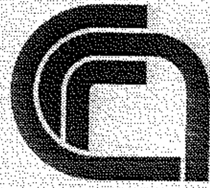

Consiglio Nazionale delle Ricerche

Conservative Multigranularity Locking for an Object Oriented Persistent Language via Abstract Interpretation

Giuseppe Amato, Fosca Giannotti, Gianni Mainetto

CNUCE: C 97 - 10

CNUCE

Conservative Multigranularity Locking for an Object–Oriented Persistent Language via Abstract Interpretation[1]

G. Amato, F. Giannotti, G. Mainetto

CNUCE–CNR, Via S. Maria 36, 56126 Pisa
e_mail: {fosca,gmsys}@cnuce.cnr.it

Abstract. *This paper presents an experiment of using a formal technique for static program analysis, based on Abstract Interpretation, in the context of an Object–Oriented Persistent Programming Language. Implicit transactions of APPL, a subset of Galileo, are statically analysed for inferring information to support a new concurrency control protocol that merges the wisdom of the Conservative Two-phase Locking with the flexibility of the Multiple Granularity Locking without requiring any predeclaration to transaction programmers. This is an improvement of a previous experiment that concentrates only on the Conservative Two-phase Locking, where the grain of the collected information was particularly significant in case of simply structured transactions. In this paper we are interested in rendering finer this grain while accessing objects of classes. More precisely, the paper presents a second analysis that detects if a transaction surely accesses all objects of a class or probably a subset of them. Collected information allows to choose the appropriate locking grain: a unique lock for the class versus a set of individual locks, one for each object of the class.*

1. INTRODUCTION

The evolution of database programming languages (DBPLs), which provide a unifying framework for data definition and manipulation, opens up opportunities for applying techniques used in the programming language area to solve problems occurring in the database area and vice versa. This exchange of experiences is more easily feasible for persistent programming languages (PPLs) because they are the subclass of DBPLs closer to programming languages. In PPLs the persistent value data model is completely integrated in a programming language [Atkinson 1987].

In the programming language area there is a particular emphasis on the use of formal techniques for guaranteeing the safety of optimisations and transformations performed from a compiler. The exploitation of formal techniques' application to PPL optimisation problems will lead to: a better understanding of such problems; the provisions of safe solutions of these problems, usually approached only on the basis of empirical experience; the addition of new functionalities to PPL systems.

A consolidated formal technique of static program analysis is *abstract interpretation* (AI) [Cousot 1977] [Abramsky 1987]. AI aims at statically gathering approximate information about program's dynamic semantics for the sake of compiler and run–time support optimisations.

We are experimenting AI in object–oriented (OO) PPLs. The goal of the experience presented in this paper is to support a new concurrency control protocol that merges the wisdom of the *conservative two-phase locking* (C2PL) with the flexibility of the *multiple granularity locking* (MGL) [Bernstein 1987]. Information for managing this new protocol, called *conservative multiple granularity locking*, is automatically inferred through AI technique from the text of an Abstract Persistent Programming Language (APPL) transaction. APPL is an OOPPL derived from Galileo [Albano 1985]. APPL

[1] This work was partly supported by the EEC under contract No ERB-CIPA-CT-93-1616, Special Project ANATRA and Bilateral Project SIENOSP of CNR Engeneering Comitee.

transactions are *implicit*, i.e. they do not require neither a predeclaration of the persistent values that will be accessed nor the explicit programming of transaction operations.

A first experiment was about C2PL protocol [Amato 1993]. In C2PL, transactions' operations can be safely interleaved when there is no overlapping among the sets of accessed data, namely *readsets* and *writesets*. Given the text of an APPL transaction and a representation of the data stored in the database, an abstract interpreter derived, in finite time, an approximation of its readset and writeset. To realise C2PL, the outcome of the interpreter was analysed from the scheduler before beginning the transaction execution.

The results of this first experiment confirmed the adequacy of the approach but in some cases the grain of the collected information was too coarse. MGL protocol addresses this problem. The shortcomings of the approach were essentially due to always considering a class as a single data item and to an insufficient attention to the peculiarities of APPL primitives. In APPL there are two different ways of visiting the database: the navigation, performed through the selection of objects' components representing associations, and the query. A navigation deals with one, two or several objects of a class and it always accesses the state of the involved objects; a query deals with all the objects of a class but it does not necessarily access their states. So we defined a new second analysis focused on those computations involving query primitives.

The main idea of this second analysis is to have an abstract semantics that expresses a notion of probability about the execution of constructs that compose APPL transactions. Intuitively, if an access primitive to an object state is applied in a branch of a conditional then its execution probability is half of the conditional execution's probability. The access probability to all the objects of a class is statically inferred considering all the execution paths involved in the evaluation of a query primitive. The access probability to all the objects of a class permits to statically choose the appropriate locking mechanism: a unique lock for the class versus a set of individual locks, one for each object.

The paper is organised as follows. Section 2 sketches the features of APPL language and system and provides an informal semantics of its primitives. Section 3 briefly describes the MGL protocol, its use in OO databases (DBs) and the conservative multiple granularity locking protocol; Section 4 describes the formal framework on which is based the static analysis; Section 5 presents the application of the analysis to few sample transactions and Section 6 contains few final remarks.

2. APPL LANGUAGE AND SYSTEM

2.1 APPL Language

APPL is the abstract syntax of a version of Galileo tailored to keep the most relevant database constructs provided by the original language [Albano 1985]. Galileo is a non-purely functional PPL with OO features. Classes with multiple inheritance (*is_a* relation) and objects are the data types used to model OO features.

A class is a "bulk" type constructor, like *set_of* or *list*, characterised by two facts: an *instance_of* relation exists between a class and the set of actual objects belonging to it (this set of objects is called the *extension* of the class); a subset relation exists between the extensions of two classes that are in an *is_a* relation. Classes have a *unique class identifier*.

Objects are instances of record types and every object has a *unique object identifier*. The same object can contemporary belong to several classes, those that are in an *is_a* relation. Equality on objects means *sameness*: two objects are equal if and only if they have the same object identifier. The identity of an object is used for modelling its

association with other objects. Objects can have values of any type as components, as for example functions used to model methods of classical OO languages.

Galileo is a unique programming language for the definition of the database schema and the manipulation of persistent values. The definition of the database schema is performed through a sequence of *top–level* declarations. Being a persistent programming language, Galileo allows to define at the top–level both classes and *individual objects* like integers, functions, records, etc. The manipulation of persistent values takes place through *implicit transactions*, i.e. top–level expressions in which transaction operations are implicitly defined. Galileo transactions use top–level declarations to solve their non local bindings.

APPL language is an expression language, where the term expression denotes also statements. In the paper we will use the following conventions:

$e \in$ Exp	generic expressions
$c \in$ Con	constants
$p \in$ Prim	primitives
$x \in$ Ide	identifiers
$lb \in$ Lab	labels of fields

Constants, primitives, identifiers and labels have different syntactical notations. In particular, primitives are reserved keywords and they will be indicated in bold italics.

APPL is the abstract syntax of a strongly statically typed language, so that its sentences are well–typed and operator overloading is resolved. For a given primitive the type of its operands is known. To improve readability of APPL sentences, the syntax of some expressions is denoted using metavariables that remind the type of the value the expression evaluates to:

$b \in$ Exp	boolean value
$f \in$ Exp	functional value
$l \in$ Exp	reference value
$o \in$ Exp	object value
$r \in$ Exp	record value

APPL language assumes that a user–defined function has a single argument, and only sequential declarations of (recursive) values are allowed.

An APPL transaction is a block primitive containing an optional sequence of declarations, that (recursively) bind an identifier to a value, followed by a sequence of expressions:

trans ::= ***block*** $(d_1, \ldots, d_n, e_1, \ldots, e_n)$ | ***block*** $(e_1, \ldots, e_n)$

d ::= ***let*** (x,e) | ***letrec*** (x,e)

An APPL expression can be a constant, an identifier, a user defined function application and the application of a language primitive:

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

In APPL there are primitives for flow control, block definitions and declarations. Other primitives are defined for each basic and constructed type that allow specific operations to be performed on values of such a type. Primitives defined on references are: reference constructor, dereferencing, assignment.

var (e)	creates and returns a reference to the value of e
at (l)	dereferences l i.e. returns the value referenced by l
ass (l, e)	assigns the value of e to the reference l

APPL primitives on classes are: insertion of an object in a base class; removal of an object from all the classes to which it belongs; specialisation of an object from a superclass to a subclass; query on all objects belonging to the extension of a class. [Ghelli 1990] demonstrates that this set of primitives is minimal that is it is sufficient to model all the expected features of classes in which objects can migrate from one class to another.

insert (x, r)	creates an object and inserts it in the class x
specialise (o, r, x)	specialises the object o with r and inserts o in the class x
remove (o)	removes the object o from all classes
for_class (x,f)	f function is applied to all objects of class x

The insertion of an object in a base class implies the construction of an object from the record value given as second parameter of the primitive. The insertion into a subclass is performed in several steps: firstly the object is inserted in a base class, then it is specialised from the base class down into the subclass, and so on. Specialisation adds and replaces the fields of the record given as second parameter to the object given as first parameter.

The query primitive *for_class* has two parameters: the extension of a class and a single argument function. It is semantically equivalent to the *mapcar* construct of Lisp [McCarthy 1962]: the function is successively applied to one object of the extension at a time, and the results are collected into a sequence. It is worth noting that the query primitive is more powerful than the traditional OO queries with associative retrievals and navigations because it allows to express a generic function to be evaluated on all objects of a class. This generic function can apply methods, update object state, etc. Due to its generality, the query primitive can not be optimised.

Classes, objects and references are the only modifiable values of the language, the ones on which side–effects are allowed. Other primitives that will appear in the rest of the paper are the followings:

same (o_1, o_2)	test for object equivalence
obj_of (o, lb)	selection of field lb from object o
ite (b , e_1, e_2)	if_then_else conditional primitive
func (x,e)	function constructor

2.2 APPL System Architecture and Operation

Our system adopts the client–server architecture, usual in OODBs [De Witt 1990] and used in several PPLs including Distributed Galileo [Di Giacomo 1993].

The server handles: the Shared Persistent Type Environment (SPTE), i.e. a mapping from global identifiers to type expressions, the Shared Persistent Value Environment (SPVE), i.e. a mapping from global identifiers to storage locations, and the Shared Persistent Store (SPS), i.e. a mapping from storage locations to storable values. Information in SPTE and SPVE constitutes the database schema; this information is frozen after being defined. SPS is the database and its state can change during the execution of APPL transactions.

SPTE, SPVE and SPS are initialised when the database designer initially inserts a sequence of top–level declarations into the centralised server. A component of APPL system analyses the initial declarations to produce the Database Representation (DBR).

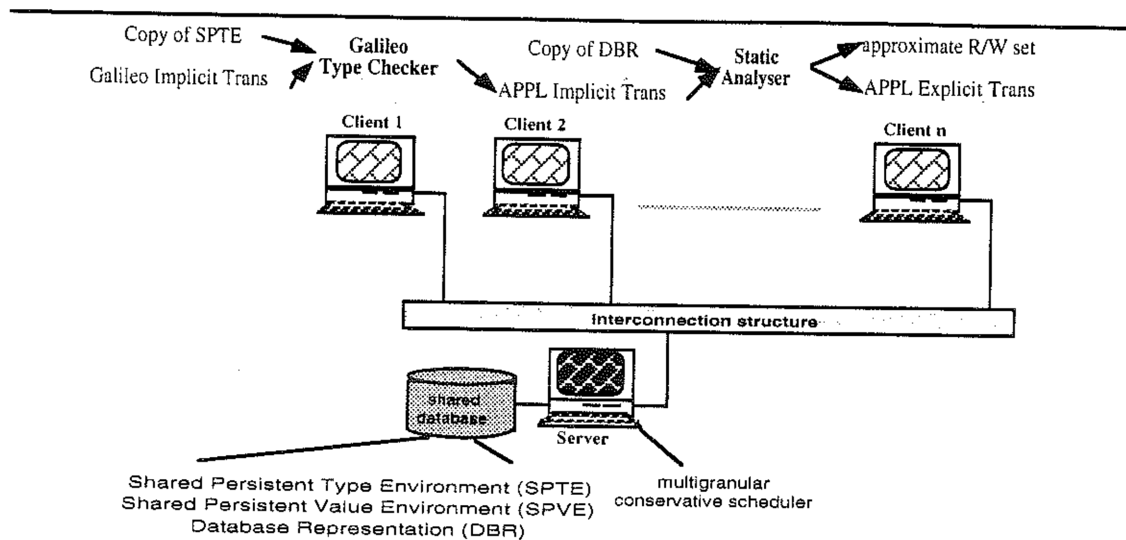

Figure 1: APPL System Architecture and Operation

DBR aims to summarise SPVE and SPS information in a format useful for an efficient static analysis of transactions.

A client has the task to compile, analyse and execute APPL transactions. When a client begins a session, it loads from the server a copy of SPTE for the sake of type checking, a copy of DBR for the static analysis, a copy SPVE for the generation of the intermediate code.

Once a transaction has been successfully type–checked, the client analyses the transaction's text and the DBR to deduce an approximation of the readset and writeset of the transaction. This information consists of lock modes on classes and modifiable individual objects. The analysis adds explicitly to APPL transaction's text the primitives for locking objects of classes when they are needed according to MGL protocol.

The scheduler of the server analyses the approximate information about transaction's readset and writeset: when all these database resources can be granted, it acknowledges the requesting client. Finally the transaction execution begins. During it communication network is used only to access shared persistent values and to request locks on objects of classes if they are needed. **Figure 1** summarises the architecture and the operation of APPL system.

3. MULTIGRANULARITY LOCKING IN APPL

3.1 Multiple Granularity Locking

MGL protocol has been firstly proposed in the context of relational DBs [Gray 1975]. The use of MGL aims to minimise the number of locks to set while a transaction accesses set of tuples of a relation. In this protocol, data items are organised in a tree where small items are nested within larger ones. Each nonleaf node represents the data associated with all its descendants; thus the root of the tree represents the whole database. Nodes of the tree are called *granules*. Transactions can lock nodes *explicitly*, which in turn locks descendants *implicitly*. Two kinds of locks are defined: *exclusive* and *shared*. An exclusive (X) lock excludes any other transaction from accessing (reading or writing) the

	NL	IS	IX	S	SIX	X
NL	√	√	√	√	√	√
IS	√	√	√	√	√	No
IX	√	√	√	No	No	No
S	√	√	No	√	No	No
SIX	√	√	No	No	No	No
X	√	No	No	No	No	No

Figure 2: Compatibility Matrix for Multiple Granularity Locking

node; a shared (S) lock permits other transactions to read the same node concurrently but prevents any updating of the node.

To grant a transaction a lock on a node, the scheduler would have to check if any other transaction has explicitly locked any ancestor of such a node. This is clearly inefficient. To solve this problem, a third kind of lock mode called *intention* lock was introduced [Gray 1978]. All the ancestors of a node must be locked in intention mode before an explicit lock can be put on the node. In particular, nodes can be locked in five different modes. A nonleaf node is locked in *intention–shared* (IS) mode to specify that descendant nodes will be explicitly locked in S mode. Similarly, an *intention–exclusive* (IX) lock implies that explicit locking is being done at a lower level in X mode. A *shared and intention–exclusive* (SIX) lock on a nonleaf node implies that the whole subtree rooted at the node is being locked in S mode and that explicit locking will be done at a lower level with X mode locks. A compatibility matrix for the five kinds of locks is shown in **Figure 2**, where *null* (NL) mode represents the absence of a request. The matrix is used to determine when to grant lock requests and when to deny them.

For a given *Lock Instance Graph* (LIG) G with a tree structure, [Gray 1975] defines the following MGL protocol based on the previous compatibility matrix:

(1) A transaction T can lock a non root node *n* of G in S or IS mode only if T holds an IX or IS lock on *n*'s parent (and by induction on all *n*'s ancestors).

(2) A transaction T can lock a non root node *n* of G in X, SIX or IX mode only if T holds an SIX or IX lock on *n*'s parent (and by induction on all *n*'s ancestors).

(3) To read (or write) the content of a node, a transaction T must own an S or X (or X) lock on some ancestor of the node (included the node itself).

(4) Locks should be released either at the end of the transaction (in any order) or in a leaf–to–root order. In particular, if locks are not held to the end of the transaction, the transaction should not hold a lock on a node after releasing the locks on its ancestors.

Notice that once a transaction acquires a node in S or X mode, no further explicit locking is required at lower levels and leaf nodes are never requested in intention modes since they have no descendants.

MGL protocol has been generalised to work for direct acyclic graphs (DAGs) of resources rather than simple trees [Gray 1978]. The key consideration is that to lock implicitly or explicitly a node of a DAG, a transaction should lock *all* the parents of the node and so by induction lock all the ancestors of the node. In particular, to lock a subgraph one must implicitly or explicitly lock all the ancestors of the subgraph in the appropriate mode.

In the example of *Lock Type Graph* (LTG) reported in **Figure 3**, to lock a record for updating, a transaction should obtain a IX lock on the database and on area, file, indices containing the record, and an X lock on the record itself. Alternatively, especially

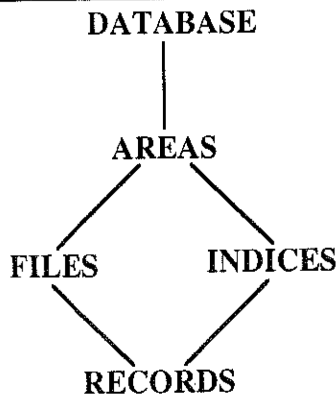

Figure 3: Lock Type Graph

when a transaction wants to update most of the records of a file, it could lock the database and the area in IX mode and the file and the indices containing the records in X mode, thereby implicitly locking records in exclusive mode.

The MGL protocol increases concurrency and decreases overhead. This is especially true when there is a combination of short transactions with a few accesses and transactions that last for a long time accessing a large number of objects such as audit transactions that access every item in the database.

3.2 Multiple Granularity Locking in OODBs

MGL protocol has been proposed and realised in ORION, an OODB with multiple inheritance [Garza 1988]. The motivation for using this protocol in the context of OODBs is that the OO data model provides a natural hierarchical organisation of data items in granules of different size: every object is a member of the set representing the extension of a class; a class can be a subclass of one or several superclasses, which implies that there is a subset relationship among the extension of a subclass and the extensions of its superclasses. In this context, when most of the objects belonging to the extension of a class are to be accessed, it makes sense to set one lock for the class, rather then one lock for each object.

In [Garza 1988] the hierarchies representing the is_a relationship and the instance_of relationship are inserted into the traditional LTG (**Figure 4**). Given an ORION database schema, a LIG will be a rooted DAG that directly connects the database root node to base class nodes (sources in the is_a hierarchy). Subclasses of base classes are represented as subnodes of base class nodes, and so on till to represent all the is_a hierarchy. Class nodes are connected to their extensions and extensions to objects thus representing the instance_of relationship. The LIG is a rooted DAG because in ORION there is multiple inheritance.

The passage from a tree to a rooted DAG implies the following changes to the first three rules previously defined:

(1) A transaction T can lock a non root node *n* of G in S or IS mode only if T holds an IX or IS lock on *some* parent of *n*.

(2) A transaction T can lock a non root node *n* of G in X, SIX or IX mode only if T holds an SIX or IX lock on *all* of *n*'s parents.

(3) To read the content of a node, a transaction T must own an S or X lock on some ancestor of the node (included the node itself). To write the content of a node, a

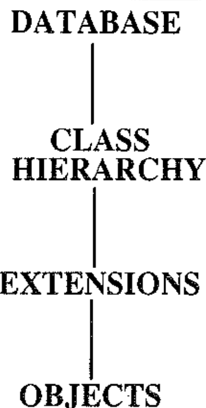

Figure 4: Object–Oriented Lock Type Graph

transaction T must own, for *every* path from the root to the node, an X lock on some ancestor of the node along that path (included the node itself).

In the ORION system there are two types of operations that involve the class hierarchy: schema change operations and queries. A schema change operation needs to lock in X mode a class definition, the definition of its subclasses, their extensions. A query on a class needs to lock in S mode the extensions of the class and of all its subclasses.

3.3 Conservative Multiple Granularity Locking in APPL

3.3.1 APPL LIG

The first step in the automatic support of a protocol that combines C2PL and MGL is to fix the organisation of persistent APPL values in terms of granules.

The LTG of a DB depends on the physical organisation of persistent data and on the operations allowed on them. In APPL every object of a class is represented by an extensible record with identity. Class extensions are represented as sets of object identities. The extensions are separate sets with no explicit subset relationship: if for example an object contemporary belongs to A and B classes then its identity is replicated in the extensions of A and B. In APPL the *for_class* primitive manipulates only all object identities of an extension, ***insert*** primitive modifies the last element of an extension, ***specialise*** modifies an object state and the last element of a (sub)extension, ***remove*** modifies an object state and an element of a set of extensions and ***obj_of*** operates on an object state. If we are able to provide an APPL LTG that allows to separate operations on class extensions from those on object states then our system can provide a fine resolution of the analysis. This is obtained organising the APPL LTG as shown in **Figure 5**. An APPL LTG has granules for class extensions containing smaller granules for extension elements and granules for class hierarchies (sets of object states) containing smaller granules for object states. In an APPL LIG a superclass S is directly related to its subclasses and to those object states having S as the most specific class. **Figure 5** shows also a simple example of APPL LIG (**nl-ee** stands for a null extension element).

The rationale for this separation is clarified by the following example. Let us suppose that there are two transactions T1 and T2: T1 uses a ***for_class*** primitive just to count the number of objects in A class, T2 specialises one object of A class into B subclass. T1 and T2 can execute in parallel because T1 reads all A extension (S lock on A extension) non interfering with T2 that modifies B's extension and specialises the object

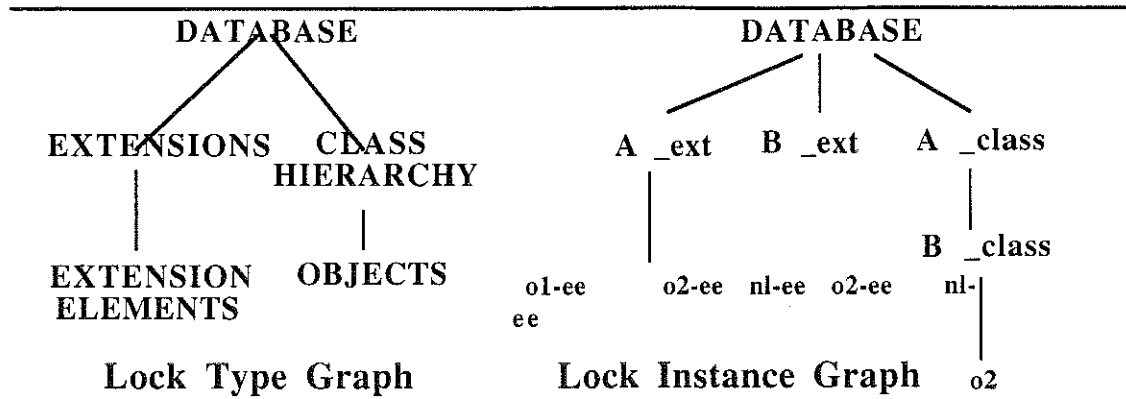

Figure 5: Lock Type Graph and a Lock Instance Graph for APPL

(X lock on B's null extension element and on the object of A). The reader can verify the correctness of the protocol on the LIG of **Figure 5** using the algorithm given in Subsection **3.1**.

Once we have clarified APPL LIG, it is clearer the aim of static analysis. The goal of the analysis is twofold: firstly to deduce an approximation of the readset and writeset expressed in terms of the leaves of APPL LTG; secondly to infer if and how the previous information on the leaves can be synthesised in coarser granules.

The first goal is accomplished through a *conservative locking analysis*, the second one through the *multigranular locking analysis*. The combination of the information gained by the two analyses produces the expected result, the *conservative multigranular locking analysis*.

3.3.2 Conservative locking analysis

The conservative locking analysis for APPL transactions is described in details in [Amato 1993]. In this subsection we give a short description of the technique used and of the obtained results.

A central point of AI is to guarantee the termination of the analysis. The termination of the conservative locking analysis relies on the fact that the abstract execution operates on finite abstract domains.

In our case, one problem is the DBR, i.e. the way in which SPVE and SPS are abstracted. It is necessary to render finite these two components so that the analysis terminates for sure, while keeping interesting information about a transaction execution. SPVE is frozen, so that it is trivial to abstract it. Every individual persistent object is represented by exactly one abstract individual object. The main problem is to make finite the abstract domains for the data structures whose cardinality cannot be statically determined: class extensions and sequences. A class extension is abstracted by a sequence of exactly two object identifiers: the first object identifier relates the class extension to one abstract object representing all the objects existing in the database *before* transaction execution, the second object identifier relates the class extension to the *new* objects potentially created by the transaction. In this way the analysis can distinguish new objects from old ones. Sequences are abstracted in the same way of class extensions.

The second problem to solve is how to render finite the number of new values generated from the analysed transaction. Indeed a real execution of a transaction could loop for ever and it could for example generate an infinite number of new persistent class objects. The solution is to bind statically the occurrence of every value constructor in a

transaction's text to just one abstract location. In this way every time a value constructor comes across during an abstract execution, it evaluates to the same abstract location and one abstract stored value plays the rôle of a set of concrete values.

The outlined analysis is sufficient to obtain information to realise a conservative locking protocol with a non fine resolution. This analysis provides satisfactory information about accesses to *individual objects*. As far as extension elements and object states are concerned, there are two deficiencies in this analysis both connected to the multigranular issue. The first is that class extension updates result in an X lock on all the extension because every class extension is treated as a single data item. The second shortcoming comes from the non distinction between navigational accesses to class objects and an access to class objects through a query primitive. These two kinds of accesses are both treated in the same manner, but while a query primitive deals with all the extension of a class and potentially with all its objects, a navigational access probably regards only a small subset of class objects. These two shortcomings are overcome by the multigranular locking analysis.

3.3.3 Multigranular locking analysis

The analysis described in the previous subsection gives us enough insight about navigation in the database and we consider this information sufficient to statically set *intention* locks on extensions and class hierarchy in the database's LIG. We a priori decide that navigation deals with a small subset of objects in an extension. To gather significant information about the objects reached through navigation, knowledge about the cardinality of the associations among objects, selectivity of predicates, etc. should be necessary. This kind of statistical information about the state of database is used for example by OO query language optimisers (see [Graefe 1993] for a comprehensive survey). The issue of verifying the usage of optimising techniques developed for query processing in languages as Galileo is out of the scope of this paper.

The multigranular locking analysis deals only with the query primitive.

The first previous shortcoming was about a class extension of APPL LIG. The query primitive is the unique primitive that deals with an extension as a whole. It is quite simple to statically determine a superset of class extensions that will be accessed during a transaction execution. Multigranular analysis infers this information just looking for every occurrence of a query primitive in the text of a transaction (and in the text of involved methods). These class extensions will be locked in S mode. A complete analysis that deals with class extensions is a simple variant of the analysis that we will present about class hierarchy.

The second shortcoming regards the class hierarchy of APPL LIG. The way in which we deal with this issue will be described and exemplified in the next two sections. Multigranular analysis provides information for setting shared, exclusive and shared intention exclusive locks on the class hierarchy of APPL LIG. This analysis will associate every node of the class hierarchy with an S, X, SIX, NL lock mode.

3.3.4 Conservative multigranular locking analysis

When the multigranular locking analysis finishes, firstly we combine intention locks given from conservative locking analysis with locks on APPL LIG class extensions. Then we combine intention locks with other locks on class hierarchy. This kind of information regards single nodes of the class hierarchy that represents a database schema with multiple inheritance: first S, SIX and X locks must propagate down in the hierarchy then IS, IX locks must propagate up in the hierarchy to statically set up information for a

	Extension	Ext_element	Class	Object
insert (*x*, *r*)	IX *x*	X nl_ee	NL	NL
specialise (*o*, *r*, *x*)	IX *x*	X nl_ee	IX *x*	X *o*
remove (*o*)	IX *o*'s exts	X *o*'s ee	IX *o*'s classes	X *o*
for_class (*x*, *f*)	S *x*	NL	NL	NL
obj_of (*o*, *lb*)	NL	NL	IS *o*'s classes	S*o*

Figure 6: Locks of Primitives in APPL Lock Instance Graph

DAG multigranular protocol. Finally the lock is determined for the root granule of APPL LIG.

The lock mode set from a transaction consisting of a single primitive is shown in **Figure 6**. When in the text of a transaction there is a combination of such primitives, some locks can be redundant and can be eliminated as for example those on extension elements and on objects. It is quite clear that given a primitive in a transaction and the set of locks on class extensions and class hierarchy derived from multigranular conservative analysis, it is always possible to statically decide if an access primitive must be explicitly preceded by a lock primitive or not. The strongly and static typing of Galileo ensures this. For example, it is always possible to foresee a set of classes to which the object operated on by a ***remove*** primitive belongs. Explicit lock primitives are added in the text of the transaction a final step of transformation resulting from analyses.

4. FORMAL FRAMEWORK

The static analysis of an APPL transaction for multigranular locking information is based on the formal framework of AI used in [Hudak 1987]. Firstly, an exact "non–standard" semantics of APPL transactions is defined. It behaves as the standard one, moreover it collects information on the dynamic behaviour of the program. An exact non-standard interpretation of an APPL transaction may not terminate as the standard one. Secondly, an abstract version of the exact non–standard semantics is defined, aimed to deduce information about how many objects of a class are manipulated and in which way. Finally, results of abstract semantics have to be proven correct wrt exact ones and that abstract semantics terminates for every possible input APPL transaction. The exact non-standard semantics is only sketched, a complete presentation may be found in in [Amato 1993. This semantics behaves exactly as the standard one, plus it keeps track of the accesses to modifiable values such as references, object states and classes. This is obtained associating two flags to each location in which a modifiable value is stored. During the non–standard execution of a transaction, flags are used to record if a location is accessed, respectively for reading or updating its content. Flags are set when read or update primitives are performed. Read and update primitives are specific for the type of value stored in the location. In our language the following cases hold:

i) A reference is read when a dereference primitive ***at*** is executed, and it is updated when an assignment ***ass*** is performed;

ii) An object state is read from an *object access primitive*: ***obj_of*** selects a field of an object, ***specialise*** updates an object;

iii) A class extension is read from a query primitive ***for_class***, a base class extension is updated from the insertion primitive ***insert***, a subclass extension is updated from the specialisation primitive ***specialise***.

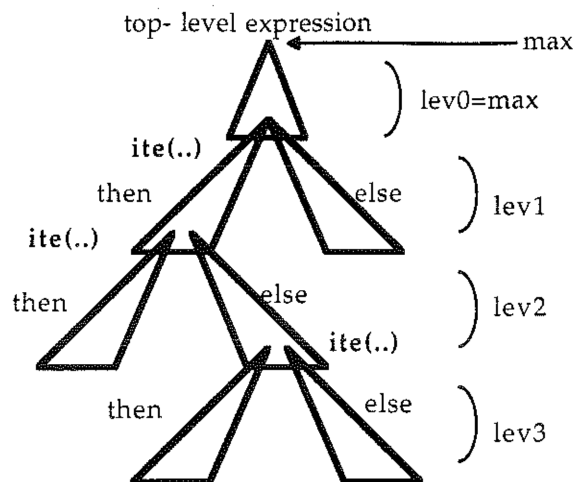

Figure 7: Levels in Abstract Execution

4.1 Abstract Semantics

The analysis we are interested in, is aimed to detect if a transaction accesses the state of every object of a class or only a subset of them. The abstract semantics formalizes the probability that a transaction accesses all objects of a class. This process builds on three concepts which are the basic items of our abstract domains:

i) *success probability*: it is the probability that the condition of an *if-then-else* statement holds. Such probability may be inferred by using statistical information on the database status.

As an example consider the following predicate *x.age > 35* , where x ranges over the class employee. Suppose that we have statical information on the distribution of the values of the attribute age of the employee. We can use such statistics to compute the the success (fail) probability of such predicate so if the 75% of employee are over 35, the success probability of the predicate is 0.75. When such statistics are not available success probability of the predicate is 0.5.

ii) *execution probability*: it is the probability that a statment will be executed. When a statment occurs within a branch of a *if-then-else,* its execution probability will depends on the success (fail) probability of the condition.

iii) *access probability*: it is the probability that all objects of a class are accessed. Notice that, there is only a primitive in our language which involves the extension of a class. Such primitive is the ***for_class*** which applies a function to all objects of a class. If such function contains an access primitive all objects of the class may (or may not) be accessed.

The above notions of probabilities are rephrased by the abstract semantics which manipulates abstract values ranging from 0 to *max*, where the constant max is a positive integer and it represents in terms of probability the value 1.

The *abstract execution probability* is computed according to the following rules:

i) let e be transaction: then its abstract execution probability is max;
ii) let e be the conditional expression $ite(b,e1,e2)$ with abstract execution probability aep, and let asp the abstract success probability of b: b will have as abstract execution probability aep, $e1$ will have as abstract execution probability aep div (max div asp) while $e2$ will have as abstract execution probability aep div (max div (max - asp)), where div is the integer division;
iii) let e be any other expression with execution probability aep: all its subexpressions will have the same execution probability aep

The *abstract access probability* is computed according to the following rules:
i) when an access primitive p is applied to the current object o during the execution of an expression reached from a ***for_class***, the *abstract access probability* of o is the *abstract execution probability* of p;
ii) when evaluating a conditional, the *abstract access probability* of all objects is the least upper bound between the sum of the *abstract access probabilities* collected from the else and then branches, and the *abstract access probability* of the condition.
iii) when any other expression is evaluated the *abstract access probability* of all objects is the least upper bound of the *abstract access probabilities* computed while executing the subexpressions.

4.2.1 Abstract Domains

The abstract domains used by the abstract semantics are defined as follows:

Env = Ide → Val + Class
Val = Aval + Aobj + Afun
Class = {1 .. *class_no*}
Aobj = {1 .. *class_no*}
St = Class → Aobj

Aval = {*none*}
Lock = Aobj → (X x S x SIX)
X = S = SIX = Aaprob
Aaprob= Aeprob = Asprob = {0, 1, 2, ..., *max* }
Fun = Val → St → Aeprob → (Val x Lock)

Basic values, references, records, object states and sequences are abstracted by the singleton {*none*} because we are not interested in the behaviour of a transaction on individual values and on associations among values. Our analysis is only interested in determining when an object is accessed during the abstract execution of ***for_class*** primitive. The extension of a class is denoted by a unique abstract object identifier. *class _no* is the number of classes in the database.

The Lock domain is used to record the *abstract access probability* of an object by associating an abstract object identifier with a triple <X, S, SIX>. The corresponding X, S, and SIX domains have 0 as bottom, *max* as top, and the ordering of the other elements is the usual natural ordering. We will indicate the bottom of the Lock domain with NL, i.e. NL=λo.<0, 0, 0>. The elements of the Aaprob, Aeprob, Asprob domains represent respectively *abstract access probability* , *abstract execution probability* , *abstract success probability*. Finally, the Fun domain is used for functions: given its argument, a store and a *abstract execution probability*, a function returns a value and a lock.

4.2.2 Tuning of the analysis

The accuracy of the probability computed by the information returned from the abstract analysis depends on the choice of *max*. The level at which expressions are evaluated decreases every time an occurrence of the conditional construct is evaluated, until the level becomes 0. Notice that, level 0 does not mean that there are no more sub-expression, but it means that our choice of *max* covers only till this depth of the execution tree. So, if there are further sub-expressions their execution level is approximated to 0. This behaviour affects also the computation of the access level and hence the access probability. So, all the object accessed by expression at level 0 will have access level 0 that represents no access. This is in the case where our analysis may give a worst approximation with the respect the real execution. The consequence of this behaviour affects only the performance (choosing an object locking instead a class locking) while correctness is preserved.

The more accurate choice of *max* should provide that the level of the nested conditionals that can be reached during execution should be greater than 0. This value is not computable statically because of recursion. An acceptable choice of *max* can be made for example considering the depth of a tree built as follow:

i) we start from a node for the main expression *e* representing the transaction to be analysed;

ii) if the node corresponding to an expression contains a conditional construct, the node has owns two children, corresponding to *then* and *else* expression;

iii) if an expression *e* contains a function call *g(x)*, *g(x)* is syntactically substituted with its body, if the substitution of *g(x)* has not been made previous in the path starting from the root of the tree and reaching *e* (this is the way we cut the recursion).

4.2.3 Semantic equations

In this subsection, we will consider the semantic functions AL for expressions and AD for local declarations.

AL: Exp → Env → St → Aeprob → (Val x Lock)
AD: Dec → Env → St → Aeprob → (Val x Lock x Env)
AP: Exp → Env → St → (Asprob)

AL and **AD** evaluate respectively an expression and a declaration in a given environment, a store and a *abstract execution probability*. **AL** returns a value and a lock, **AD** returns a value, a lock and a modified environment. **AP** returns the *abstract success probability* of a conditional expression.

Within the equation the convenctions $\rho \in$ Env, $\sigma \in$ St, $\pi \in$ Aeprob will be used.

The description of **AP** function is out of the aim of the paper. It may be depend by statistical information on the database status and it may be computed by same other analysis tool. In this sense we just say that:

AP $[e]\ \rho\ \sigma = oracle(e, \rho, \sigma)$,

where

oracle: (Exp, Env, St) → Asprob

is the function that uses the statistical information.

In the sequel of the section we present only few relevant semantic equations of our abstract semantics.

$$
\begin{array}{lll}
\mathbf{AL}\ [ite\ (b, e_1, e_2)\ ]\ \rho\ \sigma\ \pi = \mathbf{let} & & p = \mathbf{AP}\ [b]\ \rho\ \sigma \\
 & & \langle val_1, lk_1\rangle = \mathbf{AL}\ [b]\ \rho\ \sigma\ \pi \\
 & & \langle val_2, lk_2\rangle = \mathbf{AL}\ [e_1]\ \rho\ \sigma\ (\pi \text{ div } (max \text{ div } p)) \\
 & & \langle val_3, lk_3\rangle = \mathbf{AL}\ [e_2\ ]\ \rho\ \sigma\ (\pi \text{ div } (max \text{ div } (max - p))) \\
 & & lk_4 = asum\ (lk_2, lk_3) \\
 & \mathbf{in} & \langle none, sup\ (lk_1, lk_4)\ \rangle
\end{array}
$$

The boolean expression b is evaluated at the execution probability of the conditional construct itself. The *then* and the *else* branches are evaluated at the appropriate execution probability as described previously. The global result is the sup between the sum[2] of the locks resulting by the evaluation of the two branches and those locks resulting by the boolean expression evaluation.

$$
\begin{array}{lll}
\mathbf{AL}\ [for_class\ (x, f)\ ]\ \rho\ \sigma\ \pi = & \mathbf{let} & \langle fun, lk_1\rangle = \mathbf{AL}\ [f]\ \rho\ \sigma\ \pi \\
 & & cid = \rho\ (x) \\
 & & oid = \sigma\ (cid) \\
 & & \langle val, lk_2\rangle = fun\ oid\ \sigma\ \pi \\
 & \mathbf{in} & \langle none, sup(lk_1, lk_2)\ \rangle
\end{array}
$$

for_class primitive applies a function f to the unique class object bound to the class identifier x. The lock of the object changes only if it is accessed during the function application.

$$
\begin{array}{lll}
\mathbf{AL}[specialise\ (o,r,x)]\ \rho\ \sigma\ \pi = & \mathbf{let} & \langle oid, lk_1\rangle = \mathbf{AL}\ [o]\ \rho\ \sigma\ \pi \\
 & & \langle -, lk_2\rangle = \mathbf{AL}\ [r]\ \rho\ \sigma\ \pi \\
 & & lk = \mathbf{if}\ (oid\ \pi\ none) \\
 & & \qquad \mathbf{then}\ [\langle \pi, 0, \pi\rangle / oid] \\
 & & \qquad \mathbf{else}\ \mathrm{NL} \\
 & \mathbf{in} & \langle oid, sup\ (lk_1, lk_2, lk)\ \rangle
\end{array}
$$

specialise primitive modifies an object so its lock is updated in X and SIX mode. Notice that o expression evaluates to *none* if the object has not been reached through a *for_class* primitive: in this case π (execution probability), is not recorded.

$$
\begin{array}{lll}
\mathbf{AL}[obj_of\ (o\ , lb\ )\ ]\ \rho\ \sigma\ \pi = & \mathbf{let} & \langle oid, lk_1\rangle = \mathbf{AL}\ [o\ ]\ \rho\ \sigma\ \pi \\
 & & lk = \mathbf{if}\ (oid\ \pi\ none) \\
 & & \qquad \mathbf{then}\ [\langle 0, \pi, \pi\rangle / oid] \\
 & & \qquad \mathbf{else}\ \mathrm{NL} \\
 & \mathbf{in} & \langle none, sup\ (lk_1, lk)\ \rangle
\end{array}
$$

The **obj_of** primitive reads the object so the object lock is updated in S and SIX mode.

---

[2] The auxiliary function *asum*: Lock x Lock → Lock is defined as follows:

$asum\ (lk_1, lk_2) = [sm/oid]$

where $\forall\ oid \in \mathrm{Dom(Lock)}$

$sm = \langle lk_1(oid)\downarrow 1 +^a lk_2(oid)\downarrow 1, lk_1(oid)\downarrow 2 +^a lk_2(oid)\downarrow 2, lk_1(oid)\downarrow 3 +^a lk_2(oid)\downarrow 3\rangle$

and $+^a$:(X+S+SIX) x (X+S+SIX) → (X+S+SIX) is defined as follows:

$$
\begin{array}{lll}
a +^a b & = a + b & \mathbf{if}\ a + b \le max \\
 & = max & \mathbf{elsewhere}
\end{array}
$$

$$\mathbf{AL}[\textit{remove}\ (o\ )\ ]\ \rho\ \sigma\ \pi = \quad \textbf{let} \quad \langle oid, lk_1\rangle = \mathbf{AL}\ [o\ ]\ \rho\ \sigma\ \pi$$
$$lk = \textbf{if}\ (oid\pi\ none)\ \textbf{then}\ [\langle\pi,0,\pi\rangle / oid]\ \textbf{else}\ \mathrm{NL}$$
$$\textbf{in} \quad \langle none, sup\ (lk_1, lk)\ \rangle$$

The ***remove*** primitive modifies the object lock in X and SIX modes.

4.2.4 Correctness and termination

The correctness of the analysis has been proven with respect to the exact non-standard semantics of the language. The intuitive notion of correctness is the following: if after the analysis of a transaction an abstract object o_a contains *max* in X (S) then all the objects of the exact class, that corresponds to the abstract class to which o_a belongs to, will be accessed for updating (reading); if after the analysis of a transaction an abstract object o_a contains *max* in SIX then all the objects of the exact class, that corresponds to the abstract class to which o_a belongs to, will be indifferently accessed for updating or reading.

The termination of the analysis is guaranteed by the finiteness of the abstract domains and by the monotonicity of the functions that perform the analysis on such domains. In fact the termination of the fix–point computation depends on the monotonicity of the function involved in the computation and on the usage of domains with finite ascending chains [Hermenegildo 1990]. The finiteness of the domains is trivially stronger than having finite ascending chains.

5. EXAMPLES OF THE APPROACH

The analysis starts evaluating a transaction with *max* as initial level and with a DBR as previously defined (Section **4.2.2**). The goal of transaction's analysis is to know for each class the approximate access probability to all the objects of its extension in S, X and SIX mode. The table below reports the combination of the admissible results of the analysis and the final lock decisions. For example, suppose that the analysis results in a *max* value for the X component of *cl* class's abstract object. Independently from the values of the other two components, we argue that all the objects of *cl* will be dynamically modified from the transaction so we ask an X lock for *cl* class to lock implicitly all its objects (first column of the table).

X	*max*	*0*	>0,<*max*	*0*	>0,<*max*	<*max*
S	–	*max*	<*max*	>0,<*max*	*max*	<*max*
SIX	–	–	<*max*	–	–	*max*
Final Lock	X	S	IX	IS	SIX	SIX

The next three examples have the following Galileo database schema consisting of the Persons superclass and its subclass Employees.

```
use   Persons class
          person <->
               (| name: string;
               and code: string
               and address: string ... |)
```

Lock	
obj1	<0,0,0>
obj2	<0,0,0>

Lock	
obj1	<0,0,0>
obj2	<0,*max*,*max*>

Lock	
obj1	<0,0,0>
obj2	<*max*/2,*max*,*max*>

Figure 8: Analysis' Results of T1, T2, T3.

```
and   Employees subset of Persons class
               employee <->
                    (| is person
                    and salary: var int ... |);
```

The resulting DBR has two abstract classes with a single abstract object.

Env		St_C	
Persons	cl1	cl1	obj1
Employees	cl2	cl2	obj2

Example T1

T1 is a transaction that counts the objects of Employees class.

```
block ( let (Counter, var (0)),
        let (Count,func(Obj,ass (Counter,+ (at (Counter),1)))),
                 for_class (Employees,Count),
                 Counter
      )
```

T1 does not access any object state. This means that the class hierarchy in APPL LIG is not involved. The occurrence of the *for_class* primitive in the text of the transaction will only produce an S lock on Employees extension of APPL LIG.

Example T2

T2 increases the salary of all the employees except the one bound to Eobj identifier. T2 returns the sequence of the salary of all the employees.

```
block ( let (Pay,func(Obj,
                      ite    (same (Obj,Eobj),
                              at (obj_of (Obj, 'salary')),
                              block ( let (Sal,obj_of (Obj, 'salary')),
                                          ass (Sal,+ (at (Sal),const)),
                                          at (Sal))))
             )
        for_class (Employees,Pay)
      )
```

Notice that all the objects are accessed in shared mode. This leads to ask an S lock on the Employees class.

Example T3

T3 looks for one person object and specialises it.

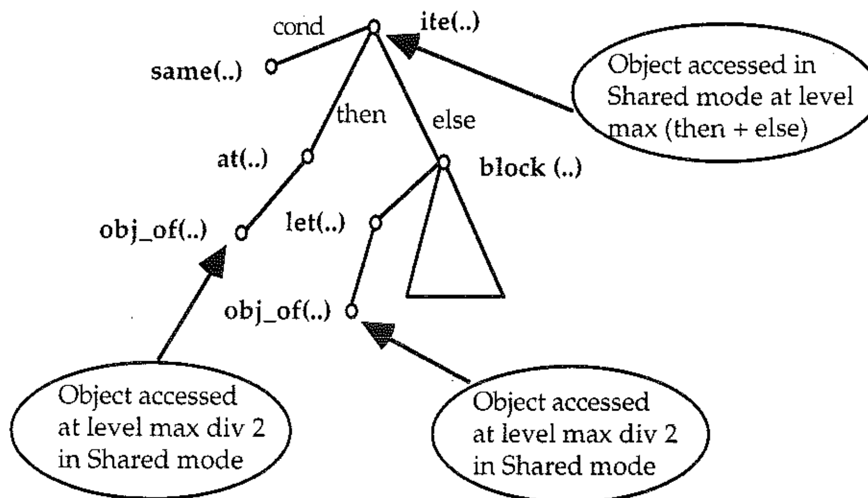

Figure 9: Tree of Abstract Execution of Example T2

```
block ( let (Upd,function (Obj,
                    ite     (same (obj_of (Obj,Cod),487502),
                             specialise (Obj,est,Employees),
                             skip ))
              ),
      for_class (Persons,Upd))
```

The transaction accesses in S mode all persons and it accesses in X mode a single person. This leads to ask a SIX lock for the Persons class.

6. FINAL REMARKS

The paper has presented an approach for dealing with conservative multigranular locking in a PPL with OO features inspired by Galileo. There are several firm believes that motivate a static analysis of the implicit transactions expressed in this language:

a) the programmer is relieved from the burden of explicitly programming a complex concurrency control protocol;

b) the supporting system knows that the concurrency control protocol has been automatically generated and it is safe, so the system is relieved from the task of controlling the correctness of the implemented concurrency control protocol;

c) the overhead of the static analysis in a client–server architecture can only burden on a client;

d) the use of conservative multigranular locking protocol reduces the run–time overhead of the supporting system while allows a satisfactory concurrency degree.

The result is a programmable system that as a whole represents a compromise between a high level of abstraction during its programming and a satisfactory efficiency at run–time. The efficiency is pursued reducing at a minimum the functionalities provided at run–time by the scheduler in the server, increasing the set of functionalities provided by a client (static analysis and part of the concurrency control functionalities at run–time), transferring in a phase that precedes transaction execution the granting of access to shared resources (conservative protocol).

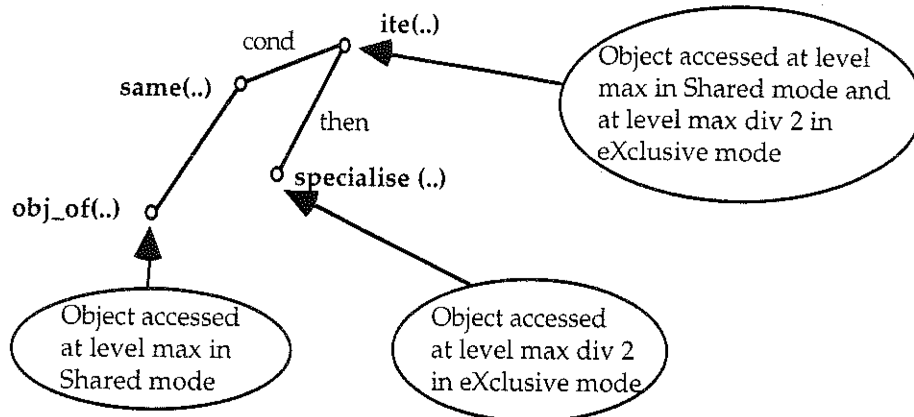

Figure 10: Tree of Abstract Execution of Example T3

The proposed approach focuses on reducing the granularity of the analysis but still the best results are for transactions with a simple structure of the control. We have done a step forward concerning the previous experiment but there is space for further improvements. To provide a better accuracy of the analysis the use of techniques developed in the context of query processing and symbolic execution might be investigated. We strongly believe that abstract interpretation is a unifying framework for all these kinds of techniques.

At the moment a prototype of the proposed system is under development. In this prototype the analyser is integrated with the existing APPL system, so that the analysis, the scheduling and the execution of transactions coexist and interact. A further step is to test some significant case studies to evaluate the performance figures obtained.

7. REFERENCES

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

[Albano 1985]
Albano A., L. Cardelli and R. Orsini, "Galileo: A strongly typed interactive conceptual language", *ACM Transactions on Database Systems, Vol. 10, N. 2*, pp. 230-260.

[Amato 1993]
Amato G., F. Giannotti and G. Mainetto, "Data Sharing Analysis for a Database Programming Language via Abstract Interpretation", *Proc. of the 19th Int. Conf. onVLDB*, Dublin, Ireland, pp. 405–415.

[Atkinson 1987]
Atkinson M. P. and O. P. Buneman, "Types and Persistence in Database Programming Languages", *ACM Computer Surveys, Vol 19, N. 2*, pp. 105–190.

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

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

[De Witt 1990]
D. J. DeWitt, P. Futtersack, D. Maier and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems", *Proc. of the 16th Int. Conf. on Very Large Database Conference*, Brisbane, Australia, pp. 107–121.

[Di Giacomo 1993]
Di Giacomo M., G. Mainetto and L. Vinciotti, "Gestione della Persistenza e delle Transazioni nel Sistema Galileo Distribuito", *Atti del Primo Convegno Nazionale su Sistemi Evoluti per Basi di Dati*, D. Saccà (editor), pp. 227–243, an extended version is contained in *CNUCE Tech. Report C93–05*.

[Garza 1988]
Garza J. F. and W. Kim, "Transaction Management in an Object–Oriented Database System", *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Chicago, Illinois, pp. 37–45.

[Ghelli 1990]
Ghelli G., "A Class Abstraction for a Hierarchical Type System", *Proc. of the 3rd Int. Conf. on Database Theory*, Paris, S. Abiteboul (ed.), Springer–Verlag, LNCS 470, pp. 56–71.

[Graefe 1993]
Graefe G., "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys, Vol. 25, N. 2*, pp. 71-170.

[Gray 1975]
Gray J., R. Lorie and G. Putzolu, "Granularity of locks and degrees of consistency in a shared database", *IBM Res. Rep. RJ1654*, IBM Research Laboratory, San Jose, CA also in *Modeling in Database Management Systems*, Nijssen (ed.), North Holland, 1976.

[Gray 1978]
Gray J., "Notes on Database Operating System", *IBM Res. Rep. RJ2188*, IBM Research Laboratory, San Jose, CA also in *Operating Systems – An Advanced Course*, R. Boyer, R. M. Graham and G. Siegmüller (eds.), Springer Verlag, LNCS 60, 1978.

[Heremenegildo 1990]
Heremenegildo M., "Abstract Interpretation and Its Applications", Invited Talk, *Advanced School on Foundations of LP*, Alghero, Italy.

[Hudak 1987]
Hudak P., "A semantic model of reference counting and its abstraction", in *Abstract interpretation of declarative languages*, Abramsky S. and C. Hankin (eds), pp. 45–62.

[McCarthy 1962]
McCarthy J., P.W. Abrahams, D.J. Edwards, T.P. Hart and M.I. Levin, *Lisp 1.5 programmers's manual*, MIT Press.