Ensuring efficiently the integrity of persistent object systems via abstract interpretation.

Véronique Benzaken, Xavier Schaefer,*
L.R.I. Bat 490
Université de Paris XI
91405 Orsay Cedex, France
e-mail: Veronique.Benzaken@lri.fr, Xavier.Schaefer@lri.fr

July 18, 1996

Abstract

In this paper, we propose an efficient and reliable method to deal with integrity constraints in a persistent object system. First we provide the application programmer with the ability to *express* integrity constraints but we also give him the possibility to use high level language constructs to help him in *writing* safe transactions.

The goal of our approach is to avoid the (run time) checking of constraints by proving formally that transactions preserve integrity constraints. We mainly use two abstract interpretation techniques to do that. Abstract interpretation is a *semantics-based* tool that yields some reliable information about the possible run-time behaviour of programs, with *fully automatic algorithms*. We present informally the methods that we use: a simple method, based on *path reachability*, and a more powerful and complex method that uses a *predicate transformer*. A predicate transformer is a function that, given a transaction and a formula describing its input data, yields a formula describing its output data.

We finally describe the current prototype that applies those different techniques. It provides in fact the O_2 compiler with an integrity constraint manager.

Keywords: abstract interpretation, reliable persistent stores, database programming languages, integrity constraints, program semantics and static analysis, predicate transformers, safety proofs.

1 Introduction

A great concern for persistent system 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, given by the user. Integrity constraints can be a very critical topic when designing and implementing a persistent object system. On one hand, integrity constraints are very useful, particularly in the field of persistent object systems that offer integrated database facilities (persistent store, database programming language...). On the other hand, a naive management of integrity constraints can burden the system, as constraints have to be checked after each update. We propose in this paper an efficient model for the implementation of integrity constraints managers in persistent object systems. This model consists in providing:

*Also in Université de Paris I - Panthéon - Sorbonne. Xavier Schaefer is currently supported by a PhD Grant from Ministère de L'Enseignement Supérieur et de la Recherche.

- The application programmer with new programming language constructs
- The compiler with the ability of statically determining whether a transaction preserves integrity constraints.

The application programmer must be able to express constraints in a global and declarative way (as stated in [BLR92, BD93, BD95]), but must also be able to write *easily* transactions that respect those constraints. That is the reason for introducing new language constructs. To this end we extend the usual boolean expressions in conditionals (`if`'s,etc.) by allowing logical quantifiers (those generalised conditions can be translated in standard OQL queries). Without that last construct, the task of the application programmer would be difficult, as he would have to write the appropriate tests with a low-level imperative language. This would result in disseminating the semantics of integrity throughout the code. The resulting transaction will be difficult to read and maintain.

The problem of statically determining transaction safety is not new and has been dealt with in different ways. A first approach in [GM79, CB80, Qia90] uses a database extension of Hoare's logic. However, their works are not fully formalised and no steps towards a mechanisation of safety proofs is clear. A second approach in [SS89] uses a complex Boyer-Moore theorem prover and is actually implemented. Their method is quite heavy and complicated, as it is in fact an expert system. Also, it is not known whether it is consistent as it uses axioms that have not been proven so. In fact, since [Qia93], the problem of integrity has been dealt with in the opposite way: instead of analysing complex transactions, [Qia93] studies the generation of safe and correct transactions from some logical specifications. [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 also deal with that problem. Our approach consists in a compiler that checks that transactions preserve integrity constraints. We present several techniques for doing so, that are all based on *abstract interpretation*, which is a method widely used in optimising compilers.

- First we use a simple compilation technique to avoid checking the constraints. That method takes into account the *inertia* of a database, that is, the data that is not altered by a transaction. For instance, a transaction that only changes the age and the spouse of a person cannot violate constraints which are not concerned with the ages or the spouses of persons. That method has already been presented in [BD93, BD95].
- We then define a much more powerful analysis that cannot be applied directly on the original database programming language. That language must first be translated into an intermediate language, that is more simple and elementary. An *abstract interpretation* using a *predicate transformer* is applied to this intermediate language. This approach produces some very precise information about the possible behaviour of transactions and can lead to some very good optimisations.

The integrity constraints manager combines those methods and we are confident that most safe transactions can be detected as such and the checking avoided. Our method has the advantage of being *fully automatic* and implementable (as opposed to [GM79, CB80, Qia90]), simple and *semantically founded* (as opposed to [SS89]).

The paper is structured as follows: we present our framework in Section 2, which is an extension of the O_2 data model. We then describe in Section 3 the simple method to avoid checking the constraints. We present in Section 4 a more powerful method based on predicate transformers. We finally describe the integrity constraints manager that is currently being implemented.

2 The data model: O_2 with integrity constraints.

Our framework consists in a very simple extension of the O_2 data model. The programmer can define, together with the usual classes and transactions, some *semantic integrity constraints* which are well formed formulas on a specific language. Most of our data model is compatible with the O_2 model. The only changes that have been made are those that are necessary, that is:

- The ability to describe integrity constraints together with a schema and transactions.
- The ability to write *easily* transactions that are safe.

We point out that using the O_2 system is not essential to our approach. We give in the following a simple O_2 schema, several transactions, and some integrity constraints on that schema. We will assume that the reader is familiar with the O_2 data model. To make the concepts clearer, we will simplify the schema and reduce the use of concepts that are not essential to our approach.

2.1 Schema definition

We use the standard O_2 data definition language. Let's start with the definition of a simple schema on Figure 1.

```
class Person inherit Object
  public type tuple ( name:  string,
                     spouse:  Person,
                     children:  set (Person),
                     money:  integer,
                     friends:  set (Person)
                    )
end;

name People:  set (Person);
```

Figure 1: Schema

A `Person` has a `name`, a `spouse`, owns some `money`, has several `children` and several `friends`. We also define a persistent name `People`, as a set of `Persons`. We the sake of simplicity , we will not consider methods.

2.2 Integrity constraints

That schema is quite simple but many constraints can already be stated:

1. There must be at least one `Person` in the set `People`.
2. The child of a `Person`, say `x`, cannot be `x`.
3. The spouse of a `Person`, say `x`, cannot be the `child` of `x`.
4. There is a `Person` who is friends with everyone else.
5. If a `Person` `x` is married to another `Person` `y`, then `y` is married to `x`.

All those constraints are given Figure 2.

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

```
C1:  exists x in People;
C2:  forall x in People,  !(x in x->children);
C3:  forall x in People, !(x->spouse in x->children);
C4:  exists x in People, forall y in People, (y in x->friends);
C5:  forall x in People, x->spouse->spouse=x || x->spouse=nil;
```

Figure 2: Constraints

complex formulas can be built with the usual connectives "`&&`" (AND), "`||`" (OR), "`!`" (NOT). 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`[1]. 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. In our case, the domain is the set of object identifiers and values and the relationships are those that are given by the schema: `name`, `spouse`, `money`, `children` and `friends`, which are all binary relationships.

2.3 Extending the transaction language

The language we use to specify transactions is very close to O_2C. In fact, a single construct has been added. Consider the following O_2C construct: [if (b) then i_1 else i_2]. `b` is a boolean expression, and i_1 and i_2 are instructions. In our language, that construct is replaced by [if (f) then i_1 else i_2], where `f` is a *first-order formula*. The only difference is that we allow *quantifiers* like `forall` or `exists` to occur in the test. That extension of the standard O_2C syntax might seem artificial and unnecessary to the reader, but consider constraint `C4` and assume that the application programmer wants to write a safe transaction which , given two `Persons` `p` and `q`, removes `q` from `p->friends`. So as not to violate `C4`, we have to check that there is a `Person` `x` in `People`, such that `(x!=p)` and every other `Person` in `People` is in `x->friends`. With the standard O_2C syntax, that transaction would be written:

```
transaction body end_friendship(p:Person,q:Person) in application family
{ o2 boolean bx=false;
 for (x in People)
  { o2 boolean by=true;
   for (y in People)
    { if ((x==p) || !(y in x->friends))
      by=false;}
   if (by) bx=true;}
 if (bx) p->friends-=q;}
```

The programmer has to introduce artificial flags (`bx` and `by`) and write the right loops in order to perform the appropriate tests. In order to relieve the programmer from the cumbersome task of writing those tests, we allow him to write instead:

[1] a slight difference with usual quantification is that formulas of the kind `(forall p in S)` and `(exists p in S)` are allowed.

```
transaction body end_friendship(p:Person,q:Person) in application family
{ if (exists x forall y (x != p) && (y in x->friends))
   p->friends -= q;
}
```

The appropriate test is stated in a declarative way with a quantified boolean expression. We insist that such a construct is the necessary counterpart of the constraints, inside the transactions. The programmer needs to be able to take some appropriate precautions with respect to integrity constraints in the same global and declarative way. That higher level construct will anyway be translated into standard O_2C instructions in a straightforward manner. We advocate that such a construct increases the application programmer productivity.

We give in Figure 3 a few transactions that might cause some problems with respect to integrity constraints. This set of transactions will be used as a running example throughout the remainder of the paper.

```
transaction body remove_People(p:Person) in application family
{ if ( exists q in People,(q != p))
   People-=p;
}

transaction body add_child(p:Person,q:Person) in application family
{ if ((q != p) && (q != p->spouse))
   p->children += q;
}

transaction body end_friendship(p:Person,q:Person) in application family
{ if (exists x forall y (x != p) && (y in x->friends))
   p->friends -= q;
}

transaction body divorce(p:Person) in application family
{ if (p->spouse!=nil)
   p->spouse->spouse=nil;
p->spouse=nil;
}
```

Figure 3: Transactions

1. `remove_People` removes a `Person p` from the set `People`. So as not to violate constraint `C1`, we have to check that there is another `Person q` in `People` such that `q!=p`.
2. `add_child` adds a child `q` to the set of `children` of a person `p`. So as not to violate constraints 1 and 2, we have to beware that `p`≠`q` and that `p->spouse`≠`q`.
3. `end_friendship` states that `q` is not `p`'s friend anymore. So as not to violate constraint 3, we have to check that somebody else than p is friends with everyone else.
4. `divorce` separates a person `p` form its spouse. To respect constraint 4, we also have to separate `p->spouse` from `p`.

3 A simple analysis: path reachability

We present in this section a first method to check out some transactions. This method is an instance of the so-called *path reachability methods* used in abstract interpretation. It is simple, cheap, and in fact quite efficient. It should be run as a preliminary analysis, before undertaking the complex analysis we present in the next section. The latter analysis is much more expensive and often unnecessary. Therefore, it should only be applied to check out "hard" transactions.

The *path reachability* method is based on the assumption that most transactions only change a specific part of the database. Similarly, constraints are only concerned about a part of the database. When a transaction does not modify that part, then it cannot violate the corresponding constraint. This analysis can be formalised in terms of abstract interpretation. However, such a complex formalisation is unnecessary for such a simple method. This method has already been presented in [BLR92, BD93, BD95] for a specific system: Thémis. We slightly reformulate it here as it will be implemented in our context. We first describe informally our method. We define in the following the paths that are relevant to an integrity constraint `C` and to a transaction `t`, denoted respectively by Δ`(C)` and ∇`(t)`. What we call a path is either a persistent name, like `People`, or an expression of the form `type->attribute`. A persistent name `S` is relevant to `C` if an insert or a delete[2] from `S` might change the truth value of `C`. Similarly, a path `t->a` is relevant to `C` if a modification of the attribute `a` of an object of type `t` might change the truth value of `C`. Also, a persistent name `S` is relevant to a transaction `S` if an object is inserted into or deleted from `S`. Similarly, a path `t->a` is relevant to a transaction `S` if the attribute `a` of an object of type `t` is modified.

The paths that are relevant to an integrity constraint `C` can be computed by induction on the structure of constraints. Intuitively, if a term `o->a` appears in `C`, then the path `type(o)->a` is relevant to `C`. Similarly, if a persistent name `S` appears in `C`, then the path `S` is relevant to `C`.

Definition 1 Let `term` be a term. We define δ`(term)` as follows.

`term`	δ`(term)`
`t->a`	δ`(t)` $\cup$ {`type(t)->a`}
`x`, `x` is a variable	$\emptyset$
`S`, `S` is a persistent name	{ `S` }

Definition 2 Let `C` be a formula. We define the paths that are relevant to `C`, Δ`(C)`, as follows.

`C`	Δ`(C)`
`!C`	Δ`(C)`
C_1 `&&` C_2	$\Delta(C_1) \cup \Delta\ (C_2)$
C_1 `||` C_2	$\Delta\ (C_1) \cup \Delta\ (C_2)$
`forall x in S,` C'	$\delta(S) \cup \Delta\ (C')$
`exists x in S,` C'	$\delta(S) \cup \Delta\ (C')$
$(t_1 == t_2)$	$\delta(t_1) \cup \delta(t_2)$
$(t_1$ `in` $t_2)$	$\delta(t_1) \cup \delta(t_2)$

Definition 3 Let `t` be a transaction. We define the paths that are relevant to `t`, ∇`(t)`, as follows.

[2] what we call persistent names are in fact repositories, that is, persistent names structured as sets.

t	∇(t)
`t->a=expr`	{ `type(t)->a` }
`S+=expr`, `S` is a persistent name	{ `S` }
`S-=expr`, `S` is a persistent name	{ `S` }
$\texttt{t}_1$; $\texttt{t}_2$	$\nabla(\texttt{t}_1) \cup \nabla(\texttt{t}_2)$
`if (b)` $\texttt{t}_1$ `else` $\texttt{t}_2$	$\nabla(\texttt{t}_1) \cup \nabla(\texttt{t}_2)$
`forall (x in s)` $\texttt{t}_1$	$\nabla(\texttt{t}_1)$

Theorem 1 Let `t` a transaction and `C` a constraint. `t` is safe with respect to `C` if $\Delta(\texttt{C})\cap\nabla(\texttt{t})=\emptyset$.
Proof sketch: if $\Delta(\texttt{C})\cap\nabla(\texttt{t})=\emptyset$, the transaction updates a part of the database that is of no concern to `C`. Its truth value is therefore not modified by `t`. As it is always assumed to be true before the execution of `t`, it remains so.

Example 1 We give $\Delta(\texttt{C})$ for each constraint `C`. `C`.

constraint	Δ(constraint)
C1	{People}
C2	{People, Person->children}
C3	{People, Person->children, Person->spouse}
C4	{People, Person->friends}
C5	{People, Person->spouse}

We also give $\nabla(\texttt{t})$ for each transaction `t`.

transaction	Δ(transaction)
remove_People	{People}
add_child	{Person->child}
end_friendship	{Person->friend}
divorce	{Person->spouse}

From those results, with theorem 1, we can conclude which transaction may violate which constraint.

transaction	constraints that might be violated
remove_People	C1, C2, C3, C4, C5
add_child	C2, C3
end_friendship	C4
divorce	C3, C5

The evaluation of the benefits of that method is quite difficult. It is more expensive to check `C2` than to check `C1`. Also, if `remove_people` is the only transaction that is run, no improvement has occured. As an indicator, if we consider that the cost of checking each constraint is equal, and that transactions are all run the same number of times, then we have in fact divided by two the checking overhead. That shows that some very satisfactory results can be obtained with simple methods. However, there is still some room for a lot of improvement. Transaction `remove_People` can obviously not violate `C2`, `C3` and `C5`. That is why we need a more powerful method, that takes into account more precisely what is done by a transaction.

4 A complex analysis: predicate transformers

We present in this section a powerful method to prove the safety of transactions with respect to integrity constraints. This method undertakes a painstaking analysis of the transaction that takes many details into account. It uses the concept of *predicate transformer* and finds its place in the general framework of abstract interpretation. A *predicate transformer* is simply a function that, given a program and a formula describing its input data, yields a formula describing its output data. The application of predicate transformers to our problem is straightforward. The input data of a transaction `t` is described by an integrity constraint `C`. We apply the predicate transformer to `t` and `C` and get another formula `C+` describing the output data of `t`. the main result of this section is that `t` cannot violate `C` if `C+`$\Rightarrow$`C` (`C` is a consequence of `C+`).

The idea of using Dijkstra or Hoare's methods to deal with integrity is not new at all. The idea was first mentioned in [GM79], further developed in [CB80] and in [Qia90]. However, the results presented in those works do not suggest an effective method of proving transaction safety. Our work is very close to those mentioned, but uses tools that allows an easy implementation. If [GM79, CB80, Qia90] ensure that a safety proof can be obtained for any safe transaction (due to completeness), *no effective way of obtaining it is given.* The reason for that is that they use non-deterministic deductive means to yield a proof, and it is not clear which choices have to be made to get a proof. On the contrary, abstract interpretation is an *algorithm*, where no arbitrary choice has to be made. Not only do we have to describe *what to get*, but also *how to get it.* The analogy with first-order theorem proving is enlightening. There are many valid proof systems for first-order logic: Natural Deduction, Hilbert systems, Tableaux, Resolution... But only Tableaux and Resolution are suited to automation. Going from Hoare's logic to abstract interpretation is not a mere matter of cosmetics, but a question of applicability. It requires a different outlook on the problem straight from the start, by privileging effective algorithms rather than logical relationships.

4.1 The intermediate language

Our method cannot be applied directly to the O_2C language. We have to translate the O_2C code into an intermediate language that is more simple. We present in this section the intermediate language we use to undertake the analysis. We also explain why that intermediate language is necessary.

O_2 is an object oriented database system. However, object orientation is not relevant to integrity constraints. The only information that is really relevant is the way the elements of the *domain* (object identifiers, values, persistent names) are related to each other. Attributes are considered as binary relationships and we have to translate the transactions into a logical form that is much more easy to handle. Constraints also have to be translated into classical first-order logic. Such a decomposition is necessary to be able to define the predicate transformer simply and clearly. Let us see that on an example. Consider the following precondition.

```
(p->spouse->spouse==q)
```

We might want to know the effect the execution of an instruction can have on that precondition. Consider the following instruction:

```
r->spouse=s;
```

We will have to consider many cases:

- `(r==p)` or `(r!=p)`.
- `(r==p->spouse)` or `(r!=p->spouse)`.

Which is not very easy with a precondition like that. It can be done easily if we decompose `(p->spouse->spouse==q)` into `(p->spouse==u && u->spouse==q)`. We translate complex terms into a conjunction of simpler terms. The effect of the instruction is then straightforward. Its effect on `(p->spouse==u)` is

- `(p->spouse==s)` if `(r==p)`.
- `(p->spouse==q && r->spouse==s)` if `(r!=p)`.

The overall effect is `((r==p && p->spouse==s)||(r!=p && p->spouse==u && r->spouse==s))`. The same can be done to get the effect on `(u->spouse==q)`:
`((r==u && u->spouse==s)||(r!=u && u->spouse==q && r->spouse==s))`. The final result is the conjunction of both effects, which is quite a big formula. However, that formula can be obtained with a succession of simple steps. We give in the following the way to translate constraints and programs into their logical form. Let us begin with the constraints

- each formula [`forall x in S, F`] is translated into [`forall x, (x in S) ==> F`].
- each formula [`exists x in S, F`] is translated into [`exists x, (x in S) && F`].
- each term of the form [`t->a`] is translated into [`a(t)`].
- each formula [`F(a(x))`] (i.e. that depends of the term `a(x)` is translated into :[`forall y, a(x,y) ==>F(y)`]

All those transformations are fully justified. Indeed, in logic $[\forall x \in S, P(x)]$ is just an abbreviation for $[\forall x, (x \in S) \Rightarrow P(x)]$, $[\exists x \in S, P(x)]$ is just an abbreviation for $[\exists x, (x \in S) \wedge P(x)]$. Attributes are defined as functions that are applied to objects, therefore `o->a` can be written `a(o)`. Finally, functions are just a special kind of relations, thus any function $y =$`a`(x) of type $D \rightarrow D$ can be rewritten `a`(x, y) of type $D \times D$. Those changes are therefore not essential, but help to decompose the structure of constraints. For `C1` the detailed translation is:

```
forall p in People !(p in p->children)
forall p (p in People) ==> !(p in p->children)
forall p (p in People) ==> !(p in children(p))
forall p forall x (children(p,x) && (p in People)) ==> !(p in x)
```

We then show how to translate transactions. We will restrict ourselves to the most common O_2C constructs. We first have to introduce the target language of the translation. It is very close to first order logic and is therefore relational. However, some of its constructs are tailored to simulate more complex O_2C instructions. The syntax of that intermediate language, where `t` is a tuple and `r` is a relation, is given Figure 4.

```
instruction ::= insert t into r
              | delete t from r
              | instruction ; instruction
              | { instruction }
              | if (b) instruction
              | forall o where b do instruction
              | forone o where b do instruction
```

Figure 4: Target Language.

In persistent object systems, we mainly deal with binary relations, some of which are in fact functions. We insist that all the attributes defined in our schema will be considered as binary relations:

`name` of type `Person`$\rightarrow$`string`, `spouse` of type `Person`$\rightarrow$`Person`, `money` of type `Person`$\rightarrow$`integer`, `children` and `friends` of type `Person`$\times$`Person`. Nearly all those constructs are very classical. The two first instructions are the classical tuple insertion and deletion. Only the last one might not be clear. The instruction [`forone o where b do instruction`] executes `instruction` for the only element `o` that satisfies `b`. If there is more than one element, then an error occurs (which never happens when that language is used as a target language). We now define Figure 5 how to translate each instruction. We will denote by `Δ(i)` the translation of instruction `(i)`.

instruction	Δ(instruction)
`i1 ; i2`	`Δ(i1) ; Δ(i2)`
`{ i }`	`{Δ(i)}`
`if (b) i`	`if (b) Δ(i)`
`for (o in x) { i }`	`forall o where (o in x) { Δ(i) }`

`o->a=expr  (*o is a tuple)`	`forone x where a(o,x) do`
	`{ delete (o,x) from a;`
	`insert (o,expr) into a}`

`o->a+=p` (`o->a` is a set)	`insert (o,p) into a`
`o->a-=p` (`o->a` is a set)	`delete (o,p) from a`
`S+=p` (S is a repository)	`insert p into S`
`S-=p` (S is a repository)	`delete p from S`

Figure 5: Translation of transactions.

A repository is just a persistent name structured as a set. For instance, `People` is a repository. The question of whether that translation preserves meaning is not relevant, as we have not previously defined the formal semantics of the O_2C language. In fact, it is what we are doing right now as that translation defines the logical semantics of O_2C instructions ! However, we can easily see that that semantics follows the intuitive meaning of O_2C instructions. Something that does not appear, for sake of brevity, is that the boolean expression `(b)` in the `if` must also be decomposed, in the same way as the constraints. We have given the translation of a few basic instructions that can be generalised to more complex ones. For instance, consider the instruction [`o->a->b->c=e`] if [`o->a->b`] is a tuple object, then we translate that by:

```
forone x where a(o,x)
 forone y where b(x,y)
  forone z where c(y,z)
   do
     {delete (y,z) from c;
      insert (y,e) into c;}
```

We insist that such a translation is necessary as it decomposes the changes that are performed by a transaction into small atomic instructions that only handle binary links, one at a time. The intermediate language can be seen as a database assembly language for the more complex O_2C language.

4.2 Defining a predicate transformer

Once the translation of (extended) O_2C transactions is done, we can define a predicate transformer that yields a post-condition according to a pre-condition. The predicate transformer is a function

∇ that has for input an instruction `i` and a formula `f`. The predicate transformer is defined recursively. It works in two steps:

- Formula parse. That parse defines a post-condition for a pre-condition in terms of the subformulas of that pre-condition.
- Instruction parse. That parse defines post-conditions for each instructions where the input formula is atomic (or a literal).

However, the order in which the parses are undertaken is only essential in the case of atomic instructions (`insert` and `delete`). The only assumption that is made is that negations are moved inside the formulas so that only atomic formulas are negated[3]. The formula parse is defined Figure 6

<table>
<tr><th>formula</th><th>∇(instruction,formula)</th></tr>
<tr><td>f && g</td><td>∇(instruction,f) && ∇(instruction,g)</td></tr>
<tr><td>f || g</td><td>∇(instruction,f) || ∇(instruction,g)</td></tr>
<tr><td>forall x, f</td><td>forall x, ∇(instruction,f)</td></tr>
<tr><td>exists x, f</td><td>exists x, ∇(instruction,f)</td></tr>
<tr><td>(f)</td><td>(∇(instruction,f))</td></tr>
</table>

Figure 6: Formula parse.

∇ is therefore a `&&`-morphism and a `||`-morphism. The two following formulas are a consequence of that, as `forall` is a generalised conjunction, and `exists` is a generalised disjunction. The transaction parse is defined only for literals, that is, atomic formulas or their negation. See Figure 7.

<table>
<tr><th>instruction</th><th>literal</th><th>∇(instruction,literal)</th></tr>
<tr><td>insert t into r</td><td>l</td><td>if l=!r(u)
then (t==u && r(u))||(t!=u && !r(u) && r(t))
else l && r(t)</td></tr>
<tr><td>delete t from r</td><td>l</td><td>if l=r(u)
then (t==u && !r(u))||(t!=u && r(u) && !r(t))
else l && !r(t)</td></tr>
<tr><td>i_1 ; i_2</td><td>l</td><td>∇(i_2,∇(i_1,l))</td></tr>
<tr><td>{ i }</td><td>l</td><td>∇(i,l)</td></tr>
<tr><td>if (b) i</td><td>l</td><td>∇(i, l && b) || (l && !b)</td></tr>
<tr><td>forone x
where a(o,x)
do i</td><td>l</td><td>exists x ∇(i, a(o,x) && l)</td></tr>
</table>

Figure 7: Transaction parse.

We have now fully defined a predicate transformer for our purposes, except for instruction `forall`. In fact, the formal semantics of the database loop is not well defined, because the order in which the elements are taken is not known, and might have some profound consequences at runtime. We need to have a clear denotational semantics to be able to define an abstract interpretation.

[3] That can be done easily by using the following equivalences: `(!(f && g) = !f || !g)`, `(!(f || g) = !f && !g)`, `(!forall x, f = exists x, !f)`, `(!exists x, f = forall x !f)`.

However, no satisfactory semantics has been defined up to now, so we have to manage without it. The reason for that is that two valid executions can lead to thoroughly different results. Usually, some very restrictive assumptions are made upon the use of loops. Roughly, they amount to use database loops only for batch processing, that is, a program that is run recurringly on a number of items where each loop is independent from the others. Instead of taking those restrictions, that have been defined in [Qia90], into account, which would be cumbersome and would not lead to very satisfactory results, we propose a more simple approach. Each transaction is sliced into parts that do not contain any loops or recursive calls. Each part must be proven safe to prove the safety of the transaction. That method is justified by the fact that any composition of safe instructions is itself a safe instruction. It is a pragmatic approach, but considering the way transactions are programmed, it produces in fact some very good results. Indeed, transactions are very often composed of small and independent updates that can be analysed separately.

Not only do we have to define a predicate transformer but we also have to prove that it is *correct*. The correctness requirement is very important and ensures that we can rely on the information that has been generated by the predicate transformer. In our case:

Theorem 2 Let `B` a database and $\mathtt{B}_+$ the same database after an execution of transaction `t`. Let `f` be a formula. If `f` is true in `B`, then ∇`(t,f)` is true in $\mathtt{B}_+$.

The proof of that theorem is done by induction both on formulas and on instructions and can be found in [BS95]. The direct consequence of that theorem is the possibility of using predicate transformers to prove transaction correctness.

Theorem 3 Let `t` a transaction and `c` an integrity constraint. Then `t` is safe with respect to `c` if ∇`(t,c)` $\Rightarrow$ `c`.

Proof : Let `B` be a database and $\mathtt{B}_+$ the same database after an execution of `t`. We can assume that `c` is true before the execution. Because of theorem 2, if `B` satisfies `c`, then $\mathtt{B}_+$ satisfies ∇`(t,c)`. Therefore, $\mathtt{B}_+$ satisfies ∇`(t,c)`. If ∇`(t,c)`$\Rightarrow$`c` then Therefore, $\mathtt{B}_+$ satisfies `c` and `t` cannot violate `c`.

We have partly solved the problem, as a proof remains to be done: that the constraint `c` is the logical consequence of ∇`(t,c)`. The problem, in general, is undecidable. However, many proof techniques, like resolution or tableaux, exist and can be applied very efficiently. A very nice property of those proof techniques is that they are *complete*. If a proof exists, it will be found sooner or later. So, if a transaction `t` is safe with respect to `c`, we will be able to say so in a finite time. The problem occurs when the transaction is not safe, because a proof will be sought indefinitely. We therefore have to put a limit on the search space. We will have to conclude, for some safe transactions, that they might not be safe. We insist that, even if we had a decision procedure (which exists in many cases, see [Gri90]), it would certainly be too expensive to be undertaken efficiently (even the decision procedure for propositional logic is NP-complete and we have to use non-deterministic deductive means to improve the process). For our prototype, we have chosen the resolution method, which is the method that is most widely used for automatic theorem provers.

4.3 A simple example

We show how the predicate transformer analysis works for one of the examples we have given. We point out that the analysis can involve much information, even for simple cases. That is why we will just outline what happens for the first example. However, that information can be dealt with easily and efficiently by a computer. We want to prove that transaction `remove_People` cannot violate constraint `C1` (which is indeed the case).

```
transaction body remove_People(p:Person) in application family
{ if (exists q in People, (q != p))
   People -= p;
}
C1:  exists p in People;
```

We first make the appropriate translation:

```
transaction body remove_People(p:Person) in application family
{ if (exists q, People(q) && (q != p))
   delete p from People;
}
C1:  exists x, People(x)
```

If `C1` is true before the execution of `remove_People`, which is always the case, then `∇(remove_People,C1)` is true after its execution.
We give the first steps to compute `∇(remove_People,C1)`:

```
∇(remove_People,C1)
  =∇(remove_People,exists x  People(x))
  =exists x ∇(remove_People,People(x))
  =exists x ∇( { if...}, People(x))
  =exists x ∇( if..., People(x))
  =exists x ∇( delete...,People(x) && (exists q People(q) && (q != p)))
            || People(x) && !(exists q People(q) && (q != p)))
  =exists x exists q ∇( delete...,People(x) && People(q) && (q != p)))
            || People(x) && !(exists q People(q) && (q != p)))
  =exists x exists q ∇( delete...,People(x))
                    && ∇( delete...,People(q))
                    && ∇( delete...,(p!=q))
            || People(p) && !(exists q People(q) && (q != p)))
  =exists x exists q (p=x && !People(x))||(p!=x && People(x) && !People(p))
                    && (p=q && !People(q))||(p!=q && !People(p) && People(q))
                    && (p!=q) && !People(p))
            || People(x) && !(exists q People(q) && (q != p)))
```

Let us call that big formula `F`. What remains to be done is the proof that: (`F ==> C1`). We then use a classical first-order logic proof technique: resolution. This amounts to showing that (`!(F ==>C1)`) is *unsatisfiable*. For the sake of brevity, we do not give the proof produced by the resolution method. We have to start by putting `F` into conjunctive normal form. Which simplifies the structure of `F` but also increases its size, as we end up with a formula that has around 150 terms! We insist that during this part of the proof, we strictly apply standard proof techniques as can be found in [Fit90]. As mentioned earlier, such a proof, if it exists, will always be obtained by the resolution method in a finite time. That is a consequence of *resolution completeness*. The real problem occurs when the transaction violates the constraint, as the resolution algorithm will go on looking for a proof whereas none exists! As said earlier, we will have to put bounds on the search space for a proof. If no proof exists within that search space, we will have to conclude that no proof exists. That conclusion might of course be wrong, but it will just result in saying we must check the constraint whereas the checking is not necessary. So, to summarise, our method:

- provides a safety proof for safe transactions.
- does not terminate for unsafe transactions.

However, when implementing a method, we have to ensure that it always terminates. The integrity manager will in fact:

- provide a safety proof for *some* safe transactions.
- will stop without having reached a conclusion for *all* unsafe and *some* safe transactions.

The power of the integrity manager depends, not surprisingly, of the time and space we are willing to devote to each proof. We are confident that most safe transactions could be detected easily. Serious tests will be done when the prototype is completed.

5 Implementation

We present in this section the prototype that is currently being implemented. It is a concrete application of the theory we have presented to persistent object systems. Instead of creating from nothing a persistent object system with integrity constraints, we have decided to add an integrity management facility to an existing system: O_2. The integrity manager works with:

- an O_2 database schema together with a set of integrity constraints.
- a set of transactions written in the extended O_2C language.

We have already seen that we have to extend the standard O_2 syntax with a new construct: we allow, instead of boolean expressions, *quantified* boolean expressions. That provides us with a simple and efficient way of making the run-time tests that are necessary when writing *safe* transactions. The transactions in the application are written in extended O_2C. We show in Figure 8 how quantified boolean expressions are translated into O_2C. This translation deserves an explanation. The truth value of the expression that is translated is stored in the `boolean bx` that is declared previously. If `M(x)` has quantifiers, then the translation must be undertaken recursively.

expression	translation

`exists x in S, M(x)`	`o2 boolean bx=false;`
	`for (x in S)`
	`{ if (M(x)) bx=true;}`

`forall x in S, M(x)`	`o2 boolean bx=true;`
	`for (x in S)`
	`{ if (!M(x)) bx=false;}`

Figure 8: Translation of quantified boolean expressions.

As an example, let us consider the instruction already seen in transaction `end_friendship`:

```
if (exists x forall y (x != p) && (y in x->friends))
  p->friends -= q;
```

We first translate the first quantification:

```
{ o2 boolean bx=false;
 for (x in People)
  {
```

```
    if (forall y (x!=p) && (y in x->friends))
      bx=true;}
 if (bx) p->friends-=q;}
```

A quantification remains to be translated:

```
{ o2 boolean bx=false;
 for (x in People)
  { o2 boolean by=true;
   for (y in People)
    { if ((x==p) || !(y in x->friends))
      by=false;}
   if (by) bx=true;}
 if (bx) p->friends-=q;}
```

The integrity manager is in fact a pre-processor that takes the different elements of an application, checks that the transactions are safe and generates the corresponding O_2C files. For the sake of simplicity, the prototype does not consider methods. However, that restriction is not essential to our approach. The main objective of the prototype is to show that our method works well for simple, straight-line transaction. We will then generalise it to any transactions involving loops, recursive calls and message passing. The generalisation, as mentioned earlier, is a simple one. Each transaction is decomposed into simpler updates, each of which is proven safe. We show in Figure 9 the general structure of the integrity constraints manager. We explain in the following the role of each part of the database manager.

- First, the simple *path reachability* technique is used to check out in a simple way transactions that do not violate any constraints.
- After that, for transactions that have not been proven safe, the more sophisticated method of *predicate transformers* is applied in different steps:
  1. The transactions and constraints are decomposed.
  2. For each transaction and each constraint, the predicate transformer is applied. Which transaction has to be checked for which constraint is determined by the path reachability analysis.
  3. An automatic first-order theorem prover based on resolution 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_2C transactions. Tests with quantified boolean expressions are translated and checking algorithms are automatically generated if the safety proof for a transaction has failed. Those can be found in [BD93, BD95].

6 Related works

The study of integrity constraints has been a topic of large interest [Sto75, Nic79, GM79, CB80, LT85, HI85, BM86, KSS87, HCN84, WSK83, BDM88, SS89]. Several works concern the optimisation of checking: [Nic79, HI85] for relational databases, [BM86] for deductive databases, and [BD95] for object-oriented databases. 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

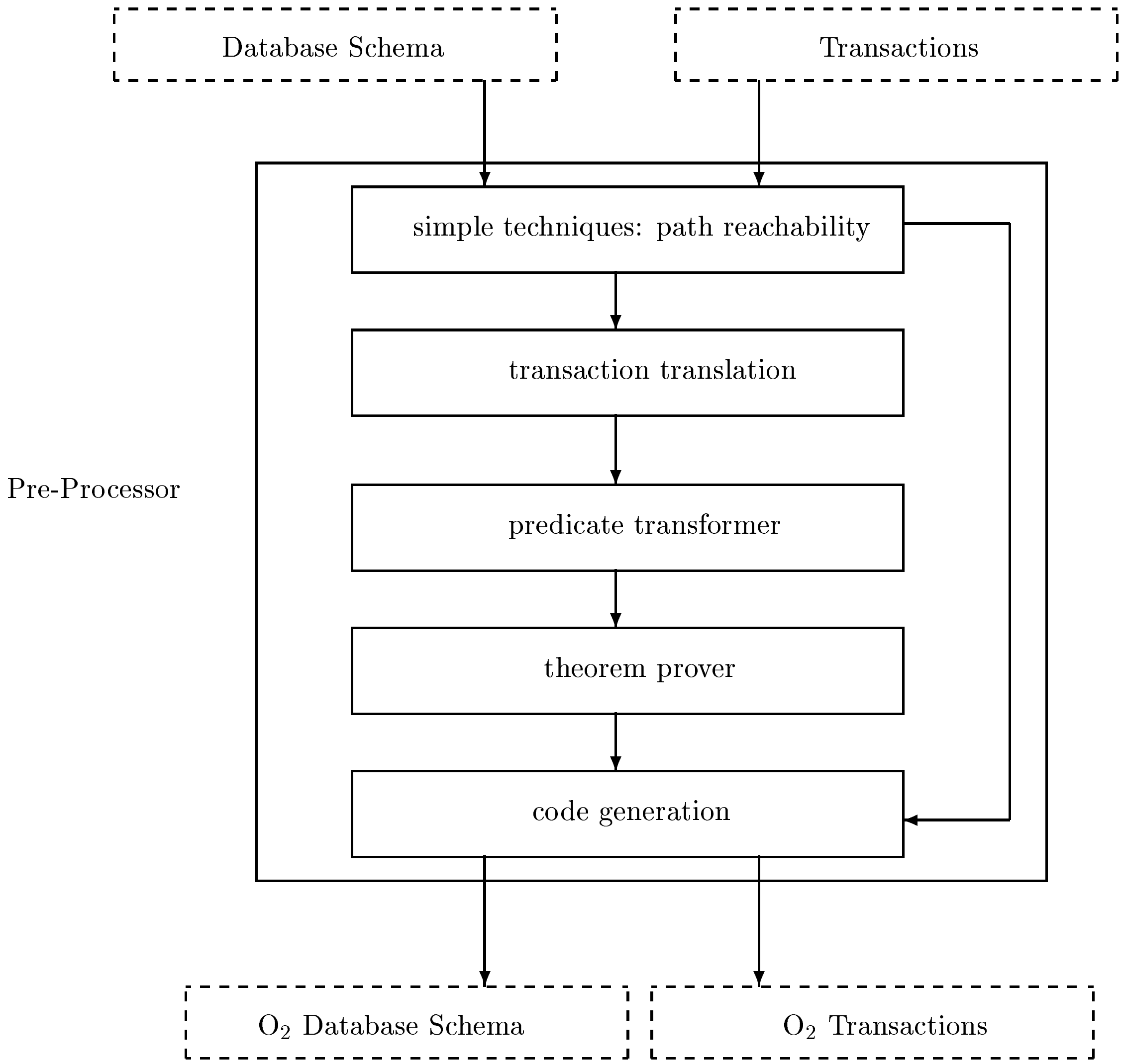

Figure 9: structure of the integrity constraint manager

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], with a special interest in dynamic logic and aggregate operations. However, the problems about the formalisation and automatisation 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. As mentioned earlier, [Qia93] assert that conventional methods are not suited to automatic transaction safety verification and takes the opposite direction of *transaction synthesis*. [SS89] is, to our knowledge, the most in-depth application of the axiomatic method to our problem. It uses a very complex theorem prover that uses Boyer-Moore's computational logic, based on higher-order recursive functions. The complexity of their method makes the task heavy and difficult to follow. Their theorem prover 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 method seems difficult to apply to a real system, as it requires very heavy proofs of theorems with hundreds of terms for only small transactions.

Our work differs from the other works on the subject of consistency proofs for transactions, in that we use the technique of *abstract interpretation* rather than *program verification*, which is used in [GM79, CB80, Qia90, SS89]. The goals of abstract interpretation are similar to those of program verification. The difference between the two approaches is that abstract interpretation is a *semantics-based* method, that yields an *approximate* description of a program's run-time behaviour, which is obtainable with *fully automatic algorithms*, whereas program verification uses *deductive* or *axiomatic* methods. By using abstract interpretation, we solve the problems that arise from [GM79, CB80, Qia90]. Having a semantic approach, rather than deductive, allows us to get an abstract description of program semantics with *algorithms*. No room is left for human intervention. As [GM79, CB80], our approach is based on first-order predicate logic, rather than higher order recursive functions, as in [SS89]. It is therefore difficult to compare the power of the two approaches. However, we point out that, our method being much *lighter*, should be more efficient and easy to handle. Moreover, whereas [SS89] seems to have reached the limits of the axiomatic semantics, conventional abstract interpretation techniques yield very good results. The reader interested in abstract interpretation can find a complete and recent overview of the domain in [JN95], and can of course consult the seminal work of [CC77]. Predicate transformers are exposed in [Dij76, DS90].

7 Conclusion

We have presented a general method to reduce statically the number of integrity constraints to check after the execution of a transaction. We have shown that, by using abstract interpretation techniques, we can obtain a simple safety proof for a transaction, with fully automatic means. First, a simple *path reachability* method is run. Then a more powerful method is run for the cases that remain to be proven safe. Assuming that the integrity constraint is true before the execution of the transaction, we show how to generate a first-order sentence that is true after the update. From that sentence, we prove that the constraint is true after the update. The use of abstract interpretation greatly reduces the use of deductive methods, which are non-deterministic and therefore inefficient. The method works well for complex straight-line updates, and can be extended to other kinds of updates. However, some problems remain:

- The use of first-order logic to express integrity constraints causes many problems: logical implication and equivalence are undecidable properties.
- The semantics of database loops is not formally defined, and therefore cannot be given a good abstract interpretation.

It seems that abstract interpretation reaches its limits when applied to integrity constraints management, because of the amount of information we have to collect to prove that a transaction is safe. However, we are confident that our approach will be used efficiently and detect safe transactions in most cases. The integrity constraint manager that is currently being implemented should give some further information about the applicability of our method.

Acknowledgements

We would like to thank Giorgio Ghelli and Atsushi Ohori who first recommended us study abstract interpretation techniques. We also would like to thank Serenella Cerrito and Giuseppe Castagna for reading an early version of this paper.

References

[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.

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

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

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

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

[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.

[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.

[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.

[Gri90] S. Grigorieff. Décidabilité et complexité des théories logiques. In B. Courcelle, editor, *Logique et Informatique*, chapter 1, pages 7–99. INRIA-collection didactique, 1990.

[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.

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

[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.

[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.