# Data Sharing Analysis for a Database Programming Language via Abstract Interpretation

Giuseppe Amato Fosca Giannotti Gianni Mainetto [*]

CNUCE, Institute of CNR,
Via S.Maria 36, I56126 Pisa, Italy

## Abstract

This paper presents an experiment in using a formal technique for static program analysis, based on abstract interpretation, in the context of persistent programming languages. The aim of the analysis is to detect opportunities for safe parallelism for transaction operation scheduling. Transaction operations can be safely interleaved when there is no overlapping among their readsets and writesets. A non-standard interpreter performs the analysis. This interpreter, given the text of a transaction and a representation of the data stored in the database, *automatically* derives in a finite time an approximation of the readset and writeset of the analysed transaction. Information obtained from the non-standard interpretation is provided to the scheduler before beginning the execution of transaction operations. In this way, we obtain a scheduler that realises a conservative two-phase locking protocol for persistent programming language transactions. We apply the analysis to a language that is a significant subset of Galileo. The goal of the analysis is to detect the accesses solely to shareable and modifiable persistent data.

## 1 Introduction

The evolution of Data Base Programming Languages (DBPLs), which provide a unifying formalism for data definition and manipulation, opens up opportunities for exchanging ideas, problems and technical solutions between the two areas of databases and programming languages. Persistent Programming Languages are the subclass of DBPLs in which the data model of persistent values is completely integrated into a programming language. In fact, the guiding principle of Persistent Programming Languages is the orthogonal persistence: it establishes that persistence is a property of arbitrary values and all values have the same rights to persistence [6]. Specifically, if a Persistent Programming Language is designed on the basis of formal semantics, formal methods for designing, transforming and optimizing database programs can be exploited. Indeed, it is a challenging research direction to investigate whether, which and how such formal methods can be successfully put to work in the database context.

In the programming language area, a consolidated class of formal techniques for static program analysis is known as ***abstract interpretation*** (AI) [1,8]. It is aimed at gathering information on the dynamic semantics of a program or specification to be used by compilers, partial evaluators and debuggers, or merely as documentation [9]. For example, logic programming has successfully used AI for detecting properties such as groundness and goal independence to provide efficient specialised parallel execution of logic programs [10]. Functional programming uses AI to detect strictness and reference independence for efficient graph reduction and optimised garbage collection [1].

We propose using AI for a novel application, i.e. to develop a conservative two-phase locking (2PL) protocol in a persistent system. An *aggressive* scheduler, that realises for example a strict 2PL protocol, analyses transaction operations one at a time, and immediately decides whether to execute, delay or reject the operation [7]. It has no knowledge of the future operations of a transaction. For example, it can decide to execute a write operation that will subsequently risk the transaction being aborted because of a deadlock. A *conservative* scheduler should know *a priori* the shared data which concurrently executing transactions are going to access, so that it can prevent deadlock and abort-

[*]email:{fosca,gmsys}@cnuce.cnr.it phone:+39-50-593339 fax:+39-50-904052

**Proceedings of the 19th VLDB Conference Dublin, Ireland, 1993**

ing due to transaction conflicts. Almost all database systems traditionally have an aggressive and a conservative version of schedulers, because there is a performance tradeoff between the two types of schedulers and each version performs especially well for certain types of applications.

The scheduler can know in advance the set of data that a transaction is going to access, called *readset* and *writeset*, if the transaction programmer predeclares it. In Persistent Programming Languages, inferring readset and writeset of a transaction is difficult because persistent data have a complex structure and both persistent and temporary functions may be involved in a transaction execution. Thus we propose a tool, based on Abstract Interpretation, that automatically performs a static analysis of the program that defines a transaction, inferring an approximation of its readset and writeset.

We analyse transactions written in PPL, a language which is a significant subset of Galileo [3]. PPL is a higher order functional language, strongly and statically typed, with single inheritance, which allows an object-oriented programming style. PPL transactions can access the most important data structures provided by Galileo: basic type values, references, records, sequences, functions, objects, and classes.

The goal of the analysis is to detect the accesses to *sharable* and *modifiable* data structures such as references, classes and objects. In fact, the concurrent execution of transactions can only create inconsistency in the case of updates. Our idea has been inspired by Hudak's proposal [15] that uses reference counting analysis to reduce the run-time memory management overhead in a first order functional language. Similar problems of run-time memory management for a higher-order functional programming language have also been investigated in [11].

The first step of our experiment consists in defining a non-standard semantics which extends the standard one by determining the number of accesses to modifiable data. Informally, every time the real execution accesses a reference or a class or an object, a corresponding access counter is incremented in either read or write mode according to the executed operation.

The second step consists in defining an abstraction of the non-standard semantics that guarantees the termination of the analysis. The termination relies on the fact that the abstract execution operates on finite abstract domains of values. The main choice is to use a finite set of "typed" locations as an abstract domain where abstract values of the corresponding type are stored. A special treatment has been proposed for abstract dynamic data structures, i.e. the data structures whose cardinality cannot be determined at compile time: sequences and classes.

The rest of the paper is organized as follows: Section 2 presents the language, the architecture of the system and the concurrency control protocol; Sections 3 and 4 introduce the formal framework of non-standard access counting semantics and its abstract version; in Section 5 two examples are illustrated; finally Section 6 contains an assessment of the proposed technique and directions for future developments.

## 2 The PPL system

In this section we will sketch the language, the architecture and the concurrency control protocol we are using for our experiment. Restrictions and simplifications to the Galileo system have been assumed in order to point out the crucial issues of the proposed analysis.

The language, which we will refer to as PPL (Persistent Programming Language), is a subset of Galileo tailored for keeping the most important database constructs provided by the original language. More specifically, PPL deals with classes and objects. It is a higher order functional language strongly and statically typed which allows side effects to values of a limited number of types: references, classes and objects. A class is a modifiable sequence of objects and an object is an instance of a record type with a ***unique object identifier***. Classes can be organised in a taxonomic hierarchy through single inheritance. Objects can have as components both functions which mime methods of classical object oriented formalisms, and other objects (instances of other classes).

Since the analysis for access counting takes place after the type checking of the transaction program, the language deals with well-typed expressions. Like in Galileo, a transaction in PPL is represented by a *top-level* expression and starts with the beginning of the expression evaluation. An expression can be a constant, an identifier, a user defined function application or the application of a language primitive. The syntax of an expression of PPL is defined as follows[1] :

$$e ::= c \mid x \mid e(e_1, \ldots, e_n) \mid p(e_1, \ldots, e_n)$$

[1]In the remainder of the paper we use the following conventions:

| | |
|---|---|
| $c, p \in Con$ | (constants and primitives) |
| $x, lb \in Ide, Lab$ | (identifiers and field names) |
| $cl, f, e, r, o, l \in Exp$ | (expressions) |

Primitives are defined for each basic and constructed type, and allow specific operations on values of that type, for instance *assignment* on references, *field selection* on records and so on. The other primitives are the language constructs for flow control and declarations such as *if – then – else*, *sequencing*, *use*, etc..

The following are some primitives which play a significant role in the counting of access to modifiable and sharable data structures.

- References:

  **var**($e$) : creates and returns a reference value to the value $e$.

  **at**($l$) : returns the value referred by $l$.

  $\leftarrow$($l, e$) : assigns the value $e$ to $l$.

- Classes and Objects:

  **class**() : creates an empty class.

  **insert**($x, r$) : creates an object, inserts it in the class identified by $x$ and returns the object identity. The object is created from the evaluation of record $r$.

  **specialise**($o, r, x$) : specialises the object $o$ by adding the fields of record $r$ and inserts it into the subclass identified by $x$.

Class creation corresponds to building a repository for objects independently of its type. We name classes through declarations. Insertion into a subclass is performed in several steps: first the object is inserted into the base class, then the object is specialised from the base class down to the subclass. In according with the object oriented principles, objects are unique even if they belong to several classes. Other primitives on classes are:

> **remove**($o$) : removes object $o$ from all classes it belongs to.
>
> **for_class**($cl, f$) : applies function $f$ to all elements of class $cl$, and returns the sequence of results.
>
> **obj_of**($o, lb$) : selects the value associated to field $lb$ of object $o$.

The shared persistent environment is the result of the design phase of the database when basic and constructed values, including classes, are declared. When the definition of a shared persistent environment is completed, it is made persistent and stored in a centralised server. Afterwards, every client can execute transactions. PPL transactions are top-level expressions that can access values of identifiers declared in the shared persistent environment. Inside a transaction, local values can be declared for the sake of computation (construct *use_in*). Figure 1 shows the architecture underlying PPL. It adopts the client-server model that is typical of Object Oriented Databases and Persistent Programming Languages architectures [12,13]. The server handles the shared persistent store, i.e. the database, and the shared persistent environment that contains mappings from global identifiers to locations of the store. The scheduler is the module of the server devoted to concurrency control. When a client begins a session, it loads a copy of the shared persistent environment from the server to solve locally identifier bindings and to use the communication network only to access the stored values. A session consists of sequential execution of many transactions.

The concurrency control protocol used by the transaction scheduler running on the server, is the *conservative two-phase locking*. Briefly, this protocol [7] entails knowing the read and write set of each transaction before its execution, if all data are available they are locked till the end of the transaction. This protocol avoids deadlocks but needs the programmer to declare the read and write set for each transaction. Our idea is that the static analysis could supply this information.

The following example shows the population of class *base* with the object ($a$ : **var**(1), $b$ : **var**(1)) and the population of the subclass *sub* with the object ($a$ : **var**(2), $b$ : **var**(2), $c$ : 2, $d$ : 2) which specialises the object ($a$ : **var**(2), $b$ : **var**(2)) of class *base*:

```
use(base,class());
use(sub,class());
insert (base,rec(a,b,var(1),var(1)));
specialise(insert(base, rec(a,b,var(2),var(2)),
                  rec(c,d,2,2), sub))
```

The first two declarations define the *Shared Persistent Environment*, i.e. a global environment shared among all transactions. Once defined, bindings of the Shared Persistent Environment can never change. Transactions can only affect values stored in locations bound to global identifiers of an updatable type. The last two expressions of the previous example are transactions.

## 3 Exact non-standard semantics

*Exact non-standard semantics* behaves like the standard one as far as control flow is concerned, whereas it keeps track of all accesses to modifiable elements. The non-standard value of a modifiable element allows one to derive the actual number of read and write accesses to that element at a certain moment in the transaction

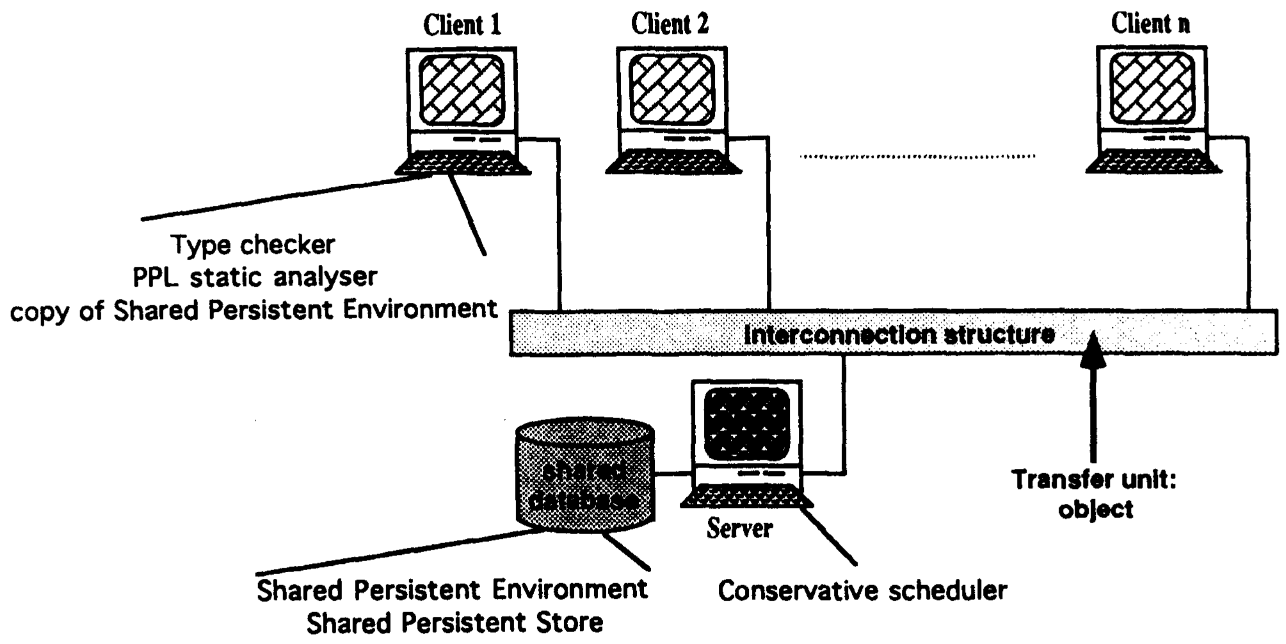

Figure 1: Architecture of the PPL system

execution. To this aim we redefine the evaluation of all the language primitives so that whenever a modifiable element is accessed, its number of read or write accesses is incremented accordingly. In this approach, the readset and the writeset of a transaction are represented by the set of elements with a positive number of accesses.

## Exact semantic domains

The following are the semantic domains on which the mapping between syntactic and semantic denotations rely:

| | |
|---|---|
| $Bve = Ide \rightarrow Val$ | Environment |
| $Val = Basic + Rec + Seq + Fun + Lfval$ | Values |
| $Basic = \{none\}$ | Basic values |
| $Rec = (Lab \rightarrow_{fin} Val)$ | Records |
| $Seq = Val^*$ | Sequences |
| $Fun = Val^* \rightarrow St \rightarrow (Val \times St)$ | Functions |
| $Lfval = Loc$ | References |
| $Lfobject = Loc$ | Identity of objects |
| $Lfclass = Loc$ | Class pointers |
| $Loc = Nat$ | Locations |
| $St = Loc \rightarrow (Ac + \{free\})$ | Store |
| $Ac = Rc \times Wc \times Rgval$ | Access counters |
| $Rc = Nat$ | Read access counters |
| $Wc = Nat$ | Write access counters |
| $Rgval = Val + Class + Object$ | Storable values |
| $Object = Rec \times Lfclass^*$ | Objects |
| $Class = Lfobject^*$ | Classes |

Values of basic types are irrelevant and are evaluated to the special value *none*. Modifiable values, *References*, *Classes* and *Objects*, are accessed through the locations which refer them. Each location is also associated with a read counter and a write counter. Object identity is established by the location where the object is stored. Objects also carry the information on the classes they belong to.

The *exact semantics* is defined by the function:

$$\mathcal{R} : Exp \rightarrow Bve \rightarrow St \rightarrow Ans$$

where $Ans = Val \times Bve \times St$. Function $\mathcal{R}$ and its abstract version $\mathcal{AR}$, presented in the next section, are defined by structural induction on the terms of the abstract syntax of the language. The rest of this section presents the equations of $\mathcal{R}$ which are relevant to the access counting. Particular emphasis is on the equations dealing with classes and objects. An exhaustive presentation of $\mathcal{R}$ and can be found in [5].

## Exact semantic function

The evaluation of *at* (the reference access primitive) is[2]:

```
$\mathcal{R}\,[at(l)]bve\ st =$
   let $< loc, bve_1, st_1 > = \mathcal{R}[l]bve\ st$
       $val = st_1(loc)\downarrow_{Rgval}$
       $st_2 = st_1[< (st_1(loc)\downarrow_{Rc}) + 1, st_1(loc)\downarrow_{Wc}, val > /loc]$
   in $< val, bve, st_2 >$
```

[2]To select a component from an element of a Cartesian product, the operator $\downarrow$ is adopted, followed by the name of the component domain; e.g. if $ans \in Ans$ then $ans \downarrow_{Val} \in Val$. When more than one component have the same domain name an index is used to solve the ambiguity.

Here the argument of *at* is a reference. Since the primitive *at* reads the reference content, we increment the read counter of the reference.

The assignment primitive is defined as follows:

```
$\mathcal{R}\ [\leftarrow (l, e)]bve\ st =$
  let $< val, bve_1, st_1 > = \mathcal{R}[e]bve\ st$
      $< loc, bve_2, st_2 > = \mathcal{R}[l]bve\ st_1$
      $st_3 = st_2[< st_2(loc) \downarrow_{Rc}, (st_2(loc) \downarrow_{Wc}) + 1, val > /loc]$
  in $< none, bve, st_3 >$
```

The arguments of $\leftarrow$ are a reference and a value respectively. The store is updated by replacing the value and by incrementing the write counter of the reference. The evaluation of this primitive returns *none* since the standard evaluation returns the basic constant *nil*.

```
$\mathcal{R}\ [class]bve\ st =$
  let $cl = new\_loc(st)$
      $st' = st[< 0, 0, <> > /cl]$
  in $< cl, bve, st' >$
```

When a class is created, a new location is allocated with an empty sequence of *object identifiers*. Read and write counters of the new class are initialized to 0.

```
$\mathcal{R}[insert(x, e)]bve\ st =$
  let $cl = bve(x)$
      $< tup, bve_1, st_1 > = \mathcal{R}[e]bve\ st$
      $obj = < tup, < cl > >$
      $objs = st_1(cl) \downarrow_{Rgval}$
      $loc = new\_loc(st_1)$
      $st' = st_1[< 0, 0, obj > /loc]$
      $st'' = st'[< ( st'(cl) \downarrow_{Rc}) + 1,$
                $(st'(cl) \downarrow_{Wc}) + 1, objs. < loc > > /cl]$
  in $< loc, bve, st'' >$
```

A new object is composed by a pair which contains the record returned by the evaluation of $e$ and the class $cl$ bound to $x$. This new object is associated with a new location with zero read/write counters and it is added to the class $cl$. The read and the write counters of $cl$ are both incremented.

```
$\mathcal{R}[specialize(e_1, e_2, x)]bve\ st =$
  let $cl = bve(x)$
      $< loc, bve_1, st_1 > = \mathcal{R}[e_1]bve\ st$
      $< tup, bve_3, st_2 > = \mathcal{R}[e_2]bve\ st_1$
      $obj = st_2(loc) \downarrow_{Rgval}$
      $updated\_obj = < obj \downarrow_{Rec} [tup], obj \downarrow_{Lfclass} . < cl > >$
      $st' = st_2[< (st_2(loc) \downarrow_{Rc}) + 1,$
                $(st_2(loc) \downarrow_{Wc}) + 1, updated\_obj > /loc]$
      $st'' = st'[< (st'(cl) \downarrow_{Rc}) + 1,$
                $(st'(cl) \downarrow_{Wc}) + 1, (st'(cl) \downarrow_{Rgval}). < loc > > /cl]$
  in $< loc, bve, st'' >$
```

The object $e_1$ is specialised with the fields of $e_2$. The temporary variable *updated_obj* is a pair. The first element contains the extension of the record referred by $e_1$ with $e_2$[3]. The second element contains the sequence of classes to which the object belongs, including the new class specified by $x$. The object is stored in its old location *loc*, and its identity is also inserted in the extension of subclass $x$. Finally, read/write counters of both class and object are incremented.

[3]Record redefinition is obtained using the metafunction []: $f[f'](x) = f'(x)$ *if* $x \in dom(f')$, $f(x)$ *elsewhere*

```
$\mathcal{R}[remove(e)]bve\ st =$
  let $< loc, bve_1, st_1 > = \mathcal{R}[e]bve\ st$
      $< cl_1, \ldots, cl_n > = st_1(loc) \downarrow_{Lfclass}$
      $st_2 = st_1[< st_1(cl_1) \downarrow_{Rc} +1,$
                $st_1(cl_1) \downarrow_{Wc} +1, rem(loc, st_1(cl_1) \downarrow_{Rgval}) > /cl_1]$
      $st_3 = st_2[< st_2(cl_2) \downarrow_{Rc} +1,$
                $st_2(cl_2) \downarrow_{Wc} +1, rem(loc, st_2(cl_2) \downarrow_{Rgval}) > /cl_2]$
                ....
      $st_{n+1} = st_n[< st_n(cl_n) \downarrow_{Rc} +1,$
                $st_n(cl_n) \downarrow_{Wc} +1, rem(loc, st_n(cl_n) \downarrow_{Rgval}) > /cl_n]$
      $st' = st_{n+1}[< st_{n+1}(loc) \downarrow_{Rc} +1,$
                $st_{n+1}(loc) \downarrow_{Wc}, st_{n+1}(loc) \downarrow_{Rgval}) > /loc]$
  in $< none, bve, st' >$
```

The object $e$ is removed from all classes $< cl_1, \ldots, cl_n >$ to which it belongs, this task is performed by function *rem*. The read/write counters of those classes are both incremented.

Other interesting primitives which operate on classes and objects are *for_class* and *obj_of*. The former allows one to visit the extension of a class, while the latter allows a component of an object to be selected. Their semantics, omitted in this paper, is a natural extension of the correspondent primitives for sequences and records.

## 4 Abstract semantics

The primary concern of our analysis is to guarantee the termination of abstract evaluation which merely entails guaranteeing that domains of the abstract semantics are finite. The basic ideas are that each abstract value is associated with a location of the appropriate type. Each *typed* location domain is assumed to be finite and the size of this domain is statically determined by a textual analysis of the transaction. In other words, each location domain is a finite set of indexes of the corresponding abstract store.

### Abstract domains

The domain of abstract values is defined as follows:

$$Aval = Aloc_l + Aloc_r + Aloc_s + Aloc_f + \{none\}$$

and abstract location domains are:

$$Aloc_{type} = \{1, \ldots, old\_loc_{type}, old\_loc_{type} + 1, \ldots, old\_loc_{type} + max\_loc_{type}\}$$

Here *old_loc* is the initial number of non-free locations in the global store; *max_loc* is the number of new locations that are expected to be used during the transaction execution. *max_loc* is defined as the number of occurrences of constructors that initialise new locations. Consequently, there is a one to one correspondence between constructor occurrences and new locations: whenever the same constructor occurrence is evaluated the abstract evaluation returns the same location, whereas the exact semantics would initialise a new location every time.

There are other ways to choose the value of *max_loc*. For instance it could be defined as the product of the number of occurences of the constructor by the number of occurences of calls of the function which uses this constructor. The way we choose *max_loc* establishes the accuracy of the analysis, it is a sort of *tuner* of how many concrete values will coincide with the same abstract value.

The store that in the exact semantics was a function from locations to storable values, is now a product of functions from typed locations to a set of abstract storable values

$$Ast = Ast_l \times Ast_r \times Ast_s \times Ast_f$$

In the following we discuss every single abstract store.

- The abstract store for references is defined as follows:

$$Ast_l \;=\; Aloc_l \rightarrow (Aac + \{free\})$$

where:

$$\begin{array}{lcl} Aac & = & Arc \times Awc \times Argval \\ Argval & = & P(Aval) + Aclass + P(Aobject) \\ Arc,\ Awc & = & \{0, \infty\} \end{array}$$

An abstract location is associated with its abstract read and write counters and with either a set of abstract values, or a set of abstract objects or one abstract class. In fact, in our language classes can be declared only at top level, so we can have only one application of a class constructor. *Arc* and *Awc* are two-value domains: $\{0, \infty\}$ with the intended meaning *not accessed* and *accessed one or more times.* The meaning of the abstract value $\infty$ is that the corresponding exact locations cannot be released before the end of the transaction. There might be other interesting abstractions, for instance the set $\{1, \ldots, max\_ac, \infty\}$ represents explicitly how many times a data structure is accessed. In our case the policy used by the scheduler only needs to know which data have been modified, but other optimizations could rely on how many times a structure has been touched. In [5] the safety of abstract interpretation has been proved with respect to the latter choice which is more restrictive than using $\{0, \infty\}$.

- The abstract store for records is:

$$Ast_r \;=\; Aloc_r \rightarrow ((Lab \rightarrow_{fin} P(Aval)) + \{free\})$$

We choose not to repeat the labels of the fields of the abstract records associated with an abstract location because they never change. In fact a single abstract record (and not a set of abstract records) is associated with an abstract record location. Therefore each field of this record can carry a set of values.

- The abstract store for sequences is:

$$Ast_s \;=\; Aloc_s \rightarrow (P(Aseq) + \{free\})$$

where an abstract sequence is:

$$Aseq \;=\; Len \times Pos \times (Pos \rightarrow P(Aval))$$

To represent sequences we use an array of sets of abstract values. The sequence elements are inserted circularly in the array. So if, for instance, the length of the array is $n$, then the $i-th$ element of the sequence is stored in the position $((i-1) \bmod n)$ of the array. The least accurate approximation is the one with an array of length 1, where all the elements of the sequence are collected into a single set. Increasing the length of the array will augment the accuracy of the representation. The first two fields of the abstract sequences denote the length and the first free position of the array respectively, while the array is a function from position to abstract values.

The following are the domains which represent the sequence length and the next free position respectively.

$$\begin{array}{lcl} Len & = & \{0, \ldots max\_seq, \infty\} \\ Pos & = & \{0, \ldots, max\_pos - 1\} \end{array}$$

On the *Len* domain the following operations are defined:

$$\begin{array}{lcl} x +^l n & = & if (x+n) \le max\_seq\ then (x+n) \\ & & else\ \infty \\ x -^l n & = & if x = \infty\ then\ \infty \\ & & else\ x - n \end{array}$$

On *Pos* domain is defined:

$$x +^p n = (x+n) \bmod max\_pos$$

- The abstract store for functions is:

$$Ast_f = Aloc_f \rightarrow (Afun + \{free\})$$

where

$$Afun = (P(Aval))^* \rightarrow Ast \rightarrow P(P(Aval) \times Ast)$$

The primitive *function* has a special treatment. In the exact semantics when the same primitive is executed a second time, a new closure is generated. In the case of abstract semantics the second closure abstracts the previous, so we can store only the last one. As a consequence, only an abstract function will be associated with a function location and not a set of abstract functions.

- The abstract domains for objects and classes are defined as follows:

$$\begin{array}{lcl} Aobject & = & Arecord \times Aclasses \\ Aclass & = & Len \times Pos \times (Pos \rightarrow P(Aloc_l)) \end{array}$$

  where the domains:

$$\begin{array}{lcl} Arecord & = & \mathcal{P}(Aloc_r) \\ Aclasses & = & \mathcal{P}(Aloc_l) \end{array}$$

denote the abstraction of the record representing the object and the set of classes to which the object belongs respectively. The abstract domain for classes is similar to the domain for sequences with the difference that a class can only contain objects.

Notice that we do not need ad hoc abstract stores for classes and objects. In fact the reference store can be used because classes and objects are modifiable elements.

- The abstract environment is defined as follows:

$$Abve = Ide \rightarrow P(Aval)$$

where an identifier is associated with a set of abstract values.

## Abstract semantic function

The abstract evaluation of an expression returns a set of abstract values, in fact it has to mime all possible paths of the exact execution. The most intuitive example is the execution of the primitive $if - then - else$, where the results of both alternatives have to be considered.

The abstract non-standard semantics is defined by the following function:

$$\mathcal{AR} : Exp \rightarrow Abve \rightarrow Ast \rightarrow P(Aans)$$

where the abstract answers are:

$$Aans = P(Aval) \times Abve \times Ast$$

The next equation defines the abstract evaluation of the primitive $at$:

$$\begin{array}{l}
\mathcal{AR}\ [at(e)]abve\ ast = \\
\quad \{< VAL, abve, ast_2 > | \\
\qquad\quad < LOC_l, abve_1, ast_1 > \in \mathcal{AR}[e]abve\ ast \\
\qquad\quad ast_l = ast_1 \downarrow_{Ast_l} \\
\qquad\quad VAL = \cup\{ast_l(loc) \downarrow_{Argval} \mid loc \in LOC_l\} \\
\qquad\quad ast_l' = ast_l\ [< \infty, ast_l(loc) \downarrow_{Arc}, ast_l(loc) \downarrow_{Argval} > /loc] \\
\qquad\qquad\qquad \forall loc \in LOC_l \\
\qquad\quad ast_2 = < ast_l', ast_1 \downarrow_{Ast_r}, ast_1 \downarrow_{Ast_s}, ast_1 \downarrow_{Ast_f} > \}
\end{array}$$

$\mathcal{AR}[at(e)]$ returns a set of triples whose first component $VAL$ is the sum of the values stored in the set of location $LOC_l$ returned from the abstract evaluation of the argument $e$. Moreover for each location the read counter is set to $\infty$.

The abstract evaluation of the assignment is:

$$\begin{array}{l}
\mathcal{AR}\ [\leftarrow (l, e)]abve\ ast = \\
\quad \{< \{none\}, abve, ast_3 > | \\
\qquad\quad < VAL, abve_1, ast_1 > \in \mathcal{AR}[e]abve\ ast \\
\qquad\quad < LOC_l, abve_2, ast_2 > \in \mathcal{AR}[l]abve_1\ ast_1 \\
\qquad\quad ast_l = ast_2 \downarrow_{Ast_l} \\
\qquad\quad ast_l' = ast_l[< ast_l(loc) \downarrow_{Arc}, \infty, \\
\qquad\qquad\quad (ast_l(loc) \downarrow_{Argval}) \cup VAL > /loc] \\
\qquad\qquad\quad \forall loc \in LOC_l \\
\qquad\quad ast_3 = < ast_l', ast_2 \downarrow_{Ast_r}, ast_2 \downarrow_{Ast_s}, ast_2 \downarrow_{Ast_f} > \}
\end{array}$$

$l$ is evaluated with respect to all the stores returned by the abstract evaluation of $e$; then all the stores are updated with the evaluation of $l$ and the write counters are set to $\infty$.

$$\begin{array}{l}
\mathcal{AR}[class_{cl}]abve\ ast = \\
\quad \{< \{cl\}, abve, ast' > | \\
\qquad\quad ast_l = ast \downarrow_{Ast_l} \\
\qquad\quad ast_l' = ast_l\ [< 0, 0, < 0, 0, [\,] >> /cl] \\
\qquad\quad ast' = < ast_l', ast \downarrow_{Ast_r}, ast \downarrow_{Ast_s}, ast \downarrow_{Ast_f} > \}
\end{array}$$

Each occurrence of the primitive $class$ has a location $cl$ associated with it. The abstract interpretation associates an empty class with this location and propagates the previous $read/write$ counters through the store $ast'$.

$$\begin{array}{l}
\mathcal{AR}[insert_{loc}(x, e)]abve\ ast = \\
\quad \{< \{loc\}, abve, ast' > | \\
\qquad\quad cl \in abve(x) \\
\qquad\quad < LOC_r, abve_1, ast_1 > \in \mathcal{AR}[e]abve\ ast \\
\qquad\quad obj = < LOC_r, \{cl\} > \\
\qquad\quad ast_l = ast_1 \downarrow Ast_l \\
\qquad\quad objs = ast_l(cl) \downarrow Ast_s \\
\qquad\quad ast_l' = \textbf{if}\ ast_l(loc) = free \\
\qquad\qquad\quad \textbf{then}\ ast_l[< 0, 0, \{obj\} > /loc] \\
\qquad\qquad\quad \textbf{else}\ ast_l[< ast_l(loc) \downarrow_{Rc}, ast_l(loc) \downarrow_{Wc}, \\
\qquad\qquad\qquad\qquad ast_l(loc) \downarrow_{Argval} \cup \{obj\} > /loc] \\
\qquad\quad ast_l'' = ast_l'[< \infty, \infty, in(objs, \{loc\}) > /cl] \\
\qquad\quad ast' = < ast_l'', ast \downarrow_{Ast_r}, ast \downarrow_{Ast_s}, ast \downarrow_{Ast_f} > \}
\end{array}$$

A new object which contains the set of records $LOC_r$ returned by the evaluation of $e$ is created and inserted into the class $cl$. The function $in$ inserts a set of abstract objects in the first free position of the array representing the class. The abstract read/write counters of the class are both set to $\infty$, while read/write counters of the other objects of the class remain the same. This implies that inserting an object into a class can be executed in parallel with any other operation on objects of that class if they are reached from somewhere else.

$$\begin{array}{l}
\mathcal{AR}[specialize(e_1, e_2, x)]abve\ ast = \\
\quad \{< LOC_l, abve, ast' > | \\
\qquad\quad cl \in abve(x) \\
\qquad\quad < LOC_l, abve_1, ast_1 > \in \mathcal{AR}[e_1]abve\ ast \\
\qquad\quad < LOC_r, abve_2, ast_2 > \in \mathcal{AR}[e_2]abve\ ast_1 \\
\qquad\quad ast_r = ast_2 \downarrow_{Ast_r}
\end{array}$$

$$\begin{aligned}
& loc_r \in LOC_r \\
& tup = ast_r(loc_r) \\
& ast_l = ast_2 \downarrow_{Ast_l} \\
& ast'_r = ast_r[(([(tup(lab) \cup ast_r(loc'_r)(lab))/lab] \\
& \qquad\qquad \forall lab \in Lab)/loc'_r], \\
& \qquad \forall loc'_r \in (obj \downarrow_{Aloc_r}), obj \in ast_l(loc_l), loc_l \in LOC_l \\
& ast'_l = ast_l[< \infty, \infty, < ((ast_l(loc_l) \downarrow_{Argval}) \downarrow_{Aloc_r}), \\
& \qquad (ast_l(loc_l) \downarrow_{Argval}) \downarrow_{Aloc_r}) \cup \{cl\} >> /loc_l] \\
& \qquad \forall loc_l \in LOC_l \\
& ast''_l = ast'_l[< \infty, \infty, in((ast'_l(cl) \downarrow_{Argval}, LOC_l) > /cl] \\
& ast' =< ast''_l, ast'_r, ast \downarrow_{Ast_s}, ast \downarrow_{Ast_f} >\}
\end{aligned}$$

This operation is very expensive. $LOC_r$ is the set of records with which to specialize all the objects in $LOC_l$: each object will have $k$ new fields with a set of abstract values for each of them.

$$\begin{aligned}
& \mathcal{AR}[remove(e)]abve\ ast = \\
& \quad \{< \{ none\}, abve, ast' > | \\
& \qquad < LOC_l, abve_1, ast_1 > \in \mathcal{AR}[e]abve\ ast \\
& \qquad ast_l = ast_1 \downarrow_{Ast_l} \\
& \qquad loc_l \in LOC_l \\
& \qquad obj \in ast_l(loc_l) \downarrow_{Argval} \\
& \qquad ast'_l = ast_l[< \infty, \infty, arem(ast_l(cl) \downarrow_{Argval}) > /cl] \\
& \qquad\qquad \forall cl \in obj \downarrow_{Aclasses} \\
& \qquad ast''_l = ast'_l[< \infty, ast'_l(loc_l) \downarrow_{Awc}), ast'_l(loc_l) \downarrow_{Argval} > /loc_l] \\
& \qquad ast' =< ast''_l, ast_1 \downarrow_{ast_r}, ast_1 \downarrow_{ast_s}, ast_1 \downarrow_{ast_f} >\}
\end{aligned}$$

The abstract removal of an object consists in changing the positions of the elements in the class. The abstract elements in position $i_{th}$ are duplicated in position $i_{th} - 1$ because we do not know which element has really been removed (this task is performed by the function *arem*). Correctness is thus preserved, but if a single transaction requires *max_pos* removals, then all positions contain all abstract elements. In this case the accuracy of the analysis is poor, but it will not affect abstract execution of other transactions.

## 4.1 Correctness and termination

In [5] the correctness is formally proved of an analysis with the domain of the read/write counters equal to $\{0, \ldots, max\_ac, \infty\}$ instead of $\{0, \infty\}$. The parameter *max_ac* is an estimation of the maximum number of accesses a transaction can do on a data structure. This analysis is more general than the one presented in this paper, in fact it determines how many times a data structure has been touched. From a correctness point of view the results obtained in the general case hold in our restriction.

Informally, the abstract evaluation $\mathcal{AR}$ is correct with respect to the exact evaluation $\mathcal{R}$, if the number of accesses to each location obtained with $\mathcal{R}$ is less than or equal to the number of accesses to the corresponding locations obtained with $\mathcal{AR}$. This means that a scheduler based on this information will lock more or equal data than those requested by the real execution, but is granted to be deadlock free.

The proof of correctness is formally developed by defining a family of typed relations $\subseteq_{type}$ between exact and abstract values of the various types of the language. Firstly, correctness for basic domains is defined, then it is defined for structured domains until correctness relation between the functions of semantic interpretation has been defined. Finally, it is proved that $\mathcal{AR}$ and $\mathcal{R}$ applied to PPL sentences are correct.

**Def. 1** *Consider the functions* $\mathcal{AF}$ *and* $\mathcal{F}$
$\mathcal{AF} : Exp \rightarrow Abve \rightarrow Ast \rightarrow Aans$ *and*
$\mathcal{F} : Exp \rightarrow Bve \rightarrow St \rightarrow Ans.$
$\mathcal{AF} \subseteq_{Fun} \mathcal{F}$ *mboxiff*
$(\forall abve \in Abve, ast \in Ast, bve \in Bve, st \in St, e \in Exp:$
$< abve, ast > \subseteq_{Env} < bve, st > \rightarrow$
$(\mathcal{AF}[e]abve\ ast \subseteq_A \mathcal{F}[e]bve\ st))$

The above definition says that two functions are respectively correct if applied to the same expression in environments and stores which are respectively correct produce results which are respectively correct. Correctness between results is established by the existence of at least one abstract result in the set returned by the abstract evaluation which is correct with respect to the exact one. This statement is established by the following:

**Def. 2** *Let* $S \in \mathcal{P}(Aans), t \in Ans,$
$(S \subseteq_A t)$ *iff* $((\exists s \in S$ *such that* $s \subseteq_{Ans} t) \vee (t = \perp_{Ans}))$

Finally the following proposition establishes correctness between the abstract and exact semantics:

**Proposition 1** $\mathcal{AR}$ *is correct respect with* $\mathcal{R}$, *iff* $\mathcal{AR} \subseteq_{Fun} \mathcal{R}$

The termination of the analysis is guaranteed by the finiteness of the abstract domains and by the monotonicity of the function that performs the analysis on such domains. In fact, the only condition in which our analysis might not terminate, is when a fix point operator is involved in the evaluation of a sentence; but the fix point operator is always used with monotonic functions defined on finite domains, hence the fix point is finitely computable.

## 5 Examples of abstract evaluation

In this section two different examples are shown. The first uses the construct *if−then−else* whose evaluation returns a set of sets of tuples. The second is concerned with our real objective: to gather information for the scheduler of transactions in a database. This example only illustrates a few steps of the abstract execution and highlights the final results.

### Example1

Let the following be the abstract evaluation for the construct *if − then − else*.

$$\mathcal{AR}[if(p,t,e)]\ abve\ ast = \bigcup\{(\ \mathcal{AR}[t]abve'\ ast' \cup \mathcal{AR}[e]abve'\ ast') \mid < \{none\}, abve', ast' > \in \mathcal{AR}[p]abve\ ast)\}$$

Then abstract evaluation of the expression:

$$(\mathbf{if}(>(\mathbf{at}(x), 100), \leftarrow (x, 10), \leftarrow (y, 10))$$

w.r.t. the environment:

$$abve = \{(x, \{loc_1, loc_2\}), (y, \{loc_3\})\}$$

and the store:

$$ast_i = \{(\ loc_1, < r_1, w_1, \{none\} >), (loc_2, < r_2, w_2, \{none\} >), (loc_3, < r_3, w_3, \{none\} >)\}$$

returns the set of triples:

$$\begin{aligned}\{ <& \{none\},\\ &\{(x, \{loc_1, loc_2\}), (y, \{loc_3\})\},\\ &\{(\ loc_1, < \infty, \infty, \{none\} >), (loc_2, < \infty, \infty, \{none\} >),\\ &\quad (loc_3, < r_3, w_3, \{none\} >)\} >,\\ <& \{none\},\\ &\{(x, \{loc_1, loc_2\}), (y, \{loc_3\})\},\\ &\{(\ loc_1, < \infty, w_1, \{none\} >), (loc_2, < \infty, w_2, \{none\} >),\\ &\quad (loc_3, < r_3, \infty, \{none\} >)\} >\}\end{aligned}$$

In this case the conservative scheduler will consider to sum of the returned triples and will lock locations $loc_1$, $loc_2$ and $loc_3$ which contain $x$ and $y$ values.

### Example2

Consider the following expressions defining a database schema:

```
use rec employees class
        employee ↔
                (| name: string
                and code: string
                and boss: var(manager)
                and ... |)
        managers subset of employees class
        manager ↔
                (| is employee
                and team: var ( seq (employee))
                and ... |) ;
```

Every employee has one manager, while every manager has a team composed of a modifiable sequence of employees. The following transactions T1 and T2 populate the previous schema with an employee object and a manager:

```
(use (mgr, get(managers,cond'));
(use(emp, rec_tloc4 (name,code,boss,...,
                    'Smith',1234,var(seq(mgr)),...));
    insert_obj4(emp,employees)))
```

```
(use (emp, get (employees,cond"));
(use(mngr, rec_tloc4(team,...,seq(),...));
    specialize(emp,mngr,managers)))
```

The index at the occurrence of each constructor like $\mathbf{insert}_{obj4}$ represents the location on which it operates. The concurrent execution of the above transactions might lead to deadlock because the former reads the class *managers* and updates the class *employees*, while the latter reads the class *employees* and updates the class *managers*.

Tables 1 and 2 show the initial abstract environment and the initial abstract store for this example.

The abstract interpretation of transaction T1 returns the store in Table 3. In the table, the identifiers in bold denote items involved by the first transaction which reads the locations $cl_2$, $obj_1$, and updates the location $cl_1$. Table 4 shows the abstract store returned by the abstract interpretation of the second transaction. The abstraction of primitive *get* returns all objects of class *employees*, so the transaction reads the locations $cl_1, obj_1, obj_2, obj_3$ and updates $cl_2, obj_1, obj_2, obj_3$. Comparing the results the scheduler detects the risk of deadlock.

## 6 Discussion and further work

The results presented in this paper are the first part of our experiment. They confirm that we can obtain a correct approximation of the read/write set of a transaction by static analysis with respect to a snapshot of the environment and the store. An implementation of such an approach must consider that the global store is affected by the execution of other transactions, i.e. it has to deal with concurrent accesses to the store.

Two important questions are involved when assessing the applicability of this approach in the Data Base context:

- what is the cost of the analysis in terms of space and time, and how much does it affects the efficiency of a transaction execution?
- is the information gathered by the analysis significant, i.e., is it really useful for realistic optimizations?

Both the answers to these questions are concerned with the *granularity* of the analysis. In fact, the accuracy of the analysis depends on whether the abstract representation of database is finer or coarser. The finest abstraction is the one corresponding to the actual database: each object which is in the Data Base at the moment of the transaction execution is represented.

| abve | |
|---|---|
| *employees* | $\{cl_1\}$ |
| *managers* | $\{cl_2\}$ |

Table 1: Initial Abstract Envinronment.

| $ast_r$ | | $ast_t$ | |
|---|---|---|---|
| $cl_1$ | $\{< 0, 0, < 3, 3, [\{obj_1\}/0, \{obj_2\}/1, \{obj_3\}/2] >>\}$ | $tloc_1$ | $[\ldots]$ |
| $obj_1$ | $\{< 0, 0, < \{tloc_1\}, \{cl_1, cl_2\} >>\}$ | $tloc_2$ | $[\ldots]$ |
| $obj_2$ | $\{< 0, 0, < \{tloc_2\}, \{cl_1\} >>\}$ | $tloc_3$ | $[\ldots]$ |
| $obj_3$ | $\{< 0, 0, < \{tloc_3\}, \{cl_1\} >>\}$ | $tloc_4$ | *free* |
| $cl_2$ | $\{< 0, 0, < 1, 1, [\{obj_1\}/0] >>\}$ | | |
| $obj_4$ | *free* | | |

Table 2: Initial Abstract Store.

| $ast_r$ | | $ast_t$ | |
|---|---|---|---|
| $\mathbf{cl_1}$ | $\{< 0, \infty, < 4, 4, [\{obj_1\}/0, \{obj_2\}/1, \{obj_3\}/2, \{obj_4\}/3] >>\}$ | $tloc_1$ | $[\ldots]$ |
| $\mathbf{obj_1}$ | $\{< \infty, 0, < \{tloc_1\}, \{cl_1, cl_2\} >>\}$ | $tloc_2$ | $[\ldots]$ |
| $obj_2$ | $\{< 0, 0, < \{tloc_2\}, \{cl_1\} >>\}$ | $tloc_3$ | $[\ldots]$ |
| $obj_3$ | $\{< 0, 0, < \{tloc_3\}, \{cl_1\} >>\}$ | $tloc_4$ | $[\ldots]$ |
| $\mathbf{cl_2}$ | $\{< \infty, 0, < 1, 1, [\{obj_1\}/0] >>\}$ | | |
| $\mathbf{obj_4}$ | $\{< 0, 0, < \{tloc_4\}, \{cl_1\} >>\}$ | | |

Table 3: Final Abstract Store for T1.

| $ast_r$ | | $ast_t$ | |
|---|---|---|---|
| $\mathbf{cl_1}$ | $\{< \infty, 0, < 3, 3, [\{obj_1\}/0, \{obj_2\}/1, \{obj_3\}/2] >>\}$ | $tloc_1$ | $[\ldots]$ |
| $\mathbf{obj_1}$ | $\{< \infty, \infty, < \{tloc_1\}, \{cl_1, cl_2\} >>\}$ | $tloc_2$ | $[\ldots]$ |
| $\mathbf{obj_2}$ | $\{< \infty, \infty, < \{tloc_2\}, \{cl_1, cl_2\} >>\}$ | $tloc_3$ | $[\ldots]$ |
| $\mathbf{obj_3}$ | $\{< \infty, \infty, < \{tloc_3\}, \{cl_1, cl_2\} >>\}$ | $tloc_4$ | $[\ldots]$ |
| $\mathbf{cl_2}$ | $\{< 0, \infty, < 2, 2, [\{obj_1\}/0, \{obj_1, obj_2, obj_3\}/1] >>\}$ | | |
| $obj_4$ | *free* | | |

Table 4: Final Abstract Store for T2.

The coarsest abstraction is one where only the types and the classes declared in the global environment are represented. Between these two extremes there is a whole range of intermediate possibilities, which correspond to trade-offs between accuracy and cost of the analysis.

Our analyser provides some *tuning* mechanisms to control the granularity of the analysis i.e. the parameters which specify the size of the abstract domains. An example of such a parameter is the cardinality of the abstract classes (the length of the array which simulates the extension of the class) which establishes the ratio between abstract objects and concrete ones. For instance, an interesting analysis can be obtained by representing classes (and sequences) with an array wich contains a single representative object. This choice is based on the fact that abstraction of search operations always returns the indistinct set of all objects in the class. The resulting analysis is very efficient. In fact, the abstract locations are statically determined (corresponding to different occurrences of the constructors), so, the only things that can change in a recursive call are the counters of the objects. In this case there is only one object thus yielding to a fast convergence of the fix-point computation.

Further study is needed to learn how to use these tuning mechanisms. We believe that this knowledge can be gained from experiments and that the mechanisms can be correlated with other application-dependent parameters such as the concurrency rate.

This part of the experiment is currently under study. The first next step is the development of a prototype of the analyser and its integration with the existing PPL system, so that the analysis, the scheduling and the execution of transactions coexist and interact. The second step is to test some significant case studies in order to evaluate the performance figures obtained.

## References

[1] S. Abramsky and C. Hankin (eds), *Abstract interpretation of declarative languages*, Ellis Horwood, Chichester, UK, 1987.

[2] S. Abramsky, Abstract Interpretation, Logical Relations, and Kan Extensions, *Journal of Logic and Computation, Vol 1, No.1*, 1990.

[3] A. Albano A., L. Cardelli and R. Orsini, Galileo: A Strongly Typed, Interactive Conceptual Language, *ACM Transaction on Database Systems, Vol. 10, No. 2*, 230-260, 1985.

[4] G. Amato, F. Giannotti and G. Mainetto, Analysis of Concurrent Transaction in a Functional Database Programming Language, *Proc. Workshop on Static Analysis*, Bordeaux,174-184,1992.

[5] G. Amato, *Definizione di un Interprete Astratto per l'ottimizzazione dell'esecuzione di transazioni*, Tesi di Laurea, Dip. di Informatica di Pisa, 1992.

[6] M.P. Atkinson and O.P. Buneman, Types and Persistence in Database Programming Languages, *ACM Comp. Surv., Vol 19, No. 2*, 105-190, 1987.

[7] P. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database System*, Addison-Wesley, Cambridge, MA, 1987.

[8] P. Cousot and R. Cousot, Abstract interpretation: An unified lattice model for static analysis of programs by construction of approximation of fixpoints, *Proc. 4th POPL*, 238-252, 1977.

[9] P. Cousot and R. Cousot, Static determination of dynamic properties of programs, *Proc. of the 2nd Int. Symp. on Programming Languages*, Dunod, Paris, 1976.

[10] P. Cousot and R. Cousot, Abstract Interpretation Frameworks, *Jour. of Logic and Comp.*, 1992.

[11] A. Deutsch, On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications, *ACM Symposium on Principles of Programming Languages*, San Francisco CA, 157-168, 1990.

[12] D.J. De Witt, P. Futtersack, D. Maier and F. Velez, A study of three alternative workstation-server architectures for Object Oriented Databases System, Proc. of VLDB '90,107-121, Brisbane, Australia,1990.

[13] M. Di Giacomo, G. Mainetto and L. Vinciotti, Gestione della persistenza e delle transazioni nel Sistema Galileo Distribuito, *Sistemi Evoluti per Basi di Dati*, Gizzeria Lido (CZ), I, 1993.

[14] J.V. Joseph, S. M. Thatte, C. W. Thompson and D. L. Wells, Object-Oriented Databases: Design and Implementation, *Proc. of IEEE, Vol. 79, No. 1*, 42-63, 1991.

[15] P. Hudak, A semantic model of reference counting and its abstraction, *Proc. of ACM Symposium on Lisp and Functional Programming*, 351-363, 1986.