7

A formal specification of an authorization model for object-oriented databases

E. B. Fernandez, R. B. France, and D. Wei
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431, USA
Tel: 407-367-3466; Fax: 407-367-2800
Email: (ed, robert)@cse.fau.edu

Abstract

We present here a formal specification, using Z, of an authorization model for object-oriented databases. We assume that the database and the authorization rules have already been defined and we describe the structure and the properties the rules should satisfy to represent a secure system. In this way, each new access rule can be tested for compliance with the requirement of maintaining a secure system. The use of implicit access rules (i.e., rules derived from others using data associations) makes this security model unique, and its representation in a formal model is of interest.

Keywords

Authorization models, formal methods, object-oriented databases, Z

1 INTRODUCTION

Formal methods are increasingly being used in the development of software systems. The use of formal methods can reduce misunderstandings among different groups of software developers and across different stages of the development process, thus improving the quality of the software product and process. Formal methods utilize formal specification techniques to precisely state and analyze system behaviors. The ability to formally prove properties of systems increases confidence in a system's ability to satisfy security and other critical requirements. The specification and analytical strengths of formal methods can be put to good use in the development and analysis of software systems that are of a critical nature (i.e., systems in which failures may result in social and/or economic catastrophes) and/or have stringent security requirements.

D. L. Spooner et al. (eds.), *Database Security IX*

There are two basic types of security models for databases: *multilevel models* (also called *mandatory models*), and *authorization models* (also called *discretionary models*). The multilevel approach models a system where data and users are assigned to different security levels, and where different types of data access are strictly constrained in certain levels. For example, any write operation can only apply to the data whose security levels are equal to or higher than the writer's level, and any read operation can only apply to the data whose security levels are equal to or lower than the reader's level. The authorization models describe a system by a set of access rules which specify users' access rights. Some of the access rules are written directly by the system's security administrator, while some others may be generated automatically according to specific security policies and run time information. The authorization models are more flexible than the multilevel models, but they tend to be less secure; formal assurance of their policies is therefore of practical value. Clearly, specifying precisely authorization policies is just a first step, there is also a need to apply assurance models to the lower implementation levels; however, we must start from the top level which is our emphasis here.

In [5], a method-based authorization model for object-oriented databases is presented. This model is characterized by the use of hierarchical data structuring to define implicit access rules. This approach was shown to be a powerful model to define authorization which is also very convenient for the security administrator [4, 5]. To make that model more precise, we need to formalize its authorization policies. This is the topic of this paper. We are not trying to say here that these specific policies are the most convenient; rather, our intent is to show that complex authorization policies can be specified precisely using a formal language and the value of doing this.

An authorization model similar to the one considered here was partially formalized in [4] using a set theoretic approach. It describes a database as a directed acyclic graph and uses this graph to model security policies for generalization that define implied accesses along the data class/subclass hierarchy. [14] discusses authorization for distributed systems using groups of objects described using Z [19]. In [13], a Z specification is presented for a multilevel security model of an Integrated Programming Support Environment. Other applications of formal methods to security models can be found in [1, 10, 11], all of which describe multilevel models. In [16] a discretionary model is expressed in Z to discuss computational complexity aspects of authorization. In [8] a Z specification of the UNIX file system, including a definition of security-related file operations, is presented. Semmens and Allen discussed the relationships among entity relationship models, data flow diagrams, and Z descriptions [18]. An example in their paper is the formalization of some security aspects of a computer system using a multilevel model. [2] describes a Z specification of a security model; although it is indicated that this approach covers both mandatory and discretionary policies, only details of the mandatory model are shown. In summary, most of these papers consider only multilevel models; only [4, 8, 14] discuss authorization models, but [8] is concerned with file systems, not database systems, [4] uses a simplified model, [14] does not consider implied or negative rules, or object-oriented databases, and [16] considers only theoretical aspects.

We present here a different style of object-oriented database authorization specification expressed in the formal language Z. One of the characteristics of our approach is the factoring out of several important sets containing security information; for example, an element in our

specification is the set *auth_rules* which contains all the access rules of the database (implicit or explicit). Such sets are abstract in the sense that there may not be a corresponding concrete data structure in the implementation of the database.

In this paper we assume that the database has already been defined and we describe the structure and the properties that must be satisfied by each authorization rule if the system is to be secure. In this way, each new access rule can be tested for compliance with the requirement of maintaining a secure system. The use of implicit access rules (i.e., rules derived from others using data associations [6]) makes this security model unique, and its representation in a formal model is of interest. In fact, this is the first complex authorization model described formally that we know of; as indicated earlier most of the formal descriptions reported in the literature are for multilevel models or for operating systems.

This paper consists of four sections. Section 2 presents some background information on authorization models and the language Z. Section 3 gives the formal specifications of the object-oriented database and the authorization policies. Section 4 gives conclusions and suggests some future work.

2 BACKGROUND

We present here the background material needed to understand concepts used in this paper. This includes a model for database authorization, and some general aspects of the language Z. We assume the reader to be familiar with basic concepts of object-oriented systems (e.g., see [17]); some of these concepts are defined formally as part of our model.

2.1 Authorization models

Security is the protection of data (information) against unauthorized disclosure, alteration, or destruction, or against attempts to deny services to authorized users [6]. Some aspects of security are enforced by *access rules* (also called *authorization rules*), which specify who has what kind of access to what data. An *authorized action*, such as displaying certain data items, abides by these rules. People who write and maintain the access rules are called security administrators.

An access rule has the following structure:

(subject, access_type, data_object)

where *subject* is the identifier of a user accessing the database; *access_type* is the intended mode of access, for example, read, write, scan, delete; and *data_object* is the data item whose access is controlled by this rule.

Access validation implies comparing a user's data access request with the corresponding access rules to determine if the requested access is authorized or not. A user's request has the same components as an access rule and a matching rule validates the requested type of access.

Access rules can be *explicit* or *implicit.* Explicit access rules are written by security administrators and they are stored in some kind of data structure; implicit access rules are not stored, (depending on the implementation, some of the implicit access rules may be stored for quicker authorization validation), they are determined at user request validation time by the authorization policies, the data structure, and run time information. The introduction of implicit access rules saves the writing of a large number of rules and simplifies the authorization view for the security administrator. In [4, 5] we make use of the class/subclass hierarchy to imply access rules. Other models use the data aggregation structure to imply rights.

Access rules can also be negative, which means that the rule specifies that a subject cannot access certain data. Negative access rules can be used to specify exceptions to granted rights.

In [5], a method-based authorization model for object-oriented databases is presented. In this model, access rules are defined with respect to methods of classes. Access rules then take the following form:

(subject, (method, class))

where subject is the same as before, and the *(method, class)* pair contains the method which performs some kind of access to the data to be controlled and the class that includes the method. The class associations can be used to imply access rules; for example, if we have a class/subclass hierarchy where *person* has the subclasses *Student* and *Faculty*, and *Student* has the subclass *Foreign_Student*, the explicit rule $(u_1, (add, Person))$ implies the rules: $(u_1, (add, Faculty))$, $(u_1, (add, Student))$, $(u_1, (add, Foreign\_Student))$.
An explicit negative rule, for example, $(u_1, (\neg add, Student))$, would stop user u_1 from adding *Students* or *Foreign_Students* (the latter because of the implicit rule $(u_1, (\neg add, Foreign\_Student))$). Negative rules can be seen as cancelling corresponding positive rules.

2.2 The language Z

Z is a language as well as a style for expressing formal specifications of computing systems [19]. It is based on typed set theory, and the *schema* construct is its key structuring mechanism. Schemas can be used to capture both static and dynamic aspects of systems. Static aspects are defined in terms of states, where a state of a system is a collection of system components. State schemas are used to declare and express constraints on state components. Dynamic aspects are expressed in terms of operations which are described by operation schemas.

In a Z specification, *basic* types are used to build more complex types referred to as *derived* types. In this paper, we use capital words to represent basic types and title words (words that begin with a capital letter) to represent derived types. The following is an example of a state schema taken from Section 3.1:

Class

name : *CLASS_NAME*
attributes : ℙ *ATTRIBUTE*
methods : ℙ *METHOD*

This is the declaration part of a state schema that defines a derived type *Class*. Three components are defined for *Class*: a *name* of type *CLASS_NAME*, a set *attributes* consisting of elements of type *ATTRIBUTE*, and a set *methods* consisting of elements of type *METHOD* (ℙ indicates a set).

Constraints on state components and specifications of operations are expressed using predicate logic and set notation. The notation *Schema_name.component_name* refers to the *component_name* component of a schema *Schema_name*. For example, *Class.name* refers to the *name* component in schema *Class*. The notation $\forall x : T \mid P \bullet Q$ means all x of type T satisfying P must also satisfy Q. A similar structure is used with the existence symbol, $\exists$. Operators such as $\in$, $\cup$, $\wedge$, $\vee$ and $\Leftrightarrow$, have their usual set theory and predicate logic interpretations.

An operation can either change the state of a system or it can leave the state unchanged when executed. If a state variable in an operation schema is preceded by a Ξ in the declaration part it means the operation does not change the state when executed. If it is preceded by a Δ it means that the operation will change the state when executed. The value of a state component, x, after a state change is denoted by x' in an operation schema. In an operation schema, variable names that end with question marks (?) represent inputs to the operation, while those ending with exclamation marks (!) represent operation outputs.

3 FORMAL DEFINITION OF AUTHORIZATION POLICIES

3.1 Object-oriented database systems

We declare the following basic types for the database:

$[USER, CLASS\_NAME, ATTRIBUTE, METHOD]$

where:

- *USER*: a set of user identifiers,
- *CLASS_NAME*: a set of class names,
- *ATTRIBUTE*: a set of attributes,
- *METHOD*: a set of methods.

The following free types are also used in our formalization:

ASSOCIATION_TYPE ::= *generalization* | *composition* | *relationship*
EVALUATION_RESULT ::= *granted* | *denied*

where *ASSOCIATION_TYPE* is a set (representing object modeling association types) consisting of three distinct values (i.e., no two are equal), *generalization*, *composition*, and *relationship* (the symbol ::= relates the name of a free type to its distinct values), representing used by OMT [17] and other object-oriented models, and *EVALUATION_RESULT* is the result of access request validation.

3.1.1 Class structure

Classes are the basic units that embed data and operations. A class has a name, a set of attributes and a set of methods. We define class as a derived type with the following schema.

Class

$$
\begin{aligned}
&name : CLASS\_NAME \\
&attributes : \mathbb{P}\ ATTRIBUTE \\
&methods : \mathbb{P}\ METHOD
\end{aligned}
$$

3.1.2 Access rule

In most object-oriented databases, a method's name is not enough to identify a specific method in a system because different classes may have similarly-named methods.

Because methods are the only path to data access, it is important that we have unique representations for each uniquely-interpreted method. We consider two ways in which a method can appear in a class: D, which means the method is *defined* in the class, and I which means the method is inherited or propagated from another class. We can express the above as a free type in Z as follows:

$$M\_type ::= D \mid I$$

A method is uniquely determined by its name, the class it belongs to and the manner in which it appears in the class. This unique representation of a method is captured by the schema *Acc_method*:

Acc_method

$$
\begin{aligned}
&method : METHOD \\
&class : Class \\
&mtype : M\_type \\
\hline
&method \in class.methods
\end{aligned}
$$

The constraint in *Acc_method* stipulates that the *method* component be a method of the *class* component.

The *(method, class)* pair of the access rule defined in Section 2 contains information about the access type and the data to be protected by this rule. The *method* part may have the

special value *all*, indicating that the subject has the right to use every method in this class. We modify the form of access rules as follows:

$$(subject, Acc\_method)$$

In a database system, each access method (element of type *Acc_method*) must refer to classes within the database system. This constraint is expressed in Z as follows:

DBAccess

$$classes : \mathbb{P}\ Class$$
$$acc\_methods : \mathbb{P}\ Acc\_method$$

$$\forall a : acc\_methods \bullet a.class \in classes$$

As discussed in section 2.1, there can be explicit and implicit access rules. Explicit rules may define, according to the authorization policies, some implicit rules. The specific implicit rules are defined based on the class structure and the type of association involved. When a rule specifies that a user can apply a specific method, it is a *positive access rule*; when it specifies that a user cannot apply a specific method, it is a *negative access rule*. Thus, there can be four types of access rules that correspond to the four combinations of these two types of rules. Rule types are pairs in which the first element indicate whether the rule is positive or negative, and the second element indicates whether the rule is explicit or implicit. This is expressed in Z as follows:

$$PosNeg ::= Pos \mid Neg$$

$$ExIm ::= Ex \mid Im$$

$$Rule\_type == PosNeg \times ExIm$$

With these definitions, the schema for access rules is as follows:

Auth_rule

$$subject : USER$$
$$acc\_method : Acc\_method$$
$$rule\_type : Rule\_type$$

An access rule of type *Auth_rule* has three components: a *subject*, which is the user whose access rights are being defined by the rule; an *acc_method* that represents the protected data and some specific operation, and a *rule_type*.

In a database, positive and negative rules exist. The following schema stipulates the constraints on these rules in the database:

Rules

$$
\begin{array}{l}
auth\_rules : \mathbb{P}\ Auth\_rule \\
pos\_rules, neg\_rules : \mathbb{P}\ Auth\_rule \\
\hline
auth\_rules = pos\_rules \cup neg\_rules \\
\forall a : pos\_rules \bullet first(a.rule\_type) = Pos \\
\forall a : neg\_rules \bullet first(a.rule\_type) = Neg \\
\forall a : neg\_rules \bullet \neg (\exists ar : pos\_rules \bullet \\
\qquad a.subject = ar.subject \wedge a.acc\_method = ar.acc\_method)
\end{array}
$$

The first constraint in the above schema states that the set *auth_rules* is the collection of all positive and negative rules defined for the database. The second constraint states the first element in a positive rule (which is a pair of elements; see above) must be the value *Pos* and the third constraint states that the first element in a negative rule must be the value *Neg*. The last constraint states that the subject that is denied access to a method through a negative rule cannot be permitted access to the same method by a positive rule.

3.1.3 Object-oriented database system

In our model a database consists of a set of *users*, *data* and *access rules*. The data is structured into a set of related *classes*. Other aspects of a database system which are not relevant to authorization are ignored here. As indicated earlier, *Acc_method* provides a unique representation for every method in a database. In the state schema *O_o_database* describing an object-oriented database we include a set, *acc_methods*, that contains all the access methods (elements of type *Acc_methods*) defined for the database.

O_o_database

$$
\begin{array}{l}
DBAccess \\
Rules \\
users : \mathbb{P}\ USER \\
\hline
\forall a : auth\_rules \bullet a.subject \wedge a.acc\_method \in acc\_methods
\end{array}
$$

3.1.4 Access validation

Access validation is a procedure to determine whether a user's access request can be granted or not. A user's request is similar in form to an access rule (except there is no *rule_type* in a request, all requests are considered explicit and positive).

User_request

$$
\begin{array}{l}
subject : USER \\
acc\_method : Acc\_method
\end{array}
$$

Access_validation

$\exists O\_o\_database$
user_request? : *User_request*
result! : *EVALUATION_RESULT*

$(\exists\, ar : pos\_rules \bullet ar.subject = user\_request?.subject \wedge$
$\quad ar.acc\_method = user\_request?.acc\_method) \Leftrightarrow result! = granted$
$\wedge$
$\neg\, (\exists\, ar : pos\_rules \bullet ar.subject = user\_request?.subject \wedge$
$\quad ar.acc\_method = user\_request?.acc\_method) \Leftrightarrow result! = denied$

The *Access_validation* operation indicates that an access is authorized (*granted*) if and only if the requested method and the requesting user match the access method and the subject of a positive access rule, otherwise the operation indicates that access is not permitted (*denied*). Note that matching applies to explicit and implicit rules.

3.1.5 Class associations

Here we describe the existing structure of the data in order to determine possible implications of the access rules defined explicitly. We are not trying to indicate how to create these data structures or to indicate their correctness properties.

Implicit access rules are determined partly by the class structure of the database system. Class associations can be classified as *generalizations*, *compositions*, and *relationships* [17]. We model associations between classes as follows:

$$association : Class \times Class \nrightarrow \mathbb{P}\, ASSOCIATION\_TYPE$$

The following are the definitions of the class associations we use in this paper:

- If class2 is a child of class1 in a generalization association, then $generalization \in association(class1, class2)$.
- If class2 is a child of class1 in a composition association, then $composition \in association(class1, class2)$.
- If there is relationship between class2 and class1, then $relationship \in association(class2, class1)$.

Generalization and composition associations are transitive, and relationship associations are symmetric. There are four possible transitive closures for class associations:

- *gen_descendant*: the set consisting of the descendants of a class in a generalization hierarchy.
- *gen_ancestor*: the set consisting of the ancestors of a class in a generalization hierarchy.

- *com_descendant*: the set consisting of the descendants of a class in a composition hierarchy.
- *com_ancestor*: the set consisting of the ancestors of a class in a composition hierarchy.

The schema *Associated_classes* defines these transitive closures and the symmetric relationship association.

Associated_classes

$$
\begin{array}{l}
O\_o\_database \\
gen\_descendant, gen\_ancestor, \\
com\_descendant, com\_ancestor : Class \pfun \mathbb{P}\ Class \\
association : Class \times Class \pfun \mathbb{P}\ ASSOCIATION\_TYPE \\
\hline
\forall c1, c2 : classes \bullet \\
\quad (c2 \in gen\_descendant(c1) \Leftrightarrow generalization \in association(c1, c2) \lor \\
\qquad (\exists c : classes \bullet c2 \in gen\_descendant(c) \land \\
\qquad\quad generalization \in association(c1, c))) \\
\forall c1, c2 : classes \bullet \\
\quad (c2 \in gen\_ancestor(c1) \Leftrightarrow generalization \in association(c2, c1) \lor \\
\qquad (\exists c : classes \bullet c2 \in gen\_ancestor(c) \land \\
\qquad\quad generalization \in association(c, c1))) \\
\forall c1, c2 : classes \bullet \\
\quad (c2 \in com\_descendant(c1) \Leftrightarrow composition \in association(c1, c2)\ lor \\
\qquad (\exists c : classes \bullet c2 \in com\_descendant(c) \land \\
\qquad\quad composition \in association(c1, c))) \\
\forall c1, c2 : classes \bullet \\
\quad (c2 \in com\_ancestor(c1) \Leftrightarrow composition \in association(c2, c1) \lor \\
\qquad (\exists c : classes \bullet c2 \in com\_ancestor(c) \land \\
\qquad\quad composition \in association(c, c1))) \\
\forall c1, c2 : classes \bullet \\
\quad relationship \in association(c1, c2) \Leftrightarrow relationship \in association(c2, c1)
\end{array}
$$

3.2 Authorization policies

As shown earlier, implicit access rules play important roles in the validation of users' access requests. The following authorization policies use the class structure to define implicit access rules. Because implicit accesses propagate down the hierarchy we start with a policy to control this propagation.

3.2.1 Propagation control for generalization

A negative rule cancels a corresponding positive rule. Negative rules also propagate, that is, a user (*subject*) that has explicit prohibited access to a given method in a class also has prohibited access to the corresponding inherited method in a subclass of that class. Schema

$P0$ defines this propagation. $P0$ is used to control the propagation of positive rules defined by the policy $P1$ below.

$P0$

Associated_classes

$$\forall\, ar : neg\_rules;\ ari : Auth\_rule \mid ari.subject = ar.subject \wedge$$
$$(ari.acc\_method).method = (ar.acc\_method).method \wedge$$
$$(ari.acc\_method).mtype = I \wedge$$
$$(ari.acc\_method).class \in gen\_descendant((ar.acc\_method).class) \wedge$$
$$ari.rule\_type = (Neg, Im) \bullet ari \in neg\_rules$$

3.2.2 Implied authorization in generalization

A subject authorized to apply a given method to a class has the same right with respect to the corresponding inherited method in a subclass of that class. The propagation of an implied inherited authorization for a method can be stopped by a rule specifying no access to that specific method (as defined by policy P0).

$P1$

$P0$

$$\forall\, ar : pos\_rules;\ ari : Auth\_rule \mid ari.subject = ar.subject \wedge$$
$$(ari.acc\_method).method = (ar.acc\_method).method \wedge$$
$$(ari.acc\_method).mtype = I \wedge$$
$$(ari.acc\_method).class \in gen\_descendant((ar.acc\_method).class) \wedge$$
$$ari.rule\_type = (Pos, Im) \wedge \neg\, (\exists\, a : neg\_rules \bullet$$
$$a.acc\_method = ari.acc\_method \wedge a.subject = ari.subject) \bullet$$
$$ari \in pos\_rules$$

This schema establishes that a positive rule ar defines a set of implicit positive rules ari which have the same subject as ar, methods inherited from ar, and apply to classes which are generalization descendants of the class to which ar applies. These implied rules can be cancelled by negative rules with the same subject and access method. The effect of an explicit negative rule and its implicit propagated rules is to close a subtree of the database to someone that has been given access to a tree including this subtree.

3.2.3 Class access

Access to a complete class implies the right to apply all the methods defined in the class as well as those inherited from higher classes (access is subject to the influence of negative rules).

P2

O_o_database

$$\forall\, ar : pos\_rules;\ ari : Auth\_rule \mid$$
$$(ar.acc\_method).method = all \wedge ari.subject = ar.subject \wedge$$
$$(ari.acc\_method).method \in (ar.acc\_method).class.methods \wedge$$
$$\neg\, (\exists\, a : neg\_rules \bullet a.acc\_method = ari.acc\_method \wedge$$
$$a.subject = ari.subject) \bullet ari \in pos\_rules)$$

Policy P2 can be viewed as the definition of class access (as opposed to method access).

3.2.4 Propagation control for aggregation

In aggregations only some methods propagate to the components. Similarly to generalization we need to control this propagation. Prohibited access to a method in a composite object implies prohibited access to the corresponding method in components of the objects.

P3

Associated_classes

$$\forall\, ar : neg\_rules;\ ari : Auth\_rule \mid ari.subject = ar.subject \wedge$$
$$(ari.acc\_method).method = (ar.acc\_method).method \wedge$$
$$(ari.acc\_method).mtype = I \wedge$$
$$(ari.acc\_method).class \in com\_descendant((ar.acc\_method).class) \wedge$$
$$ari.rule\_type = (Neg, Im) \bullet ari \in neg\_rules$$

3.2.5 Propagation for aggregation

Access of some type to an object implies similar type of access for the components of the object where this type of access normally propagates. A negative access rule can stop the propagation of implied access in aggregations.

P4

P3

$$\forall\, ar : pos\_rules;\ ari : Auth\_rule \mid ari.subject = ar.subject \wedge$$
$$(ari.acc\_method).method = (ar.acc\_method).method \wedge$$
$$(ari.acc\_method).mtype = I \wedge$$
$$(ari.acc\_method).class \in com\_descendant((ar.acc\_method).class) \wedge$$
$$ari.rule\_type = (Pos, Im) \wedge$$
$$\neg\, (\exists\, a : neg\_rules \bullet a.acc\_method = ari.acc\_method \wedge$$
$$a.subject = ari.subject) \bullet ari \in pos\_rules$$

3.2.6 Composite objects

Access to all the methods of a class implies the right to apply any method in the component classes (subject to negative rules).

$P5$

Associated_classes

$$\begin{aligned}
&\forall\, ar : pos_rules;\ ari : Auth_rule \mid (ar.acc_method).method = all \wedge \\
&\quad ari.subject = ar.subject \wedge \\
&\quad ((ari.acc_method).method \in (ar.acc_method).class.methods \vee \\
&\qquad (\exists\, c : classes \mid c \in com_descendant((ar.acc_method).class) \bullet \\
&\qquad\quad (ari.acc_method).method \in c.methods)) \wedge \\
&\quad \neg\, (\exists\, a : neg_rules \bullet a.acc_method = ari.acc_method \wedge \\
&\qquad a.subject = ari.subject) \bullet ari \in pos_rules)
\end{aligned}$$

Using the above policies, a secure database state can be defined as follows:

$$Secure\_O\_o\_database \mathrel{\widehat{=}} P1 \wedge P2 \wedge P4 \wedge P5$$

The schema *Secure_O_o_database* defines precisely the positive and negative rules (both explicit and implicit) that can coexist at any given time in a secure database. Any newly defined rule must satisfy this condition to be accepted as a permanent rule. New rules can be validated against this schema. Validation of new rules also requires that they are shown not to violate enterprise policies, for example, an employee cannot have more rights than those required to perform his job functions [15].

4 CONCLUSIONS AND FUTURE WORKS

We have presented a formal specification of implied authorization policies for object-oriented databases. A key point of our specification is the definition of a set *pos_rules* which contains all the authorized accesses of the database system. This set denotes a virtual existence because there may not be a corresponding data structure in an actual system and some of the rules may only be implicit. The derived type *Auth_rule* specifies the format of the elements in *pos_rules* while the security policies specify the relative properties of each element in this set; for example, some of these rules are explicit while others are implied by them in specific ways, according to the structuring of the data. When an access rule is issued by the security administrator, a corresponding explicit authorization rule and a set of implicit rules are added into the set *pos_rules*; when an explicit rule is deleted, the corresponding implicit authorization rules also disappear from the set *pos_rules*. The security policies specify the relationship between the defined access rules and the implied ones. Except for the moment when one of the *pos_rules* is being modified, at any other time during the lifetime of the database system, it is necessary for each of its access rules to satisfy these specifications in order for the database to be secure. This can be used to validate new rules, which appears

to be more efficient than validating existing rules as is done in other systems, for example, see [9].

This work is the first formalization of a complex object-oriented authorization model that we are aware of. As discussed earlier, all the existing security models either apply to multilevel databases (which have a simpler security policy), or to file systems (which have a simpler data semantics).

These specifications can be used in the design of secure database systems. the policies define a precise model of user access, the lower levels must assure in turn that this model is enforced.

It remains to include database administration aspects into our formal definition. These aspects include the specification of security administrator operations and the associated implied access rules. We have formalized elsewhere the grouping of subjects where rights are implied by this grouping [7]. More work is needed to show specific specific ways to use these specifications in the design of secure systems, that is, their combination with lower-level mechanisms.

There are several proposed extensions to Z to define object-oriented systems, e.g. see [3, 12]. The extensions incorporate the semantics of some object-oriented models into Z. An interesting study would be to reformulate our specifications using some of these languages. Since they may have more generality than we need for this purpose, it is not clear that those specifications will be simpler. However, they should be easier to extend to consider related policies.

References

[1] D. M. Berry. Towards a formal basis for the Formal Development Method and the Ina Jo specification language. *IEEE Trans. on Software Engineering*, SE-13(2), Feb. 1987, pages 184-201.

[2] A. Boswell. Specification and validation of a security policy model. *IEEE Trans. on Software Engineering*, SE-21(2), Feb. 1995, pages 63-68.

[3] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In *Formal Description Techniques II.* S. T. Vuong (ed.), Elsevier Science Pub. B. V. (North Holland), 1990, pages 281-296.

[4] E. B. Fernandez, E. Gudes, and H. Song. A model for evaluation and administration of security in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering*, 6(2), April 1994, pages 275 - 292.

[5] E. B. Fernandez, M. M. Larrondo-Petrie, and E. Gudes. A method-based authorization model for object-oriented databases. In *Proc. of the OOPSLA 1993 Workshop on Security in Object-Oriented Systems*, 1993.

[6] E. B. Fernandez, R. C. Summers, and C. Wood. *Database Security and Integrity.* System Programming Series, Addison-Wesley, Reading, Massachusetts, 1981.

[7] R. B. France, E. B. Fernandez, and D. Wei. A formal specification of a user group model for data security. Technical report, Florida Atlantic University, 1995.

[8] M. Henning and A. Rohde. On the security of Z for the specification of verifiably secure systems. In *Computer Security in the Age of Information.* W. J. Caelli (ed.), Elsevier Science Publisher B. V. (North-Holland), 1989, pages 197 - 221.

[9] A. Heydon and J. D. Tygar. Specifying and checking UNIX security constraints. *Computing Systems*, The Journal of the USENIX assoc., 7(1), Winter 1994, pages 91-112.

[10] J. Jacob. A uniform presentation of confidentiality properties. *IEEE Trans. on Software Engineering*, 17(11), Nov. 1991, pages 1186 - 1194.

[11] C. E. Landwehr. Formal methods for computer security. *ACM Comp. Surveys*, 13(3), Sept. 1981, pages 247 - 278.

[12] K. Lano. Z++, an object-oriented extension to Z. In *Proc. of the Z User Workshop, Oxford 1990.* J. E. Nicholls (ed.), Springer- Verlag, 1990, pages 151-172.

[13] J. A. McDermid and E. S. Hocking. Security policies for integrated support environments. In *Database Security, III: Status and Prospects.* D. L. Spooner and C. Landwehr (Eds.), Elsever Science Publ. B. V., 1990, pages 41-74.

[14] J. D. Moffet. *Delegation of authority using domain-based access rules.* PhD thesis, Imperial College of Science, Technology and Medicine, London, U.K., 1990.

[15] J. D. Moffet and M. S. Sloman. The source of authority for commercial access control. *Computer*, 21(2), Feb. 1988, pages 59-69.

[16] G. O'Shea. On the specification, validation and verification of security in access control systems. *The Computer Journal*, 37(5), 1994, pages 437-448.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[18] L. Semmens and P. Allen. Using Yourdon and Z: An approach to formal specification. In *The Proceedings of the Z User Workshop.* J. E. Nicholls (ed.), Oxford, 1991, Springer-Verlag, New York, 1990, pages 228-253.

[19] J. M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

BIOGRAPHIES

Eduardo B. Fernandez is a professor in the Department of Computer Science and Engineering at Florida Atlantic University in Boca Raton, Florida. He has worked at the NASA Satellite Tracking Station in Santiago,Chile , taught at the University of Chile and the University of Miami, and researched the security and performance aspects of database systems at the Los

Angeles Scientific Center of IBM. His current research interests are Object-oriented system design , database security ,and fault-tolerant systems.

Eduardo Fernandez holds a MS degree in Electrical Engineering from Purdue University and a Ph.D.in Computer Science from UCLA. He has authored three books and a large number of papers. He is a Senior Member of IEEE and a member of ACM. He has consulted for several companies,including IBM, Bendix, Motorola, and Harris.

Robert B. France is an Assistant Professor in the Department of Computer Science and Engineering at Florida Atlantic University. From 1990 to 1992 he was a Research Associate in the University of Maryland's Institute for Advanced Computer Studies, where he carried out research on systematic software reuse and software reengineering. His current interests include systematic software reuse, formal specification techniques, requirements engineering, and software design techniques.

Robert France holds a BSc (First Class Honours) in Natural Sciences from the University of the West Indies, and a PhD in Computer Science from Massey University, New Zealand.