A Semantic Database Management System: SIM

by

Saurabh Boyed
(*boyed@cs.utexas.edu*)

Advisor: Professor Philip Cannata
(*cannata@cs.utexas.edu*)

COMPUTER SCIENCES HONORS THESIS
CS 379H
Spring 2003

Department of Computer Science
The University of Texas at Austin

Abstract

SIM is a database management system based on the semantic data model. The goal of this research project was the design and implementation of SIM. SIM, an abbreviation for Semantic Information Manager, uses a data model that thrives in capturing the meaning of the data more than other database models. Thus, this higher-level database model enables the database designer and the users to see a conceptual view of the database. This paper presents an overview of some of the benefits of the semantic data model and emphasizes how SIM incorporates the semantic data model. We describe how SIM overcomes some of the weaknesses of the other database modeling systems. Several SIM and SQL examples are also provided to illustrate this matter and serve as the basis for comparative analysis. We also discuss some implementation considerations and software tools used for design and implementation of SIM. Finally, we propose some hitherto unapplied uses and applications of SIM that have the potential to streamline current database systems.

Keywords

SIM, Semantic Information Manager, Semantic Data Model, Database Management, Semantic Abstractions

1. Introduction

Over the last few decades a number of data models have been developed. Numerous database management systems based on different data models have been made available for commercial use. Intuitively, we can acknowledge that different data models have their own strengths and shortcomings. The use of database management has also progressively evolved over the years ranging from organizational bookkeeping to complex topics such as bioinformatics. Relatively simple database models might be sufficient for organizational bookkeeping, but might fall short for applications that require complex relationships and rich constructs. We believe the semantic database model has significant advantages over some other database models for more complex data manipulation needs. Such advantages motivated us to undertake the research project of designing and implementing the Semantic Information Manager.

SIM was initially developed at Unisys Corporation during the eighties to run on Unisys A Series machines. Lack of applicability of SIM in PC systems during that time led to business decisions to not implement SIM for PCs. This decision made SIM inaccessible to the majority of computer users so we embarked on the development of the SIM database management system for PC users. We have successfully implemented the fundamental features of SIM for PC systems that can demonstrate the novel features of SIM. Although the current version falls short of becoming commercially viable due to the lack of resources and time, it would be very useful for further research in the database community and academia.

This paper proceeds as follows. In section 2 we describe the problems associated with some of the current database models in detail. Section 3 describes the Semantic Data Model and how it overcomes the problems. This section also illustrates how SIM builds on the Semantic Data Model. Section 4 discusses the Object Definition Language (ODL) for SIM, while section 5 discusses the Object Manipulation Language (OML) for SIM. Next, in section 6 we use a simple *University* schema to demonstrate some queries in SIM. Section 7 uses sample queries to compare SIM with SQL. Subsequently, in section 8, we discuss some of the implementation considerations and software tools used for the implementation of SIM. Finally, we present future work on SIM and conclusions. The *University* schema code is provided in Appendix A for reference.

2. The Problem

Different database models have different aspects to them. The relational model is unarguably the most widely used database model today. Nevertheless, it is weak in capturing the semantics of a database. Thus, in the relational model, the semantics have to be separately described making it difficult to manage and use [2]. As in relational modeling languages like SQL, database designers have to convert the real-world structures of the data to low level language constructs requiring an extra level of indirection. Likewise, database users who already have knowledge of the application domain will not be able to perform data manipulation until they have knowledge of the model constructs of the database languages.

New applications are being developed everyday that demand data models that support complex relationships, rich constraints, and large-scale data handling. New fields such as bioinformatics and computer-aided design are evolving with great intensity. The amount of data stored in a database system is increasing astronomically as new applications of database emerge. We believe that the traditional database models -- e.g., relational model -- are not adequate to meet the new demands of database management.

3. The Semantic Data Model and SIM

The Semantic Database Model (SDM) was developed jointly by Michael Hammer from Massachusetts Institute of Technology and Dennis McLeod at the University of Southern California. The SDM is a high-level semantics-based database description and structural formalism for databases [1]. It was developed to address the problems encountered in the relational model as described in the previous section. Although capturing all the semantics of a database in its schema has been unattainable so far, SDM successfully incorporates most of the semantics. Some data models like the hierarchical, relational, and network models, use abstraction that still require the database developers and users to think in terms of the structure of the computer rather than structure of the problems on hand. These data models expose the low-level limitations of the computer. SDM, however, focuses on real-world perception of the problems and the relationship between them. It does not expose the low-level limitations that are inherent to computers.

SDM also facilitates performing queries on a database from different perspectives, which can prove beneficial to users who have different perspectives of a large database with complex relations. Although giving different perspectives to the database users would lead to some redundancy of data storage and manipulation, it will have enormous benefits for better understanding of the database relationship. Moreover, it can dramatically improve the efficiency of queries made, as we will see in later sections. The Relational data model fails to give this advantage to its users, as they often have to perform complex operations like multi-table join.

The data model used by SIM is closely related to SDM. As per [1, 3, 4], the fundamental features of semantic models include entities, relationships between entities, integrity constraints and abstraction. SIM not only incorporates all these fundamental features of a semantic model but also adds some of its own features to make it more robust and user-friendly. SIM uses classes to represent a collection of entities of the same type. Entities are objects defined in the database that have common characteristics and represent a conceptual component of the problem at hand. SIM presents two types of classes: *base class* and *subclass*. The base class is the most generalized form of a class, while a subclass represents further specialization of one of more *superclasses*. SIM also facilitates multiple inheritance; that is, a subclass can have multiple superclasses. It follows a hierarchical convention; the hierarchy has only one root that is the base class and the rest of the nodes are subclasses. A subclass can have multiple subclasses above it in the hierarchy as superclasses. With the use of inheritance among classes, SIM introduces a level of abstraction.

The building blocks of classes are attributes. There are two basic types of attribute in SIM: *data-valued attribute* (DVA) and *entity-valued attribute* (EVA). *Data-valued attributes* are properties that each entity of the class possesses. There can be three types of DVAs: *primitive types* that are built-in data types, *compound types* that are composed of multiple primitive types, and *symbolic types* that are analogous to enumerated types in programming languages. EVAs are used to define relationships between entities. There are two classes of interest for each EVA. One is the class that defines the EVA, also referred as the perspective class; the other is the class that the EVA points to, referred to as the target class. In some cases, the perspective class and the target class can be the same class resulting in an entity pointing to another entity of the same class type. EVAs are also bidirectional; thus, for every EVA, there is an inverse EVA. This formation provides the advantage of performing queries on a database from the perspective of either of the two classes involved. Both the DVAs and EVAs can be associated with special constraints to capture much of the semantics of an application. These special constraints come in the form of attribute options.

The final constituent used to capture the semantics of the application is the *verify* declaration. The *verify* declaration should be used to handle special cases of integrity constraints. Thus, SIM uses class hierarchy, EVA and DVA constraints, and *verify* declarations to capture the semantics of the application to the greatest extent possible.

4. Object Definition Language for SIM

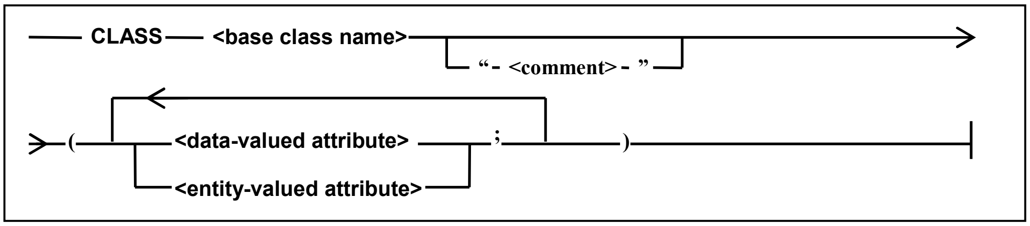
This section describes the fundamental Object Definition Language (ODL) features we have implemented for the current PC version (V 1.2.x) of SIM. The ODL for any DBMS specifies the syntax for defining elements in its schema. The current version of SIM ODL covers the essence of SDM. Now we describe each of the ODL elements that have been implemented in SIM. We have used railroad diagrams for precise syntax reference as it is supported by SIM V 1.2.

4.1 Classes

A class is a structural definition of entities with common characteristics. It is used to store a collection of entities of the same basic type. Entities by themselves are not defined in the schema of a database. In SIM, there are two types of classes, *base classes* and *subclasses*. A base class is a type of class that is not defined in terms of other classes, while a subclass is defined in terms of other classes, also referred to as its *superclasses*. We define a class by associating it with attributes that describe the class. The attributes can be data-valued attributes (DVAs) or entity-valued attributes (EVAs) which we will describe in the next sub-section.

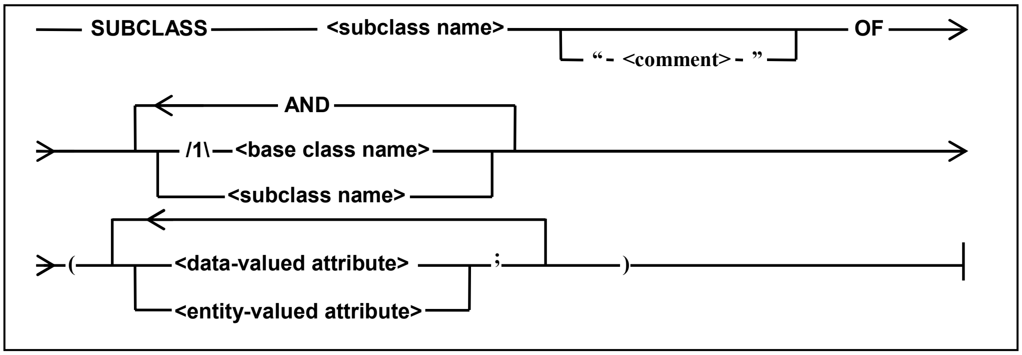
SIM gives us the ability to arrange classes in hierarchies. A hierarchy can be presented by defining a collection of classes that has subclass and superclass relationships among its classes. SIM allows a subclass to have multiple immediate superclasses. All class hierarchies must contain exactly one base class. Thus, all superclass paths in a hierarchy must end at the same base class. Also, SIM does not allow hierarchies to be cyclic, as a subclass cannot have itself as its own superclass.

The following railroad diagram illustrates the syntax for defining a base class:



Syntax element <base class name> represents the unique name of the base class and <comment> is an optional brief description of the class. <data-valued attribute> and <entity-valued attribute> represent DVA and EVA attributes respectively; the syntax is presented in section 4.2. We can define multiple attributes, either DVA or EVA, in a class definition by separating them with semicolons.

Now we present the syntax for defining a subclass.



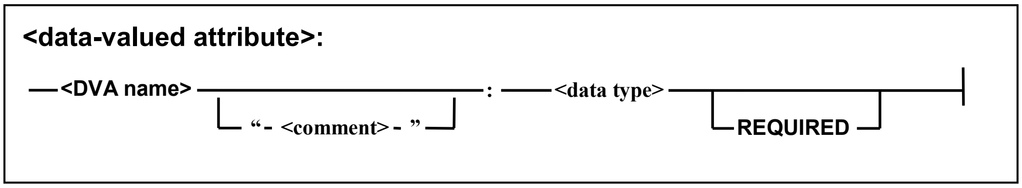
The key difference between defining a base class and subclass is the representation of superclasses. The "OF/AND <base class name>" and "OF/AND <subclass name>" syntax elements are used to represent the superclasses of the subclass being defined. It should be noted that "/1\" means that at most one base class can be used in defining the superclasses of a subclass. Also, the order of superclass definition is not relevant.

In SIM, each base class entity is associated with a unique surrogate value that is assigned by the system. Surrogate values that are totally maintained by SIM and are transparent to its users are used to connect pieces of an entity to form a single entity. EVA relationships between entities also use surrogate values for association. Surrogate values are also used to distinguish between entities, even those with the same attribute values.

4.2 Attributes

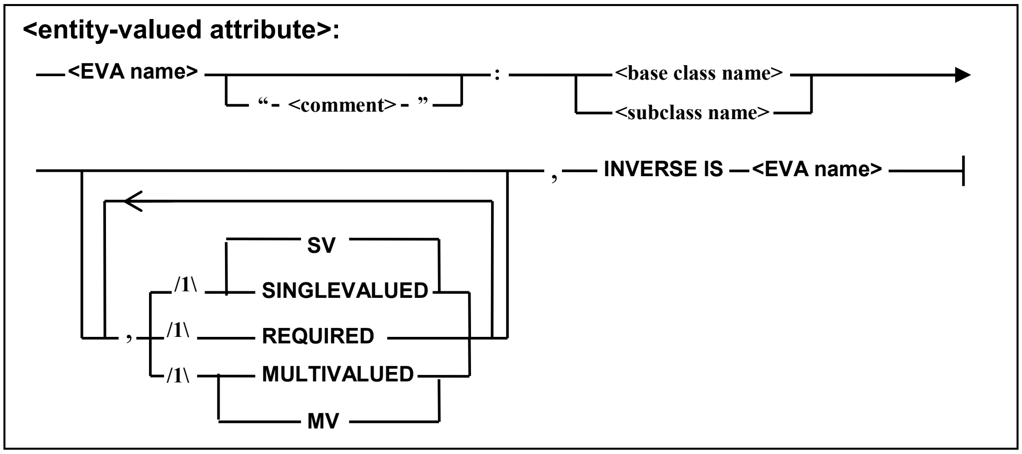
Class structures are comprised of attributes. Attributes are properties that all the entities in a class possess. SIM provides two basic attribute types: data-valued attributes (DVA) and entity-valued attributes (EVA). A DVA is used to store the data of one or more entities that participate in the class in which the DVA is defined. In the current PC version of SIM, a DVA uses only primitive data types as discussed in the next subsection. An EVA is used to correlate entities of one class (perspective class) to that of another class (target class). In a special case, both the perspective class and the target class can be of the same type, which leads to an entity referencing another entity of its own type. The relationship between entities of a target class and perspective class can be one-to-one, one-to-many, many-to-one or many-to-many. Every EVA also has a corresponding inverse EVA. SIM guarantees that an EVA and its inverse EVA automatically stay synchronized, thus eliminating the potential for dangling inverse references.

The following railroad diagram can be used as a reference for the syntax for defining a data-valued attribute inside a base class or subclass.



The REQUIRED keyword is used to ensure that the DVA value is never NULL. The syntax for expanding <data type> is shown in the diagram in the next subsection. The <comment> and

REQUIRED constructs are optional. Next, we give the railroad diagram for an entity-valued attribute.

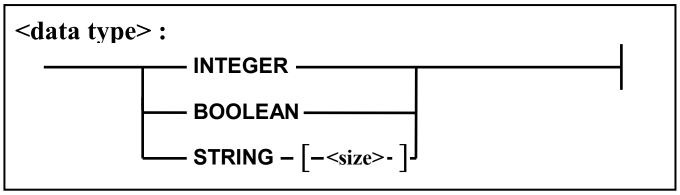


The <base class name> or <subclass name> element will be used to refer to the target class. The perspective class will be the one in which the EVA is defined. Next, we indicate the EVA relationship type. The SINGLEVALUED (SV) option is used if the EVA can point to only one entity in the target class. The MULTIVALUED (MV) option is used if the EVA can point to multiple entities in the target class. SV is used by default if neither of the relationship types is mentioned. Again, we can use the REQUIRED keyword to guarantee that the EVA always references some entity in the target class. Finally, we have to specify the name of the corresponding inverse EVA for the EVA being defined.

From the perspective of a class, an attribute can be referred to as immediate, inherited or extended. The immediate attributes are the ones that are defined within the declaration of the perspective class. Inherited attributes are all the attributes that are defined in the superclasses of the perspective class, while the extended attributes of a perspective class are the attributes that have to be referenced by navigating through an EVA reference.

4.3 Data Types

The current version of SIM supports three primitive data types: INTEGER, BOOLEAN and STRING. The INTEGER data type represents positive and negative integer values. The BOOLEAN data type is used to store logical true and false values. The STRING data type is used to store a sequence of characters of a specified size. A DVA must use one of these three types. The following is the syntax for data types:



The SIM constructs that are not yet part of the current PC version of SIM are Class Attribute, Constructor Type, User Defined Type, Index, Subrole, Verify and the following primitive data types: REAL, CHAR, KANJI, TIME, DATE, and NUMBER. A complete and detailed description of the SIM ODL is outside the scope of this paper and can be found in [5].

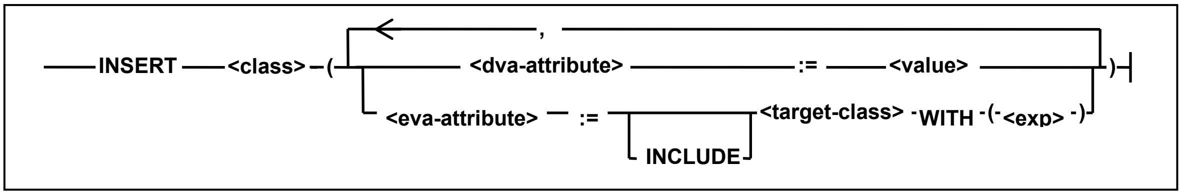
5. Object Manipulation Language for SIM

We now describe the Object Manipulation Language (OML) that is supported by the current version (v1.2.x) of SIM. The OML for a database management system specifies the syntax and interpretations for performing queries and modifications to a database. While the ODL for SIM is based on SDM, SIM has developed its own OML. Again we present railroad diagrams for precise syntax reference. The syntax represented by the railroad diagrams reflects the current version of SIM.

5.1 Insert

We begin our discussion of OML constructs with the INSERT statement, which is the most basic type of construct in OML. An INSERT statement is used to insert a new entity into a class. The INSERT operation can be performed on either a base class or a subclass. If it is performed on a subclass, then appropriate entities are automatically added to its superclasses. In an INSERT statement, all the required attributes have to be assigned some non-NULL value, or else the operation will fail. The non-required attributes that are not assigned any values will automatically be assigned NULL values.

The following diagram provides the syntax for an INSERT statement:

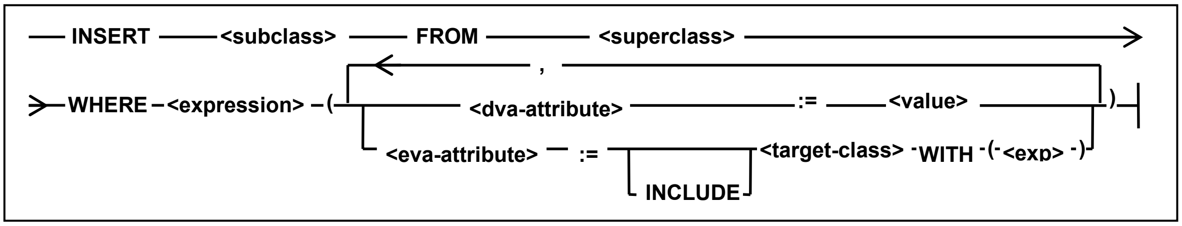


Inside the parenthesis we assign values to the immediate and inherited attributes of the class. The DVA attributes are assigned values depending on the DVA data type. To assign values to an EVA, the INCLUDE keyword has to be used if the EVA is multi-valued. <target-class> is used for specifying the target class name and <exp> refers to a Boolean expression that qualifies entities to which the EVA should point.

5.2 Insert from super class

This section describes a second version of the INSERT statement which allows an entity to be added to a class from its superclass. The class in which the entity is inserted can be many levels below the superclass it is inserted from. If no entity exists between the inserted class and the super class, SIM creates intermediate entities. These new intermediate entities can also be assigned values in the assignment list of the INSERT statement.

The following is the railroad diagram for the second type of INSERT statement for adding an entity from another class:

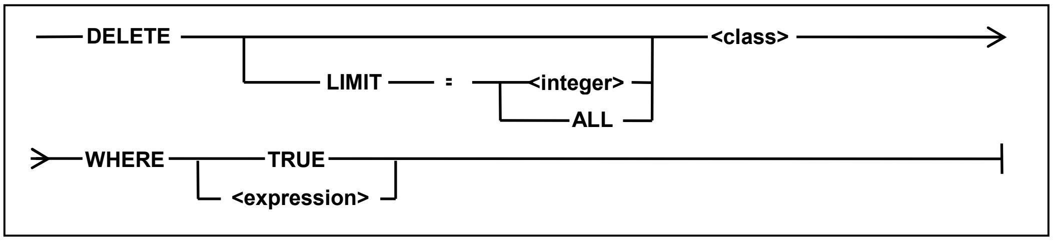


In the above diagram <expression> is a Boolean expression that qualifies a single entity from the <superclass>. An entity has to uniquely satisfy the conditional expression; otherwise, the insertion operation is not performed. The satisfying entity is extended to the <subclass> that is in the same hierarchy. The assignment statements are similar to the INSERT version described in section 5.1.

5.3 Delete

The DELETE statement is used to remove entities from a class. The deletion statement removes selected entities from the indicated class and all its subclasses. It does not remove the selected entities from any of the superclasses of the indicated class.

Below is the diagram for syntax reference of the DELETE statement:

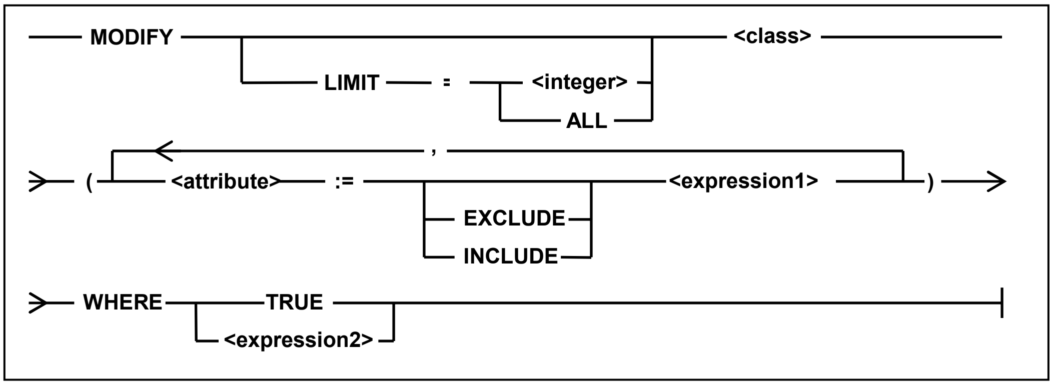


The LIMIT clause is used to create an upper bound on the number of entities to be deleted. If the number of entities selected to be deleted is more than the number specified by the LIMIT clause, then no change are made to the database. The keyword ALL can be used to delete all the entities that satisfy the delete condition without setting any upper bound. If the LIMIT clause is not defined, the upper bound is set to one by default. <expression> must evaluate to a Boolean value.

5.4 Modify

The MODIFY statement is used to change the value(s) of the existing attributes of an entity. From a class's perspective, only its immediate and inherited attributes can be modified. Extended attributes cannot be altered. If we use attributes as a part of the assignment expression, the attribute values will resolve to what they were before the start of the modify query. Thus, the MODIFY statement cannot refer to the result of its own modification.

The following diagram shows the basic structure of a MODIFY query:

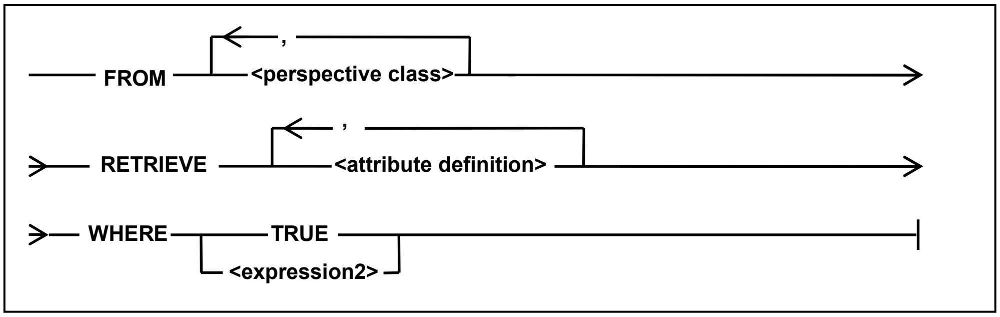


Here again we use the LIMIT clause to limit the number of entities being modified. If the LIMIT clause is omitted, only one entity can be modified. The EXCLUDE and INCLUDE keywords are used for operations on multi-valued attributes. The type of <expression1> depends on the data type of the attribute. For example, if the attribute being assigned values is a DVA of type INTEGER, then <expression1> should resolve to an integer value. The <expression2> clause, which is used to select the entities that are to be modified, should always be a Boolean expression.

5.5 Retrieve

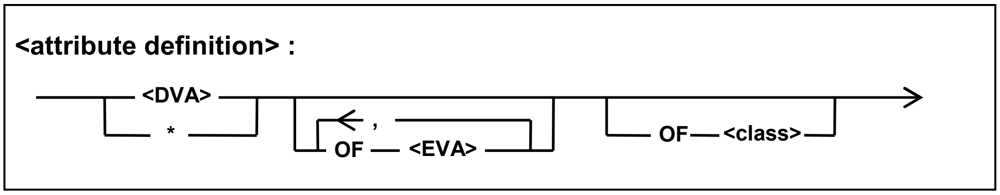
The RETRIEVE statement is used for data retrieval from a database. The RETRIEVE statement starts with a FROM clause where we define all the perspective classes. If more than one perspective class is used, a join operation is performed between the entities of the different perspective classes. After perspective class names, we have to mention the attributes we want to display. The immediate, inherited, and extended DVAs of the perspective classes can be displayed. EVA values cannot be displayed directly but are used to refer to extended DVA attributes. Extended DVA attributes with multiple levels of indirection can also be retrieved. The RETRIEVE query is most efficient when only a single perspective class is used. Use of multiple perspectives classes is less efficient. So, if multiple perspective classes are needed for the majority of queries, then the database schema should be modified by adding new EVA relations, so that a single perspective class can be used.

The following is the railroad diagram for the RETRIEVE statement:



The <expression2> clause evaluates to a Boolean expression. <attribute definition>s are used to list the DVAs that we want to display in order.

In the next diagram we show how a DVA can be qualified using the class name and EVAs.



If two different classes have an attribute with the same name, we can qualify the attribute we want to use by using the "OF <class>" clause. The target class can be reached by navigating through multiple EVAs. Finally, the <DVA> clause can be used to specify the DVA to be displayed. An asterisk '*' can be used to display all the immediate DVAs of the referenced class.

5.6 Expression

Expressions are part of all the OML statements. Expressions in SIM can return values of type Boolean, Integer or String. Expressions are comprised of conditional, relational, and arithmetic operators and the corresponding operands. The conditional expression constructs are OR, AND, and NOT; relational expression constructs <, <=, >, >=, = and <> are supported; arithmetic operations can be performed on +, -, * and DIV operators. The operands in the expression can be of type String literal, Null literal, Integer literal, Boolean literal or Identifier trail. Nested expressions are also supported.

A String literal is a sequence of characters inside double quotes. The keyword NULL is used to refer to null literal. Integer literals can either have a positive or negative integer value. Boolean literals have the value of either *true* or *false*. Finally, Identifier trail is used to refer to immediate, inherited, and extended attributes of a class. The notation for qualifying an attribute is the same as the one specified for the <attribute definition> clause in the previous sub-section.

The current version of SIM does not support functions of the following types: arithmetic functions such as ROUND and ABS; aggregate functions like AVG, MAX, and SUM; and String functions like EXTRACT and LENGTH. It also does not implement ISA role testing; All, Some and None qualifiers; Inverse Is and Transitive functions.

As has been described above, most of the fundamental features of SIM OML are supported in the current version. Although not complete, this version is comprehensive enough to support an array of complex database applications. A more comprehensive and in-depth description of the SIM OML can be found in [6].

6. The University Database

In this section we use a simple *University* database to illustrate the features and uses of SIM. The schema code is provided in Appendix A, towards the end of the paper.

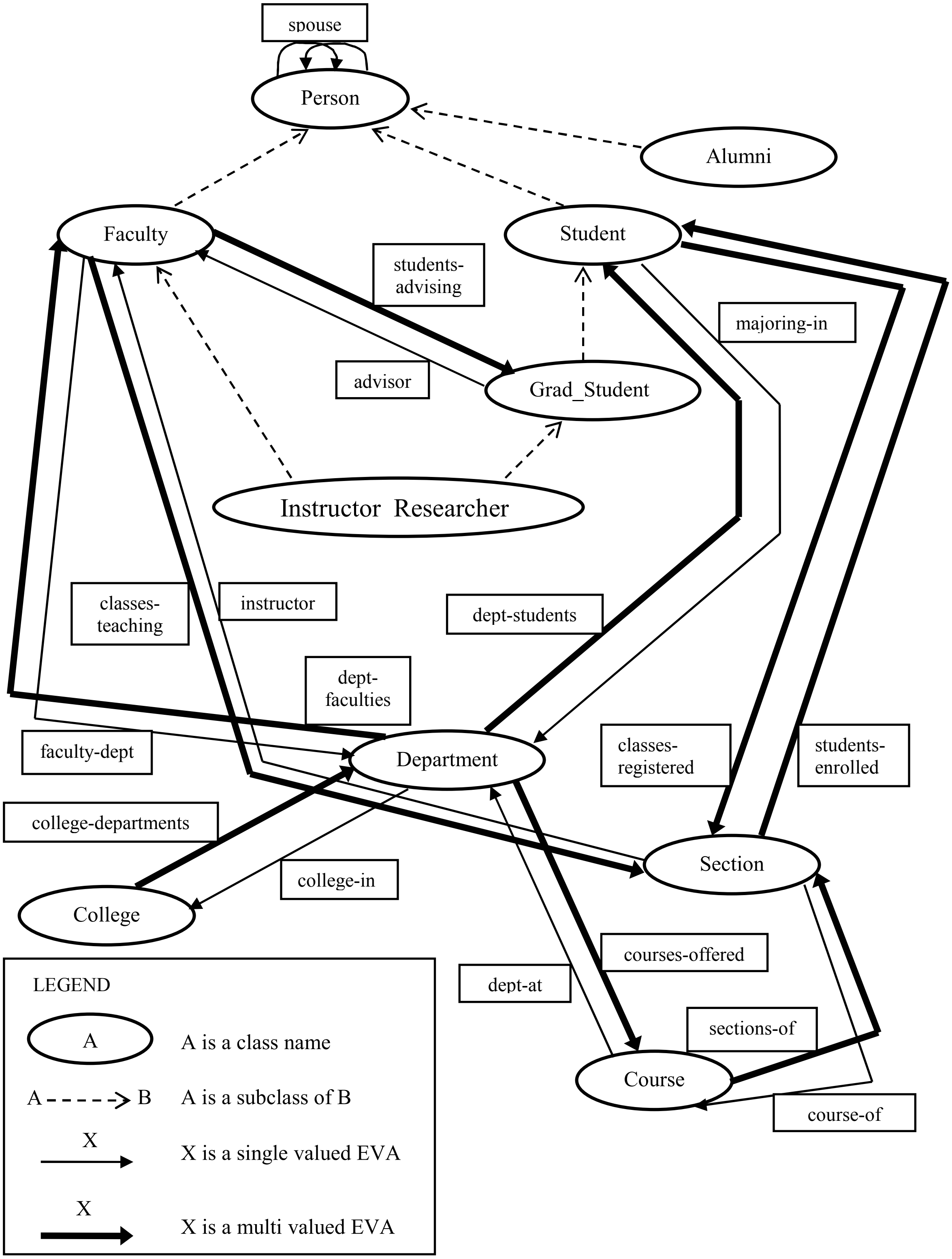


Figure 1: *University* database diagram

The *University* database diagram gives us an overview of the structure of its schema. This simple schema captures the essence of SIM. It will be used to illustrate the concepts of single inheritance and multiple inheritance. We will also encounter one-to-one, one-to-many, and many-to-many EVA relationships. The base classes in the schema are *Person*, *Department*, *College*, *Section* and *Course*. The subclasses are *Faculty*, *Student*, *Grad_Student*, *Instructor_Researcher* and *Alumni*. It should be noted that *Instructor_Researcher* uses multiple inheritance, as it inherits from both *Faculty* and *Grad_Student*.

In our *University* database we have a number of EVA relationships. For example, *classes-registered* is an EVA of class *Student* and it points to the *Section* class. As all EVA relationships are bidirectional, we have a corresponding inverse EVA *students-enrolled* in *Section* that points to the *Student* class. The thicker solid line arrow represents that this is a many-to-many relationship. Thus, as expected, a *Student* can point to multiple *Section* entities and a *Section* entity will be able to point to multiple *Student* entities. As we have mentioned before, we can take advantage of multiple perspectives to retrieve required information. We will be able to select a section and directly retrieve all the students registered for that class with a simple retrieve operation. Similarly, from a student perspective, retrieving the classes he or she is registered for will be relatively simple and efficient. It should be observed that EVA *spouse* is a special case as both the perspective class and target class is *Student*.

We have not represented any DVAs in our database diagram for abstraction reasons and to not clutter the diagram. The DVAs can be looked up from the schema code in Appendix A. In the *Person* class declaration, *person-ss, first-name, last-name* and *us-citizen* are DVAs of type INTEGER, STRING, STRING, and BOOLEAN, respectively. None of the DVA values can be NULL as they are all required fields.

We will now illustrate some sample OML statements and briefly describe them.

- Insert Ewald Quak as a faculty.

```
INSERT Faculty (  person-ss := 653725952,
                  first-name := "Ewald",
                  last-name := "Quak",
                  us-citizen := true,
                  faculty-id := 2,
                  faculty-dept :=  Department WITH (dept-abbreviation = "ACC"),
                  classes-teaching  :=  INCLUDE LIMIT = ALL Section WITH ( unique-no  = 48855
                                                                            OR  unique-no =  48255
                                                                            OR unique-no = 48215))
```

In the above INSERT statement, a new *Faculty* entity is created. Since *Faculty* is a subclass of *Person*, an entity in the *Person* class is automatically created. *person-ss*, *first-name*, *last-name* and *us-citizen* are inherited DVA attributes from the *Person* class. *faculty-id* is an immediate DVA attribute of the *Faculty* class, while *faculty-dept* and *classes-teaching* are immediate EVA attributes of the *Faculty* class. *faculty-dept* is a single valued EVA and has been assigned a reference to the “ACC” department, whereas *classes-teaching* is a multi-valued attribute that refers to three entities in the *Section* class.

- Laurence Lebihan is made an instructor-researcher from a student.

```
INSERT Instructor_Researcher FROM Student
           WHERE first-name = "Laurence" AND last-name = "Lebihan"
                   ( faculty-id := 106,
                    advisor  :=  Faculty WITH ( faculty-id = 7 ),
                    faculty-dept :=  Department WITH (dept-abbreviation = "CS"))
```

In this statement, an entity is inserted into *Instructor_Researcher* from the *Student* class. Assuming the “Laurence Lebihan” entity was a member of the *Student* class, entities will be created automatically for the *Instructor_Researcher* superclasses, *Faculty* and *Grad_Student*. It is to be noted that the where clause should not hold more than one *Student* entity. Thus, there should be only one student named Laurence Lebihan for this INSERT statement to execute.

- Professor Mata Rebecca in the math department will not be teaching any courses in fall 2003.

```
DELETE LIMIT = 3 Section WHERE ( year =  2003
                                 AND semester = "fall"
                                 AND dept-abbreviation OF dept-at OF course-of = "M"
                                 AND first-name OF instructor = "Rebecca"
                                 AND last-name OF instructor = "Mata")
```

This DELETE statement is used to delete sections to be taught by the professor named Rebecca Mata during fall 2003. *dept-abbreviation OF dept-at OF course-of* is of special interest as it navigates thorough two EVAs to check if the course is offered in the math department. The LIMIT clause ensures that if more than three sections satisfy the condition, the delete operation is terminated.

- Students with student-id 922014 and 134880 have changed their major to CS and dropped all their registered classes for fall 2003.

```
MODIFY LIMIT = 2 Student  ( majoring-in := Department WITH ( dept-abbreviation = "CS"),
                           classes-registered := EXCLUDE LIMIT = ALL Section WITH (year =  2003
                                                                                  AND semester = "fall"))
                  WHERE student-id = 922014 OR student-id =134880
```

- Print the unique course number, course name and department name of all the classes offered by the business department during the fall 2003 semester.

```
FROM Section RETRIEVE unique-no,
                       course-name OF course-of,
                       dept-name OF dept-at OF course-of
             WHERE year = 2003
                    AND semester = "fall"
                    AND college-name OF college-in OF dept-at OF course-of = "Business"
```

- Display details of people and their spouses.

```
FROM Person RETRIEVE *,
                      * OF spouse
             WHERE person-ss OF spouse <> NULL
```

In this query, the “*” construct is used to display all the DVA attributes of a person and his or her spouse. Both the perspective class and the target class of the spouse EVA is *Person*. The relational operator <> is used against NULL to check if the person’s spouse is entered in the database.

- Output faculty, college pairs where faculty does not belong to the college

```
FROM Faculty, College RETRIEVE first-name,
                                          last-name,
                                          college-name
       WHERE college-name OF college-in OF faculty-dept <> college-name
```

In the above FROM statement, a join operation is performed between entities of the Faculty and College classes. Although join operations are not usually needed in SIM, they are necessary when two or more otherwise unrelated entities are involved in the query.

7. Comparing SIM with SQL

In this section, we discuss how SIM has addressed some of the weaknesses of relational data model languages like SQL with the help of comparative examples. Queries are performed in SIM using the logical path of the problems at hand, thus facilitating the ease of query use. The majority of the SIM queries do not require join operation, which makes it more efficient than SQL. The time required for join operation increases exponentially as the number of tables involved increases.

- Print the name of all the departments and their respective students in the Engineering College.

In SIM,

```
FROM Department
RETRIEVE dept-name, first-name OF dept-students, last-name OF dept-students
WHERE college-name OF college-in = "Engineering"
```

In SQL,

```
SELECT dept-name, first-name, last-name
FROM Department, College, Student
 WHERE  College.CollegeNo = Department.CollegeNo
           AND Department.DepartmentNo = Student.DepartmentNo
           AND College.Name EQ 'Engineering'
```

- If a professor in the Natural Sciences College is an advisor to at least one grad student who is a US citizen, print the professor's name and classes he or she teaches.

In SIM,

```
FROM Faculty
RETRIEVE first-name, last-name, course-name OF course-of OF classes-teaching
            WHERE college-name OF college-in OF faculty-dept = "Natural Sciences"
                    AND us-citizen OF SOME(students-advising) = true
```

In SQL,

```
SELECT first-name, last-name, course-name
FROM College, Department, Faculty, Section, Course
WHERE College.CollegeNo = Department.CollegeNo
         AND Department.DepartmentNo. = Faculty.DepartmentNo
         AND Course.SectionNo = Section.SectionNo
         AND Faculty.FacultyID = Section.FacultyID
         AND  college-name = 'Natural Sciences'
         AND faculty-id IN (SELECT faculty-id
                                   FROM Faculty, Grad_Student
                                   WHERE Faculty.FacultyID = Grad_Student.AdvisorFacultyID
                                            AND Grad_Student.us-citizen = true)
```

The above examples illustrate how the SIM queries are more streamlined than SQL queries. The multi-table join operation performed between the tables in SQL is not intuitive and is not directly related to the problem we are trying to solve. SIM does not have to perform the join operation, as it can just navigate through EVA references.

8. Implementation Considerations and Software Tools Used

We now describe the primary implementation considerations for SIM and elaborate on the various software tools used for the implementation. Our implementation for SIM is Java-based, making it a platform independent implementation. Java's object-oriented model and relatively fast development environment also made it the preferred language for implementation. One of the drawbacks of the Java language is that it runs slower compared to languages like C++. As the goal of this research project was not to implement SIM for PC at a commercial level, efficiency was not a major issue. The current version of SIM does not implement all the features of the SIM database management system due to limitations such as the amount of development that could be undertaken by a team of one in a semester. As we covered in sections 5 and 6, we have implemented the fundamental features of SIM that builds on the semantic data model.

The lexical analysis of our implementation of SIM is carried out by a tool called Java Compiler Compiler (JavaCC). JavaCC is a parser generator for Java applications. JavaCC takes the source program and generates JavaCC-defined tokens using the programmer-provided grammar. These tokens are used as input to a syntax analyzer tool called JJTree. JJTree constructs a tree from the tokens using the provided language grammar. The next stage is interpretation of the tree and performing actions or printing error messages based on its contents.

We have used a Berkeley Database (Berkeley DB) to store the database information on the disk using the B+tree feature. Berkeley DB is an embedded database system that can be used for concurrent storage and retrieval of data in key/value pairs. Although Berkeley DB is written in C for efficiency reasons, it provides a Java API. Berkeley DB effectively satisfies ACIDity which is a core part of any database system. That is, it satisfies the properties of atomicity, consistency, isolation, and durability by providing a collection of services like crash recovery, checkpoints, two-phase locking, and multiple concurrency.

9. Future Work on SIM

SIM's streamlined approach to database development has the potential to serve as a major platform for building large-scale database applications. We believe that using this current PC version of SIM, we will be able to demonstrate lots of advantages and uses of SIM and its SDM model. If these advantages are properly availed of in research works and academia, the current SIM version would provide motivation for its expansion to include all of its intended features.

10. Conclusions

In this paper, we have presented an overview of SIM, how it relates to SDM, and how SIM encompasses the fundamental features of SDM including entities, relationships between entities, integrity constraints and abstraction. We have shown how SIM uses class hierarchy, EVA and DVA constraints and verify declarations to capture the semantics of the application. We have gone over how SIM facilitates retrieval operations on a database from different perspective classes. Railroad diagrams have been provided for easy syntax reference for the current version of SIM. The use of SIM has been illustrated using the UNIVERSITY database and we have also demonstrated the ease of use of SIM over SQL.

We believe SIM can be very useful for bioinformatics as it encompasses an enormous amount of data storage and manipulation requirements, complex relationships and constraints between entities. We are looking into using SIM for bioinformatics and proving its advantages as the next step of our research project. SIM will also be used in academia, as it will be incorporated by some of the professors in their database classes including the ones at the University of Texas at Austin.

11. Acknowledgments

I would like to express my utmost gratitude to Philip Cannata for guiding me throughout my research work. This project would not have been possible without his active involvement. I am also very grateful to Greg Lavender for his comments on the write-up and Mohamed Gouda, the departmental honors program supervisor for facilitating this research project. I thank Doug Tolbert who provided me with documentation on SIM. I also thank Unisys Corporation for giving me permission to implement SIM and to Sun Microsystems for allowing me to use its facilities.

12. References

[1] Michael Hammer, Dennis McLeod: Database Description with SDM: A Semantic Database Model. TODS 6(3): 351-386(1981)

[2] D. Jagannathan, R. L. Guck, B. L. Fritchman, J. P. Thompson, D. M. Tolbert: SIM: A Database System Based on the Semantic Data Model. SIGMOD Conference 1988: 46-55

[3] C.J. Data. An Introduction to Database Systems, Volume 2. Addison-Wesley 83.

[4] R. King, D. McLeod. Semantic Database Models. In S.B. Yao (Ed). Principles of Database Design. Prentice Hall 84.

[5] MCP/AS InfoExec™ Semantic Information Manager (SIM) Object Definition Language (ODL) Programming Guide (8600 0189–101)

[6] MCP/AS InfoExec™ Semantic Information Manager (SIM) Object Manipulation Language (OML) Programming Guide (8600 0163–102)

Appendix A

Following is a SIM schema for a simple *University* database:

```
% University Schema
%%%%%%%%%%%%%%%%%%%%%%%%%%%% Person %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CLASS Person
(
   person-ss     "social security number "  : INTEGER, REQUIRED, UNIQUE;
   first-name    "first name"               : STRING[20], REQUIRED;
   last-name     "last name"                : STRING[20], REQUIRED;
   us-citizen    "U.S. citizenship status"  : BOOLEAN, REQUIRED;

   spouse        "Person's spouse if married"    : Person, SV, INVERSE IS spouse;
);

%%%%%%%%%%%%%%%%%%%%%%%%%%%% Student %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SUBCLASS Student OF Person
(
   student-id          "student identification"   : INTEGER, REQUIRED, UNIQUE;
   hours-completed  "hours completed"          : INTEGER;

   classes-registered  "classes registered for"      : Section, MV, INVERSE IS students-enrolled;
   majoring-in          "student's major"             : Department, SV, INVERSE IS dept-students;
);

%%%%%%%%%%%%%%%%%%%%%%%%%%%% Faculty %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SUBCLASS Faculty OF Person
(
   faculty-id     "unique employee identification"  : INTEGER, REQUIRED, UNIQUE;
   salary          "current yearly salary"             : INTEGER;
   office-phone  "office phone number"               : INTEGER;

   faculty-dept         "department under which faculty teaches"  :  Department, SV, INVERSE IS dept-faculty;
   classes-teaching    "classes currently taught"                 :  Section, MV, INVERSE IS  instructor;
   students-advising  "students advised by the faculty"          :  Grad_Student ,MV, INVERSE IS advisor;
);

%%%%%%%%%%%%%%%%%%%%%%%%%%%% Grad_Student %%%%%%%%%%%%%%%%%%%%%%%%%%

SUBCLASS Grad_Student OF Student
(
   fellowship-amount "fellowship amount, if any"     : INTEGER;

   advisor "faculty advisor"          : Faculty, SV, INVERSE IS students-advising;
);

%%%%%%%%%%%%%%%%%%%%%%%%%%%% Alumni %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SUBCLASS Alumni OF Person

%%%%%%%%%%%%%%%%%%%%%%%%%% Instructor_Researcher %%%%%%%%%%%%%%%%%%%%%%%

SUBCLASS Instructor_Researcher "Current employees of the company" OF Grad_Student AND Faculty
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Section %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CLASS Section
(
   unique-no    "unique number"                       : INTEGER, REQUIRED, UNIQUE;
   year         "year, format: 2003"                  : INTEGER, REQUIRED;
   semester     "semester: spring, summer or fall"    : STRING[10], REQUIRED;

   course-of           "course of this section"              : Course, SV,  INVERSE IS sections-of;
   students-enrolled  "students enrolled in this section"    : Student, MV, INVERSE IS classes-registered;
   instructor          "instructor of this section"          : Faculty, SV, INVERSE IS classes-teaching;
);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Course %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CLASS Course "Courses offered in the University"
(
   course-name      "course name"         : STRING [40], REQUIRED, UNIQUE;
   course-number  "course number"        : INTEGER, REQUIRED;
   course-suffix    "course suffix"         : STRING [1];

   sections-of     "sections of this course"          : Section, MV, INVERSE IS course-of;
   dept-at          "course offered by department"   : Department, SV, INVERSE IS courses-offered;
);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% College %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CLASS College "Colleges in the University"
(
    college-name    "college name"     : STRING [20], REQUIRED, UNIQUE;

    college-departments "departments under this college" : Department, MV, INVERSE IS college-in;
);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Department %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CLASS Department "Departments in the University"
(
        dept-name              "Department name"            : STRING [20], REQUIRED, UNIQUE;
        dept-abbreviation    "Department abbreviation"   : STRING [3], REQUIRED, UNIQUE;
        dept-phone             "Department phone"            : STRING [10], REQUIRED;

        courses-offered  "courses offered by this department"        : Course, MV, INVERSE IS dept-at;
        dept-students     "students enrolled under this department"  : Student, MV, INVERSE IS majoring-in;
        dept-faculty       "faculties under this department"            : Faculty, MV, INVERSE IS faculty-dept;
        college-in          "department under college"                  : College, SV, INVERSE IS college-departments;
);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% schema ends %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```