EGG--10617-1081

DE90 010840

Unifying The Object-Oriented Paradigm With Semantic Data Models

Donald W. Baltz,[1] Thomas Nartker,[2] Kazem Taghva[2]

[1]*E G & G Energy Measurements, Inc.*
P. O. Box 1912
Las Vegas, Nevada 89125

[2]*Department of Computer Science and Electrical Engineering*
University of Nevada, Las Vegas
Las Vegas, Nevada 89154

ABSTRACT

The object-oriented paradigm can be used to model behavior, and to a lesser extent, the structure of a problem domain. Semantic data models describe structure and semantics. This paper unifies the behavioral focus of the object-oriented paradigm with the structural and semantic focus of semantic data models. It presents abstractions to model static and derived data, composite objects, part hierarchies, semantic constraints, and abstractions for identifying behavior. The abstractions keep the model close to the problem domain, are independent of language features, and can be translated into object-oriented, relational or network implementations.

This paper makes three principal contributions. First, a comprehensive set of data structuring abstractions is described. Second, semantic constraints inherent in the graphical representation of the abstractions are identified. Third, abstractions for identifying behavior are described.

1. STRUCTURING ABSTRACTIONS

An approach for modeling the structure, semantics, and operators of a problem domain is described. The approach, called Problem Domain Modeling (PDM), extends notations introduced for the IFO semantic data model [1,7], the extended entity relationship semantic data model [4], and the object-oriented Object Modeling Technique [10].

Structurally, an object is an entity that does not derive its type from any other entity. The type of an object is called an object type. A set of objects is represented by a triangle enclosing its name (Fig 1). Specialized entity

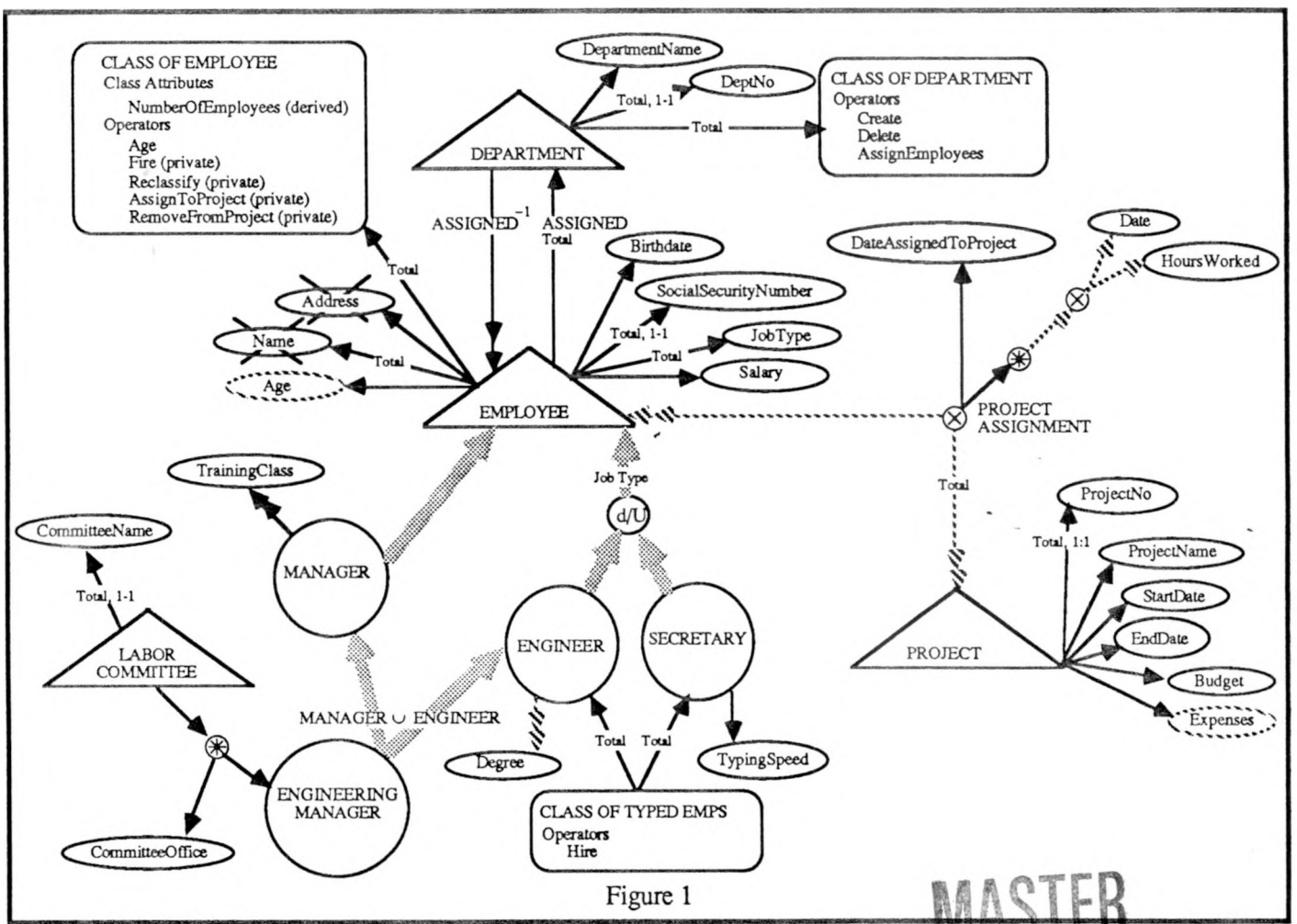


Figure 1

---

sets, which derive their type from other entities, are represented by a circle with a directed edge drawn with a lightly shaded fat arrow (called an inheritance arrow) to the higher-level entity set, which may be an object set or another specialized entity set.

The assignment of attributes is based on the mathematical concept of a function from a domain to a range, where the domain is the object (or entity) set and the range is the set of values for each attribute. Notationally, an attribute function name is enclosed in an ellipse. The single-valued nature of the function is represented by a directed edge to the attribute function name. Attribute function values are described in a separate document.

Annotations *Total* and *1:1* on the directed edge describe the nature of the function. Values for *Total* attributes cannot be null. Values for *1:1* attributes can be used as the domain of only one entity. Edges with no annotation have no restrictions, i.e., they represent *Partial* functions and attribute values can be null. Attribute functions that are *Total, 1:1* functions are candidates for key attributes in a relational implementation. Unless otherwise annotated, specialized entities inherit the attributes of higher-level entities. The functional mapping from the higher-level to the specialized entity is always *Partial, 1:1*. The mapping from specialized to higher-level entity is always *Total, 1:1*.

The aggregation of attributes to form higher-level attributes is called Cartesian aggregation. Attributes so constructed are called composite attributes. The domain of values for a composite attribute is the cross product of several domain sets. Cartesian aggregation is represented by placing an $\times$ through the ellipse, which indicates aggregated attributes are omitted from the graphical representation. The composite attribute *EMPLOYEE.Name*, for example, consist of the attributes *FirstName*, *M I*, and *LastName*. The aggregated attributes can be shown at a lower level of abstraction and are always included in the domain descriptions. Composite attributes can be nested.

Multivalued attributes have multiple values from a domain of values. The domain of values can be simple or composite attributes. Multivalued attributes are represented with a double arrowhead on the directed edge to the ellipse identifying the attribute domain, e.g., *ENGINEER.Degree*, with values BS, Phd, etc. A dashed directed edge indicates an order is imposed, which is identified in the domain descriptions.

Derived attributes [4,6,7] have their value mathematically determined from other attribute values. Derived attributes are represented by a dashed ellipse, e.g., *EMPLOYEE.Age*. Domain descriptions for derived attributes consists of two parts; a structural description and a derivation rule. The derivation rules can be complex and can use previously defined derived attributes. Functional dependencies and derivation rules between derived attributes are described in a separate document.

1.1 Relationships

The representations of one-to-one and one-to-many relationships are also based on the mathematical concept of a function, where the domain is one entity set, and the range is the related entity set. A single arrowhead on the directed edge between related entities represents a *one* cardinality, a double arrowhead represents a *many* cardinality. Both the relationship and the inverse relationship are modeled. The inverse relationship is usually not implemented. Many-to-many relationships, e.g., PROJECT ASSIGNMENT, are represented using the aggregate-constructor (⊗). Attributes can be assigned to relationships modeled with the aggregate-constructor. One-to-many relationships can alternatively be represented using the aggregate-constructor when attributes are associated with the relationship. Dashed directed edges from aggregate-constructors indicate ordered pairs.

1.2 Class Attributes

An object type is treated as an object itself, thereby permitting it to have attributes; i.e., a type is an object. Such attributes are called class attributes. A class attribute, then, is a single attribute that applies to each object in the set. Three kinds of class attributes are identified; *shared-value* attributes, *default-value* attributes [2], and *derived-value* attributes. A shared-value attribute is shared by each object (and specialized entity). A default-value class attribute is shared by each object (and specialized entity) when the object (or specialized entity) has a null value for its similarly named attribute. A *derived-value* class attribute represents summary information aggregated from the set as a whole. Notationally, class attributes are represented by a box with rounded corners. Figure 2 shows the object set CAR having CLASS OF CAR class attributes with *(shared)*, *(default)*, and *(derived)* appended to the attribute names. The shared-value class attribute *NumberOfWheels* applies to all cars; the default-value class attribute *FuelCapacity* applies to those cars that have a null value for their similarly named attribute (because a standard fuel tank is installed in most cars); the derived-value class attribute *NumberOfCars* represents the number of cars in the object set. Class attributes are *total* functions. Object sets that have class attributes have a type that extends to include the type of the class attributes.

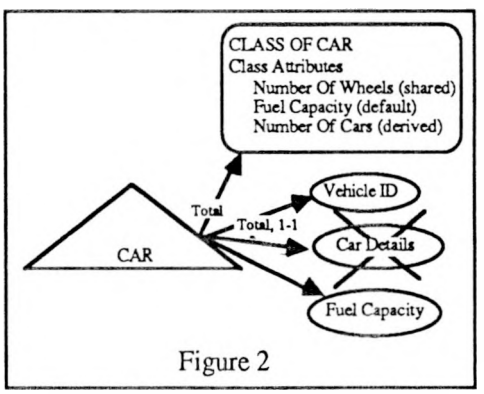


Figure 2

1.3 Specialization

Specialization is an abstraction that models ISA relationships by considering similar entities as a sub-classification of a higher-level set. There are two independent reasons for specialization. First, there are attributes relevant to the specialized entity that do not apply to the more general entity. Second, specialized entity sets can participate in relationships that the more general entity set cannot. When specialization is applied,

the higher-level set is defined so that it contains attributes common to all similar entity sets. Attributes that are not common are assigned to specialized entity sets. Specialized entities may themselves be specialized, forming a hierarchy or lattice, e.g., ENGINEERING MANAGER (Fig 1).

There are three forms of specialization; attribute, predicate, and user defined. Attribute defined specialization is characterized by a defining attribute in the higher-level entity, e.g., *EMPLOYEE.JobType*. Notationally, the name of the defining attribute is placed on the inheritance arrow. Predicate defined specialization is characterized by a defining logic condition, e.g., MANAGER ∪ ENGINEER. Notationally, the defining predicate is placed on the inheritance arrow. ENGINEERING MANAGER has no additional attributes. It is included in the model because (some) engineering managers enter into the LABOR COMMITTEE relationship. User defined specialization does not have a defining attribute in the higher-level entity, e.g., there is no attribute in EMPLOYEE that distinguishes managers from other employees. Entities included in a user defined specialization are identified by the user at the time the entity is created or updated rather than by any condition that may be evaluated. Notationally, inheritance arrows for user defined specializations are not annotated.

Specialized entity sets can be *disjoint*, and/or can *cover* the higher-level entity set. Disjoint entity sets are represented by connecting the inheritance arrows of the specialized entity sets to a small circular node circumscribing the letter *d*, followed by one inheritance arrow from the circular node to the higher-level entity set. *Covering* is represented by connecting the inheritance arrows to a small circular node circumscribing the letter *U*, followed by one inheritance arrow from the circular node to the higher-level entity set. Entity sets that are disjoint and covering have the letters *d/U* in the circular node. Unless otherwise shown, specialized entity sets are neither disjoint nor covering. Figure 3 shows two equally reasonable specialization hierarchies, depending on the problem domain being modeled. Figure 3(a) models the fact that EMPLOYEE is covered by the disjoint sets MALE and FEMALE. Figure 3(b) models an application where the *sex* of an employee may not always be known, perhaps because it is not a field on the job application.

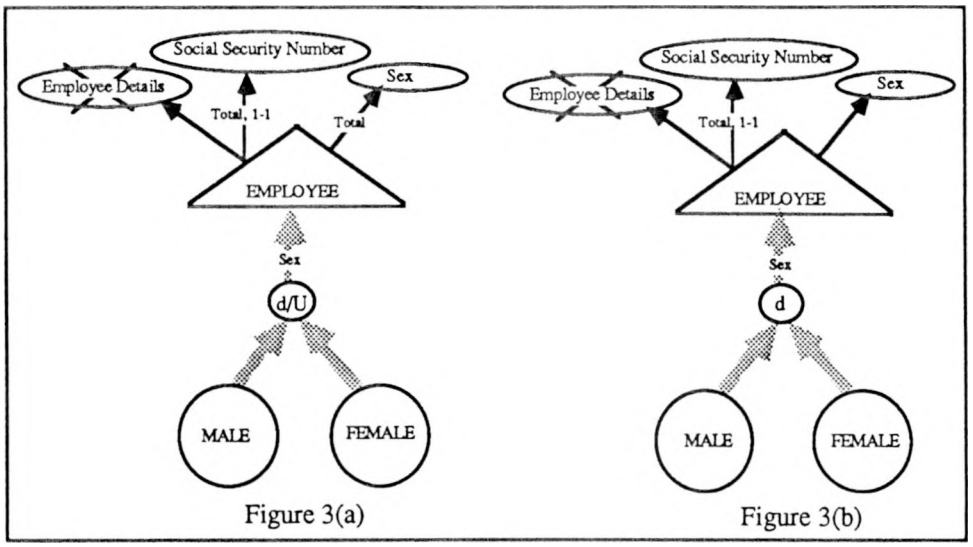


Figure 3(a) Figure 3(b)

1.4 Attribute Inheritance

The concept of attribute inheritance from higher-level entities to specialized entities is closely coupled with the programming language concept of *type* [3,14]. There are two prevailing concepts; type as a prototype and type as a template. Type as a prototype supports *default inheritance*, i.e., inheriting attributes from the higher-level entity is by default unless specifically excluded by the specialized entity so as to be able to deal with exceptions. Type as a template supports *strict inheritance*, i.e., all attributes from the higher-level entity are inherited by the specialized entity, which then adds additional attributes. Another concept, attribute *override*, which applies to both default and strict inheritance, restricts the domain of the inherited attribute in a manner that does not contradict the domain of the higher-level entity. An employee entity set may, for example, have a derived attribute *age* with a domain of 18-120, and a specialized entity set for retired employees that restricts the domain of age to 65-120. Finally, there is the concept of *public* vs *private* attributes. Private attributes are never inherited, whereas public attributes are always inherited (unless excluded or overriden).

An accurate model of a system requires that all forms of inheritance be supported in the description of the problem domain. The details of the actual implementation are left to the later design phase, with the design reflecting the limitations and capabilities of the implementation language. The notation for representing all forms of attribute inheritance is to append *(exclude)*, *(override)*, or *(private)*, to the attribute name. (*Public* need not be a modifier because anything that is not private is public.)

1.5 Composite Objects

One of the fundamental structuring abstractions from semantic data models is the concept of constructing object sets out of other object sets. The abstraction has two forms, composite objects and part hierarchies. The two forms are closely related; a composite object embodies the concept of a collection or an additive whole; a part hierarchy embodies the concept of a more structured, tightly coupled whole. Each adds a different dimension to the concept of an object than that addressed by a specialization hierarchy.

Figure 1 shows the composite object LABOR COMMITTEE, which is composed of ENGINEERING MANAGERs. The relationship from LABOR COMMITTEE to ENGINEERING MANAGER is represented by an attribute function to a set-constructor (⊛). The attribute *CommitteeOffice* (e.g., treasurer) is an attribute of engineering managers who are members of a

labor committee. Composite objects can be nested. Membership in a composite object is also attribute defined, predicate defined, or user defined [5]. Notationally, the defining attribute or predicate annotates the directed edge to the set-constructor. Unlabeled edges are user defined.

Figure 4 shows another form of composite object, a *generalized object*, formed by the disjoint union of otherwise unrelated objects [3]. Registered vehicles are either cars or trucks, but any specific registered vehicle is only one or the other. (The domain of the reference attribute that represents this relationship is either a *Car ID* or a *Truck ID*.) Figure 4 also shows that all trucks must be registered (the attribute function is *total*), but cars need not be registered (the attribute function is *partial*). A generalized object is different than an ISA relationship.

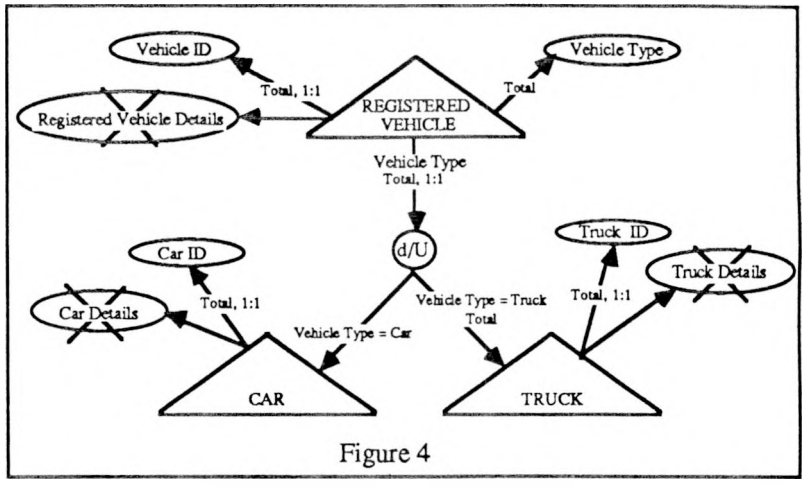


Figure 4

1.6 Part Hierarchies

The objects modeled in CAD/CAM and similar applications are assemblies that are themselves aggregates of other objects. At one level of abstraction an assembly is defined and manipulated as a single object, while at another level of abstraction the complete and detailed aggregated structure may be viewed. An assembly of such objects is called a *part hierarchy* [2,8,9]. A part hierarchy enforces referential integrity that is not enforced with a composite object.

The notation for representing part hierarchies is slightly different than that for representing composite objects because the relationship is stronger. Both the relationship and its inverse need to be modeled and implemented because a part mediates the behavior of its components, and the components affect the behavior of the assembly. Figure 5 models the part hierarchy between CAR and BODY. The dark fat directed edge represents the part hierarchy. The assembly to component relationship is *partial* when the assembly is constructed from the top down and *total* when the assembly is constructed from the bottom-up, and is always *1:1*. The inverse relationship from component to assembly is always *1:1* for a physical parts hierarchy, cannot be 1:1 for a logical parts hierarchy, and can be either *total* or *partial* depending on whether components can exist independent of assemblies.

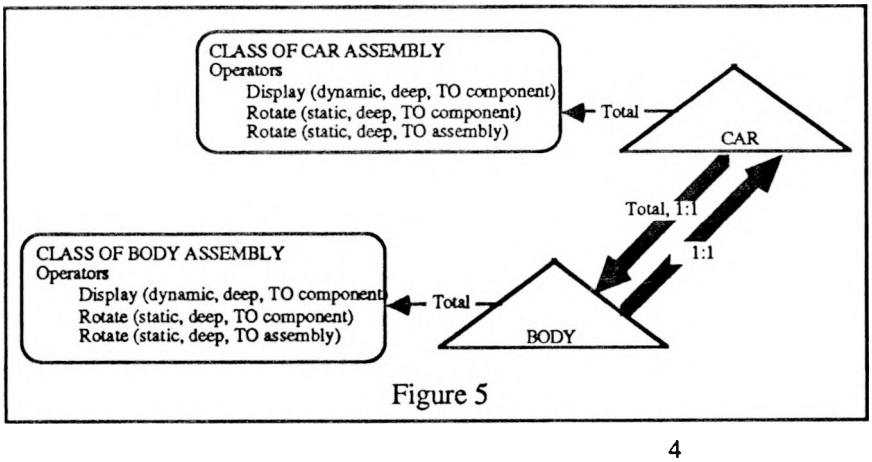


Figure 5

2. SEMANTIC CONSTRAINTS

Semantic constraints are restrictions on entity Create/Delete operations, and on attribute value updates, that guarantee data always reflects the underlying data model. Consider Figure 1. The functional mapping from DEPARTMENT to *DeptNo* is *total*, *1:1*, indicating each department must have a unique value from the domain *DeptNo*. The functional mapping from DEPARTMENT to *DepartmentName* and *StaffingLevel* is *partial*, indicating these attribute values may have null values. The attribute *JobType* is a defining attribute between EMPLOYEE and specialized entities which are *disjoint* and *cover* EMPLOYEE. *JobType*, then, must be *total* (cannot have a null value) if the *disjoint / cover* set constraints are maintained. Furthermore, inserting a new employee will cause an insertion in either ENGINEER or SECRETARY and deleting one of the specialized entities will cause a deletion in EMPLOYEE. The relationship between EMPLOYEE and DEPARTMENT is *total* meaning departments that are the domain of the relationship cannot be deleted. If the relationship were *partial*, deleting a department would require a null value for interobject references from EMPLOYEE to the deleted department.

Figure 6 shows four functional representations of the ASSIGNED relationship between EMPLOYEE and DEPARTMENT that further illustrates the semantic information provided by functional modeling. Figure 6(a) models the fact that all employees are assigned to departments, and all departments are assigned employees. Figure 6(b) models the fact that some employees may not be assigned to a department, but all departments have assigned employees. Figure 6(c) models the fact that all employees are assigned to a department, but some departments may have no employees. Figure 6(d) models the fact that some employees may not be assigned to a department, and some departments may have no employees. The Create/Delete/Update semantics of each is different.

These examples show that the rigorous application of *functions* and *set constraints* identifies semantic constraints for Create/Delete/Update operators.

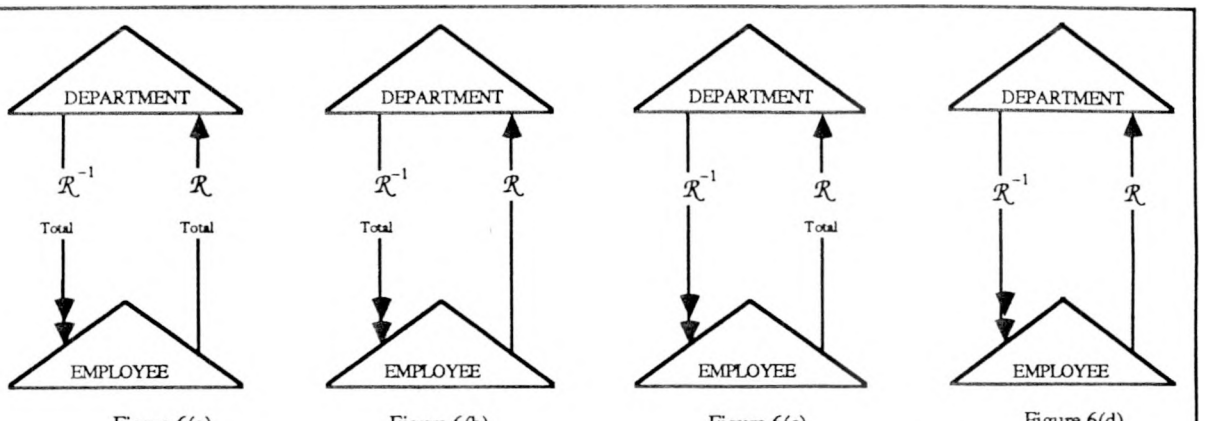


Figure 6(a) Figure 6(b) Figure 6(c) Figure 6(d)

Four representations of the ASSIGNED relationship $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$

Figure 6

Semantic constraints are inherent in the functional representation. They require no special knowledge of the problem domain. The procedure for identifying semantic constraints can be automated if a data description language is formalized that represents the data structure and relationships. The text box on the next page describes semantic constraints inherent in the conceptual representation.

3. BEHAVIORAL ABSTRACTIONS

Adding operators to the structure of an object set defines a *class*. A class encapsulates attributes and operators. Objects are class instances that cooperate, usually through message passing. Objects have state. The state of an object is the value of its attributes. The state of an object is only accessible through the operators defined for the class. A class assumes full responsibility for managing the state of its objects.

The relationship between operators and objects is similar to that of class attributes. Just as one class attribute applies to each object in an object set, so too, one operator applies to each object in an object set. The similarity suggests the same functional notation be used. Figure 1 shows EMPLOYEE having operators identified within the CLASS OF EMPLOYEE symbol. The operator *EMPLOYEE.Age*, for example, returns a value from the domain determined by its specification. There is no distinction between attribute functions and operator functions. Operators to read, write, or modify simple attribute values are assumed and are not usually shown [10]. Operators to read, write, or modify composite attribute values may or may not be shown. Operator specifications are usually described in a separate document.

Just as a specialized entity inherits attributes from all higher-level entities, so too, they inherit operators. The operator *SECRETARY.Age*, for example, returns the same value as the operator *EMPLOYEE.Age* for each secretary. Inherited operators can also be modified in much the same manner as inherited attributes. Operator *override* causes the operator of the specialized entity to be executed instead of the operator of the higher-level entity. Operator *augmentation* causes the operator of the specialized entity to be applied first, followed by the same-named operator in the higher-level entity, or vice versa. The *Create* operator of a specialized entity, for example, must augment the *Create* operator of its higher-level entity because specialized entities have more attributes than their higher-level entities. The distinction between *override* and *augmentation*, and the order of augmentation, is described in the operator specification. *Private* operators are not inherited. Operator semantics, for example, often differ in the substructure of a specialization hierarchy. The modeler may choose to inhibit operator inheritance in lieu of augmenting the operator in the specialized entity to account for slightly different semantics. Operator *exclusion* permits exceptions to an otherwise common set of operators [13]. The notation for indicating operator modification is to append *(private)*, *(exclude)*, *(augment)* or *(override)* to the operator name. Although attribute inheritance occurs only within a specialization hierarchy, operator inheritance is not similarly restricted. Operators are inherited across different object sets, as, for example, the CLASS OF TYPED EMP operator *Hire* (Fig 1).

3.1 Semantic Operators

Semantic operators reduce the gap between the problem and its conceptual representation. Semantic create and delete operators deal with an object in its entirety, i.e., with the object, all specialized entities, and all relationships. The operators *ENGINEER.Hire* and *SECRETARY.Hire*, for example, each create employees who are engineers and secretaries respectively, and assigns them to departments and to projects. Semantic operators to read, write, and modify attribute values are declarative in nature, rather than procedural as would be required if the application were required to perform the join operation in a relational implementation. The operator to read *EMPLOYEE.ProjectNo*, for example, returns a non-empty (the relationship is *total*) list of projects to which an employee is assigned.

3.2 Operator Propagation

Operators in a parts hierarchy can trigger same-named operators of component objects [10]. Triggering iterates over the component objects according to a sequence identified in the operator specification. The component

Semantic Constraints Inherent in the Conceptual Representation

Update Constraints

Constraint U1. Values for *total* attributes cannot have a null value.

Constraint U2. Values for *1:1* attributes can be used as the domain of only one entity.

Constraint U3. The defining attribute for an attribute (or predicate) defined group of subclasses that *cover* a superclass must be *total*.

Constraint U4. The defining attribute for an attribute (or predicate) defined generalized object covered by its member objects must be *total*.

Constraint U5. Modifying the value of an attribute that is used in the derivation rule of a derived attribute makes the derived attribute appear (to the database user) as if its value was dynamically recomputed the next time it is read.

Constraint U6. Modifying the *value* of an attribute that is used as the defining attribute in an attribute-defined or predicate-defined subclass (or composite object) will cause the entity to be added to (or deleted from) the related subclass entity (or object) set as required. This rule applies recursively to all subclass entity sets that are dependent on the (possibly) new subclass entity set.

Constraint U7. In an implementation that references objects by value (instead of distinct object identifiers), changing the value of an attribute that is part of the object identifier (e.g., primary key in a relational system) causes the value to change in all related relations.

Delete Constraints

Constraint D1. An entity that is the domain of a *total* relationship cannot be deleted. This rule takes priority over all other deletion constraints.

Constraint D2. Deleting an entity that is the domain of a *partial* relationship implies either: 1) the reference from the related entity will set to a null value; or, 2) or the aggregate-constructed entity that represents the relationship is also deleted.

Constraint D3. Deleting an entity implies that it is also deleted from all subclass entities (if any) and all composite entities (if any).

Constraint D4. Deleting an entity from a group of subclasses that *cover* a superclass will also delete the superclass entity.

Constraint D5. Deleting an entity that is used in the derivation rule of a derived attribute makes the derived attribute appear (to the database user) as if its value was dynamically recomputed the next time it is read.

Create Constraints

Constraint C1. Inserting an entity that is the range of a *total* relationship implies the relationship with a related entity will be made.

Constraint C2. Inserting an entity into a superclass causes the entity to be inserted into all attribute-defined or predicate-defined subclasses or composite objects as determined by the defining-attribute or the defining-predicate.

Constraint C3. Inserting an entity into a superclass that is covered by a specialization hierarchy causes the entity to be inserted in at least one of the subclass.

Constraint C4. A subclass entity cannot be inserted into a group of disjoint subclasses if it is already an entity in one of the other subclasses.

Constraint C5. Adding an entity into a group of subclasses that *cover* a superclass will also add the entity to the superclass. If the *covering* subclass is user-defined, then additional user information may be required for the insert.

Constraint C6. Inserting an entity that is used in the derivation rule of a derived attribute makes the derived shared class attribute appear (to the database user) as if its value was dynamically recomputed the next time it is read.

objects, themselves, may be assemblies, which may again trigger same-named operators of their component objects. Consider, for example, the assembly, CAR, which is composed of component assemblies BODY, DRIVE TRAIN, and ELECTRICAL SYSTEM, which themselves are assemblies, down to some primitive level. How far should the *Display* operator propagate? Also, the application may need to control the depth of propagation such that the detailed structure of each component can dynamically be made visible or invisible. And what of the inverse relationship? Displaying the detailed structure of a component assembly should not necessarily cause the higher level assembly to be displayed; whereas rotating the component should cause a rotation of the entire assembly, an operation the application should not be able to inhibit. The problem is similar to *shallow* vs *deep* operators that occurs in object-oriented programming.

Different operator propagation requirements are modeled by showing propagation attributes for both relationships and inverse relationships [11,12]. There are three propagation attributes; the *constraint* attribute, which has values *dynamic* or *static*; the *depth* attribute, which has values *none*, *shallow*, or *deep*; and the *link* attribute, which has values *TO component* or *TO assembly*. Figure 5 shows the application may dynamically control the *display* operator (i.e., the detailed structure of a car body may be made visible or invisible when the detailed structure of a car is visible), and the *rotate* operator propagates to all components and assemblies in the parts hierarchy.

4. VERSIONS

Many design applications require the ability to manage multiple versions of an object before selecting the one that satisfies the requirements. Versions are also important for publishing applications, and for historical databases, such as those used in accounting, legal, and financial applications that require access to the past information of the database.

Simplifying the more extensive treatment of Banerjee [2], an object can be versioned in one of two manners, i.e., a *transient version* or the *working version*. A transient version may be created from scratch or derived from an existing transient or working version. A transient version may be deleted or updated at will, or explicitly upgraded to the working version. The working version may be updated at will, but it may not be deleted. It must first be replaced by a transient version that is promoted to the working version.

Associated with each versioned object is a generic object with attributes that include an object identifier; *NextVersionNumber*; *DefaultVersionNumber*; and a set of version descriptors, one for each instantiation of the corresponding versioned object. Individual versioned objects are differentiated by the object identifier along with the version number. *NextVersionNumber* is a derived attribute because version numbers are automatically generated by either the underlying database or the application. *DefaultVersionNumber* is the version number of the working version. In the absence of a user-specified default, the most recent transient version is the default version. The version descriptors, one for each version of the object, includes application specific information.

A versioned object is created by a *Create* command, which also creates a generic object. (The generic object does not have a create command.) A versioned object may also be deleted. When a versioned object is deleted the fact that the version existed is not deleted from the generic object. (The generic object does not have a delete command.) References to a versioned object may be either specific or generic. A reference to a specific object version is statically bound. A reference to a generic version is dynamically bound to a default version of the object.

Notationally, a versioned object is represented by appending *(version)* to the object name, and a generic object is represented by appending (*generic*) to the object name. The functional relationship from the versioned object set to the generic object set is *total* because each versioned object is associated with a specific generic object. The inverse relationship is *partial* because a generic object is not deleted when a versioned object is deleted.

The close coupling between versioned objects and generic objects produces inter-object constraints called inclusion dependencies. An inclusion dependency between attributes X of a set R and attributes Y of a set S specifies the constraint that, at any given instance, the domain of attribute values for X must be a subset (not necessarily a proper subset) of the active domain of values for Y.

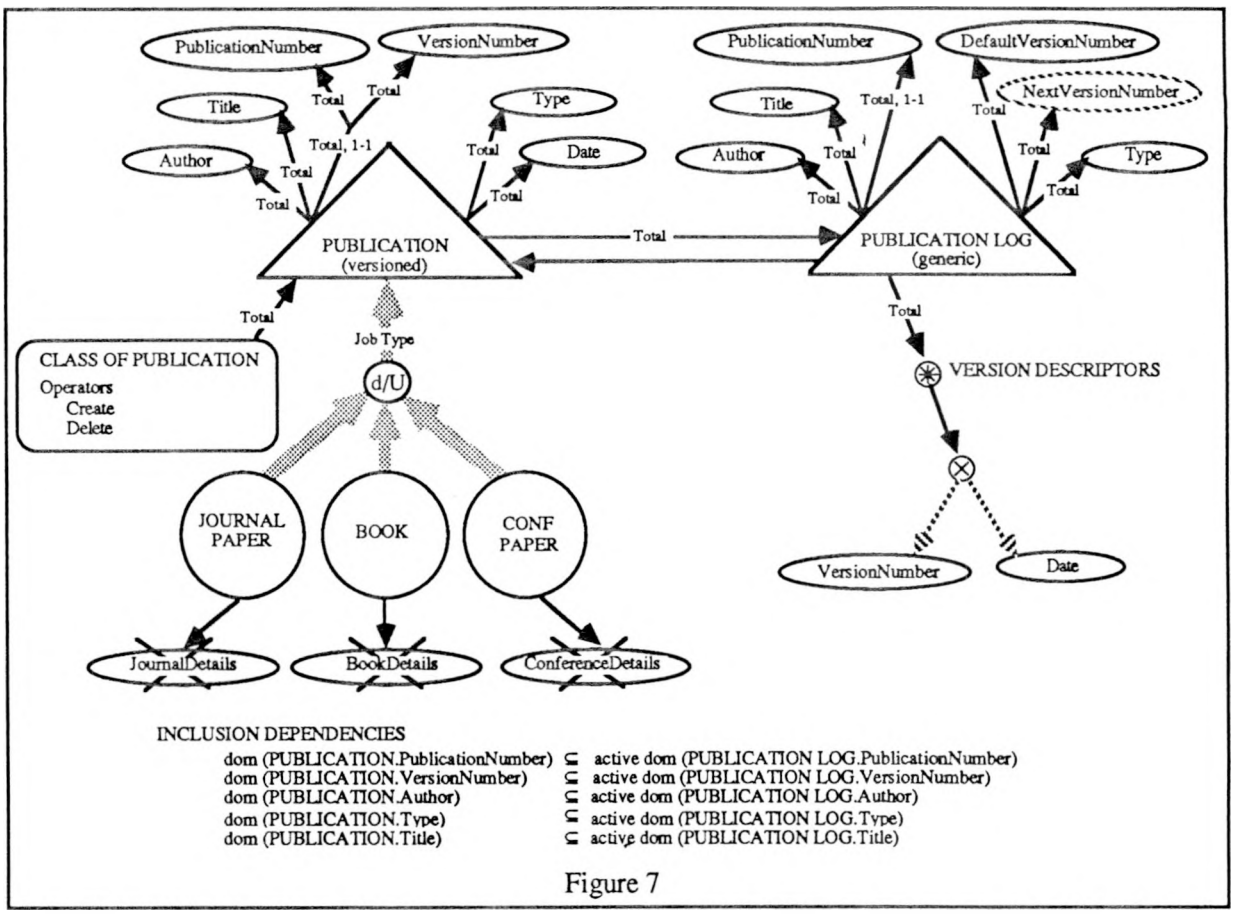


Figure 7

These concepts are illustrated in Figure 7. PUBLICATION is a versioned object of, say, all publication versions that are currently in the distribution network of a publisher. PUBLICATION LOG is the associated generic object, and includes information specific to all versions, even those that are no longer in the distribution network. *PublicationNumber* is the object identifier, which is sufficient to identify individual objects in PUBLICATION LOG, but must be paired with *VersionNumber* to identify individual versioned objects in PUBLICATION. The other attributes are as previously described.

5. OBJECT-ORIENTED LANGUAGES

An object is the fundamental construct of object-oriented programming languages. Objects encapsulate type and operators, and cooperate with related objects by message passing. This is in contrast to more traditional languages that separate type from operators and use a procedural style of programming. The notion of an object from the language perspective is as an embedded, bottom-up component used to construct more complex objects. *Everything is an object.* The notion of an object from the problem domain modeling perspective is as an independent, top-down component which has stepwise refinement by decomposition applied.

More fundamentally, the underlying goals of problem domain modeling and object-oriented programming languages are substantially different. The goal of problem domain modeling is to model the problem domain. An object is viewed as an abstract representation of an entity that exists in the mini-world being modeled. Common *type* (i.e., data structure) is the conscious design decision used to model entities. The goal of object-oriented programming languages is modular software construction, code sharing, and code reusability. Common *behavior* of data is the conscious design decision. Problem domain modeling and object-oriented programming languages approach the problem from different directions and with different goals. The two approaches are complementary.

6. SUMMARY

A comprehensive set of data structuring abstractions was presented. The first abstraction was that of an *object* being an entity with a *type* that is not derived from the type of any other entity. Attribute assignments were made to a small number of objects rather than collecting attributes into a data dictionary, identifying functional dependencies, and synthesizing tables. The assignment of attributes to objects was represented as a function. This added semantic information that is not present in the ER and relational models. Attribute functions that are *total* identify attributes that cannot have a null value. This is enforced in either the data description language of an underlying data base management system, or in the applications code. Attribute functions that are *total, 1:1* identify candidate keys for a relational implementation. The abstraction of aggregating simple attributes into *composite*, *multivalued*, and *derived attributes* kept the model close to the problem domain. The abstraction of *class attributes* modeled real-world situations that are difficult to represent in the relational model. None of these abstractions conflict with the underlying theory of the relational model, nor do they add any additional power. They serve to keep the model close to the problem domain and in a form that can readily be translated into an object-oriented, relational, or network implementation.

Specialization hierarchies (or latices) model roles of an object. Three forms of specialization were identified, i.e., *attribute*, *predicate*, and *user defined*. The need for *public*, *private*, and *exclude* constraints to restrict attribute visibility in a specialization hierarchy was identified. *Composite objects* extended the object abstraction to include objects that represented collections of other objects. *Part hierarchies* added another dimension to composite objects by addressing hierarchically organized collections of interrelated objects.

The use of *functions* and *set constraints* identified semantic constraints for Create/Delete/Update operators. The semantic constraints are inherent in the functional representation of the model. They required no special knowledge of the problem domain. The procedure for identifying semantic constraints can be automated if a data description language is defined that represents the data structure and relationships.

An approach for identifying *behavior* was defined. The need for *public*, *private*, and *exclude* inheritance modifiers to restrict operator visibility was identified, as well as *propagation constraints* and *propagation attribute values* in a parts hierarchy. Operators deal with objects in their entirety; are close to the problem domain; have a broader meaning than the tuple selection, projection, and join operators of the relational model; and are constrained to not violate the Create/Delete/Update constraints of the semantic constraints. Encapsulating operators with object type defined classes.

The graphical notation for representing these abstractions uses a small set of easily recognizable symbols. The notation permits the definition of structures of any complexity, while at the same time keeping the model conceptually close to the problem domain. The domain of problems that can be modeled is more extensive than the domain of either the extended entity relationship or semantic data models.

REFERENCES

[1] S. Abitebol, and R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, Vol 12, No 4, pp 525-565, Dec 1987.

[2] J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H.-J. Kim. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, Vol 5, No 1, pp 3-22, Jan 1987.

[3] A. Borgida, J. Mylopoulos, and H. K. T. Wong. Generalization/Specialization as a Basis for Software Specification. In M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *On Conceptual Modeling - Perspectives From Artificial Intelligence, Databases, and Programming Languages*, pp 87-114, Springer-Verlag, 1984.

[4] R. Elmasri and S. B. Navathe. Fundamentals of Database Systems. Benjamin / Cummings Publishing Co., 1989.

[5] M. Hammer and D. McCleod. Database Description With SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, Vol 6, No 3, pp 351-386, Sept 1981.

[6] S. E. Hudson and R. King. CACTIS: A Database System for Specifying Functionally-Defined Data. In K. Dittrich and U. Duyal, editor, *Proceedings of The International Workshop on Object-Oriented Database Systems, Pacific Grove, California, Sept 23 - 26*, 1986, pages 26-37, IEEE Computer Society Press.

[7] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, Vol 19, No 3, pp 201-260, Sept 1987.

[8] W. Kim, N. Ballou, J. Banerjee, H.-T. Chou, J. F. Garza, D. Woelk. Composite Object Support in an Object-Oriented Database System. *ACM/SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications, (OOPSLA '87) Orlando, Florida October 1987*, SIGPLAN Notices Vol 22, No 12, pp 118-125, Dec 1987.

[9] W. Kim, E. Bertino, J. F. Garza. Composite Objects Revisited. *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, SIGMOD Record, Vol 18, No 2, pp 57-347, June 1989.

[10] M. E. S. Loomis, A. V. Shah, J. E. Rumbaugh. An Object Modeling Technique For Conceptual Design. In J. Bexivin, J.-M. Hullot, P. Cointe, & H. Lieberman, editors, *Proceedings of ECOOP '87: European Conference on Object-Oriented Programming, Paris, France, June 1987*, pp 192-202, Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[11] J. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. *ACM/SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications, (OOPSLA '87) Orlando, Florida October 1987*, SIGPLAN Notices Vol 22, No 12, pp 466-481, Dec 1987.

[12] J. Rumbaugh. Controlling Propagation of Operations using Attributes on Relations. *ACM/SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications, (OOPSLA '88), San Diego, Calif, September 1988*, SIGPLAN Notices Vol 23, No 11, pp 285-296, Nov 1988.

[13] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *OOPSLA 86: Object-Oriented Programming Systems, Languages, and Applications, September 29-October 2, 1986, Portland, Oregon*, SIGPLAN Notices, Vol 21, No 11, pp 38-45.

[14] P. Wenger and S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing and K. Nygaard, editors, *ECOOP '88: European Conference on Object-Oriented Programming, Oslo, Norway, August 1988*, pp 55-77, Lecture Notes in Computer Science 322, Springer-Verlag, 1988.

Unifying The Object-Oriented Paradigm With Semantic Data Models

Donald W. Baltz

E G & G Energy Measurements, Inc.
P. O. Box 1912
Las Vegas, Nevada 89125

Thomas Nartker and Kazem Taghva

Department of Computer Science and Electrical Engineering
University of Nevada, Las Vegas
Las Vegas, Nevada 89154

Presenter:
Donald Baltz

EG&G/EM, Inc.

Unification

Semantic data models describe

- Structure
- Semantics

Object-Oriented Paradigm describes:

- Behavior
- Structure (somewhat)

These concepts can be used to model the structure, semantics, and operators of a problem domain.

EG&G/EM, Inc.

Description of the Problem Domain

- Conceptually close to the problem domain.

  Verifiable by the customer / user

- Easily translatable to code.

  Uses constructs closely aligned tp programming language concepts

- Easily translatable to object-oriented, relational, or network data base implementations.

- Easily translatable to data entry (forms) screen design for populating the data base.

EG&G/EM, Inc.

Functions

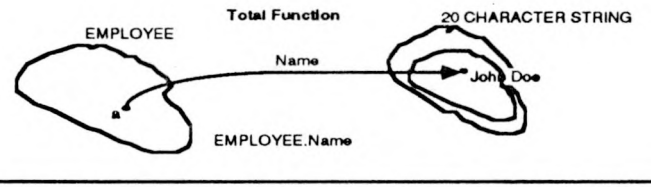
A function, *f*, from set *A* to set *B*, expressed as $f:A \rightarrow B$, is a subset of the Cartesian product $A \times B$ with the property that for each $a \in A$ there is a unique $b \in B$ such that the ordered pair $(a,b) \in A \times B$.

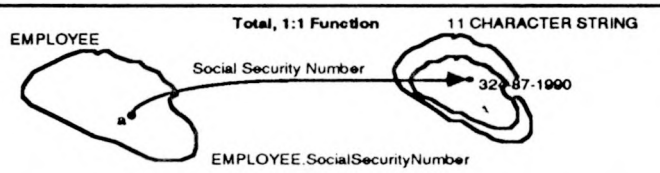
- *f* is sometimes called a mapping of *A* to *B*.
- The set *A* is called the domain of *f*.
- The members of *B* which occur in $(a,b) \in f$ is called the range of *f*.

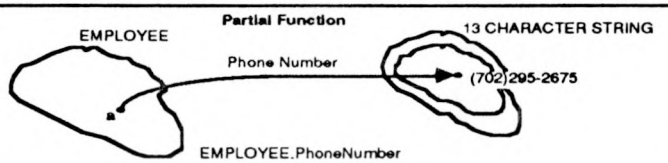
- *Total* function — every member of *A* maps to a member of *B*.
- *Partial* function — some members of *A* do not map to *B*.
- *One-to-one* function (*1:1*) — no two members of *A* map to the same member of *B*.
- Dot notation — *b* is called the image of *a* under *f*, expressed as *a.f*.

EG&G/EM, Inc.

Functions (cont)







EG&G/EM, Inc.

DWB 5

An Object Is an Independent Component of the Problem Domain

- An entity that does not derive its type from any other entity
- State
- Operators
- Cooperate with related objects

EG&G/EM, Inc.

DWB 6

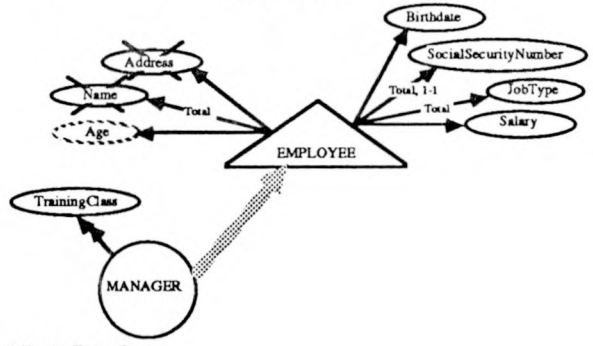
Agenda

- Structuring Abstractions
- Semantic Constraints
- Behavioral Abstractions
- Comparison with Object-Oriented Languages

The manner in which objects cooperate is not addressed.

EG&G/EM, Inc.

Representing Objects, Specialized Entities, and Attributes



Attribute Functions

EMPLOYEE.Name

EMPLOYEE.Address

.

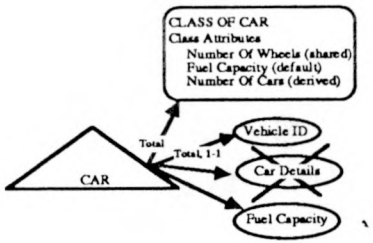
.

EMPLOYEE.Age

MANAGER.TrainingClass

EG&G/EM, Inc.

Representing Class Attributes



EG&G/EM, Inc.

Specialization Models Roles for an Object

There are two independent reasons for specialization.

- There are attributes relevant to the specialized entities that do not apply to the more general entity.
- The specialized entity sets can participate in relationships that the more general entity set cannot.

Specialization models ISA relationships.

EG&G/EM, Inc.

Set Constraints

Two equally reasonable specialization hierarchies

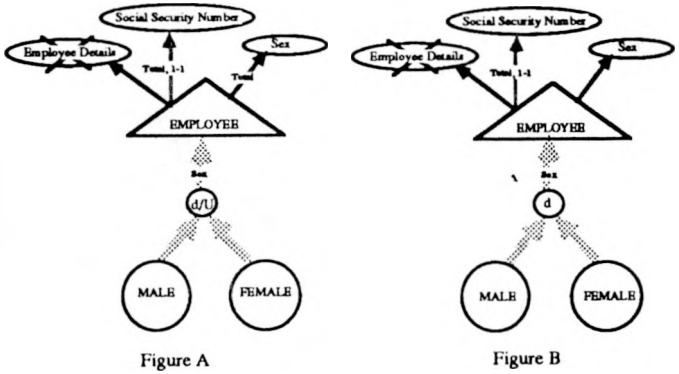


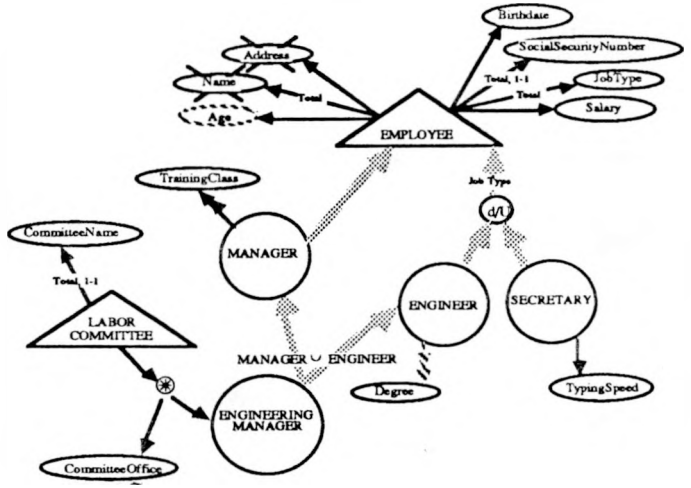
Figure A

Figure B

EG&G/EM, Inc.

DWB 11

Attribute, predicate, and User Defined Specialization



EG&G/EM, Inc.

DWB 12

Attribute Inheritance

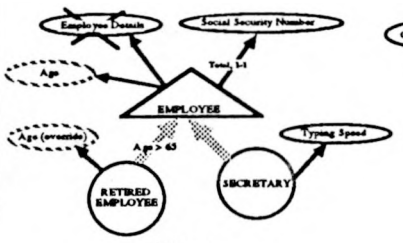


Figure A

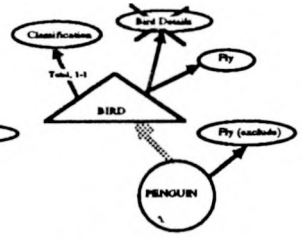
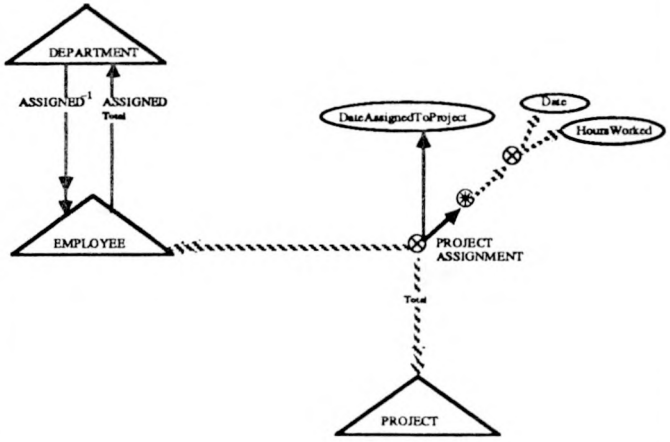


Figure B

EG&G/EM, Inc.

Representing Relationships



EG&G/EM, Inc.

Composite Objects

One of the fundamental structuring abstraction from semantic data models is the concept that objects can be constructed out of other objects.

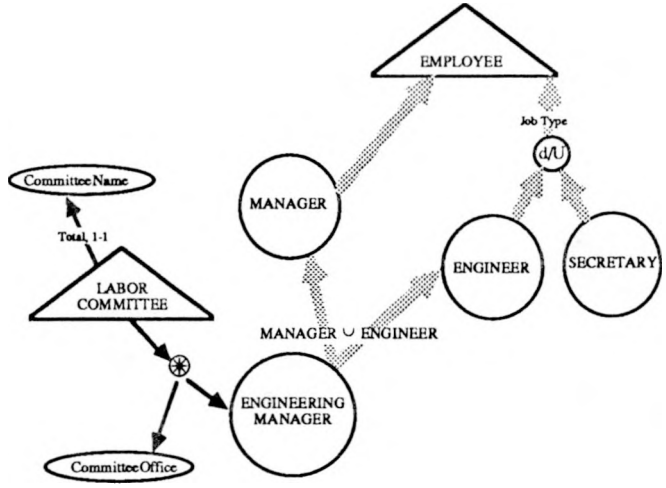
The abstraction has two forms:

- A composite object embodies the concept of a collection or additive whole
- A part hierarchy embodies the concept of a more structured, tightly coupled whole

Each adds a different dimension to the concept of an object than that addressed by a specialization hierarchy

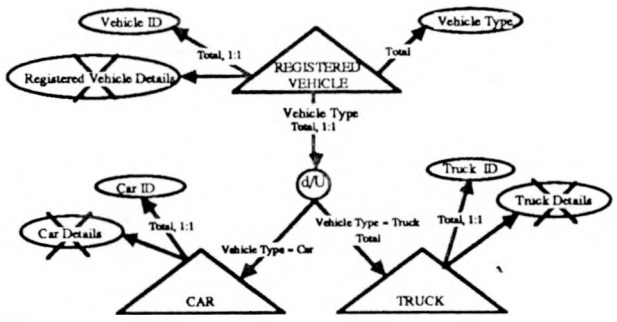
EG&G/EM, Inc.

Composite Objects - A Collection



EG&G/EM, Inc.

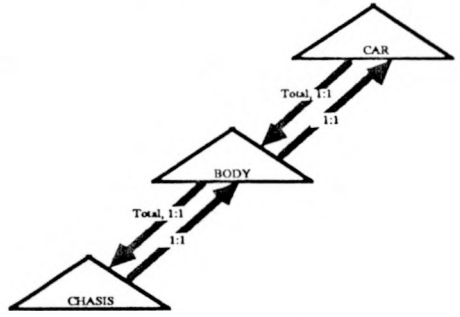
Generalized Composite Object



Generalized objects are formed by the disjoint union of otherwise unrelated objects

EG&G/EM, Inc.

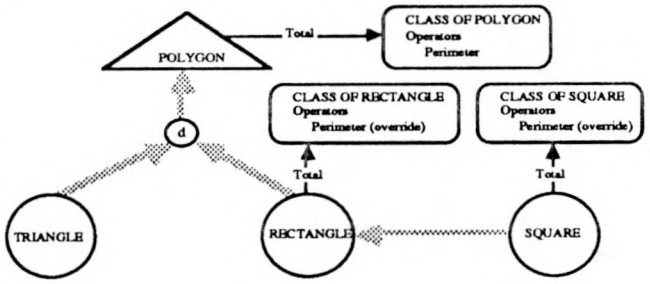
Part Hierarchy



A part hierarchy enforces referential integrity that is not enforced with a composite object

EG&G/EM, Inc.

Operator Override



POLYGON.Perimeter: Sum the length of the edges

POLYGON.Rectangle: 2 × the length of two consecutive edges

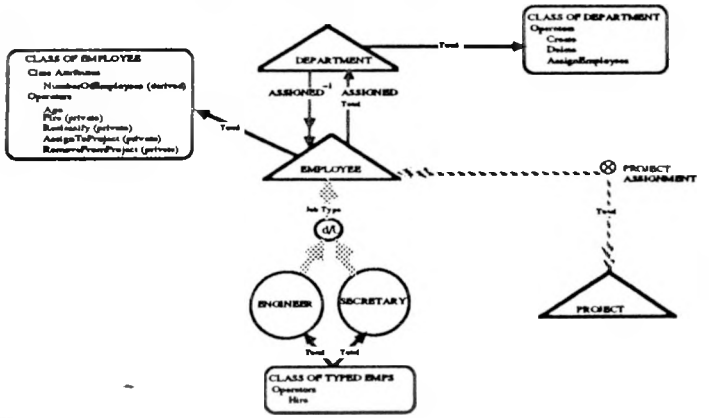
POLYGON.Square: 4 × the length of any edge

EG&G/EM, Inc.

Operator Inheritance (cont)

- *Private* operators are not inherited.
- *Exclusion* permits exceptions to an otherwise common set of operators.

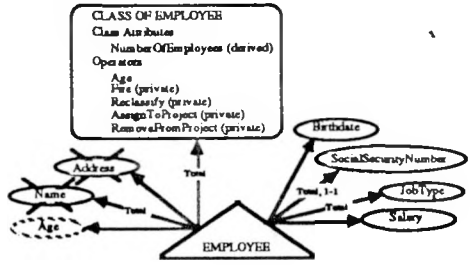
Operators are inherited across different object types.



EG&G/EM, Inc.

Behavioral Abstractions

- The relationship between operators and objects is similar to that of class attributes.
- Just as one class attribute applies to each object in an object set, so too, one operator applies to each object in the set.
- The similarity suggests the same functional notation be used.



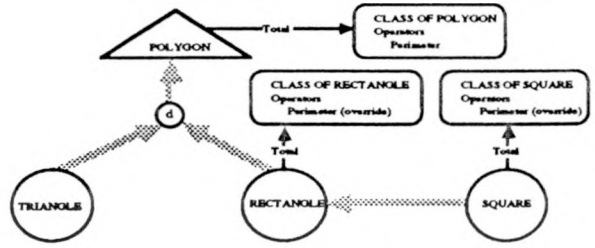
EG&G/EM, Inc.

Operator Inheritance

Inherited operators can be modified in much the same manner as inherited attributes.

- *Override* causes the operator of the heir to be executed instead of the inherited operator.
- *Augmentation* causes the operator of the heir to be applied first, followed by the same named inherited operator, or vice versa.

The distinction between override and augmentation, and the order of augmentation, is described in the operator specification document.



EG&G/EM, Inc.

Semantic Constraints are Inherent in the Functional Representation

- The application of functions and set constraints identifies many semantic constraints for Create / Delete / Update operators.
- They require no special knowledge of the problem domain.
- The procedure for identifying semantic constraints can be automated if a data description language is formalized.
- *User defined constraints are not identified*

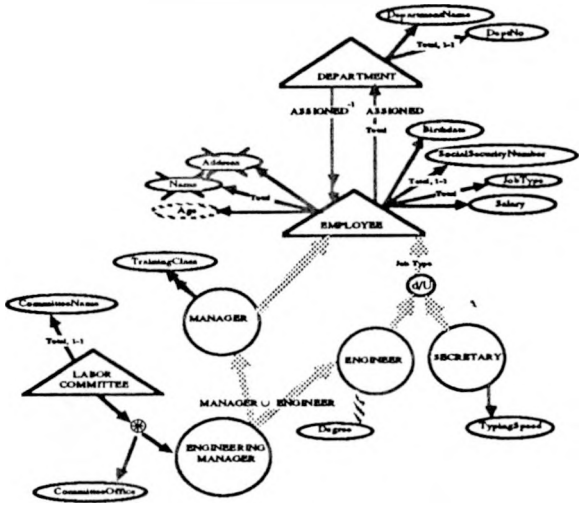
EG&G/EM, Inc.

An Object is an Independent Component of the Problem Domain

- An entity that does not derive its type from any other entity
- State
- Operators
- Cooperate with related objects

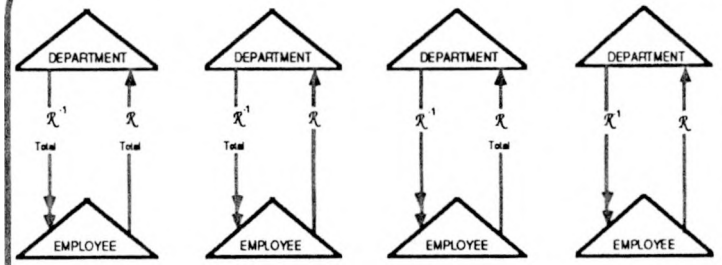
EG&G/EM, Inc.

Semantic Constraints



EG&G/EM, Inc.

Four Representations of a One-to-Many Relationship



Four representations of the ASSIGNED relationship $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$

The Create / Delete / Update semantics of each are different

EG&G/EM, Inc.

Semantic Operators

- Semantic operators reduce the gap between the problem domain and its conceptual representation.
- Semantic create and delete operators deal with objects in their entirety.

  ENGINEER.Hire, for example, creates an employee who is an engineer and makes the assignment to DEPARTMENT and to PROJECT.

- Semantic operators to read, write, and modify attribute values are declarative in nature, rather than procedural.

  EMPLOYEE.ProjectNo, for example, returns a non-empty list of projects to which an employee is assigned.

EG&G/EM, Inc.

Comparison with Object-Oriented Languages

Object-Oriented Languages

- An object is the fundamental construct of object-oriented programming languages.
- Objects encapsulate type and operators, and cooperate with related objects by message passing.
- The notion of an object from the language perspective is as an embedded, bottom-up component used to construct more complex objects.
- *Everything is an object.*

Problem Domain Modeling

- The notion of an object from the problem domain modeling perspective is as an independent, top-down component which has stepwise refinement by decomposition applied.

EG&G/EM, Inc.

The Underlying Goals of Problem Domain Modeling and Object-Oriented Programming Languages are Different

The goal of problem domain modeling is an accurate representation of the problem.

- An object is viewed as an abstract representation of an entity that exists in the mini-world being modeled.
- Common *type* is the conscious modeling decision.

The goal of object-oriented programming languages is modular software construction, code sharing, and code reusability.

- Common *behavior* of data is the conscious design decision.

EG&G/EM, Inc.