A Model of Methods Access Authorization in Object-oriented Databases

Nurith Gal-Oz,[1] Ehud Gudes[1] and Eduardo B. Fernandez[2]

[1] Dept of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, Israel.
e-mail: {nourith,ehud}@bengus.bitnet

[2] Dept of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431, USA.
e-mail: ed@cse.fau.edu

Abstract

Object-oriented databases are a recent and important development and many studies of them have been performed. These consider aspects such as data modeling, query languages, performance, and concurrency control. Relatively few studies address their security, a critical aspect in systems like these that have a complex and rich data structuring.

We developed previously a model of authorization for object-oriented databases which includes a set of policies, a structure for authorization rules and their administration, and evaluation algorithms. In that model the high-level query requests were resolved into read and writes at the authorization level. In this paper we extend the set of access primitives to include ways to control the execution of methods or functions. Policy issues are discussed first, and then algorithms for access evaluation at compile-time and at run-time.

Keywords: Database Security, Object-oriented Security, Methods, Object-oriented Systems

1 Introduction

Object-oriented databases are a recent and important development and many studies of them have been performed [12,13,17]. These consider aspects such as data modeling, query languages, performance, and concurrency control. Relatively few studies address their security, a critical aspect in systems like these that have a complex and rich data structuring [3,10,15,16].

We developed previously a model of authorization for object-oriented databases which includes a set of policies, a structure for authorization rules and their administration and evaluation algorithms [5,6,9,11]. In that model the high-level query requests were resolved into read and writes at the authorization level. Read and write primitives model adequately access to class attributes, however, they are not in line with the *encapsulation* property, a fundamental feature of object-oriented systems. In this paper we show that through the use of one basic access type, *Execute*, which is required to invoke the execution of a method, we can control access to objects. Because execution of a method may require using other methods not directly authorized to the user, we combine access type Execute with a form of rights amplification similar to the *set-user-id* concept of Unix. Our previous model used the idea of implied access rights through the inheritance structure of the data classes. This idea was embodied in three policies, which are extended here to consider the effect of the new access mechanism. The model also considers the effects of negative authorization and multiple inheritance.

In Section 2 we review our previous model and give some examples. In Section 3 we present the new model with methods and define security policies for it. In Section 4 we discuss algorithms for the evaluation of access according to these policies. Section 5 contains discussions and a summary.

We know only one recent model for OODBs that controls access through methods, the one of the IRIS DBMS [1], and we compare our model to that in Section 5. There are other models for authorization in object oriented databases which control read/write access to attributes, but do not consider methods [15]. Other models apply to multilevel security policies, e.g. [10], and have a completely different emphasis.

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

2 Background

2.1 Database model

The database model involves three types of concepts: *classes*, *attributes*, and *methods*, two types of associations between them: *generalization*, to describe the hierarchical relationship between a class and a subclass, and *aggregation* to describe the relationship between a class and its attributes. A class defines a data structure composed of *attributes* and a set of relevant operations, called *methods.*

Both attributes and methods of a class are *inherited* by the subclasses of this class. Inheritance propagates down the hierarchy unless an attribute or method is redefined by a subclass, in which case the redefined feature is propagated down. It is possible for a subclass to have multiple parents and therefore to inherit attributes or methods from all of them. In case of some conflicts in multiple inheritance, it is resolved by some arbitrary rule, and the inheriting parent is called the "direct father" with respect to an attribute or a method.

An object is an instance of a class. The concept of encapsulation implies that access to the data in an object can only be performed through a method. Some methods may be very basic, e.g. just read or write the value of some attributes, while others may perform complex calculations for which they need to invoke other methods (and usually the basic read/write methods). In the sequel we assume that every attribute has the read/write methods implicitly defined for it.

Figure 1 illustrates a portion of a university database. Class Person (P) has *attributes* Name, Social Security Number (SSN) , Birthdate (and maybe others). Person has the method *age* which computes the age of a person based on his birthdate. Classes Student (S) and Teacher (T) are subclasses of Person. The generic properties of Student and Teacher define Person through a generalization association (G in Figure 1). Attribute Year (year of graduation) is defined for Student and attributes "Rank" and "Course" for Teacher. Student has two methods: *gpa* which computes the grade point average, *find_yb* - find young and bright which invokes the two other methods: age and gpa to retrieve the young and bright students. Teacher contains the methos *salary* which computes the salary based on Rank and other information. Foreign Student (FS) is a subclass of Student. Attributes defined for subclasses reflect the fact that some features or properties only apply to specific subclasses, e.g, Visa is only meaningful for Foreign Students. Figure 2 shows the same database with all methods (including the imlicit read/write) shown.

Note that the values inherited by a class are a subset of those of the superclass, i.e. SSN as an attribute of Student represents only the SSNs of Students, while the values of SSN as an attribute of Person represent SSNs of Students as well as of Teachers. Similarly, Foreign_student inherits the method age from Person, which when invoked at this level will only find the ages of foreign students.

2.2 Security policies

In previous work [5] we proposed the following policies:

P_1 (inheritance) – a user that has access to a class is allowed to have similar type of access in the subclasses to the corresponding attributes inherited from that class.

P_2 (class access) – access to a complete class implies access to the attributes defined in that class as well as to attributes inherited from a higher class (but only to the class-relevant values of these attributes). If there is more than one ancestor (multiple inheritance) there is access to the union of the inherited attributes.

P_3 (visibility) – an attribute defined for a subclass is not accessible by accessing any of its superclasses.

We also proposed policies to consider negative authorization, content-dependent restrictions, and for resolving conflicts between several implied authorizations (see [11]). In section 3 we extend these policies to apply to access through methods.

A general model for authorization rules is discussed in [4]. Here, we define an authorization rule, as a triple (U, A, AO) where U is a user or user group, A is an access type or set of access types, and AO is the set of attributes of the object O to be accessed, i.e. AO={ O. a_1, O. a_2 ...}. A rule can either refer to AO as a whole or to its individual components. Attribute a_i must be defined for object O or inherited by it.

For example, consider again the graph of Figure 1. Assume the following authorization rules are defined:

R1: (SA, R, S.SSN) – The Student Advisor can read SSN of students.

R2: (FSA, R, (FS.SSN, FS.Visa)) – The Foreign Student Advisor can read SSN and visa of foreign students.

A Student Advisor (SA) could have access to SSNs of all students (P_1) but no access to their visas (P_3), a Foreign Student Advisor (FSA) could have access to visas but only to SSNs of Foreign Students (P_2). Note that SA has access also to SSNs of foreign students; a negative authorization rule (SA, -R,FS.SSN) could prevent this access if necessary.

2.3 Validation of access requests

Access validation occurs by extracting a *data request* from a user query or from an executing program. This request has a structure (U', A', AO'), where U' is the subject (user, process) making the request, AO' is the requested object (set of attributes), and A' is the requested access type. This request is validated against the authorization rules to decide if the request should be granted totally or partially.

A *Security Context* is a set of object classes grouped together for security purposes. A Security Context defines a partially ordered set of object classes (in terms of the associations) which delimits the access for user queries, i.e. a data request is validated using the rules in a specific context. Referring to Figure 1 we can define a Security Context SC1 to include classes Person, Student, and Foreign Student, as well as their corresponding associations. In the rest of the paper we'll use examples from this security context.

Object-oriented databases are very often distributed and do not have a centralized schema. Therefore, authorization rules also are not stored, in general, in a centralized location but rather distributed throughout the object-hierarchy. The place of storage of authorization rules affects considerably the way of validating access requests. Authorization rules can be placed at special classes (e.g., a context root), at the class to which they refer, or propagated throughout the hierarchy. Their placement has no effect in the logical aspects of the model but is important with respect to performance of the access validation algorithm.

For the example and rules above we define the following two queries each of which is issued by SA and FSA.

Q1: read SSN for all students.

Q2: read SSN and visa for all foreign students.

According to the policies of Section 2.1 we expect the following behaviour as a result of the evaluation of the indicated requests.

(SA, Q1) = (SA, Read S.SSN) - all SSNs of students can be read (Policy P_1).

(SA, Q2) = (SA, Read,{FS.Visa, FS.SSN}) - only SSNs of foreign students are to be read and not their visas (Policy P_3).

(FSA, Q1) = (FSA, Read, S.SSN) - only foreign student SSNs are to be read (Policy P_2).

(FSA, Q2) = (FSA, Read {FS.Visa, FS.SSN}) - both foreign student SSNs and visas are to be read. (Policy P2).

In [5] we presented an algorithm to evaluate access based on the above model, and showed how the queries above are answered correctly by this algorithm. The basic inputs to this algorithm are the *query graph* which represent the query in form of a graph, and the *security graph* which represent all the rules relevant to the query in the relevant security context. The algorithm assumes that authorization rules are stored with the object to which they refer, i.e the following placement rule is used:

An access rule (U, A, {Oi.a1,Oi.a2...}) can be placed at any node Oi such that:
a_i is known for object Oi (defined or inherited from above).

In [8] we improved the algorithm by efficiently supporting multiple node queries and by assuming a more general placement rule which allowed placing rules at ancestors of the objects they reference. That algorithm is the basis for the one presented here.

For lack of space, we cannot present that algorithm here. The most important concept there is the AT data structure(Authorization Tree - called there: AT_yes). The AT contains information from the query graph as well as the results of each step of the authorization validation. Each node contains information about the attributes which need to be accessed at that node, specifically whether the node (attribute) has full, partial, or no authorization (see later). The algorithm scans the security graph and the AT and updates the various fields in AT. At the end, each node of AT contains flags indicating its specific authorization (for example, for the query Q1 above and FSA, the node corresponding to Foreign-student will be fully authorized, while the Student node will be partially authorized).

3 Policies for Authorization of Methods

3.1 Access Control Mechanism

We can have two types of authorization models based on methods:

1. Methods correspond to access types in the access rule. This is consistent with the notion of abstract data types, namely an object can only be accessed via its predefined methods. For example, class Student could only be accessed by methods such as add-student, enroll-student, etc. The internal actions of the methods, i.e. what other methods they call, and what attributes they access is the responsibility of the system, and no special authorization for them is required of the user.

2. Methods correspond to authorization objects in the access rule. In this case some special access

type is required to control the use of methods, e.g *call* in [1], *execute* in our model. If the method needs access to other methods the user needs authorization for those as well.

The first model is described in detail in [7]. We adopt here the second model and we use a unique acees type, *Execute*, which authorizes to apply a given method to a given object.

In operating systems it is often the case that users need to execute programs in a mode which has more rights than their own. This is called **amplification**. For example, in Unix, this is handled by the "Set user-id" mechanism. We add a similar mechanism here, and therefore we need an access type called "Execute with set user-id". Execute, and Execute with set user-id, can be denied to provide for negative authorization.

3.2 Security policies

We extend the policies of Section 2.2 to consider the new model. We complete them by adding a policy for negative authorization, a policy for handling implied accesses and a policy to consider the effect of Setuid:

P_1' – a user that has Execute access to a method in a class is allowed to have similar Execute access in its subclasses to the corresponding inherited methods.

P_2' – access to a complete class implies Execute access to the methods defined in the class as well as to methods inherited from a higher class. If there is more than one ancestor (multiple inheritance) there is access to the union of the inherited methods.

P_3' – A method defined for a subclass is not accessible by having Execute access to the methods of a superclass. This lack of visibility applies also to inherited methods redefined in the subclass.

P_4 (Negative authorization) – An explicit negative Execute access to a method overides any implied or explicit positive Execute access (of the same type).

P_5 (Priority) – When two implied Execute accesses apply to a method, the closest one has priority.

P_6 (Invocation) – If a method invokes other methods during execution, the user must have Execute access to these other methods as well (recursively). This applies also to the basic read/write methods.

P_7 (Amplification) – A user can be authorized to apply the Execute access type to some method with rights recieved from another user (who has Execute access right to the method).

The mechanism to apply policy P7 is that of Setuid. As explained above, a setuid authorization on a method can be created to overcome situations where a user does not have authorization for an invoked method called by an invoking method and therefore is not authorized to execute the invoking method by virtue of policy P6, even if it is important that he will. The problem is solved by making this user dependent on another user's authorizations, for the purpose of the invoked method execution. The user granting the right should not have recieved it through another setuid but must have been authorized directly by the administrator (i.e. there is no recursive setuid policy...)

Note that to apply policy P6, the knowledge which methods are invoked within a method must be known to the access evaluation algorithm. If this algorithm applies at compile time, this knowledge must be stored with the methods definitions. This point is further elaborated on in Section 4.

As an example consider the database in Figure 1. An accountant is a user of this database and his job is to compute the salary of the teachers using the method Teacher.salary. This method accesses the attribute Rank which is one of the parameters needed to compute the salary. The accountant has an Execute access to the method but is unauthorized to access Rank. The personnel manager is another user of this database which has access to all the attributes and methods of class teacher, and he uses the accountant services. Therefore, it is convenient to arrange that the accountant becomes dependent on the personnel manager who gives him setuid[personnel manager] access to the method Teacher.salary.

As another example, assume a query that requests the age of all students is evaluated. Two methods need to be accessed: Student.age which uses the attribute Student.Birthdate, and Foreign_student.age which uses the attribute Foreign_student.Birthdate. (note that these attributes would be accessed through the read/write methods of the corresponding attributes). Now, there may be several cases:

- Method age is not defined in foreign student and is therefore inherited from student. If the user has an Execute access right on student.age then he has an implicit Execute right on foreign-student.age. Now the access depends on the type of access allowed on attribute birthdate. If, for example, the user is authorized to access Student.Birthdate but denied access to Foreign_student.Birthdate, then by Policy 6 there will only be a partial answer to student.age query. A similar result will be if the user has Execute right on student.age but deny-execute on foreign_student.age. A partial answer will also be generated if the user has no access to student.age (birthdate) but positive Execute acess to foreign_student.age (birthdate).

- Foreign_student.age is redefined in foreign_stu-

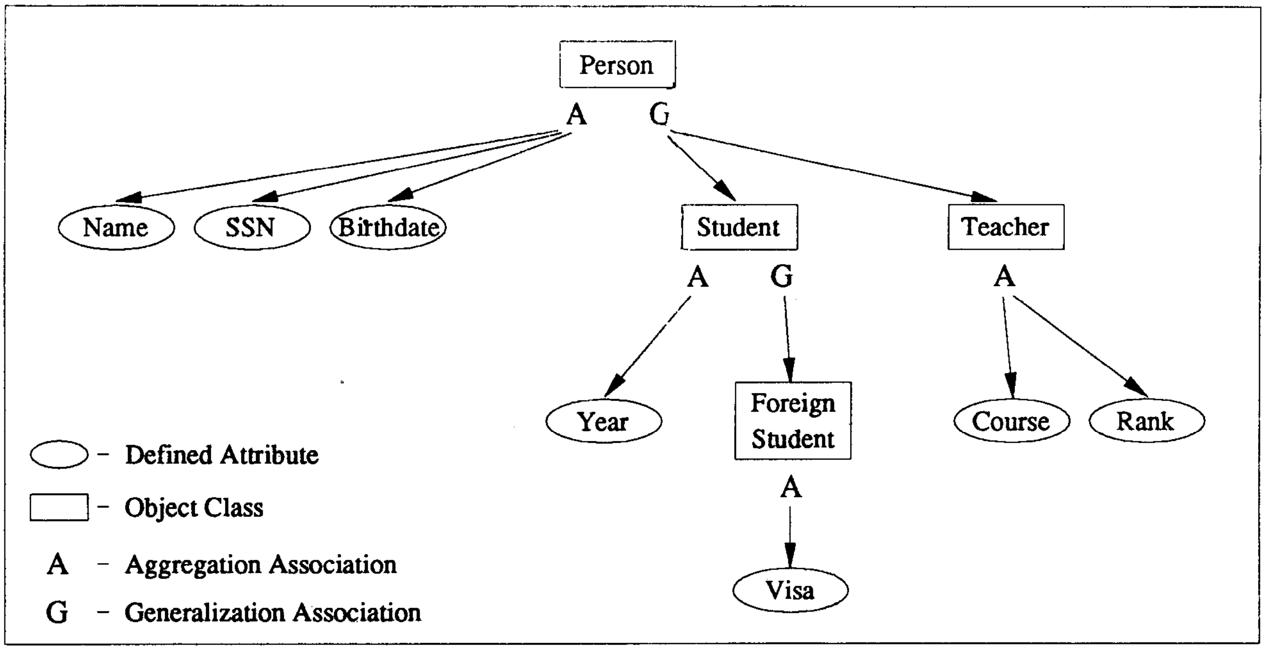

Figure 1: A University Database

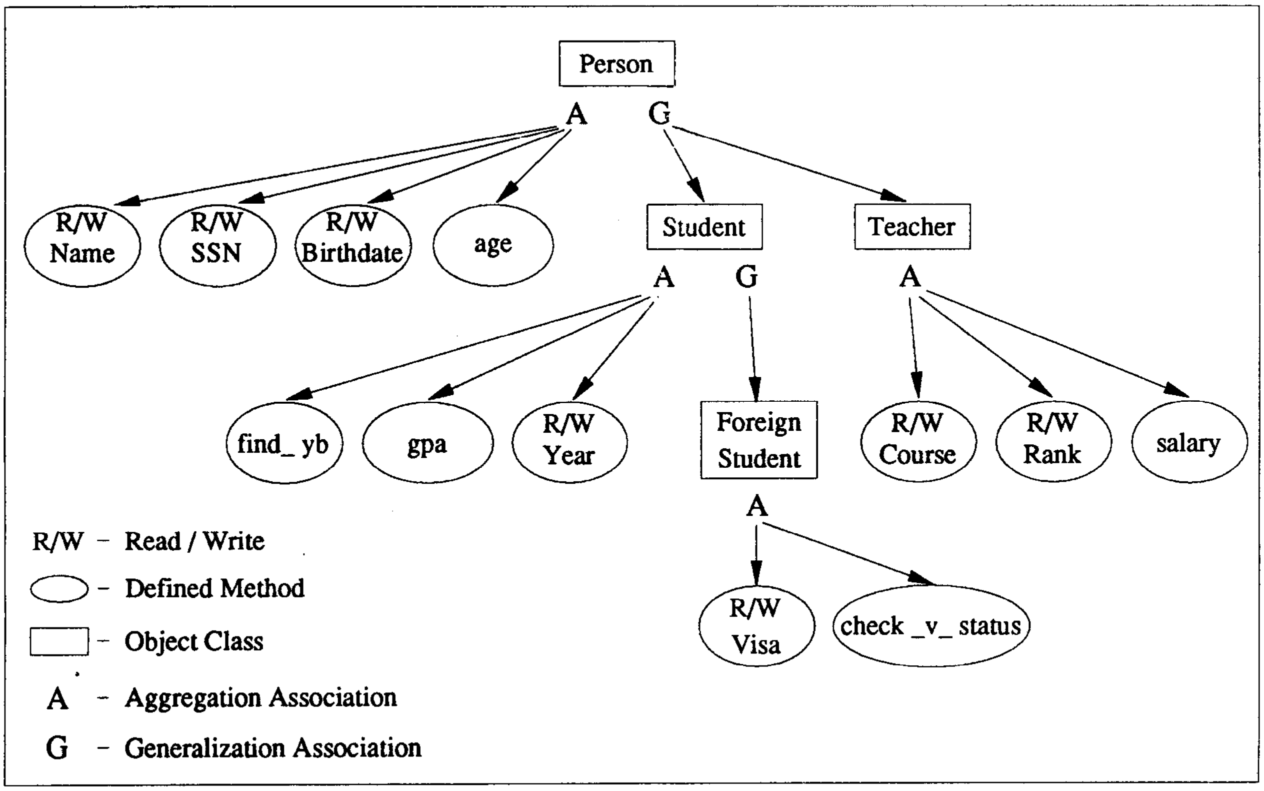

Figure 2: The Database With Methods

dent. Now according to policy 3 an Execute access on student.age is not carried over to foreign_student.age and an explicit access to foreign_student.age is required (same comments for birthdate).

We discuss next the algorithms to evaluate access based on the above policies.

4 Evaluation Algorithms

The evaluation algorithms are similar to those mentioned in Section 2, that is, it is assumed that the query translator generates a query graph to represent the query. The query graph is converted into a data structure called AT, using the security context concept. Figure 3 clarifies the above concepts. It shows three entities:

1. The query - (SA, E, Student.find_yb).
2. The security context which involves basically the relevant portions of the database shown in Figure 1.
3. The AT which contains all nodes touched by the query including their ancestors and descendants. This is needed to assure that all relevant rules will be searched.

Next, the evaluation algorithm scans the AT data structure and the security graph, searches for appropriate authorization rules and updates the AT accordingly. The results is that each method node in the tree is associated with one of four states:

- **fully granted** - a node is fully granted if it is granted and in each one of the subclasses of its class this node is fully granted (empty subclasses are considered fully granted).
- **fully denied** - a node is fully denied if it is denied and in each one of the subclasses of its class this node is fully denied.
- **partially granted** - a node is partially granted if it is granted and in one or more of the subclasses of its class, the node is not fully granted.
- **partially denied** - a node is partially denied if it is denied and in one of the subclasses of its class, the node is not fully denied.

These concepts are important at *query run-time*. At run-time the retrieval algorithm may skip some searching of the object-hierarchy (this can also be used by the query optimizer). For a class which is fully granted (denied), all (none) object instances are retrieved and there is no need to go down the AT graph hierarchy for these types of nodes (although for fully granted the query run-time may have to go down the database hierarchy to fetch all instances). For partially granted nodes the query evaluation algorithm knows that partial results are possible and will go down the AT graph hierarchy and the database hierarchy to check which specific nodes are granted and fetch their corresponding instances.

We now discuss the access evaluation algorithm. In [5] and [9] we discussed several algorithms for scanning the AT tree. The difference was mainly with the direction of search along the tree. Here, because of the existence of negative authorization, there is always the need to scan the tree all the way to the bottom, therefore the search can start at the bottom level and go up. This is what the algorithm does by recursively evaluating the sons of a node before its father.

Also, in our earlier papers we had two types of placement rules. Placement rule 1 which was discussed in Section 2, and placement rule 2 which allows the placement of rules anywhere in the hierarchy above the object to which they refer. Placement rule 2 has advantages with complex queries and positive authorization, but these advantages are smaller when negative authorization is allowed. In this paper therefore we assume the simpler placement rule 1, where a rule is placed with the object it references.

The main difference of the algorithm below and our previous algorithms derives from the existence of methods. Since, by Policy 6, in order to have authorization to execute a method it is necessary also to have the correct authoriztion for the methods it invokes, it is advantageous to evaluate the authorization for each method separately, since that evaluation may call recursively for the evaluation of other methods (and maybe repeat that for another user in case of a setuid situation). However, if at any point the evaluation of authorization for a method is completed, it is not repeated again. This can be seen easily from the algorithm below.

Another issue is Compile-time vs. Run-time. The compile-time algorithm uses only information known at compile-time. This has two implications. First it means that all methods invoked by a particular method must be known. This is assumed to be done at the time the AT is generated by having the the schema of each method containing the names of methods it directly calls. Second, the access evaluation algorithm assumes the "worst-case" in which all potentially invoked method are actually invoked, and check autho-

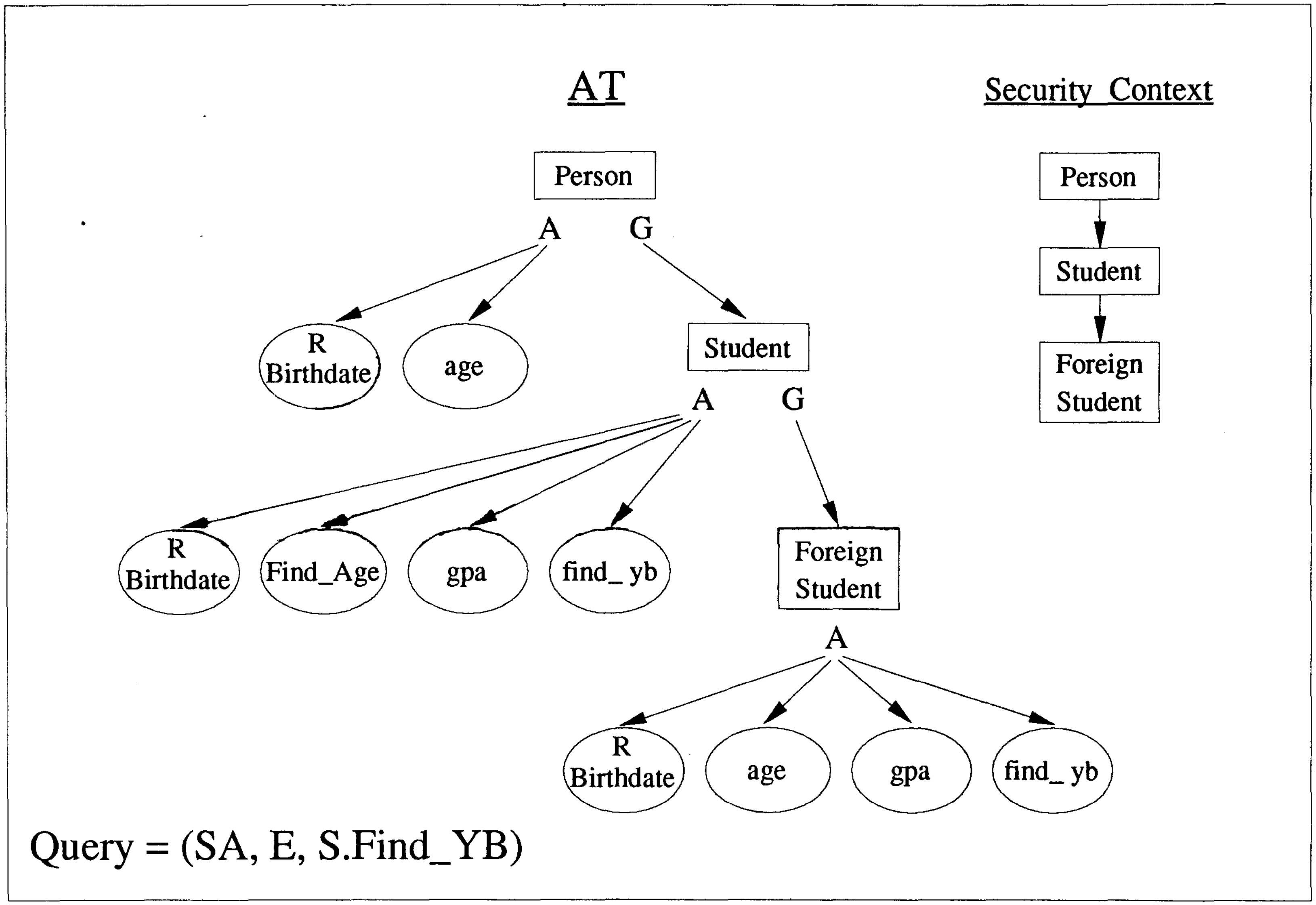

Figure 3 : An Example AT

rization accordingly. This may be too restrictive, since no consideration is given to methods flow of control. A run-time algorithm may provide the needed security without being overprotective. This is discussed below.

We now describe in detail the two algorithms.

4.1 Compile-Time Algorithm

```
procedure evaluate

Starting with the highest class in the
query graph:
for each class
  for each method of this class appearing in AT
    evaluate_method(method, class)

end evaluate

procedure evaluate_method (Method, Class)

1) If there are subclasses to this Class then
  A) for each subclass of this class
    a) find the inherited method in subclass
    b) evaluate_method (method, subclass)

2)
  A) search for authorization rule in the
     method node. If such a rule is found
     then method execution is authorized,
     or denied according to the rule.
  B) if there are no relevant rules then
       if the method is inherited then
         a. find the same method in the
            direct father.
         b. go back to step (A) with the
            father's method.
         c. the method gets authorization found
            for the father method.
       else (the method is defined)
         according to the idea of a closed system
         method execution is denied.
  C) if the method execution is authorized then
       a. for each child_method invoked by
          the method:
            evaluate_method(child_method,class)
       b. if all child methods are authorized
            then the method is granted
          else
            the method execution is denied
            in the class.
  D) if the method execution is denied and
     setuid is granted then
       a. find the authorizing user.
       b. repeat steps 2-A through 2-C with the
          authorizing user.

3) if the method is granted/denied then
     if the Class has no subclasses then
       set method to fully granted/denied
     else if the methods in all the subclasses
             are fully granted/denied then
       set method to fully granted/denied
     else
       set method to partially granted/denied

end  evaluate_method
```

The following example demonstrates the operation of the above algorithm.

Assume the following rules refer to the database presented in Figure 1:
R3 = (FSA,E,FS.age)
R4 = (FSA,-E,S.read_Birthdate)
R5 = (SA,E,FS.read_Birthdate)
R6 = (SA,E,FS.age)
R7 = (FSA,Setuid[SA],FS.age)

Let's evaluate the query Q = (FSA,E,S.age) according to the algorithm. (algoritm's steps are denoted in brackets).
The Security Graph contains the two methods S.age and FS.age. We start in class Student (S), with method age. Class S has one subclass - Foreign-Student (FS) so FS.age is evaluated first [*1*]. In FS.age the explicit rule R3 is found , authorizing FSA to access FS.age [*2.A*], but since method age uses attribute Birthdate we need Read access to FS.Birthdate as well (i.e. Execute right on read_Birthdate method)[*2.C*]. An explicit rule concerning FSA is not found in FS.Birthdate, but as it is an inherited attribute we search for a rule in the direct father, in S.Birthdate, and we find R4. Rule R4 denies FSA to Read S.Birthdate and this denial is inherited by FS.Birthdate[*2.B*]. Therefore, method age is denied for FSA by direct authorization.
Next, we search for an indirect authorization[*2.D*] and we find R7 which allows FSA to execute FS.age if SA is authorized. For user SA the search is much shorter because of the two explicit rules R5 and R6 which together authorize SA to execute the method FS.age, and indirectly authorize FSA as well. FS.age is therefore authorized for user FSA and since FS has no subclasses, it is fully-granted. Now, we turn back to evaluate S.age but no explicit rule is found, there is no implicit one in class Person either, which is the direct father class of Student with respect to method age. In class Person, age is a defined method, so according to the idea of a closed system, Person.age is denied and this denial is inherited by S.age. Again we search for an alternative indirect authorization rule but there isn't any. S.age is therefore denied but since FS.age is

fully-granted, we say that S.age is partially-denied.

Now we know that the answer to the query will be partial because only the age of the Foreign Students will be computed, *i.e.*, FSA will be allowed to compute age of foreign students only.

The complexity of this algorithm is in the worst case O(q*n*k*m) where q is the number of methods in the query, n is the number of nodes in AT, k is the number of rules per node, and m is the total number of methods invoked by a single method. This complexity derives mainly from the fine granularity of security we support, in particular the ability to apply security checks at each method invokation. Using the model in [7], the complexity is much reduced. Furthermore, average performance is much better as discussed for the originial algorithm in [6].

4.2 Run-Time Algorithm

An evaluation algorithm is measured in terms of security and precision [2], applied to our model as follows:

security: if an object is unauthorized according to the security policies, then it is unauthorized according to the algorithm.

precision: if an object is unauthorized according to the algorithm, then it is unauthorized according to the security policies.

The proposed compile-time algorithm is overprotective since it provides security but not precision. The problem arises when methods are involved, because a methods flow of control is unknown until run-time. In the compile-time algorithm, authorization of a method requires the authorization of all access rights needed in the attributes involved in the method. However, this requirement is too strong because an access to an attribute involved in a method may not be needed with some specific flow of control within the method.

A possible way to solve this problem is to use the results of the compile- time algorithm and to conduct checks at run-time as follows: In the compile-time algorithm, a method which is not authorized because one of the nodes involved in it is unauthorized, will be marked "undecided". At run-time, if a method is marked "undecided", then each time before the method reaches a point where an access to a node is needed, the authorization to this node is checked (according to the results of the compile-time algorithm). If the node is authorized (partially or fully), then the method is executed, while if the node is fully denied the method is unauthorized and thus aborted.

5 Discussion and Conclusions

Any model for security in object-oriented databases is dependent on the underlying data model, therefore some comparisons are not always meaningful. However, since there is a related model [1], a comparison with that model can be enlightening. Similarly to our model, in [1] all accesses and administration of access is expressed through functions calls (methods in our case), which is an advantage because of its uniformity. On the other hand, because attributes and methods are not distinct, security is usually based on the arguments of methods. This may be quite cumbersome as indicated by the authors themselves. Also, in [1] there is no specific mechanism for amplification such as our set-uid mechanism. Other differences are:

- The Call Privilege
  In [1] the call authority is argument specific, i.e. a user can have an authority to call a function for certain types of arguments. In our model a user may call a method with any argument, the authority to use it is automatically checked by the fact that the user must have Execute authorization on any method invoked from the original method. This allows for a more precise and flexible sceme of authorization.
- Static and Dynamic Authorization of Derived Functions.
  In [1] derived functions are defined in terms of other functions. Similarly, methods in our model may use other methods and attributes. Two approaches are suggested in [1] with respect to derived functions. The first is Static Authorization in which, in order to evaluate a derived function, a call authority on the function is sufficient. This is similar to the first model discussed in Section 3. The second is Dynamic Authorization in which a user must have call authority on the derived function as well as on the function it uses. Dynamic authorization matches our policy regarding methods , i.e. a user may call a method if he has an execution authority on the method as well as Execute access rights on each of the methods used by the derived method. We have suggested two algorithms to accomplish this action. The first, evaluated at compile time, checks the existence of proper access rights to all methods invoked by the derived method. The second is a run time algorithm which checks existence of proper access rights only on methods required at run time. This algorithm apparently, suits the spirit of Dynamic Authorization as presented in [1].
- The interaction of Authorization and Function Resolution.
  [1] presents two authorization approaches regarding generic functions : Generic Function Authorization

and Specific Function Authorization. In the Generic Function Authorization if a user is authorized to evaluate a generic function , he can evaluate it for any object. The Specific Function Authorization approach supports specifying authorization to specific functions, and in order to select the right specific function two function resolution approaches are suggested : the first is Authorization Independent Resolution in which, when a specific function is selected, the authorization on it is checked. The other is Authorization Dependent Resolution in which, the most specific function the user is authorized to evaluate is executed. In our model resolution has nothing to do with security. A generic method (in the sense of applying it to inherited subclasses) is connected to a class and according to database policy, the right method is selected for each class and subclass. For example, if the method *salary* is connected to class Admin, and Manager is a subclass of class Admin, then a method by the name salary is connected to class Manager as well, only for security purposes, we treat them as two different methods Admin.salary and Manager.salary and a user may have different authorities on them. This is also the way we can provide different authorization for polymorphic methods.

Guard Functions.
Guard functions are used to restrict evaluation of other functions for security puposes. In our model, we can use predicates in order to restrict access to attributes or methods in some conditions, although we have not shown it in the paper.

In conclusion we believe that our model supports the security policies regarding methods which are common in object-oriented databases, and provides flexibility and power. Our algorithms for access evaluation can be integrated into a typical query translator and runtime system. The model can be extended with more general methods and for message-oriented databases.

References

1. Ahad R., Lyngbaek P., and Onuegbe E., "Supporting access control in an object-oriented database language," *Proc. EDBT-92*, Vienna, March, 1992, pp. 184-200.
2. Denning, D., *Cryptography and Data Security*, Addison Wesley,1982.
3. Dittrich, K., Hartig, M., Pffefferle, H., "Discretionary Acces Control in Structually Object-Oriented Database Systems", in *Database Security II: Status and Prospectus*, C.E.Landwehr(ed.), Elsevier Science Publ., 1989, 105-121.
4. E.B. Fernandez, R. C. Summers and C. Wood, *Database security and Integrity*, Addison-Wesley, 1981.
5. E.B. Fernandez, E. Gudes, and H. Song, "A security model for object-oriented databases", *Proc. of the 1989 IEEE Symp. on Security and Privacy*, Oakland, CA., 1989, 110-115.
6. E.B.Fernandez, E.Gudes, and H.Song, "A model for evaluation and administration of security in object-oriented databases", to appear in *IEEE Trans. on Knowledge and Data Eng.*
7. E.B.Fernandez, Larrondo-Petrie M., and E.Gudes, "A method-based authorization model for object-oriented databases", submitted for publication.
8. Song, H., "Evaluation of authorization in object-oriented and semantic databases", MSc Thesis, Florida Atlantic University, 1990.
9. Gudes E., Song, H., Fernandez E B., "Evaluation of negative and predicate-based authorization in object-oriented databases," *Database Security IV: Status and Prospectus*, S Jajodia and C. E. Landwehr (Ed.), Elsevier Science Publishers, 1991, 85-98.
10. T.F.Keefe, W.T.Tsai, and M.B. Thuraisingham, "SODA:A secure object-oriented database system" ,*Computers and Security* , 8 (1989), 517-533.
11. M. Larrondo-Petrie, E. Gudes, H. Song and E. B. Fernandez, "Security Policies in object-oriented databases," in *Database Security III: Status and Prospects*, D. L. Spooner and C. Landwehr (Eds.), Elsevier Science Publishers, 1990, 257-268.
12. J.G.Hughes, *Object-oriented Databases*, Prentice Hall Intl., 1991.
13. Kim, W. *Introduction to object-oriented databases*, MIT Press, 1990.
14. T. Lunt, "Access control policies for database systems," in *Database Security II: Status and Prospectus*, C.E.Landwehr(ed.), Elsevier Science Publ., 1989, 41-52.
15. F. Rabitti, E. Bertino, W. Kim, and D. Woelk, "A model of authorization for next-generation database systems", *ACM Trans. on Database systems*, 16,1 (March 1991) 88-131.
16. Spooner, D., L. "The impact of inheritance on security in object-oriented database systems," *Database security II: Status and Prospectus*, C. E. Landwehr, Elsevier Science Publ., 1989, 141-150.
17. Stonebraker M. (Ed), Special issue on Database Prototype Systems, *IEEE Trans. on Knowledge and Data Engineering*, Vol 2, No. 1, March, 1990.