Controlling concurrent accesses in an object-oriented environment

CARMEL MALTA, JOSÉ MARTINEZ

Université des Sciences et Techniques du Languedoc
Centre de Recherche en Informatique de Montpellier (ura CNRS 815)
860, rue de Saint-Priest, 34090 Montpellier, FRANCE
tél.: (33) 67 63 04 60, Email: <malta, martinez@crim.fr

ABSTRACT: The purpose of this paper is to propose a way of controlling concurrent accesses in an object-oriented environment. The object-oriented approach places specific constraints on data model and on accesses, which we discuss. We then propose a lock-based protocol. Its qualities are: an unlimited range of access modes and the automatic creation of the corresponding locks; the possibility of hierarchical accesses; the ability to deal with the well-known problem of “phantom objects”, and the similar problem of modifying the inheritance graph.

KEYWORDS: concurrency control, object-oriented environment, inheritance graph, locking.

1. Introduction

Current achievements in the field of object-oriented systems are numerous: IRIS [Fishman et al. 87], GemStone [Maier et al. 86] [Penney & Stein 87], ORION [Banerjee et al. 87], O2 [Lécluse & Richard 89], Trellis/Owl [Schaffert et al. 86], etc. These systems manage data, the structures of which are more complex than those of the relational model. The relational model offers fixed structures, tuples, and a few predefined operations: select, project, join. Object-oriented systems go a step further. They give the user the ability to define new data structures and the operations which manipulate them.

The use of these systems in a multi-user environment immediately raises the concurrency control problem. The richness of this new model allows one to perceive various access modes and to enable potential parallelism.

We offer in this paper a technique for controlling concurrent accesses in an object-oriented system. As the model permits the definition of operations, our proposition allows the extension of the authorized access modes. It then derives automatically the necessary elements of the control. Unfortunately, space limitation inflicts us not to compare our technique with the other single proposition that we know of: the protocol of ORION [Garza & Kim 88].

This paper is organized as follows: In section 2, we introduce a non-formal and fairly general model of objects. We also present the associated access modes. Section 3 resumes the classical transactional model. Concurrency control by locking is decomposed in three parts: access types to control the set of objects which can be used by a transaction; access modes to determine in which way are used the objects; intentional access to use all or part of the objects. Our protocol is then proposed and finally extended to special cases of interest.

2. Object model

Objects and the operations which manipulate them are closely related in an object-oriented environment. In fact, the creation of data and operations takes place at the same time. Around this concept, numerous models have been developed: abstract data types stemming from computer engineering, frames from AI, etc. We are interested in class-based environments with multiple inheritance, which prototype is Smalltalk [Goldberg & Robson 83], without keeping the notion of meta-classes. O2 and ORION are two such systems.

2.1. Data

2.1.1. Classes

A class is an abstract data type. It has properties which describe a data structure or type, and defines operations linked to that type.

The main characteristic of the type is complete invisibility outside of the class; this is called encapsulation.
Among the operations, only some ones are visible from the outside of the class; they constitute the interface, determine the semantics of the objects, and are activated via messages.

2.1.2. Inheritance graph
Classes are related by a partial order relation. This relation is represented by an acyclic rooted graph: the inheritance graph. If the graph is a tree, one speaks of single inheritance; if it is a lattice, one speaks of multiple inheritance. When two classes, say cl and c2, are related to each other, cl is called the super-class of c2 and c2 the sub-class of cl.

The inheritance relation leads to the increase of the properties of a class by adding the properties of its super-classes. The data structure of a class is composed of the aggregate of its own type definition and of those of its super-classes. It comes out a sub-typing relation, isomorphic to the inheritance relation [Lecluse et al. 88]. The class inherits also the set of methods usable by its super-classes, and defines, (itself), new methods to manipulate its data structure.

2.1.3. Instances
Classes are not only abstract data types but also collections of lower level objects: instances. Instances are in fact the real objects of the system. Classes often simulate the behaviour of an object, such as the possibility of answering a restricted and predefined set of messages. The typical example is the "new" message that is sent to a class and therefore might be defined in a meta-class.

An instance is a variable of the type defined in a class. We then refer to it as a proper instance of the class. Conversely, we speak about the proper class of an instance.

2.1.4. Subgraphs
An instance is also of a sub-type of each super-class of its proper class, and, more generally, of any ancestor class of its proper class. This means that an instance can be manipulated by the methods defined in any of its ancestor classes, being thus indistinguishable from the proper instances of these classes. Consequently, we introduce the notion of general instances of a class, to denote the union of the proper instances of the class, and of the general instances of its sub-classes.

The notion of a subgraph made up of a class and of all its descendant classes in the inheritance graph, and the notion of general instances are directly related. Thereafter, we are only interested in that kind of subgraphs, and use the term "subgraph" with restricted meaning. The general instances of a class are the instances of the associated subgraph of that class.

2.2. Object accesses
The activation of a method is made by sending a message. This call is composed of the name of the object, the name of the method and its parameters.

Different accesses can naturally be considered:
- to an instance;
- to a class, (more precisely to its properties);
- to the classes of a subgraph;
- to the classes of a subgraph, and simultaneously to their instances.

access to an instance
It is the privileged access used by messages.

access to a class
Reading access to a class is needed when creating an instance, to instantiate the type, as well as during the creation of a sub-class, to inherit the properties.

access to the classes of a subgraph and their instances
Accessing class C to modify its definition implies write accesses on all the classes of the subgraph rooted to C, but also on all the general instances of that subgraph.

access to the classes of a subgraph
On the other hand, modifying the methods of a class just modifies the classes of the subgraph.

From an implementation point of view, the system must access to meta-data such as pointers, internal tables, indices, etc. These meta-data can be treated separately or integrated into the data of the system, especially into classes.

3. Concurrent-access control
Concurrent-access control is one of the main element of transactions [Gray 78]. Generally, concurrency control criterion is serializability: the concurrent execution of a set of transactions must be equivalent to a serial execution. To ensure serializability, we rely on locking, and more precisely on two-phase locking [Esowan et al. 76].

The objects which will be locked are classes and instances. To deal with all the accesses listed above, objects will not be treated independently from each other. Accesses are decomposed into access types and access modes. Access types select the objects which may be used. Access modes indicate which operations are made on the selected objects.

3.1. Access types
The inclusion semantics of the inheritance relationship implies that the inheritance graph can be used to determine the subsets of selected objects. We only use two kinds of subsets: either a class and its proper instances, or a subgraph and the general instances of its root.

3.1.1. Locking in an inheritance graph
Being a logical object, the locking of a subgraph depends on the locking of some classes. We present two properties which determine the classes that have to be involved to "totally" lock a subgraph.

definition 1
You might recall that we associate to each class the set of all its descendant classes and that we call this set a subgraph (See section 2.1.4.). Here we note that we also associate the instances to the classes in the definition of a subgraph.

definition 2
The frontier of a subgraph is the set of classes belonging to that subgraph and having at least a super-class not in that subgraph (Figure 1).
In addition, the root class of any subgraph is a frontier class, (which is always verified except for the root of the inheritance graph).

We dwell upon the fact that a class is a frontier class only with respect to another class, and not in an absolute way.

Subgraphs are not independent objects as classes and instances are. Two subgraphs may share all or parts of their classes. More precisely, a subgraph may be entirely included in another one; with multiple inheritance, two subgraphs may have a strict intersection, i.e., a non empty one which is not one of the two subgraphs.

The notion of frontier classes allows us to produce a property for the strict intersection, and another for the inclusion.

**Property 1**
If two subgraphs have a strict intersection, there exists at least one common class which is frontier class of each subgraph (Figure 2).

**Definition 3**
We define a path to a subgraph G1 as being any path going from the root of the inheritance graph, (noted r), to any class of G1.

We call path class of G1 each class situated in one path to G1 before the first frontier class of G1, (therefore, the frontier class is not a path class).

**Property 2**
If two subgraphs, G1 and G2, are such that G1 is included in G2, then on each path from r to any class of G1, there is at least one class which is frontier class of G2 (Figure 3).

**3.1.2. Lock types**
To lock a subgraph, i.e., its classes and instances, we lock the particular classes highlighted by the two properties of the previous section. These classes have not the same role. We distinguish:
- frontier classes on which we use frontier locks;
- isolated classes on which we use class locks;
- path classes on which we use path locks.

We introduce a compatibility relation for lock types:

\[ C_T : \text{TYPES} \times \text{TYPES} \rightarrow \{\text{true}, \text{false}\} \]

where TYPES = \{F, C, P\} with an obvious meaning.

The extension of C_T is given in Table 1. Two lock types are incompatible if the intersection of their selected objects is not empty.

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>C</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>C</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**Table 1:** lock type compatibility relation \( C_T \)

**3.2. Access modes**
[Korth 83] showed that it is desirable to associate an access mode to each operation. Conversely, an access mode represents a set of operations; for instance, access mode W, used to write an instance, groups the set of all the different operations which write an instance. Thus, an access mode has a semantics which is logically implied by that of each operation that it includes.

All operations cannot be done concurrently without possibly violating serializability. It is the same for access modes. The restrictions are represented by another compatibility relation:

\[ C_M : \text{MODES} \times \text{MODES} \rightarrow \{\text{true}, \text{false}\} \]

Generally, access modes are compatible when transactions only read common objects. However, compatibility can be defined with less restricted conditions, namely as being unnecessary to control the concurrent execution within the operations [Cart & Ferrié 90].
Lack of space imposes us not to develop this point. For that reason, we only define “classical” accesses: R and W access modes on instances, which respectively synthesize reading and writing instances; RD on classes, which allows the reading of the properties; and MD, which permits the modification of the data structure of a class and consequently of the subgraph, the root of which it is. The compatibility relation is given in Table 2.

<table>
<thead>
<tr>
<th>R</th>
<th>W</th>
<th>RD</th>
<th>MD</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>W</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>RD</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>MD</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 2: access mode compatibility relation.
(for some modes)

### 3.3. Implicit, intentional and hierarchical locking

We saw (Section 3.1.) that locking a subgraph is useful and locking a class is only a subcase. However, we still have to deal with locking instances. That is tricky due to different access modes which use either set of instances, (as MD), or just one instance, (as W).

In order to minimize the number of locks that a transaction must obtain when accessing to a subgraph, (in mode MD, for example), locking that subgraph, (i.e. path and frontier classes), should imply the implicit locking of all the instances of the subgraph.

Then, we must be able to detect an incompatibility with another transaction which wants to access to one instance of that subgraph in mode W, for example. The second transaction must lock the instance, and also the proper class of this instance in intentional mode W. Thus, using implicit locking implies using intentional locking.

It is also possible that W access should be repeated on all the instances of a class or a subgraph, (or a majority of them). In this case, it is interesting to avoid locking explicitly and successively each instance. We then lock the class or the subgraph in hierarchical mode W. In point of fact, it is a hierarchical locking limited to two levels: instances and set of instances (See IS and S, IX and X in [Gray 781]).

### 3.4. Locks

We distinguish two kinds of locks: class locks and instance locks. Locks are constructed. To simplify writing and reading, we give them names.

#### 3.4.1. Lock construction

A class lock \( cl = (t,m,i) \) is a component of:

\[
CL: \text{TYPES} \times \text{MODES} \times \text{INTENTIONS}
\]

where \( \text{INTENTIONS} = \{ \text{true}, \text{false} \} \).

The lock compatibility function is defined as follows:

\[
\text{CL} \times \text{CL} \rightarrow \{ \text{true}, \text{false} \}
\]

The lock type is used in order to select the subset of the objects which may be used by an operation.

The access-mode semantics indicates which are the selected objects actually used, (classes and/or instances), and in which manner they are used.

If an access mode implies that all the selected instances are used, then the value of the intentional component must be “false”. Conversely, if the access mode does not imply the use of all the selected instances, then “true” indicates intentional access and further locking on the actually used instances, “false” indicates hierarchical and implicit locking of all the instances.

The disjunction expresses the independence of the three components. For two locks to be compatible, it is sufficient either that their transactions do not access to common objects, or that their access modes be compatible, or that their common instances be locked explicitly by both transactions.

With the data model that we present in this paper, a lock on an instance is limited to \( IL = \text{MODES} \).

#### 3.4.2. Lock names

To simplify the writing, we give to class locks names representing “graphically” the meaning of each component: \( \text{TYPE, INTENTION, MODE} \). The three lock types, Path, Frontier, and Class, are respectively represented by the symbols \( P \), \( V \), and \( E \) (the empty string). Intention, “true” or “false”, is represented by \( \exists \) and \( \forall \). The ambiguity due to using twice the symbol \( \forall \) will be removed by the mode. Each mode is directly represented by its name.

<table>
<thead>
<tr>
<th>Intention</th>
<th>Path</th>
<th>Frontier</th>
<th>Class</th>
<th>Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>( \downarrow \exists W )</td>
<td>( \downarrow \forall W )</td>
<td>( \forall W )</td>
<td>W</td>
</tr>
<tr>
<td>false</td>
<td>( \downarrow \exists R )</td>
<td>( \downarrow \forall R )</td>
<td>( \forall R )</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>( \downarrow \exists RD )</td>
<td>( \forall RD )</td>
<td>RD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \downarrow \exists MD )</td>
<td>MD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: locks associated to the access modes of Table 2

The lock names are obtained by concatenation of the type symbol, possibly the intention symbol, and the mode name. The intention symbol is only used with the modes which apply to an instance, (i.e., for ours, R and W).

With the instance mode W, we construct six class locks: \( \downarrow \exists W, \downarrow \forall W, \forall \exists W, \forall \forall W, \exists W, \) and \( \forall W \). With the class mode RD, we obtain three class locks: \( \downarrow RD, \forall RD \), and RD. From the subgraph mode MD, one constructs two locks: \( \downarrow MD \) and \( \forall MD \); as MD is semantically absurd, we use MD in place of \( \forall MD \). One finds in Table 3 the different locks built from the modes R, W, RD and MD presented in Section 3.2.

### 3.5. The locking protocol

The protocol can be exposed in four points:

1. A transaction uses two-phase locking (2PL) [Eswaran et al. 76], i.e., before using any object, this object must have been locked either explicitly or implicitly.
2) To access to a set of objects, a transaction must lock:
   a) all the frontier classes of the subgraph (resp. the class);
   b) all the classes of any path, (the shorter, the better),
      from the root of the inheritance graph to a frontier class of the
      subgraph (resp. the class).
3) If the locking of the subgraph (resp. the class) does
   not imply the use of all the instances, then each instance must
   be locked separately before using it.
4) We use strict 2PL, i.e., locks are only released either
   when validation has completed or, if the transaction is rejected,
   when all its effects are undone. Lock release order has no
   importance.

Points (2) and (3) are enforced in the Pascal-like procedures
of Figure 4. This protocol does not impose any lock-request order.
Nonetheless, in order to avoid some deadlocks, it might be
 desirable to choose a system-wide canonical order.

Our protocol applies to logical objects, classes and
instances. Their representation into physical pages needs an
adapted protocol. We assume that one can use the IBM System
R protocol, with "long locks" on logical objects and "short
locks" on physical pages, which can be furthermore
generalized to nested transactions [Beeri et al. 88] (See
particularly [Cart & Ferrié 90]).

 procedure LockTheInstance (AnInstance : Instance;
                          Mode : AccessModes);
  begin
    if not AlreadyLocked(TheProperClassOf(AnInstance),Mode)
      then LockTheSubgraph(TheProperClassOf(AnInstance),
                     Mode,true,true);
  { actually lock the instance }
  end;

Figure 4: points (2) and (3) of the protocol

Using the locks made from the modes R and W, we detail
the example of Figure 5. Let us assume that there are two
concurrent transactions: T1 accesses in write mode to all the
general instances of c1; T2 accesses in read mode to some
instances of c4. We have an example of read/write conflict
between T1 which implicitly locks general instances of c1, and
consequently of c4, and T2 which locks explicitly some
instances of c4. T1 must request the locks situated on the left
of the classes r, c1 and c3 of the example, while T2 must
request the locks on the right of r, c2, c3 and c4. Besides, T2
has to individually lock each instance that it uses.

We assume that locks are requested top-down and left-to-
right first in the inheritance graph, next on instances if
necessary:
1. T1 locks r with $\forall W$;
2. T2 locks r with $\forall R$: both locks are path locks and
   consequently compatible;
3. T2 locks c2 with another $\forall R$;
4. T1 locks c1, root and frontier class of the subgraph,
   the instances of which T1 wants to read, with $\forall W$;
5. T2 locks c3 with $\forall R$;
6. T1 requests a lock $\forall\forall W$ on the class c3, but this lock
   is incompatible with the path lock of T2, so T1 is blocked;
7. T2 just needs to lock c4 with $\exists R$ to succeed in locking
   the inheritance graph;
8. T2 can now try to lock with R the instances that it
   wants to access to.

Figure 5: an example

In this example, the critical point is 5. In effect, c3 is the
only common class. As a consequence, the first transaction
that holds a lock on it is the one which can go further while the
second will wait.

If T2, in place of reading some instances of c4, would have
read the definition of c4, using the access mode LD (locking r,
c2 and c3 with $\forall LD$ and c4 with LD), then both transactions
could have been executed together.

{
\begin{verbatim}
typedef AccessModes = (R, W, RD, MD, ...);
ClassLock = record
  Type : (Frontier, Class, Path);
  Mode : AccessModes;
  Intention : boolean
end;
InstanceLock = AccessModes;

procedure LockTheSubgraph (ItsRoot : Class;
                           AccessMode : AccessModes;
                           Intention : boolean;
                           RootAlone : boolean);
  var
    Lock : ClassLock;
    AClass : Class;
  begin
    Lock.Mode := AccessMode;
    Lock.Type := Path;
    Lock.Intention := Intention;
    for AClass in APathFromTheRootTo(ItsRoot) - [ItsRoot]
      do LockTheClass(AClass,Lock);
    if RootAlone
      then begin
        Lock.Type := Class;
        LockTheClass(ItsRoot,Lock)
      end
      else begin
        Lock.Type := Frontier;
        for AClass in FrontierClassesOf(ItsRoot) \ [ItsRoot]
          do LockTheClass(AClass,Lock)
      end;
  end;

procedure LockTheInstance (AnInstance : Instance;
                          Mode : AccessModes);
  begin
    if not AlreadyLocked(TheProperClassOf(AnInstance),Mode)
      then LockTheSubgraph(TheProperClassOf(AnInstance),
                     Mode,true,true);
  { actually lock the instance }
  end;
\end{verbatim}
}
3.6. Particular cases
We now only need to apply our protocol to some particular cases: virtual accesses which happen during the late binding of methods; "phantom objects" during instance creations or deletions; at last, concurrent modifications of the inheritance graph.

3.6.1. Virtual accesses
A method call corresponds to a real access to an instance of one class, C, and to some virtual accesses to the ancestors of C, (in order to dynamically solve the reference to the method code which has to be applied).

Not locking the read classes for a virtual access is the proposed solution. On the other hand, the modification or the creation of a method in a class C needs not only the locking of the class but also of the subgraph rooted at C, in mode MM: modification of a method.

The locking of the subgraph rooted at C allows detecting any incompatibility with all accesses using general instances of C, (i.e., being able to be called with a method defined in C). This is necessary to avoid a transaction which applies method M to general instances of C, to apply, for the first instances, the method before the modification, and, for the last ones, the method after the modification.

This solution is proposed because we hope that read accesses will be more numerous than write accesses. Transactions which modify or add a method in a class C, are disadvantaged because they have to wait when some general instances of C are used. This limitation might be avoided by associating with each method a particular mode; however, the number of modes would be tremendously important. Using the parameters of the operations, here the names of the methods, would overcome this problem.

3.6.2. Creating and deleting instances
The problem can be quite merely solved or not depending on the degree of knowledge that the classes have of their instances. A class can:
- have no knowledge of its instances: it is the default case in O2;
- know its proper instances;
- know its general instances: it is the case in O2 if you create a class with the "with extension" clause [Lécluse & Richard 89];
- know its proper and general instances: it is the case in ORION [Banerjee et al. 87].

no knowledge of the instances
In that case, no locking is necessary! The instance identity is only known by the creating transaction, and any setting of that value to another instance variable requires the obtaining of a lock W on this other instance.

knowledge of proper and/or general instances
In the three other cases, hierarchical locking in an exclusive mode on the proper class of the created instance is sufficient to solve the problem of "phantom objects". We must create a new mode applicable to a class, CDI: creation and/or deletion of instances, incompatible with W, R, MD and itself, but compatible with RD. This mode may be applied to a subgraph using the frontier lock ∀CDI.

3.6.3. Modifying the inheritance graph
The modifying operations of the inheritance graph entails new concurrency control problems, essentially due to the changes of the frontier and path classes of some classes. We propose an extension of the protocol able to deal with these modifications on a concurrent and serializable way, but with some difficulties for the lock manager.

Among the modifications of the inheritance graph, from the simplest to the most difficult, we find:
(i) inheritance edge deletion;
(ii) inheritance edge creation;
(iii) class creation;
(iv) class deletion.

The semantics of these operations is described in [Banerjee et al. 87] [Penney & Stein 87]. We consider them just from the concurrency point of view.

These four modifications will be managed similarly, using the most exclusive access mode, i.e., MD (See Section 3.2.). They can be handled in the same way, based on the following analysis: At the time of the addition or deletion of one (resp. several) inheritance edge (c1,c2), c1 being the super-class of c2, the coherence must be maintained relatively to transactions which want to access to a part of the c2 rooted subgraph. We distinguish the transactions which request locks on a path going through unmodified edges from those which use a path going through added or removed edges.

Figure 6: the edge (c1,c2) is created or deleted; the frontier classes of c'1 and c1 are modified
For example, in Figure 6, the edge (c1,c2) is created (resp. deleted): from c'1 point of view, c2 has to become (resp. is no more) a frontier class; for c1, c2 does not yet (resp. no more) belong to its descendants.

To ensure the coherence for the two cases, the protocol becomes:
1) lock the c2 rooted subgraph in MD mode, i.e., request MD locks on the frontier classes and ∃MD locks on one path going only through unmodified edges;
2) lock each path going through a modified edge with ∃MD locks.

inheritance edge creation
It is sufficient to apply the above protocol.
After the presentation of the general characteristics of an object-oriented system, we detailed the set of possible accesses to the data. Then, and more classically, we distinguished various access modes, and even if we used as examples only well-known ones, much more can be added. However, we consider that the whole of the modes proposed in this paper offers a wide range of possibilities. The major improvement might be obtained by specializing W for each writing method.

Next, we achieved hierarchical locking. In our object-oriented environment, this scheme is restricted to two levels: classes and instances, i.e., locking a class may imply the implicit locking of all the proper instances of this class. Further, we extended hierarchical locking to a subgraph.

These three points of view led us to a systematic lock construction from access types, access modes, and their intentional nature.

At last, we proposed a locking protocol which can control all the data accesses that we mentioned at the beginning of this paper. Some particular cases have been treated separately; it seems to us that such cases must be left to a prototyping phase during the development of an application, because they imply a lot of restructuring work from the system.

The fault-tolerance problem, not discussed in this paper, may be influenced by the fact that we give a lot of freedom on the access mode compatibility relation. If one utilizes the large compatibility definition given in [Cart & Ferrie 90], the resolution of the problem is complicated because one instance may be concurrently modified by several transactions if they write different parts of that instance. Taking into account recovery, parameterized commutativity, and nested transactions are our current research interests.

ACKNOWLEDGMENTS: We gratefully acknowledge the helpful comments of Michèle Cart and Jean-François Pons. A lot of improvements are due to the numerous readings of Jean Ferrie.

5. Appendix

definition

The inheritance graph is \( G = (X, U) = (X, \Gamma) \) where \( X \) is the set of vertices, or classes, and \( U \) the inheritance relation, a set of couple from \( X \times X \). \( U \) is a partial order. \( G \) is rooted; its root is noted \( r \). \( \Gamma \) is the multi-valuated relation of a vertex successors, equivalent to \( U \). We also use \( \Gamma^+ \) for the predecessors.

We note \( \lambda = (s_0, ..., s_n) \) and call it a path of \( G \) if and only if \( \{ (s_0,s_1), (s_1,s_2), ..., (s_{n-1},s_n) \} \subseteq U \).

An induced subgraph of descendants, from now on just a subgraph, \( G_1 = (X_1, U_1) \) of \( G \) is composed of \( X_1 \), a subset of \( X \) formed of the set of descendants in \( X \) of a vertex \( r_1 \), and of \( U_1 \), the restriction of \( U \) to \( X_1 \). Formally, we define:

\[
X_1 = \{ x \in X \mid \exists \lambda = (r_1, ..., x) \text{ a path in } G \}
\]

\( r_1 \) is the root of the subgraph \( G_1 \). We will note a subgraph of \( G, G_{r_1} \), where \( r_1 \) is the root vertex of this subgraph.

definition

We note a frontier vertex of \( G_1 = (X_1, \Gamma) \) a vertex of \( X_1 \) which has at least one predecessor not in \( X_1 \). Conventionally,
we say that $r_1$ is a frontier vertex, (which is always true except for $r$, root of $G$). Formally, we describe the set of frontier vertices of $G_1$ by:
\[ F(G_1) = \{ x \in X_1 \mid \exists y \in \Gamma(x) \setminus y \in X_1 \} \cup \{ r_1 \} \]

**definition**

Two subgraphs, $G_1$ and $G_2$, have a strict intersection when:
- $X_1 \cap X_2 \neq \emptyset$
- $X_1 \cup X_2 \neq X_1$
- $X_1 \cup X_2 \neq X_2$

**definition**

Let $G_1 = (X_1, U_1)$ and $G_2 = (X_2, U_2)$ be two subgraphs with a strict intersection, we define:
\[ X_1 \cap X_2 \rightarrow \mathbb{N} \]
\[ d: x \rightarrow l_1 + l_2 \]
with $\lambda_1 = (r_1 = t_0, ..., t_{l_1} = x)$ and $\lambda_2 = (r_2 = s_0, ..., s_{l_2} = x)$ maximal length paths in $G_1$ between $r_1$ and $x$ (respectively $G_2$ and $r_2$).

**property**

\[ \forall x \in X_1 \cap X_2 \text{ with } d(x) \text{ minimal}, \]
\[ \exists y \in \Gamma(x) \mid y \in X_1 \text{ and } \exists y' \in \Gamma(x) \mid y' \in X_2 \]

**proof**

Let us proceed by contradiction and, without loss of generality, let us suppose that
\[ \forall y \in \Gamma(x), y \in X_1 \]
then
(i) if $\forall y \in \Gamma(x), y \notin X_2$ then $x \notin X_1 \cap X_2$ because $G_2$ is a descendants subgraph, thus contradiction;
(ii) if $\exists y \in \Gamma(x) \mid y \in X_2$ then $y \in X_1 \cap X_2$ and $d(y)$ is defined with $d(y) \leq d(x) - 2$, whereas $d(x)$ is minimal, whence another contradiction.

**corollary**

Two subgraphs having a strict intersection have at least one common frontier vertex.

**proof**

The minimal vertices, as defined above, are frontier vertices.

**property**

\[ \forall \lambda = (s_1 = t_0, ..., t_n = s_2) \text{ with } n > 0, \]
\[ \forall \mu = (s_1 = r_0, ..., r_m = s_2) \text{ with } m > 0, \]
\[ \forall t_j: 1 \leq i \leq n, \]
\[ \exists r_j: 1 \leq j \leq m \]
\[ r_j \in G_t \text{ and } \]
\[ r_j - 1 \notin G_t \]

**proof**

Proceeding by contradiction, we obtain:
(i) $r_j - 1 \notin G_t$, then $r_m \notin G_t$, and $s_2 \notin G_2$: a contradiction:
(ii) $r_j - 1 \in G_t$, and then $r_0, ..., r_{m-1} \in G_t$: another contradiction.

**corollary**

Let $G_1$ and $G_2$ be two subgraphs with $G_1 \subseteq G_2 \subseteq G$, then on any path from a vertex of $G_1$ to $r$, root vertex of $G$, there is at least a frontier vertex of $G_2$.

**proof**

We only need to specialize the vertex $s_1$ in the previous property to $r$, root of $G$. The $r_j$ are frontier vertices. Finally, $r = r_0$ is, by definition, a frontier vertex.

6. References


[Goldberg & Robson 83] A. Goldberg, D. Robson; SmallTalk-80, the language and its implementation; Addison-Wesley, Reading, Massachusetts, 1983

[Gray 78] J. N. Gray; Notes on database operating systems in Lecture notes in computer science 60, advanced course on operating systems; Springer-Verlag, New-York, 1978


190


[Penney & Stein 87] D. J. Penney, J. Stein; Class modification in the GemStone object-oriented DBMS; OOPSLA '87 proceedings, October 1987, Orlando, Florida

[Schaffert et al. 86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, C. Wilpolt; An introduction to Trellis/Owl; OOPSLA '86 proceedings, September 1986, Portland, Oregon, pp. 9-16