être obtenu à l'adresse internet suivante: http://www.mcs.anl.gov/home/mccune/ar/otter.

[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.

[Sch86] D. A. Schmidt. *Denotational Semantics*. Allyn and Bacon, Inc., 1986.

[Sch97] X. Schaefer. *Bases de données orientées objet, contraintes d'intégrité et analyse statique*. PhD thesis, Université de Paris 1 - Panthéon - Sorbonne, 1997.

[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