A Taxonomy for Secure Object-Oriented Databases

Martin S Olivier and Sebastiaan H von Solms
Department of Computer Science
Rand Afrikaans University
Johannesburg *

(Published in *ACM Transactions on Database Systems*, **19**, 1, 3–46, 1994)

Abstract

This paper proposes a taxonomy for secure object-oriented databases in order to clarify the issues in modelling and implementing such databases. It also indicates some implications of the various choices one may make when designing such a database.

Most secure database models have been designed for relational databases. The object-oriented database model is more complex than the relational model and object-orientation is not based on a formal (mathematical) model like the relational model. For these reasons, models for secure object-oriented databases are more complex than their relational counterparts. Furthermore, since views of the object-oriented model differ, each security model has to make some assumptions about the object-oriented model used for its particular database.

A number of models for secure object-oriented databases has been proposed. These models differ in many respects, because they focus on different aspects of the security problem, or because they make different assumptions about what constitutes a secure database or because they make different assumptions about the object-oriented model.

The taxonomy proposed in this paper may be used to compare the various models: Models that focus on specific issues may be positioned in the broader context with the aid of the taxonomy. The taxonomy also identifies the major aspects where security models may differ and indicates some alternatives available to the system designer for each such design choice. Lastly, we show some implications of using specific alternatives.

Since differences between models for secure object-oriented databases are often subtle, a formal notation is necessary for a proper comparison. Such a formal notation also facilitates the formal derivation of restrictions that apply under specific conditions. The formal approach also gives a clear indication about the assumptions

*Address of authors: Department of Computer Science, Rand Afrikaans University, PO Box 524, Auckland Park, Johannesburg, 2006 South Africa; Email: molivier@rkw.rau.ac.za, basie@rkw.rau.ac.za

made by us—given as axioms—and the consequences of those assumptions (and of design choices made by the model designer)—given as theorems.

Keywords: Multilevel secure databases, Information Security, Formal Security Models, Object-orientation

1 Introduction

A number of models for multilevel secure object-oriented databases has been proposed. The variety exhibited by the proposed models is an indication of the great number of possibilities that exist.

In this paper, rather than proposing another model, we approach the design of a secure object-oriented database by structuring the issues that should be considered when such a database is designed. This makes it possible to compare the existing models in a structured way to highlight the real differences. Specific choices for the discussed issues influence choices elsewhere in a given model: the structured approach allows one to consider these effects systematically. A number of these effects are given and proven in this paper. The structured approach further allows researchers to focus on specific issues, rather than on a whole model at a time. Lastly, the described structure may be used to generate new models for secure object-oriented databases; this has been done in [15] and [18].

Unfortunately, classifying such models is not an easy task. As Sandhu put it [20]: "The underlying assumptions adopted by each one and their motivating forces are somewhat different. This makes a relative comparison difficult since different assumptions and motivations inevitably lead to different design trade-offs." This does mean that the taxonomy proposed by us may not be universally accepted. However, if this paper prompts someone to design a model that does not fit the proposed taxonomy, the taxonomy also serves a purpose.

The paper by Varadharajan and Black [25] follows a similar approach to this paper: they indicate some issues that are relevant when designing a secure database model and also consider some of the alternatives that are available for each issue. However, their approach is less formal and less general than ours.

The next section introduces existing work on security models for object-oriented databases, before the taxonomy is described in the following sections. Section 3 describes the deductive approach we follow: definitions and axioms are given from which theorems regarding secure database models will be proven. Section 4 contains a structured list of the parameters available to the designer of a secure database model. Further, the axioms and definitions given earlier are used to derive the implications and limitations of choices available to the designer. Section 5 places the modeling stage (addressed by this paper) into context by looking at other stages in the development of a secure database. Section 6 discusses remaining (complex) issues not covered by the current paper, such as polyinstantiation and inference.

The primary original content of this paper is its structured approach to the design of secure object-oriented database models, simplifying the generation of new models and putting other research efforts into (a) context. In addition, the restrictions for

classification of related entities (see parameter X2.3 later) are more comprehensive than restrictions given by other authors. Further, the notation employed by us to describe dynamic labeling (see parameter X3 later) has not been used for this purpose before and some of the possibilities we give for this parameter are also original.

2 Security in object-oriented databases

The following security models serve as examples for our discussion. Here we only mention the models; specific issues will be addressed when the design parameters are discussed.

- SODA [7, 8] is a model for a secure database based on a general object-oriented model. Objects or instance variables are assigned ranges of sensitivity levels. Subjects are assigned clearance levels. Every message that travels through the system, carries with it a current sensitivity level and a clearance level. The current sensitivity level is adjusted whenever an object or variable with a higher sensitivity is accessed. Rules, based on the current sensitivity level and the clearance level of a message and on an object's or variable's sensitivity level, determine whether the method should be permitted access to an object or variable.
- SORION [24] is based on the ORION object-oriented data model. Entities in the system (subjects, objects, variables, messages, etc) are assigned security levels. An extensive list of properties is given constraining assignment of levels to entities. A given security policy constrains access to protected items based on these security levels.
- Lunt has given some initial properties for a multilevel object-oriented database system [11]; here we will refer to these properties as the Lunt model. The properties specify minimum requirements for a secure database based on a general object-oriented data model.
- Views [6] allows an object to display different "views" of itself to other objects. This is done by restricting from which other objects any given method may be invoked. An alternative approach for defining views is given in [21]: multiple interfaces may be defined for a single object. A 'client' object may then decide through which view it wants to access the object. The first approach is more applicable for our purposes since it is precisely specified through which interface a given client object is permitted to access the protected object. Note that the use of views in relational databases for security is well-known, see for instance [12].
- The access control language of Mizuno and Oldehoeft [14] is based on extended access control lists (ACLs): a four-tuple ACL entry specifies (1) which user may activate a method, (2) through which class the request may come, (3) through which specific object (class instance) the request may come and (4) lastly names the protected method.
- SECDB [17] is based on the Path Context Model (PCM) [2, 3, 4, 16]. A profile object is associated with every protected entity. A request collects baggage as

it moves through the system. Before a request is allowed to access an entity, the baggage carried by the request is considered by the entity's profile object; based on this baggage, the request is either allowed to proceed, or rejected. When a protected entity is accessed, that entity's profile is tagged to any messages subsequently sent. Rules specify how the protection of other entities are influenced when they are accessed by a message that has such associated profiles.

- In the law-based approach as proposed by Minsky [13] a set of Prolog rules or laws may be specified to manage the exchange of messages in the system. This approach is intended to describe general aspects of object-orientation such as inheritance; it can be used to describe security restrictions, but is too general to be practical for this application. We include it nonetheless because the ability to give constraints based on logic rules is useful, amongst other to control the assignment of sensitivity levels and to ensure integrity in general.

3 Approach

Our goals are twofold. Firstly we want to give a classification structure that will enable one to compare different models for secure object-oriented databases. Secondly, we want to indicate some implications of the possible choices one can make.

To realise the first goal, we give a number of design parameters—issues one should consider when designing a model. We give eight such design parameters that represent the major issues for consideration. These eight parameters are grouped into three categories:

X1 Labeling semantics: *Underlying model* and *Protection interpretation*;

X2 Structural labeling: *Protectable entities*, *Label instantiation* and *Relationship restrictions*; and

X3 Dynamic labeling: *Authorisation flow*, *Sensitivity flow* and *Information flow restrictions*.

For most of these design parameters we list a number of alternatives that are available. These parameters are described in detail in section 4.

Implications of choosing a specific alternative for a design parameter are given as theorems. Some proposed models already describe restrictions to their models [24, 11]; often it is not clear whether a model imposes such restrictions because they simplify the model in some way, or because they are necessary to ensure that security is not compromised. Further, it is not clear from those models how generally applicable such restrictions are, in other words do the restrictions still apply if some aspects of the model are modified? To address these problems, our assumptions are given as axioms; for theorems the circumstances under which they apply are given.

In order to classify and analyse models, the meaning of terms have to be clear; we describe the important terms as definitions. It is important to note that we do not claim that our definitions are the only proper definitions of the terms; nor do we claim that our classification structure gives the final word on secure object-oriented

database classification. We do claim that if our definitions fit a given model, that model may be classified using the described classification structure and, also, that the given theorems are valid for such a model. We also hope that our definitions and classification structure will prompt researchers to investigate models that do not fit these definitions or the classification structure.

We now define a secure database for the purposes of this discussion.

Definition 1 (Secure database) *An object-oriented database is secure if*

1. *No subject is able to obtain information without authorisation;*
2. *No subject is able to modify information without authorisation;*
3. *No mechanism exists whereby a subject authorised to obtain information can communicate that information to a subject not authorised to obtain it; and*
4. *No subject is able to activate a method without authorisation.*

This definition does not allow any covert channels—neither storage, nor timing channels—which may mean that it is too strict for practical use. However, these points cover the four primary issues addressed by models for a secure database. Issues (1) and (3) are normally addressed by models to enforce mandatory security, while issues (1), (2) and (4) are usually addressed by models to enforce discretionary access control (see section 6.3).

In the remainder of this section we introduce notation and list the axioms that reflect the relevant assumptions we make about object-orientation. See [26] and [9] for an introduction to the object-oriented paradigm. Our view of this paradigm comes from the Smalltalk programming language [5].

An object o is a set of facets (methods, instance variables, etc)—ie if m is a method of o, we will denote it by $m \in o$; similarly, the fact that o has an instance variable v, will be denoted by writing $v \in o$. The system consists of a set of objects. We will refer to this set as U. (More precisely, v is the name of the corresponding variable, m is the signature of the corresponding method and o additionally contains a unique identifier that distinguishes it from all other objects in the system U.)

For the purposes of this paper a class is considered as a template for its instantiations; similarly the facets of a class are viewed as templates for the facets of the instantiations of the class. Note that in many object-oriented systems (including Smalltalk, our reference paradigm for this paper) classes are also objects (they are instances of metaclasses) and are therefore also members of U. They also have facets (class methods and class variables) that behave like their object counterparts. If a class is indeed an object, any remarks made about objects also hold for classes. However, as stated, the term *class* in this paper is used to refer to the 'template function' of a class, to address specific issues concerning this aspect.

Let C be the set of all classes.

For any class $c \in C$, we will denote the set of superclasses of c by *sup(c)*. In the case of single inheritance, *sup(c)* will consist of a single element. For any object $o \in U$ we will denote the class of o by *class(o)*.

The assumptions we make about object-orientation are now stated as axioms.

Axiom 1 *If an object o is in the system, then the class of o is also in the system; formally*

$$o \in U \rightarrow class(o) \in C$$

Axiom 2 *Any facet x of an object $o \in U$ is also a facet of the class of o; formally*

$$x \in o \rightarrow x \in class(o)$$

Axiom 3 *If a class c has a facet x, any instance o of c will also have the facet x; formally*

$$(x \in c) \wedge (\exists o \in U)[c = class(o)] \rightarrow x \in o$$

Axiom 4 *If a class c is in the system, then all the superclasses of c are also in the system; formally*

$$c \in C \rightarrow (\forall d \in sup(c))[d \in C]$$

Axiom 5 *For any class $c \in C$, if x is a facet of a superclass of c, then x is a facet of c (it is either inherited or re-defined); formally*

$$d \in sup(c) \wedge x \in d \rightarrow x \in c$$

These axioms do not model polyinstantiation—we will return to polyinstantiation later (see section 6.1).

4 Design parameters

This section describes design parameters for a secure object-oriented database. Parameters are grouped into the three categories

- Labeling semantics;
- Structural labeling; and
- Dynamic labeling.

To avoid confusion, we will use the term *entity* to refer to a passive item in a computing system—the term *object* is usually used in the literature for such a passive item; we will use the term *object* with the meaning usually associated in the object-oriented environment. The term *subject* is used to refer to an active item. Referring to the mathematical notation introduced earlier, an entity can be any object in U; it can be any class in C; it can also be any facet of such an object or class; formally the set of entities E is defined as follows

$$E = U \cup \{\langle x, o\rangle | x \in o \wedge o \in U\} \cup C \cup \{\langle x, c\rangle | x \in c \wedge c \in C\}$$

Note that not all models allow all entities to be protected (see X2.1).

In a security model *subjects* normally send requests to *entities*. In the object-oriented environment an object is both the target for requests (ie it acts as an entity in the security sense) and an object is the issuer of requests to other objects (ie it acts as a subject). Consider any message. We will use the term *subject* to refer to

the object that sent the message under consideration or, depending on the model, to refer to the sequence of objects that were involved in the sending of the sequence of messages prior to the message under consideration; the first item in such a sequence is usually the human 'object' that initiated the chain of messages; the other items are normal objects that received a message and then sent messages to other objects. We expand on this description when we discuss the underlying model (X1.1) and the flow of authorisation (X3.1). Although the precise definition of *subject* depends on the specific security model, we will assume that it is defined and that a set S of all such subjects exists; the exact composition of S will be discussed later. The term *entity* refers to an object, method or variable (or other facet) in the role of receiving the message under consideration.

4.1 Labeling semantics

We use the term *labeling* to refer to the assignment of a security category to an item. In the case of a subject a *clearance* is usually assigned. In the case of an entity a *sensitivity* or *classification* is usually assigned.

Two aspects deserve consideration under the heading *labeling sematics*:

(X1.1) The model on which labeling is based; and

(X1.2) Exactly what is protected if an item is labeled.

X1.1 Underlying model

The model on which labeling is based falls in one of three broad categories, or a combination of some of these categories:

- *Explicit levels*: sensitivity levels are assigned to entities and clearance levels to subjects. These levels are normally integers. Rules determine when a subject may access an entity; often a subject may read an entity if the subject's clearance level dominates the entity's sensitivity level. In general, the level labels need not be integers—as long as the $\geq$ relation is defined for (some of) the labels associated with the subjects and the entities. In many models the same label act as an indication of an item's clearance when viewed as a subject and its sensitivity when viewed as an entity.
- *Access control lists* (ACLs) are lists associated with entities, containing the identities of subjects that are authorised to access the entity. Extensions of ACLs have been proposed that do not only contain the identity of authorised accessors, but also the path such a request has to follow; see for example [2, 14].
- *Capabilities* are non-forgeable identifiers possessed by subjects. Such a capability is similar to a key for a padlock: a subject will be allowed to access a protected entity only if it presents an acceptable capability.

A combination of the first two approaches is popular: Entities are classified using a sensitivity level and a category. Only subjects with a proper clearance level and belonging to the specified category are allowed to access the entity. Classifications thus

form a (partially ordered) lattice. However, most models based on this combination ignore the category aspect of the classification, and only address the (fully ordered) classification levels when developing the model.

With the underlying models in mind, we can consider the set of subjects S again. A subject may be an object (including human 'objects' using the database), a combination of objects, etc. The following are examples of subjects (ie elements of S) that may be authorised to access an entity e:

- An object with a clearance level that dominates the sensitivity level of e;
- An object in possession of a capability to access e;
- An object listed in the access control list of e; or
- An acceptable access path (as defined in PCM [3] or Mizuno and Oldehoeft's extended access control lists [14]) via which a request may reach e.

Whichever underlying model is used, it is often necessary to ensure that one entity is 'more protected' than a second entity, implying that only a subset of the authorised users of the less protected entity is allowed to access the more protected entity. In such a case we will say that the 'more protected' entity has a ***higher classification***, even if we are not using explicit levels as the underlying model. We will also say that the classification or clearance of an item *dominates* that of another item, meaning that the first has a higher (or equal) classification or clearance than the second item. To formalise this, let $subj(e)$ be the set of all subjects authorised to access e, with *subj* a function

$$subj : E \rightarrow \mathcal{P}S$$

and $\mathcal{P}S$ the powerset of S.

Note that the definition of *subj* represents a generalisation of the simple security property of the Bell-LaPadula model [1].

Without defining the sensitivity of an entity formally, we will denote the sensitivity of an entity e by $L(e)$.

Suppose that all the subjects authorised to access an entity e_1 are also authorised to access an entity e_2, in other words that $subj(e_1) \subseteq subj(e_2)$. This means that either e_1 and e_2 are equally sensitive, or an $s \in S$ exists that may use e_2 but not e_1, in which case e_1 is 'more sensitive' than e_2. We will indicate this by writing $L(e_1) \geq L(e_2)$. Formally

$$L(e_1) \geq L(e_2) \Leftrightarrow subj(e_1) \subseteq subj(e_2)$$

Let L_{low} indicate the sensitivity of an entity that may be accessed by all subjects in the system and L_{high} indicate the sensitivity of an entity that may be accessed by no subject in the system. Then, for any entity $e \in E$,

$$L_{low} \leq L(e) \leq L_{high}$$

Because this inequality holds for any entity, the *least upper bound* and the *greatest lower bound* of any set of sensitivities are defined. (In mathematical terminology, $L(E)$ is a lattice.) We will denote the least upper bound of a sequence

of entities $e_1, e_2, e_3, ..., e_n$ by $\lceil e_1, e_2, e_3, ..., e_n \rceil$, and the greatest lower bound by $\lfloor e_1, e_2, e_3, ..., e_n \rfloor$.

The function *subj* returns the subjects that may access a given entity. It is also useful to define an 'inverse' function, indicating which entities may be accessed by a given subject. For any subject $s \in S$ let *ent(s)* be the set of all entities that may be accessed by s. Formally, *ent* is a function

$$ent : S \rightarrow \mathcal{P}E$$

where $\mathcal{P}E$ denotes the powerset of E and, for all $s \in S$

$$ent(s) = \{e \in E | s \in subj(e)\}$$

X1.2 Protection interpretation

The second aspect regarding labeling semantics is the question exactly what is protected if an item is labeled. Some models attempt to protect the fact that an item exists, while others protect the contents of an item; Smith [22] identified three "dimensions of protection", where

1. The data is classified;
2. The fact that the data exists is classified;
3. The rule for classifying is itself classified.

We will refer to the models that hide the existence of data as existence protection and to models that only protect the contents against illegal access as access protection. Note that, in the case of access protection, users may know that a variable or method exists, but will get 'access denied' error messages when they try to activate a method or read or modify an instance variable without authorisation. Further note that the use of access protection presents a possible covert channel—a highly cleared subject may create an object under certain circumstances, and an uncleared subject may observe the fact that this object has been created (even though not able to access it) [10].

Definition 2 *In an* existence protected model *the fact that a labeled entity exists is hidden from unauthorised subjects.*

In an existence protected model, if the existence of one entity implies the existence of a second entity, then the sensitivity of the first entity must be at least as high as that of the second—ie

Lemma 1 *In an existence protected model, if*

$$(e_1 \in E) \wedge (e_2 \in E) \wedge (e_3 \in E) \wedge ... \wedge (e_n \in E) \rightarrow f \in E$$

and $s \in subj(e_i)$ *for every* i *then* $s \in subj(f)$.

□

If this were not the case then a subject cleared to access every e_i, but not f, will be able to infer that f exists, contrary to the definition of existence protection. This fact, together with the axioms given earlier, leads to a number of interesting theorems—see later.

To illustrate existence protection, suppose that a class EMPLOYEECLASS has a protected method INCREASESALARY; unauthorised subjects will simply not 'see' this method in the class. Since an unauthorised subject cannot determine that this method exists, it cannot possibly access it. Similarly, this class may have instances 'John' and 'James'. A subject cleared to access John's information, but not James's, will simply not know that James exists.

One advantage of existence protection is illustrated by the following example: If a subject can see that EMPLOYEECLASS has a GETUNDERCOVERASSIGNMENT method, the user will be able to infer that some employees have undercover assignments even if the user is not able to activate that method.

A disadvantage of existence protection is that a subject that does not know that an object already exists, may create it; in the example above, a subject that does not know about John's existence may insert a John object into the database—if the create request is rejected, the user will infer that the object existed, contrary to the labeling interpretation. This interpretation therefore necessitates polyinstatiation with its associated problems (see section 6.1). A similar problem occurs when a subject defines a class that already exists without the subject knowing.

Conjecture 1 *An existence protected model must support polyinstantiation.*

Lunt (amongst others) uses existence protection. Property P3 of SORION also appears to have this objective.

In some cases existence protection is not practical. One example is when sensitivity cannot be pre-determined; for example when it depends on

- Content;
- Context; and/or
- Time.

In the case of content-dependent sensitivity the sensitivity varies according to the content of the item: a salary may be classified *secret* if it is below $100 000, but *top secret* if it is above that amount. Context-dependent security addresses the aggregation problem: the sensitivity of a combination of entities are often higher than that of the individual entities. As an example, the list of employees in a firm may be unclassified, the list of salaries earned may be *secret*, but the list of employees with the salary each earns may be *top secret*. An example of time-dependent sensitivity is a military database entry containing the date on which the enemy will be attacked—it will be *top secret* up to that date but unclassified after that. In these cases existence protection may be meaningless: if a subject is allowed to access a SALARY variable in one object and the SALARY field does not exist in another similar object, it is easy to infer that the particular variable does in fact exist, but is classified. Thuraisingham [24] gives some solutions to this problem, but it still does not solve the problem that the existence of hidden items may be inferred.

Definition 3 *In an* access protected model, *an unauthorised subject is not allowed to access a protected entity; 'not allowed to access' means that*

- *Any unauthorised message sent to a protected object will fail;*
- *Any unauthorised message sent to a protected method will fail; and*
- *Any method attempting to access (read or write) an instance variable illegally will fail.*

Note that 'accessing an object' does not include passing that object as a parameter; a message that is not authorised to acces an object may pass it as a parameter when it sends a message to another object.

Some underlying models are only aimed at preventing an unauthorised subject from obtaining information from a protected entity (compare the simple security property of Bell-LaPadula [1]). Information flow restrictions (X3.3) then restrict the locations where information may be sent or written (compare the *-property of Bell-LaPadula). To accommodate these models, we allow a weak form of access protection where *access* means obtaining information with the aid of a method or *reading* a variable.

Some remarks on message failing. Note that failing does not only occur in access protection: In an existence protected model, an object may receive a message from an authorised subject and activate the corresponding method. If this method now

- Sends a message to an object that does not exist as far as the original subject is concerned;
- Sends a message to another method that does not exist as far as the original subject is concerned; or
- Accesses a variable that does not exist as far as the original subject is concerned

then, in any of these cases, can the message sent by this subject not complete.

For consistency, if a message fails for security reasons, we will assume that it behaves exactly as if it fails in the case where the object, message or variable really does not exist. One possibility is to return a special value *nil* whenever a variable is read that does not exist (or is hidden) and to ignore any attempt to write a value to a non-existent (or hidden) variable. Similarly, messages to such non-existent (or hidden) methods may just be ignored, and the value *nil* may be returned by any such message from which a return value is expected. This solution could be dangerous because errors in the database software could be overlooked; other solutions should be investigated. Further, if a variable contains a value for some subjects and *nil* for others, the model employs a form of polyinstantiation (see section 6.1)—we therefore do not devote further attention to this possibility in this paper.

Note that neither existence protection, nor access protection implies that the contents of an object can never be accessed by a subject that is not authorised to access the object: in the case of a composite object a constituent part may be independently accessable and have a different (lower) classification; it is then possible that the part may be accessed by subjects not authorised to access the composite object. This may, for example, occur where the relationship between two entities is more sensitive than

the contents of the entities themselves. This aspect will be mentioned again in the next section and discussed in the section on aggregation.

Further note that existence and access protection are not the only protection models. Other protection models worth consideration include the following:

- Use existence protection for objects, but hide classes totally from all subjects (except the database system itself which needs to access it to create instances of the class). Thus no subject can directly gain information from the class and, from there, infer information about the instances of the class. In this case the model will not attempt to limit conclusions about the class, but will attempt to limit the inferences one can make about one instance of a class by accessing another instance.
- Use access protection for classes (and therefore do not attempt to hide the structure of objects), but do hide the fact that an instance exists from a subject not authorised to access the instance.

We do not consider these possibilities further in this paper.

4.2 Structural labeling

Here we consider the influence of the structure of the data on the labeling of entities.

The object-oriented model has a rich variety of entities with relationships between such entities. For example, an object is an *instantiation* of a class; an object may be an *aggregation* or *composition* of other objects; objects *contain* variables and methods; etc. These entities and relationships describe the structure of an object-oriented data model.

Three aspects deserve attention:

(X2.1) Which entities may be labeled? Possibilities include objects, classes, methods and instance variables.

(X2.2) How and when are entities labeled?

(X2.3) Does the model place restrictions on the labeling of related entities? For example, is a subclass necessarily more sensitive than its superclass?

X2.1 Protectable entities

A model for a secure object-oriented database must specify which entities may be protected. In this section we consider some possibilities and indicate implications of allowing certain entities to be protected. Examples of protectable entities are objects, methods, instance variables, classes, class methods, class variables, etc.

If only objects are allowed to be labeled, the whole object has the same sensitivity; we will refer to such an object as a *single level object*. If portions of an object (ie methods and instance variables) may be labeled individually, it provides finer granularity from a security viewpoint. Because the sensitivity of portions of such an object may be different, we will refer to such an object as a *multilevel object*.

Models that only support single level objects and models that support multilevel objects have been proposed and both types seem practical [11]. Models supporting

only single level objects have the benefit of being simple; also many of the relationship restrictions given later (X2.3) are trivial in such a case. On the other hand, it seems quite natural to label the query and update methods of the same object differently, because there is an inherent difference between the sensitivities of these two methods. Similarly, it seems natural to label the SURNAME and SALARY instance variables of an EMPLOYEE object differently, also supporting the case for multilevel objects.

Multilevel objects present the *multilevel update problem* [24]: suppose that some variables are classified higher (or merely different) than other variables of the same object. The problem to be answered is at what clearance level must a subject be to update the object. If the subject is at the high level, and writes variables with a lower sensitivity, then the subject has 'written down' possibly compromising security. If the subject is at a lower level, it is not authorised to access the more sensitive variables anymore. The only solutions are to either log out and log in for every concerned sensitivity or to polyinstantiate some variables (or the whole object). As Thuraisingham points out, neither of these solutions is desirable. Note that this problem only occurs when instance variables are allowed to have a different sensitivity than the containing object; methods may differ without any adverse effects. Further, see [10] for a discussion of a number of problems associated with multilevel objects.

Classes also have entities that may be protected. Remember that classes are objects themselves; they have methods (known as class methods, such as CREATE), they have variables (known as class variables) and they are instantiations of metaclasses. Therefore any remarks about labeling of objects or their facets also hold for the 'object aspects' of a class. Note that a class variable must be labeled in the class because it is available to all instances of the class (although the sensitivity in a particular instance may be higher than the sensitivity specified in the class—see X2.3). It is also possible to label the 'template aspects' of classes (including methods and variables defined there) with the intention that the given sensitivity label should apply to all instances of the class, rather than to protect the class itself—we address this below (X2.2).

The protectable entities of SODA are instance variables and objects; however SODA allows either the entire object to be labeled, or the individual instance variables, but not both.

In addition to the usual protectable entities SECDB also allows 'layers' of objects to be labeled: The class of an object may have many superclasses. Portions of an object are therefore defined in a number of (super-) classes. These portions form layers—the innermost layer defined in the 'highest' superclass, while the outermost layer is defined in the (immediate) class of the object. Protecting the outermost layer corresponds to protecting the entire object in other models; labeling any other layer protects those portions of the object that were defined in the corresponding superclass and any of its superclasses.

X2.2 Label instantiation

An object-oriented system is a dynamic system: objects are instantiated and destroyed continually. In order not to compromise security, newly created (instantiated)

objects must be protected immediately. The initial sensitivity of an entity reflects the inherent sensitivity of the entity. For example, it can be pre-determined which subjects will be allowed to invoke the INCREASESALARY method of an EMPLOYEE object. Normal database activities will have no influence on the sensitivity of this method. Similarly, the inherent sensitivities of the instance variables of such an object may be pre-determined reflecting the sensitivity of the value of such a variable or the sensitivity of the relationship between the object and the contents of that variable. However, some models allow the sensitivity of such a variable to be increased dynamically if a particularly sensitive value is stored in that variable. Stated differently, the initial sensitivity of a variable reflects the sensitivity of the 'container'; in some cases the 'contents' of the variable will be more sensitive than the 'container' itself, and at that point the sensitivity of the variable may be higher than its initial sensitivity. Here we are only interested in the initial sensitivity; see *dynamic labeling* later for details about re-labeling of entities.

Three primary possibilities exist for determining the initial sensitivity of an object.

1. The class must be labeled and the label(s) specified for the class must apply for all instances of the class;
2. Every object (and possibly its variables and methods) must be explicitly labeled when or after the object is instantiated; [25] proposes that the method that instantiates a new object may specify the sensitivities of the interface variables (parameters) from which the class of the new object may then derive the sensitivities of all instance variables; or
3. Constraints may be specified—ie separate (logic) rules that determine the sensitivity of a newly instantiated object and then ensures that the entity is sensitivity labeled immediately.

Of course, a combination is also possible with default labels derived from the class and individual labels given after instantiation where the default labels do not suffice.

Since it is unreasonable to trust normal methods to sensitivity label a newly instantiated object, and since constraints fall outside the scope of this work, the mechanism that will be used in this paper for labeling of newly instantiated objects is inheritance: mechanism (1) above. SODA and SECDB use this mechanism.

X2.3 Relationship restrictions

The third question—on the labeling restrictions of related entities—leads to some interesting results. The relationships to consider are:

- *Aggregation*: The relationships that exist between an object and its facets (name, instance variables, methods);
- *Instantiation*: The relationships that exist between a class and its instances;
- *Inheritance*: The relationships that exist between a class and its subclasses;
- *Composition*: The relationships between objects that are combined into a larger object;

- *Association*: The relationships for objects that exist in order to associate two or more other objects;
- *Data structure membership*: The relationships between a data structure (such as a list) and a member of the data structure; also the relationships amongst members themselves.

This paper focusses on the first four types of relationships; to address data structures additional axioms will have to be included for any specific data structure.

Relationship restrictions may be divided into *compulsory* and *additional* restrictions. Compulsory relationship restrictions are those restrictions that a model must enforce as a result of design choices made elsewhere or as a result of the inherent object-oriented structure. Additional relationship restrictions are other restrictions a model may prescribe because they simplify the model or have some other benefit. We discuss compulsory relationship restrictions first.

The first relationship restrictions we consider, are imposed by the *aggregation* of an object. An object encapsulates everything inside it. This implies that a facet thus encapsulated cannot be accessed by a subject that is not allowed to access the encapsulating object in the first place. This holds whether the model uses existence protection or access protection.

Lemma 2 (Encapsulation corollary) *The sensitivity of a facet of an object dominates the sensitivity of the object itself, ie $L(\langle x, o\rangle) \geq L(o)$ for every facet x of any object o. Similarly, the sensitivity of a facet of a class dominates the sensitivity of the class itself, ie $L(\langle x, c\rangle) \geq L(c)$ for every facet x of any class c.*

Proof: This lemma follows from the fact that, if a subject is not authorised to access o, that subject is also not authorised to access any facet x of o, that is $s \notin subj(o) \rightarrow s \notin subj(\langle x, o\rangle)$. Therefore, by contrapositive, $s \in subj(\langle x, o\rangle) \rightarrow s \in subj(o)$. Hence, $subj(\langle x, o\rangle) \subseteq subj(o)$. From the definition of L follows that $L(\langle x, o\rangle) \geq L(o)$.

The proof for the second part of the lemma is similar. □

This lemma is similar to the *facet property* (property 3) of Lunt.

Our second group of relationship restrictions are imposed by *instantiation*: the relationships that exist between a class and its instances.

If the model uses existence protection, both inheritance and instantiation restrict labeling: If (part of) a class is existence protected, that (part of the) class does not exist as far as an unauthorised subject is concerned. However, if such an unauthorised subject is authorised to access a subclass or an instance of the protected class, this protected information becomes visible to the subject (even if not accessible by the subject): from the subclass the subject can 'see' the names of methods, instance variables and even the composition of the superclass; similar information can be gathered from an instance about the concerned class. This motivates the next theorems.

Lemma 3 *In an existence protected model, the sensitivity of an instance must dominate the sensitivity of its class, ie*

$$(\forall o \in U)[L(o) \geq L(class(o))]$$

Also, the sensitivity of a facet in an instance must dominate the sensitivity of the facet in the class and also dominate the sensitivity of the instance itself, ie

$$(\forall o \in U)(\forall x \in o)[L(\langle x, o\rangle) \geq \lceil L(\langle x, class(o)\rangle), L(o)\rceil]$$

Proof: From axiom 1 we know that $o \in U \rightarrow class(o) \in C$. Since $U \subseteq E$ and $C \subseteq E$, it follows that, for every $o \in U$,

$$o \in E \rightarrow class(o) \in E$$

Applying lemma 1 proves that $L(o) \geq L(class(o))$.

From axiom 2 $\langle x; o\rangle \in E \rightarrow \langle x, class(o)\rangle \in E$. Applying lemma 1 to this proves $L(\langle x, o\rangle) \geq L(\langle x, class(o)\rangle)$. From lemma 2 follows that $L(\langle x, o\rangle) \geq L(o)$ and therefore

$$(\forall o \in U)(\forall x \in o)[L(\langle x, o\rangle) \geq \lceil L(\langle x, class(o)\rangle), L(o)\rceil]$$

□

From the discussion on label instantiation above (X2.2) we may assume that similar restrictions exist for the access protected model. We state this as an axiom.

Axiom 6 *In an access protected model, if a class (including variables and methods defined in that class) is sensitivity labeled, the intention is that such labels should apply for all instantiations of the class; in other words, the sensitivity of an instance must dominate the sensitivity of its class; and the sensitivity of any facet of an instance should dominate the sensitivity of that facet as defined in the class.*

From lemma 3 and axiom 6 above we have

Theorem 4 (Instantiation restriction) *The sensitivity of an instance must dominate the sensitivity of its class, ie*

$$(\forall o \in U)[L(o) \geq L(class(o))]$$

□

Property 2 of Lunt (when interpreted for classes and instances) and property P7 of SORION are the same as our instantiation restriction (theorem 4).

Theorem 4 constrained the sensitivity of an instance of a class. The following theorems constrain the sensitivity of a facet of such an instance.

Theorem 5 *In an existence protected model, the sensitivity of a facet x of an object o is given by*

$$L(\langle x, o\rangle) = \lceil L(\langle x, class(o)\rangle), L(o)\rceil$$

Proof: Let $c = class(o)$. From lemma 3, $L(\langle x, o\rangle) \geq \lceil L(\langle x, c\rangle), L(o)\rceil$. Thus $subj(\langle x, o\rangle) \subseteq subj(\langle x, c\rangle) \cap subj(o)$. Let $s \in subj(\langle x, c\rangle) \cap subj(o)$. Then $s \in subj(\langle x, c\rangle)$ and $s \in subj(o)$. From axiom 3, since $o \in U$ and $c = class(o)$ and

$x \in c$, the fact that $x \in o$ is implied. According to lemma 1, $s \in subj(\langle x, o\rangle)$. Therefore $subj(o) \cap subj(\langle x, o\rangle) \subseteq subj(\langle x, c\rangle)$. As indicated earlier, $subj(\langle x, o\rangle) \subseteq subj(\langle x, c\rangle) \cap subj(o)$. Therefore $subj(\langle x, o\rangle) = subj(\langle x, c\rangle) \cap subj(o)$, and thus

$$L(\langle x, o\rangle) = \lceil L(\langle x, class(o)\rangle), L(o)\rceil$$

□

The practical implication of this theorem is given in the following corollary.

Corollary 6 *In an existence protected model, the only way to adjust the sensitivity of a facet x of an object o from its inherited sensitivity—$L(\langle x, class(o)\rangle)$—is by increasing the sensitivity of the entire object o.*

□

Theorem 7 *In an access protected model, the sensitivity of a facet x of an object o dominates both the sensitivity of the facet in its class and the sensitivity of the object itself, ie*

$$L(\langle x, o\rangle) \geq \lceil L(\langle x, class(o)\rangle), L(o)\rceil$$

Proof: From lemma 2, we know that $L(\langle x, o\rangle) \geq L(o)$. From axiom 6 follows that $L(\langle x, o\rangle) \geq L(\langle x, class(o)\rangle)$. □

Our third group of relationship restrictions are imposed by *inheritance*: the relationships that exist between superclasses and their subclasses.

Lemma 8 *In an existence protected model, the sensitivity of a subclass must dominate the sensitivity of its superclass(es), ie for any class $c \in C$*

$$(\forall d \in sup(c))[L(c) \geq L(d)]$$

Proof: This proof is similar to the proof of the first part of lemma 3. □

Axiom 7 *In an access protected model, if a class (including variables and methods defined in that class) is sensitivity labeled, the intention is that such labels should be inherited by all its subclasses (unless the variable or method is re-defined, in which case it may be re-labeled); in other words, the sensitivity of a subclass must dominate the sensitivity of its superclass(es); and the sensitivity of any facet* inherited *from a superclass should dominate the sensitivity of that facet in the superclass.*

Note that this axiom does not apply to SECDB: if a class is labeled in SECDB, the intention is that that label should be applied to the corresponding layer of any instance of that class or of any instance of some (eventual) subclass of the labeled class. Results based on this axiom will thus not hold for SECDB.

From lemma 8 and axiom 7 above we have

Theorem 9 (Inheritance restriction) *The sensitivity of a subclass must dominate the sensitivity of its superclass(es), ie for every class $c \in U$*

$$(\forall d \in sup(c))[L(c) \geq L(d)]$$

□

Property 2 of Lunt (when interpreted for superclasses and subclasses) and property P9 of SORION are the same as our inheritance restriction (theorem 9).

Lemma 10 *In an existence protected model, if any class $c \in U$ inherits a facet x from a superclass $d' \in sup(c)$ or re-defines a facet x that occurs in a superclass d, then*

$$L(\langle x, c\rangle) \geq \lceil L(\langle x, d'\rangle), L(c)\rceil$$

Proof: This proof is similar to the proof of the second part of the lemma 3. □

Property 4 of Lunt (when interpreted for superclasses and subclasses) is the same as lemma 10.

Theorem 11 *In an access protected model, if any class $c \in U$* inherits *a facet x from a superclass $d' \in sup(c)$, then*

$$L(\langle x, c\rangle) \geq \lceil L(\langle x, d'\rangle), L(c)\rceil$$

Proof: This proof is similar to the proof of the second part of the lemma 4. □

Multiple inheritance is supported if the data model allows a class to, simultaneously, be a subclass of more than one superclass. In general, if multiple inheritance is supported it is necessary to specify which facet (variable or method) will be inherited if it is defined in more than one superclass. In the case of existence protection problems may occur: Assume that a subject is cleared to access facet F in class C1. Assume further that class C3 is a subclass of both classes C1 and C2. If the subject does not see the facet F in C3 it implies that facet F is inherited from C2. The subject can therefore infer that facet F exists in both C2 and C3, although the subject is not cleared to know about the facet's existence. This is addressed by the following theorem.

Theorem 12 (Facet inheritance restriction) *If, in an existence protected model, x is a facet of a class c and x also appears in (at least) one superclass of c, then the sensitivity of $\langle x, c\rangle$ is bounded as follows, for every superclass $d \in sup(c)$ that has a facet x*

$$L(c) \leq L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$$

Further, if x is inherited from a specific superclass $d' \in sup(c)$, then, for every superclass $d \in sup(c)$ that has a facet x

$$\lceil L(\langle x, d'\rangle), L(c)\rceil \leq L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$$

Proof: The fact that $L(c) \leq L(\langle x, c\rangle)$ was stated in lemma 2. The fact that, if x is inherited from a specific superclass d', then $\lceil L(\langle x, d'\rangle), L(c)\rceil \leq L(\langle x, c\rangle)$ was stated in lemma 10.

To prove the rest of the inequality, ie to prove

$$L(\langle x, c\rangle) \leq \lceil L(\langle x, d\rangle), L(c)\rceil$$

select any $d \in sup(c)$ such that $x \in d$. According to axiom 5 this implies that $x \in c$. According to lemma 1 for any $s \in S$ for which both $s \in subj(\langle x, d\rangle)$ and $s \in subj(c)$ it must also hold that $s \in subj(\langle x, c\rangle)$. Thus $s \in subj(\langle x, d\rangle) \wedge s \in subj(c) \rightarrow s \in subj(\langle x, c\rangle)$ and therefore

$$subj(\langle x, d\rangle) \cap subj(c) \subseteq subj(\langle x, c\rangle) \tag{1}$$

Now assume, contrary to what is being proven, that $L(\langle x, c\rangle) > \lceil L(\langle x, d\rangle), L(c)\rceil$. Then $L(\langle x, c\rangle) > L(\langle x, d\rangle)$ and $L(\langle x, c\rangle) > L(c)$. From the definition of L it follows that $subj(\langle x, c\rangle) \subset subj(\langle x, d\rangle)$ and $subj(\langle x, c\rangle) \subset subj(c)$. Thus we have

$$subj(\langle x, c\rangle) \subset subj(\langle x, d\rangle) \cap subj(c)$$

which contradicts equation 1 and proves the theorem. □

This theorem makes a number of statements about inherited (and re-defined) facets in an existence protected model. Some of these statements are given in the next corollary.

Corollary 13 *In an existence protected model*

1. *If a class c* inherits *a facet $\langle x, c\rangle$, the sensitivity of $\langle x, c\rangle$ may only be different from its sensitivity in the superclass when $L(\langle x, c\rangle) = L(c)$; in other words, the only way to increase the sensitivity of an inherited facet $\langle x, c\rangle$, is by increasing the sensitivity of the entire class c. (Corollary 6 made a similar remark about instances.)*
2. *The sensitivity of a* re-defined *facet $\langle x, c\rangle$ must be dominated by that of every like-named facet in any superclass of c, whenever the sensitivity of the like-named facet dominates the sensitivity of c.*
3. *If a facet is defined in only one superclass of a given class c (or the class c has only one superclass, or the object-oriented model only allows single inheritance), that facet may be inherited without any problems; the sensitivity of the inherited facet will be the least upper bound of its sensitivity in the superclass and the sensitivity of the class c.*
4. *The sensitivity of an inherited facet $\langle x, c\rangle$ must be dominated by the sensitivity of all like-named facets in superclasses of c, whenever the sensitivity of the like-named facet dominates the sensitivity of c.*

□

The last point of the corollary above indicates two strategies an existence protected model may follow to prevent the problems presented by multiple inheritance:

1. Ensure that the like-named facet with the lowest sensitivity is always inherited; or
2. Ensure that the sensitivity of a subclass is an upper bound for the sensitivities of all concerned facets in superclasses.

Strategy 1 is only feasable if a facet with a lowest sensitivity does indeed exist; if the sensitivities are partially ordered the existence of such a facet is not guaranteed. Of course, a model may also solve these problems by disallowing multiple inheritance.

Property 5 of Lunt requires that the facet with the lowest sensitivity must be inherited—ie strategy 1 above. Properties P15 and P16 of SORION require that the facet with the highest sensitivity must be inherited—contradictory with this theorem.

Next, we consider the implications of two objects that are related because one is part of the other object, or because the two objects are associated in some other way. To simplify matters, we will assume that one object contains (in one of its instance variables) the object identifier of the other object; the object identifier is a unique value associated with every object. All the properties previously derived for facets of an object also hold for the object identifier.

Note that the sensitivity of an object is not important if that entire object is retrieved from a variable, passed as a parameter to another object, etc. Since the object is an encapsulated unit it still has its original protection wherever it is moved, copied or stored; access to the *contents* of the object (such as its variables and methods) is what is protected: a retrieved object cannot even be compared to another object without using one of the methods of the concerned object(s). This holds for both existence and access protection. In the case of existence protection the ability of a subject to retrieve an object without permission to access the object will not compromise security: without accessing the object the subject will not be able to determine whether the retrieved object is the *nil* object (indicating that no actual value was contained in the variable) or some actual object. The subject is therefore not able to use this ability to make any inferences about the existence of objects.

However, suppose that a subject can retrieve an object identifier (and not the object as a whole). Further suppose that the subject can 'inspect' the identifier before attempting to access the corresponding object. Then the subject may infer that an object exists (a problem in the case of existence protection). Further, by comparing this identifier with other identifiers obtained from other variables it becomes possible to make complex inferences (see section 6.2). An access protected model that allows subjects access to object identifiers can simply inform the user that access is not permitted if the subject is not properly authorised to access the identifier (but, possibly, the variable itself). In the case of existence protection most subjects will have access to the *nil* object—therefore if access to an object identifier is denied the subject will be able to infer that the accessed object is not *nil* and make the inference that some other object exists—contrary to the definition of existence protection. This is covered by the next axiom.

Axiom 8 *In an existence protected model that allows direct manipulation of object identifiers, any subject authorised to access a variable, must also be able to access the object identifier of the object contained in the variable. The sensivity of any such variable is the least upper bound of that of the variable itself and the sensitivity of the contained object identifier.*

However, since the sensivity of a variable in an existence protected model is uniquely determined by theorem 5, this means that the sensitivity of the contained

object cannot exceed that of the containing variable:

Theorem 14 *In an existence protected model that allows direct manipulation of object identifiers, the sensitivity of a variable dominates the sensitivity of the object it contains, ie if variable v of object o_v contains object o_i then*

$$L(o_i) \leq L(\langle v, o_v \rangle)$$

Proof: If i is the object identifier of object o_i then it follows from lemma 2 that

$$L(o_i) \leq L(\langle i, o_i \rangle)$$

From axiom 8 we know that the sensitivity of variable $\langle v, o_v \rangle$, while it contains object o_i, is the least upper bound of the sensitivity of variable $\langle v, o_v \rangle$ itself and the sensitivity of the contained object identifier. However, according to theorem 5 the sensitvity of variable $\langle v, o_v \rangle$ (irrespective of its value) is precisely determined by the sensitivity of that variable in the class ($L(\langle v, class(o_v) \rangle)$) and the sensitivity of the variable's object ($L(o_v)$). Therefore the the sensitivity of a variable cannot be more than its labeled sensitivity, which means we must have

$$L(\langle i, o_i \rangle) \leq L(\langle v, o_v \rangle)$$

Therefore

$$L(o_i) \leq L(\langle i, o_i \rangle) \leq L(\langle v, o_v \rangle)$$

which proves the theorem. □

Note that this theorem deals with a static situation: viewed at any specific instant the contents of a variable cannot be more sensitive than the variable; however, it may be possible to *increase* the sensitivity of a variable when more sensitivity is required (if the circumstances of theorem 5 apply, the sensitivity of the variable is increased by increasing the sensitivity of the entire object). This is addressed later when dynamic labeling (X3) is discussed.

Properties P18 and P19 of SORION state that the sensitivity of an aggregate object dominates the least upper bound of all its components. Since SORION uses single level objects, this property corresponds to our theorem 14. One may conclude from property P3 of SORION that SORION does use existence protection and that it allows direct access to the object identifier (or "object name" as referred to in property P3). (Note that property P20 of SORION makes a similar remark about the sensitivity of relationship objects—objects that solely exist to relate other objects; our axioms about object-orientation do not make separate provision for such objects; however if relationship objects are seen as special cases of objects theorem 14 does make a statement about such objects.)

We conclude our discussion of compulsory restrictions with a few remarks about restrictions imposed by data structure membership. In an array-like structure it is usually possible to infer the structure of one element from that of another. This indicates that all members of the array must have the same sensitivity in an existence

protected model. Of course, if the elements of such a data structure are not necessarily homogeneous, this requirement may be dropped. We do not address the sensitivity of elements of data structures in detail in this paper, but give one last example: property P4 of SORION specifies that the sensitivity of a 'set object' is the least upper bound of the sensitivities of the element objects.

The labeling restrictions above raises two questions:

1. The discussion above deals with the sensitivities of classes, objects, variables and methods. What if the model support fewer or additional protectable entities (parameter X2.1); and
2. Is the list of restrictions given complete, or are some restrictions overlooked?

The first question is relatively simple: If some of the entities that we used are not labeled explicitly by a model, they must be labeled implicitly and the consequences of that labeling taken into account. For instance, if single level objects are used then, although the facets of objects are not explicitly labeled, the intention is that all facets have the same label as the containing object and facets must conform to the restrictions given. As a slightly more complex example, assume that classes are not labeled for some reason (say, because, they are protected by discretionary security measures and therefore cannot be accessed by subjects not highly trusted). In such a case implicit labels have to be derived for classes from the restrictions given; these (implicit) labels will then affect the labels of other entities that are (explicitly) labeled. For example, even if classes and facets of classes are not labeled, security may be compromised in the following existence protected database: both JOHN and JAMES are instances of the same class; both objects have a CONFIDENTIAL sensitivity; JOHN has a TOP SECRET method GETSALARY and a SECRET method GETJOBTITLE, while JAMES has a SECRET method GETSALARY and a TOP SECRET method GETJOBTITLE. This is solved by implicitly labeling GETSALARY and GETJOBTITLE in the class (from the explicit labels in the instances) and ensuring that the labels of the methods in the instances conform to the given restrictions.

If a model has more protectable entities than the ones we described, they may either be special cases of entities that we did consider (such as the *relationship objects* mentioned earlier), or axioms will have to be added to describe their relationships with other entities and new restrictions derived for them.

The second question is more difficult to answer (and also more critical if we want to trust the any model we design). Given the relative small set of axioms and the fact that we are ony interested in theorems about sensivity levels of related entities, it may be possible to check that the given restrictions are exhaustive. It may even be possible to automate the generation of restrictions (or the checking of a given set). However, this does not solve the problem—how do we know that the axioms cover all relationships? For example, without axiom 8 it is not possible to prove theorem 14, and it is not obvious from the problem that an axiom such as axiom 8 should be included. The only solution seems to be to have as many people as possible proposing axioms and then selecting those applicable to one's model. Such a brute force method is not unheard of in the security community—this method has long been used in cryptology.

Theorem 4 :	$L(o) \geq L(class(o))$ for every object $o \in U$
Theorem 5 :	$L(\langle x, o \rangle) = \lceil L(\langle x, class(o) \rangle), L(o) \rceil$ for every object $o \in U$ and every facet x of o
Theorem 9 :	$L(c) \geq L(d)$ for every class $c \in C$ and every superclass $d \in sup(c)$
Theorem 12 :	$L(c) \leq L(\langle x, c \rangle) \leq \lceil L(\langle x, d \rangle), L(c) \rceil$ for every class $c \in C$ and every superclass $d \in sup(c)$ of c that has a facet x; $\lceil L(\langle x, d' \rangle), L(c) \rceil \leq L(\langle x, c \rangle) \leq \lceil L(\langle x, d \rangle), L(c) \rceil$ if c inherited the facet x from $d' \in sup(c)$
Theorem 14 :	$L(o_i) \leq L(\langle v, o_v \rangle)$ if variable $\langle v, o_v \rangle$ contains object o_i and the system allows direct manipulation (comparison) of object identifiers

Table 1: Relationship restrictions for an existence protected model.

Theorem 4	$L(o) \geq L(class(o))$ for every object $o \in U$
Theorem 7	$L(\langle x, o \rangle) \geq \lceil L(\langle x, class(o) \rangle), L(o) \rceil$ for every object $o \in U$ and every facet x of o
Theorem 9	$L(c) \geq L(d)$ for every class $c \in C$ and every superclass $d \in sup(c)$
Theorem 11	$L(\langle x, c \rangle) \geq \lceil L(\langle x, d' \rangle), L(c) \rceil$ where $c \in C$ is any class and x a facet that c inherited from a superclass $d' \in sup(c)$

Table 2: Relationship restrictions for an access protected model.

The restrictions on labeling of related entities described above are necessary because security will be compromised if the restrictions are not enforced; some models have additional restrictions because they simplify the model or enhance security in some other way. We briefly look at some of these.

Some models (eg Lunt property 1 and SORION property P2) specify that basic objects (or system objects) must have a system-low sensitivity (L_{low}). If a model includes such a specification, the model has to specify exactly what is meant by basic objects or system objects. If such a requirement is not included, the system security officer and/or the database system have to define the sensitivity of all objects supplied with the database.

The relationship restrictions imposed on existence protected models are summarised in table 1. The restrictions imposed on access protected models are summarised in table 2.

The relevant restrictions from the Lunt model (in our notation) are given in table 3. Property 6 does not appear in that table because it deals with run-time access restrictions, which we discuss under the heading *dynamic labeling* later—see equations 2 and 3. Also note that, since Lunt does not distinguish between a subclass and

Property 1	$L(c) = L_{low}$ for every system-defined class c $L(o) = L_{low}$ for every system-defined object o
Property 2	$L(c) \geq L(d)$ for every class c and every superclass $d \in sup(c)$ $L(o) \geq L(class(o))$ for every object o
Property 3	$L(\langle x, c\rangle) \geq L(c)$ for every facet x of any class c $L(\langle x, o\rangle) \geq L(o)$ for every facet x of any object o
Property 4	$L(\langle x, c\rangle) \geq L(\langle x, d'\rangle)$ for every facet x of any class c where $\langle x, c\rangle$ is inherited from a superclass $d' \in sup(c)$ $L(\langle x, o\rangle) \geq L(\langle x, class(o)\rangle)$ for every facet x of any object o
Property 5	$L(\langle x, d'\rangle) \leq L(\langle x, d\rangle)$ for every $d \in sup(c)$ whenever any class c inherits a facet x from a superclass $d' \in sup(c)$
Property 7	$L(c) \geq L(\langle x, d\rangle) \rightarrow L(\langle x, c\rangle) \leq L(c)$ for every superclass $d \in sup(c)$ of any class c that has a facet x $L(o) \geq L(\langle x, class(o)\rangle) \rightarrow L(\langle x, o\rangle) \leq L(o)$ for every facet x of any object o

Table 3: Relationship restrictions imposed by the Lunt model.

an instance of an object, we have interpreted most of the Lunt properties for subclasses and for instances. Remember that the Lunt model uses existence protection and, therefore, table 3 has to be compared to table 1. Lunt's property 2 is identical to our theorems 4 and 9. Nothing in [11] necessitates Lunt's property 1, and therefore it falls in the *additional relationship restriction* category. Lunt's properties 3, 4 and 7 are easily derived from our theorems 5 and 12 , but the converse is not true. Lunt's property 5 is stricter than necessary; for those situations where it is required, it can be derived from our theorem 12.

4.3 Dynamic labeling

This section concerns itself with the flow of authorisation (X3.1), the flow of sensitive data (X3.2) and the restrictions on such data flows to ensure secrecy despite such flows (X3.3). Restrictions based on flow of authorisation and flow of information that a model introduces to ensure that security will not be compromised, represent a generalisation of the *-property of the Bell-LaPadula model [1].

The dynamic activities in a system may be modeled by the following simple grammar:

$$\begin{aligned} \Sigma &\rightarrow M \\ M &\rightarrow a_i T \\ T &\rightarrow MT \mid r \mid \epsilon \end{aligned}$$

Here Σ represents the *primary accessor*, in other words the (probably human) object that sends the original message to the database. The non-terminal M represents a message; the production $\Sigma \rightarrow M$ models the message sent by the primary

accessor. A message causes a method (of a specific object) to be activated; in the grammar above a_i represents such an active object (or, more precisely method-object pair $\langle m, o \rangle$). The list of activities such an active method performs are represented by T. The production $M \rightarrow a_i T$ indicates that a specific method (a_i) is activated on receipt of a message, after which that method 'executes' a list of activities T. The latter production represents a set of productions, each with a different a_i for every method that may be activated. The production $T \rightarrow MT$ represents the case where a list of activities T consists of sending a message, before executing some more activities; the production $T \rightarrow r$ represents the activity that terminates execution of the active method and sends a reply to the calling method, while the production $T \rightarrow \epsilon$ represents the activity that terminates execution of the active method and returns control to its calling method without sending a reply to the calling method. Note that the intention of these productions is not to generate strings, but to reflect events in the system.

Authorisation flow deals with the question whether and how the clearance of a subject is influenced by the method activations acting on its behalf—see X3.1.

Information flow deals with the flow of sensitive information through the system (as messages) and, more particularly, the restrictions that a model may enforce to ensure that such information does not flow to somewhere where it is less protected.

As long as an object moves as a unit from one location to another location in the system (probably as a parameter of a message) information will not be compromised: The object will still have its original sensitivity label; similarly, the methods and variables of the object will still have their original sensitivity labels. As an example, suppose that a SALARY object has a PRINTSALARY method. A subject that obtains SALARY from the EMPLOYEE object will still not be able to invoke the PRINTSALARY method if it is not authorised to—even though it obtained the SALARY it cannot do anything with it. The normal encapsulation feature of object-orientation, combined with sensitivity labels, provides very natural protection.

However, if, in the example above, the salary was not encapsulated in a SALARY object, but rather stored as a normal real number, the value would have no natural protection once it leaves the EMPLOYEE object. A similar problem occurs anywhere where an object provides methods that return values of instance variables—when the value leaves the protection of the encapsulated object, security may be compromised.

In the final part of this section we consider restrictions that may be dynamically applied to *information flow* through the system to ensure that, even if such information is removed from the encapsulated object, it will still not be exposed to unauthorised access. We assume that information retrieved from a variable or object is as sensitive as the label for that variable or object indicates and that such information must stay at least as sensitive wherever it might flow in the system. One cannot reasonably make this assumption about information obtained via a sensitive method; another (much less sensitive) method may return the same information. To avoid such inconsistencies, we only consider restrictions for information flowing from a sensitive variable or sensitive object. One possibility to include the sensitivity of methods when considering information flow, is to ensure that any variable is at least as sensitive as the greatest lower bound of any method that has access to the variable. We do not investigate this option in the current paper.

Under dynamic labeling we consider three aspects:

(X3.1) Messages act on behalf of a subject and therefore the clearance of the message depend on that of the subject. A model has to specify how the clearance (or authorisation) of a message is determined.

(X3.2) Messages also carry information—this information may be sensitive, requiring labels. A model has to specify how the sensitivity of a message is determined.

(X3.3) If some of the sensitive information contained in a message is stored in variables of the receiving object, it must be ensured that an unauthorised subject cannot now access the information in this object. This can be ensured by either re-labeling the object or variable with a suitable label (if the existing labels are not suitable) or by disallowing information to be saved if the existing labels are not suitable. The model should indicate any flow restrictions and any conventions for re-labeling.

X3.1 Authorisation flow

Consider the grammar above. Which objects should be taken into account when the clearance of a message is determined? The following possibilities exist:

1. Consider only the *primary accessor* (represented by Σ);
2. Consider all method activations that have been activated since the original message was sent, but ignore those that have already returned control to their calling methods (ie consider all 'active objects'), including the primary accessor; or
3. Consider all objects on the access path of the request, ie every method activation a_i that has been activated since the original message was sent, whether that method has terminated execution or not, including the primary accessor.

In SODA the primary accessor determines the clearance of a message (possibility 1 above).

Lunt's property 6 states (in our notation)

$$L(s) \geq L(\langle m, o \rangle) \geq L(o) \tag{2}$$

$$L(M) = L(s) \tag{3}$$

where $s \in S$ is any subject, $\langle m, o \rangle$ is a method of object o that s wants to execute and M is a message sent by s to this effect. It seems from this property that Lunt denotes the *clearance* of a subject s by $L(s)$; if the clearance of the subject dominates the sensitivity of the method to be activated, the subject is allowed to activate it (2). Further, the message M has exactly the same *clearance* that its sending subject had (3). It therefore seems that this property indicates that authorisation in the Lunt model depends only on the primary accessor (possibility 1 above).

SECDB uses the complete access path to determine authorisation (possibility 3 above); however, an individual profile is free to ignore any part of the access path

when it determines whether access should be granted. As an example why consideration of the entire access path might be useful, consider a stock control system, where the INVENTORY object may only be asked to issue stock (decrease its current stock level) if a message have previously been sent to the CHECKCREDIT method of the requesting object.

In practice, clearances are assigned as follows for the various possibilities:

Primary accessor: All messages sent (by any method of any object) on behalf of the primary accessor have the same clearance—that of the primary accessor;

Active objects: All messages sent by a method of an object will have the same clearance for a given activation of the method; that clearance depends on the clearance of the object-method pair and on the clearance of the message that activated the method; and

Access path: The clearance of any message sent depends on all objects and methods involved anywhere previously in the request.

The clearance of messages are influenced by other objects involved in the request if either possibility 2 or 3 above is used. As an example, a model may require that the clearance of a message is not higher than the clearance of any object involved in the request thus far. To generalise this concept, we introduce semantic actions to associate a clearance attribute $M.c$ with every message sent. These rules are given in tables 4, 5 and 6. We also associate a sensitivity attribute $e.l$ with every protected entity e. The possible values of these attributes depend on the underlying model (X1.1); for the time being we assume that it is possible to determine whether message M is permitted to access entity e given the attribites $M.c$ and $e.l$. $\Sigma.c$ denotes the clearance of the primary accessor, $a_i.c$ denotes the clearance of an individual method activation and the function *clear* maps the clearance of a message and a method activation to their combined clearance.

Event	**Semantic rule**
$\Sigma \rightarrow M$	$M.c := \Sigma.c$
$M \rightarrow a_i T$	$T.c := M.c$
$T \rightarrow M T_1$	$M.c := T.c$
	$T_1.c := T.c$
$T \rightarrow r$	
$T \rightarrow \epsilon$	

Table 4: Semantic rules for message clearance based on the primary accessor

In addition to the method a secure database model uses, it thus have to specify up to four additional pieces of information:

- The format of the clearance attributes ($\Sigma.c$, $a_i.c$ and $M.c$);
- The format of the sensitivity attribute ($e.l$);

Event			Semantic rule		
Σ	$\rightarrow$	M	$M.c$	$:=$	$\Sigma.c$
M	$\rightarrow$	$a_i T$	$T.c$	$:=$	$clear(M.c, a_i.c)$
T	$\rightarrow$	MT_1	$M.c$	$:=$	$T.c$
			$T_1.c$	$:=$	$T.c$
T	$\rightarrow$	r			
T	$\rightarrow$	ϵ			

Table 5: Semantic rules for message clearance based on active objects

Event			Semantic rule		
Σ	$\rightarrow$	M	$M.c$	$:=$	$\Sigma.c$
M	$\rightarrow$	$a_i T$	$T.c$	$:=$	$clear(M.c, a_i.c)$
			$M.r$	$:=$	$T.r$
T	$\rightarrow$	MT_1	$M.c$	$:=$	$T.c$
			$T_1.c$	$:=$	$M.r$
			$T.r$	$:=$	$T_1.r$
T	$\rightarrow$	r	$T.r$	$:=$	$T.c$
T	$\rightarrow$	ϵ	$T.r$	$:=$	$T.c$

Table 6: Semantic rules for message clearance based on the access path

- For which values of $M.c$ and $e.l$ message M may access e; and
- How clearances are combined, ie define *clear*.

Having given rules to determine the clearance of a message we will now consider how the clearance attribute may be used to specify when a given message is authorised to access an entity. As stated earlier, this depends not only on the clearance attribute of the message, but also on the sensitivity attribute of the entity.

The entities that need protection are methods and instance variables. (The object-oriented model does not allow direct access to other entities at all.)

In order to model access to instance variables, the following two productions have to be added to tables 4, 5 and 6:

$$\begin{aligned} M &\rightarrow px_i \\ M &\rightarrow gx_i \end{aligned}$$

These productions model accessing variables by sending messages to them: $M \rightarrow px_i$ models writing ('putting') a variable x_i (where $x = \langle v, o \rangle$ for some object o and some variable $v \in o$). Similarly $M \rightarrow gx_i$ models reading ('getting') such a variable x_i. We assume that accessing a variable does not affect the clearance of a message; these productions therefore have no semantic actions to modify the clearance attribute; in

table 6 it is necessary to add the action $M.r := M.c$ to both productions to indicate that the clearance does not change. It is now easy to attach semantic actions to the productions $M \rightarrow a_i T$, $M \rightarrow px_i$ and $M \rightarrow gx_i$ to let the message fail when access rights are insufficient; table 7 contains an example of a complete specification of authorisation flow for a model based on explicit levels and using the complete access path to determine the clearance of a message.

Event	**Semantic rule**
$\Sigma \rightarrow M$	$M.c := \Sigma.c$
$M \rightarrow a_i T$	$T.c := \lfloor a_i.c, M.c \rfloor$ $M.r := T.r$ if $M.c < a_i.l$ then fail
$M \rightarrow px_i$	$M.r := M.l$ if $M.c < x_i.l$ then fail
$M \rightarrow gx_i$	$M.r := M.l$ if $M.c < x_i.l$ then fail
$T \rightarrow MT_1$	$M.c := T.c$ $T_1.c := M.r$ $T.r := T_1.r$
$T \rightarrow r$	$T.r := T.c$
$T \rightarrow \epsilon$	$T.r := T.c$

Table 7: Example of access checking rules for a specific model

Some concrete examples of the security attributes denoting the clearance of a subject and the sensitivity of an entity are now given. Note that specific models may differ from these examples.

Explicit levels: The clearance attribute will typically be an integer indicating the clearance level of the primary accessor ($\Sigma.c$), individual method ($a_i.c$) or a message ($M.c$). The sensitivity attribute $e.l$ for an entity e will also be an integer. A message M will be allowed to access an entity e if $M.c \geq e.l$. The combination of two clearance levels may be the greatest lower bound of the two, ie $clear(M.c, a_i.c) = \lfloor M.c, a_i.c \rfloor$.

Access control lists: The clearance attribute will typically be the identity of the primary accessor ($\Sigma.c$), the identity of the individual method ($a_i.c$) or the collected identities associated with the message ($M.c$). The sensitivity attribute $e.l$ will be an access control list, giving the identities of subjects permitted to access the entity. (The access control list may be given explicitly as a list; SECDB, for example, gives it as a formal grammar.) A message M will be allowed to access an entity e if the identities collected in $M.c$ appear in the access control list $e.l$. The combination of two clearance attributes may be the concatenation of the two, ie $clear(M.c, a_i.c) = M.c||a_i.c$ where $||$ denotes concatenation.

Capabilities: The clearance attribute may contain the capabilities presented by the primary accessor ($\Sigma.c$), the capabilities presented by the individual method ($a_i.c$) or the capabilities collected by the message ($M.c$). The sensitivity attribute $e.l$ may contain the capability necessary to access entity e. A message M will be allowed to access an entity e if the capability $e.l$ appears in the list of capabilities $M.c$ presented by M. The combination of two clearance attributes may be the union of the two, ie $clear(M.c, a_i.c) = M.c \cup a_i.c$.

As an example SECDB builds a string representing the access path which is subsequently used to verify access restrictions. Therefore the 'clearance' of the primary accessor ($\Sigma.c$) and the methods ($a_i.c$) are *baggage vectors*, representing the identity of the object, the method, the domain (processor) on which the object resides and the integrity state (or mode) the domain uses while executing the method. These vectors are concatenated to form strings; the clearance combination function *clear* concatenates its arguments, ie $clear(M.c, a_i.c) = M.c||a_i.c$. The sensitivity of any entity $e \in E$, represented by $e.l$, is a *profile*. The profile is given as a formal grammar; M may access e only if $M.c$ is derivable in the grammar associated with $e.l$.

Although the informal notion of the set of subjects S used thus far is sufficient for our purposes, we are now in a position to formalise it: a subject is any sentential form ending with an M obtainable by a leftmost derivation from Σ using the productions from the grammar above:

$$S = \{\omega M \in V^* | \Sigma \stackrel{*}{\underset{lm}{\Longrightarrow}} \omega M\}$$

where V represents the alphabet of the grammar. We do not use this definition again in this paper.

X3.2 Sensitivity flow

A message carries information. Some of that information may be explicit values in the form of parameters. The mere fact that a message has been sent, is implicit information about the contents of the database. This information—both explicit and implicit—might be sensitive. Nobody can access this information while it is carried by the message; however, the receiver of the message may assign the information to an instance variable where it is possible to access it. It is therefore necessary to keep track of the sensitivity of information contained by a message. We will use the notation $L(M)$ to indicate the sensitivity of any message M with the same meaning that $L(e)$ has for any entity e. Later (see X3.3) we will discuss how the current sensitivity of a message will be used to prevent information from being assigned to a variable that is not as least as protected as the information contained by the message.

The following rules may be used to keep track of a the sensitivity of a message:

1. The initial message from the primary accessor has low sensitivity (L_{low}) because it does not contain any information;
2. A method starts execution with the sensitivity of the message that activated it;
3. The sensitivity of the method activation is adjusted whenever it receives a reply after sending a message—the sensitivity is adjusted to the least upper bound of its current sensitivity and the sensitivity of the reply;

4. All messages sent by a method activation has the sensitivity currently associated with the activation; whenever the sensitivity of the method activation is adjusted, all subsequent messages will be sent with the new sensitivity level; and
5. If the method sends a reply to its calling method, the reply has the last 'current' sensitivity of the method activation.

For the purposes of sensitivity flow, access to variables is modeled in terms of messages: writing to a variable is done by sending a message to the variable and reading by sending a message and receiving a response. The sensitivity of the response is normally the sensitivity of the concerned variable. However, if the object identifier (and not the object as a whole) is retrieved in an existence protected model, the sensitivity of the reponse is the least upper bound of the sensitivity of the variable and the sensitivity of the object identifier (see theorem 14).

The rules are formalised in table 8. The suffix *.l* again represents the sensitivity of an entity; for example $M.l$ is an attribute representing the sensitivity of message M. The suffix *.u* is used to keep track of the sensitivity of information received by a method that was returned as the result of a message. The value l_{low} is the attribute value that corresponds to the lowest sensitivity L_{low}. Note that the productions $M \rightarrow gx_i$ ('get variable x_i') and $M \rightarrow px_i$ ('put variable x_i') have been included again to model reading and writing of variables.

Event			**Semantic rule**		
Σ	$\rightarrow$	M	$M.l$	$:=$	l_{low}
M	$\rightarrow$	a_iT	$T.l$	$:=$	$\lceil a_i.l, M.l \rceil$
			$M.u$	$:=$	$\lceil M.l, T.u \rceil$
M	$\rightarrow$	px_i	$M.u$	$:=$	l_{low}
M	$\rightarrow$	gx_i	$M.u$	$:=$	$x_i.l$
T	$\rightarrow$	MT_1	$M.l$	$:=$	$T.l$
			$T_1.l$	$:=$	$M.u$
			$T.u$	$:=$	$T_1.u$
T	$\rightarrow$	r	$T.u$	$:=$	$T.l$
T	$\rightarrow$	ϵ	$T.u$	$:=$	l_{low}

Table 8: Semantic rules for message sensitivity

These rules correspond largely to rules 2.1 to 2.4 of SODA. SODA further requires (in its rule 2.3) that the current sensitivity of a method activation must be increased when a value is written (or "added to a polyinstantiated set") at a higher sensitivity level. This restriction is not necessary: any messages that will be rejected because of this restriction subsequent to such a write may be avoided by reordering the statements (messages) in the concerned methods. Of course, nothing prevents a model from including such a specification; it may be included by changing the se-

mantic rule for $M \rightarrow px_i$ to $M.u := x_i.l'$ where $x_i.l'$ denotes the initial sensitivity (see X2.2) of variable x_i.

Since Lunt does not distinguish between sensitivity levels and and clearance levels, Lunt's property 6 (see equation 3 above) probably also indicates that the sensitivity of a message is the same as the 'classification level' (clearance) of the subject that sent it. This 'classification level' of a message is only defined in that model and then never referred to again.

An alternative to the rules given above come from [25]: They suggest that it is possible for a method to start execution at the system-low sensitivity (see our rule 2 above). This may compromise security: method m_1 may access a sensitive variable and based on its value decide to send a message to method m_2; if m_2 does not access any variable and send a (non-sensitive) message to m_3 then m_3 may assign a value an unclassified variable. If it is known that this sequence of events will only occur if the sensitive value accessed by m_1 has a specific (range of) values, it is possible to infer whether that variable does indeed have such a value by examining the unclassified variable written by m_3, contrary to the information flow restrictions discussed later (X3.3). Of course, if the composition of methods is not public knowledge (and cannot be inferred) this strategy is viable.

Note that some secure database models partition the set of entities E such that any subject $s \in S$ is allowed to access all the entities in at most one partition E_i of E, that is if $s_1 \in S$ and $s_2 \in S$ then either $ent(s_1) = ent(s_2)$ or $ent(s_1) \cap ent(s_2) = \emptyset$. Further, these models do not transform the subject because of authorisation flow (X3.1) or, if it does transform a subject s_i into a subject s_{i+1} when a new method joins the composite subject, it ensures that $ent(s_i) = ent(s_{i+1})$. In such a model a subject cannot cause information to flow to a less protected location and it is unnecessary to keep track of the sensitivities of messages or to restrict information flow based on the sensitivity of the information. A model with this property is said to support *single level subjects*.

X3.3 Information flow restrictions

Information contained by a message cannot be accessed directly. However, an object that receives a message, may store the received data in instance variables. It is then possible that other objects may obtain the information from this object (by sending messages to this object). Therefore it must be ensured that sensitive information is not stored in variables that may be accessed by subjects not authorised to access the sensitive information. This can be done by either ensuring that the variable(s) storing the sensitive information may only be accessed by properly cleared subjects, or by ensuring that the entire object may only be accessed by properly cleared subjects.

Suppose that a message containing sensitive information arrives at an object and that the activated method attempts to write information to a variable. If the sensitivity of either the concerned object or concerned variable dominates the sensitivity of the received message no problem occurs: even if sensitive information is stored in the receiving object will it not be compromised because it is at least as protected as in its original location. In other words, any message M, accessing any entity e, must be allowed to proceed if $L(M) \leq L(e)$—and, of course, if the subject represented by

M is in *subj*(e). Note that, in the case of a system where the conditions of theorem 14 apply (*ie* existence protection and object identiefiers directly manipulatable), a value cannot be written to a variable if it will write an invalid value to the variable (according to theorem 14)—steps similar to those when the sensitivity of the message is too high will have to be taken.

Apart from allowing the message to access the protected entity when the sensitivity of the entity dominates the sensitivity of the message, the model may also decrease the sensitivity of the variable (or object) if its sensitivity strictly dominates that of the message. If the sensitivity is decreased, it must still dominate that of the message. Further, the sensitivity of the variable (or object) must never be decreased below its initial (inherent) sensitivity (see X2.2).

If the sensitivity of the object or variable to be accessed does not dominate the sensitivity of the received message, security may be compromised. So, if the message attempts to modify the state of the object, protective steps must be taken. The following possible strategies exist:

1. Reject the message;
2. Increase the sensitivity of the receiving variable or object; or
3. Polyinstantiate the receiving variable or object.

To formalise the previous remarks we will associate semantic actions with the production

$$M \rightarrow px_i$$

because information will only be compromised if information is actually written to a variable; information temporarily 'inside' an object if a message is just 'passing through' without modifying the state of the object will not compromise information. We identify four possible behaviours when a message attempts writing to a variable:

Proceed/Reject: If the sensitivity of the message strictly dominates that of the variable, reject the write; otherwise the protection of the variable is adequate and the write is allowed to proceed. This is accomplished by associating the following semantic action with the production $M \rightarrow px_i$:

$$\text{if } L(M) > L(x_i) \text{ then fail}$$

Increase sensitivity: Here the sensitivity of a variable (or its containing object) is increased if the information to be written is too sensitive for the current sensitivity of the variable. If the sensitivity is increased, it is increased to that of the message; this may be done by either setting $x_i.l$ or $o.l$ (where $x_i = \langle v, o \rangle$) to $M.l$. Formally the production $M \rightarrow px_i$ has the semantic action

$$\text{if } L(M) > L(x_i) \text{ then } x_i.l := M.l$$

In an existence protected model it is not possible to change the sensitivity of a variable at will—see theorem 5; it is necessary to change the sensitivity of the entire object. A similar situation exists if single level objects are used. If the

level of the entire object is to be adjusted, the *then* part of the semantic action has to be changed:

$$\text{if } L(M) > L(x_i) \text{ then } o.l := M.l, \text{ where } x_i = \langle v, o \rangle$$

Note that we are changing the level of a 'terminal symbol'—on a real system it means that the actual security attributes of the related entity must be updated.

Modify sensitivity: Here the sensitivity of the variable is adjusted to reflect the sensitivity of the new value stored in it. This means that the sensitivity of the variable may be increased or decreased. The only provisos are that the sensitivity of the variable must never be decreased below its inherent (initial—see X2.2) sensitivity, or below the sensitivity of its containing class (see lemma 2). Put differently, the sensitivity of a variable $\langle v, o \rangle$ being written to by a message M, is changed to $\lceil L(M), L(o), L'(\langle v, o \rangle) \rceil$ where $L'(\langle v, o \rangle)$ is the initial sensitivity of $\langle v, o \rangle$. Formally, $M \rightarrow px_i$ takes the semantic action

$$\begin{array}{l} \text{if } L(M) > \lceil L(o_j), L'(x_i) \rceil \text{ then } x_i.l := M.l \\ \text{elseif } L(o) > L'(x_i) \text{ then } x_i.l := o.l \\ \text{else } x_i.l := x_i.l' \end{array}$$

where $x_i = \langle v, o \rangle$, $L'(x_i)$ represents the initial sensitivity of x_i and $x_i.l'$ is the corresponding sensitivity attribute.

Again, as was the case when sensitivity is increased, it may be necessary to adjust the sensitivity of the entire object.

Note that in the case of structured variables (for example array-like structures) the sensitivity of the entire variable cannot be decreased if a single low-sensitivity element is written. This is not a problem if structured variables are objects contructed from the individual 'element' variables. In such a case only the relevant element variable will be re-labeled.

Polyinstantiate: See section 6.1 for a discussion of polyinstantiation.

SECDB takes the option of *increasing* the sensitivity of the receiving object or variable. In SECDB the security profiles associated with the message are attached to the concerned variable. In effect this means that the variable is re-labeled with the least upper bound of its original sensitivity and the sensitivity of the message—ie the sensitivity of the variable is 'increased' so that it dominates the sensitivity of the active message.

SODA uses polyinstantiation here—see section 6.1 for a discussion of polyinstantiation. SODA also specifies a highest sensitivity that may be associated with any given variable; if the sensitivity of the message exceeds this maximum, the message will also fail (but in this case the database will inform the user that the message failed). Formally, if the maximum sensitivity for a variable x_i is $L_{max}(x_i)$, the semantic action

$$\text{if } L(M) > L_{max}(x_i) \text{ then fail} \tag{4}$$

associated with $M \rightarrow px_i$ will ensure this.

We stated earlier that either the sensitivity of the concerned variable or the concerned object must be increased if the *increase sensitivity* strategy is used. Corollary

6 restricts strategy 2 for existence protected models: Since the sensitivity of a variable may only be increased by increasing the sensitivity of the entire object, it is not possible to re-label variables; the containing object will have to be re-labeled. In an access protected model, it is possible to re-label either the modified variable or the containing object; it seems that it is better to re-label the variable, since it minimises the risk that an (authorised) user attempting to access another part of the object, will be denied, only because this variable contains sensitive information. SECDB therefore re-labels the variable. SODA provides another solution by allowing either the entire object or its instance variables (but not both) to be labeled; in such a model the already labeled entity will simply be re-labeled.

5 Modeling in context

Pfleeger [19] lists four stages in the development of a secure system:

1. *Modeling* is the process where the environment and the concept of security is studied abstractly. This paper is primarily concerned with modeling.
2. In the *design* stage the means of implementing the model is selected.
3. In the third phase *trust* is demonstrated. This is done by demonstrating that the design reflects the model accurately and taking steps to ensure that the code correctly implements te design.
4. The fourth phase is the *implementation* stage where the system is constructed.

The later three stages do not represent primary design alternatives for the construction of a conceptual model. However they must be considered to evaluate a given model.

5.1 Implementation

Most models do not address implementation. However, an impressive model that is difficult to implement is not worth much. We briefly address some considerations leading up to implementation; a thorough study is necessary before detailed comments can be made.

5.2 Design

According to Pfleeger, *design* follows modeling and precedes the demonstration of trust; this section briefly touches design possibilities that makes demonstration of trust simpler.

As always one should strive to make security related code as small as possible—a small section of code is easier to verify, increasing trust and reliability. This may be done by building the database on a trusted computing base (TCB); SeaView [12] is an example of a secure relational database making use of a TCB. The database partitions information according to its sensitivity and stores it in entities (usually files) managed and protected by the TCB. If the TCB has already been verified,

this provides a quick, reliable way to implement a secure database. It also makes it possible to port a secure database from one computer to another, if the other supports a comparable TCB.

Lunt [11] suggested that a secure object-oriented database can be built on top of a trusted relational database. Much work has been done in the field of trusted relational databases which makes this option attractive. Such a database will probably map classes to relations, objects to tuples and instance variables to tuple entries. However, we feel that protection of methods is the most exciting possibilty offered by object-oriented databases and, sadly, nothing exists in the relational model to which methods may directly be mapped.

5.3 Trust

Trust is normally increased by

- Isolating the security functions and making it as small as possible (as discussed above); and by
- Using mathematically sound models.

Both these aspects have been mentioned elsewhere in this paper.

Another interesting possibility is a database where the checking may be done statically before objects are included in the database (thereby eliminating the necessity for run-time security code and this increasing trust; compare [25]). Consider a secure database model that

- Is based on explicit levels, and uses the same label to indicate the clearance and the sensitivity of any labeled entity (X1.1);
- Labels (at least) methods and variables (X2.1);
- Uses the entire access path to determine the authorisation of a message (X3.1) where the combination of two security attributes (*clear*) is the greatest lower bound of the two;
- Only allows a message to activate a method if the security attribute of the message dominates that of the method (X3.1);
- Only allows a method to read a variable if the security attribute of the method dominates that of the variable (X3.1); and
- Only allows such a method to write to a variable if the security attribute of the variable dominates that of the method (X3.3).

In such a model the sensitivity of any message (X3.2) is the same as the clearance (X3.1) of such a message as computed by the rules in table 8. It is therefore not necessary for such a model to keep explicit track of the sensitivity of a message. Obviously such a model employs the *proceed/reject* option to restrict information flow (X3.3).

The described model will not re-label variables or objects since a value will never be written to a variable that is not already sensitive enough. Therefore the security level of any method will dominate that of all variables that it *may potentially* read

(and not only those that it actually reads during a given activation); further the security labels of all variables that *may potentially* be written to will dominate the label of the method. This makes it possible to statically check the security of any object that is added to the database. In this case an underlying monitor is not necessary to check access to sensitive information at run-time. It is only necessary to ensure that the primary accessor (the human operator) only activates methods this accessor is entitled to—that is

- Methods that have a higher security level than the accessor if the method will not send a reply to the accessor; or
- Methods with the same security level as the accessor if the method will send a reply to the accessor.

It is relatively simple to implement a view mechanism that will only allow an accessor to 'see' methods the accessor is entitled to access. In such a model it is only necessary to trust the message passing mechanism; if this mechanism does nothing other than passing messages we can trust the database. However, it needs to be shown that this model is flexible enough for practical use.

6 Remaining issues

In this section we briefly look at polyinstantiation, aggregation and inference, discretionary access controls and integrity constraints—all aspects that need consideration when one defines a secure database model. However, too few models have addressed them to see any clear alternatives.

6.1 Polyinstantiation in an object-oriented database

Thuraisingham [24] described the many faces of polyinstantiation in an object-oriented system:

- Different subjects may see differing values (or contents) in the same object;
- Different subjects may see differing (class) structures for the same object;
- Different subjects may see a different set of methods supported by the same object; and/or
- A single method may have a different definition for different subjects.

Abstractly, one may view a database that supports polyinstantiation as a set of databases that co-exist—one database for every sensitivity level. Rules then describe which 'databases' must be updated whenever an update request is received. Similarly, rules describe from what database a query request must be answered. The axioms and theorems given earlier may be adapted, based on this view of polyinstantiation. The affected productions given while discussing authorisation flow (X3.1) are $M \rightarrow a_i T$, $M \rightarrow p x_i$ and $M \rightarrow g x_i$. In the case of polyinstantiation more than one a_i or x_i may exist. In the first case rules must be given to determine to which method a_i the message must be directed based on $M.c$. In the second case rules must

be given to determine which set of variables x_i (often more than one, but possibly none) must receive the new value. In the third case, rules must determine from which variable x_i the read must be attempted; in some cases more than one value is read and the set of values returned. The functions *class*and *sup*and the corresponding axioms about object-orientation also needs adaptation: the class and superclass may depend on which subject wants to know. The rules for determining the initial sensitivity of a newly created object may also have to be adapted. We do not elaborate at this time.

Clearly, polyinstantiation can cause integrity problems: a user not cleared to access a value, may insert it into the database; if the new value differ from the existing one, it is a good question which value should be viewed as the *correct* value—ie the more sensitive value or the more recent value?

In an access protected model, polyinstantion is not necessary. In that case it is possible to introduce a weaker form of polyinstantiation, such as that suggested by SECDB: Protected entities (primarily methods) may be replicated in a cascaded fashion. When a request arrives at such a protected entity, the appropriate access checks are performed; if access is granted, the request proceeds by accessing the corresponding entity; if access is denied, access to the next 'polyinstantiated' entity in the chain will be requested. This process will be continued, until either an entity is found that that allows access, or until the chain of polyinstantiated entities have been exhausted, in which case the request must be aborted. This form of polyinstantiation enables *ad hoc* entities to be polyinstantiated; it is possible to apply it without the integrity problems associated with 'full' polyinstantiation; however, it is not strong enough to support an existence protected model. As an example where weak polyinstantiation may be useful, consider some political candite's database where the VOTESFORUS object may have a polyinstantiated method NUMBEREXPECTED; this method may return different values depending on whether it is invoked by the candidate, by the candidate's sponsor or by the candidate's press secretary.

Another 'weak' form of polyinstantiation is the view concept as defined by [21] where multiple interfaces may be defined for a single object. As it is described in [21], other 'client' objects may decide through which view it wants to access the object; however, when it is intended as a form of polyinstantiation, security constraints may specify through which view another object accesses the multiview object.

6.2 Aggregation and inference

Aggregation is a security problem because the sensitivity of a number of pieces of information combined is often higher than the sensitivity of the individual pieces.

A related problem, is the problem of inference. Often, if it is possible to obtain some portions of information, it is possible to infer the missing (sensitive) portions. One common example occurs where it is possible to obtain the average salary of a set of employees—if it is possible to make that set very small (say only one employee) the salary of that employee is easily obtainable.

The encapsulation facility of object-orientation provides a natural way to sensitivity label the relationship between objects higher than the individual objects: If an object is classified, the composition of that object is hidden from unauthorised subjects; ie no unauthorised subject will be able to access the instance variables of

the object, and therefore such a subject will not know about the relationship between them which is encapsulated in the object. Similarly, if an instance variable is labeled, the fact that the contents of that variable is related to the rest of the object is protected, although the contents of that variable may be unclassified. See [11] for a discussion. Suitably labeled methods can also be used to protect information regarding relationships.

6.3 Discretionary access controls

Discretionary access controls refer to the rights subjects have to access entities; especially the *discretionary* power they have to decide what restrictions should apply for entities under their control and the power to grant rights to other subjects. For instance, the 'owner' or creator of a file often has the power to grant other users the right to access the file; similarly such an 'owner' may have the power to revoke the access rights of other users to the file.

In a database, discretionary access rights are of less concern: a database contains information shared by many users and usually owned by the enterprise. It is therefore not important that a model for a secure database addresses discretionary access controls. However, a scenario is possible where a central database administrator or system security officer is not responsible for the security of all information contained in the database; in stead a number of 'guardians' may be appointed to be responsible for various sections of the database. Mechanisms enable the existence of such 'owners' of information may be worth investigation; this includes investigation of mechanisms to verify their actions.

Often access contol lists or capabilities are used to implement discretionary access control. Note that we included those facilities in our treatment of mandatory security.

The taxonomy was also used to generate the discretionary access control model described in [18]

6.4 Integrity constraints

Ensuring the integrity of information has two aspects:

- Ensuring that an unauthorised subject does not modify information; and
- Ensuring that an authorised subject does not modify information to unacceptable values such values inconsistent with other entries in the database.

Although we do not address the latter aspect in this work, it is worth pointing out that some models for secure databases also have definite integrity benifits in that area.

One such example is security models that include the whole access path and not only the primary accessor when access rights are determined. In such a model it is possible to ensure that an (authorised) accessor may not access an object without following an 'authorised' path. For example, if a LOCATION object consists of a LONGITUDE and a LATTITUDE object, it is easy for such a model to ensure that LONGITUDE and LATTITUDE are only updated if the request is routed via LOCATION; if the update method of LOCATION sends messages to both the update method

of LONGITUDE and the update method of LATTITUDE, it is impossible to update only one constituent object, leaving the database in an inconsistent state. An inconsistent state may easily result if a subject is allowed to directly send a message to update, say, LATTITUDE.

Note that the described models do not completely solve the integrity problems that arise if an 'impossible' value is inserted into the database by an (authorised) subject; for example if a person's age is given the value 999 years. This may be solved by including integrity constraints—logic rules constraining what values may be inserted into the database.

7 Conclusion

A brief summary of the classification structure proposed in this paper follows. Options considered by us are mentioned.

Labeling semantics

X1.1 Underlying model
We considered explicit security levels, access control lists, capabilities and extensions based on these mechanisms, as well as cobinations of these. Other mechanisms are possible.

X1.2 Protection interpretation
Only two possibilities were considered: existence protection and access protection. Other possibilities were mentioned.

Structural labeling

X2.1 Protectable entities
Work in this paper was based on the premise that any object or class or any facet of such an object or class may be labeled. Note our interpretation of the term *class*. Weaker forms of the restrictions apply if single-level objects are used.

X2.2 Label instantiation
This paper used only one mechanism to label entities: inheritance; in other words, a class is labeled and the labels are inherited by subclasses and instances.

X2.3 Relationship restrictions
The compulsory restrictions are summarised in tables 1 and 2. Other restrictions a model may require and/or enforce were indicated.

Dynamic labeling

X3.1 Authorisation flow
This paper indicated how clearance may depend on the *primary accessor*, the *set of active objects* or the *access path*. The relevant attributes were also identified. In addition to the method a secure database model uses, it thus have to specify up to four additional pieces of information: (1) The format of the clearance attributes ($\Sigma.c$, $a_i.c$ and $M.c$); (2) The format of the sensitivity attribute

($e.l$); (3) For which values of $M.c$ and $e.l$ message M may access entity e; and (4) How are clearances combined, ie define *clear*.

X3.2 Sensitivity flow

A list of rules were given that may be used to sensitivity label messages—see table 8. Models may use variations of these and also additional rules to label messages.

X3.3 Information flow restrictions

Four possible strategies have been identified to prevent information from being compromised as a result of it flowing through the system, namely (1) reject it if the sensitivity of the receiving variable is not high enough; (2) increase the sensitivity of the receiving variable or object if necessary; (3) modify the sensitivity of the receiving variable up or down (but not lower than a fixed lower limit); or (4) use polyinstantiation.

References

[1] DE Bell and LJ LaPadula, "Secure computer system: unified exposition and Multics interpretation", *Rep. ESD-TR-75-306*, March 1976, MITRE Corporation

[2] WH Boshoff and SH von Solms, "A Path Context Model for Addressing Security in Potentially Non-secure Environments", *Computers & Security*, **8**, 1989, 417–425

[3] WH Boshoff, *A Path Context Model for Computer Security Phenomena in Potentially Non-Secure Environments*, Ph.D Dissertation, Rand Afrikaans University, 1989

[4] WH Boshoff and SH von Solms, "Application of a Path Context Approach to Computer Security Fundamentals", *Information Age*, **12**, 2, April 1990, 83–90

[5] A Goldberg and D Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983

[6] B Hailpern and H Ossher, "Extending Objects to Support Multiple Interfaces and Access Control", *IEEE transactions on Software Engineering*, **16**, 11 (November 1990), 1247–1257

[7] TF Keefe, WT Tsai and MB Thuraisingham, "SODA: A Secure Object-oriented Database System", *Computers & Security*, **8** (1989), 517–533

[8] TF Keefe and WT Tsai, "Prototyping the SODA Security Model", pp 211–235 in [23], 1990

[9] W Kim and FH Lochovsky (eds), *Object-oriented Concepts, Databases, and Applications*, Addison-Wesley, 1989

[10] TF Lunt and JK Millen, *Secure Knowledge-based Systems*, Technical Report SRI-CSL-90-04, SRI International, 1989

[11] TF Lunt, "Multilevel Security for Object-Oriented Database Systems", pp 199–209 in [23], 1990

[12] TF Lunt, DE Denning, RR Schell, M Heckman and WR Shockley, "The SeaView Security Model", *IEEE Transactions on Software Engineering*, **16**, 6, 1990, pp 1247–1257

[13] NH Minsky and D Rozenshtein, "A Law-Based Approach to Object-Oriented Programming", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, October 1987, 482–493

[14] M Mizuno and AE Oldehoeft, "An Access Control Language for Object-Oriented Programming Systems", *Journal of Systems Software*, **13**, 1990, 3–12

[15] MS Olivier, *Secure object-oriented databases*, Ph.D. Thesis, Rand Afrikaans University, 1991

[16] MS Olivier and SH von Solms, "An Object-based Version of the Path Context Model", In preparation, 1992

[17] MS Olivier and SH von Solms, "Building a Secure Database Using Self-protecting Objects", *Computers & Security*, **11**, 3, 1992, 259–271

[18] MS Olivier and SH von Solms, "DISCO: A Discretionary Security Model for Object-oriented Databases", *IFIP/SEC'92*, Singapore, 1992

[19] CP Pfleeger, *Security in Computing*, Prentice-Hall, 1989

[20] R Sandhu, "Multilevel Object-oriented Databases", *13th National Computer Security Conference*, Washington DC, 1990, 597–598

[21] JJ Shilling and PF Sweeney, "Three Steps to Views: Extending the Object-Oriented Paradigm" *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, October 1989, 353–361

[22] GW Smith, "Identifying and representing the security semantics of an application", *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, December 1988

[23] DL Spooner and C Landwehr (eds), *Database Security III: Status and Prospects*, North-Holland, 1990

[24] MB Thuraisingham, "Mandatory Security in Object-Oriented Database Systems", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM, October 1989, 203–210

[25] V Varadharajan and S Black, "Multilevel Security in a Distributed Object-Oriented System" *Computers & Security*, **10** (1991), 51–67

[26] P Wegner, "Concepts and Paradigms of Object-Oriented Programming", *OOPS Messenger*, **1**, 1 (1990), 7–87

Appendix

Example 1: SODA

Labeling semantics

X1.1 Underlying model: Explicit (numeric) levels.

X1.2 Protection interpretation: Access protection.

Structural labeling

X2.1 Protectable entities: Either the instance variables of an object or the entire object.

X2.2 Label instantiation: Inheritance.

X2.3 Relationship restrictions: Nothing specified explicitly; normal restrictions for access protected models (table 2) apply.

Dynamic labeling

X3.1 Authorisation flow: Authorisation based on clearance of primary accessor (table 4). Clearance of subjects and sensitivity of entities are integers (range not specified). A subject with clearance $s.c$ may access an entity with sensitivity $e.l$ if $s.c \geq e.l$.

X3.2 Sensitivity flow: The normal rules (table 8) apply, except that the production $M \rightarrow px_i$ has the semantic action $M.u := x_i.l'$ where $x_i.l'$ denotes the initial sensitivity of variable x_i.

X3.3 Information flow restrictions: SODA uses polyinstantiation. Additionally SODA rejects messages that are higher than the specified upper limit for the concerned variable—see equation 4 on page 34.

Example 2: Lunt

Labeling semantics

X1.1 Underlying model: Explicit levels.

X1.2 Protection interpretation: Existence protection.

Structural labeling

X2.1 Protectable entities: Objects and facets of an object (methods and instance variables).

X2.2 Label instantiation: Not specified.

X2.3 Relationship restrictions: Summarised in table 3.

Dynamic labeling

X3.1 Authorisation flow: Message clearance is based on the primary accessor according to property 6—see equations 2 and 3 on page 26 with the accompanying discussion.

X3.2 Sensitivity flow: The sensitivity of a message seems to be equal to the classification level of the subject that sent it.

X3.3 Information flow restrictions: Not specified.

Example 3: SECDB

Labeling semantics

X1.1 Underlying model: Path Context Model (PCM).

X1.2 Protection interpretation: Access protection, but axiom 7 does not apply.

Structural labeling

X2.1 Protectable entities: Layers of an object, methods and variables.

X2.2 Label instantiation: Inheritance.

X2.3 Relationship restrictions: Nothing specified explicitly; normal restrictions for access protected models (table 2 on page 23) apply, except those based on axiom 7.

Dynamic labeling

X3.1 Authorisation flow: Authorisation based on complete access path (table 6 on page 28). Clearance attributes are *baggage vectors* identifying the object, method, domain (processor) and processor integrity state. Clearance attributes are combined by concatenating them. Sensitivity attributes (*profiles*) are (parts of) a formal grammar. A subject with clearance attribute $s.c$ may access an entity with sensitivity attribute $e.l$ if the string $s.c$ is a word of the language specified by $e.l$.

X3.2 Sensitivity flow: The normal rules (table 8 on page 31) apply.

X3.3 Information flow restrictions: Sensitivity of variable is increased.