OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

SIDUR - An Integrated Data Model

Michael J. Freiling
Computer Science Department
Oregon State University

SIDUR --- AN INTEGRATED DATA MODEL

Michael J. Freiling

Computer Science Department
Oregon State University

ABSTRACT

The expanding complexity of database application worlds, and the accelerating pace of change of these worlds mandate data models which support some of the tasks of data interpretation which formerly were borne by the applications programs. Chief among these are the support for virtual data, definition of transactions, and enforcement of semantic constraints. In the past, data models have typically been divided into a data definition component and a data manipulation component. The functions which semantic data models attempt to support however, require a closer integration between these two components. In this paper, we show how one semantic data model, SIDUR, incorporates data manipulation descriptions into a data definition framework by means of a formal notation called sigma expressions. We also show how this framework forms the basis for defining higher semantic level manipulation operators, which incorporate query and transaction capabilities.

I INTRODUCTION

New technologies must be developed to aid database schema and applications design. Reasons for this need are the shortage of trained programmers and designers, the expansion of database technology into medium and small scale businesses, the rapid pace of change in business conditions and practices. Current practice usually separates the schema design and applications programming tasks, even to the point of having two different languages, the data definition language (DDL) and the data manipulation language (DML).

Typical of the tasks performed by application programs are supporting views of data not specifically defined in the database schema, defining transactions which change stored data, and enforcing constraints which preserve the data integrity. Examples of these tasks from a university application world are:

virtual data -- the association between a teacher and a student is defined if the teacher teaches a course and the student is enrolled in the same course.
transaction -- enrolling a student in a course involves adding that student to the students already taking the course.
constraint -- No student may enroll in a course unless he has taken all prerequisites for the course and there is room in the course.

Most application tasks involve the retrieval, and manipulation of stored data. Data models which support these tasks must incorporate such functions into the data model itself. The historical distinction of data definition and data manipulation is no longer adequate . Even with high level DML's such as relational calculus [4], the integration of data manipulation with data definition is not tight enough. Manipulations and updates can be defined, to be sure, but the transactions are left in a disembodied state -- it is not clear whether the defined transactions are sufficient, or redundant, or whether they are even motivated by the natural requirements of the data. A closer coupling is needed. In this paper we show how semantic data models can be developed to achieve a better integration of DDL and DML capabilities. We do this using a semantic data model, SIDUR, which incorporates manipulative information into the data model using a declarative notation called sigma expressions. These manipulative components are then combined to form high-level semanticaly motivated operations. While using examples from SIDUR, we intend to show that this method provides a general paradigm for constructing a wide class of semantic models.

II A DATA DEFINITION FRAMEWORK

Every data model starts with a set of basic constructs whose descriptions define the structure of the entire database. In data level and access level models [7], these are typically pictured as specific data structures. For example in the relational model[4], there is one such construct, the relation. In the DBTG model[6] there are two, the record type and the owner-coupled set. Semantic level models on the other hand, typically have constructs closer in nature to the application world than to the data storage structures. These go by different names but usually include objects (entities) of interest and their descriptive properties (attributes), connections between these objects (relationships, associations), and important behavior (events, actions, transactions).

SIDUR comprises five basic constructs

data value classes --- important classes of numbers and strings
object classes --- entities meaningful to the application
situations --- associations between objects, properties of objects
computations --- complex operations on numerical values
actions --- behavior in the world which affects the database

Each of the SIDUR constructs is defined by a set of *slots* which specify the form of the construct and its connections to other constructs. These slots can be divided into two classes: *descriptive slots*, which determine the inherent properties and constraints of a construct and *interpretive slots*, which describe the connections between constructs of the same or different types. The descriptive slots of SIDUR are the following

data value classes:
form --- syntactic structure of member data values
size --- the storage required for each member data value
minval --- minimum range value for a numeric class
maxval --- maximum range value for a numeric class
precision --- number of significant digits for a real number class

object classes:

representative --- name of a particular data value class whose elements can serve to "stand in place for" the objects

superclasses --- names of object classes which the class under definition is a specialization of

names --- publically available names for objects of this type

situations:

participants --- the objects whose participation defines the situation, and the role they play

cardinalities --- maximum occurrences of participants in situations

extension --- whether the situation is expected to obey the open-world or closed-world assumption [11]

computations:

participants --- objects and values which serve as inputs or outputs for the computation

actions:

participants --- objects involved in the defined behavior

The use of these descriptive slots, as well as the interpretive slots to be described later, is illustrated by a sample schema at the end of this paper. Explanations of constructs will be made by reference to this sample schema.

Data Value Classes

There are only five data value classes which must be defined for the sample schema at the end of this paper. The reason for this is that classes need only be defined for publically available data. A special generic class, TOKEN, provides a pool of data values for all other uses. The "form:" slot for string classes such as COURSE-NAME-V or PERSONAL-NAME-V permits a regular expression style of definition. Notice for instance that course names are composed of a two letter department code, a dash, and a three digit number. The "maxval:", "minval:", and "precision:" (number of significant digits) serve likewise to determine the range of numeric classes, such as GPA-V.

Object Classes

SIDUR enforces a strict distinction between data values and objects. Data values are purely syntactic --- they acquire meaning only when designated as *representatives* [8] of objects, i.e. to have a one-to-one correspondence with the objects. Each object in a SIDUR database must have a representative. However, not all objects need be represented by public data values. Objects of class PERSON, STUDENT, and COURSE, for instance, are represented by a special type of data value, the TOKEN. Tokens, also called *surrogates* [5,8], are unique, non-public data values which make it possible to separate the representation of an object from any of its properties, including its name.

When members of an object class are represented by tokens, the database must contain information linking the objects with their publically available names. The "names:" slot provides this -- it contains the names of two-way associations connecting the tokens and public names. The "names:" slot for the object class PERSON, for instance, lists the situation HAS-NAME as providing this connection.

When object classes are *generalizations* [12] of others, the "superclass:" slot enables specialized classes to inherit representative and name information from its superclass. The object classes STUDENT and INSTRUCTOR are examples.

Situations

While data value classes and object classes are important semantic concepts, they cannot alone define data storage structures --- meaningful structures can only come from associating more than one object. The situation provides a semantic structure which can be mapped onto physical storage.

Several situations are defined in our sample application. Some especially interesting situations are TEACHES-COURSE, GRADE-FOR, and MAY-TAKE. As we shall see in section III, not all situations define data storage structures. Some of them turn out to define derived structures.

The "participants:" slot is the most important of the descriptive slots for situations. The filler for this slot is a sequence of triples of the form

< role name> / < variable> / < object class> .

Each participant in any *instance* of a situations must play a particular role. In general, the roles are chosen from the fixed set *agent, object, value, source, destination, time,* and *location*. The variable exists for internal identification of the participant. Finally, the object class name provides a domain to which possible participants must belong.

Each instance of a situation provides a connection between the representatives of the objects which participate. The representative of this connection itself is called a *binding tuple*, because it binds the participant names to actual data values. The binding tuple for one instances of the situation HAS-NAME, for example, would be

< (agent -> T-047) (value -> "JAMES MANGAN") >

where T-047 stands for the unique token representing the individual whose name is James Mangan. It is these binding tuples which are constructed and manipulated by the manipulation operators of subsequent sections.

The set of binding tuples which are valid for a situation at any point in time are termed the *extension* of the situation. For purposes of intuition it is easiest to think of the extension of a situation as a (real or virtual) relation. Figure 1 shows sample extensions for HAS-NAME, HAS-TITLE, and TAKES-COURSE.

Most extensions in current databases are assumed to obey the "closed world assumption" [11] that any instance which does not occur in the stored extension does not hold. SIDUR provides that certain extensions can be chosen to be "open world", that is, instances which do not hold must be explicitly declared not to hold. CAN-TEACH, for example, is such a situation.

Cardinality restrictions form an important part of the

HAS-NAME:

agent	value
T-047	JAMES MANGAN
T-062	PAT PEARSE
T-133	SEAN CAROLAN

HAS-TITLE:

agent	value
T-455	CS-101
T-368	CS-102
T-219	CS-211

TAKES-COURSE:

agent	object
T-047	T-455
T-047	T-368
T-062	T-455

Figure 1 --- Sample Extensions for Situations

semantic constraints on data. SIDUR provides a mechanism which limits unique combinations of participants. The cardinality restriction on GRADE-FOR provides an example. Any single combination of a student and course is limited to one occurence among all the student/course/time triples which make up the extension of GRADE-FOR.

Computations

In order to keep the number of different concepts to a minimum, computations have the same conceptual structure as situations, although its interpretation is different. The lone computation in our sample schema is GPA-OF, which produces a GPA value for each student.

Notice that the "participants:" slot remains the same, except that the roles are different. SIDUR uses a separate set of role names for computations. Particularly important is the *result* role, which signifies the value produced by the computation. All other participants are considered to be arguments or parameters of the computation.

Actions

Actions share the same conceptual structure as situations and computations --- an association of participants. The only descriptive slot for actions is again the "participants:" slot, using the same roles as situations.

The relevant actions defined in our sample schema are ENROLLS-IN, when a student enrolls in a course, and COMPLETES, which occurs when the student completes a course. The primary difference between situations and actions is their relationship to time. Situations may hold or not, but actions *occur* once, after which their effects will hold. The action construct provides SIDUR with a structure around which to build transaction definitions.

How are transactions actually defined? How are inferred situations given a definition? Performing these tasks requires the incorporation of data manipulation capabilities in our descriptive framework.

III INCORPORATING DATA MANIPULATION

To incorporate data manipulation capabilities into a model, a set of manipulation operators must be defined. There are three ways in which this can be done:

procedural --- data manipulations are written as programs in a standard programming language [10].

algebraic --- a specific set of data manipulation operators is defined. Expressions are constructed by nesting operators [2,4,5].

declarative --- expressions describing desired data in terms similar to predicate logic are assigned a manipulative interpretation [4].

SIDUR uses a declarative notation to which several different manipulative interpretations can be assigned. Instances of this notation are referred to as *sigma expressions*, because they are a situational version of the well-known lambda expressions [3,9].

Construction of Sigma Expressions

The starting point for building sigma expressions is the *atomic sigma expression*, which has the form

(C1 (R1: P1) (R2: P2) ...)

where C1 may be a situation or computation name, the Ri are role names, and the Pi may be constants or variables. For example,

(HAS-NAME (agent: x) (value: "JAMES MANGAN"))

is an atomic sigma expression.

Open sigma expressions are built from atomic sigma expressions using the connectives

(and S1 ... Sk) (not S1)
(or S1 ... Sk) (empty S1)

For instance

(and (TAKES-COURSE (agent: x)(object: y))
(HAS-TITLE (agent: y) (value: "CS-211"))
(HAS-NAME (agent: x) (value: "JAMES MANGAN"))

is an open sigma expression referring to the set of individuals who

take CS-211 and have the name James Mangan.

Finally, a *closed sigma expression* is built from an open expression via the form

(sigma (V1 V2 ... Vk) S1)

where S1 is an open sigma expression and the Vi are variables which may or may not appear in the expression. An example closed sigma expression is

```
(sigma (w z)
  (and (HAS-NAME (agent: x) (value: w))
       (HAS-TITLE (agent: y) (value: z))
       (TAKES-COURSE (agent: x) (object: y))))
```

The closed sigma expression delineates one or more variables as being the primary focus of the expression.

The First Manipulative Interpretation

Sigma expressions themselves form a set of purely syntactic structures. To incorporate them usefully into a semantic data model, some "semantics" must be assigned to the expressions. These semantics take the form of a manipulative interpretation -- an interpretation of the syntactic object in terms of the usual activities involved in data manipulation. There are actually three different manipulative interpretations which can be assigned in SIDUR, which go by the names *enquire, *assert, and *deny, corresponding to query, addition of information, and removal of information respectively. We shall cover each of these interpretations in turn.

The first manipulative interpretation, *enquire, is purely query based, and assumes no change to the stored data. This interpretation retrieves for each sigma expression its associated extension, the set of binding tuples which "match" it. The rules become complex in their totality, but in general are simple:

--- The extension of an atomic sigma expression is the exension of its underlying situation.

--- The extension of two sigma expressions joined by "and" corresponds to the intersection (and sometimes the equijoin) of the extensions of the two sigma expressions.

--- The extension of two sigma expressions joined by "or" corresponds to the union of the extensions of the component sigma expressions.

--- The extension of an atomic sigma expression enclosed in "not" corresponds to the negative extension if the indicated situation if open world. Otherwise "not" is interpreted to mean set subtraction, and can only be used where this interpretation makes sense.

--- The extension of a sigma expression enclosed in "empty" is interpreted as being only a Boolean (true or false) value.

--- The extension of a closed sigma expression corresponds to the projection of the extension of the enclosed open sigma expression onto the variables of interest.

To show just one example, the extension resulting from the *enquire interpretation of the following expression

```
(sigma (w z)
  (and (HAS-NAME (agent: x) (value: w))
       (HAS-TITLE (agent: y) (value: z))
       (TAKES-COURSE (agent: x) (object: y))))
```

relative to the sample extensions of figure 1 is shown in figure 2.

w	z
JAMES MANGAN	CS-101
JAMES MANGAN	CS-102
PAT PEARSE	CS-101

Figure 2 --- Sample Extension from *enquire

The Second And Third Interpretations

Data manipulation operations concerned with retrieval are by themselves insufficient to fully define all the operations needed by a semantic data model --- interpretations which permit changes to the database must also be included. In SIDUR there are two, called *assert and *deny. As mentioned, the extension which *enquire associates with a sigma expression can sometimes be empty. The purpose of *assert is to insure that this exension of the sigma expression argument *is not empty*, while that of *deny is to insure that the extension *is empty*. For atomic sigma expressions this intent is quite clear. Consider the following operation.

```
*assert
 [ (HAS-NAME (agent: "T-047") (value: "JAMES MANGAN")) ]
```

Since the sigma expression here contains no variables, there is a single binding tuple, namely

< (agent -> "T-047") (value -> "JAMES MANGAN") >

whose presence can matter in determining whether the sigma expression has a full or empty expression. So the effect of the *assert operation would be to add this binding tuple to the extension for HAS-NAME, (provided of course that no cardinality constraints are violated). Similarly, this one tuple would be removed on interpretation of the same sigma expression via *deny.

With a sigma expression that contains variables, such as

(sigma (x) (TAKES-COURSE (agent: "T-047")(object: x)))

the expression can match several possible binding tuples. Interpretation under *deny would result in the removal of all of them. Thus, starting from the extensions of figure 1, interpretation of

*deny [(TAKES-COURSE (agent: "T-047") (object: x))]

would result in the extension pictured in figure 3.

agent	object
T-062	T-455

Figure 3 --- Result Extension of TAKES-COURSE

Figure 4 shows the result of then interpreting

*assert [(TAKES-COURSE (agent: "T-047")(object: x))]

on the extension of figure 3. In order to create a new non-empty extension, some value for the unspecified "object" participant must be invented. As a result, the token T-992 is created to fill this role. This type of interpretation can only be performed when the indicated participant has a representative of type token.

agent	object
T-047	T-992
T-062	T-455

Figure 4 --- TAKES-COURSE after *assert

In some cases, these interpretations of sigma expressions produce ambiguity. Consider the following operation.

```
*deny [ (and (HAS-NAME (agent: x) (value: "PAT PEARSE"))
             (HAS-TITLE (agent: y) (value: "CS-101"))
             (TAKES-COURSE (agent: x) (object: y)))) ]
```

The goal of making the extension for this entire expression empty could be achieved by removing a single instance from TAKES-COURSE, from HAS-NAME, or from HAS-TITLE. Though it is often possible to infer what choices are intended from the context, a data manipulation language alone cannot be expected to have such capabilities. SIDUR's answer is to invoke an extraneous arbitration function called CHOICE which is assumed to be capable of resolving these ambiguities. The CHOICE function could, for example, return to the user for more information, or infer the appropriate choice from context, or carry out other complex computations. The only requirement is that CHOICE completely resolve ambiguities before updates are performed on the data.

Integrating The Sigma Expressions

In different contexts, then, the sigma expressions can be assigned different manipulative interpretations, and used to hold manipulative definitions in a declarative framework. The exact means for integrating such definitions into the schema is the use of a set of "interpretive" slots in the schema itself. The value of each of these interpretive slots is a sigma expression, and serves to link SIDUR construct definitions.

The most useful of these interpretive slots is the "definition:" slot. Earlier we alluded to the fact that some situations are actually stored, while others are inferred from stored situations. Situations which are actually stored have a "definition:" slot marked PRIMITIVE, while inferred situations fill the slot with a sigma expression. Examples from the sample schema include the situations TEACHES-STUDENT and FILLED.

The interpretation assigned to this sigma expression depends on the mode in which the definition is accessed. If a query is in progress, *enquire mode is used. If an assertion or denial of this information is attempted, that same mode is transferred to the sigma expression. Thus an expression of the form

```
*assert
  [(TEACHES-STUDENT
       (agent: "T-129") (object: "T-047")) ]
```

is translated into one like

```
*assert
  [(and (TEACHES-COURSE (agent: "T-129") (object: x))
        (TAKES-COURSE (agent: "T-047") (object: x)))]
```

involving updates to the two primitive situations.

Two other interpretive slots for situation definitions are "necessary:" and "required:". These slots contain prerequisite information and consistency criteria which must hold before a situation can be asserted.

Object class definitions also contain a "definition:" slot. The value is a situation name, and is used to link the object class definition with a situation that defines the members of the class. This is appropriate only for object classes with token representatives, since other classes are assumed to include all representatives as representing valid members. Examples from the sample schema include PERSON, COURSE, STUDENT, and INSTRUCTOR.

Computations have a "definition:" slot as well. Primitive computations are those which are implemented via special programs. Those with sigma expression definitions utilize other, simpler computations. The "definition:" slot for the computation GPA-OF in our sample schema shows how the grade point average can be defined based on a primitive computation AVERAGE, and a situation GRADE-VALUE which maps the letter values of grades to their numeric values.

Instead of a definition slot, actions have two interpretive slots which are labeled "prerequisites:" and "results:". Upon request for performance of an action, the "prerequisites:" sigma expression is handled via the *enquire interpretation. If this succeeds (i.e. produces a non-empty extension) the "results:" sigma expression is handled via *assert mode. The use of this pair of slots is illustrated in the action ENROLLS-IN which appears in

the sample schema.

The modes in which these interpretive links are handled depends on the nature of the semantic level operatior being used. The development of such a set of operators, of course, is intimately coupled with the interpretive slots which must exist to support each operator. In the next section some high level operators will be defined, based on the slots presented here.

IV BUILDING SEMANTIC MANIPULATION OPERATORS

To provide an appropriate semantic data manipulation language, manipulation must be supported on the semantic constructs themselves. Such operations are usually easy to give names to, for instance "create an object", or "retrieve a situation", or "perform an action", but often turn out to be prohibitively difficult to define clearly. Designers of semantic data models have usually chosen to ignore the problem altogether, or to provide for the inclusion of arbitrary procedures to accomplish this task [1,10], a powerful approach, yet one with serious practical difficulties.

In this section we provide examples of a few of the SIDUR semantic level manipulation constructs, and show how they are defined by using the interpretive slots we have already seen. These operators are a small subset of the actual SIDUR operators, and are meant primarily to illustrate the technique. SIDUR can be easily extended to meet other needs with additional slots and operators.

The most fundamental of SIDUR's semantic level operators is termed ENQUIRE, and is developed as a straightforward extrapolation of the *enquire mode of sigma expression interpretation mentioned earlier. ENQUIRE takes a sigma expression as argument, and returns its extenstion. Expressed in an algorithmic style, ENQUIRE acts as shown.

ENQUIRE (S) :

[1] Check that S is legal, i.e. that all constants are of the appropriate object class.
[2] If S is atomic and the situation is primitive, perform a database retrieval to get the extension.
[3] If S is atomic but not primitive, perform ENQUIRE recursively on the definition of S, substituting constants where appropriate.
[4] If S is not atomic, perform ENQUIRE on the components of S, and merge the resulting extensions as required.

For instance, when faced with the expression

```
ENQUIRE
[ (sigma (x)
    (and (TEACHES-STUDENT (agent: x)(object: y))
         (HAS-NAME (agent: y) (value: "JAMES MANGAN"))) ]
```

which translates as "Who teaches James Mangan?", ENQUIRE first expands the expression to

```
ENQUIRE
[ (sigma (x)
    (and (TEACHES-COURSE (agent: x) (object: z))
         (TAKES-COURSE (agent: y) (object: z)))
         (HAS-NAME (agent: y) (value: "JAMES MANGAN"))) ]
```

The resulting sigma expression is processed by performing retrievals for the extenstions of TEACHES-COURSE and TAKES-COURSE, and merging these extensions so as to effect an equijoin on the participant labelled "y".

After ENQUIRE is defined, a more specialized version called CHECK, which returns boolean values is defined. CHECK returns the empty extension if ENQUIRE does, otherwise CHECK return the full extension, which acts as Boolean "true".

CHECK can be used in the definition of other operations. A weak version of the *assert interpretation of sigma expressions, called REFLECT, will update a situation as long as its necessary and required conditions hold. Simplistically, this can be defined as shown.

REFLECT (S) :

[1] Check to see that S is valid.
[2] If S is atomic, perform CHECK on the expressions in the "necessary:" and "required:" slots.
[3] If the checks succeed, and S describes a primitive situation, perform a database update to reflect the new information.
[4] If S is not primitive, recursively perform REFLECT on the sigma expression defining S.
[5] If S is (AND S1 ... Sk) , perform a REFLECT recursively on each of S1 through Sk
[6] If S is (OR S1 ... Sk) , call the CHOICE operation to choose one sub-expression, and perform REFLECT on this sub-expression.

A stronger form of ASSERT is also defined, which checks only the "necessary:" slot, and tries to recursively ASSERT missing conditions from the "required:" slot. This provides two levels of update, one of which has the power to make exceptions if necessary to achieve the goal.

Other operations can be built up as well, such as the PERFORM operation, which simulates the occurrence of an action"

PERFORM (A) :

[1] perform CHECK on the "prerequisites:" slot of A.
[2] If successful, perform REFLECT on the "results:" slot of A.

Progressively more complex operations, which include defaults, multiple levels of protection, etc. can also be built up in this fashion.

V CONCLUSIONS

What we have demonstrated in this paper is a method for building data models which capture data semantics and support high level, semantically motivated operations, without resorting to arbitrary procedural inclusion. The essential steps of this method are to:

--- establish a descriptive framework for the data model which defines the basic constructs and their form.
--- produce a declarative language for representing interpretive links between semantically defined constructs.
--- assign the necessary query and update semantics to these expressions.
--- incorporate a set of interpretive links into the basic descriptive data model framework.
--- define semantic level manipulations as combinations of varying interpretations of the links.

While this scheme sacrifices the general power of procedural inclusion it has two compensating advantages. First, it is more tractable than arbitrary procedures for automatic schema design. Declarative expressions are much simpler to generate than arbitrary programs. Second, the links defined by such a model provide a theory of the necessary and important relationships between schema constructs, which can be exploited in the schema design process. Transactions are not defined in isolation from the schema, but have a semantic motivation. Missing information in a partially completed schema is also much easier to discover.

The use of SIDUR to illustrate this method does not imply that SIDUR is the only model which could be generated in this fashion. Many variants are possible. What is important is that the method provides a tractible substitute for semantic modelling techniques which rely on inclusion of arbitrary procedures. Much work remains to be done in providing simple declarative forms which are at the same time amenable to a manipulative interpretation.

SAMPLE SIDUR SCHEMA

data values

```
data-value-class COURSE-NAME-V
        type: STRING
        size: 6
        form: ({"A"-"Z"}]2 "-" [1-5] ([0-9]]2

data-value-class PERSONAL-NAME-V
        type: STRING
        size: 14
        form: ({"A"-"Z"}]< 5 "-" ({"A"-"Z"}]< 8

data-value-class GPA-V
        type: REAL
        minval: 0.0
        maxval: 4.0
        precision: 2

data-value-class COURSE-LIMIT-V
        type: INTEGER
        minval: 10
        maxval: 100

data-value-class GRADE-V
        type: STRING
        size: 1
        form: ["A","B","C","D","F"]
```

object classes

```
object-class PERSON
        representative: TOKEN
        definition: IS-PERSON

object-class NAME
        representative: PERSONAL-NAME-V

object-class COURSE-NAME
        representative: COURSE-NAME-V

object-class GPA
        representative: GPA-V

object-class GRADE
        representative: GRADE-V

object-class COURSE-LIMIT
        representative: COURSE-LIMIT-V

object-class COURSE
        representative: TOKEN
        definition: IS-COURSE

object-class STUDENT
        superclass: PERSON
        definition: IS-STUDENT

object-class INSTRUCTOR
        superclass: PERSON
        definition: IS-INSTRUCTOR
```

situations defining object classes

```
situation IS-COURSE
        participants: agent/x/COURSE
        definition: PRIMITIVE
        extension: CLOSED-WORLD

situation IS-PERSON
        participants: agent/x/PERSON
        definition: PRIMITIVE

situation IS-STUDENT
        participants: agent/x/STUDENT
        definition: (and (IS-PERSON (agent: x))
                         (TAKES-COURSE (agent: x) (object: y)))

situation IS-INSTRUCTOR
        participants: agent/x/PERSON
        definition: (TEACHES-COURSE (agent: x))
```

other situations

```
situation HAS-NAME
        participants: agent/x/PERSON , value/y/NAME
        cardinalities: 1 < x>
        definition: PRIMITIVE
        extension: CLOSED-WORLD

situation HAS-TITLE
        participants: agent/x/COURSE , object/y/COURSE-NAME
        cardinalities: 1 < x>
        definition: PRIMITIVE
        extension: CLOSED-WORLD

situation CAN-TEACH
        participants: agent/x/INSTRUCTOR , object/y/COURSE
        definition: PRIMITIVE
        extension: OPEN-WORLD
```

```
situation TEACHES-COURSE
        participants: agent/x/INSTRUCTOR , object/y/COURSE
        cardinalities: 1 < y>
        necessary: (CAN-TEACH (agent: x) (object: z)))
        definition: PRIMITIVE
        extension: CLOSED-WORLD

situation TAKES-COURSE
        participants: agent/x/STUDENT , object/y/COURSE
        definition: PRIMITIVE
        extension: CLOSED-WORLD

situation TEACHES-STUDENT
        participants: agent/x/INSTRUCTOR , object/y/STUDENT
        definition: (and (TEACHES-COURSE (agent: x) (object: z))
                         (TAKES-COURSE (agent: y) (object: z)))

situation GRADE-FOR
        participants:
          agent/x/STUDENT , object/y/COURSE , value/z/GRADE
        cardinalities: 1 < x y>
        definition: PRIMITIVE
        extension: CLOSED-WORLD

situation GRADE-VALUES
        participants: agent/x/GRADE , object/y/INTEGER
        cardinalities: 1 < x>
        definition: PRIMITIVE

situation PREREQUISITE-FOR
        participants: agent/x/COURSE , object/y/COURSE
        definition: PRIMITIVE
        extension: CLOSED-WORLD

situation MAY-TAKE
        participants: agent/x/STUDENT , object/y/COURSE
        necessary:
          (empty
            (or (and (PREREQUISITE-FOR (agent: z) (object: y))
                     (FLUNKED (agent: x) (object: z)))
                (and (PREREQUISITE-FOR (agent: z) (object: y))
                     (empty (COURSE-GRADE (agent: x) (object: z))))))
        required:
            (empty (FILLED (agent: y)))
        definition:
          PRIMITIVE


situation LIMIT
        participants: agent/x/COURSE , value/y/COURSE-LIMIT
        cardinalities: 1 < x>
        definition: PRIMITIVE
        extension: CLOSED-WORLD

situation FILLED
        participants: agent/x/COURSE
        definition:
         (GREATER-THAN-OR-EQUAL-TO
           (agent:
             (COUNT
               (domain:
                     (sigma (y) (TAKES-COURSE (agent: y) (object: x))))
           (object:
             (value-of (LIMIT (agent: x)))))
```

computations

```
computation GPA-OF
        participants: agent/x/STUDENT , result/y/GPA
        definition:
          (AVERAGE-OF
                (domain:
                  (sigma (z)
                   (GRADE-VALUES
                         (agent:
                          (sigma (w)
                           (COURSE-GRADE (agent: x) (value: w))))
                         (value: z)))))
```

actions

```
action ENROLLS-IN
        participants: agent/x/STUDENT , object/y/COURSE
        prerequisites:
            (MAY-TAKE (agent: x) (object: z)))
        results: (TAKES-COURSE (agent: x) (object: y))

action COMPLETES
        participants: agent/x/STUDENT , object/y/COURSE , value/z/GRADE
        prerequisites: (TAKES-COURSE (agent: x) (object: y))
        results: (and (not (TAKES-COURSE (agent: x) (object: y))
                      (GRADE-FOR (agent: x) (object: z)))
```

REFERENCES

[1] Abrial, J.R.; "Data Semantics"; in *Data Base Management;* J.W. Klimbie and K.L. Koffeman, ed.; North Holland; Amsterdam; 1974

[2] Buneman, P., Frankel, R., and Nikhil, R.; "An Implementation Technique for Database Query Languages"; in *A.C.M. Transactions on Database Systems;* Volume 7, No. 2; June 1982

[3] Church, Alonzo; "The Calculi of Lambda Conversion"; in .ul Annals of Mathematical Studies; Volume 6; Princeton University Press, Princeton, NJ; (reprinted by Klaus Reprints, New York, 1965)

[4] Codd, E.F. ; "A Relational Model of Data for Large Shared Data Banks" ; in *Communications of the ACM* ; Volume 6 ; June 1970

[5] Codd, E.F.; "Extending the Database Relational Model to Capture More Meaning"; in *ACM Transactions on Database Systems* Volume 4, No. 4; December, 1979

[6] CODASYL Data Base Task Group Report ; April 1971 ; available from: ACM ; New York

[7] Freiling, M.J.; *Understanding Database Management;* Alfred Publishing Co.; Sherman Oaks, CA; 1982

[8] Kent, W.; *Data and Reality* North Holland; Amsterdam; 1978

[9] McCarthy, J.; "Recursive Functions of Symbolic Expressions and their Computation by Machine"; in *Communications of the A.C.M.* Volume 3, No. 4; April 1960

[10] Mylopoulos, J., Bernstein, P., and Wong, H.; "A Language Facility for Designing Database-Intensive Applications"; in *ACM Transactions on Database Systems* Volume 5, No. 2; June 1980

[11] Reiter, R.; "On Closed World Data Bases"; in *Logic and Databases;* H. Gaillaire and J. Minker, eds.; Plenum Press, New York, 1978

[12] Smith, J.M. & Smith, D.C.P.; "Database Abstractions: Aggregation and Generalization"; in *ACM Transactions on Database Systems* Volume 2, No. 2; June 1977