PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

SIDUR - A STRUCTURING FORMALISM FOR KNOWLEDGE INFORMATION PROCESSING SYSTEMS

Dan D. Kogan [1]
Michael J. Freiling [2]

Oregon State University

ABSTRACT

A new class of information applications is emerging (Fuchi 82; Ohsuga 82) for which neither Artificial Intelligence (AI) nor Database technologies alone are well suited. The work presented in this paper describes a formalism called SIDUR, which integrates a manipulation mechanism with its representation components using a declarative notation known as the sigma expression. Declarative components can then be combined to form high-level, semantically motivated schema designs which include the specification of virtual data, the definition of transactions, maintenance of semantic integrity constraints as well as the ability to define high level data manipulation operations. Such an information model with precisely limited (in this case non-combinatorial) inferential capabilities forms an appropriate level of interface between the more general deductive powers of an AI component and the back-end data storage and manipulation mechanism.

1.0 INTRODUCTION AND MOTIVATION

Database and Artificial Intelligence (AI) technologies represent the extremes of a continuum on which the solutions to information intensive problems fall. Database applications are usually well understood and can be realized through algorithmic methods. They require the maintenance of large collections of facts which may change over time, but have a somewhat regular structure. The overriding concerns include data independence, integrity and consistency of the stored information, and efficiency of manipulation (Date 81).

Artificial Intelligence, on the other hand, is usually applied to problems which are not fully understood and require the use of heuristic inference techniques (Barr and Feigenbaum 81). The information these problems require may not have a regular structure and is often highly interconnected. Furthermore, the individual pieces of information are often much fewer than in database domains, numbering only in the thousands, rather than in the millions. The important considerations for AI formalisms are representational richness and ease of manipulation by the inference mechanism.

Recently, it has become apparent that there are many applications that require combined features from Database and Artificial Intelligence technologies. These applications are characterized by a need to maintain large databases while also requiring the inference capabilities of Artificial Intelligence systems. Systems which address these applications have been called 'Knowledge Information Processing' (Fuchi 82; Ohsuga 82; Suwa et al. 82), 'Knowledge Management' (Kellogg 82) and 'Knowledge Base' (Wiederhold 84) systems.

Applications of this type arise in two ways. The first is from database applications which "outgrow" existing database technologies requiring the incorporation of inferential capabilities. The second comes from the growing complexity of Artificial Intelligence applications and the need to provide a "Knowledge Base" with limited inferential capabilities but independent of the primary inference engine (Brachman and Levesque 84).

Such systems must be capable of performing some deduction not only during retrieval operations, but during updates as well (Ohsuga 82; Wiederhold 84). Furthermore, they must present a uniform conceptual structure which can lend itself to manipulation by general purpose reasoning mechanisms (such as those based on first order logic).

The characteristics of these applications require inference mechanisms which are not search-based, but capable of representing and performing simple inferences on their own, as well as interfacing to more general and powerful deductive systems. Such mechanisms must meet the functional requirements of these new applications while at the same time reconciling two sets of conflicting demands: *representational adequacy* or naturalness and *computational effectiveness*. Representational adequacy refers to the ease with which important aspects of the application can be expressed within the constructs of the model. Computational effectiveness includes the pragmatics of an application, such as avoiding combinatorial searches to minimize response times.

This paper describes a formalism for structuring information based on a user oriented model of information. This formalism, called SIDUR, integrates a manipulation mechanism with its representation components using a declarative notation known as the sigma expression. These components can then be combined to form high-level, semantically motivated schema designs. In particular, they enable the specification of transactions, constraints and virtual information in a declarative and natural way.

2.0 SIDUR FRAMEWORK

SIDUR's main strength lies in its ability to provide a framework for defining natural and semantically motivated query operations. For example, a typical data manipulation

Author's current addresses:
1 System Development Corporation, Santa Monica, Ca., 90406.
2 Tektronix Inc., Beaverton, Or., 97077.

operation in SIDUR would be

```
PERFORM [(TransferEmployee
        (agent "John Brown")
        (source "System Design")
        (destination "Formal Verification"))]
```

to transfer an employee from one project to another.

The active nature of such primary tasks as the definition of transactions arises from **interpretation functions** which are applied to declarative expressions in the database schema. Selection of the appropriate expression is achieved by access of slots in an object-based schema framework. The major objects in this framework are known as: **Data Value Classes, Object Classes, Situations, Computations** and **Actions.**

A complete treatment of the syntax and semantics of the model is given in (Kogan 84). This paper outlines the structure of SIDUR and its manipulation language. Examples used are taken from the schema of a decision support system for project managers (Kogan 84).

2.1. Sigma Expressions

SIDUR's declarative notation is called the **sigma expression.** Its syntax is similar to that of logic-based languages such as the predicate calculus or their database variant, the relational calculus (Ullman 81).

An **atomic sigma expression** has the form

$$(C_1 (R_1 P_1) \ldots (R_n P_n))$$

where C_1 is the name of a situation or computation, the R_i are role names and the P_i may be constants or variables. Figure 1 shows a atomic sigma expression.

```
(HasEmployeeSkills (agent E)(object S))
```

Figure 1 - Sample atomic sigma expression.

Open sigma expressions are built from atomic sigma expressions using the connectives **and, or, not** and **empty.** The open sigma expression in Figure 2 specifies that the skills required by a project P are the same as those associated with employee E.

```
(AND (HasEmployeeSkills (agent E) (object S))
     (HasSkillRequirements (agent W) (object S)))
```

Figure 2 - Sample open sigma expression.

Open sigma expressions are equivalent syntactically to expressions in first order logic, though their interpretation is different in some respects.

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

$$(sigma\ (V_1 \ldots V_k)\ S_1)$$

where S_1 is an open sigma expression and the V_i are variables which may or may not appear in the expression. Figure 3 shows a closed sigma expression denoting projects whose expected completion dates are not met.

```
(sigma (P)
  (AND
    (ExpectedCompletionDate (agent P) (time D1))
    (CurrentDate (agent D2))
    (LESSTHAN (agent D1) (object D2))

)
```

Figure 3 - Sample closed sigma expression.

With respect to the current database state, a sigma expression is taken as denoting an abstract data structure called its **extension.** An extension is a set of unique **binding tuples** each of which assigns values to variables of the expression. Figure 4 shows a possible extension for the sigma expression of figure 1.

```
{<(E -> "John Brown")(S -> "Database Systems")>
 <(E -> "Jack Smith")(S -> "Expert Systems")>}
```

Figure 4 - Sample extension.

This is different from the first order logic expressions, which are usually taken as denoting predicates rather than extensions.

2.2. Interpretations of Sigma Expressions

In order to utilize the syntactically declarative sigma expression to support inference and update, an operational interpretation must be assigned to these structures. Most attempts to add inferential capabilities to an information model (Brachman 83; Konolige 81; Kellogg 78), usually stop with a single semantic interpretation of intensional expressions. This single interpretation relates intensional expressions as to their use in query, or data retrieval, but not to their use in updates.

The SIDUR approach is to explicitly define several interpretation functions relative to the model in which an expression is used. In this fashion a reliable update semantics can also be assigned to these expressions by adding update semantics to the more standard **query** interpretation function. Accordingly, we assign three distinct interpretation functions to sigma expressions, known as ***enquire, *assert** and ***deny.** These interpretations correspond respectively to inquiry, addition and removal of information.

2.2.1. *enquire

The *enquire interpretation function returns the **extension** which a sigma expression denotes with respect to the current database state. It is analogous to DADM's QUERYANALYSIS function or the Prolog interpreter viewed as an interpretation function for Horn clauses (Clocksin and Mellish 81) The rules for determining which binding tuples belong to the extension of a sigma expression are outlined below.

- The extension of an atomic sigma expression is the extension of its underlying situation.
- The extension of two sigma expressions joined by **and** corresponds to the cartesian product of their extensions if they do not have common variables and the equijoin of their extensions if they do have variables in common.
- The extensions of two sigma expressions joined by **or** corresponds to the union of the extensions of their component sigma expressions.

- The extension of an atomic sigma expression enclosed in **not** corresponds to the negative extension if the indicated situation is interpreted under the open world assumption. Otherwise not is interpreted as set subtraction and can only be used under certain conditions.
- The extension of a sigma expression enclosed in **empty** is a Boolean (true or false) value, depending on whether its sigma expression has an empty extension or not.
- The extension of a closed sigma expression corresponds to the relational projection of the extension of the enclosed open sigma expression onto the sigma variables.

2.2.2. *assert and *deny

Retrieval operations are only part of the complete set of capabilities an information manipulation mechanism must possess. The purpose of the *assert interpretation is to perform such actions as are necessary to ensure that the extension underlying a sigma expression is not empty. The *deny interpretation, on the other hand, acts to ensure that the underlying extension is empty.

If an atomic sigma expression contains no variables, then its interpretation is straightforward. Under *assert a binding tuple corresponding to the constants in the expression is added to the extension of the situation in the sigma expression, whereas under the *deny interpretation it is removed. This is done only if the appropriate constraints are not violated. For example, the extension of

```
*assert [(HasName (agent TOK001) (value 'John Brown'))]
```

adds the binding tuple

```
<(E -> TOK001) (N -> 'John Doe')>
```

to the extension of HasName. That same expression under the *deny interpretation would effect the removal of the binding tuple. The rules for determining what actions are taken under the *assert and *deny interpretations are given in APPENDIX 1.

Atomic expressions containing variables can match several binding tuples. For example,

```
(EmployeeAssignment (agent X) (object TOK999))
```

represents all employees working for the project represented by the token TOK999. Under the *deny interpretation, **all** binding tuples matching the above expression are removed. Under the *assert interpretation, binding tuples are added with new tokens created to fill the slots represented by variables (in a similar fashion to the way GENSYM creates new atoms in Lisp). However, new tokens are created only for slots whose fillers are token values; other slots must be filled with actual values. Although SIDUR does not provide null values, the **action** construct enables them to be specified in a way meaningful to the application (section 3.3 and Figure 12).

Sometimes the interpretation of sigma expressions produces ambiguity. This is the case with disjunction under *assert and conjunction under *deny. Consider, for example, the expression in Figure 5 which can be satisfied by non-deterministically removing appropriate instances of either conjunct.

```
*deny [(AND
      (HasEmployeeSkills (agent E) (object S))
      (HasSkillRequirements (agent W) (object S)))
    ]
```

Figure 5 - *deny in a conjoined sigma expression.

Though it may be possible under certain conditions to infer which choice to make from context, it is not reasonable to encode these decisions into the data model itself. A general solution to this problem would require generation of combinatorial search requests, resulting in unacceptable performance costs. SIDUR's solution is to invoke a function extraneous to the model called CHOICE which is assumed to be able to resolve these ambiguities. A particular implementation of CHOICE depends on the demands of the application (Roth 84). At its simplest, CHOICE returns to the user for more advice.

2.3. Semantic Data Manipulation

Data manipulation in SIDUR is not performed through the primitive semantic interpretations *enquire, *assert and *deny. Rather it is performed via a more developed set of semantic manipulation operators which in turn are defined in terms of the primitive semantic interpretations.

The following is a partial catalog of these operations. ENQUIRE determines the extension of its query extension. CHECK treats a query extension as a predicate and returns true or false. ASSERT establishes an extension. DENY removes an extension. PERFORM performs the indicated transaction. PERMIT? determines whether it is possible to perform a transaction. PERMIT! ensures that it is possible to perform a transaction. Section 3.6 shows examples of these operations in use and describes the mechanism which enables their definition.

3.0 SIDUR DETAILS

SIDUR provides five basic constructs: **Data Value Classes, Object Classes, Situations, Computations** and **Actions.** Each SIDUR construct is defined by a set of **slots** which specifies the form of the construct and its connection to other constructs. These slots can be divided into two classes: **descriptive** and **interpretative.** Descriptive slots describe the inherent properties and constraints of a construct, while interpretive slots describe the connections between constructs of the same or different types.

3.1. Situations

A **situation** defines associations among objects which represent meaningful information about the application. It unifies descriptive concepts commonly known as attributes and relationships without imposing arbitrary distinctions between them; this is a desirable feature since such distinctions are notoriously ambiguous (Kent 78) and subject to change depending on the user's view.

In a SIDUR implementation, the situation is the only structure which is actually mapped into physical storage. Each instance of a situation provides a connection between the representatives of the participants of that situation. An instance of that connection is called a **binding tuple** because it binds the participants to actual data values. The set of binding tuples which are valid for a situation at any point in time is called the **extension** of the situation.

The situation construct has three descriptive slots: **participants, cardinalities,** and **extension.** It also has three interpretive slots: **definition, necessary** and **required.**

The **participants** slot specifies those objects which participate in the situation as well as the roles they play via a sequence of triples of the form:

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

Roles provide a position-independent label for the participants; they do not imply any semantic properties of the filler other than those explicitly stated in the definition. The variable is used for identification of the participant within the situation description itself and the object class provides a domain from which the participant must be drawn.

The **cardinalities** slot represents a common form of integrity constraint (Roussopoulus 79; Brachman 79). It specifies the maximum number of instances in the current extension with common values for the individual participants. Figure 6 shows a partial definition of a situation specifying that a manager, can be in charge of at most 3 projects.

```
(situation: IsTechnicalManager
  (participants:
     agent/E/Employee object/W/WorkOrder)
  (cardinalities: 1 <E>, 3 <P>)
)
```

Figure 6 - Situation with explicit constraints.

The **extension** slot specifies whether the extension of a situation conforms to a closed or open world assumption (Reiter 78).

In addition to the descriptive slots described above, situations have three manipulative slots called **definition, necessary** and **required.** The definition slot either specifies that a situation can be instantiated directly via database lookup, ie: the extension of the situation is stored directly in the database, or it provides a formula (via a sigma expression) for deducing the extension. The necessary and required slots contain expressions representing consistency criteria which must hold before a situation can be asserted.

The simplest filler for the definition slot is the atom PRIMITIVE. It stipulates that the extension of a situation is stored directly in the database. In this respect, it acts much like the "support indicators" associated with each predicate in DADM (Kellogg 77).

However, not all situations need be explicitly stored. An advantage of higher order information structuring formalisms is their ability to specify inferred data. A sigma expression filling the definition slot indicates how the extension of the situation can be deduced. Figure 7 shows a non-primitive situation defining a qualified employee to be one having those skills required by a particular work order. The extension of this situation is the extension of the expression filling the definition slot. As explained earlier, this extension can be computed via an equijoin over S on the extensions of HasSkils and HasSkillRequirements.

```
(situation: IsQualifiedFor
  (participants:
     agent/E/Employee object/W/WorkOrder)
  (definition:
   (AND
    (HasEmployeeSkills (agent E) (object S))
    (HasSkillRequirements (agent W) (object S)))))
```

Figure 7 - Sample non-primitive situation

The necessary and required slots are filled by sigma expressions and represent two types of constraints (Sergot 82). The term "necessary" can be viewed in the sense of logically necessary; the expression filling it must always hold. The required slot, on the other hand, refers to conditions which in general must hold, but can admit to exceptions, for example, administrative policy which can be violated when good reason exists.

Operationally, the differences between these two types of constraints are realized in the way the two slots are interpreted. The sigma expression filling the necessary slot is checked before any update is performed by either the REFLECT or ASSERT operations (see Data Manipulation Interface) The sigma expression filling the required slot is only checked by the REFLECT operator, during an ASSERT operation; it may be overridden. For example, the situation defined in Figure 8 specifies that a project must have a funding source and a work order number.

```
(situation: IsProject
  (participants: agent/P/Project)
  (required: (HasProjectName (agent P) (value N)))
  (necessary:
   (AND
    (HasFundingSource (agent P) (value S))
    (HasProjectWorkOrder (agent P) (value W))))
  (definition: PRIMITIVE)
  (extension: CLOSED)
)
```

Figure 8 - The necessary slot.

3.2. Computations

Computations are special forms of situations which can be thought of as associations between several "argument" participants and a "result" participant such that unique combinations of non-result participants (called arguments) determine a unique result. However, since the potential set of arguments can be very large, it is clearly impossible to store the extension of a computation. The computation definition, therefore, provides a method by which the unique result participant can be determined.

Computations can be defined over individual instances of situations and used in sigma expressions, such as EARLIER-THAN in Figure 9 (SYSTEM computations are those implemented by a primitive algorithm). This use of computations is analogous to DADM's "compute relations" (Kellogg 77) or the use of "experts" in (Stonebraker 80).

```
(computation: EARLIER-THAN
  (participants: domain-1/X/DATE domain-2/Y/DATE)
  (definition: SYSTEM)
)

(situation: LateProject
  (participants: agent/P/Project)
  (definition:
   (sigma (P)
        (AND
         (ExpectedCompletionDate (agent P) (time D1))
         (CurrentDate (agent D2))
         (EARLIER-THAN (agent D1) (object D2)))))
)
```

Figure 9 - Computations over individual instances.

Computations can also be defined over whole extensions, permitting the specification of aggregate information, a common component of database applications. Figure 10 specifies a computation listing the number of employees assigned to work on each project.

```
(computation: COUNT-OF
 (participants: domain/X/EXTENSION-OF (S)
            measure/Y/ROLE-OF (S)
            result/Z/INTEGER)
 (definition: SYSTEM)
)

(computation: NumberOfEmployeesPerProject
 (participants: agent/P/Project value/N/INTEGER)
 (definition:
  (sigma (P N)
       (AND
        (EmployeeAssignment (agent E1) (object P))
        (COUNT-OF
         (domain:
          (sigma (E2)
               (EmployeeAssignment
                (agent E2) (object P))))
         (measure E2)
         (result N)))))
)
```

Figure 10 - Computations over whole extensions.

Notice that no additional notation is required to specify the partition on the extension of the EmployeeAssignment situation. Such additional constructs as "group by" found in languages like QUEL (Youssefi 77) and SQL (Chamberlin 76), are unnecessary because the attributes which induce a partition can be explicitly delineated by linking the appropriate sigma variables.

3.3. Actions

Actions describe events in the application domain which affect the underlying database. These constructs permit the declarative specifycation of transactions.

Actions are syntactically similar to situation definitions. However, rather than denoting an extension or a method for determining an extension, actions specify operations on the extension of some situations, or alternately, a set of update functions which map one set of current extensions into another. Figure 11 shows a simple action describing the transfer of an employee from one work order to another.

```
(action: TransferEmployee
 (participants: agent/E/Employee
            object1/W1/WorkOrder
            object2/W2/WorkOrder)
 (prerequisites:
     (AND
      (NOT (EMPTY
       (EmployeeAssignment (agent E) (object W1))))
      (EMPTY
       (EmployeeAssignment (agent E) (object W2)))
      (IsQualifiedFor (agent E) (object W2))))
 (results:
     (AND
      (EMPTY
       (EmployeeAssignment (agent E) (object W1)))
      (NOT (EMPTY
       (EmployeeAssignment
            (agent E) (object W2))))))
)
```

Figure 11 - Sample action definition.

The **participants** slot identifies the object classes which participate in the action as well as the roles they play, just as for situations.

The **prerequisites** slot permits database administrators to control the conditions under which certain actions can take place. This slot is filled by a sigma expression possibly having constants substituted for variables. In order for the action being defined to be carried out (by the PERFORM! operator), the expression in the prerequisites slot must have a non-empty extension.

The **results** slot describes the state of the database after the action has been carried out. Like the prerequisites slot, the value for the results slot is a sigma expression. When an action is PERFORMed, this slot is **REFLECT**ed causing the database to be appropriately updated so that a non-empty extension is created.

Actions also permit schema designers to specify methods for handling incomplete knowledge, frequently without having to resort to 'null' values. For example, Figure 12 shows how an employee's salary can be initialized to the lower bound of his/her salary grade.

```
(action: InitializeSalary
 (participants:
      agent/E/Employee
      value/S/Salary)
 (prerequisites:
  (AND
   (EMPTY (HasSalary (agent E) (value S)))
   (HasSalaryGrade (agent E) (object G))
   (HasSalaryLevels
    (agent G) (lowerBound S) (upperBound S1))))
 (results:
   (HasSalary (agent E) (value S)))
)
```

Figure 12 – Incomplete information and actions.

Currently, database mechanisms must typically resort to arbitrary procedural inclusion (Abrial 74; Britton-Lee 83; Mylopoulos 80), either via application programs or specially designed transaction facilities for performing updates while preserving integrity and consistency.

3.4. Data Value Classes

Data Value Classes specify displayable or publicly available data and the form these data may take. They are analogous to data type specifications in traditional programming languages.

The purpose of the data value class definitions is to allow the schema designer to assign names to recognizable classes of data type values, which may later serve as representatives for specific objects of interest to the application. Two aspects of these definitions permit initial levels of integrity constraints: the interpretation of the class and the precise formats to which values in the class must adhere.

A data value class specification can have up to six descriptive slots defined below. Figure 13 shows the definition of three data value classes from the sample schema.

The primary reason for the **type** slot is for checking the suitability of individual values as arguments to computations. The **size** and **form** slots are optional; respectively they permit the schema designer to allocate a maximum number of characters for string data and a regular expression format for defining acceptable members of the class. The **maxval** and **minval** slots specify the range of permissible values for numeric data. The **precision** slot guarantees that all operations on data of type real will be carried out to the specified number of digits.

```
(data-value-class: EmployeeID
  (type: INTEGER)
  (minval: 1)
  (maxval: 99999)
)

(data-value-class: PersonName
  (type: STRING)
  (size: 30)
  (form: ["A"-"Z"]["a"-"z"] < 15)
)

(data-value-class: Salary
  (type: REAL)
  (minval: 0.0)
  (maxval: 99999.00)
  (precision: 8.2)
)
```

Figure 13 - Data Value Class definitions.

In addition to the displayable data values, represented by strings, integers and real numbers, SIDUR also supports a special internal data value class called a TOKEN. Values from this class are unique, similar to Lisp GENSYMs (Xero83) and their use will be made clear in the next section.

3.5. Object Classes

Philosophers, logicians and more recently computer scientists have recognized the problems which can result from the failure to distinguish between an object and its representation (Quine 40; Kent 78). Therefore, **object classes** indicate the major components of an application and are roughly equivalent to the specification of the domains underlying a universe of discourse in logic. It is important to note that data value classes are a purely syntactic construct; they acquire meaning only when they serve as a **name** or **representative** for objects.

Three descriptive slots are associated with object classes: **representative, superclass,** and **names.** Figure 14 shows examples of object class definitions.

```
(object-class: Employee
  (representative: TOKEN)
  (names: (PersonName EmployeeId))
  (definition: IsEmployee)
)

(object-class: Manager
  (representative: TOKEN)
  (superclass: Employee)
  (definition: IsManager)
)

(object-class: Project
  (representative: TOKEN)
  (names: (ProjectName WorkOrderNumber))
  (superclass: WorkOrder)
  (definition: IsProject)
)
```

Figure 14 - Object Class definitions.

Each object must have a **representative** which is the name of a data value class. Notice that although each object must have a representative drawn from some data value class, not all elements from a given class will necessarily be representatives for some object. Furthermore, the representatives of important object classes will generally be TOKENS rather than displayable data. TOKENS, also known as **surrogates** (Kent 78), 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.

The **names** slot provides a way for objects represented by TOKENS to be externally referenced, for example by users. It contains the name of one or more associations connecting the tokens with the object's publicly available names. The **definition** slot specifies the valid members of an object class. It is similar in intent to the Be-Relations of (Borkin 79) and provides a means to enforce a degree of referential integrity (Date 81). The **superclass** slot induces a type hierarchy (Smith and Smith 77) where the specialized class can inherit such properties as representatives and names from its superclass.

3.6. Defining Manipulation Operations

Four operations on situations are shown: **ENQUIRE, CHECK, ASSERT** and **REFLECT.** ENQUIRE and ASSERT are expanded applications of the *enquire and *assert interpretations of sigma expressions. CHECK returns a boolean value depending on whether a situation holds or not. REFLECT will update a situation as long as both its necessary and required slots hold.

ENQUIRE accepts a sigma expression as argument and returns its extension. Figure 15 shows the semantics of ENQUIRE expressed in terms of the underlying *enquire interpretation.

[1] Check that all constants filling slots in the expression S are of the correct type, i.e.: they belong to the correct data value class.

[2] Apply *enquire interpretation to S

Figure 15 - ENQUIRE (S).

CHECK is a closed (Gallaire and Minker 78) form of ENQUIRE which returns a boolean value. CHECK returns the EMPTY extension if ENQUIRE does, otherwise it returns the FULL extension which acts as boolean 'true'.

REFLECT updates the extension of the sigma expression in its argument. It is a weak application of the *assert interpretation which only performs the updates if both the necessary and required slots do not return an EMPTY extension. Figure 16 shows the semantics of REFLECT in the same style used above.

[1] Check that all constants filling slots in the expression S are of the correct type, i.e.: they belong to the correct data value class.

[2] Perform CHECK on the expressions filling the **necessary** and **required** slots of S; if either returns EMPTY, REFLECT fails and must be backed out.

[3] Apply *assert interpretation to S, backing out of the operation if *assert fails at any step.

Figure 16 - REFLECT (S).

A stronger application of the *assert interpretation, called ASSERT, differs from REFLECT in that it performs the CHECK only on the necessary slot. It is shown in Figure 17.

[1] Check that all constants filling slots in the expression S are of the correct type, i.e.: they belong to the correct data value class.

[2] Perform CHECK on the filler of the **necessary** slot of S; If CHECK returns EMPTY, ASSERT fails and must be backed out.

[3] Perform REFLECT on the filler of the **required** slot of S.

[4] Perform REFLECT on S.

Figure 17 - ASSERT (S).

In addition to the four operations on sigma expressions defined above, several operations can also be defined which take actions as arguments. The simplest one is PERFORM which models the occurrence of an action, as shown in Figure 18.

[1] Check that all constants filling slots in A are of the correct type, i.e.: they belong to the correct data value class.

[2] Perform CHECK on the filler of the **prerequisite** slot of A. If CHECK returns EMPTY, the action can not be performed.

[3] If CHECK succeeds, perform REFLECT on the filler of the **results** slot of A.

Figure 18 - PERFORM (A).

More operations can be built in this fashion; for example, PERMIT? determines whether an action can be performed by CHECKing its prerequisites slot, while PERMIT! ensures that an action **can** be performed by ASSERTing the expression filling its prerequisites slot (this ensures that the database remains consistent). The definitions for these operations are given in appendix C.

As a complete example, we show how how an event, the transfer of employee "John Brown" from one project, "System Design" to another, "Formal Verification" is modeled by the following request:

```
PERFORM [(TransferEmployee
        (agent "John Brown")
        (source "System Design")
        (destination "Formal Verification"))]
```

The first step is to ensure that the constants filling each of the arguments belongs to data value classes representing the expected object class. This is equivalent to ensuring that the person being transferred (John Brown in this case) is a valid employee and that the two projects are similarly valid, thus providing an initial level of referential integrity. This is performed by consulting the extension for the situation filling the **definition** slot of the object class person.

The next step, ensuring that the prerequisites are met, invokes a CHECK operator with the arguments substituted:

```
CHECK [(CanSupport
      (agent "John Brown")
      (object "Formal Verification"))]
```

CHECK returns the EMPTY extension if the project can not support an additional employee, otherwise it returns the FULL extension, representing boolean 'true'. If CHECK returns EMPTY and the transaction fails, the user can issue

```
PERMIT [(TransferEmployee
        (agent "John Brown")
        (source "System Design")
        (destination "Formal Verification"))]
```

to generate

```
REFLECT [(CanSupport
        (agent "John Brown")
        (object "Formal Verification"))]
```

and force a state of affairs where the action can be carried out. Finally, enabling the transaction to be carried out consists of REFLECTing the sigma expression in the results slot of the TransferEmployee, with the appropriate values instantiated:

```
REFLECT
 [(AND
   (NOT (EmployeeAssignment
        (agent "John Brown") (object "System Design")))
   (EmployeeAssignment
    (agent "John Brown") (object "Formal Verification")))
 ]
```

4.0 RELATED EFFORTS

The integration of different representation and manipulation paradigms is not a novel approach. Other independent work can be roughly classified into three major lines of development. The first of these attempts to integrate the inferencing capabilities of logic based formalisms with the descriptive powers of other knowledge representation schemes. (Brachman 83) is representative of this work. The second group attempts to enhance the expressive power of data definition languages with first order logic in order to support deductive question answering, as exemplified in (Konolige 81). There is a third effort that bridges these two thrusts. The KM-1 architecture (Kellogg 82) uses a logic-based inference engine to support deductive question answering from relational databases, but also enhances its knowledge structuring capabilities by providing a semantic network-like concept graph (Kellogg 81).

The KRYPTON system (Brachman 83; Brachman 82) differentiates between terminological and assertional competence. Terminological competence refers to the ability to represent the specialized vocabulary used in application domains and to maintain the relationships between the various terms. This ability is best embodied by such knowledge representation mechanisms as KL-ONE (Brachman 79). Assertional competence, on the other hand, implies the ability to form a theory of the world knowledge required to solve problems in a particular domain and to reason with this theory. This type of competence is best achieved in a first order logic framework, because the inferences required to support it are much more complex. Brachman suggests that the two forms of competence can be integrated in a system where a KL-ONE style classifier (Lipkis 82) provides the terminological component, while a general theorem prover provides the assertional capability. Related works in this area include (Rich 82) and (Moore 82).

(Konolige 81) presents a method of formally representing the information contents of a relational database with the primary aim of supporting deductive question answering. This is done by taking the view that the database forms a model for a first order language based on the tuple relational calculus (Ullman 80). The application domain itself is a model of another first order language, called a metalanguage, based on the domain relational calculus. User queries concern the application domain itself, not merely the database, so they are posed in the metalanguage. The two languages are integrated by mappings which generate database requests when answering a query requires extensional information. Other related works are found in (Gallaire and Minker 78) and (Gallaire et al. 81).

The KM-1 architecture (Kellogg 82; Kellogg 84) supports the definition of virtual relations derived from explicitly stored data. It consists of an inference machine and a searching engine, each maintaining its own separate database. An Intensional Data Base contains first order logic statements (called premises), semantic advice rules and a type hierarchy used by the deductive component known as DADM (Kellogg 81) for developing plans for searching and computing over the extensional store. The Extensional Data Base can be any database although most KM-1 development has focused on relational databases and the current KM-1 configuration connects the deductive engine to a Britton-Lee IDM-600 relational database machine (Britton-Lee 83). The inferential component of the KM-1 architecture includes a concept graph to aid database administrators in maintaining potentially large numbers of virtual relations (derived predicates). Its maintenance is a cooperative task between the database administrator and the system itself.

Although these works bear some relationship to the SIDUR effort, they do not provide a **complete** database interface. The KM-1 application must resort to mechanisms provided by the underlying database management system (currently the IDM-600) in order to perform updates. The KRYPTON system aids the maintenance of complex descriptions for the development of expert systems (for example a computer systems configurator (Freeman et al. 83)), however it does not address the problem of managing large databases. Finally, while Konolige's work addresses issues important to deductive question answering, other elements of a database interface, namely updates, are not treated.

These works assume a static application domain where meaningful world events do not reflect changes in database states. In contrast, one of the basic constructs of the SIDUR formalism provides a structure around which database transactions may be built. The emphasis in SIDUR is to provide useful inference capabilities in all phases of Knowledge Management, including updates.

5.0 CONCLUSIONS

SIDUR, a structuring formalism for Knowledge Bases, permits the definition of virtual information, specification of transactions and the enforcement of constraints. It does so by integrating manipulation and representation components of the model via the declarative formalism of the sigma expression.

Because SIDUR does not resort to arbitrary procedural inclusion to define its manipulation operators, it becomes a more tractable vehicle around which applications for Knowledge Information Processing Systems can be built and a valuable complement to AI languages such as Prolog.

It should be stressed that SIDUR does not enhance the representational power of first order logic. Rather, it adds discipline to the use of an underlying inference mechanism for database intensive applications. This discipline is applied to structuring as well as maintaining the information.

The six SIDUR constructs suggest a discipline around which a set of rules of "good" schema design can be developed. Two such rules suggest that the representatives of important object classes should be TOKENs and that each situation should be used to represent only one meaningful association among its participants. Schemas designed in accordance to these rules seem to be amenable to evolutionary growth in accordance to the demands of the application in a manner similar to normalized relations (Kent 83). The design of the Manager's Assistant knowledge base was made more tractable by following these guidelines. It should be noted that these design rules can serve as the basis for automated design tools once they evolve to a mature state.

In addition to a structuring discipline, SIDUR also provides a manipulation discipline. This discipline is embodied by the *assert and *deny interpretations of sigma expression when used in conjunction with the required and necessary slots. These control pragmatics permit the specification of transaction definitions while ensuring the consistency of the information. An implementation of database transactions in Prolog (Kowalski 74), for instance would require the incorporation of a teleological semantics as well as control pragmatics similar to SIDUR's This would permit the specification of conditions under which clauses are evaluated when performing assert's and deny's to the database of ground clauses. These notions are approximated in the metalevel control suggested by (Bowen 82; Gallaire and Lasserre 82).

APPENDIX 1

The rules for determining the actions to be performed under the *assert and * deny interpretations are given below.

***deny**

- If S is an atomic sigma expression and conforms to the open world assumption, then apply *assert interpretation to negative extension of S.
- If S is an atomic sigma expression and conforms to the closed world assumption, then remove applicable instances from its extension.
- If S is an open sigma expression and its connective is AND - Invoke CHOICE to decide which conjunct will be applied the *deny interpretation.
- If S is an open sigma expression and its connective is OR - recursively apply the *deny interpretation to each of the constituent disjuncts.
- If S is an open sigma expression and its connective is NOT - apply the *assert interpretation to S.
- If S is an open sigma expression and its connective is EMPTY - apply the *assert interpretation to S.
- If S is a closed sigma expression, apply *deny to its underlying open sigma expression.

*assert

- If S is an atomic sigma expression and it is fully instantiated, update the extension of S to reflect the binding tuple.
- If S is atomic but only partially instantiated, If the representative of the uninstatiated participant(s) is(are) TOKEN(S), then fill the missing participant(s) with generated (GENSYM) token(s). If the missing participant(s) can not tokens, but elements of other data value classes, the *assert fails as those cannot be generated automatically.
- If S is an open sigma expression and the connective is AND - Then, ensure that the common variables are filled with same values, and assert each of the constituent conjuncts.
- If S is an open sigma expression and the connective is NOT - And the underlying sigma expression conforms to the OPEN WORLD assumption, Then apply *assert to the negative extension and *deny to the positive extension.
- If S is an open sigma expression and the connective is NOT - And the underlying sigma expression conforms to the CLOSED WORLD assumption, Then remove applicable instances from its extension.
- If S is an open sigma expression and the connective is EMPTY - Apply the same interpretation as for negation.
- If S is an open sigma expression and the connective is OR - Invoke CHOICE to decide which disjunct receives the *assert interpretation.
- IF S is a CLOSED sigma expression apply *assert to its underlying open sigma expression.

REFERENCES

Abrial, J. R.; Data Semantics; *Data Base Management;* J. W. Klimbie and K. L. Koffemann (eds.); North Holland Pub. Co.; Amsterdam, 1974.

Barr, A., and Feigenbaum, E.A.; *The Handbook of Artificial Intelligence.;* HeurisTech Press, Stanford, California; 1981.

Britton Lee Inc.; *IDM Software Reference Manual;* Version 1.4; January, 1983.

Borkin, S.A.; *Equivalence Properties of Semantic Data Models for Database Systems;* Laboratory for Computer Science, Massachussets Institute of Technology, Technical Report 206; Jan. 1979.

Bowen, K.A. and Kowalski, R.A.; Amalgamating Language and Metalanguage in Logic Programming; *Logic Programming;* Clark, K.L. and Tärnlund, S.-A. (eds.); Academic Press; 1982.

Brachman, R.J. and Levesque, H.J.; Tractability of Subsumption in Frame-Based Description Languages; Proc. AAAI-84; pp 34 - 41; 1984.

Brachman, R.J., Fikes, R.E. and Levesque, H.J.; KRYPTON: Integrating Terminology and Assertion; *Proc. AAAI—83;* pp 31 - 35; 1983.

Brachman, R.J. and Levesque, H.J.; Competence in Knowledge Representation; *Proc. AAAI—82;* pp 189 - 192; 1982.

Brachman, R.J.; An Introduction to KL-ONE *Research in Natural Language Understanding, Annual Report;* Report No. 4274, Bolt Beranek and Newman Inc.; Aug 1979.

Chamberlin, D.D, Astrahan, M.M., Eswaran, K.P., Griffiths, P.P, Lorie, R.A., Mehl, J.W., Reisner, P. and Wade, B.W.; SEQUEL 2: A unified Approach to Data Definition, Manipulation and Control; *IBM Journal of Research and Development;* Vol 20, No. 6, pp 560 - 575; Nov. 1976.

Clocksin W.F. and Mellish C.S; *Programming in Prolog;* Springer-Verlag, New York; 1981.

Date, C.J.; Referential Integrity; *Proc. VLDB;* 1981.

Date, C.J.; *An Introduction to Database Systems,* third edition; Addison Wesley; 1981.

Epstein, R.; *Techniques for Processing of Aggregates in Relational Database Systems;* Memorandum No. UCB/ERL M79/8; University of California, Berkeley; Feb 1979.

Fillmore, C. J.; The Case for Case; *Universals in Linguistic Theory;* Bach and Harms eds.; Holt, Reinhart and Winston Inc.; Chicago, Il.; 1968.

Freeman, M.W., Hirschman, L., McKay, D.P., Miller, F.L. and Sidhu, D.P.; Logic Programming Applied to Knowledge-Based Systems, Modelling and Simulation; *Artificial Intelligence Conference, Oakland University, Rochester, Mi.;* Apr. 1983.

Freiling, M.J., Carter, A., Ecklund, E., Hakanson, M., Kalvin, P., Kogan, D., Roth, S., Rood, A.; *The SIDUR 2.0 Reference Manual;* Computer Science Department, Oregon State University, Corvallis; 1983.

Fuchi, K.; Aiming for Knowledge Information Processing Systems; *International Conference of 5th. Generation Computer Systems;* Moto-Oka, T., ed.; North-Holland Pub. Co., Japan; 1982.

Furukawa, K., Nakajima, R., Shigeki, G., Aoyama, A.; Problem Solving and Inference Mechanisms; *International Conference of 5th. Generation Computer Systems;* Moto-Oka, T., ed. North-Holland Pub. Co., Japan; 1982.

Gallaire, H. and Lasserre, C.; Metalevel Control for Logic Programs; *Logic Programming;* Clark, K.L. and Tärnlund, S.-A. (eds.); Academic Press; 1982.

H. Gallaire, J. Minker and J. M. Nicolas (eds.); *Advances in Data Base Theory, Vol. 1;* Plenum Press, New York; 1981.

Gallaire, H. and Minker, J. (eds.); *Logic and Data Bases;* Plenum Press, New York; 1978.

Kellogg, C.; The Transition from Data Management to Knowledge Management; *Proc. IEEE Conference on Data Engineering;* Los Angeles; 1984.

Kellogg, C.; Knowledge Management: A Practical Amalgam of Knowledge and Data Base Technology; *Proc. AAAI—82;* 1982.

Kellogg, C. H. and Travis, L.; Reasoning with Data in a Deductively Augmented Data Management System; *Advances in Data Base Theory, Vol. 1;* H. Gallaire, J. Minker and J. M. Nicolas (eds.); Plenum Press, New York; 1981.

Kellogg, C.H. and Klahr, P.; Deductive Methods for Large Databases; *Proc. 5th. IJCAI;* Cambridge, Ma.; Aug 1977.

Kent, W., A Simple Guide to Five Normal Forms in Relational Database Theory, *Communications of the ACM,* Vol 26, No. 2, Feb 1983.

Kent, W.; *Data and Reality;* North-Holland Pub. Co., Amsterdam; 1978.

Kogan, D., The Manager's Assistant, an Application of Knowledge Management; *Proc. IEEE Conference on Data Engineering;* Los Angeles; 1984.

Kogan, D., Sidur - A Formalism for Structuring Knowledge Bases; M.S. Thesis, Dep. Computer Science, Oregon State University, Corvallis, Oregon; 1984.

Konolige, K.; A Metalanguage Representation of Relational Databases for Deductive Question Answering; *Proc. 7th. IJCAI;* pp 496 - 503; 1981.

Kowalski, R.A.; Predicate Logic as a Programming Language; Proc. IFIP-74 Congress; North-Holland, 1974.

Lipkis, T.; A KL-ONE Classifier; *Proc. 1981 KL—ONE Workshop;* Schmolze, J.G. and Brachman, R.J. (eds.); BBN Report No. 4842; 1982.

Moore, R.C., The Role of Logic in Knowledge Representation and Commonsense Reasoning, in *Proc. AAAI—82,* pp 428 - 433, 1982.

Mylopoulos, J., and Wong, H. K. T.; Some features of the TAXIS data model; *Proc. Sixth International Conference on Very Large Data Bases;* Montreal, pp 399 - 410; 1980.

Ohsuga, S., Knowledge Based Systems as a New Interactive Computer System of the New Generation, in *Japan Annual Review in Electronics, Computers and Telecommunications: Computer Science and Technologies,* Kitagawa, T. (ed.), North-Holland Pub. Co., Japan; 1982.

Quine, W.V. *Mathematical Logic;* Harvard University Press; 1940, revised 1981.

Reiter, R.; On Closed World Data Bases; *Logic and Data Bases;* Gallaire, H. and Minker, J. (eds.); Plenum Press, New York; 1978

Rich, C.; Knowledge Representation and Predicate Calculus: How to Have Your Cake and Eat It Too; *Proc. AAAI—82;* pp 193 - 196; 1982.

Roth, S.; *Implementing a Semantic Data Model;* Masters Thesis, Computer Science Department, Oregon State University; 1984.

Roussopoulos, N.; CSDL: A Conceptual Schema Definition Language for the Design of Data Base Applications; *IEEE Transactions on Software Engineering;* Vol SE - 5 No. 5; September 1970.

Sergot, M.; Prospects for Representing the Law; *Logic Programming;* Clark, K.L. and Tärnlund, S.-A. (eds.); Academic Press; 1982.

Smith, J.M, and Smith, D.C.P.; Database Abstractions: Aggregation and Generalization; *ACM Transactions on Database Systems;* Vol 2, No 2, pp 105 - 133; Jun 1977.

Stonebraker, M. and Keller, K.; Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System; *Proc. SIGMOD—80;* pp 58 - 66; 1980.

Suwa, M., Furukawa, K., Makinouchi, A., Mizoguchi, T., Yamasaki, H., Knowledge Base Mechanisms; *International Conference of 5th. Generation Computer Systems;* Moto-Oka, T., ed.; North-Holland Pub. Co., Japan; 1982.

Ullman, J.D.; *Principles of Database Systems;* Computer Science Press; 1980.

Wiederhold, G.; Knowledge and Database Management; *IEEE Software;* Vol 1, No. 1; Jan 1984.

Xerox Corporation; *Interlisp Reference Manual;* October, 1983.

Youssefi, K., Ubell, M., Ries, D., Hawthorn, P., Epstein, B., Berman, R. and Allman, E.; *INGRES Reference Manual — Version 6;* Memorandum No. ERL-M579; Engineering Research Laboratory, College of Engineering, University of California, Berkeley; 14-Apr-1977 (revised).