AN ABSTRACT OF THE THESIS OF

Shirley Roth for the degree of Master of Science in Computer Science presented on October 13, 1986

Title: IMPLEMENTING A SEMANTIC DATA MODEL

Abstract approved: Redacted for Privacy
Michael J. Freiling

OSIRIS is an integrated information architecture which was developed at Oregon State University. SIDUR is the data model upon which the semantic level is based. The semantic level is the mediating level between user's information needs and the stored data. The advantages of providing a semantic database environment include flexibility and independence in data access and schema changes and increased representational power.

The purpose of this paper is to describe the prototype implementation of SIDUR in Franz Lisp on a VAX®-11/750. This implementation maps semantic-level database operations into a descriptive formalism, called the BAGAL query language, based solely on present data which is amenable to optimization.

Implementing a Semantic Data Model

by

Shirley L. Roth

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed October 13, 1986

Commencement June 1987

APPROVED:

Redacted for Privacy

Assistant Professor of Computer Science in charge of major

Redacted for Privacy

Chairman of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented October 13, 1986

Typed by researcher for Shirley L. Roth

ACKNOWLEDGEMENT

This paper is presented as a product of the OSIRIS project research group under the direction of Professor Michael J. Freiling.

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

IMPLEMENTING A SEMANTIC DATA MODEL

I. IMPLEMENTING A SEMANTIC MODEL

1.1 Overview

This thesis is a description of an implementation of the SIDUR data model. SIDUR is the semantic level in the OSIRIS Integrated Information System, which is an information system architecture developed by Michael J. Freiling and his students in the Computer Science Department at Oregon State University in Corvallis, Oregon. The fundamental assumption of the OSIRIS project is: the data that will be stored will be sufficiently important, complex, and massive to warrant almost any amount of "consideration." "Consideration" in this sense means dedication of adequate computer resources to provide a user-friendly environment that is tolerant of change in data storage or schema description, that allows arbitrary interactive data access and update operations, and that allows non-technical users to build or revise schema.

1.2 Defining a Model

In order to provide a flexible, user-friendly system, the OSIRIS architecture is based on the following three design assumptions:

1) At least three levels of data representation and support are needed. The _semantic level_ is concerned with the interpretation of stored data in that it enforces constraints on allowable data for data update operations and derives virtual information from stored data for data retrieval operations. The _data level_ is

concerned with the nature, management, and manipulation of stored data in a manner that is independent of both high-level interpretation and low-level access to actual data structures and physical storage devices. The file level manages the actual data structures and the access of the data on physical devices.

2) Semantic-level queries and operations should be expressed without regard to data-level and file-level concerns, and vice versa. In order both to insure independence of the data level from matters of interpretation and to allow it adequate flexibility in query optimization, each semantic-level query must be fully translatable into a single procedure described by the data-level language.

3) Data-level query planning should be independent of the actual access algorithms but should transparently reflect both the semantics of the higher-level query as well as the processing cost considerations used in query optimization.

This paper addresses the implementation concerns of the second assumption described above. It is hypothesized that semantic-level queries can be expressed independently of lower-level concerns and that each semantic query can be totally mapped to a lower-level data access language procedure, which is independent of semantic concerns. This result is discussed theoretically in (Freiling, 1983a) and is demonstrated here by describing an actual implementation of SIDUR in Franz Lisp on a VAX-11/750 running Berkeley VAX/UNIX (4.1bsd revised 1 Sept. 1981.)

Since the purpose of a semantic-level model is to mediate between real-world information and stored data, the model must be defined in such a way that this "mediating" role is clearly specified by the functions that it performs. SIDUR has been defined to perform these functions:

1) enforce pre-conditions and constraints on allowable data-store states,

2) provide high-level query operators that can be used to attain the allowable states and to access the allowable states, and

3) allow flexible representation of information.

1.3 Defining a Data-level Target Language

The data-level target language called BAGAL (BAG ACCESS Language) has been designed to allow expression in it of any semantic-level query whether written in the SIDUR language or in other semantic-level languages that may be developed later. In addition, this general purpose data access language provides data independence as well as the opportunity to optimize query processing. BAGAL is partially declarative and partially procedural. Procedural portions of a query correspond to explicit ordering or conditional processing necessary to preserve the original query semantics. Those portions of the query that are not so constrained remain expressed in a declarative form in order to permit further processing to optimize control choices.

1.4 The Translation Process

The SIDUR translation process is based on re-expressing each SIDUR operation in terms of two primitive semantic operations, <u>*enquire</u> and <u>*assert</u>, and in terms of how the results of these primitive operations are to be related to each other. <u>*enquire</u> is the primitive data access operation. <u>*assert</u> is the primitive data update operation. These primitive semantic operations are translated to BAGAL access and update operations. Virtual semantic-level schema objects are re-expressed in terms of the lower-level schema objects, which are known to the data level. Semantic relationships are re-expressed as BAGAL set and computation operations, which specify associations between results of update and access operations. Update pre-conditions and constraints are re-expressed as BAGAL conditional and procedural operations. So, the semantic-level meaning of a SIDUR operation is expressed at the data level as a set of these partially ordered BAGAL operations acting on data-level schema objects. After the translation of a SIDUR operation is completed, the BAGAL access and update operations are interpreted at the data level to produce extensions of data values. The resulting extensions are merged together at the data level using the BAGAL set and computation operators to produce the final virtual extension that was specified by the relationship in the SIDUR operation.

1.5 Providing an Environment

The SIDUR semantic-level data model has been implemented in Franz Lisp on a VAX-11/750 running the Berkeley VAX/UNIX operating system (4.1bsd revised 1 Sept. 1981.) The implementation is currently a stand-alone set of Franz Lisp functions that provides a simplified user interface to prompt for SIDUR operations. Each SIDUR operation is translated in its entirety into a single data-level query. The SIDUR module has been implemented so as to be suitable for incorporation as the semantic level of the OSIRIS Integrated Information System.

The SIDUR implementation currently provides the user with only the capabilities to define data in semantically meaningful terms, to perform semantic-level operations, and to view the resulting data-level queries. These queries were somewhat simplified for pedagogical purposes and used as the data-level examples in this paper. Later, after the data-level interface is completed, the user will be presented with actual answers in response to SIDUR operations. It is expected that these capabilities will provide the mechanism for further research on which functions are needed by users at the semantic level and which are the most useful forms for providing these functions.

1.6 Literature Review

A number of data models and approaches to representing information have been proposed in the literature (Abrial, 1974; Chen, 1976; Kogan and Freiling, 1984; Smith and Smith, 1977a; Smith and Smith, 1977b; Hammer and McLeod, 1978; and Shipman, 1981.) These models have been based on a framework of levels of abstraction within an information system (ANSI/X3/SPARC, 1975; Senko, 1976; and Freiling, 1983a.) The development of the semantic level has focused on dealing with information, rather than data, at this level (Senko, 1973; Nijssen, 1976; Kent, 1978; Kogan, 1984; and Tsichritzis and Lochovski, 1982.)

Although the notion of semantic data modeling has been discussed by various authors, very little has been published on implementing data models. Many of the proposed models have not been implemented (Cattell, 1983.) Even much of the design work in mapping an information structure to an implementation data model has been very specific to one particular system being developed (Bracchi, 1982.)

Many of the proposed data models have considered data access concerns but have not made explicit provision for data manipulation. The TAXIS data model offers the ability to define transactions that can cause database state changes (Mylopoulos and Wong, 1980.) Incorporating a data manipulation language at the semantic level can be been done in three ways:

1) procedural--programs in a standard programming language (Mylopoulos et. al., 1980,)

2) algebraic--a defined set of data manipulation operators (Buneman and Frankel, 1979; Codd, 1979; and Cattell, 1983,) and

3) declarative--expressions similar to predicate logic are given a data manipulation interpretation (Freiling, 1983b; Freiling, 1982; Clocksin and Mellish, 1981; and Warren, 1980.)

Optimization, at both the semantic level and at the data level below it, are major concerns in implementing a data model. Various semantic-level optimization techniques such as indexing the schema and caching have been discussed (Cattell, 1983.) Optimization at the data level is crucial but must be obtained without the loss of independence between the semantic and access levels. One approach to achieving both adequate performance and data independence is to translate a semantic operation entirely into a single data-access-level language procedure (Freiling, 1983a.) The access level is then free to optimize the entire operation and yet be unconcerned with semantic issues.

II. THE SIDUR MODEL

The purpose of the SIDUR level in the OSIRIS architecture is to mediate between users' information needs and the stored data. SIDUR handles this interpretation of data by insuring that the stored data is in compliance with various kinds of schema-defined constraints on allowable database conditions and by providing a real-world-oriented mechanism for accessing and changing states. SIDUR is based on two components that enable it to derive and manipulate virtual data. First, there is a schema that describes the stored data in terms and associations related to actual user information needs. Second, there is a high-level data access and manipulation language.

The SIDUR level achieves its independence of lower-level concerns by allowing users to deal exclusively with high-level schema constructs rather than with constructs that resemble computer data structures. These high-level constructs are "close" to those into which users map their real-world conceptions of information. The SIDUR model provides the following types of information modeling constructs: situations, data-value classes, object classes, computations, and actions. The schema for a particular application will contain multiple occurrences of each of these construct types. The name and description for each occurrence of a schema construct is mapped from a real object, action, etc. in the application world.

The semantic-level access and update operations are described by a language of declarative statements composed of system operators, construct names, and sigma expressions, which describe relationships between data values. These operations are also "close" to the descriptions and procedures used in the real-world application to request information or to update information. SIDUR allows the user to manage the schema in the same manner as application information is handled and thus allows use of these same operations on the schema.

2.1 Information Modeling Constructs

The SIDUR model supports five basic types of information-modeling constructs:

situations

data-value classes

object classes

computations

actions.

Situations are the fundamental information storage constructs in that each defined situation is associated with an underlying real or virtual extension of stored data. Values cannot be stored into situation extensions unless certain pre-conditions defined in the schema are true. For closed-world situations, we assume that any missing values indicate that they do not belong in the situation. Open-world situations have two extensions. The affirmative or positive extension contains values known to belong in the situation extension. The negative extension contains values known not to belong in the situation extension. Values that are missing from both extensions are unknown with respect to this situation extension. Data-value classes and object classes are used to partially specify which data values may be associated with each other in a situation extension. Each object class has associated with it a data-value class to specify which values may legally belong to the object class. Each data-value class describes a set of data values, INTEGER, REAL, STRING, or TOKEN. A TOKEN is a system-generated value used to represent a real-world entity that must be uniquely and permanently identified in an application. Each data value participant in a situation must belong to the schema-prescribed object class for its corresponding role in the situation. A computation is an aggregate function whose value is determined by the values of its arguments, which are derived from situational extensions. Actions describe user-level update operations, which are defined in terms of their effects on situational extensions. Further information on these constructs may be found in the "SIDUR Manual" (Freiling, 1983c.)

Each occurrence of a SIDUR construct is described syntactically in the schema for an application by assigning it a unique name and by filling in values for the various slots specified for that type of construct. Each slot describes a specific aspect of the construct in relation to other constructs or in terms of system primitives. The semantic significance of a construct is determined by its type, by its use of system primitives, and by its specific connections with the schema constructs whose names appear in its various slots or who mention it in their slots. Thus, the schema can replace hard-coded application programs and user memory in fully describing these semantics. Figure 1 lists the slots for each of the SIDUR constructs.

As an example of how these slots are used, Figure 2 shows a schema definition for some TEACHES-STUDENT situation that defines a relationship between two objects, one of type PROFESSOR and one of type STUDENT. The definition: slot of this situation is a non-procedural description of a virtual extension that can be derived from the extensions of TEACHES-COURSE and TAKES-COURSE by finding pairs of instances from the latter two situations having in common the same object participant (labeled 'z' in the definition.) As it turns out, TEACHES-COURSE and TAKES-COURSE are also defined in terms of other situations as shown in Figure 3. It is possible, however, by a substitution procedure, to replace non-PRIMITIVE situations by their defining expressions until only PRIMITIVE situations remain in the expression, which now describes the original non-PRIMITIVE virtual extension. PRIMITIVE situations are the only ones known at the data level. TEACHES-OFFERING, OFFERING-OF, and TAKES-OFFERING are all PRIMITIVE situations. The virtual extension of the expression in Figure 2 can be computed using the SIDUR-defined connectives 'and', 'or', and 'not' logically as having their ordinary meaning, or procedurally as performing certain algebraic operations on extensions of data values. This will be discussed in Section 2.2. Figure 4 shows the PRIMITIVE situation expression that is equivalent to

```
(TEACHES-STUDENT (agent a) (object b)).
```

This expression is used to compute the virtual extension of the definition shown in Figure 2.

DATA-VALUE CLASSES

type:	- INTEGER, REAL, STRING, or TOKEN
form:	- syntactic structure of STRING 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 real numbers

OBJECT CLASSES

representative:	- name of a data-value class whose elements "stand in place for" real objects
superclasses:	- names of object classes of which the class under definition is a specialization
names:	- publicly available names for TOKEN-type objects
definition:	- situation used to define a TOKEN class

SITUATIONS

participants:	- the objects whose participation defines the situation, and the roles they play
cardinalities:	- maximum occurrences of participants
extension:	- whether the situation is expected to obey open-world or closed-world (the default) assumption
necessary:	- pre-conditions that are logically necessary for consistency
required:	- pre-conditions set by policy
definition:	- PRIMITIVE (stored) or a sigma expression specifying a virtual extension
sufficient:	- minimum that can suffice for definition:

COMPUTATIONS

participants:	- objects and values that serve as inputs or outputs for the computation
definition:	- the expression that specifies how to carry out the computation

ACTIONS

participants:	- objects involved in the defined behavior
prerequisites:	- pre-conditions of the action
results:	- sigma expression describing the action's result

Schema Construct Slots
Figure 1

```
situation TEACHES-STUDENT
   participants:  ((agent x PROFESSOR) (object y STUDENT))
   definition:  (and (TEACHES-COURSE (agent x) (object z))
                     (TAKES-COURSE (agent y) (object z)))
```

TEACHES-STUDENT Schema Definition
Figure 2

```
situation TEACHES-COURSE
   participants:  ((agent x INSTRUCTOR) (object y COURSE))
   definition:  (and (TEACHES-OFFERING (agent x) (object z))
                     (OFFERING-OF (agent y) (object z)))

situation TAKES-COURSE
   participants:  ((agent x STUDENT) (object y COURSE))
   definition:  (and (TAKES-OFFERING (agent x) (object z))
                     (OFFERING-OF (agent y) (object z)))
```

PRIMITIVE Schema Definitions
Figure 3

```
(and (TEACHES-OFFERING (agent a) (object w))
     (OFFERING-OF (agent z) (object w))
     (TAKES-OFFERING (agent b) (object w)))
```

PRIMITIVE TEACHES-STUDENT Expression
Figure 4

2.2 The Language of Sigma Expressions

Sigma expressions describe relationships between data values. Each sigma expression has associated with it a unique extension that is a collection of data values from the current database. The sigma expression notation is declarative in nature and thus resembles logic-based languages such as predicate calculus or PROLOG (Clocksin and Mellish, 1981; and Warren, 1980.) However, the object denoted by a sigma expression is an extension of data values rather than a single boolean truth value. Thus, each sigma expression is a non-procedural specification of a uniquely determined collection of stored data. Every sigma expression has a procedural interpretation with respect to the primitive data access operation *enquire. This interpretation determines the computation required to produce the needed extension. This collection is composed of a set of instances or tuples. Each tuple consists of a set of data values. Each data value is bound to, that is can be substituted for, a role in the sigma expression. Under this *enquire interpretation the set of all such tuples is the extension of the sigma expression. For example,

```
(TAKES-COURSE (agent y) (object z))
```

is a simple sigma expression whose current extension can be represented as shown in Figure 5. Sigma expressions such as the one above are also called situation expressions. A single binding tuple from the extension can be represented as:

```
<(agent --> P-10) (object --> C-1)>.
```

When the values from the tuple are substituted back into the sigma expression

```
(TAKES-COURSE (agent P-10) (object C-1)),
```

the resulting sigma expression, considered as a predicate, is an accurate statement with respect to the stored database.

The power of SIDUR's declarative representation stems from the fact that each sigma expression is subject to other interpretations than just *enquire. Under the *assert procedural interpretation, sigma expressions are used to describe operations on extensions of data values. These operations will cause the stored database to change with the result that the expression will become true with respect to the stored database. In the example above, the *assert interpretation of

```
(TAKES-COURSE (agent P-11) (object C-1))
```

would cause the binding tuple:

```
<(agent --> P-11) (object --> C-1)>
```

to be added to the TAKES-COURSE extension providing that it is not already there.

agent x	object y
P-21	C-1
P-10	C-1
P-3	C-1

TAKES-COURSE Extension
Figure 5

Note that a *deny interpretation is allowed for in the SIDUR model but is not strictly needed because *assert is defined such that the expression following it is made to be true in the stored database. Making the expression true can result in values being added and/or deleted so that all of the SIDUR update operators can be mapped to the *assert interpretation. In order to express deletion of values at the SIDUR level, the operators REFLECT-NOT or DENY can be used. Another way to denote deletion is by using REFLECT or ASSERT together with the 'not' connective. These operators will be explained later in this section. In general, the *deny interpretation of an expression such as

```
*deny (TAKES-COURSE (agent P-10) (object C-1))
```

can be defined in terms of *assert as follows

```
*assert (not (TAKES-COURSE (agent P-10) (object C-1)).
```

Sigma expressions, under both the *enquire and *assert interpretation, can also be used to:

1) request an extension with fewer participants than all of the ones appearing within the situation expression, as is provided by the relational algebra projection operation (Ullman, 1980),

2) specify constants in place of some or all of the variables, and

3) refer to multiple situations using the extensional connectives 'and' and 'or'.

Under the *enquire interpretation of a sigma expression:

1) 'and' roughly corresponds to the intersection of extensions,

2) 'or' roughly corresponds to the union of extensions,

3) 'not' roughly corresponds to set subtraction of extensions, and

4) 'empty' determines whether there are any tuples in the extension described by its situation expression.

Under the *assert interpretation of a sigma expression

1) 'and' means to *assert all of the nested expressions,

2) 'or' means to *assert one of the nested expressions,

3) 'not' means to carry out operations to make the nested expression not true in the current database, and

4) 'empty' has the same meaning as 'not'.

Further detail regarding sigma expressions can be found in Section 1.2 of the "SIDUR Manual" (Freiling, 1983c.)

A complete SIDUR operation consists of an operator and the sigma expression(s) upon which the operator is to act. There are six situation operators in SIDUR:

```
ENQUIRE
CHECK
REFLECT
REFLECT-NOT
ASSERT
DENY.
```

An operation such as

```
(REFLECT (TAKES-COURSE (agent P-1) (object C-1)))
```

is defined in terms of its effect on the stored database extension or in terms of stored data to be retrieved. The operator acts on the sigma expression that follows it. In this case, the stored database will be updated to include the information specified in the sigma expression.

ENQUIRE is a data access operator that specifies the retrieval of every tuple of values that meets the description in the situation expression.

(ENQUIRE <sigma expression>) is defined simply as

```
(*enquire <sigma expression>).
```

CHECK is a data access operator that determines whether there is at least one stored tuple that meets the description in the situation expression. In other words, CHECK determines whether the extension that is represented by the expression is 'full' or 'empty'. 'full' means that there are tuples qualified to be in the extension but that they are not returned in the extension of the query. 'empty' means that no stored tuples qualify to be in the extension.

(CHECK <sigma expression>) is defined as

```
    (not (empty (*enquire <sigma expression>))).
```

The last four situation operators are *assert operators. They cause additions and deletions of values in the stored database. Data values will be added only if no cardinality constraints (restrictions on the maximum number of times a data value may occur in a particular role extension) and if no necessary or required pre-conditions are violated.

The REFLECT operator causes the database to be changed so that the sigma expression will make a true statement with respect to the stored data. That is, the stored-data extension will be changed so that it "reflects" the information in the sigma expression. However, if necessary or required pre-conditions are defined in the schema for any of the situations that will have values added, the pre-conditions must all hold in the current database before it is changed.

(REFLECT <sigma expression>) is defined as

```
(*enquire <all pre-conditions and cardinality constraints>)
 if all pre-conditions and constraints hold then
      (*assert <sigma expression>).
```

ASSERT, a stronger update operator, can cause the stored database to be updated so that, not only does the sigma expression become true, but the sigma expression that represents the required pre-conditions becomes true. However, the necessary pre-conditions must already be met because the ASSERT operator does not cause any changes to bring them about. For ASSERT, required pre-conditions are policy changes or bookkeeping updates that will be handled automatically for the user. In contrast, the necessary pre-conditions are logically necessary to maintain database integrity and cannot be overridden.

(ASSERT <sigma expression>) is defined as

```
(*enquire <necessary pre-conditions/cardinality constraints>)
if necessary pre-conditions/cardinality constraints hold then
   begin
   (*enquire <required pre-conditions>)
   if all required pre-conditions do not hold then
```

```
      (REFLECT <required pre-conditions>)
   if all required pre-conditions hold in the database then
      (*assert <sigma expression>)
   end.
```

REFLECT-NOT causes changes in the stored database so that the sigma expression does not make a true statement about the stored database. If the changes result in data values to be added, the policies defined for REFLECT will apply to the operation.

(REFLECT-NOT <sigma expression>) is defined as

```
(REFLECT (not <sigma expression>)).
```

DENY also causes changes so that the sigma expression does not make a true statement, but any resulting data value additions are handled as if they were part of an ASSERT operation.

(DENY <sigma expression>) is defined as

```
(ASSERT (not <sigma expression>)).
```

There are three action operators

PERFORM
PERMIT?
PERMIT!.

These operators act on the sigma expressions that are found in the prerequisites: and results: slots of the action named in the operation. For example, Figure 6 shows the schema definition of the action COMPLETES. The PERFORM operator in an expression such as

```
(PERFORM (COMPLETES (agent P-6)
                    (object O-4)
                    (value "A")))
```

will act on the prerequisites: and results: slots of the action COMPLETES. PERFORM applies the ENQUIRE operator to the prerequisites: expression. If the ENQUIRE succeeds, the extension resulting from the ENQUIRE is used to apply REFLECT to the results: expression. PERMIT? applies the CHECK operator to the prerequisites: expression. PERMIT! applies the ASSERT operator to the prerequisites: expression.

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

Action COMPLETES
Figure 6

SIDUR also includes two compound operators

FOR

SINCE.

Each takes a "domain expression" and a list of simple operation requests. FOR applies the ENQUIRE operator to the domain sigma expression listed in the first part of the operation and then uses the resulting extension to carry out the simple operation requests listed in the second part of the operation. For example, Figure 7 shows a FOR operation that retrieves the IS-STUDENT/HAS-NAME extension and then PERFORMs the indicated action on it. SINCE applies the REFLECT, rather than ENQUIRE, operator to the domain expression and then uses the resulting extension to carry out the remaining simple operations. Figure 8 shows a SINCE expression that acts on the same sigma expression as does the FOR operator in Figure 7. The FOR operation will only graduate SALLY if she is already recorded as a student. The SINCE operation will first record SALLY as a student if she is not already, and then will graduate her. The usefulness of SINCE is that the domain expression represents an assumption that is enforced if it turns out not to be true at the time the operation is performed.

```
(FOR (x)
    (and (IS-STUDENT (agent x))
         (HAS-NAME (agent x) (value "SALLY-BROWN")))
    (PERFORM (GRADUATES (agent x) (object 3.96))))
```

FOR Operation
Figure 7

```
(SINCE (x)
    (and (IS-STUDENT (agent x))
         (HAS-NAME (agent x) (value "SALLY-BROWN")))
    (PERFORM (GRADUATES (agent x) (object 3.96))))
```

SINCE Operation
Figure 8

The object operators are

CREATE

DESTROY.

CREATE results in the creation of a system-generated object TOKEN whose first character is that of the object-class name listed after CREATE. DESTROY removes an object TOKEN from use in the stored database. Any tuples in which it is associated are also removed.

Please refer to the "SIDUR Manual" (Freiling, 1983c) for further details on the SIDUR operations and the language of sigma expressions. However, it should be mentioned that the SIDUR philosophy does not rely specifically on this exact set of operators or on their definitions. The methodology can be used to easily build new semantically motivated definitions.

III. THE DATA ACCESS INTERFACE

3.1 The OSIRIS Architecture

The architecture of the OSIRIS Integrated Information System (Freiling, 1983a) is designed to provide a hierarchy of independent modules. Each module has a well-defined function appropriate to the level at which it occurs in the system, and each is appropriate for interfacing with other modules. For example, the data-access level is designed to be a general purpose module that can support SIDUR as well as other high-level database modules that might be developed.

The primary focus in the design of the OSIRIS architecture is to provide support for an implementation of a semantic data model and schema. The three major components in the OSIRIS architecture are:

1) the semantic-level model,
2) the data-access-level model, and
3) the file-level model.

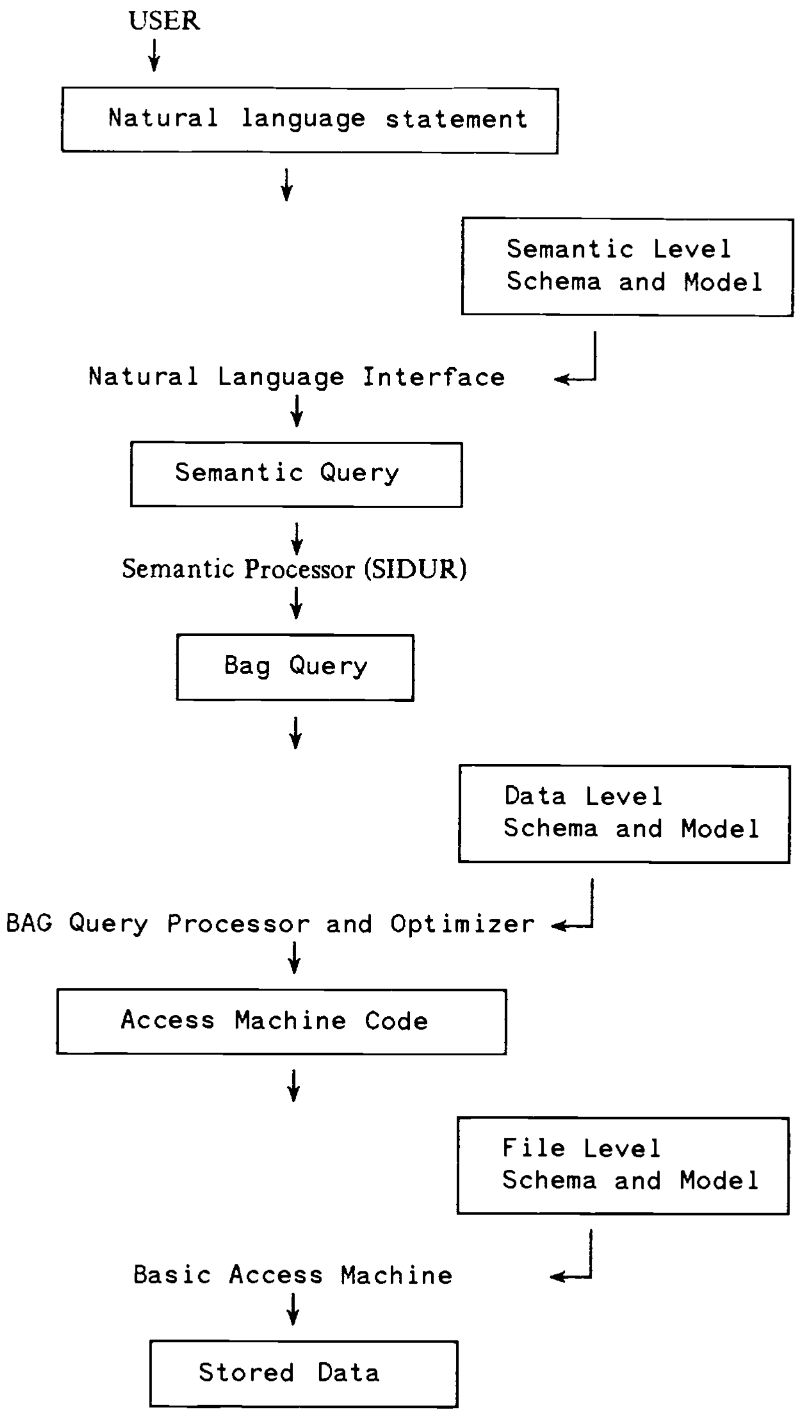
The division of OSIRIS into these three levels is based on placing the functions to accomplish the following major database tasks at the corresponding level:

1) semantic level--interpretation of stored data,
2) data level--specification of stored data value strings, and
3) file level--choice of storage structures to be used.

An expanded diagram of the OSIRIS architecture is shown in Figure 9.

The reason for choosing three levels, as well as for assigning the above three areas of function to each, is based on the ability of such an architecture to provide:

1) support for a semantic model,
2) data and schema independence,
3) isolation of user information concerns and data storage concerns,
4) ability for non-technical users to design schema,


USER
Natural language statement
Semantic Level Schema and Model
Natural Language Interface
Semantic Query
Semantic Processor (SIDUR)
Bag Query
Data Level Schema and Model
BAG Query Processor and Optimizer
Access Machine Code
File Level Schema and Model
Basic Access Machine
Stored Data

The OSIRIS Architecture--Expanded View
Figure 9

5) incremental, non-cumbersome re-design of schema,
6) expandable data storage, and
7) processing of complex queries in an efficient manner.

3.2 The Data-level Model

The data level of the OSIRIS architecture is the go-between level for the semantic and file levels. As such, it is unconcerned with matters of information interpretation or of data structure. It is concerned solely with the data that is actually present in storage, which in the case of the SIDUR model is composed of the collective extension of the PRIMITIVE situations. The data level is free from matters of interpretation. Consequently, it is not responsible for insuring the validity of inserted data values or for enforcing semantic-level constraints and pre-conditions. The semantic level handles constraints and pre-conditions by writing a data-level query that will perform such tasks transparently to the data level. This can be done by including low-level operators in the data-level query whose overall effect will cause such tasks to be performed.

The data-level query descriptions do not have reference to physical storage data structures. However, the higher-level associations between data values, which embody much of the information in a database system, are of prime concern to the data level. The associations at the data level represent logical structure and cannot be removed without losing information at the semantic level. The OSIRIS project has carefully considered alternative data-level representational models, including the relational model (Codd, 1970; and Date, 1975) and binary models (Bracchi, 1976.) These models were analyzed in terms of their capacity not only to represent these logical structures but also to be free of lower-level physical data structure concerns (Freiling, 1983a.) The <u>binary</u> <u>functional</u> <u>association</u> <u>(BFA)</u> model has been developed as the OSIRIS solution to the representational needs of this level. A BFA embodies the functional dependency (Codd, 1977) between one participant in a tuple grouping and a unique system-generated identifier called an <u>instance</u> <u>TOKEN</u>. Figure 10d shows examples of BFAs. Both the BFA model and the binary models overcome most of the rigidity of the relational model by allowing separation of attributes in storage.

However, because they do not require instance TOKENs, simple binary data models allow even more flexibility. But, because information taken from a relational model and re-expressed in a simple binary model cannot be fully reconstructed from its binary representation, the simple binary data models can lose information. The BFA model, like the binary models, treats all associations as being between pairs of data values. But unlike the binary models, the BFA model, is able to re-associate the pairs of values into the larger groupings of the original relation. This is done by using the instance TOKEN to provide a common reference for each relational group. Like the standard relational database tuple-id, it is used to identify a group of participants from the relation. In addition, in the BFA model a functional dependency between an instance TOKEN and each of the other attributes in the tuple is created.

So in OSIRIS, the structure that is known at the data level, and that is represented by the data-level schema, consists entirely of the BFA associations. These associations are the only data-level associations and are always between the instance TOKEN and each of the other participants in the tuple. As shown in Figure 10, the semantic level can tie each participant grouping together by means of the common instance TOKEN that is associated with each of its participants. Because all of the participants in an instance can still be associated into one grouping, the BFA model does not lose information. Because it does not force the physical data structures to be based on entire tuples, it allows independence and flexibility in the choice of a file-level access structure. The notation used for expressing BFAs is to concatenate situation name and role name so that the BFAs for a situation, for example TAKES-COURSE, become:

```
TAKES-COURSE / agent
TAKES-COURSE / object.
```

JOHN-BLACK	CS-430	10:00

a. Relational tuple

I-3	JOHN-BLACK	CS-430	10:00

b. Tuple id added to relational tuple

JOHN-BLACK	CS-430

CS-430	10:00

JOHN-BLACK	10:00

c. Binary associations

I-3	JOHN-BLACK

I-3	CS-430

I-3	10:00

d. Instance token related to each participant to form BFAs

I-3	JOHN-BLACK	CS-430	10:00

e. All participants with instance token I-3 can be re-grouped.

Binary Functional Associations
Figure 10

3.3 The Goal of Total Query Translation

The OSIRIS architecture specifies that each SIDUR operation is to be mapped entirely into a single data-level-language procedure. This one-to-one mapping allows maximum independence of the two levels from each other. The semantic level translates a query into the corresponding data-level form in order to isolate the lower levels from SIDUR's semantic concerns while still specifying, in the lower-level language, how to achieve the result needed at the semantic level. The data-level processor is free to optimize an entire operation. The semantic-level processor does not need to be concerned with preliminary results of operations or with determining the order in which the low-level access operations will be carried out.

3.4 The Data-level Language

SIDUR maps each semantic operation, in its entirety, into a data-level BAG ACCESS Language (BAGAL) query. BAGAL queries are partially a declarative, and partially a procedural, description of what data-level actions are needed in order to carry out the semantic operation. The declarative portions of the BAGAL query describe which data must be accessed, or which computations must be computed, in order to process the operation. The procedural portions of the BAGAL query are partially ordered control and update operations. Only those portions of an operation that must be carried out in a particular order to retain the meaning of the operation are forced to retain their order at the data level. Thus, the BAGAL queries produced by SIDUR must impose a partial ordering and conditional execution of data-level statements in order to achieve the proper result. The procedural operations are performed in the order found in the BAGAL query unless the data-level processor determines that altering the order will not affect the final result of the operation. The order of two operations can only be changed if neither one depends on the result of the other one. For example, the data-level statements to carry out a database *assert operation (see Chapter IV) should not be performed if preliminary *enquire results indicate that the current database extension does not meet the semantic pre-conditions necessary to permit the update. But, in general, the data-level processor is free to optimize the query and to maintain independence between storage and semantic concerns.

3.4.1 BAG Assignments

An ACCESS BAG assignment is a declarative specification of the binary functional associations of data values that are to be accessed. It is the fundamental declarative unit in the BAG query language. Data values are referenced in the ACCESS BAG by means of the BFAs in which they are associated. No ordering is imposed on the accessing of these values. Each set of accessed values forms one tuple. The entire extension of tuples is specified by the format list and becomes the value of the BAG. NO-NULL after an element in the format list indicates that the corresponding value in each tuple cannot be null. The extension of values can be referred to by referencing the BAG identifier of the ACCESS BAG.

The primary component of an ACCESS BAG assignment is the tuple pattern or description, which specifies:

1) the names and role names, i.e. the BFAs, of the PRIMITIVE situations to be accessed,

2) the description of any restrictions or constraints on which values need to be accessed.

The restrictions on retrieved values fall into two classes:

1) The values in the tuple must be such that any "constraint" statements that are made in the BAG are true.

2) Some of the values specified in the pattern consist of variables. Common use of a variable name to represent several different participants within a pattern implies that, for each tuple that is retrieved, the value of these participants will be equal.

Figure 11 shows an example of an ACCESS BAG whose BAG identifier is b-1 and whose format list is '(y-1 V-1 V-2)'. Two constraints '(V-1 = S-1)' and '(V-2 = "CS-430")' are specified. For each tuple, two BFAs from the MAY-TAKE situation extension are to be retrieved. I-1 is the variable denoting the instance TOKEN. For each tuple, each BFA retrieved from MAY-TAKE must have the same instance TOKEN. For each tuple, two BFAs from the IS-COURSE-NAME situation are to be retrieved. I-2 denotes the instance TOKEN. Each set of BFAs retrieved from the IS-COURSE-NAME situation must have an agent participant y-1 equal to the object participant y-1 of the corresponding MAY-TAKE BFAs. Both sets of BFAs are combined to form a tuple as specified in the format list. The SIDUR operation corresponding, in part, to this ACCESS BAG might be

```
(ENQUIRE
   (and (MAY-TAKE (agent S-1) (object y))
        (IS-COURSE-NAME (agent y) (object "CS-430")))).
```

```
((BAG b-1 (y-1 V-1 V-2))    <-
  (ACCESS                        < tuple pattern >
       (V-1 = S-1)
       (V-2 = "CS-430")
       (MAY-TAKE / agent I-1 V-1)
       (MAY-TAKE / object I-1 y-1)
       (IS-COURSE-NAME / agent I-2 y-1)
       (IS-COURSE-NAME / object I-2 V-2)))
```

ACCESS BAG Assignment
Figure 11

3.4.2 Tuple Operations

Operations on tuples consist of tuple variable assignments and update operations. Tuple variable assignments assign a single data value to one of the variables in the format list for each tuple in the extension. For example in

Figure 12,

(V-2 <- "TRUE")

assigns the string value "TRUE" to V-2.

The update operations are CHANGE, DELETE, and CINDEX. CHANGE causes data values to be added to the stored database. For example,

(CHANGE INTERESTING / agent I-1 V-1)

performs an update or an addition of a BFA. DELETE causes deletion of values. For example,

(DELETE INTERESTING / agent I-1 V-1)

removes the BFA. CINDEX is used as a precheck prior to CHANGE or DELETE operations in order to determine

1) whether BFAs being stored are already present in order to avoid duplicate data value insertions,

2) whether BFAs being deleted are really stored, and

3) whether an addition will violate any cardinality constraints.

For example as Figure 12 shows,

```
(I-1 <- (CINDEX (INTERESTING (agent V-1))))
(c-1 <- (COUNT I-1 I-1))
```

performs an access on the INTERESTING situation and assigns to I-1 an extension of all the tuples with the requested participant value of V-1. The COUNT operation counts the number of tuples in I-1. The result of the COUNT operation is stored in the variable c-1. The update is performed if the cardinality constraint will not be violated _or_ if the BFAs mentioned in the cardinality constraint are already stored. If they are already stored, the update will continue by storing the remainder of the BFAs.

Operations on tuples can occur in TUPLES BAGs, computation BAGs, and update BAGs. TUPLES BAGs (Figure 12) are BAGs containing the constants extracted from the semantic-level query. Computation BAGs (Figure 13) are any BAGs containing a data-level computation operator. Update BAGs (Figure 12) contain any of the update operators and are the only type of BAG that can cause a change in the stored data.

3.4.3 BAG Operations

Operations on BAGs are control procedures that act on entire extensions of tuples and result in constructing an extension that is assigned as the value of a new BAG. The BAG operations are described below.

1) The most often used BAG operations include AND-MERGE, OR-MERGE, and MINUS whose definitions correspond roughly to the SIDUR situation connectives 'and', 'or', and 'not'.

2) Other BAG construction operators, EMPTY and FULL, create BAGs corresponding to SIDUR's definition for 'empty' and 'full'.

3) COLLECT is a special data-level operator that assigns an entire extension as the value of a variable so that the extension can be passed to a computation as argument. COLLECT takes as arguments the BAG containing the extension as well as two or more lists of variables. The first list contains the variables common to other portions of the query so that accesses for the same values indicated both inside and outside the computation can be optimized and also performed in the proper order. Subsequent lists begin with a new variable. The remainder of each list chooses variables to be collected under the new variable. COLLECT is needed for two reasons. The BAGAL computational operators take variables (not BAGs) as participants so COLLECT binds the needed extension to the variable. If several participants to a computation take extensions as participants, each different extension can be bound to a different variable without the need for forming multiple BAGs. In addition, in complex expressions that include ACCESS expressions as well as computations, the results of some of the accesses must be computed prior to doing the computations in order to pass their results as arguments to the computations. Refer to Figure 43 for an example of this. COLLECT is used to force ACCESS BAGs to be participants of the computations by merging their results wih ACCESS results obtained for the participant expressions of the computations. Otherwise, the result of some computational expressions will be wrong. Recall that the data level is free to optimize the order of BAG ACCESS statements unless the BAGAL query specifies the order.

4) TUPLES is an operator that creates a constant TUPLES BAG extension using constant data values from the SIDUR operation.

5) FOR causes an operation to be performed on each tuple in the extension of a BAG.

6) IF determines whether the indicated BAG condition holds and, if it does hold, causes the associated operation to be performed.

7) CASE uses the value of its index variable to determine which of the associated BAG operations is to be performed.

Figure 13 shows a data-level query that accesses the PRIMITIVE extension corresponding to SIDUR's FINAL-GPA extension, COLLECTs the resulting extension as the value of V-1, forms a TUPLES extension of constants in the operation, AND-MERGEs the BAGs containing arguments to the computation, and then performs the computation ACCUMULATE. The meaning of this query at the semantic level is to find the sum of all the of value participants in the FINAL-GPA situation.

```
(REFLECT (INTERESTING (agent V-1 P-1)
                      (result V-2 "TRUE")))
```

```
     < TUPLES BAG forms the constant tuple to be reflected >

((BAG B-1 (V-2 V-1)) <-
 (TUPLES ((V-2 <- "TRUE") (V-1 <- P-1))))

< Update BAG--This command checks cardinalities, verifies that
values are not already present, and then performs the update >

((BAG B-2 (V-2 V-1)) <-
 (FOR (BAG B-1 (V-2 V-1))
                           < data access to check cardinality >
   ((I-1 <- (CINDEX (INTERESTING (agent V-1))))
          < data access statement--check if template exists >
    (I-2 <- (CINDEX (INTERESTING (agent V-1) (result V-2))))
    (c-1 <- (COUNT I-1 I-1))
                         < check cardinalities--see if count is
                           less than the cardinality constraint
                           or if already stored >
    (IF (OR (c-1 < 1) (I-2 <> NULL)) THEN
        ((IF (I-2 = null) THEN      < if not already stored >
             ((I-2 <- (ICREATE INTERESTING))
                                             < perform update >
              (CHANGE INTERESTING / agent I-2 V-1)
              (CHANGE INTERESTING / result I-2 V-1)))
         (I-3 <- (CINDEX (IS-PERSON (agent V-1))))
         (IF (I-3 = null) THEN      < see if already stored >
             ((I-3 <- (ICREATE IS-PERSON))
                         < perform update of new object type as
                           required by query semantics >
              (CHANGE IS-PERSON / agent I-3 V-1))))))))

                                              < RETURN result >
(RETURN (BAG B-2 (V-2 V-1)))
```

Data-level *assert Procedure
Figure 12

```
(ENQUIRE
    (SUM
         (domain V-1 (sigma (x-1)
                     (FINAL-GPA (value x-1))))
         (result y-1)))

(ENQUIRE
    (ACCUMULATE          < sigma expansion of SUM >
         (domain V-1 (sigma (x-1)
                     (FINAL-GPA (value x-1))))
         (mapping V-2 "PLUS")
         (result y-1)))
```

```
                                    < data retrieval for *enquire >

((BAG b-1 (x-1))  <-
 (ACCESS (FINAL-GPA / value I-1 x-1)))

                       < BAG to COLLECT extension argument into V-1 >

((BAG B-2 (x-1 V-1))  <-
 (COLLECT (BAG b-1 (x-1)) nil (V-1 x-1)))

                        < TUPLES BAG to form extension of constants >

((BAG B-3 (V-2))  <- (TUPLES ((V-2 <- "PLUS"))))

                        < AND-MERGE to gather computation arguments >

((BAG B-4 (x-1 V-1 V-2))  <-
 (AND-MERGE ((BAG B-2 (x-1 V-1))
             (BAG B-3 (V-2)))))

                                          < BAG to do computation >

((BAG B-5 (y-1 x-1 V-1 V-2))  <-
 (FOR (BAG B-4 (x-1 V-1 V-2))
      (y-1 <- (ACCUMULATE (V-1 V-2 y-1)))))

                                                  < RETURN result >
(RETURN (BAG B-5 (y-1)))
```

Computation BAGAL Query
Figure 13

IV. TRANSLATION OF SEMANTIC QUERIES

4.1. The Query Translation Process

4.1.1 Description of the Translation Process

The interpretation of SIDUR query operations is based on re-expressing each SIDUR expression in terms of two primitive semantic operations, *enquire and *assert, and in terms of how the results of these primitive operations are to be related to each other. *enquire is the primitive data access operation. *assert is the primitive data update operation. These primitive operations correspond to data-level language access and update operations. For query processing in the OSIRIS architecture, the primitive operations are translated into a set of partially ordered BAGAL (BAG ACCESS Language) operations. BAGAL conditional and procedural operations are used to control update operations. BAGAL set and computation operators are used to specify the relationships between the results of the primitive access and update operations. Thus, each SIDUR operation is mapped in its entirety into a single BAGAL procedure. The statements in the BAGAL procedure are interpreted at the data level to produce extensions of data values. Extensions are merged together at the data level using the set and computation operators to produce the final virtual extension specified in the original SIDUR query.

4.1.2 The Canonical Form of an Operation

In an effort to simplify the translation of operations such as the one shown in Figure 14 and also to provide a predictable control structure in what might otherwise be a data-level query of arbitrarily complex structure, an expression simplification algorithm has been developed. This algorithm converts each complex situation expression into a simplified form and includes with the simplified form all other information that will be needed by the translation routines to produce a corresponding data-level query. The design of this algorithm and the form of the resulting simplified expression are the result of dealing, at least in part, with many of the difficult issues of implementing the SIDUR architecture. As discussed in the following paragraphs, these issues include:

1) simplifying the translation process and the data-level query while retaining the original meaning of the operation,

2) re-expressing non-PRIMITIVE situations and non-SYSTEM computations in terms of their schema definitions,

3) insuring that all update operations are atomic so that an entire logical transaction either fails or succeeds thereby maintaining the integrity of the database,

```
(ENQUIRE
    (or
        (IS-FACULTY (agent V-1 S-1))
        (and
            (or
                (IS-STUDENT (agent V-1 S-1))
                (IS-STAFF-STUDENT (agent V-1 S-1)))
            (FINAL-GPA (agent V-1 S-1)))))
```

Operation Using Connectives
Figure 14

4) handling disjunctive update operations,

5) determining that constant data value participants are of the proper type,

6) insuring that all schema-specified pre-conditions and cardinality constraints hold in the current database extension prior to an update operation, and

7) allowing operations on an extension of tuples as well as on a single instance of data values.

The canonical form of an *enquire operation consists only of the *enquire expression. The canonical form of an update operation consists of exactly one *assert expression and one *enquire expression and certain other expressions containing information needed to process the query. The *assert expression represents those extensions that must be updated. The *enquire expression represents the pre-conditions that must be met for a tuple to be eligible for the update. The other expressions in the canonical form of an update operation consist of cardinality constraints and situation names needed to resolve ambiguous updates. These other expressions are used only in update operations and will be discussed in Chapter V.

The general approach used to prepare a situation expression such as the one in Figure 14 for translation is to re-phrase each expression into a semantically equivalent *enquire and/or *assert expression. An expression in the *enquire or *assert part is in the form of a disjunct of conjuncts

```
(or  (and S11 S12 S13)
     (and S21 S22 S23)
     (...            ))
```

where Snn is a single situation expression. Each set of situation expressions following an 'and' connective is called a branch or conjunct. Figure 15 shows the ENQUIRE operation of Figure 14 after it has been converted into this form. Note that in some of the Figures in Chapter IV and V, system variables of the form 'V-1' have been inserted in front of constant participant values. This is to aid the reader in comparing later figures that show further processing of these same expressions. Section 5.6 discusses why these variables are added.

For an ENQUIRE or CHECK operation, the simplification algorithm uses the situation expression in the ENQUIRE operation to produce the *enquire expression of the canonical form. The canonical form is then used to guide the SIDUR translation routines.

For a REFLECT operation, the simplification algorithm produces the *assert expression of the canonical form from the situation expression in the operation. Both

```
(ENQUIRE
   (or
      (IS-FACULTY (agent V-1 S-1))                 BRANCH 1
      (and                                         BRANCH 2
         (IS-STUDENT (agent V-1 S-1))
         (FINAL-GPA (agent V-1 S-1)))
      (and                                         BRANCH 3
         (IS-STAFF-STUDENT (agent V-1 S-1))
         (FINAL-GPA (agent V-1 S-1)))))
```

Operation With Simplified Connective Expression
Figure 15

required and necessary pre-conditions belonging to the *assert expression become the *enquire expression of the canonical form. Note that the algorithm classifies required pre-conditions as part of the *enquire expression. The list structure of the canonical form is used to guide the SIDUR *assert translation routines as will be discussed in later paragraphs. An example of a REFLECT operation and its re-expressed form is shown in Figure 16. This example will be discussed in greater detail in this chapter and in Chapter V. Appendix I contains the schema for this operation.

ASSERT is handled in the same manner as REFLECT with the exception that the simplification algorithm lists required pre-conditions as part of the *assert expression instead of as part of the *enquire expression. Only necessary pre-conditions will be listed as the *enquire expression.

REFLECT-NOT is handled by inserting a 'not' at the front of the situation expression. Then, the simplification algorithm treats it as a REFLECT operation. For example,

```
(REFLECT-NOT (and (<expr 1>)
                  (<expr 2>)))
```

is treated as

```
(REFLECT (not (and (<expr 1>)
                   (<expr 2>)))).
```

```
                    ORIGINAL OPERATION

(REFLECT (and
             (IS-EVENT-NAME (agent x) (object V-10 "AI-TOPICS"))
             (or
                 (IS-NOON-MEETING (agent x))
                 (IS-SEMINAR (agent x)))))

      OPERATION EXPRESSED AS A DISJUNCT OF CONJUNCTS

 (REFLECT (or
             (and                                      BRANCH 1
                  (IS-EVENT-NAME (agent x)
                                 (object V-10 "AI-TOPICS"))
                  (IS-NOON-MEETING (agent x)))
             (and                                      BRANCH 2
                  (IS-EVENT-NAME (agent x)
                                 (object V-10 "AI-TOPICS"))
                  (IS-SEMINAR (agent x)))))
```

REFLECT Operation and Simplified Expression
Figure 16

DENY is handled in the same manner as is REFLECT-NOT with the exception that the simplification algorithm treats the operation as an ASSERT operation after the 'not' is inserted at the beginning of the situation expression.

So, the simplification algorithm produces a generic, update canonical form for any of the four update operators. After this, SIDUR *assert translation routines are guided solely by the list structure of the canonical form, which represents the full semantics of the operation. The *assert translation routines are not explicitly aware of which particular update operation is being translated. Furthermore, the lower level *enquire expression translation routines are not aware of which type of operation, update or data access, an *enquire expression belongs to. The canonical form is the first step in expressing the semantics of an operation in lower level statements that are unconcerned with semantics but whose actions will have the effect of preserving the semantics.

4.1.3 Sigma Expansion

Each non-PRIMITIVE situation definition is a declarative specification of which stored (i.e. PRIMITIVE) data must be combined to form a useful extension of derived or virtual data. Non-PRIMITIVE situations are unknown at the data level because only the extensions of PRIMITIVE situations are actually stored. All computation expressions represent specifications of virtual data. However, only the SYSTEM computations are implemented at the data level so non-SYSTEM computations are defined in the SIDUR schema in terms of SYSTEM computations. Therefore, an expression that contains only PRIMITIVE situations and SYSTEM computations must be derived from any SIDUR expression that contains non-PRIMITIVE or non-SYSTEM expressions. In order to derive an expression that represents the same extension of data values, a semantic query processor must insure that the participant data values specified by the original expression are also specified by the derived expression and that the same associations between these participants hold.

When a situation expression is being simplified, each non-PRIMITIVE situation or non-SYSTEM computation included in the expression must be re-expressed in an extensionally equivalent manner using only situations and computations with PRIMITIVE or SYSTEM definitions. The new expression, which is derived from the original and which contains only PRIMITIVE and SYSTEM components, is then substituted in place of the original expression. The process of substituting new expressions for old is called _sigma expansion_. A PRIMITIVE or SYSTEM expression is obtained by a process of associating the participants of the original expression with their corresponding role names and variables in the situation's schema specification and then substituting them into the schema _definition_: slot. This process is called _sigma binding_. Depending on the amount of nesting of non-PRIMITIVE situations in the schema _definition_: slots, it may be necessary to apply the sigma binding process multiple times, first to a non-PRIMITIVE situation and then to each derived non-PRIMITIVE _definition_: until all situations and computations are PRIMITIVE or SYSTEM.

For example, the expression in Figure 17 might, at the SIDUR level, represent the extension shown in Figure 18. This extension may be interpreted as a list of all tuples of TOKEN data values that represent the instructors who teach, and the students who take, the particular course represented by C-1. However, these particular situations, as well as the association between values that is implied by the expression, are unknown at the data access level. Figure 19 shows the schema definitions of these two situations. The PRIMITIVE schema definition: slots of TEACHES-OFFERING, OFFERING-OF, and TAKES-OFFERING indicate that what is known at the data access level is that the three extensions shown in Figure 20 are part of the stored database extension. Nothing about the derived association between INSTRUCTOR-STUDENT-COURSE is known. Figure 21 is an equivalent re-expression of Figure 17. So, the data-level query must be expressed in terms of the three PRIMITIVE situations and must contain elements of procedure control and/or data specification to cause the virtual extension to be produced. Figure 22 shows the non-PRIMITIVE situation expression from the example that we are looking at in detail. Also shown is its equivalent sigma-expanded form.

```
(and
    (TEACHES-COURSE (agent x) (object V-7 C-1))
    (TAKES-COURSE (agent y) (object V-7 C-1)))
```

SIDUR Representation of an Extension
Figure 17

x	y	V-7
P-2	P-1	C-1
P-2	P-3	C-1

SIDUR Virtual Extension
Figure 18

```
situation TEACHES-COURSE
     participants:  ((agent x INSTRUCTOR) (object y COURSE))

     definition:  (and (TEACHES-OFFERING (agent x) (object z))
                       (OFFERING-OF (agent y) (object z)))

situation TAKES-COURSE
     participants:  ((agent x STUDENT) (object y COURSE))

     definition:  (and (TAKES-OFFERING (agent x) (object z))
                       (OFFERING-OF (agent y) (object z)))
```

PRIMITIVE Schema Definitions
Figure 19

	agent	object
TEACHES-OFFERING	P-2	O-1
TAKES-OFFERING	P-1 P-3	O-1 O-1
OFFERING-OF	C-1	O-1

Present Extensions
Figure 20

```
(and
   (TEACHES-OFFERING (agent x) (object z))
   (TAKES-OFFERING (agent y) (object z))
   (OFFERING-OF (agent C-1) (object z)))
```

Result of Sigma Expansion
Figure 21

SCHEMA DEFINITIONS

```
situation IS-SEMINAR
    participants: ((agent x OFFERING))
    necessary: (IS-COURSE (agent x))
    required: (HAS-TITLE (agent x) (object "SE-400"))
    definition: (and
                      (IS-OFFERING (agent x))
                      (LIMIT (agent x) (value 12)))
    extension: (CLOSED-WORLD)

situation IS-OFFERING
    participants: ((agent x OFFERING))
    definition: (OFFERING-OF (object  x))

situation OFFERING-OF
    participants: ((agent x COURSE) (object y OFFERING))
    definition: (PRIMITIVE)
    extension: (CLOSED-WORLD)

situation LIMIT
    participants: ((agent x OFFERING) (value y CLASS-LIMIT))
    cardinalities: ((1 x))
    definition: (PRIMITIVE)
    extension: (CLOSED-WORLD)
```

ORIGINAL OPERATION

```
(REFLECT
  (and (IS-EVENT-NAME (agent x) (object V-10 "AI-TOPICS"))
       (or (IS-NOON-MEETING (agent x))
           (IS-SEMINAR (agent x))))          < non-PRIMITIVE >
```

OPERATION WITH THE NON-PRIMITIVE SITUATION REPLACED BY ITS SIGMA-BOUND DEFINITION

```
(REFLECT
  (and (IS-EVENT-NAME (agent x) (object V-10 "AI-TOPICS"))
       (or (IS-NOON-MEETING (agent x))
           (and                  < sigma-expanded definition: slot >
                (OFFERING-OF (object x))
                (LIMIT (agent x) (value V-14 12))))))
```

Expanded REFLECT Expression
Figure 22

4.2 The Code Generation Process

4.2.1 Use of Set Operators to Combine Results

One of the most often used features of the SIDUR language is the arbitrary nesting of the connectives 'and', 'or', 'not', and 'empty' within the scope of a situation operator. Figure 14 shows a typical operation of this type.
Under the *enquire interpretation of a sigma expression

1) 'and' corresponds roughly to the intersection of extensions,

2) 'or' corresponds roughly to the union of extensions,

3) 'not' corresponds roughly to set subtraction of extensions, and

4) 'empty' determines whether there are any tuples eligible to be in the extension described by its situation expression.

Under the *assert interpretation of a sigma expression

1) 'and' means to *assert all of the nested expressions,

2) 'or' means to *assert one of the nested expressions,

3) 'not' means to carry out operations to make the nested expression not true in the current database, and

4) 'empty' has the same meaning as 'not'.

Based on the definitions given in the "SIDUR Manual" (Freiling, 1983c) for the 'and', 'or', and 'not' connectives, the extensions specified by an expression and by its corresponding simplified form are equivalent. The BAGAL operators that correspond to these connectives are AND-MERGE, OR-MERGE, and MINUS, respectively. For example, the extension of Figure 23 can be produced by taking the union (as defined for the 'or' connective) of the extensions of <expr 2> and <expr 3> and then intersecting (as defined for the 'and' connective) this resulting extension with the extension of <expr 1> to produce the extension represented by the entire expression. An equivalent form of the expression is obtained by distributing the simple operand of the 'and' connective over the nested 'or' connective. In other words, the first part of the 'and' expression is listed twice, once with each operand of the 'or' expression to obtain the structure shown in Figure 24. This is the basic algorithm that computes the *enquire or *assert expression. Figure 16 shows how this algorithm was applied to the example being followed in this chapter.

```
(and (<expr 1>)
     (or (<expr 2>)
         (<expr 3>)))
```

Connective Expression
Figure 23

```
(or (and (<expr 1>)
         (<expr 2>))
    (and (<expr 1>)
         (<expr 3>)))
```

Simplified Connective Expression
Figure 24

The extension of the expression in Figure 24 would be computed by intersecting the extension of <expr 1> first with the extension of <expr 2> and then with the extension of <expr 3> to obtain two preliminary extensions. Then a union of these two extensions would be taken to produce the final extension. If the extensions of the three expressions are as shown in Figure 25, then the result of computing the final extension using the first form of the expression (Figure 23) is shown in Figure 26 and the result using the second form (Figure 24) is shown in Figure 27.

<expr 1>

x	y
a	b
c	d

<expr 2>

x	y
a	b
e	f

<expr 3>

x	y
g	h

Sample Extensions
Figure 25

(or (<expr 2>)
    (<expr 3>)

x	y
a	b
e	f
g	h

= r1

(and (<expr 1>)
    (r1))

x	y
a	b

Result of Computing Expression in Figure 23
Figure 26

```
(and ((<expr 1>)
     ((<expr 2>))
```

x	y
a	b

= r2

```
(and ((<expr 1)
     ((<expr 3>))
```

x	y

= r3

```
(or (r2)
    (r3))
```

x	y
a	b

Result of Computing Expression in Figure 24
Figure 27

4.2.2 Use of Negation

In an *enquire expression, when the 'not' connective encloses a closed-world situation, it denotes set subtraction. The equivalent BAGAL operator is MINUS. The extension that is computed from the expression within the scope of the 'not' is subtracted from the extension that is computed from the outer portion of the expression. The outer portion, as well as the 'not' expression, must be enclosed within the scope of the 'and' connective as show here

```
(and ((<expr 1>)
     ((<expr 2>)
     (not ((<expr 3>))).
```

This expression is computed by taking the outer portion, which is the intersection of the extension of <expr 1> and the extension of <expr 2>, and then subtracting the extension of <expr 3> from the result.

Any situation within the scope of 'not' will be listed in the canonical form with 'not' preceding it. See Section 4.3.5 for a discussion of the scope of the 'not' connective. This results in a list of situation expressions comprising each of the branches. For each set of situations in a branch, the extension of all closed-world situations that are preceded by the 'not' connective is subtracted from the extension produced by intersecting the extensions of the other situation expressions in the branch. A BAGAL example of this is shown in Figure 30.

For open-world situations that are preceded by 'not', data accesses on the negative extension of the situation are done along with the closed-world situation accesses, if any, in the branch.

4.2.3 Building Extension of the *enquire Part

As mentioned earlier, each semantic operation is processed into a canonical form consisting mainly of a single *enquire expression and optionally a single *assert expression. The first task in processing a data-level query is to build the data-level representation for the extension for the *enquire operation. Each branch of the disjunct will contain situations under the scope of the 'and' connective. One or two ACCESS BAGs are created for each *enquire expression. An ACCESS BAG is created for each set of situations that are not preceded by 'not'. This BAG is called the positive BAG. Another ACCESS BAG is created if 'not' precedes any closed-world situations in the branch. Then for each branch, BAGAL code is created to build a BAG representing the subtraction of the negative extension from the positive BAG.

For ENQUIRE and CHECK, a data-level BAG assignment to produce the OR-MERGE (union) of the extensions resulting from each branch is then created. If there are no computations associated with any of the branches, the final result of the *enquire will be that of the OR-MERGE. Examples are shown in Figure 28 and Figure 30. Computations are discussed in Section 4.3.4. Updates are discussed in the next sections.

```
(ENQUIRE
    (or
        (IS-FACULTY (agent V-1 S-1))
        (and
            (or
                (IS-STUDENT (agent V-1 S-1))
                (IS-STAFF-STUDENT (agent V-1 S-1)))
            (FINAL-GPA (agent V-1 S-1)))))

< See Figure 54 for the canonical form >
```

```
                              < ACCESS BAG for BRANCH 1 of disjunct >

((BAG b-1 (V-1)) <- (ACCESS
   (V-1 = S-1) (IS-FACULTY / agent I-1 V-1)))

                              < ACCESS BAG for BRANCH 2 of disjunct >

((BAG b-2 (V-1 y-1 z-1)) <- (ACCESS
   (V-1 = S-1)
   (IS-PERSON / agent I-2 V-1)                < IS-STUDENT expansion >
   (TAKES-OFFERING / agent I-3 V-1)           <      "          "    >
   (TAKES-OFFERING / object I-3 z-1)          <      "          "    >
   (OFFERING-OF / agent I-4 y-1)              <      "          "    >
   (OFFERING-OF / object I-4 z-1)             <      "          "    >
   (FINAL-GPA / agent I-5 V-1)))

                              < ACCESS BAG for BRANCH 3 of disjunct >

((BAG b-3 (V-1)) <- (ACCESS
     (V-1 = S-1) (IS-STAFF-STUDENT / agent V-1 I-7)
     (FINAL-GPA / agent I-6 V-1)))

                          < OR-MERGE the extensions from each branch >

((BAG B-4 (y-1 z-1 V-1)) <-
 (OR-MERGE
    ((BAG b-1 (V-1))
     (BAG b-2 (V-1 y-1 z-1))
     (BAG b-3 (V-1)))))

                                     < RETURN the requested extension >
(RETURN (BAG B-4 (V-1)))
```

Data-level ACCESS BAGs and Merge Operations
Figure 28

4.2.4 Update Atomicity

The SIDUR definition prescribes failure for any update that fails to meet any of the pre-conditions, cardinality constraints, or object-type specifications for the underlying situations. SIDUR also allows compound operations on an extension of tuples. If one tuple does not meet its set of pre-conditions, should the operation fail for all of the tuples or for only the failing tuple? The integrity of the database can be guaranteed in either case. The question of whether atomicity of transactions should be implemented at the tuple level or at the extension level depends on which approach would be most useful to users of the database. Since it was felt that implementing a tuple level atomicity of transactions is the more fundamental method and that users may find this added granularity more useful, this SIDUR implementation has taken the tuple level approach. It was also felt that converting to an extension level approach would be easier than converting to a tuple level approach if later it is learned that users prefer the extension level approach.

The tuple level approach has necessitated that the data-level query contain operations to determine which pre-conditions apply to each tuple, which tuples meet their pre-conditions, and which tuples do not. The *assert operation is ignored for each of the tuples that does not meet its pre-conditions. Recall that the data level itself is unaware of semantic level pre-conditions so BAGAL operations that will have the required effect must be translated. The major justification of our use of a canonical form is that it reduces much of the complexity of translating and performing an update. An update operation, which can have an arbitrarily complex structure of 'and', 'or', and 'not' connectives, is transformed into the sets of expressions represented by the canonical form. Each set corresponds to one branch of the disjunct of the canonical form of the *assert expression. Each branch is uniform in structure and contains all of the information needed to update a tuple of data values into the stored database extension. For example, all of the necessary pre-conditions that apply to the situations in a branch become the *enquire expression for that branch. All of the situations that are to be updated become part of the *assert expression for that branch. The next section discusses how a particular branch is chosen for the update of each tuple. An example of this is shown in Figure 31.

4.2.5 Update Ambiguity

SIDUR defines four situation update or *assert operators, REFLECT, ASSERT, REFLECT-NOT, and DENY. These operators can act on arbitrarily complex sigma expressions containing any of the connectives 'and', 'or', and 'not'. The complex sigma expression can represent an extension of a single or multiple tuples. In addition, when used within the scope of an insertion operation, 'or' presents a potentially ambiguous update operation. For example,

```
(REFLECT (or (IS-STUDENT (agent P-1))
             (IS-INSTRUCTOR (agent P-1))))
```

means to *assert either the IS-STUDENT or the IS-INSTRUCTOR expression.

When used within the scope of a deletion operation, 'and' likewise presents a potentially ambiguous deletion of stored data. The meaning of 'and' nested within one of these operations is that changes should be performed so that the expression being denied does not have a 'full' extension in the current database. For example,

```
(REFLECT-NOT (and (IS-STUDENT (agent P-1))
                  (IS-INSTRUCTOR (agent P-1))))
```

is equivalent to

```
(REFLECT (not (and (IS-STUDENT (agent P-1))
                   (IS-INSTRUCTOR (agent P-1))))).
```

These expressions are also equivalent to

```
(REFLECT (or (not (IS-STUDENT (agent P-1))
             (not (IS-INSTRUCTOR (agent P-1))))).
```

Therefore, these expressions involve deleting a tuple either from the extension of the IS-STUDENT situation or from the extension of the IS-INSTRUCTOR situation in order to make the original expression false with respect to the stored database.

Eventually, the OSIRIS project will include as one of its research projects the automated disambiguation of a disjunctive expression that is within the scope of an update operator. Until automated methods are devised, and even then in cases where the system is unable to choose between the branches of a disjunct, it is the intent of the SIDUR design that the user be given the opportunity, on a tuple-by-tuple basis, to choose which situations will be affected by the operation.

For example, the operation in Figure 16 contains the disjunctive expression

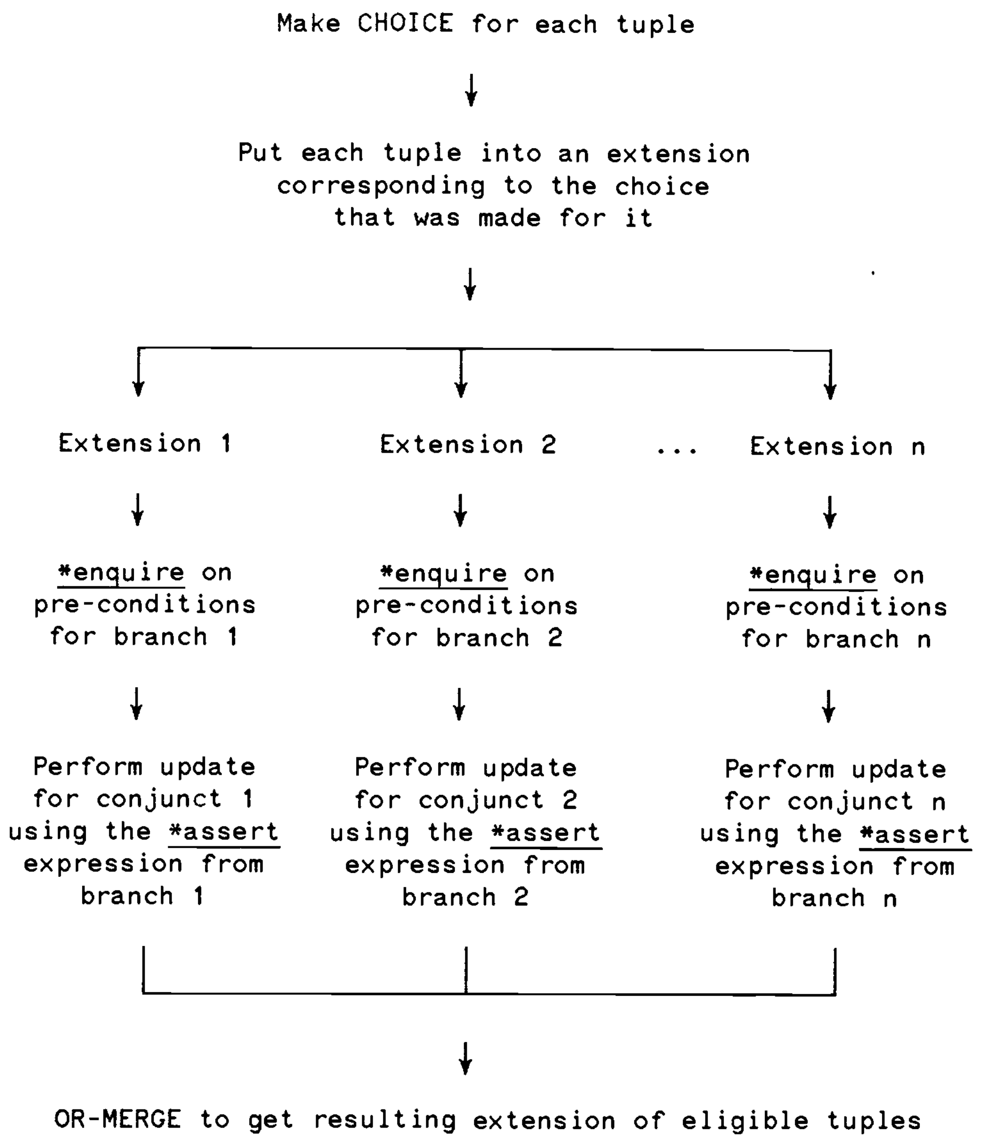
```
(or
   (IS-NOON-MEETING (agent x))
   (IS-SEMINAR (agent x))).
```

This means that for each tuple a choice will be made between IS-NOON-MEETING and IS-SEMINAR. The canonical form of the operation includes the following two branches

```
                                                BRANCH 1
(and (IS-EVENT-NAME (agent x)
                    (object "AI-TOPICS"))
     (IS-NOON-MEETING (agent x)))
                                                BRANCH 2
(and (IS-EVENT-NAME (agent x)
                    (object "AI-TOPICS"))
     (IS-SEMINAR (agent x))))).
```

Branch 1 corresponds to a choice of IS-NOON-MEETING. Branch 2 corresponds to a choice of IS-SEMINAR. If IS-NOON-MEETING is chosen for a tuple, the tuple will be determined eligible for the update according to the pre-conditions of the *enquire expression of branch 1 and then updated into the *assert expression of branch 1. If IS-SEMINAR is chosen for a tuple, then likewise the *enquire and *assert expressions of branch 2 will be used for its update. The flowchart for a data-level disjunctive operation is shown in Figure 29. How the correspondence between choices and branches is implemented is discussed in Section 5.4.1.

In an *assert operation, the *assert expression is equivalent to the expanded form of the original expression. For the expression illustrated in Figure 31, the corresponding *assert version represents the same two distinct branches or sets of situations between which the user will choose in order to disambiguate the update operation. The transformation to the canonical form, while it does not create expressions that are optimal in terms of performance, is necessary in order to permit appropriate use of the CHOICE function in the data-level query. The CHOICE function will determine which branch is chosen for the update operation for each tuple. The user will be asked to make this choice for every tuple being considered for the update operation. The choice process will be discussed further in Chapter V.


Make CHOICE for each tuple
Put each tuple into an extension corresponding to the choice that was made for it
Extension 1
Extension 2
...
Extension n
*enquire on pre-conditions for branch 1
*enquire on pre-conditions for branch 2
*enquire on pre-conditions for branch n
Perform update for conjunct 1 using the *assert expression from branch 1
Perform update for conjunct 2 using the *assert expression from branch 2
Perform update for conjunct n using the *assert expression from branch n
OR-MERGE to get resulting extension of eligible tuples

Disjunctive Operation Flow Chart
Figure 29

4.3 Query Interpretation

4.3.1 General Problems

Prior to translation, the syntactic form of each semantic operation is verified to be in compliance with the SIDUR BNF definition for that type of data manipulation operation. The SIDUR BNF is shown in Appendix E. Any ill-formed queries must be corrected by the user.

Because of implementation concerns, the SIDUR definition restricts the use of the 'not' connective when it is used in an ENQUIRE or CHECK operation. If the 'not' expression contains a closed-world situation, the 'not' can occur only within the scope of an 'and' connective that contains at least one other situation expression whose result can be obtained by a BAGAL access operation. This is so that there will be two extensions for the set subtraction (MINUS) operation. For open-world situations that are enclosed by 'not', the negative extensions can be queried directly like any other situation so set subtraction need not be performed. The SIDUR BNF definition also restricts the expression under the scope of the 'not' to be a single situation expression or a single computation expression. This implementation allows any well-formed situation expression including those with connectives to appear within the scope of the 'not' connective because use of the implementation showed that:

1) Non-PRIMITIVE situations are often defined by complex expressions so the implementation needs to be able to handle this sort of structure within a 'not' when processing schema definition: slots.

2) Such structures are convenient to use in SIDUR operations as well as in schema definition: slots.

However, this implementation does not permit computation expressions within the scope of the 'not' connective.

Prior to translating an operation that will add data values to a situation extension, the query processor verifies user-supplied data values to be of the correct type as specified in the schema.

Objects whose corresponding data value type is INTEGER or REAL must be verified to be within the allowable range as determined by the minval: and maxval: schema slots.

Object classes whose admissible data values are of type STRING include values that must comply with the form: definition found in the schema specification of the data value class. For example, prior to translating

```
(REFLECT (HAS-TITLE (agent P-1)
                    (object "CS-430"))),
```

the data value "CS-430" must be determined to be a valid representative of the COURSE-NAME object class. The corresponding data value class is COURSE-NAME-V, which is of type STRING and whose form: slot is

```
(& [ "A" - "Z" ] $ 2 "-" [ "1" - "5" ] & [ "0" - "9" ] $ 2).
```

This form: definition means that a valid representative must consist of two alphabetic characters, followed by a "-", followed by one digit whose value is between "1" and "5" inclusive, followed by two digits whose values are between "0" and "9" inclusive. Refer to Appendix G for a complete description of this syntax.

4.3.2 Explicit Pre-conditions

*assert operations, which may result in adding data values to virtual or to stored extensions, must meet the two types of explicitly defined, schema pre-conditions. These are necessary and required pre-conditions. A third schema specification category, TOKEN object types, has the effect of providing implied pre-conditions for an *assert operation. TOKEN types will be discussed in the next section.

Any tuple whose values will be used to update the stored database must meet the pre-conditions specified in every schema necessary: slot of the PRIMITIVE and non-PRIMITIVE situations to which data values will be added. This is equivalent to saying that the tuple must meet the conjunction of all of the necessary pre-conditions

```
(and (<pre-condition 1>)
     (<pre-condition 2>)
```

```
                    :
             (<pre-condition n>))
```

of the situations involved in the update. This conjunct is part of the *enquire expression of an update operation.

If the update operation involves a disjunctive expression, the user will choose a branch for each tuple. Recall that a binding tuple is a set of data values that taken together hold in the current database and that make the expression a true statement. Each of the branches may have different necessary pre-conditions. Therefore, different tuples computed in the same operation may have different sets of pre-conditions. That is, a disjunctive update does not have a global set of pre-conditions that are applicable to all tuples. A tuple needs only to meet those pre-conditions that are related to the branch that is chosen for it. For example, the operation in Figure 16 has two necessary pre-conditions. IS-NOON-MEETING has the pre-condition

```
(HAS-TITLE (agent x)).
```

Any tuple chosen to be *asserted into the IS-NOON-MEETING situation will need to meet this pre-condition. IS-SEMINAR has the necessary pre-condition

```
(IS-COURSE (agent x)).
```

Any tuple chosen to be *asserted into the IS-SEMINAR situation will need to meet this pre-condition.

In addition to necessary pre-conditions, update operations must meet any required pre-conditions as specified by required: slots in schema entries for all PRIMITIVE and non-PRIMITIVE situations to be *asserted. The set of required: pre-conditions for each of the branches in a simplified expression is derived in the same manner as are its necessary pre-conditions with the exception that the required: schema slots rather than the necessary: slots are consulted.

For REFLECT and REFLECT-NOT the effect of required pre-conditions on operations is the same as the effect of necessary pre-conditions on these operations. Therefore, when dealing with REFLECT and REFLECT-NOT operations, the simplification algorithm does not distinguish between these two types of pre-conditions. They both become part of the *enquire expression. For any tuple being considered for a REFLECT or REFLECT-NOT operation the *enquire expression equivalent to

```
(and (and <set of all necessary pre-conditions>)
     (and <set of all required pre-conditions>)).
```

This expression, by rules of associativity of the set operations that implement this combination, is equivalent to

```
(and (<necessary pre-condition 1>)
     (<necessary pre-condition 2>)
                   :
     (<necessary pre-condition n>)
     (<required pre-condition 1>)
     (<required pre-condition 2>)
                   :
     (<required pre-condition n>))
```

So, for REFLECT and REFLECT-NOT operations, the simplification algorithm makes an *enquire expression consisting of one list of all necessary and required pre-conditions for each branch in the simplified *assert expression. For the operation in Figure 16, the necessary and required pre-conditions for each branch are merged to form the *enquire expression for each branch. These *enquire expressions are shown in Figure 31.

For ASSERT and DENY the meaning of required pre-conditions is different than the meaning of necessary pre-conditions. For these two operators, a required pre-condition is also one that must hold in the database extension prior to the successful addition of values to an extension. The difference, however, is that if the pre-condition does not initially hold in the database extension, the operation implies that an attempt should be made to make the pre-condition hold. This is the same as saying that the database system will attempt to perform sub-operations that will be semantically equivalent to the following REFLECT operation

```
(REFLECT (and (<required pre-condition 1>)
              (<required pre-condition 2>)
                            :
              (<required pre-condition n>))).
```

After the data-level equivalent of this operation is performed, any tuples now meeting the required pre-conditions will still be considered for the update operation. The intent of the stronger ASSERT operator is to supply automatically for the user any policy requirements that have been forgotten. How this required pre-condition sub-operation is merged with the main *assert operation to form one *assert expression will be discussed in the next chapter.

4.3.3 Preserving Referential Integrity

Object classes whose allowable data values are of type TOKEN have a schema specification slot referred to as their definition: slot. The value of this slot is a situation name called the defining situation. In order for a TOKEN value to designate an object that is a member of the given object class, the TOKEN value must be an agent participant in the defining situation. This concept of implicit pre-conditions is known as referential integrity (Date, 1983.) During an *assert operation, all TOKEN values that are being added to extensions of PRIMITIVE situations must also be added to the corresponding object class defining situations if they are not already there. For example, part of the operation shown in Figure 16 is

```
(REFLECT (IS-EVENT-NAME (agent x)
                        (object V-10 "AI-TOPICS"))).
```

The situation in this sub-operation has an agent of type EVENT. The schema definition of EVENT

```
object-class EVENT
   representative: (TOKEN)
   definition: (IS-EVENT)
   names: (HAS-TITLE)
```

indicates that its defining situation is IS-EVENT. This implies that

```
(REFLECT (IS-EVENT (agent x)))
```

will also be performed. These two operations can be combined to form the expanded operation

```
(REFLECT (and
              (IS-EVENT-NAME (agent)
                             (object V-10 "AI-TOPICS"))
              (IS-EVENT (agent)))).
```

It is important to note that if the

```
(REFLECT (IS-EVENT (agent x)))
```

update fails for any tuples, then these tuples are removed from the REFLECT extension and are not added to the other situation extensions mentioned in the *assert expression. Because this addition to a defining situation is, itself, defined semantically as a REFLECT operation, the same policies and procedures govern this sub-operation. For example, the defining situation

1) might not be a PRIMITIVE situation,

2) might have necessary:, required:, and cardinality: slots, and

3) might involve disjuncts that must be resolved by the user in the same manner as other disjuncts encountered in the operation.

Therefore, it is observed that the REFLECT sub-operation on the new object's defining situation is simply an additional, but implied, portion of the user's original operation. As discussed in Chapter V, this sub-operation will be merged with the main *assert operation.

4.3.4 Interpreting Data Access Operations

A SIDUR ENQUIRE operation is defined simply as a primitive *enquire operation on the *enquire expression of the canonical form. Refer to Section 4.2.3 for a description of how an *enquire expression is translated into a BAGAL procedure. Figure 28 shows a BAGAL example of an ENQUIRE operation.

CHECK is handled by translating the situation expression as it would be for the ENQUIRE operator with the exception that, to enhance performance, there is a limit of one tuple placed on the BAG that computes the final extension. This means that the data-level may stop processing after one tuple qualifies to be in the extension. Limits, in the general case, cannot be placed on earlier BAGs because their entire extensions may be needed to compute extensions involving set subtraction and union. However, the existence of only one tuple in the final OR-MERGE extension is sufficient to determine the result of the CHECK operation. Then, the CHECK algorithm produces statements to generate a BAG with the single TOKEN

```
(CHECK
    (and
        (not (IS-COURSE (agent V-1 T-1)))
        (OFFERING-OF (agent V-1 T-1)
                     (object V-2 T-2))))
```

```
                                  < ACCESS BAG for BRANCH 1 of disjunct >

((BAG b-1 (V-1 V-2)) <-
 (ACCESS
     (V-1 = T-1)
     (OFFERING-OF / agent I-1 V-1)
     (V-2 = T-2)
     (OFFERING-OF / object I-1 V-2)))


                          < ACCESS BAG for 'not' situations of BRANCH 1  >

((BAG b-2 (V-1)) <-
  (ACCESS
    (V-1 = T-1)
    (IS-COURSE / agent I-2 V-1)))


                                  < BAG to subtract the 'not' extension >

((BAG B-3 (V-2 V-1)) <-
 (MINUS (BAG b-1 (V-1 V-2)) (BAG b-2 (V-1))))


                                  < BAG to limit extension to 1 tuple and
                                    return BAG condition FULL or EMPTY >

(IF (FULL (BAG B-3 (V-2 V-1) 1)) THEN
    ((BAG B-4 (nil)) <- (FULL))
 ELSE
    ((BAG B-4 (nil)) <- (EMPTY)))


                                  < BAG to RETURN the requested result >
(RETURN (BAG B-4 (nil)))
```

CHECK Data-level Query
Figure 30

FULL representing the SIDUR 'full' extension if there is one tuple in the extension or the TOKEN EMPTY representing the SIDUR 'empty' extension if there are zero tuples. An example is shown in Figure 30. The entire extension of both the OFFERING-OF and the IS-COURSE expressions need to be retrieved in order that they be available for the set subtraction operation.

ENQUIRE and CHECK operations may involve computations. These operations are translated such that all of the needed BAGAL query retrievals are done first in order that an extension of arguments to the computations can be formed. Computations are done by passing the BAG that holds the extension of arguments to the first BAGAL computation operation. As results are computed, they are added to the extension of arguments, and the BAG holding the arguments and results is passed to the next BAGAL computation. The order in which computations are carried out is determined by the *enquire routine, which schedules first those computations whose results are needed as arguments for other computations. Computations will be discussed further in Chapter V.

4.3.5 Interpreting Update Operations

As mentioned earlier, update operations are re-expressed as an *assert expression and an *enquire expression. In summary, the *assert expression represents the extension that may be updated in the stored database. Any tuple that does not meet the pre-condition tests of the *enquire expression is removed from the *assert extension and is not updated in the stored database. Figure 31 shows an example of a REFLECT operation and its corresponding *assert and *enquire expressions. Appendix I shows the schema definitions that were used to derive the *assert and *enquire expressions in Figure 31. Figure 55 shows the canonical form of the operation.

The *assert and *enquire expressions from Figure 31 are translated into the BAGAL procedure that is shown in Figure 32. This procedure will be discussed in more detail later. In general, this procedure performs tuple-at-a-time processing. An

ORIGINAL OPERATION

```
(REFLECT
  (and (IS-EVENT-NAME (agent x) (object V-10 "AI-TOPICS"))
       (or (IS-NOON-MEETING (agent x))
           (IS-SEMINAR (agent x)))))
```

OPERATION EXPRESSED AS A DISJUNCT OF CONJUNCTS

```
(REFLECT (or                                                BRANCH 1
          (and (IS-EVENT-NAME (agent x) (object V-10 "AI-TOPICS"))
               (IS-NOON-MEETING (agent x)))
                                                            BRANCH 2
          (and (IS-EVENT-NAME (agent x) (object V-10 "AI-TOPICS"))
               (IS-SEMINAR (agent x)))))
```

PRIMITIVE SEMANTIC LEVEL *enquire AND *assert EXPRESSIONS

```
                                      <Refer to Appendix I and Figure 55 >
(*enquire                                                   BRANCH 1
   (or (and (HAS-TITLE (agent x)                 necessary:
              (object V-11 "MEETING"))             pre-condition
            (MEETING-TIME (agent x)              required:
                   (object V-12 "12AM"))           pre-condition
                                                            BRANCH 2
       (and (IS-COURSE (agent x)))               necessary: pre-cond'n
            (HAS-TITLE (agent x)                 required:
                   (object V-13 "SE-400")))        pre-condition

(*assert
   (or (and (IS-EVENT-NAME (agent x)                        BRANCH 1
             (object V-10 "AI-TOPICS"))          PRIMITIVE definition:
            (IS-NOON-MEETING (agent x))          PRIMITIVE definition:
            (IS-EVENT (agent x))                 TOKEN definition:
            (not (NOT-IS-NOON-MEETING            negative extension
                   (agent x))))                    of open-world
                                                   situation
        (and (IS-EVENT-NAME (agent x)                       BRANCH 2
               (object V-10 "AI-TOPICS"))        PRIMITIVE definition:
             (OFFERING-OF (object x)))           PRIMITIVE definition:
             (LIMIT (agent x) (value V-14 12)) sigma-bound defn.:
             (IS-EVENT (agent x)))))             TOKEN definition:
```

Update Operation and Primitive Expressions
Figure 31

```
(REFLECT (and (IS-EVENT-NAME (agent x-10)
                             (object V-10 "AI-TOPICS"))
              (or (IS-NOON-MEETING (agent x-10))
                  (IS-SEMINAR (agent x-10))))
```

```
((BAG B-1 (V-10 V-14)) <-  (TUPLES    < TUPLES BAG for constants >
  ((V-10 <- "AI-TOPICS) (V-14 <- 12))))

                                          < BAG to get users CHOICE >
((BAG B-2 (V-10 c-10)) <- (FOR (BAG B-1 (V-10))
 ((c-10 <- (CHOICE "choose one" (IS-NOON-MEETING IS-SEMINAR)
                                 (V-10)))
  (CASE (((c-10 = IS-NOON-MEETING) ((c-10 <- 1)))
         ((c-10 = IS-SEMINAR) ((c-10 <- 2))))))))

   < split reflect extension based on user CHOICE for each tuple >
((BAG B-3 (V-10)) <-                                  < BRANCH 1 >
 (FOR (BAG B-2 (V-10 c-10)) (IF (c-10 <> 1) THEN nil)))

((BAG B-4 (V-10)) <-                                  < BRANCH 2 >
 (FOR (BAG B-2 (V-10 c-10)) (IF (c-10 <> 2) THEN nil)))

                                 < *enquire expression BRANCH 1 >
((BAG b-5 (x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL)) <-
  (ACCESS (V-11 = "MEETING")    (V-12 = "12AM")
   (HAS-TITLE / agent I-1 x-10)  (HAS-TITLE / object I-1 V-11)
   (MEETING-TIME/agent I-2 x-10) (MEETING-TIME/object I-2 V-12)))

                                 < *enquire expression BRANCH 2 >
((BAG b-6 (x-10 V-13 I-3 NO-NULL I-4 NO-NULL)) <- (ACCESS
   (V-13 = "SE-430")       (IS-COURSE / agent I-3 x-10)
   (HAS-TITLE / agent I-4 x-10) (HAS-TITLE / object I-4 V-13)))

  < merge constants and *enquire extensions to get update tuples >
                                                      < BRANCH 1 >
((BAG B-7 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL)) <-
 (AND-MERGE ((BAG B-1 (V-10 V-14))
             (BAG b-5 (x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL)))))

                                                      < BRANCH 2 >
((BAG B-8 (V-10 V-14 x-10 V-13 I-3 NO-NULL I-4 NO-NULL)) <-
 (AND-MERGE ((BAG B-1 (V-10 V-14))
             (BAG b-6 (x-10 V-13 I-3 NO-NULL I-4 NO-NULL)))))

      < BAGs for each branch to do cardinality check, precheck to
        see if already stored, and update operations  BRANCH 1 >
((BAG B-9 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL)) <-
 (FOR (BAG B-7 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL))
  ((IF (x-10 = NULL) then (x-10 <- (OCREATE EVENT)))
   (I-5 <- (CINDEX (IS-EVENT-NAME (agent x-10) (object V-10)))
    (IF (I-5 = null) THEN                               < precheck >
        ((I-5 <- (ICREATE IS-EVENT-NAME))
         (CHANGE IS-EVENT-NAME / agent I-5 x-10)          < update >
         (CHANGE IS-EVENT-NAME / object I-5 V-10)))
    (I-6 <- (CINDEX (IS-NOON-MEETING (agent x-10))))
    (IF (I-6 = null) THEN                               < precheck >
        ((I-6 <- (ICREATE IS-NOON-MEETING))
         (CHANGE IS-NOON-MEETING / agent I-6 x-10)))      < update >
    (I-7 <- (CINDEX (IS-EVENT (agent x-10))))
    (IF (I-7 = null) THEN                               < precheck >
        ((I-7 <- (ICREATE IS-EVENT))
         (CHANGE IS-EVENT / agent I-7 x-10)))             < update >
    (I-8 <- (CINDEX (NOT-IS-NOON-MEETING (agent x-10))))
    (IF (I-8 <> null) THEN                              < precheck >
        ((DELETE NOT-IS-NOON-MEETING / agent I-8 x-10))))))
                                                        < BRANCH 2 >
((BAG B-10 (V-10 V-14 x-10 V-13 I-3 NO-NULL I-4 NO-NULL)) <-
 (FOR (BAG B-8 (V-10 V-14 x-10 V-13 I-3 NO-NULL I-4 NO-NULL))
  ((IF (x-10 = NULL) then (x-10 <- (OCREATE EVENT)))
   (I-9 <- (CINDEX (LIMIT (agent x-10))))            < cardinality >
   (I-10 <- (CINDEX (LIMIT (agent x-10) (value V-14))))
   (c-11 <- (COUNT I-9 I-9))                         < cardinality >
   (IF (OR (c-11 < 1) (I-10 <> NULL)) THEN
     ((I-11 <- (CINDEX (IS-EVENT-NAME (agent x-10)(object v-10))))
      (IF (I-11 = null) THEN                            < precheck >
        ((I-11 <- (ICREATE IS-EVENT-NAME))
         (CHANGE IS-EVENT-NAME / agent I-11 x-10)         < update >
         (CHANGE IS-EVENT-NAME / object I-11 V-10)))
      (I-12 <- (CINDEX (OFFERING-OF (agent x-10))))
      (IF (I-12 = null) THEN                            < precheck >
        ((I-12 <- (ICREATE OFFERING-OF))
         (CHANGE OFFERING-OF / object I-12 x-10)))        < update >
      (IF (I-10 = null) THEN                            < precheck >
       ((I-10 <- (ICREATE LIMIT))
        (CHANGE LIMIT / agent I-10 x-10)                  < update >
        (CHANGE LIMIT / value I-10 V-14)))
      (I-13 <- (CINDEX (IS-EVENT (agent x-10))))
      (IF (I-13 = null) THEN                            < precheck >
       ((I-13 <- (ICREATE IS-EVENT))
        (CHANGE IS-EVENT / agent I-13 x-10))))))))        < update >

             <Merge result from each branch to get final result >
((BAG B-11 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL V-13
           I-3 NO-NULL I-4 NO-NULL)) <-
 (OR-MERGE
   (BAG B-9 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL))
   (BAG B-10 (V-10 V-14 x-10 V-13 I-3 NO-NULL I-4 NO-NULL))))

(RETURN (BAG B-11 (V-10 x-10)))                     < RETURN results >
```

BAGAL Update Operation
Figure 32

extension of tuples that are candidates for the update operation is formed from the constant data values in the *assert expression. The CHOICE BAG causes the user to be prompted to chose a branch for each tuple. (In this example, there is only one tuple in the extension. Compound operations will be discussed in Section 4.3.8.) Depending on the user's choice, the tuple will be put into either the extension for branch 1 or for branch 2. The *enquire expression for each branch is translated into an ACCESS BAG that will only retrieve tuples whose necessary and required pre-conditions are met. The two *enquire extensions are then each intersected with the constant tuples extension to eliminate ineligible tuples and form the *assert extension for each branch. These extensions are stored in BAGs called the update BAGs. Each update BAG is processed by the update statements for its branch. As an update BAG is processed, any tuple that fails a cardinality constraint is eliminated from the extension. Any tuple that is already stored is not updated but remains in the update extension. Each remaining tuple that is not already stored is updated into each of the situations from the *assert expression. The union of the update extensions from each branch is returned as the result of the operation.

REFLECT, REFLECT-NOT, ASSERT, and DENY are the four situation operators that can result in changes being made to the stored database extension. Each of these four update operators can cause data values to be added as well as to be deleted from the stored database. It is apparent from the general descriptions given the chapter titled "THE SIDUR MODEL" that REFLECT and ASSERT can result in adding values to the database and that REFLECT-NOT and DENY can cause deletions. However, if the expression on which the update operator will act contains the 'not' connective, then the "polarity" of the update operation with respect to the sub-expression that is within the scope of the 'not' will reverse. For example, both

```
(REFLECT     (IS-COURSE (agent C-1)))
(REFLECT-NOT (not (IS-COURSE (agent C-1))))
```

indicate that an addition of a tuple to the IS-COURSE situation is being considered. REFLECT-NOT becomes REFLECT with respect to the second IS-COURSE expression because it contains the 'not' connective. The following two operations

```
(REFLECT-NOT (IS-COURSE (agent C-2)))
(REFLECT     (not (IS-COURSE (agent C-2))))
```

both indicate the removal of a tuple from the extension of IS-COURSE. Again, REFLECT becomes REFLECT-NOT with respect to the second IS-COURSE expression.

SIDUR defines each of the four situation update operators in terms of the results of applying a primitive *assert operator to the extension of tuples that is determined to be eligible for the operation. The *assert operator adds or removes each eligible tuple from the situation extension according to the following criteria:

1) Any atomic situation description nested directly within one of these three elements: REFLECT-NOT, DENY, or the 'not' connective is a negated situation.

2) A 'not' connective that is nested within an operation changes the operator that is currently in effect for that portion of the expression. If an "add" (REFLECT or ASSERT) operator is in effect, it is changed to "remove" (REFLECT-NOT or DENY, respectively.) If a "remove" operator is in effect, it is changed to "add". When "remove" is in effect, any PRIMITIVE situation expressions nested directly within it are considered negated situations.

3) Affirmed closed-world situations indicate addition of values.

4) Negated situations with a closed-world extension indicate removal of values.

5) Negated open-world situations are handled by low level translation routines as an addition into the negative extension and a removal from the affirmative situation.

6) Addition to affirmed open-world situations is handled as an addition to the affirmative situation and a removal from the negative extension.

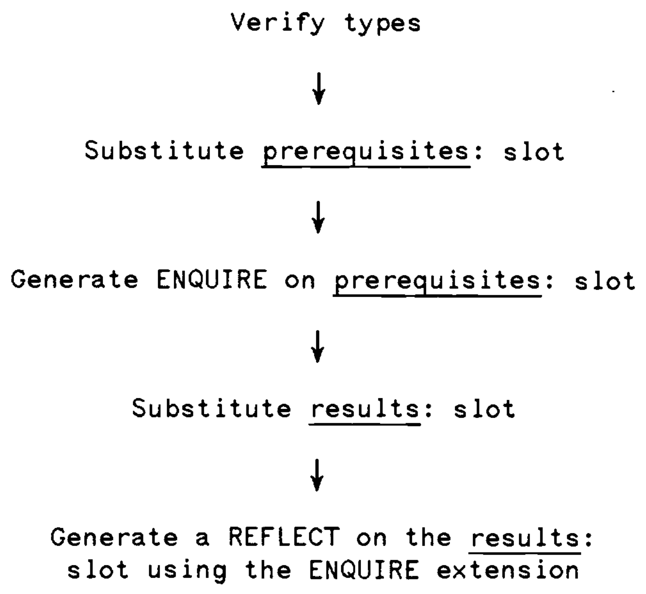
After statements are translated that will produce the extension of tuples that meet their pre-conditions for the update operation, the primitive *assert operation translation routines emit statements to perform the update. A flowchart of the BAGAL processing was shown in Figure 29. These are the BAGAL operations needed to perform the update of the *assert extension:

1) check cardinality constraints,
2) precheck to insure that values to be added are not already stored,
3) precheck to insure that values to be deleted are stored,
4) create object TOKENs and instance TOKENs,
5) perform the additions and deletions of tuples, and
6) return the resulting extension of the operation.

These statements were illustrated in Figure 32 and will be further explained in Chapter V.

4.3.6 Interpreting Action Operations

PERFORM is handled by verifying the object classes of the participants and then calling the ENQUIRE routine on the sigma-bound prerequisites: slot from the action's schema specification. The resulting extension of data values is used as the input extension to REFLECT the sigma-bound results: slot from the action's schema specification. A flowchart for this operation is shown in Figure 33. Refer to Section 4.3.8 for a discussion of how updates that involve an entire extension are handled.


Verify types
Substitute prerequisites: slot
Generate ENQUIRE on prerequisites: slot
Substitute results: slot
Generate a REFLECT on the results: slot using the ENQUIRE extension

PERFORM Operation Flow Chart
Figure 33

PERMIT? is handled by verifying the object classes of the participants and then calling the CHECK routines with the sigma-bound prerequisites: slot from the action's schema specification as the argument. An example is shown in Figure 34.

PERMIT! is handled by verifying the object classes of the participants and then calling the ASSERT routines with the sigma-bound prerequisites: slot from the action's schema specification.

SCHEMA DEFINITION OF ENROLLS-IN

```
action ENROLLS-IN
  participants: ((agent x STUDENT) (object y OFFERING))
  prerequisites:
    (and
         (OFFERING-OF (agent z) (object y))
         (MAY-TAKE (agent x) (object z)))
  results:  (TAKES-OFFERING (agent x) (object y))
```

```
(PERMIT? (ENROLLS-IN (agent V-23 S-1001)
                     (object V-24 O-1002)))
```

```
                              < ACCESS BAG to compute extension >

((BAG b-1 (V-23 V-24 z-11)) <-
 (ACCESS
     (V-23 = S-1001)
     (V-24 = O-1002)
     (OFFERING-OF / agent I-1 z-11)
     (OFFERING-OF / object I-1 V-24)
     (MAY-TAKE / agent I-2 V-23)
     (MAY-TAKE / object I-2 z-11)))

                         < BAG to determine FULL or EMPTY condition
                           of the resulting extension >

(IF (FULL (BAG b-1 (V-23 V-24 z-11) 1)) THEN
    ((BAG B-2 (nil)) <- (FULL))
 ELSE
    ((BAG B-2 (nil)) <- (EMPTY)))

                                      < BAG to RETURN result >

(RETURN (BAG B-2 (nil)))
```

PERMIT? Data-level Query
Figure 34

4.3.7 Interpreting Object Operations

CREATE is defined to translate to a data-level statement that will generate a new object TOKEN whose first character will be the first character in the name of the specified TOKEN's object class.

DESTROY is defined to translate to a data-level language statement that will remove the object TOKEN from use in the stored database. All tuples in which it occurs will also be removed. An example of a BAGAL DESTROY operation is shown in Figure 38.

4.3.8 Interpreting Compound Operations

FOR is defined by calling the ENQUIRE routine on the domain sigma expression in the first portion of the operation. The resulting ENQUIRE extension may contain more than one tuple. Statements to take a projection of the resulting extension onto the variables in the FOR variable list are translated. This resulting extension is passed as the input to each of the simple operations in the second portion of the FOR operation.

SINCE is defined in the same manner as FOR with the exception that the REFLECT routines are used instead of the ENQUIRE routines to produce the resulting extension of the domain extension of the FOR operation.

It has been a goal of the OSIRIS design to allow update operations to involve an extension of tuples of data values as well as single tuples. As currently defined, an update operation that is part of a compound operation can result in an extension of tuples to be updated. Note that a PERFORM operation can also result in an extension to be updated.

```
(For (x y)
   (and (HAS-NAME (agent x)
                  (object V-9 "MICHAEL-FREILING"))
        (CAN-TEACH (agent x) (object y)))
   (REFLECT
        (TEACHES-COURSE (agent x)(object y))))
```

Compound Operation Resulting In an Extension
Figure 35

x	V-9	y
P-6	MICHAEL-FREILING	C-20
P-6	MICHAEL-FREILING	C-21
P-6	MICHAEL-FREILING	C-28

Extension of Tuples
Figure 36

x	y
P-6	C-20
P-6	C-21
P-6	C-28

Projected Extension of Tuples
Figure 37

The situation operators therefore all need to be able to handle an extension of tuples as argument. The need to handle an extension of tuples has especially influenced how updates on disjunctive situation expressions are handled, how atomicity of transactions at the tuple level is provided, and how pre-conditions are dealt with. In order to simplify both the translation algorithms and the BAGAL queries, operations involving single tuple instances are handled as an extension of one tuple. The simplification algorithm has been especially designed to enable the translation routines to produce BAGAL queries that can handle an extension of single or multiple tuples. Note that because ENQUIRE and CHECK may also occur (although more infrequently) in the second portion of a compound expression, they are also implemented to accept an initial extension of values. This is done by intersecting their input extension with the result of the ENQUIRE or CHECK to produce the final result.

For example, the operation in Figure 35 may result in the extension in Figure 36 after computing:

```
(ENQUIRE
    (and
        (HAS-NAME (agent x)
                  (object V-9 "MICHAEL-FREILING"))
        (CAN-TEACH (agent x) (object y)))).
```

The projection, which is indicated by '(x y),' results in the extension in Figure 37 to be considered by the REFLECT operation. Figure 38 shows an example of a BAGAL compound operation. In this operation all occurrences of the object TOKEN whose corresponding name in the HAS-NAME situation is "SALLY-BROWN" will be removed from the database.

```
(FOR
    ((x-30)
     (HAS-NAME (agent x-30)
               (value V-30 "SALLY-BROWN"))
     (DESTROY x-30)))
```

```
                                      < TUPLES BAG for constants >
((BAG B-1 (V-30))
 <-
 (TUPLES ((V-30 <- "SALLY-BROWN"))))

              < data accesses for *enquire on domain expression >
((BAG b-2 (x-30 V-30)) <-
 (ACCESS
     (V-30 = "SALLY-BROWN")
     (HAS-NAME / agent I-1 x-30)
     (HAS-NAME / value I-1 V-30)))

                                     < produce domain expression >
((BAG B-3 (x-30 V-30)) <-
 (AND-MERGE ((BAG B-1 (V-30))
             (BAG b-2 (V-30 x-30)))))

                < discontinue processing if no eligible tuples >
(IF (EMPTY (BAG B-3 (x-30 V-30))) THEN
    (FAIL "NO EXTENSION PRODUCED"))

                                            < object operation >
((BAG B-4 (V-30 x-30 d-30)) <-
 (FOR (BAG B-3 (V-30 x-30))
      (d-30 <- (DESTROY x-30))))

                                               < RETURN result >

(RETURN (BAG B-4 (V-30 x-30 d-30)))
```

Data-level Compound Operation
Figure 38

V. IMPLEMENTATION OF THE TRANSLATION PROCESS

5.1 Goals

This implementation of SIDUR is part of the first phase of the prototype implementation of the OSIRIS architecture. As part of the prototype, the SIDUR implementation has two primary goals. The major goal is to provide the function of semantic data modeling. The second goal is to provide this implementation within a context that can evolve with the OSIRIS project as it addresses additional theoretical and implementation issues related to semantic information systems. With these goals in mind, the SIDUR implementation provides most of the function of the semantic level as defined in the "SIDUR Manual" (Freiling, 1983c) as well as several additional features that are likely to be useful at this level. Although designed to meet its specified interface requirements in the OSIRIS architecture, SIDUR is also able to run as an independent module by providing its own simple user interface and schema management routines and by relying on the Franz Lisp run-time system for such routines as input/output, number conversion, file and memory management, and standard functions. When running independently, data-level queries can be produced but not carried out, and schema operations can be entirely carried out.

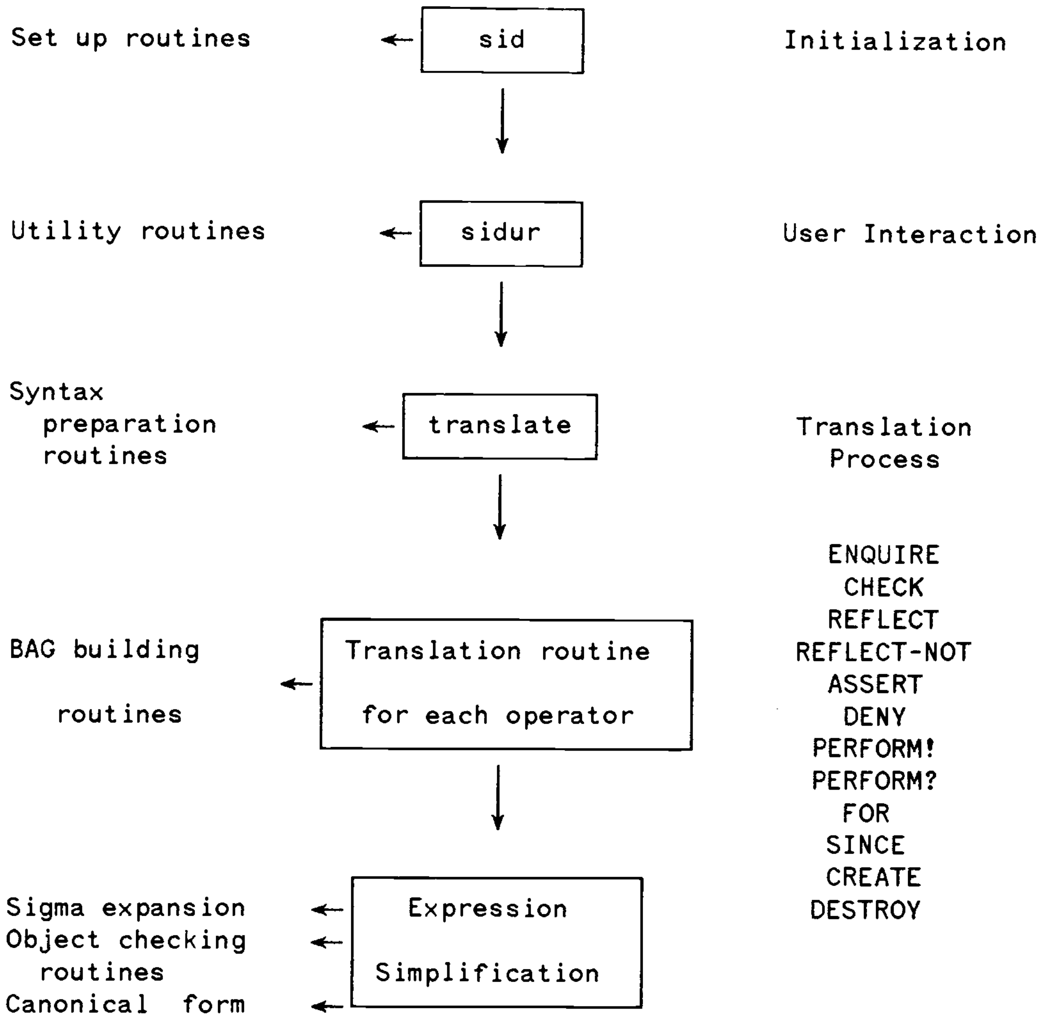
5.2 General Strategy

The interface of the SIDUR implementation prompts the user to provide one operation at a time. Each operation is treated as a single unit and is translated in its entirety into one data-level query. The general strategy used in processing an operation is as follows: A complete check of the operation for syntactic correctness is done. Then the operator being used is determined and a corresponding routine is called. As needed, the operation routines are called recursively by each other. In preparation for the translation process, situation expressions are simplified into an equivalent canonical form. Non-TOKEN constant data value participants are type-checked. Non-PRIMITIVE situations are expressed in terms of their PRIMITIVE definitions. All relevant schema information is categorized according to the effect that the information will eventually have on the operation. If the operation involves schema access or manipulation, it is carried out totally at the SIDUR level. Otherwise, the operation, which involves only data access and/or manipulation, is translated into a BAGAL query for processing by the data-level processor of the OSIRIS architecture.

5.3 Main Control Routines

The SIDUR implementation is controlled by three routines, 'sid', 'sidur', and 'translate' as shown in Figure 39. The first routine, called 'sid', is the initial entry to the SIDUR module. 'sid' prompts the user to type in the database schema and result file names. 'sid' then calls routines to read the schema into the Franz Lisp environment. If a new schema is being established, the schema initialization routines, as discussed in the chapter titled "SCHEMA BUILDING," are called.

After the schema has been set up, the function called 'sidur' handles the subsequent interaction with the user. 'sidur' is composed of a read-translate-break loop that prompts the user to type in an operation, calls a routine named 'translate' to control the translation process, and then invokes the break package feature of Franz Lisp to enable the user to view or to save the results. In addition to operation translation, 'sidur' allows the user to exit to the Franz Lisp level or to call several SIDUR utility functions, which are described later in the appendix titled "Use of the SIDUR Implementation." If an operation fails during the translation process or if an operation involves a schema construct, control will return directly to 'sidur' without returning results to intermediate functions.



SIDUR Flowchart
Figure 39

'translate' is the main translation control routine and is called by 'sidur' with the user's operation as its argument. The operation that the user types in may consist of any of the following:

1) a symbol previously set equal to the Franz Lisp list of symbols that comprise the operation,

2) a Franz Lisp list consisting of two elements, the first being an operator name and the second being a symbol previously set equal to the list of symbols that comprise the remainder of the operation, or

3) the operation typed out entirely.

'translate' converts the first two forms of input into the third form and calls the syntax checking routines. Further processing will not occur if the operation is syntactically ill-formed. Next, two routines are called to prepare the situation expressions by creating unique variables and substituting them into the expressions in place of the user's variables and in front of each constant. This is done in order to avoid variable name conflicts. 'translate' then calls a routine corresponding to the specific operator in the operation. These operation translation routines are discussed in the next section.

5.4 Translation Routines

Translation routines corresponding to each of the SIDUR data manipulation operators provide the core of the SIDUR implementation. These routines first determine which translation actions should be taken and then call other routines that create a data-level query that correctly represents the SIDUR operation. Two of the situation operators, ENQUIRE and REFLECT, are the basis on which the other situation operators--CHECK, ASSERT, DENY, AND REFLECT-NOT--are implemented. ENQUIRE and REFLECT are implemented in terms of the primitive data access operator called *enquire* and the primitive data update operator called *assert*. Furthermore, the action and compound operators are implemented in terms of these six situation operators.

The situation operation algorithms all call the simplification algorithm routines to produce the list structure on which they will operate. This list structure, called the canonical form, contains as sub-lists all the expressions that will be needed to translate the operation. The simplification algorithm routines determine the exact function of each element of the canonical form by where they place each element in the form.

enquire is the primitive access operation. Its translation routine calls the simplification algorithm to produce the *enquire* expression on which it operates. The *enquire* expression, whose form is shown in Figure 40, is a situation or computation expression that specifies the retrieval or computation of all tuples of data values that satisfy the expression. Expressions that involve schema access are used by SIDUR routines to search the schem. data structure, and the result is returned directly to the user. For each branch of a non-schema *enquire expression, situations that are not preceded by the 'not' connective or that have an open-world extension are translated into a single ACCESS BAG assignment statement. Closed-world situations that are preceded by 'not' are put into another ACCESS BAG whose resulting extension is subtracted from the extension of the first ACCESS BAG. A BAG is translated to OR-MERGE the resulting extension obtained from each branch. If there are any computation expressions, the extension is COLLECTed (i.e. bound) to a variable that can later be passed as argument to the computations. If any of the computation participants are situation expressions needing data accesses, BAGs are translated to perform the accesses. These computation arguments are then OR-MERGEd with the extension resulting from the situation expressions in the *enquire expression. Since the planned query optimizer for the OSIRIS architecture works on single ACCESS BAG retrievals, an effort has been made in the SIDUR implementation, after all semantic considerations have been correctly provided for, to produce large, rather than small, ACCESS BAGs. An example of an ENQUIRE data-level access procedure is shown in Figure 41. Its canonical form is discussed later and is shown in Figure 54.

After data access BAGs are translated, the *enquire algorithm determines the order in which computation expressions will be carried out by repeatedly examining each computation expression. A computation expression is translated if its arguments are not taken from the results of other computations or if all computations whose results are needed as arguments for this expression have already been translated. At the data level, computations are carried out in the order in which they occur in the query unless it can be shown that changing the order will not affect the final result. After all of the computation expressions have been translated, data-level statements are translated to do an AND-MERGE of the resulting computation extension with each extension previously obtained by the ACCESS BAG retrievals done for each branch. A final OR-MERGE of the resulting extensions from each branch is then translated to produce the correct final extension of retrieved and computed values. A flowchart of this algorithm is shown in Figure 42. An example of an ENQUIRE computation query is shown in Figure 43.

```
(*enquire
  (or                                          BRANCH 1
    (and (<PRIMITIVE situation expression 1>)
         (<PRIMITIVE situation expression 2>)
                    :                        :
         (<PRIMITIVE situation expression n>)
         (<SYSTEM computation 1>)
         (<SYSTEM computation 2>)
                    :          :
         (<SYSTEM computation n>))

                                               BRANCH 2
    (and (<PRIMITIVE situation expression 1>)
         (<PRIMITIVE situation expression 2>)
                    :                        :
         (<PRIMITIVE situation expression n>)
         (<SYSTEM computation 1>)
         (<SYSTEM computation 2>)
                    :          :
         (<SYSTEM computation n>))))
    . . .
```

***enquire Expression**
Figure 40

*assert is the primitive update operation and controls adding data values to, and deleting data values from, a PRIMITIVE situation extension such as that shown in Figure 44. Each REFLECT, REFLECT-NOT, ASSERT, and DENY operation is translated into a primitive *assert operation. Any situation in an *assert expression that is preceded by 'not' indicates a data value deletion. The other situations involve data value additions. Open-world extensions are also handled by the primitive update operation, which translates statements in the data-level query to add or delete values as needed to complete the open-world update. The primitive *assert expression for each branch is translated into one BAGAL update routine.

```
(ENQUIRE
    (or
        (IS-FACULTY (agent V-1 S-1))
        (and
            (or
                (IS-STUDENT (agent V-1 S-1))
                (IS-STAFF-STUDENT (agent V-1 S-1)))
            (FINAL-GPA (agent V-1 S-1)))))

< See Figure 54 for the canonical form >
```

```
                                < ACCESS BAG for BRANCH 1 of disjunct >

((BAG b-1 (V-1)) <- (ACCESS
   (V-1 = S-1) (IS-FACULTY / agent I-1 V-1)))

                                < ACCESS BAG for BRANCH 2 of disjunct >

((BAG b-2 (V-1 y-1 z-1)) <- (ACCESS
   (V-1 = S-1)
   (IS-PERSON / agent I-2 V-1)             < IS-STUDENT expansion >
   (TAKES-OFFERING / agent I-3 V-1)        <     "          "     >
   (TAKES-OFFERING / object I-3 z-1)       <     "          "     >
   (OFFERING-OF / agent I-4 y-1)           <     "          "     >
   (OFFERING-OF / object I-4 z-1)          <     "          "     >
   (FINAL-GPA / agent I-5 V-1)))

                                < ACCESS BAG for BRANCH 3 of disjunct >

((BAG b-3 (V-1)) <- (ACCESS
     (V-1 = S-1) (IS-STAFF-STUDENT / agent V-1 I-7)
     (FINAL-GPA / agent I-6 V-1)))

                         < OR-MERGE the extensions from each branch >

((BAG B-4 (y-1 z-1 V-1)) <-
 (OR-MERGE
    ((BAG b-1 (V-1))
     (BAG b-2 (V-1 y-1 z-1))
     (BAG b-3 (V-1)))))

                                 < RETURN the requested extension >
(RETURN (BAG B-4 (V-1)))
```

Data-level ACCESS BAGs and Merge Operations
Figure 41

$$\begin{tikzcd}
\text{Simplify query} \arrow[d] \\
\text{Do data access for each branch} \arrow[d] \\
\text{OR-MERGE to get arguments to computation} \arrow[d] \\
\begin{array}{c}\text{COLLECT argument extensions under a variable}\\ \text{to pass to computations}\end{array} \arrow[d] \\
\text{Do data accesses included in computation participants} \arrow[d] \\
\text{OR-MERGE with other arguments} \arrow[d] \\
\text{Perform Computations} \arrow[d] \\
\begin{array}{c}\text{For each branch, use AND-MERGE to combine}\\ \text{its computation and data access extensions}\end{array} \arrow[d] \\
\text{OR-MERGE the branches}
\end{tikzcd}$$

Data-level Flowchart for Computation Queries
Figure 42

```
(ENQUIRE
   (and (COUNT-UNIQUE
              (domain V-6 ("TAKES-COURSE"))
              (measure x-6 "agent" ("TAKES-COURSE"))
              (result y-6))
        (IS-PRESIDENT (agent x-6))))
```

```
                                    < ACCESS for *enquire expression >

((BAG b-1 (x-6)) <- (ACCESS (IS-PRESIDENT / agent I-1 x-6)))

                      < ACCESS for computation argument participant >
((BAG b-2 (x-6 z-6 y-7)) <-
 (ACCESS (TAKES-OFFERING / agent I-2 x-6)
         (TAKES-OFFERING / object I-2 z-6)
         (OFFERING-OF / agent I-3 y-7)
         (OFFERING-OF / object I-3 z-6)))

 < AND-MERGE IS-PRESIDENT/TAKES-COURSE to get argument extension >

((BAG B-3 (x-6 z-6 y-7)) <-
 (AND-MERGE ((BAG b-1 (x-6)) (BAG b-2 (x-6 z-6 y-7)))))

     < COLLECT extension under a variable to pass to computation >

((BAG B-4 (x-6 z-6 y-7 V-6)) <-
 (COLLECT (BAG B-3 (x-6 z-6 y-7)) nil (V-6 x-6 y-7 z-6)))

                                          < perform the computation >

((BAG B-5 (x-6 z-6 y-7 V-6 y-6)) <-
 (FOR (BAG B-4 (x-6 z-6 y-7 V-6))
     (y-6 <- (COUNT-UNIQUE (V-6 x-6 y-6)))))

            < Combine computation result with data ACCESS results
              to form the requested extension   >

((BAG B-6 (x-6 y-6)) <-
 (AND-MERGE ((BAG b-1 (x-6))
             (BAG B-5 (x-6 z-6 y-7 V-6 y-6)))))

                                       < RETURN the final extension >
(RETURN (BAG B-6 (x-6 y-6)))
```

Computation Query
Figure 43

1) The first elements of the BAGAL *assert routine are data retrieval statements followed by statements to check all of the cardinality constraints that are associated with the branch. Any tuple that fails one cardinality test is eliminated from the extension.

2) Next are precheck statements to see if the values are already stored. Values to be removed from the stored situation extension are verified to be present in the stored extension. Values to be added are verified to not already be present. No tuples are removed from the *assert extension at this step because the update BAG will need to return an extension of eligible tuples.

3) Placed after precheck statements are the statements needed to perform the corresponding additions and deletions. A tuple to be added will be added only if not already present. A tuple to be deleted will be deleted if it is present.

4) A resulting extension is returned that includes all the tuples that passed all of the cardinality checks regardless of whether the physical update was actually needed. An example of this type of data-level procedure is shown in Figure 45.

```
(*assert                                          BRANCH 1
    (or
         (and (<PRIMITIVE situation expression 1>)
              (<PRIMITIVE situation expression 2>)
                                :              :
              (<PRIMITIVE situation expression n>))

                                              BRANCH 2
         (and (<PRIMITIVE situation expression 1>)
              (<PRIMITIVE situation expression 2>)
                                :              :
              (<PRIMITIVE situation expression n>))))
```

<u>*assert</u> Expressions
Figure 44

```
(REFLECT (INTERESTING (agent V-1 P-1)
                      (result V-2 "TRUE")))
```

```
      < TUPLES BAG forms the constant tuple to be reflected >

((BAG B-1 (V-2 V-1)) <-
 (TUPLES ((V-2 <- "TRUE") (V-1 <- P-1))))

< Update BAG--This command checks cardinalities, verifies that
values are not already present, and then performs the update >

((BAG B-2 (V-2 V-1)) <-
 (FOR (BAG B-1 (V-2 V-1))
                          < data access to check cardinality >
   ((I-1 <- (CINDEX (INTERESTING (agent V-1))))
         < data access statement--check if template exists >
    (I-2 <- (CINDEX (INTERESTING (agent V-1) (result V-2))))
    (c-1 <- (COUNT I-1 I-1))
                        < check cardinalities--see if count is
                          less than the cardinality constraint
                          or if already stored >
    (IF (OR (c-1 < 1) (I-2 <> NULL)) THEN
       ((IF (I-2 = null) THEN      < if not already stored >
           ((I-2 <- (ICREATE INTERESTING))
                                           < perform update >
            (CHANGE INTERESTING / agent I-2 V-1)
            (CHANGE INTERESTING / result I-2 V-1)))
        (I-3 <- (CINDEX (IS-PERSON (agent V-1))))
        (IF (I-3 = null) THEN      < see if already stored >
           ((I-3 <- (ICREATE IS-PERSON))
                        < perform update of new object type as
                          required by query semantics >
            (CHANGE IS-PERSON / agent I-3 V-1))))))))

                                            < RETURN result >
(RETURN (BAG B-2 (V-2 V-1)))
```

Data-level *assert Procedure
Figure 45

5.4.1 Disjunctive Update Operations

Some *assert expressions that include the 'or' or 'and' connectives are ambiguous disjunctive update operations. For example, an *assert expression that contains the 'or' connective specifies that either one of the branches can be made true in order to satisfy the update. Likewise, an *assert that contains the 'not' connective enclosing the 'and' connective

```
not (and <sigma expression>
              <sigma expression>)
```

specifies that either one of the sub-expressions can be made false in order to satisfy the update. In either case, it is ambiguous which sub-expression is to be chosen. In SIDUR this ambiguity is to be resolved on a tuple-by-tuple basis by the user. One might wonder whether responsibility to communicate with the user lies at the semantic or data level. OSIRIS has chosen to place this function at the data level for the following reasons. The four update operators can occur within a PERFORM or a compound FOR or SINCE operation. These types of operations specify an extension of data values to be acted upon. Figure 46 shows an example of this. As the BAGAL query is being interpreted, prior to the execution of the update operation, the extension of a compound operation will be derived from the stored database. This retrieved extension of tuples is not available to the SIDUR routines during the time when the update portion of the operation is being translated. Therefore, the SIDUR functions are not able to disambiguate the operation while it is being translated. This implies that the data level must be able to determine which branch or set of situations, and which corresponding set of pre-conditions, is applicable to each tuple in the retrieved extension.

```
(FOR ((x)
      (IS-STUDENT (agent x))
      (REFLECT
         (or
            (TAKES-OFFERING (agent x) (object O-9))
            (TAKES-OFFERING (agent x) (object O-3))))))
```

Compound Disjunctive Update Operation
Figure 46

Resolving ambiguous updates will be done at the data level by providing it with a user interface. The user will be prompted with each tuple along with the names of the situations among which choices must be made in order to disambiguate the operation. The user then indicates which situation names have been chosen for each tuple. The input/output format of the situation names will depend on the particular CHOICE implementation. For example, Figure 47 represents, depending on the result of the first choice made, a query needing either one or two user interactions with the CHOICE function to determine which situations are to be updated. If IS-COURSE is chosen, no further CHOICE interactions are needed. Otherwise, a CHOICE between TEACHES-OFFERING and HAS-TITLE is also needed. When simplified into a set of branches and with non-PRIMITIVE situations expanded, the operation would be expressed as shown in Figure 48. As will be explained later, it is also possible for the simplified form to contain expressions to cause a REFLECT operation for TOKEN types and for required pre-conditions of ASSERT or DENY operations. This simplified form is, equivalently, one choice with three different alternatives:

```
1) IS-COURSE (agent C-3)
2) (and (TEACHES-OFFERING (agent P-5) (object C-3))
        (OFFERING-OF (object O-4)))
3) (and (HAS-TITLE (agent C-3) (object "CS-430))
        (OFFERING-OF (object O-4)))
```

Presenting the user with one arbitrarily large and complex choice of this form, possibly structured differently than the original operation and containing different situation names, is thought to be too confusing. Therefore, the canonical form is not used as the structure in which to present choices to the user.

```
(REFLECT
  (or (IS-COURSE (agent C-3))
      (and (or (TEACHES-OFFERING (agent P-5) < non-PRIMITIVE >
                                 (object C-3))
               (HAS-TITLE (agent C-3)
                          (object "CS-430")))
           (IS-OFFERING (agent O-4))))    < non-PRIMITIVE >
```

Disjunctive Update Operation
Figure 47

```
(REFLECT (or (IS-COURSE (agent C-3)
             (and (TEACHES-OFFERING (agent P-5)
                                    (object C-3))
                  (OFFERING-OF (object O-4)))
             (and (HAS-TITLE (agent C-3)
                             (object "CS-430"))
                  (OFFERING-OF (object O-4))))
```

Simplified Disjunctive Update Operation
Figure 48

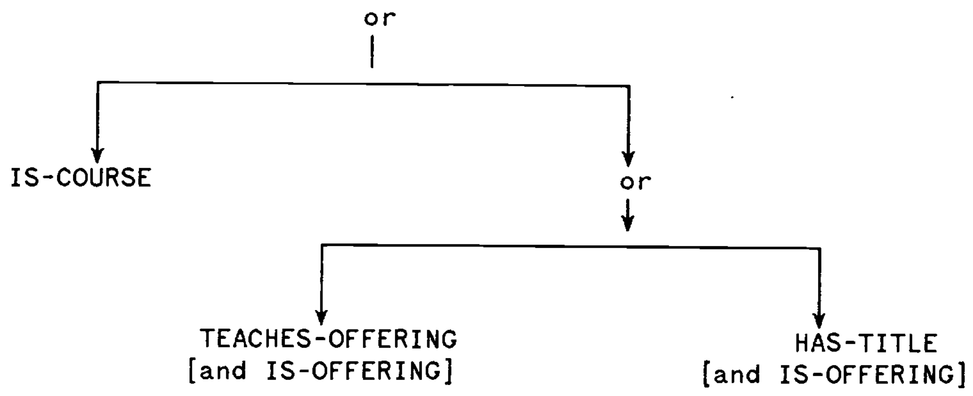
The method of presenting choices to the user is based on constructing a CHOICE tree representing the structure and situation names of the original expression rather than on the structure and names of the *assert expression of the canonical form. The original structure is in many cases closer to the structure of the user's real-world conception of the operation. Note however, that disjunctive TOKEN-type and required pre-condition REFLECTs are also added to the CHOICE tree so there may still be surprises in the presentation of choices to the user. Each path in the tree from root to a leaf represents a possible sequence of choices. The list of situation expressions on each path and the choices made for each tuple specify the update to be performed for the operation. One of the branches in the canonical form is associated with each leaf of the tree. So, each tuple is updated into the branch associated with the leaf that the user has chosen for it.

For the expression in Figure 47, the CHOICE tree represents one or two choices in order to determine the alternative and is shown in Figure 49. This CHOICE tree is obtained by the simplification algorithm from the original expression by transformations on each 'and' expression so that any 'or' expressions that it encloses become nodes with branches indicating the choices that are needed. In this case the choices that are needed are

1) a choice between IS-COURSE and the remaining situations, and

2) if the remaining situations are chosen, a choice between TEACHES-OFFERING and HAS-TITLE.

The CHOICE names become part of the canonical form and are stored as the first element of each branch. Refer to Figure 55.



CHOICE TREE FOR DISJUNCTIVE Update
Figure 49

A simple implementation of choice that presents, for each tuple, all of the alternatives in the choice tree to the user has been considered but rejected because in complex expressions some choices may become irrelevant based on earlier choices, and there is no need to collect unneeded information. For example, if the left branch of a tree is chosen, none of choices that are represented by the right branch will be needed. After SIDUR has determined what the structure of the choice tree is, it is translated into BAGAL statements that present the choices to the user. This is done by looking at the CHOICE names in each branch of the canonical form. The CHOICE names in each branch are stored in a nested list structure that reflects the structure of the tree. Each level of nesting indicates another CHOICE node in the tree. The first-level situation names or lists from each branch are presented to the user for the first CHOICE. In our example, branch 1 would have the simple element IS-COURSE. Branch 2 and branch 3 would each have a list (or a summary of a long list.) The elements in the nested lists are joined into one list that represents one alternative. The first CHOICE is between the element IS-COURSE and the list (TEACHES-OFFERING HAS-TITLE.) The second CHOICE function will look at the CHOICE names and lists nested one level deep within the CHOICE names of each branch. In this case, only two single elements remain so the second CHOICE is between TEACHES-OFFERING and HAS-TITLE. Figure 50 shows the data-level statements needed to present these choices to the user.

After translating the statements needed to present the choices to the user, the next step is to translate data-level statements that will put the tuples into separate extensions according to the CHOICE made for each tuple. Remember that each of these alternatives corresponds to one of the branches of the canonical form of the operation. SIDUR then translates separate statements in the BAGAL query for each branch. Each set of BAGAL statements essentially consists of a separate update operation for each of the overall alternatives that can be chosen by the user. This is done in order to simplify and to provide some regularity to the structure of a data-level disjunctive update operation as well as to allow both the data level and the SIDUR level to handle arbitrarily complex operations of this type. Of course, optimization is still possible at the data level because the entire SIDUR operation remains expressed in one BAGAL query. The algorithm for the BAGAL operation is illustrated in Figure 51. Figure 52 shows the Chapter IV BAGAL query, which has a procedure to present a CHOICE to the user. After the CHOICE statements, it contains separate sets of routines to perform the update operation for each branch of the disjunct. Figure 55 shows the full canonical form of the operation in Figure 52.

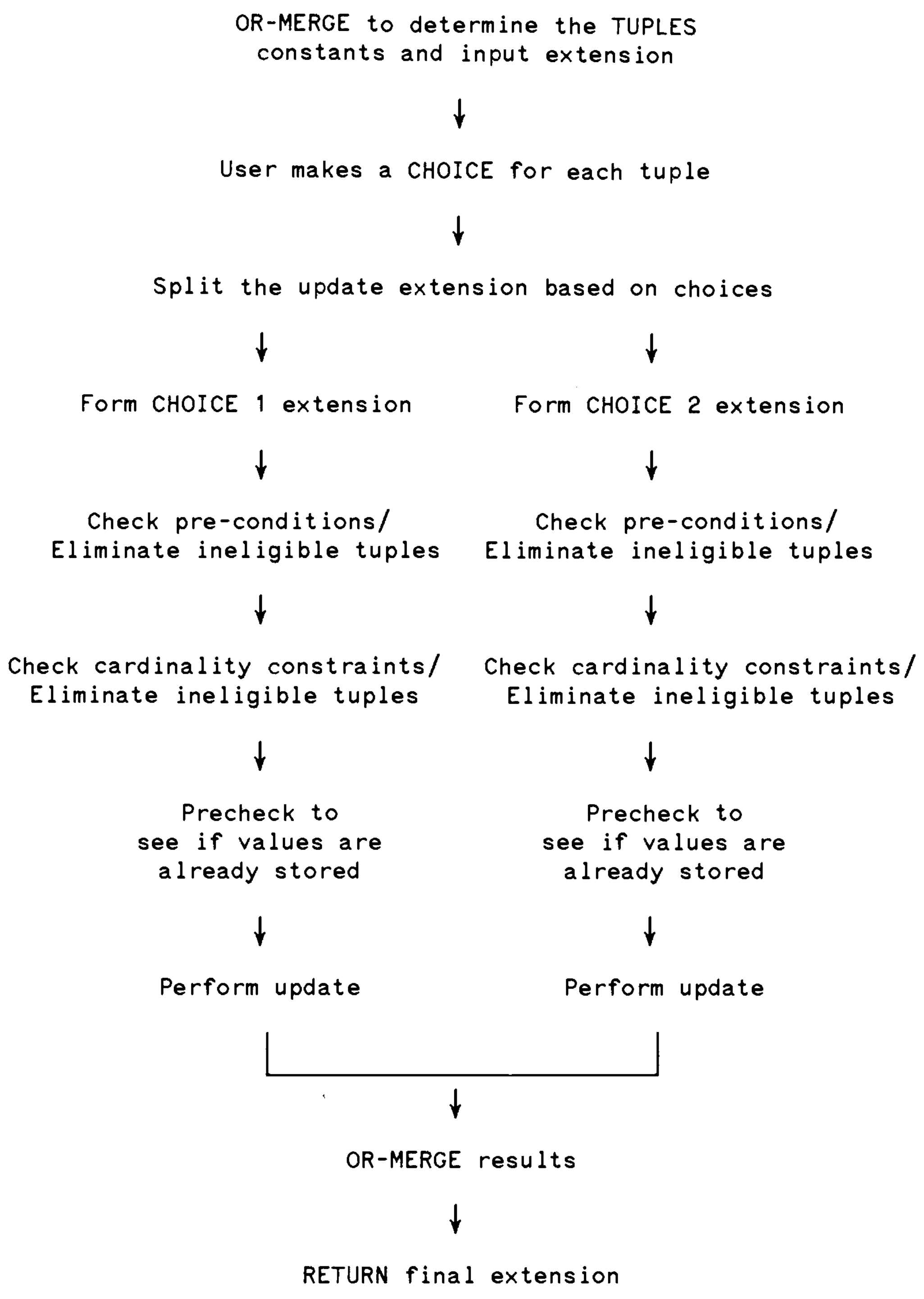
CHOICE NAMES FROM THE CANONICAL FORM:

(IS-COURSE)	BRANCH 1
((TEACHES-OFFERING))	BRANCH 2
((HAS-TITLE))	BRANCH 3

CHOICE PORTION OF A BAGAL QUERY:

```
                                             < BAG to get user's choice >

((BAG B-2 (V-10 B-11 c-10)) <-
 (FOR
   (BAG B-1 (V-10 B-11))
     ((c-10 <-                                          < first CHOICE >
        (CHOICE "choose one" (IS-COURSE
                              ((TEACHES-OFFERING  HAS-TITLE)))
               (V-10 B-11)))
    (CASE
        (((c-10 = IS-COURSE)                   < BRANCH 1 is chosen >
          ((c-10 <- 1)))
         ((c-10 = ((TEACHES-OFFERING  HAS-TITLE)))
          ((c-10 <-                                    < second CHOICE >
             (CHOICE "choose one" (TEACHES-OFFERING
                                   HAS-TITLE)
                    (V-10 B-11)))
           (CASE
             (((c-10 = TEACHES-OFFERING)
               ((c-10 <- 2))                   < BRANCH 2 is chosen >
              ((c-10 = HAS-TITLE)
               ((c-10 <- 3))))))))))           < BRANCH 3 is chosen >

 < The expression above is a fragment of a BAGAL query. >
```

Presenting the Choice to the User
Figure 50


OR-MERGE to determine the TUPLES constants and input extension
User makes a CHOICE for each tuple
Split the update extension based on choices
Form CHOICE 1 extension
Form CHOICE 2 extension
Check pre-conditions/ Eliminate ineligible tuples
Check pre-conditions/ Eliminate ineligible tuples
Check cardinality constraints/ Eliminate ineligible tuples
Check cardinality constraints/ Eliminate ineligible tuples
Precheck to see if values are already stored
Precheck to see if values are already stored
Perform update
Perform update
OR-MERGE results
RETURN final extension

Data-level Disjunctive Update
Figure 51

```
(REFLECT (and (IS-EVENT-NAME (agent x-10)
                             (object V-10 "AI-TOPICS"))
              (or (IS-NOON-MEETING (agent x-10))
                  (IS-SEMINAR (agent x-10))))
```

```
((BAG B-1 (V-10 V-14)) <-  (TUPLES    < TUPLES BAG for constants >
  ((V-10 <- "AI-TOPICS) (V-14 <- 12))))

                                            < BAG to get users CHOICE >
((BAG B-2 (V-10 c-10)) <- (FOR (BAG B-1 (V-10))
 ((c-10 <- (CHOICE "choose one" (IS-NOON-MEETING IS-SEMINAR)
                                (V-10)))
  (CASE (((c-10 = IS-NOON-MEETING) ((c-10 <- 1)))
         ((c-10 = IS-SEMINAR) ((c-10 <- 2))))))))

   < split reflect extension based on user CHOICE for each tuple >
((BAG B-3 (V-10)) <-                                  < BRANCH 1 >
 (FOR (BAG B-2 (V-10 c-10)) (IF (c-10 <> 1) THEN nil)))

((BAG B-4 (V-10)) <-                                  < BRANCH 2 >
 (FOR (BAG B-2 (V-10 c-10)) (IF (c-10 <> 2) THEN nil)))

                                 < *enquire expression BRANCH 1 >
((BAG b-5 (x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL)) <-
  (ACCESS (V-11 = "MEETING")    (V-12 = "12AM")
   (HAS-TITLE / agent I-1 x-10)  (HAS-TITLE / object I-1 V-11)
   (MEETING-TIME/agent I-2 x-10) (MEETING-TIME/object I-2 V-12)))

                                 < *enquire expression BRANCH 2 >
((BAG b-6 (x-10 V-13 I-3 NO-NULL I-4 NO-NULL)) <- (ACCESS
   (V-13 = "SE-430")       (IS-COURSE / agent I-3 x-10)
   (HAS-TITLE / agent I-4 x-10) (HAS-TITLE / object I-4 V-13)))

  < merge constants and *enquire extensions to get update tuples >
                                                      < BRANCH 1 >
((BAG B-7 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL)) <-
 (AND-MERGE ((BAG B-1 (V-10 V-14))
             (BAG b-5 (x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL)))))

                                                      < BRANCH 2 >
((BAG B-8 (V-10 V-14 x-10 V-13 I-3 NO-NULL I-4 NO-NULL)) <-
 (AND-MERGE ((BAG B-1 (V-10 V-14))
             (BAG b-6 (x-10 V-13 I-3 NO-NULL I-4 NO-NULL)))))

      < BAGs for each branch to do cardinality check, precheck to
        see if already stored, and update operations  BRANCH 1 >
((BAG B-9 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL)) <-
 (FOR (BAG B-7 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL))
  ((IF (x-10 = NULL) then (x-10 <- (OCREATE EVENT)))
   (I-5 <- (CINDEX (IS-EVENT-NAME (agent x-10) (object V-10))))
```

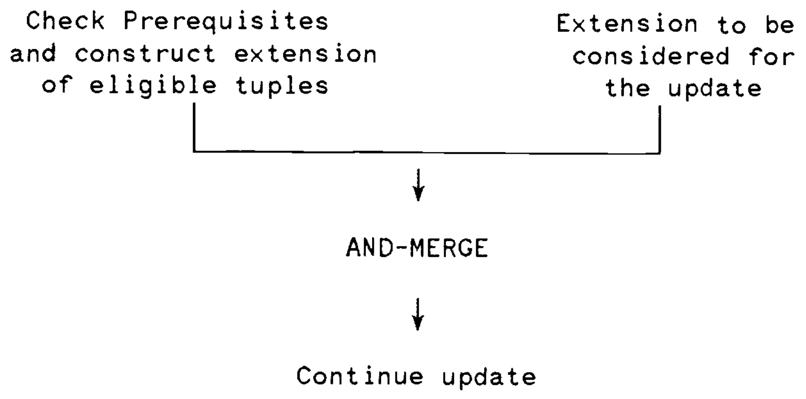
```
    (IF (I-5 = null) THEN                                    < precheck >
       ((I-5 <- (ICREATE IS-EVENT-NAME))
        (CHANGE IS-EVENT-NAME / agent I-5 x-10)                < update >
        (CHANGE IS-EVENT-NAME / object I-5 V-10)))
    (I-6 <- (CINDEX (IS-NOON-MEETING (agent x-10))))
    (IF (I-6 = null) THEN                                    < precheck >
       ((I-6 <- (ICREATE IS-NOON-MEETING))
        (CHANGE IS-NOON-MEETING / agent I-6 x-10)))            < update >
    (I-7 <- (CINDEX (IS-EVENT (agent x-10))))
    (IF (I-7 = null) THEN                                    < precheck >
       ((I-7 <- (ICREATE IS-EVENT))
        (CHANGE IS-EVENT / agent I-7 x-10)))                   < update >
    (I-8 <- (CINDEX (NOT-IS-NOON-MEETING (agent x-10))))
    (IF (I-8 <> null) THEN                                   < precheck >
       ((DELETE NOT-IS-NOON-MEETING / agent I-8 x-10))))))
                                                             < BRANCH 2 >
((BAG B-10 (V-10 V-14 x-10 V-13 I-3 NO-NULL I-4 NO-NULL)) <-
 (FOR (BAG B-8 (V-10 V-14 x-10 V-13 I-3 NO-NULL I-4 NO-NULL))
  ((IF (x-10 = NULL) then (x-10 <- (OCREATE EVENT)))
   (I-9 <- (CINDEX (LIMIT (agent x-10))))              < cardinality >
   (I-10 <- (CINDEX (LIMIT (agent x-10) (value V-14))))
   (c-11 <- (COUNT I-9 I-9))                           < cardinality >
   (IF (OR (c-11 < 1) (I-10 <> NULL)) THEN
     ((I-11 <- (CINDEX (IS-EVENT-NAME (agent x-10)(object v-10))))
      (IF (I-11 = null) THEN                                 < precheck >
        ((I-11 <- (ICREATE IS-EVENT-NAME))
         (CHANGE IS-EVENT-NAME / agent I-11 x-10)              < update >
         (CHANGE IS-EVENT-NAME / object I-11 V-10)))
      (I-12 <- (CINDEX (OFFERING-OF (agent x-10))))
      (IF (I-12 = null) THEN                                 < precheck >
        ((I-12 <- (ICREATE OFFERING-OF))
         (CHANGE OFFERING-OF / object I-12 x-10)))             < update >
      (IF (I-10 = null) THEN                                 < precheck >
       ((I-10 <- (ICREATE LIMIT))
        (CHANGE LIMIT / agent I-10 x-10)                       < update >
        (CHANGE LIMIT / value I-10 V-14)))
      (I-13 <- (CINDEX (IS-EVENT (agent x-10))))
      (IF (I-13 = null) THEN                                 < precheck >
       ((I-13 <- (ICREATE IS-EVENT))
        (CHANGE IS-EVENT / agent I-13 x-10))))))))             < update >

            <Merge result from each branch to get final result >
((BAG B-11 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL V-13
         I-3 NO-NULL I-4 NO-NULL)) <-
 (OR-MERGE
   (BAG B-9 (V-10 V-14 x-10 V-11 V-12 I-1 NO-NULL I-2 NO-NULL))
   (BAG B-10 (V-10 V-14 x-10 V-13 I-3 NO-NULL I-4 NO-NULL))))

(RETURN (BAG B-11 (V-10 x-10)))                       < RETURN results >
```

BAGAL Update Operation
Figure 52

5.4.2 Tuple-level Atomicity

*assert operations are defined to fail if any pre-condition of any of the situations that will have values added does not hold with respect to the current database extension. However, the entire operation may involve an extension of multiple tuples each of which is a candidate for the update operation. In this implementation each tuple that satisfies its pre-conditions will be updated into its corresponding branch of the *assert expression. Other tuples may fail their pre-condition tests and will not be used in the update operation.

In order to allow optimization at the data level, to allow update operations on an extension of retrieved values, and to permit arbitrarily complex pre-condition expressions, the determination of which tuples within an extension meet their pre-conditions is implemented by means of the data-level set operators--AND-MERGE, OR-MERGE and MINUS. A routine is called to translate the entire pre-condition expression into BAGAL procedures that will retrieve the extension of tuples that meet all the pre-conditions. Recall that a separate set of BAGAL update operations will be translated for each branch. Then BAGAL statements are translated to use the AND-MERGE data operation, which corresponds to the SIDUR 'and' connective, to do a set intersection with the extension of tuples being considered for the update operation. This extension can be a TUPLES BAG extension, an extension resulting from the first phase of a PERFORM or compound operation, or the OR-MERGE of these two extensions. Those tuples remaining in the intersection set are still "eligible" for the update operation. Figure 52 shows an example of this type of operation. Figure 53 shows the flowchart for this operation for each branch of a disjunctive update.


Check Prerequisites and construct extension of eligible tuples
Extension to be considered for the update
AND-MERGE
Continue update

Algorithm of Data-level Update Operation
Figure 53

5.5 Syntax Checking and Input String Validation

The structure and content of the syntax checking routines are based directly on the data manipulation portions of the SIDUR BNF definition as shown in Appendix E. The routines are initially entered through the function 'dml-command?' and are called from the 'sidur' function in order to check the syntactic correctness of the user's original operation. Later, during the actual expression preparation and translation process, various subsections of these routines are called to determine the type of the next element in the current expression. Because some of the syntax routines involve inefficient Franz Lisp symbol-name character manipulation, an effort has been made to minimize the number of calls to these routines. So, due to performance considerations, because a thorough initial check of the syntax of the operation is done, and because it is assumed that the schema construct specifications are syntactically correct, only minimal syntax checking is done during translation.

Several conventions have been followed in the syntax routines. The names of these routines all end with a question mark "?" in order that they be easily recognized by a reader of the SIDUR source code. In addition, the names of the syntax routines are those of the type of structure being looked for. Each syntax routine returns only a true or false indication of whether the structure being looked for has been found and is correct.

A future change to the SIDUR implementation might include making the syntax checking routines "smart" rather than "dumb." That is, rather than leaving the decision to high-level routines as to which type of structure is permissible as the next element in the operation and then calling a corresponding syntax routine, the syntax routines themselves could analyze the element and return a type value rather than a boolean value. This change might improve performance by combining element type determination with syntax checking.

Included with the syntax routines are a few simple "access" routines, which return the value of an element of an operation. For example, given the situation expression

```
(TAKES-COURSE (agent x) (object y))
```

the 'terms-of' function will return the participants

```
(agent x) (object y).
```

The purpose of these routines is twofold. Reading the Franz Lisp translation functions is often easier if an element is referred to by name such as 'situation-name' rather than as the Franz Lisp equivalent '(car expr)'. In addition, changes to data structures can be more easily made by changing these routines rather than making many changes in the high-level routines.

5.5.1 Data-value Type Checking

Prior to an *assert operation that will add data values to a situation extension, user-supplied data values are verified to be of the correct type as specified in the schema. That is, STRING, INTEGER, and REAL data values are checked against the form:, minval:, and maxval: slots in the corresponding data-value class schema specifications. (Refer to Section 5.9.3 for a discussion of how TOKEN values are added to extensions.) A set of object checking routines is entered through the 'objects-in-prescribed-class?' function. Object checking is done when the expression simplification routines encounter a situation to which user-supplied constant data values will be added. Failure of the entire operation will occur if any data value does not comply with the type definition. Because values are checked prior to being added by an update operation, it is semantically unnecessary to check values in an ENQUIRE operation or in an expression whose values will be deleted from an extension database.

Objects whose corresponding data-value type is INTEGER or REAL are verified to be within the allowable range as determined by the minval: and maxval: schema slots. However, both of these types are simply treated as Franz Lisp numbers so, in this implementation, no attempt is made to provide strict type checking for INTEGER and REAL types other than the use of Franz Lisp functions to determine that the values are numbers within their ranges.

At the data level, values that are retrieved from the database during the first part of a compound operation can be added directly to the extensions of other situations referred to in the second part. There is at this time no provision implemented in OSIRIS for checking the form:, minval:, or, maxval: slots of these retrieved values prior to addition.

5.6 Operation Preparation Routines

After the syntax of an operation is checked, the operation is prepared for translation by:

1) replacing each of the user's simple variables with a unique SIDUR-generated variable, and

2) inserting a unique variable in front of each constant data value in the operation because variables are used in the BAGAL query to refer to constants.

Variables are replaced in order to avoid using the same variables in the operation as those which may be found in the schema specification slots. Otherwise, because expressions from the schema may be added to the original expression, duplicate use of a variable name to stand for different values might occur and would result in incorrect translations. 'make-vars-unique' is the routine that replaces the simple variables in an operation. Simple variables, as defined in the SIDUR BNF, may consist of a single lower case alphabetic character. It is suggested that, if needed, the definition later be expanded to include a lower case letter followed by a single digit. For now, however, performance of the syntax routines has been improved by restricting the definition to simple symbol operations rather than character operations on symbol names. A list of distinct variables in the operation is made and a unique system variable for each is created using the Franz Lisp 'gensym' function. These system variables are symbol names in the form of a lower case letter followed by a hyphen and digits. The lower case letter is the same character as the user's original variable. This list is then used to substitute each new variable for each occurrence of the corresponding simple variable in the original operation.

Next the 'find-constants' routine lists each distinct constant from the operation along with a corresponding, unique, system-generated variable of the form "V" followed by a hyphen and digits. The system variables are then inserted into the expression in front of each occurrence of their corresponding constants. This variable insertion is done because the data-level language refers to a constant by its corresponding variable name when a TUPLES BAG is created to hold the extension of constants in an update expression. In addition, constraints in ACCESS BAGs are expressed as a condition statement such as 'V-31 = "SMITH"'. Finally, the data language provides for returning the results of a BAG operation or ACCESS by forming an extension of all of the data values that are referred to in the format list by their corresponding variable names.

To show an example of the process of operation preparation

```
(HAS-NAME (agent x) (object "JOHN-JONES"))
```

becomes

```
(HAS-NAME (agent x-4) (object V-31 "JOHN-JONES")).
```

5.7 Expression Simplification Routines

5.7.1 The Canonical Form

Figure 54 and Figure 55 show examples of the canonical form. The list structure of the canonical form represents an analysis of all of the information needed to complete an operation. It consists of sub-lists each representing one branch of the disjunctive. Each branch is in turn composed of the following items:

1) CHOICE names--the situation names which, if chosen by the user for a tuple, determine that this branch is to be used in the update of the tuple,

2) *enquire or *assert--the set of situation expressions (This is the *assert expression for this branch if this is an update operation. This is the data retrieval portion of the *enquire expression for this branch if this is a data access operation.),

3) cardinalities--the cardinality constraints,

4) *enquire--the pre-conditions (the *enquire expression of an update operation),

5) computations--set of computation expressions (part of the *enquire expression of a data access operation),

6) empty--expressions in the scope of the 'empty' connective (part of the *enquire expression of a data access operation.)

This structure is used to guide the translation routines in producing the data-level query. If the operation is ENQUIRE or CHECK, items 2, 5, and 6 are used by the translation routines. Item 2 is the main data access *enquire expression. Items 5 and 6 are part of the *enquire expression but are handled by the translation routines separately from item 2. Computation and 'empty' expressions are translated into data-level operations whose results are merged with the other *enquire expression results. The *assert operation translation routines use items 1, 2, 3 and 4 to create BAGAL queries. Item 2 is the *assert expression, and item 4 is the *enquire expression, which contains necessary pre-conditions and required pre-conditions for a REFLECT or REFLECT-NOT operation. 'empty' expressions are listed with the *assert expression with 'empty' replaced by the 'not' connective.

```
(ENQUIRE
  (or (IS-FACULTY  (agent V-1 S-1))
      (and (or (IS-STUDENT (agent V-1 S-1))
               (IS-STAFF-STUDENT (agent V-1 S-1)))
           (FINAL-GPA (agent V-1 S-1))))))
```

```
                                                  BRANCH 1
          (nil)
*enquire  ((IS-FACULTY (agent V-1 S-1)))
          (nil)
          (nil)
          (nil)
          (nil)

                                                  BRANCH 2
          (nil)
*enquire  ((IS-PERSON (agent V-1 S-1))
           (TAKES-OFFERING (agent V-1 S-1) (object z-1))
           (OFFERING-OF (agent y-1) (object z-1))
           (FINAL-GPA (agent V-1 S-1))))
          (nil)
          (nil)
          (nil)
          (nil)

                                                  BRANCH 3
          (nil)
*enquire ((IS-STAFF-STUDENT (agent V-1 S-1))
          (FINAL-GPA (agent V-1 S-1))))
          (nil)
          (nil)
          (nil)
          (nil)

< The three *enquire extensions will be OR-MERGED to produce
  the final resulting extension. >
```

Canonical Form of An ENQUIRE Operation
Figure 54

```
(REFLECT (and (IS-EVENT-NAME (agent x)
                             (object V-10 "AI-TOPICS"))
              (or (IS-NOON-MEETING (agent x))
                  (IS-SEMINAR (agent x))))
```

< See Appendix I for the schema and Figure 52 for the BAGAL query >

```
                                                      BRANCH 1
CHOICE names    (IS-NOON-MEETING)
*assert         ((IS-EVENT-NAME (agent x)
                  (object V-10 "AI-TOPICS")) PRIMITIVE definition:
                (IS-NOON-MEETING (agent x))  PRIMITIVE definition:
                (IS-EVENT (agent x))         TOKEN definition:
                (not (NOT-IS-NOON-MEETING    negative extension
                        (agent x))))            of open-world
                                                situation
cardinalities   (nil)                        none
*enquire        ((HAS-TITLE (agent x)        necessary:
                     (object V-11 "MEETING")) pre-condition
                 (MEETING-TIME (agent x)     required:
                     (object V-12 "12AM")))     pre-condition
computations    (nil)                        none
empty           (nil)                        none

                                                      BRANCH 2
CHOICE names    (IS-SEMINAR)
*assert         ((IS-EVENT-NAME (agent x)    PRIMITIVE
                   (object V-10 "AI-TOPICS"))   definition:
                 (OFFERING-OF (object x))    TOKEN-type and
                                               PRIMITIVE definition:
                 (LIMIT (agent x)            sigma-bound
                   (value V-14 12))             definition:
                 (IS-EVENT (agent x)))       TOKEN definition:
cardinalities   ((1 LIMIT (agent x)))
*enquire        ((IS-COURSE (agent x))       necessary: pre-cond'n
                 (HAS-TITLE (agent x)        required:
                   (object V-13 "SE-400")))     pre-condition
computations    (nil)                        none
empty           (nil)                        none
```

Canonical Form of a REFLECT Operation
Figure 55

5.7.2 The Simplification Algorithm

The expression simplification algorithm used to produce the canonical form is based on a set of recursive algorithms that convert an arbitrarily complex situation expression into the form of a disjunct of conjuncts (i.e. branches.) For an ENQUIRE or CHECK operation, as each sub-expression is encountered, the following actions are taken:

1) The algorithm determines whether it is a single situation or computation expression or whether it contains any of the connectives 'and', 'or', 'not', and 'empty'.

2) Computation expressions and expressions that are enclosed within the scope of the 'empty' connective are kept track of separately from the rest of the situation expression for each branch. They are listed separately from the situations because they are handled separately by the data-level procedure and by the translation algorithms.

3) If a complex sub-expression is found, the algorithm recursively converts it into a disjunct of conjuncts. This is done by distributing the operands of a nested 'or' connective over an enclosing 'and' connective. The 'and' expression is listed twice, once with each operand of the 'or' expression. This partial result is combined with other partial results to obtain the final simplified expression. If the 'not' connective is found, the polarity of the negation indicator is changed so that negation will be applied directly to the enclosed situations. The algorithm is called recursively to analyze each expression that is enclosed by a connective.

4) If the sub-expression found is a single situation or computation expression, it is analyzed by the algorithm.

4a) PRIMITIVE and SYSTEM expressions are added to the appropriate branch of the canonical form If the negation indicator is 'on', the expression is preceded by 'not'.

4b) Non-PRIMITIVE situation definitions and non-SYSTEM computation expressions are replaced by equivalent expansions containing only PRIMITIVE or SYSTEM definitions after a re-binding of variables. Then, the resulting expression is simplified by calling the algorithm recursively.

5.7.3 Definition Expansion

Prior to sigma expansion of non-PRIMITIVE expressions, their participants must be correctly associated with the participants in the schema definition. This is the sigma binding process, which consists of the following steps:

1) Variables that are internal to a schema construct specification are listed in the participants: slot of the schema definition next to their corresponding role names. These variables are used to indicate where each participant of the original situation or computation is to be substituted into the definition: expression for that construct.

2) Each of the original participants is bound to, which means it is associated with, the variable defining its corresponding role in the schema definition.

3) Each occurrence of a variable in the definition: slot is replaced by the original participant, which is bound to it.

4) This substituted definition: is substituted in the main expression in place of the non-PRIMITIVE expression. For example, the expression

```
(TEACHES-COURSE (agent P-20) (object C-20))
```

is extensionally equivalent to

```
(and (TEACHES-OFFERING (agent P-20) (object z))
     (OFFERING-OF (agent C-20) (object z)))
```

when the extension that is represented by the latter expression is projected onto the two agent roles. As shown in Figure 56, the equivalent schema-specified expression is obtained by associating the agent value P-20 with the variable x, representing the 'agent' participant in the schema entry for TEACHES-COURSE. Likewise, the 'object' participant C-20 is associated with the variable y. Then every occurrence of the variable x in the schema definition: slot is replaced by the value P-20, and every occurrence of the variable y is replaced by C-20. This substitution produces the association between data values that the schema specifies to be the "definition" of the non-PRIMITIVE situation.

Schema Specification	Expression Participant	Associated Variable and Data Value
agent x INSTRUCTOR	agent P-20	x P-20
object y COURSE	object C-20	y C-20

Mapping Table
Figure 56

5.7.4 *assert Operations

If the operation being analyzed by the simplification algorithm is one of the *assert operations, REFLECT, ASSERT, REFLECT-NOT, or DENY, the algorithm determines the structure of the *assert expression. In addition, the situation names that will be presented to the user by the data-level CHOICE function are determined and listed with the corresponding branch. The algorithm also keeps track of all other sub-operations that will be needed for each branch found in the simplified expression.

The implied sub-operations for a REFLECT or REFLECT-NOT operation are:

1) Verify STRING-type object participants,
2) REFLECT TOKEN-type object participants,
3) Enforce cardinality constraints,
4) Enforce necessary pre-conditions, and
5) Enforce required pre-conditions.

The implied sub-operations for an ASSERT or DENY operation are:

1) Verify STRING-type object participants,
2) REFLECT TOKEN-type object participants,
3) Enforce cardinality constraints,
4) Enforce necessary pre-conditions, and
5) REFLECT required pre-conditions.

As will be explained later, the algorithm determines which sub-operations are needed by examining the schema specifications of all single situation expressions to determine whether they contain any other information that will have an effect on the operation. For example, in order to insure referential integrity (Date, 1983) for each object of type TOKEN, a sub-expression representing the implied REFLECT operation on the defining situation is added to the simplified *assert expression. Note that required pre-conditions for an ASSERT operation are semantically equivalent to the *assert expression, and so they are included with the *assert expression. For REFLECT operations required pre-conditions are semantically equivalent to necessary pre-conditions and are included with the *enquire expression. Figure 57 shows the different effects on the canonical form of an ASSERT and a REFLECT operation that operate on the same sigma expression. Note that the ASSERT operation became disjunctive due to the implied REFLECT of the required pre-condition, which was disjunctive. How these pre-condition expressions are merged with an *assert or *enquire expression will be explained soon.

When simplifying an *assert expression, the simplification algorithm keeps track of which operation--add or remove--is to be applied to each situation. Each time the 'not' connective is encountered, the "polarity" of the operation that is to be applied is reversed. All situation expressions that will be involved in the *assert operation are listed in the canonical form as discussed earlier. Situation expressions that will have data values removed are listed with a 'not' preceding them.

The simplification algorithm for an update operation is:

1) the algorithm determines whether it is a single situation or computation expression or whether it includes any of the connectives 'and', 'or', 'not', and 'empty'.

2) For an update operation, computation expressions are not allowed. Expressions that are enclosed within the scope of the 'empty' connective are handled as if they were in the scope of the 'not' connective.

3) If it is a complex sub-expression, the algorithm recursively converts it into a disjunct of conjuncts. This is done by distributing the operands of a nested 'or' connective over an enclosing 'and' connective. The algorithm identifies the CHOICE

```
situation IS-FACULTY
    participants: ((agent x EMPLOYEE))
    necessary: (FINAL-GPA (agent x) )
    required: (or (TEACHES-OFFERING (agent x) (object z))
                  (CAN-TEACH (agent x)))
    extension: (OPEN-WORLD)
    definition: (PRIMITIVE))
```

```
                 ASSERT   IS-FACULTY

 *enquire (and                 BRANCH 1
              (FINAL-GPA)                  < necessary: >
              (OFFERING-OF)                < necessary: for
              (CAN-TEACH))                   TEACHES-OFFERING,
                                             which is REFLECTed
                                             as a required
                                             pre-condition >
          (FINAL-GPA))         BRANCH 2

 *assert  (and (IS-FACULTY)    BRANCH 1    < definition: >
               (TEACHES-OFFERING)          < required: >
               (IS-INSTRUCTOR)             < TOKEN type >
               (IS-EMPLOYEE)               < TOKEN type >
               (OFFERING-OF))              < TOKEN type >

          (and (IS-FACULTY)    BRANCH 2    < definition: >
               (CAN-TEACH)                 < required: >
               (IS-PERSON)                 < TOKEN type >
               (IS-EMPLOYEE)))             < TOKEN type >

                 REFLECT   IS-FACULTY

 *enquire (or (and             BRANCH 1
                  (FINAL-GPA)              < necessary: >
                  (TEACHES-OFFERING))      < required: >
              (and
                  (FINAL-GPA)              < necessary: >
                  (CAN-TEACH)))            < required: >

 *assert  (and (IS-FACULTY)    BRANCH 1    < necessary: >
               (IS-EMPLOYEE))              < TOKEN type >

< The expressions above are abbreviated sigma expressions. >
```

ASSERT and REFLECT Operations
Figure 57

names for each of the 'or' branches by taking the situation names enclosed by an 'or' connective. The 'and' expression is listed twice, once with each operand of the 'or' expression. This partial result is used to combine with other partial results to obtain the final simplified expression. If the 'not' connective is found, the polarity of the negation indicator is changed so that negation will be applied directly to the enclosed situations. The algorithm is called recursively to analyze each expression enclosed by a connective.

4) If the sub-expression is a single situation, it is analyzed by the algorithm. Necessary and required pre-conditions are handled as described later in this chapter. In summary, necessary pre-conditions are placed in the *enquire expression. For REFLECT operations, required pre-conditions are also placed in the *enquire expression. But, for ASSERT operations, a required pre-condition is placed in the *assert expression and is then recursively analyzed by the algorithm as if it were a REFLECT operation.

4a) PRIMITIVE expressions will be placed on the list structure of the canonical form. Cardinality constraints are listed if not already present. TOKEN data types will be handled as described later in this chapter. In summary, their defining situations become part of the *assert expression as if

```
(REFLECT (<defining situation>))
```

were part of the original operation.

4b) Non-PRIMITIVE situation definitions and non-SYSTEM computation expressions will be replaced by equivalent expressions containing only PRIMITIVE or SYSTEM definitions after a re-binding of variables. Then, the resulting expression is simplified by calling the algorithm recursively.

5.8 Cardinality Constraints

The semantic update operations, REFLECT, ASSERT, DENY, and REFLECT-NOT, are defined to fail if the addition of any tuple to a PRIMITIVE situation's extension will cause a violation of any of the situation's schema-specified cardinality constraints. Cardinality constraints belonging to situations from which values will be deleted do not affect the operation because constraints only determine the maximum allowable number of occurrences of participants in an extension and not the minimum number. For an update operation, the simplification algorithm keeps track of all relevant cardinality constraints that involve the same, or a subset of, the participants about to be added. The cardinality constraints are re-expressed as situation expressions. Each expression is preceded by a digit denoting the maximum allowable number of occurrences of the participants.

For example, the cardinalities: slot in the schema specification for the GRADE-FOR situation is shown in Figure 58

```
(( 1 x y )).
```

This slot has one constraint whose interpretation is: No more than one occurrence of any specific pair of participants for the agent and object roles of this situation is allowed in the extension.

```
situation GRADE-FOR
    participants:  ((agent x STUDENT)
                    (object y OFFERING)
                    (value z GRADE))
    cardinalities:  (( 1 x y ))
    definition:  (PRIMITIVE)
    extension:  (CLOSED-WORLD)
```

Situation With Cardinality Constraint
Figure 58

The simplification algorithm re-expresses the constraint as

```
(1 GRADE-FOR (agent x) (object y))
```

by determining from the participants: slot which roles correspond to the variables in the constraint. Then, the participants that will be added to the agent and object roles are sigma-bound to this expression of the constraint. For example, if the expression in the update operation is

```
(GRADE-FOR (agent S-13) (object O-13) (value "A")),
```

then the sigma-bound constraint will be

```
(1 GRADE-FOR (agent S-13) (object O-13)).
```

The sigma binding process was discussed earlier in Section 5.7.3 and in Chapter IV.

This constraint is added to the list structure of the canonical form of the operation. The translation routines use this expression to translate the constraint into data-level statements. If the *assert will violate a cardinality constraint for any situation being *asserted for a tuple, then the entire *assert for this tuple will fail. An example of these statements is shown in Figure 59. Statements to cause the check on the constraint as well as the possible subsequent failure can be placed in the same data-level BAG with the data level update operators. They can be placed in the same BAG because each constraint is concerned with only a single PRIMITIVE situation. Furthermore, each data retrieval to check a cardinality constraint corresponds to exactly one of the tuples of constant values that is about to be added. Thus, there is no possibility that a complicated expression involving multiple situation retrievals and OR-MERGE (union) operations will be needed. The simplification algorithm thus lists the cardinality constraints for each branch separate from the necessary pre-conditions because the translation routines handles those two types of expressions differently. Figure 59 shows the BAGAL code that enforces the cardinality constraint of (1 x y) on a situation INTERESTING.

5.9 Pre-conditions

5.9.1 Necessary Pre-conditions

An update operation's necessary pre-conditions consist of the sigma-expanded necessary: slots of both PRIMITIVE and non-PRIMITIVE situations being *asserted by the operation. The necessary: slots of situations from which values will be removed do not affect the operation and, therefore, are not included in the list of pre-conditions. This is equivalent to saying that the tuple must meet the

```
< Update BAG--This command checks cardinalities, verifies that
values are not already present, and then performs the update >

((BAG B-2 (V-2 V-1)) <-
 (FOR (BAG B-1 (V-2 V-1))

                              < data access to check cardinality >
   ((I-1 <- (CINDEX (INTERESTING (agent V-1))))
    (c-1 <- (COUNT I-1 I-1))

              < data access statement--check if template exists >
    (I-2 <- (CINDEX (INTERESTING (agent V-1) (result V-2))))

                < check cardinalities--see if count is less than
                the cardinality constraint or if already stored >
    (IF (OR (c-1 < 1) (I-2 <> NULL)) THEN
        ((IF (I-2 = null) THEN      < if not already stored >
             ((I-2 <- (ICREATE INTERESTING))
                                               < perform update >
             (CHANGE INTERESTING / agent I-2 V-1)
             (CHANGE INTERESTING / result I-2 V-1)))

              <perform remainder of the update>
 .......
```

Data-level Cardinality Check
Figure 59

conjunction of all of the necessary pre-conditions

```
(and (<necessary pre-condition 1>)
     (<necessary pre-condition 2>)
        :         :         :
     (<necessary pre-condition n>))
```

of the situations that are involved in the insert. This conjunct is part of the *enquire expression of an update operation. As the simplification algorithm analyzes an *assert expression, it keeps track of the expression's necessary pre-conditions by listing them in the *enquire expression of the branch of the canonical form where the situation having the pre-condition was found. If duplicate expressions would occur in the *enquire expression for a branch, the duplicate is not listed. The merge algorithm, shown in Figure 60, handles adding expressions to the canonical form.

5.9.2 Required Pre-conditions

Update operations must meet any required pre-conditions as specified by required: slots in schema entries for both PRIMITIVE and non-PRIMITIVE situations being *asserted. The set of required pre-conditions for each of the sets of situations in a simplified expression is derived in the same manner as are its necessary pre-conditions with the exception that the required: schema slots rather than the necessary: slots are consulted.

For a REFLECT or REFLECT-NOT operation, the effect of required pre-conditions on the operation is the same as the effect of necessary pre-conditions on these operations. Therefore, when dealing with REFLECT and REFLECT-NOT operations, the simplification algorithm does not distinguish between these two types of pre-conditions. They both become part of the *enquire expression. The merge algorithm that joins two expressions is shown in Figure 60.

For an ASSERT or DENY operation, the data-level operations that will need to be performed in order to REFLECT its required pre-conditions include *assert as well as *enquire operations. These are exactly the same kind of operations that are needed to perform the main ASSERT operation. The difference between these operations lies solely in how required pre-conditions are analyzed at the SIDUR level. So, the simplification algorithm, when dealing with the analysis of the specified REFLECT on the required pre-conditions for an ASSERT or DENY operation, isolates the sub-expressions of the implied REFLECT operation. When the simplification algorithm encounters a PRIMITIVE or non-PRIMITIVE situation into whose extension values will be ASSERTed and whose schema specification includes a required: slot, it sigma binds the required: slot as described earlier in this section. The simplification algorithm is then called recursively to include the resulting REFLECT expression in the same *assert expression that is being built for the ASSERT or DENY operation. That is, the simplification algorithm analyzes the required: expression as if it is part of a REFLECT operation, but, by placing its results onto the same list structure that had been built for the ASSERT/DENY operation, causes the two operations to be correctly merged. This results in adding three groups of sub-expressions to the canonical list structure of the ASSERT operation.

First, the required: slot is REFLECTed. This results in adding the sigma-expanded required: slot to the *assert expression as shown below:

```
(and (<set of expressions in the ASSERT situation expression>)
     (<set of *assert expressions obtained by REFLECTing
       required pre-conditions>)).
```

Any TOKEN-type participants of the sigma-bound required: slot are also added to the *assert expression. Circularities among TOKEN-type definitions can arise when this is done and will be described in the next section. In order to prevent redundancies in the *assert expression, which will result in "circularities" in processing the operation, expressions that are duplicates or subsets of ones already on the list are not added. Further analysis of a duplicate new expression is discontinued because all of the information needed for the operation was obtained the first time the expression was put in the *assert expression. If a subset of the new expression is already on the list, the subset is removed and the new one is added. This is the merge operation shown in Figure 60.

The second group of sub-expressions that result from semantically REFLECTing a required pre-condition are obtained from any necessary: and required: slots found in situation definitions of the situations in the required: slots. These kinds of sub-expressions represent pre-conditions that must already exist in the database extension prior to any update for some situation. They are, therefore, semantically equivalent to the necessary pre-conditions of the main ASSERT or DENY operation. The simplification algorithm puts these sub-expressions on the same list that it is building for the *enquire expression for the original ASSERT or DENY operation.

```
(*enquire
     (and (<necessary pre-conditions of the original ASSERT>)
          (<necessary and required pre-conditions of the
            REFLECT of the required  pre-conditions>)))
```

The merge algorithm is used to join the expressions. The translation routines then can handle all of these pre-conditions as one expression to which the *enquire operator is applied. Note that there is a possibility that a duplicate expression might not have been added to the *assert expression because it had already been added by the original ASSERT operation. If the duplicate expression had a required pre-condition, the required pre-condition should not be put into the *enquire expression. This is because it has already been dealt with by the stronger ASSERT operator, which put it in the *assert expression. Therefore, putting it into the *enquire expression would cause incorrect results if tuples were erroneously eliminated from the update extension by not passing the *enquire pre-condition.

The third group of ASSERT / DENY sub-expressions obtained from the implied REFLECT of required pre-conditions are cardinality constraints. These are added to the list of cardinality constraints for the main operation if they are not already present.

```
Determine if the new situation is already listed
If an expression with the same situation name is listed then
   Begin
      Determine if the new expression is a duplicate of one
           already listed
      If it is a duplicate then
         Do not add it to the list
      Determine if the new expression is a subset of one
           already listed
                   (i.e. its participants are a
                    subset of those of an existing
                    expression involving this situation)
       If it is a subset then
          Do not add it to the list
       Determine if the new expression is a superset of one
            already listed
       If it is a superset then
          Begin
             Remove the old expression
             Add the new expression to the list
          End
  End
Else
   Add the new expression to the list
```

Algorithm for Merging Two Expressions
Figure 60

5.9.3 TOKEN Definitions

During an *assert operation, all TOKEN values that are being added to extensions of PRIMITIVE situations must also be added to the corresponding object-class defining situations. Because this addition to a defining situation is, itself, defined semantically as a REFLECT, the same policies and procedures govern this sub-operation. Therefore, it is observed that the REFLECT sub-operation on the TOKEN type's defining situation is simply an additional, but implied, pre-condition of the user's original operation. Within a REFLECT operation this is semantically equivalent to another REFLECT operation whose expression includes the conjunct of the user's original REFLECT expression and the expression needed to cause the REFLECT of the TOKEN's defining situation. For example,

```
(REFLECT (HAS-TITLE (agent C-21)
                    (object "CS-430")))
```

has an agent of type COURSE. The schema definition of COURSE indicates that its defining situation is IS-COURSE. This implies that

```
(REFLECT (IS-COURSE (agent C-21)))
```

will also be performed. These two operations can be combined to form a single equivalent operation.

```
(REFLECT (and (HAS-TITLE (agent C-21)
                         (object "CS-430"))
              (IS-COURSE (agent C-21)))).
```

It is important to note that if the

```
(REFLECT (IS-COURSE (agent C-21)))
```

update fails for any tuples, then these tuples must be removed from the REFLECT extension and not added to the other situation extensions mentioned in the *assert expression. However, if C-21 is already present in the IS-COURSE extension, the REFLECT operation of IS-COURSE should be considered as already successful and should not be repeated. Since the two REFLECTs are semantically equivalent to one REFLECT of the conjunct of their situation expressions, the REFLECT of the object-class defining situation is implemented by merging this implied REFLECT expression with the *assert expression. This is done when the simplification algorithm processes the *assert expression. However, if the implied REFLECT expression is a duplicate or a subset of an expression already listed in the *assert expression, the implied REFLECT is not placed on the list. If a subset of the implied REFLECT is already on the list, it is removed. Figure 60 shows the algorithm for merging the REFLECT expression with the implied REFLECT on the TOKEN types.

When performing the implied REFLECT on TOKEN types, it is possible to encounter a circularity in the schema definitions of the object type and of the defining situation. Example schema definitions are shown in Figure 61. For example, when a REFLECT operation involves IS-SEMINAR, the TOKEN type EVENT also needs to be REFLECTed into its defining situation IS-EVENT. But, note that the object type of the participant of the defining situation is the same type

```
object-class EVENT
     representative:  (TOKEN)
     definition:  (IS-EVENT)


situation IS-EVENT
     participants: ((agent x EVENT))
     definition:  (PRIMITIVE)
     extension: (CLOSED-WORLD)


situation IS-SEMINAR
     participants: ((agent x EVENT))
     definition:  (PRIMITIVE)
     extension: (CLOSED-WORLD)
```

Circular TOKEN-type Definitions
Figure 61

already involved in the implied TOKEN REFLECT. This type of circularity can also be indirect in that several situations and TOKEN types may be dependent on each other for their definitions. The simplification algorithm handles this type of circularity by making a list of TOKEN types being REFLECTed. When a duplicate is encountered, the cycle is broken, and the data value is scheduled for addition to all defining situations specified in the schema. In this example, the cycle is quickly stopped, and values would be scheduled for addition to the IS-EVENT situation.

For an ASSERT operation, the implied REFLECT expression for TOKEN types is also added to the *assert expression as shown in the example above. When the schema specifications related to the implied REFLECT are examined by the simplification algorithm, their effects on the canonical expression that is being built will be those defined for the REFLECT operator rather than those of the ASSERT operator. For example, if the schema specification of a TOKEN type's defining situation contains a required pre-condition, this pre-condition will be treated as part of the required pre-conditions for the implied REFLECT. For a REFLECT operation, required pre-conditions are semantically equivalent to necessary pre-conditions. Therefore, the required pre-conditions of the implied REFLECT operation are equivalent to the necessary pre-conditions of the original ASSERT operation and are listed with the necessary pre-conditions in the *enquire expression as shown below:

```
   (*enquire (<required pre-conditions of the REFLECT of the
                  defining situation of the TOKEN type >)
   (*assert (<defining situation of the TOKEN type>)>
```

Figure 62 shows an example of a defining situation with a required pre-condition. In order to ASSERT a situation having an object participant of type ARTICLE, the *enquire expression must include

```
                        (IS-SUBMITTED (agent x)),
```

and the *assert expression must include

```
                        (IS-ARTICLE (agent x)).
```

```
object-class ARTICLE
      representative:  (TOKEN)
      definition:  (IS-ARTICLE)


situation IS-ARTICLE
      participants: ((agent x ARTICLE))
      required: (IS-SUBMITTED (agent x))
      definition:  (PRIMITIVE)
      extension: (CLOSED-WORLD)
```

Defining Situation With Required Pre-condition
Figure 62

5.10 Bag Building Routines

The translation routines call SIDUR's code generation routines to build a Franz Lisp list structure that will represent the data-level query. The code generation routines consist of a set of routines that

1) initialize the BAGAL query or initialize a new BAG within the query,
2) add sub-lists to a BAG and return the BAG as value,
3) add statements to the BAGAL query, and
4) build sub-lists and return them for use by other routines that later add them to BAGs.

Many of the BAG building routines build small lists. Those routines are called by higher-level routines. By isolating these routines from the higher-level routines, changes can be made in the data-level language syntax without disturbing the translation routines.

5.11 Utility Routines

The utilities consist of a set of routines that print results of operations, save results, pre-define operations and expressions, and handle files containing pre-defined operations and expressions. These are all very simple Franz Lisp functions whose usage and results are described in the appendix titled "Use of the SIDUR Implementation."

VI. SCHEMA BUILDING

6.1 Schema and Data Equivalence

SIDUR users can access and update the schema in the same manner that data is accessed and updated. This symmetry between schema and data management enables the information system to provide users with a unified method of representation and information access. This equivalence is achieved by defining, for each type of SIDUR construct, a system situation whose participants are the slots for the construct. The role names for these situations are the relevant slot names for each construct type. Figure 63 shows the system situations whose extensions contain the SIDUR schema.

6.2 Run-time Schema Management

Since the OSIRIS data level is not yet available, the run-time schema for this implementation of SIDUR is stored in the form of Franz Lisp p-list structures. The schema information for each construct is stored on the p-list of the Franz Lisp symbol whose print-name is the construct name. The name of each slot in the construct schema is a property on the p-list, and the contents of the slot are the value of the property. Figure 64 shows the p-list data structure for the TEACHES-COURSE situation.

```
(situation IS-DATA-VALUE-CLASS
          participants:  ((agent a DATA-VALUE-CLASS)
                           (type: b DATA-VALUE-TYPE)
                           (form: c NAME-RULE)
                           (size: d UNSINT)
                           (minval: f NUMBER)
                           (maxval: e NUMBER)
                           (precision: g UNSINT))
          definition:  (SYSTEM))

 (situation IS-OBJECT-CLASS
          participants:  ((agent a OBJECT)
                           (representative: b DATA-VALUE-CLASS)
                           (superclasses: c OBJECT-LIST)
                           (names: d STRING-LIST)
                           (definition: e SITUATION))
          definition:  (SYSTEM))

 (situation IS-SITUATION
          participants:  ((agent a SITUATION)
                           (participants: b PARTICIPANT-LIST)
                           (cardinalities: c CARDINALITY-LIST)
                           (extension: g EXTENSION-DESCRIPTOR)
                           (necessary: e SIGMA-EXPRESSION)
                           (required: f SIGMA-EXPRESSION)
                           (definition: d SIGMA-EXPRESSION)
                           (sufficient: e SIGMA-EXPRESSION))
          definition: (SYSTEM))

 (situation IS-COMPUTATION
          participants:  ((agent a COMPUTATION)
                           (participants: b PARTICIPANT-LIST)
                           (definition: c SIGMA-EXPRESSION))
          definition:  (SYSTEM))

 (situation IS-ACTION
          participants:  ((agent a ACTION)
                           (participants: b PARTICIPANT-LIST)
                           (prerequisites: c SIGMA-EXPRESSION)
                           (results: d OPERATION-LIST))
          definition:  (SYSTEM))
```

System Situations
Figure 63

```
TEACHES-COURSE
   (construct-type situation
    participants:  ((agent x INSTRUCTOR) (object y COURSE))
    definition:
       (and (TEACHES-OFFERING (agent x) (object z))
            (OFFERING-OF (agent y) (object z))))
```

Schema Storage of TEACHES-COURSE
Figure 64

6.3 Initializing a Schema

One of the long-range design goals of the OSIRIS architecture is to support the automated generation of database schemas. However, in the meantime in order to facilitate independent development of this phase of the SIDUR implementation, a set of batch schema input routines and a simplified schema BNF have been developed. If the user wishes to establish an entire schema, he/she may use any standard text editor to create a schema file in which schema constructs are defined using a notation very similar to that established in the "SIDUR Manual" (Freiling, 1983c.) In addition to the departures from the "SIDUR Manual" BNF for defining schema constructs, the implementation described in this thesis does not verify the syntactic or semantic soundness of a schema. Appendix G contains the BNF used in this SIDUR implementation.

After a text file containing the schema has been prepared, the user should enter the SIDUR system as outlined in Appendix A. In response to the initial SIDUR prompt, the user should type the word 'new'. The response to the next prompt should be the name of the file containing the text of the schema. After the third prompt, the user should choose and type the name of the file that will contain the run-time schema p-list structure. The run-time schema will be stored into this file when the user terminates the SIDUR session by typing '(wrap-up)'.

6.4 Schema Operations

Since the user can express schema operations in the same language in which data operations are expressed, SIDUR must intercept the schema operations and process them at the SIDUR level. With one exception noted below, data-level queries are not created for operations that mention any of the system situations listed in Figure 63 or that mention the situations MEMBER and MENTIONS. A restriction of this implementation, but not of the SIDUR model itself, is that operations involving the schema may not include data accesses or updates and may not refer to more than one situation.

The user can access information about the schema by using the SIDUR situation operator ENQUIRE in an operation such as

```
        (ENQUIRE (IS-SITUATION (name TEACHES-COURSE)
                               (definition: x))).
```

Quotation marks, which are used by the Franz Lisp read functions to denote Lisp strings, are not used to surround the constant data values such as TEACHES-COURSE in operations that mention the system situations. Operations of this type should always include the name participant and any slot names about which information is being requested. The role 'name' is converted to 'agent' by SIDUR.

The situation MEMBER is implemented as discussed in the "SIDUR Manual" (Freiling, 1983c.) MEMBER tests for construct type membership. For example,

```
            (ENQUIRE (MEMBER (agent "TEACHES-COURSE")
                             (object "SITUATION")))
```

will return a true or false value indicating whether TEACHES-COURSE is a situation name. If the agent participant of this query is an object TOKEN and if the object participant is the name of an object class of type TOKEN, then SIDUR will prepare a data-level query to determine whether the TOKEN value is stored in the defining situation of the object class. The mechanism by which this is accomplished is an ENQUIRE operation on the defining situation.

The MENTIONS situation will return the list of construct names that are mentioned in the indicated slot of the situation named in the operation. For example,

```
            (ENQUIRE (MENTIONS (name "TEACHES-COURSE")
                               (object "definition:")
                               (value x)))
```

will return a list of construct names found in the definition: slot of TEACHES-COURSE. This implementation corresponds to a definition of MENTIONS from in an early version of the SIDUR definition and does not correspond with the current SIDUR definition of MENTIONS.

The user can update the run-time schema file (but not the original user-created schema file) by using the REFLECT and REFLECT-NOT operators. An operation such as

```
            (REFLECT (IS-SITUATION (name TEACHES-COURSE)
                                   (definition:  (PRIMITIVE)))
```

will revise the definition: slot of the TEACHES-COURSE situation. Again, double quotes around the schema slot values should not be used, and these operations may involve only one situation.

VII. SUMMARY

This paper has discussed some of the major concerns of implementing a semantic-level information system. Some of the major concerns include:

1) the advantage of translating a semantic operation into only one data-level operation,

2) providing for operations on single tuples as well as on an extension of tuples,

3) handling disjunctive update operations,

4) insuring the integrity of stored data by enforcing pre-conditions and cardinality constraints,

5) insuring that updates are atomic, and

6) removing semantic concerns from data-level queries while yet expressing in data-level terms what needs to be done to bring about the requested semantic-level result.

The SIDUR implementation has approached each of these concerns in a manner designed to allow flexibility and evolvability of the system as the research needs of the OSIRIS project evolve. The OSIRIS project has reached many of its preliminary design goals and is now ready to refine and test the models and interfaces that have been developed. For the SIDUR portion of the project, the next steps will be:

1) tuning the usability of the SIDUR model,

2) interfacing with the data level,

3) integrating with the user interface module, and

4) using the OSIRIS architecture as the vehicle for continued study of semantic-level issues.

BIBLIOGRAPHY

1. Abrial, J. R. Data semantics. In Database Management, J. W. Klimbie and K. L. Koffeman, Eds. North-Holland Pub. CO., Amsterdam, 1974.

2. ANSI/X3/SPARC (Standards Planning and Requirements Committee). Interim report from the study group on database management systems. FDT (Bulletin of ACM SIGMOD) 7, 2 (1975).

3. Bracchi, G., Paolini, P., and Pelagatti, G. Binary logical associations in data modelling. In Modelling in Data Base Management Systems, G. M. Nijssen, Ed. North-Holland Pub. CO., Amsterdam, 1976.

4. Bracchi, G. Methodologies and tools for logical database design. In Data Base Management: Theory and Applications, C W. Holsapple and A. B. Whinston, Eds. D. Reidel Pub. CO., Holland, 1982.

5. Buneman, P., and Frankel, R. E. FQL--A functional query language. In Proc. ACM SIGMOD Int. Conf. Management of Data, Boston, Mass., 1979.

6. Cattell, R. G. G. Design and Implementation of a Relationship-Entity-Datum Data Model. Xerox Corporation, Palo Alto, Ca., 1983.

7. Chen, P. P. S. The entity-relationship model: Toward a unified view of data. ACM Trans. Database Syst. 1, 1 (March 1976), 9-36.

8. Clocksin, W. F., and Mellish, C. S. Programming in Prolog. Springer-Verlag, Berlin, 1981.

9. Codd, E. F. Extending the database relational model to capture more meaning. ACM Trans. Database Syst. 4, 4 (Dec. 1979), 397-434.

10. Codd, E. F. Further normalization of the data base relational model. In Database Systems, Courant Computer Science Symposia 6, R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, N.J., 1971.

11. Codd, E. F. A relational model for large shared data banks. Commun. ACM 13, 6 (June 1970), 377-387.

12. Date, C. J. An architecture for high level language database extension. In Proc. ACM SIGMOD Int. Conf. Management of Data, Washington, D.C., 1975.

13. Date, C. J. An introduction to database systems, Vol. 2. Addison-Wesley Publishing Company, Reading, Mass., 1983.

14. Foderaro, J. L., and Skiower, K. L. The FRANZ LISP Manual. Regents of the University of California, Sept. 1981.

15. Freiling, M. J. "OSIRIS" Project. OSIRIS Internal Document, Oregon State University, Corvallis, Or., 1983a.

16. Freiling, M. J. SIDUR - An integrated data model. In Proc. Comp. Soc. Int. Conf., Arlington, Va., Sept. 1983b.

17. Freiling, M. J., et. al. The SIDUR 2.0 Reference Manual. Project OSIRIS Internal Document, Oregon State University, Corvallis, Or., 1983c.

18. Freiling, M. J. Understanding Database Management. Alfred Publishing Co., Sherman Oaks, Ca., 1982.

19. Hammer, M., and McLeod, D. The semantic data model: A modelling mechanism for database applications. In Proc. ACM SIGMOD Int. Conf. Management of Data, Austin, Tex., 1978.

20. Kent, W. Data and Reality. North-Holland Pub. CO., Amsterdam, 1978.

21. Kogan, D. SIDUR--A Formalism for Structuring Knowledge Bases.. Masters Thesis, Dep. Computer Science, Oregon State University, Corvallis, Or., 1984.

22. Kogan, D. D., and Freiling, M. J. SIDUR - A structuring formalism for knowledge information processing systems. In Proc. Int. Conf. Fifth Generation Computer Systems, Tokyo, Japan, Nov. 1984.

23. Mylopoulos, J., Bernstein, P. A., and Wong, H. K. T. A language facility for designing interactive database-intensive applications. In Proc. ACM SIGMOD Int. Conf. Management of Data, Austin, Tex., 1978.

24. Mylopoulos, J., and Wong, H. K. T. Some features of the taxis data model. In Proc. of the 6th Int. Conf. Very Large Databases, Montreal, Canada, Oct., 1980.

25. Nijssen, G. M. A gross architecture for the next generation data base management systems. In Modelling in Data Base Management Systems, G. M. Nijssen, Ed. North-Holland Pub. Co., Amsterdam, 1976.

26. Senko, M. E., et. al. Data structuring and accessing in data-base systems. IBM Syst. J. 12, 1 (1973), 30-93.

27. Senko, M. E. DIAM as a detailed example of the ANSI/SPARC architecture. In Modelling in Data Base Management Systems, G. M. Nijssen, Ed. North-Holland Pub. Co., Amsterdam, 1976.

28. Shipman, D. W. The functional data model and the data language DAPLEX. ACM Trans. Database Syst. 6, 1 (March 1981), 140-173.

29. Smith, J. M., and Smith, D. C. P. Database abstractions: Aggregation. Commun. ACM 20, 6 (June 1977), 405-413.

30. Smith, J. M., and Smith, D. C. P. Database abstractions: Aggregation and generalization. ACM Trans. Database Syst. 2, 2 (June 1977), 105-133.

31. Tsichritzis, D., and Lochovski, F. Data Models. Prentice-Hall, Englewood Cliffs, N.J. 1982.

32. Ullman, J.D. Principles of Database Systems. Computer Science Press, Potomac, Md., 1980.

33. UNIX Time-Sharing System: UNIX Programmer's Manual, 7th Ed., Vol. 2A. Bell Telephone Laboratories, Inc., Murray Hill, N. J., Jan. 1979.

34. UNIX Programmer's Manual, 7th Ed., Virtual VAX-11 Version. Computer Science Division, Department of Electrical and Computer Science, University of California, June 1981.

35. Warren, D. H. D. Logic Programming and compiler writing. Software-Practice and Experience, 10 (Feb. 1980), 97-125.

APPENDICES

Appendix A. Use of the SIDUR Implementation

UNIX®

SIDUR is implemented on the Oregon State University Computer Science Department VAX-11/750, which runs Berkeley VAX/UNIX (4.1bsd revised 1 Sept. 1981). The reader is referred to the standard UNIX documentation (UNIX, 1979; and UNIX, 1981) if further information is needed. This document will provide the reader with enough detail about the UNIX environment to enable him/her to do the basic functions needed in using the SIDUR system.

Login Procedure

The procedure for logging in is:

1) Obtain <account name> and <password>.

2) Type the 'break' key until the login prompt appears.

3) When the login prompt appears, type the <account name>.

4) When the password prompt appears, type the <password>.

5) When the '(VT100)' prompt appears, type the 'carriage return' key.

6) Type 'lisp'.

7) The Franz Lisp and SIDUR environment will be initialized and ready for the user to begin a session.

Starting a Session

SIDUR will prompt the user for the database name that will be used to perform subsequent data and schema operations. The user will usually supply the name of an existing database that he wishes to access. If a new database is to be created at this time, the user will need to specify a name for it. Database names must be unique from any other name in the UNIX directory. SIDUR will then prompt the user to supply a UNIX file name to store the results of operations. This file name will be handled as a Franz Lisp atom. The form of the name can be any valid UNIX file name. A sample of a SIDUR session is shown in Appendix D.

Lisp

Because the SIDUR implementation is a set of Franz Lisp (OPUS 38.22) functions, the user who is knowledgeable of Lisp may find it helpful to refer to "The Franz Lisp Manual" (Foderaro and Skiower, 1981) for further detail if unusual circumstances arise. Enough detail will be provided here to enable the user to handle ordinary events and to recover from typical errors. An ordinary SIDUR session will consist of a series of prompts 'sidur-->' to the user from the Franz Lisp driver function. However, the user who is knowledgeable of Lisp may wish to leave SIDUR in order to perform Lisp functions. This can be done by typing '(lisp)' in response to the SIDUR prompt. The user can return from Franz Lisp to the SIDUR system by typing '(sidur)'. In addition, the user may leave the Franz Lisp-SIDUR system in order to execute UNIX commands. This can be done by typing CONTROL-Z. Re-entry to the Franz Lisp-SIDUR system can be accomplished by typing 'fg' in response to the UNIX prompt.

Data Manipulation Operations

After a session has been started, the user will be presented with the SIDUR prompt 'sidur-->'. In response, the user can type in a SIDUR operation. If the operation is ill-formed, SIDUR will respond with an error message. Otherwise SIDUR will respond with an indication that the operation has been processed. In either case, SIDUR will enter the Franz Lisp break package whose prompt is '<1>:'. The user can then perform any Franz Lisp function or any of the SIDUR utilities mentioned later in this section. If the user types '(pprint query)', the BAGAL translation of the operation will be displayed on the screen. When the user is ready to resume, the response to the break package prompt should be '(return t)'.

Saving Results and Getting Output

The SIDUR implementation includes a utility routine that allows the user to save the results of the most recent query by typing:

```
(pprint query %p)
```

after the SIDUR-break prompt occurs. A user who is familiar with Lisp and with the SIDUR functions may wish to invoke the pprint function to print other values. '%p' denotes the port (file) that was opened for the user when the SIDUR environment was initialized. Other files may be opened by typing

```
(setq <port name> (outfile <file name>)).
```

Then, subsequent statements of the form

```
(pprint <variable name> <port name>)
```

will cause values to be written to the file. <file name> will be a file name on the UNIX current directory and should be distinct from any other names in the directory. Any new ports opened by the user in this manner should be closed by typing

(close <port name>)

prior to leaving the SIDUR environment. '%p' will be closed automatically by SIDUR when the user invokes the '(wrap-up)' function.

In order to obtain a line printer listing of the result file, the user will need to leave the SIDUR environment either temporarily by typing CONTROL-Z or permanently by typing '(exit)'. When at the UNIX command level, the user can type

lpr <filename>

to obtain a line printer listing.

Pre-Defined Operations and Sigma Expressions

The user may pre-define operations and sigma expressions using a standard text editor. Operations can be defined by using the 'valdef' utility function:

```
    (valdef <op1>
            (ENQUIRE (TAKES-COURSE (agent x) (object y)))).
```

<op1> can be typed in response to the SIDUR prompt and its corresponding operation will be performed. Sigma expressions can be defined in the same manner:

```
    (valdef <expr1>
            (TAKES-COURSE (agent x) (object y))).
```

If '(ENQUIRE <expr1>)' is typed in response to the SIDUR prompt, the ENQUIRE operator will be applied to the sigma expression of <expr1>.

A file that contains pre-defined operations and sigma expressions can be read into the SIDUR environment by typing

(read-in-operations).

The user will be prompted to type the name of the file in which the operations are saved. The SIDUR utility routine

(save-next-operation)

will add an operation to the file that was read in by '(read-in-operations)'. This command will prompt the user to provide a name for the operation or expression and then to type it in. The revised operation file may be saved by typing

(store-operations).

Batch Entry

A batch entry function called 'do-queries' is provided so that pre-defined operations can be processed. The user should type '(do-queries)' in response to the SIDUR prompt. The user then will be prompted to provide the name of the file that contains the pre-defined operations. If the file has not yet been loaded, the user will be prompted to type in '(loadf <file name>)'. Then the user will be asked whether all operations in the file should be processed or whether he/she should be prompted to decide individually whether each operation should be done. The user will also be asked whether the results of each operation should be displayed on the terminal and whether the results of each operation should be written out to the result file. The user should respond with 'y' for yes or 'n' for no to each of these last prompts.

Recovery

Since SIDUR is running in the Franz Lisp environment, various user terminal entry errors or misuse of SIDUR may invoke the Lisp break package. If this occurs the prompt '<1>:' will appear on the screen along with a description of the error. If the user is able to identify and correct the error, he/she can re-enter SIDUR and try the operation again. In order to do this, CONTROL-D should be typed in order to leave the Lisp break package. Then the user should type '(sidur)' to re-enter the SIDUR System. The SIDUR prompt will re-appear, and the user can re-try the transaction. If the user does not wish to continue the session, the normal procedure for ending a session should be followed. That is, '(wrap-up)' can be invoked from Franz Lisp as well as from within SIDUR. Occasionally, the user may wish to suspend a SIDUR session and then to resume later in exactly the same state. This may be helpful in order to temporarily exit Franz Lisp to do such things as make a back-up copy of the database or get a line printer copy of output. The UNIX system provision for temporarily halting a job may be used by typing CONTROL-Z. When the user is ready to re-enter SIDUR, he/she may type 'fg'. If the user wishes a longer suspension of the session, he/she may type '(dumplisp <file name>)'. This command will cause a file to be created on the SIDUR account and to store in it the entire contents (core dump) of the current Franz Lisp-SIDUR environment. The user may log off or perform other UNIX jobs as needed. Later, SIDUR can be re-entered when the user is at the UNIX command level by typing '<file name>' in response to the UNIX prompt.

Ending a Session

In order to end a SIDUR session, the user should type the command '(wrap-up)'. 'wrap-up' is a Franz Lisp function that will update and close the file that contains the database. The user will be asked whether the Franz Lisp schema should be stored in pretty-printed form or not. The user should respond by typing 'pretty' or 'not.' It will then close the result file and cause the Franz Lisp environment to be exited. In order to log off Unix, the user should type 'logout'.

Appendix B. VAX® Environment

List of Modules

Module	File	Main Routines
Main Control	q-interp	sid
Syntax Checking	syn-check	dml-command? <others individually>
Operation Preparation	q-interp	make-vars-unique find-constants
Expression Simplification	q-interp	flatten-expr
Object-checking	obj-check	objects-in-prescribed-class
Translation	q-interp	<correspond to name of the operator>
BAG Building	bag-p	<called individually by translation routines>
Utilities	util	<called individually by user>
Schema Operations	sc-ops	<called individually by translation or initialization routines>
Initialization Wrap-up	read-db	read-db initialize-schema wrap-up

List of Global Variables

Variable	Purpose
query	The data-level query
db-filename	The stored database file name
db-schema	The name of the schema file
result-file	The name of the output file
%p	The name of the port corresponding to result-file
opfile	The name of the operation file
constant-bag	Holds the extension of constants
list-of-slot-names	Valid slot names

```
<any pre-defined operation or expression>

Lists of schema construct names:
      list-of-data-value-names
      list-of-object-names
      list-of-situation-names
      list-of-computation-names
      list-of-action-names
      list-of-protected-system-sits
      list-of-construct-situations
```

Appendix C. Algorithm and Data Structures

Driver Routines

Description: The driver routines control the user's interaction with SIDUR and call the translation routines.

Data Structure: The lists representing the SIDUR operation and the BAG query are the data structures.

Algorithm:

```
  'sid'

     Set up schema--Call 'initialize-schema'
     Set up result file
     Call user routine 'sidur'

  'sidur'

     Prompt user to type in the operation
     Read in a list structure representing the operation
     If a utility was typed, call the corresponding routine
     Else call 'translate'

  'translate'

     Initialize query
     If operation is pre-defined, get its value
     Generate unique variables--Call 'make-vars-unique'
                                     'find-constants'
     Call a translation routine corresponding to the
       operator of this operation
```

Translation Routines

Description: There is a translation routine corresponding to each of the SIDUR operators.

Data Structure: The lists representing the operation, the canonical form of the operation, and the BAG query.

Algorithm:

```
'enquire'

    If the expression begins with a connector or a construct name
      Begin
      Call 'flatten-expr' to get the canonical form
      Create a data access BAG for each of the sets of situations
          to query situations not enclosed by a 'not' connective
      Create BAGs to merge the prior results with the BAGs
          representing 'empty' expressions from each branch
      Create a data access BAG for each of the sets of situations
          to access situations enclosed by the 'not' connective
      Create a BAG for each of the sets of situations to perform
          set subtraction (MINUS) on the prior two results
      Call 'handle-computations' to create computation BAGs and
          to merge the results of the computations with the
          extension of the MINUS BAG
      Make the format list of the final BAG
      End
    Else if the expression begins with the 'empty' connective
      Begin
      Call 'enquire' to translate the expression within 'empty'
      Create a BAG to indicate the FULL or EMPTY condition of the
          extension computed by the 'enquire'
      End
    Else if the expression is a closed sigma expression
      Begin
      Call 'enquire' to translate the expression
      Make the format list correspond to the sigma variables
      End
    If there is an input extension, merge it with the result
```

```
'check'
   Call 'enquire' with the expression as parameter
   Put a limit of 1 tuple on the result
   Create a BAG to indicate the FULL or EMPTY condition of the
       extension computed by the 'enquire'

'reflect'

   Call 'update-operation' with REFLECT and the sigma expression
        as parameters
   If a closed sigma expression, make the format list

'reflect-not'

   Put 'not' in front of the expression
   Call 'update-operation' with REFLECT and the expression
        preceded by 'not' as parameters
   If a closed sigma expression, make the format list

'assert'

   Call 'update-operation' with ASSERT and the sigma
        expression as parameters
   If a closed sigma expression, make the format list

'deny'

   Put 'not' in front of the expression
   Call 'update-operation' with ASSERT and the expression
        preceded by 'not' as parameters
   If a closed sigma expression, make the format list

'perform'

   Determine that the binding tuple is complete
   Determine that the object classes are correct
   Call 'enquire' on the sigma-bound prerequisites: slot
   Call 'reflect' on the sigma-bound results: slot

'permit!'

   Determine that the binding tuple is complete
   Determine that the object classes are correct
   Call 'assert' on the sigma-bound prerequisites: slot

'permit?'

   Determine that the binding tuple is complete
   Determine that the object classes are correct
   Call 'check' on the sigma-bound prerequisites: slot
```

```
'since'

   Create a BAG to hold the extension of constant values
   Call 'reflect' on the domain sigma expression
   Create a BAG FAIL operation if the resulting extension
       is EMPTY
   Call the appropriate translation routine to translate each
       simple operation using as input the result of the
       'reflect' on the sigma expression
   Create a BAG to OR-MERGE the results of each operation

'for'

   Create a BAG to hold the extension of constant values
   Call 'enquire' on the domain sigma expression
   Create a BAG FAIL operation if the resulting extension
       is EMPTY
   Call the appropriate translation routine to translate each
       simple operation using as input the result of the
       'enquire' on the sigma expression
   Create a BAG to OR-MERGE the results of the simple
       operations

'create'

   Create a BAG to OCREATE the object token

'destroy'

   Create a BAG to DESTROY the object token

'update-operation'

   Call 'flatten-expr' to get the canonical form
   Create BAGs to form the extension of constants in the
       operation and AND-MERGE with the input BAG
   Create the BAG to prompt the user to choose which branch
       is to be used to update each tuple in the input extension
   Create BAGs to split the input extension into a separate
       extension for each branch
   Call 'enquire' to create BAGs to hold the extensions of
       tuples that meet the necessary pre-conditions related
       related to each branch
   Create AND-MERGE BAGs to determine which of the input
       tuples meet the *enquire pre-conditions
   Create update BAGS for each of the sets of situations that
       check cardinality constraints, insert tuples if they are
       not already stored, delete tuples if they are stored, and
       handle open-world updates
   Create a BAG to OR-MERGE the result from each branch
   Make the format list of the final BAG
```

Expression Preparation Routines

Description: The preparation routines locate constants and variables in the operation and generate unique variables for each. Variables are replaced by their unique corresponding variables ('make-vars-unique',) and unique variables are inserted in front of each occurrence of a constant ('find-constants'.)

Data Structure: The list representing the SIDUR operation.

Simplification Routines

Description: Determine the canonical form of the operation.

Data Structure: The lists representing the SIDUR operation, the canonical form of the operation, and the p-lists of the schema constructs.

Algorithm:

```
'flatten-expr' (alias 'f-e')

    If ((the expression begins with 'or') and
         (the expression is not enclosed by the 'not' connective))
       or ((the expression begins with 'and') and
            (the expression is enclosed by the 'not' connective))
      Make a list structure for each branch of the disjunctive
      Call 'f-e' to put the needed construct names on each list
    If ((the expression begins with 'and') and
         (the expression is not within a 'not'))
      or ((the expression begins with 'or') and
           (the expression is within a 'not'))
      Call 'f-e' for each branch to put the needed
          construct names on the path
      Take the union of the resulting paths

    If the expression begins with 'not', toggle the 'not-on' flag
      Call 'f-e' to handle the enclosed expression

    If the expression begins with 'empty', add the 'empty'
        expression to the 'empty' portion of the path
```

```
If the expression begins with a PRIMITIVE situation
   Call 'augment-path' to add construct names to the path
        based on the schema specification of the situation

If the expression begins with a non-PRIMITIVE situation
   Call 'augment-path' to add construct names to the path
      based on the schema specification of the situation
   Call 'f-e' on the sigma-bound sufficient: slot or
      on the sigma-bound definition: slot

If the expression begins with a SYSTEM computation, add the
    computation to the 'comp' portion of the path

If the expression begins with a non-SYSTEM computation
   Call 'f-e' on the sigma-bound definition: slot
```

'augment-path'

```
If the operation is an update, check the STRING object form:
    definitions and minval: and maxval: slots for numbers
If the situation is PRIMITIVE or SYSTEM, add the expression
    to the portion of the branch that represents the
    extension of the operation ('defn')
If the situation has a necessary: slot
   and the operation is an update
   add the sigma-bound necessary: slot to the 'nec' portion
      of the path
If the situation has a required: slot
   and the operation is reflect:
   Add the sigma-bound required: slot to the 'nec' portion
If the situation has a required: slot
   and if the operation is 'assert'
   Call 'f-e' as if for a 'reflect' operation on the
       sigma-bound required: slot
If the situation has cardinality constraints involving one or
   more of the participants in this operation
   and the operation is an update
   Re-phrase each constraint into a situation expression
   Add the cardinality expression to the 'card' portion
      of the path
If the situation involves REFLECTing TOKEN object types
   Re-phrase each TOKEN definition: slot into a situation
       expression
   Call 'f-e' on the sigma-bound expression with REFLECT,
       the TOKEN sigma expression, and the current branch
       as parameters
```

´sigma-bind´

```
Associate each of the expression participants with the
    internal variable of its corresponding role in the
    list of participants
Replace each internal variable in the definition: slot with
    its corresponding participant
Make list of un-associated internal variables in the
    expression
Generate a unique variable for each un-associated variable
    and replace each internal variable with its new unique
    variable
Generate a unique variable for each constant in the
   definition: slot and insert the variable in front of
   the constant
```

Computations

Description: The main computation routine is called once by the ´enquire´ function to translate all of the computations required by the operation. The BAGs containing the results of all of the data accesses for all of the branches are used to produce an extension that will be passed as the input to the computation BAGs. As each of the computations is translated, the BAG identifier of the BAG containing the results of the previous computations and data accesses is passed as input to it. Computations are translated in the order in which they are found in the expression unless the result of a computation is a participant to another computation in which case the computation whose result will be the argument is translated first. After all of the computations are translated, the results are merged with the extensions of the data access BAGs to produce the final extension specified by the operation.

Data Structure: The computation routines take as input the list structure that represents the canonical form of an expression and produce as output elements of a BAG query.

Algorithm:

```
 'handle-computations'

    Form extension from results of data accesses
    Translate the computations for each of the
       branches--Call 'do-computations'
    Form final resulting extension by merging data
      access BAGs and computation BAGs

 'do-computations'

    Translate each computation 'do-comp'

 'do-comp'

    Determine whether this computation needs as
       argument the result of an untranslated computation.
    Translate the computation--Call 'enquire-comp-bag'

  'enquire-comp-bag'

     Translate BAGs to compute the value of each participant
     Translate a BAG to perform the computation
```

Object Checking Routines

Description: The object checking routines determine whether a constant value of type STRING meets the corresponding form: slot specification. They are called recursively to determine whether each character in the constant is valid as the next character. Since STRINGs may contain variable numbers of elements, the routines must back-up to the previous variable element if failure occurs. Back-up occurs as long as there is any valid combination of variable elements as yet untried. Other object checking routines verify the minval: and maxval: slots of INTEGER and REAL types and also that they are numerical types.

Data Structure: The constant data value is represented as a Franz Lisp list of symbols. Each symbol corresponds to a letter in the data value. A pointer to the current character is kept as well as pointers to each character location that may be backed-up to.

Syntax Checking Routines

Description: These routines determine whether an operation is syntactically correct. They correspond closely to the SIDUR BNF. Some of the routines also provide simple list structure access to improve readability of the source code.

Data Structure: The list representing an operation.

BAG Building Routines

Description: The BAG building routines are a set of very short list building routines. They include functions that build lists to represent the various data-level language features.

Data Structure: The data-level query is represented by a Franz Lisp list structure. Each element of the list is a sub-list whose structure corresponds to the BAGAL BNF.

Initialization and Wrap-up Routines

Description: These routines initialize a new database schema, read in a schema already initialized, and finish a session.

Data Structure: The lists representing the user's schema text file, the text file of system schema constructs, the initialized schema, and lists of construct and role names.

Algorithm:

```
'initialize-schema'

   Read in the user's schema text file
   Define the p-lists for each construct read in
   Read in the system schema text file
   Define the p-lists for each construct read in

'read-db'

   Read in the file containing an initialized schema

'wrap-up'

   Write the current schema to the database file
   Close files
```

Schema Operations

Description: During initialization, these routines set up a construct specification by re-phrasing the user's STRING form: slots into an internal representation and by setting up schema specifications for open-world situations. These routines also include functions defined in a prior version of SIDUR: 'define', ' mentions','undefine', 'establish', 'disestablish', and 'mentioned-by'.

Data Structure: The lists representing the user's schema and the lists representing the internal form of the schema.

Utilities

Description: The utilities are a set of routines that provide file storage and printing capabilities. The 'pprint' routine is a slightly altered version of the SIDUR 'pprine' routine.

Data Structure: The lists representing the SIDUR operation, pre-defined operations and expressions, and the BAG query.

Appendix D. A Sample User Session

```
 Oregon State University(VAX 11/750 + 4.1 BSD UNIX)

login: shirl                          <account name>     <---- user
Password:                             <password--no echo>
Last login: Mon Jul 25 18:52:16 on tty01
Switching to new tty driver...
TERM = (vt100)                         <user typed carriage return>
Erase is control-H
Kill is delete
users:
meagher regan shirl shirl shirl uy youfengw zhu
1:lisp                                                   <---- user
Franz Lisp, Opus 38.22

FILES LOADED ---->

     fl.util
     fl.pprint
     fl.edit
     fl.matcher

     read-db
     util
     q-interp
     comps
     syn-check
     obj-check
     bag-p
     sc-ops
     ops-in
     exfile

|type in database name-->  or type new-->|stored-s       <---- user
|type in result filename-->|rfile                        <---- user
sidur?-->s2                                              <---- user
finished

Break '|type (return t) when ready|
<1>: (pprint query)                                      <---- user
((comment Begin--------------------- q-0004)
 (comment ******* (-------> s2 <-------) |******* comment end|)
 (comment
     *******
     (ENQUIRE (MAY-TAKE (agent V-0006 S-1018) (object y-0005)))
     |******* comment end|)
```

```
  ((BAG b-0008 (y-0005 V-0006))
   <-
   (ACCESS
       (V-0006 = S-1018)
       (MAY-TAKE / agent I-0009 V-0006)
       (MAY-TAKE / object I-0009 y-0005)))
  (RETURN (BAG b-0008 (y-0005 V-0006))))
DONE
<1>: (return t)                                             <---- user
sidur?-->(DENY (IS-COURSE (agent C-12345)))                 <---- user
finished

Break '|type (return t) when ready|
<1>: (pprint query)                                         <---- user
((comment Begin---------------------- q00010)
 (comment
     *******
     (DENY (IS-COURSE (agent V-0011 C-12345)))
     |******* comment end|)
 (comment ******* DENY |******* comment end|)
 (comment ******* ASSERT |******* comment end|)
 (comment ******* |constant tuples bags| |******* comment end|)
 ((BAG B-0012 (V-0011))
  <-
  (TUPLES ((V-0011 <-  C-12345))))
 (comment ******* |bags to do cardinality precheck and
                   update operations    |******* comment end|)
 ((BAG B-0013 (V-0011 I-0014))
  <-
  (FOR
      (BAG B-0012 (V-0011))
      ((I-0014 <- (CINDEX (IS-COURSE (agent V-0011))))
       (IF (I-0014 <> null) THEN
           ((DELETE IS-COURSE / agent I-0014))))))
 (RETURN (BAG B-0013 (V-0011))))
DONE
<1>: (pprint query %p)                                      <---- user
DONE
<1>: (return t)                                             <---- user
sidur?-->(wrap-up)                                          <---- user
|type in pretty or not-->|pretty                            <---- user
153.5u 16.5s 6:36 42% 76+387k 242+57io 193pf+0w
2:lpr rfile                                                 <---- user
3:logout                                                    <---- user
/csm/shirl/.delfiles/: No such file or directory

Reverting to old tty driver...
Oregon State University(VAX 11/750 + 4.1 BSD UNIX)
```

Appendix E. SIDUR BNF

BNF syntax for SIDUR 2.0

```
        .        = delimiter (space , cr , tab , etc.)
        [x]      = optional element in expansion
        {x}*     = 0 or more repetitions of x
        {x}+     = 1 or more repetitions of x
        {x}'d'*  = 0 or more repetitions of x separated by d
        {x}'d'+  = 1 or more repetitions of x separated by d
        ']'      = BNF character used literally
        '''      = quote character used literally
        ;        = remainder of line is comment
```

General Syntactic Entities

```
<uc letter> ::=
        A | B | C | D | E | F | G | H | I | J | K | L | M |
        N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<lc letter> ::=
        a | b | c | d | e | f | g | h | i | j | k | l | m |
        n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::=
        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<unsint> ::=
        { <digit> }+

<int> ::=
        <unsint> | - <unsint> | + <unsint>

<real> ::=
        <int> '.' | <int> '.' <unsint>

<fpn> ::=
        <real> E <int>
```

```
<sp char> ::=
        # | ! | ? | _ | & | % | - | '*' | '+' | $ | "" | ''' |
        ( | ) | ' ' | :

                              ; The two double quotes are a syntactic
                              ; device to designate one double quote.

<letter> ::=
        <uc letter> | <lc letter>

<character> ::=
        <letter> | <digit> | <sp char>

<simple variable> ::=
        <lc letter>                     ; This could be expanded.

<variable> ::=
        <letter> - [ <unsint> ]

<role> ::=
        agent | object | value | source | destination |
        time | location | { <lc letter> }+

<computation role> ::=
        <domain role> |
        <measure role> |
        <mapping role> |
        <result role>

<domain role> ::=
        domain | domain- <unsint>

<measure role> ::=
        measure | measure- <unsint>

<mapping role> ::=
        mapping | mapping- <unsint>

<result role> ::=
        result

<identifier> ::=
        { <identifier segment> } - unsint

<identifier segment> ::=
        { <uc letter> }+

<string> ::=
        "" { <character> }+ ""

<data value> ::=
        <token> | <string> | <int> | <real>
```

```
<token> ::=
        <uc letter> - <unsint>

<constant> ::=
         <data value> | nil | null
```

Data Definition Language

```
<ddl command> ::=
        <data value class definition> |
        <object class definition> |
        <situation definition> |
        <computation definition> |
        <action definition>
```

Data-value Classes

```
<data value class definition> ::=
        data-value-class <data value class name> .
                type: <data value type> .
                [  form:  <name rule> . ]

                                ; for type STRING only
                                ; required for type STRING

                [  size:  <unsint> . ]

                                ; for type STRING only

                [  maxval: <int> . | maxval: <fpn> . ]

                                ; for numeric type only
                                ; required for numeric type

                [  minval: <int> . | minval: <fpn> . ]

                                ; for numeric type only
                                ; required for numeric type

                [  precision: <unsint> . ]
```

```
                                        ; for reals only
                                        ; required for reals

<data value type> ::= STRING | INTEGER | REAL

<data value class name> ::=
        <identifier>

<name rule> ::=
        { <name rule term> }+

<name rule term> ::=
        <string> | <range> | <repetition>

<range> ::=
        '[' "<character>" - "<character>" ']' |
        '[' <name rule term> {, <name rule term> }+ ']'

<repetition> ::=
        '{' <name rule> '}' <repetition factor>

<repetition factor> ::=
        <unsint> | '<' <unsint> | '<' <unsint> '>' <unsint>

                                    ; repetition factors
                                    ; {rule}n -- exactly n
                                    ; {rule}<n>m -- between m and n
                                    ; {rule}<n -- n or less
```

Object Classes

```
<object class definition> ::=
    object-class <object class name> .
    [ definition: <object class definition slot> . ]
    [ superclasses: <object class name>{,<object class name>}+ . ]
    [ representative: <representative descriptor> . ]
    [ names: { <situation name> }','+    . ]

                                ; The names: slot is only valid for
                                ; situations with representative: TOKEN.


<object class name> ::=
        <identifier>
```

```
<object class definition slot> ::=
        SYSTEM | <situation name>

<representative descriptor> ::=
        TOKEN | <data value class name>
```

Situations

```
<situation definition> ::=
    situation <situation name> .
    participants: ( <participant> {<participant>}+ )
    [ cardinalities: {<cardinality constraint>}',' + ]
    definition: <situation definition slot>
    [ necessary: <necessary> . ]
    [ required: <required> . ]
    [ extension: <extension option> . ]
    [ sufficient: <sufficient> . ]

<situation name> ::=
        <identifier>

<participant> ::=
        ( <role>  <variable>  <object class name> )

<cardinality constraint> ::=
        <unsint> "<" <variable> { . <variable>}* ">"

                        ; example:  1 <x y>
                        ; Interpretation is that at most n
                        ; tuples in the extension at any time
                        ; may have identical sub-tuples of
                        ; the indicated sort.

<situation definition slot> ::=
        <open sigma expression> | PRIMITIVE | SYSTEM

<necessary> ::=
        <open sigma expression>

<required> ::=
        <open sigma expression>

<sufficient> ::=
        <open sigma expression>
```

```
                              ; Assuming integrity of the database,
                              ; the sufficient: sigma expression
                              ; should produce an extension identical
                              ; to the definition:.

<extension option> ::=
       OPEN-WORLD | CLOSED-WORLD

                              ; This slot is only meaningful for
                              ; PRIMITIVE situations.
```

Computations

```
<computation definition> ::=
    computation <computation name> .
    participants: { <computation participant> .}+
    definition: <computation definition slot>

<computation participant> ::=
       <computation role> / <variable> / <computation type>

<computation role> ::=
       result | <argument role>

<argument role> ::=
       domain | value | mapping | measure | <role>

                                     ; The last one can be removed
                                     ; if some effort is made to
                                     ; permit multiple arguments
                                     ; of the same type.

<computation type> ::=
       <simple computation type> |
       <second order type>

<second order type> ::=
       vector-of ( <computation type> ) |
       role-of ( <situation literal> ) |
       instance-of ( <situation literal> ) |
       extension-of ( <situation literal> ) |
       object-of ( <role literal> , <situation literal> ) |
       operation-from <computation type> to <computation type>

<simple computation type> ::=
       INTEGER | REAL | NUMBER | <variable> | <object class name>
```

```
                                        ; Variables must define
                                        ; other participants in
                                        ; the computation.

<situation literal> ::=
        <variable> | <situation name>

                                        ; Variables must define
                                        ; other participants in
                                        ; the computation.

<role literal> ::=
        <variable> | <role name>

<computation definition slot> ::=
        SYSTEM |
        <open sigma expression> |
        <computation literal> |
        <object literal>
```

Actions

```
<action definition> ::=
        action <action name> .
                participants: { <participant> .}+
                results: <action result slot>
                [ prerequisites: <open sigma expression> ]

<action result slot> ::=
        <open sigma expression>
```

Sigma Expressions

```
<sigma expression> ::=
        <open sigma expression> |
        <closed sigma expression> |

<closed sigma expression> ::=
        (sigma ( {<sigma variable specification>}+ )
              <open sigma expression> )
```

```
<vector sigma expression> ::=
        (sigma ( <sigma variable specification> )
               <open sigma expression> )


<sigma variable specification> ::=
        ( <variable> )

<open sigma expression> ::=
        <sigma term> |
        ( and { <open sigma expression> }+ ) |
        ( or { <open sigma expression> }+ )  |
        ( empty <open sigma expression> )

<sigma term> ::=
        <sigma literal> | ( not <open sigma expression> )

                        ; Except when 'not' precedes PRIMITIVE
                        ; situations with an open-world extension,
                        ; it may only be used in the immediate
                        ; scope of an 'and' expression
                        ; where all free variables specified in
                        ; negated conjuncts are also specified
                        ; in affirmed conjuncts.
                        ; Can only be <sigma literal> in SIDUR 2.0.
                        ; The second alternative is allowed by
                        ; this implementation of SIDUR 2.0.
<sigma literal> ::=
        <extension literal> |
        <computation literal>

<extension literal> ::=
        ( <situation name> { ( <role> : <object literal> ) } + )

<object literal> ::=
        <simple object literal> |
        <computed object literal>

<simple object literal> ::=
        <constant> | <variable>

<computed object literal> ::=
        ( <computation name> { . <argument literal> } + )

                        ; Computed object literal is the same
                        ; as computation literal except that the
                        ; result: computation role may not
                        ; appear. A computed object literal is
                        ; taken to denote the object computed.
```

```
<computation literal> ::=
    ( <computation name>
        { . { <argument literal> | <result literal>} } + )



<argument literal> ::=
        ( <domain role>: <object literal> ) |
        ( <domain role>: <closed sigma expression> ) |
        ( <domain role>: <situation name> ) |
        ( <measure role>: " <role name> " ) |
        ( <mapping role>: " <computation name> " )

<result literal> ::=
        ( <result role>: <object literal> )

<role literal> ::=
        <role> | <variable>
```

Data Manipulation Language

```
<dml command> ::= <operation>

<operation> ::=
        <simple operation> |
        <compound operation>

<simple operation> ::=
        <action operation> |
        <object operation> |
        <situation operation>

<object operation> ::=
        ( CREATE <object class name> [ <variable> ] ) |
        ( DESTROY <simple object literal> )

<situation operation> ::=
        ( <situation operator> <sigma expression> )

<situation operator> ::=
        ASSERT | DENY | CHECK | ENQUIRE | REFLECT | REFLECT-NOT

<action operation> ::=
        ( <action operator> <action literal> )
```

```
<action operator> ::=
        PERFORM | PERMIT? | PERMIT!

<action literal> ::=
        ( <action name> { <role>: <simple object literal> }+ )

<compound operation> ::=
        ( <compound operator> ( <variable> {. <variable> }+ )
          <open sigma expression>
          { <simple operation> }+ )

<compound operator> ::=
        FOR | SINCE

                        ; FOR performs the indicated operations
                        ; for each binding tuple in the extension
                        ; of the open sigma expression.
                        ; SINCE does the same, but a REFLECT
                        ; is performed on the expression to force an
                        ; extension to occur. If the REFLECT fails,
                        ; no operations are performed.
```

Index Of Reserved Words And Prefixes

The following is a list of reserved words and prefixes that have special meaning to SIDUR. The grammatical constructs in which they are defined are listed as well.

action	<action definition>
agent	<role>
and	<open sigma expression>
ASSERT	<situation operator>
cardinalities:	<situation definition>
CHECK	<situation operator>
CLOSED-WORLD	<extension option>
computation	<computation definition>
context	<argument role>
context	<argument literal>
CREATE	<object operation>
data-value-class	<data value class definition>
definition:	<computation definition>
definition:	<object class definition>
definition:	<situation definition>
DENY	<situation operator>
destination	<role>
DESTROY	<object operation>

domain	<argument literal>
domain	<argument role>
empty	<open sigma expression>
ENQUIRE	<situation operator>
extension:	<situation definition>
extension-of	<second order type>
FOR	<compound operator>
form:	<data value class definition>
instance-of	<second order type>
INTEGER	<data value type>
INTEGER	<simple computation type>
location	<role>
mapping	<argument literal>
mapping	<argument role>
maxval:	<data value class definition>
measure	<argument literal>
measure	<argument role>
minval:	<data value class definition>
necessary:	<situation definition>
not	<sigma term>
NUMBER	<simple computation type>
object	<role>
object-class	<object class definition>
OPEN-WORLD	<extension option>
operation-from	<second order type>
or	<open sigma expression>
participants:	<action definition>
participants:	<computation definition>
participants:	<situation definition>
PERFORM	<action operator>
PERMIT?	<action operator>
PERMIT!	<action operator>
precision:	<data value class definition>
prerequisites:	<action definition>
PRIMITIVE	<computation definition slot>
PRIMITIVE	<situation definition slot>
REAL	<data value type>
REAL	<simple computation type>
REFLECT	<situation operator>
REFLECT-NOT	<situation operator>
representative:	<object class definition>
required:	<situation definition>
result	<computation role>
result:	<computation definition>
results:	<action definition>
role-of	<second order type>
sigma	<sigma expression>
SINCE	<compound operator>
situation	<situation definition>
size:	<data value class definition>
source	<role>
STRING	<data value type>

superclasses:	<object class definition>
SYSTEM	<computation definition slot>
SYSTEM	<object class definition slot>
SYSTEM	<situation definition slot>
T-	<token>
time	<role>
to	<second order type>
TOKEN	<representative description>
type:	<data value class definition>
value	<argument literal>
value	<argument role>
value	<role>
vector-of	<second order type>

Appendix F. BAGAL Query Language BNF

Specification of New BAG Query Format

The new BAG query format has been designed to separate out the declarative (access oriented) from the procedural (action oriented) aspects of query definition. Under the new format, the BAG query itself is a collection of BAG assignments and control expressions. The expressions that define BAGs in BAG assignments are of two types, ACCESS BAG expressions and BAG operations.

Access BAG expressions form the declarative portion of the BAG query. They include non-directional specifications of binary functional associations (BFAs) that must be accessed, specifications of values that must be computed on the basis of other values, and constraints on the possible values acceptable as output to the BAG.

The rest of the BAG assignments involve BAGs defined by BAG operations. BAG operations are operations that can be performed on already defined BAGs to yield new BAGs. Some of these are operations on the whole BAG, such as merge and sort operations, while others are composed of tuple-at-a-time augmentation of an existing BAG. Using BAG operations, it is possible to fully proceduralize the declarative information contained in BAG definitions. In this fashion, the query planning and optimization process can be implemented purely in terms of transformations on the BAG query, which fully proceduralize the declarative aspects of an initial ACCESS BAG.

Preliminaries

```
<uc letter> ::=

        A | B | C | D | E | F | G | H | I | J | K | L | M |
        N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<lc letter> ::=
        a | b | c | d | e | f | g | h | i | j | k | l | m |
        n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::=
        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9


<unsint> ::=
        { <digit> }+

<int> ::=
        <unsint> | - <unsint> | + <unsint>

<real> ::=
        <int> '.' | <int> '.' <unsint>

<fpn> ::=
        <real> E <int>

<sp char> ::=
        # | ! | ? | _ | & | % | - | '*' | $ | "" | '''

<comparator> ::=
        = | <> | <= | >=

<letter> ::=
        <uc letter> | <lc letter>

<character> ::=
        <letter> | <digit> | <sp char>

<situation name> ::=
        { <uc letter> }+

<role name> ::=
        { <uc letter> }+
<variable> ::=
        { <letter> }+ - <unsint>

<string> ::=
        "" { <character> }* ""
```

BAG Query Definition

```
<bag query> ::=
        { <comment> | <bag operation> } *
        <return operation>

<comment> ::=
        ( comment { <string> | <variable> } * )  |
        "<" character* ">"

<return operation> ::=
        ( RETURN { <bag projection> }* ) |
        <fail operation>

<fail operation> ::=
        ( FAIL <string> )
```

BAG Operations

```
<bag operation> ::=
    <bag assignment> |
    <return operation> |
    ( FOR <bag projection> { <tuple operation> }+ ) |
    ( IF <bag condition> THEN <bag operation>
                     [ ELSE <bag operation> ] )
    ( CASE { <bag case clause> }* [ <bag else clause> ] )


<bag projection> ::=
        ( BAG <bag label> <bag projection list> )

<bag projection list> ::=
        { <variable> | ( <variable> '<-' <variable> ) }*
```

```
<bag case clause> ::=
        ( <bag condition> <bag operation> )

<bag else clause> ::=
        ( ELSE <bag operation> )

<bag condition> ::=
        ( FULL <bag projection> ) |

                        ; true if the BAG either has tuples or
                        ; is the constant BAG "FULL"

        ( EMPTY <bag projection> ) |

                        ; true only if the BAG is the
                        ; the constant BAG "EMPTY"

        ( TUPLES <bag projection> ) |

                        ; true only if the BAG contains
                        ; actual tuples

        ( OR  { <bag condition> }+ ) |
        ( AND { <bag condition> }+ ) |
        ( NOT   <bag condition>    )
```

BAG Assignments

```
<bag assignment> ::=
   (( BAG <bag label> <bag variable list> [ <limit expression> ] )
          '<-' <bag construction> )

<limit expression> ::=
        ( LIMIT <integer> )

                        ; The limit expression limits the number
                        ; of tuples produced as a result of the
                        ; BAG ACCESS or operation.

<bag variable list> ::=
        ( { <bag variable specification> }* )

<bag variable specification> ::=
        <variable> |
        ( <variable> NO-NULL ) |
        ( <variable> AGGREGATE <initial value> <partition> )
```

```
<initial value> ::=
        <simple value expression>

            ; Aggregate variables are initialized prior to
            ; being used. The variable itself may be referred
            ; to on the right hand side of the expression that
            ; computes its value in each iteration.

<partition> ::=
        ( { <variable> }* )

            ; The partition induced divides the tuples of
            ; a BAG into unique classes based on each possible
            ; binding of values to the partition variables.
            ; Aggregate variables are treated as if there is
            ; a separate variable for each partition. The value
            ; associated with each tuple in the BAG is the
            ; value for the partition to which that tuple
            ; belongs.
```

BAG Constructions

```
<bag construction> ::=
        <access operation> |
        ( FULL ) |
        ( EMPTY ) |
        ( TUPLES { <tuple definition> } + ) |
        ( FOR <bag projection>  { <tuple operation> } + ) |
        ( LOOP <bag label>  { <tuple operation> } + ) |

            ; The semantics of LOOP is that the sequence
            ; of tuple operations are repeated beginning with
            ; the initially named BAG, until the BAG stabilizes,
            ; i.e. there is an iteration in which nothing is
            ; added or removed from the BAG. This provides
            ; an iterative mechanism for creating structures
            ; like transitive closure. LOOP will not be
            ; implemented on the first pass.

        ( AND-MERGE { <bag projection> } + ) |
        ( OR-MERGE { <bag projection> } + ) |
        ( MINUS <bag projection> <bag projection> ) |
        ( SORT <bag projection> <sort specification> ) |
        ( SEQUENTIAL-INSTANCES <situation name> ) |
        ( SEQUENTIAL-VALUES <situation name> )
        ( COLLECT <bag projection> <variable list>
                  { <variable list> } + )

                ; The first variable list to COLLECT is
                ; the list of common values over which
                ; BAG bindings are to be COLLECTed. Subsequent
                ; lists begin with the new variable. The
                ; remainder of each list chooses variables to
                ; be COLLECTed under the new variable.

        ( SPREAD <bag projection> { <variable> } + )

                ; This function is the inverse of COLLECT,
                ; expanding variables that have been
                ; previously COLLECTed. If the variables
                ; expanded have not been COLLECTed, an
                ; error occurs.



<sort specification> ::=
        ( <variable> + ) | ( <variable> - )

<tuple definition> ::=
        ( { <tuple variable assignment> } * )


<tuple operation> ::=
    <access operation> |
    <tuple variable assignment> |
    <update operation> |
    <fail operation> |
    ( IF <tuple condition> THEN <tuple operation>
                          [ ELSE <tuple operation> ] ) |
    ( CASE { <tuple case clause> }* [ <tuple else clause> ] )

<tuple variable assignment> ::=
        ( <variable> '<-' <value expression> )

<tuple case clause> ::=
        ( <tuple condition> { <tuple operation> }* )

<tuple else clause> ::=
        ( ELSE { <tuple operation> }* )

<update operation> ::=
        ( CHANGE <situation name> / <role name> <variable>
                                    <value expression> )
        ( DELETE <situation name> / <role name> <variable> )
```

```
<value expression> ::=
    <simple value expression> |
    ( INDEX (<situation name> ( <role name> <variable> )) |
    ( CINDEX (<situation name> {(<role name> <variable>)} + )) |
    ( VALUATION (<situation name> ( <role name> <variable> ))

            ; The result of the valuation function is always single
            ; valued. The result of the index or cindex function
            ; may not be.
            ; When multiple values are returned, these
            ; cause multiple tuples to be produced by
            ; the tuple operation.
```

<u>ACCESS Operations</u>

```
<access operation> ::=
        ( ACCESS { <access expression> } + )

             ; ACCESS operations may be used to define
             ; an entire BAG or to define assignment of
             ; values to variables within a tuple-by-tuple
             ; assignment loop. In the latter case, values
             ; already assigned in the loop are considered
             ; fixed (constant). If the ACCESS expression
             ; generates multiple values, these become extra
             ; tuples for the rest of the loop.

<access expression> ::=
        <bfa expression> |
        <tuple condition>

<bfa expression> ::=
    ( <situation name> / <role name> <variable> <variable> )


<tuple condition> ::=
    ( OR { <tuple compare> } + )
    ( AND { <tuple compare> } + )
    ( NOT { <tuple compare> } )
    ( <variable> MEMBER { <constant> }+ )

<tuple compare> ::=
    ( <variable> <comparator> <simple value expression> )
```

```
<simple value expression> ::=
        <constant> |
        <variable> |
        <computation expression>

<constant> ::=
        <string> | <int> | <real> | <fpn>

<computation expression> ::=
        ( ICREATE <situation name> ) |
        ( OCREATE <object class name> ) |
        ( CHOICE { <string> | <variable> } + ) |
        ( <computation name> { <simple value expression> }* )
```

The BAG assignments and operations defined in the BAG query are assumed to be listed in order of execution. For optimization reasons, this order may be permuted when the following two conditions hold:

(1) The second operation does not depend on the first operation or on any BAG that does depend on the first operation.

(2) The second operation does not access any BFAs that are updated by the first operation, nor update any BFAs that are accessed by the first operation.

The goal is that any permutation of BAG operations that will make a semantic difference in the result of the query is not permitted. Only when it can be demonstrated that this does not happen can the order of the BAG operations be permuted.

Appendix G. Simplified BNF

Data Definition Language

```
<ddl command> ::=
        <data value class definition> |
        <object class definition> |
        <situation definition> |
        <computation definition> |
        <action definition>
```

Data-value Classes

```
<ddl command> ::=
<data value class definition> ::=
        ( data-value-class <data value class name> .
                type: <data value type> .
                [  form: ( <name rule> ) . ]

                                ; for type STRING only
                                ; required for type STRING

                [  size:  <unsint> . ]

                                ; for type STRING only
                                ; not required

                [  maxval: <int> . | maxval: <real> . ]

                                ; for numeric type only
                                ; required for numeric type

                [  minval: <int> . | minval: <real> . ]
```

```
                                  ; for numeric type only
                                  ; required for numeric type

                 [  precision: <unsint> . ]    )

                                  ; for reals only
                                  ; required for reals

<data value type> ::= STRING | INTEGER | REAL

<data value class name> ::=
        <identifier>

<name rule> ::=
        { <name rule term> }+

<name rule term> ::=
        <string> | <range> | <repetition>

<range> ::=
        '[' "<character>" - "<character>" ']' |
        '[' <name rule term> { <name rule term> }+ ']'

<repetition> ::=
        ( '&' <name rule> '$' <repetition factor> )

<repetition factor> ::=
   <unsint> | '<' <unsint> | '<' <unsint> '>' <unsint> | '*' | '+'

                              ; repetition factors
                              ; {rule}n -- exactly n
                              ; {rule}<n>m -- between m and n
                              ; {rule}<n -- n or less
                              ; * -- 0 or more
                              ; + -- 1 or more
```

Object Classes

```
<ddl command> ::=
<object class definition> ::=
    ( object-class <object class name> .
        [ definition: ( <object class definition slot> ) . ]
        [ superclasses: ( <object class name>
                          {<object class name>}+ ).
        [ ( representative: <representative descriptor> ) . ]
        [ names: { ( <situation name> }+ ) . ] )
```

```
                              ; The names: slot is only valid for
                              ; situations with representative: TOKEN.


<object class name> ::=
       <identifier>

<object class definition slot> ::=
       SYSTEM | <situation name>

<representative descriptor> ::=
       TOKEN | <data value class name>
```

Situations

```
<ddl command> ::=
<situation definition> ::=
    ( situation <situation name> .
       participants: ( <participant> {<participant>}+ )  .
       [ cardinalities: ( {<cardinality constraint>}+ ) ] .
       definition: <situation definition slot>  .
       [ necessary: <necessary> . ]
       [ required: <required> . ]
       [ extension: ( <extension option> ) . ] )
       [ sufficient: ( <sufficient> ) . ] )

<situation name> ::=
       <identifier>

<participant> ::=
       ( <role> / <variable> / <object class name> )

<cardinality constraint> ::=
       ( <unsint>  <variable> { . <variable> } * > )

                                ; example:  ( 1 x y )
                                ; Interpretation is that at most n
                                ; tuples in the extension at any time
                                ; may have identical sub-tuples of
                                ; the indicated sort.
<situation definition slot> ::=
       <open sigma expression> | ( PRIMITIVE ) | ( SYSTEM )

<necessary> ::=
       <open sigma expression>
```

```
<required> ::=
        <open sigma expression>

<sufficient> ::=
        <open sigma expression>

                              ; Assuming integrity of the database,
                              ; the sufficient: sigma expression
                              ; should produce an extension identical
                              ; to the definition:.

<extension option> ::=
        OPEN-WORLD | CLOSED-WORLD

                              ; This slot is only meaningful for
                              ; primitive situations.
```

Computations

```
<ddl command> ::=
<computation definition> ::=
    ( computation <computation name>  .
        participants: ( { <computation participant> .}+ )
        definition: <computation definition slot> ) .

<computation participant> ::=
    ( <computation role> / <variable> / <computation type> )

<computation role> ::=
    result | <argument role>

<argument role> ::=
    domain | value | mapping | measure | <role>

                                     ; The last one can be removed
                                     ; if some effort is made to
                                     ; permit multiple args of the
                                     ; same type.

<computation type> ::=
       <simple computation type> |
       <second order type>
```

```
<second order type> ::=
        vector-of ( <computation type> ) |
        role-of ( <situation literal> ) |
        instance-of ( <situation literal> ) |
        extension-of ( <situation literal> ) |
        object-of ( <role literal> , <situation literal> ) |
        operation-from <computation type> to <computation type>

<simple computation type> ::=
    INTEGER | REAL | NUMBER | <variable> | <object class name>

                                    ; Variables must define
                                    ; other participants in
                                    ; the computation.

<situation literal> ::=
        <variable> | <situation name>

                                    ; Variables must define
                                    ; other participants in
                                    ; the computation.

<role literal> ::=
        <variable> | <role name>

<computation definition slot> ::=
        ( SYSTEM ) |
        <open sigma expression> |
        <computation literal> |
        ( <object literal> )
```

Actions

```
<ddl command> ::=
<action definition> ::=
        ( action <action name> .
              participants: { <participant> .}+
              results: <action result slot>
              [ prerequisites: <open sigma expression> ] )

<action result slot> ::=
        ( <open sigma expression> )
```

Sigma Expressions

```
<sigma expression> ::=
        <open sigma expression> |
        <closed sigma expression> |

<closed sigma expression> ::=
        (sigma ( {<sigma variable specification>}+ )
              <open sigma expression> )

<vector sigma expression> ::=
        (sigma ( <sigma variable specification> )
              <open sigma expression> )


<sigma variable specification> ::=
        <variable>

<open sigma expression> ::=
        <sigma term> |
        ( and { <open sigma expression> }+ ) |
        ( or { <open sigma expression> }+ )  |
        ( empty <open sigma expression> )

<sigma term> ::=
        <sigma literal> | ( not <open sigma expression> )

                         ; Except when 'not' precedes PRIMITIVE
                         ; situations with an open-world extension,
                         ; it may only be used in the immediate
                         ; scope of an 'and' expression
                         ; where all free variables specified in
                         ; negated conjuncts are also specified
                         ; in affirmed conjuncts.
                         ; Can only be <sigma literal> in SIDUR 2.0.
                         ; The second alternative is allowed by
                         ; this implementation of SIDUR 2.0.

<sigma literal> ::=
        <extension literal> |
        <computation literal>

                         ; This implementation does not allow a
                         ; computation within the scope of 'not'.

<extension literal> ::=
        ( <situation name> { ( <role> . <object literal> ) } + )
```

```
<object literal> ::=
        <simple object literal> |
        <computed object literal>

<simple object literal> ::=
        <constant> | <variable>

<computed object literal> ::=
        ( <computation name> { . <argument literal> } + )

                         ; Computed object literal is the same
                         ; as computation literal except that the
                         ; result: computation role may not
                         ; appear. A computed object literal is
                         ; taken to denote the object computed.


<computation literal> ::=
    ( <computation name>
         { . { <argument literal> | <result literal>} } + )



<argument literal> ::=
        ( <domain role> <object literal> ) |
        ( <domain role> <closed sigma expression> ) |
        ( <domain role> <situation name> ) |
        ( <measure role> " <role name> " ) |
        ( <mapping role> " <computation name> " )

<result literal> ::=
        ( <result role> <object literal> )

<role literal> ::=
        <role> | <variable>
```

Data Manipulation Language

```
<dml command> ::= <operation>

<operation> ::=
        <simple operation> |
        <compound operation>
```

```
<simple operation> ::=
       <action operation> |
       <object operation> |
       <situation operation>

<object operation> ::=
       ( CREATE <object class name> [ <variable> ] ) |
       ( DESTROY <simple object literal> )

<situation operation> ::=
       ( <situation operator> <sigma expression> )

<situation operator> ::=
       ASSERT | DENY | CHECK | ENQUIRE | REFLECT | REFLECT-NOT

<action operation> ::=
       ( <action operator> <action literal> )

<action operator> ::=
       PERFORM | PERMIT? | PERMIT!

<action literal> ::=
       ( <action name> { <role> <simple object literal> }+ )

<compound operation> ::=
       ( <compound operator> ( <variable> {. <variable> }+ )
         <open sigma expression>
         { <simple operation> }+ )

<compound operator> ::=
       FOR | SINCE

                        ; FOR performs the indicated operations
                        ; for each binding tuple in the extension
                        ; of the open sigma expression.
                        ; SINCE does the same, but a REFLECT
                        ; is performed on the expression to force an
                        ; extension to occur. If the REFLECT fails,
                        ; no operations are performed.
```

Appendix H. A Sample Schema--Simplified Form

```
( data-value-class AGE-V
   type: INTEGER
   minval: 0
   maxval: 90 )

( data-value-class CLASS-LIMIT-V
   type: INTEGER
   minval: 10
   maxval: 100 )

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

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

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

( data-value-class PERSONAL-NAME-V
   type: STRING
   size: 14
   form: ( & [ "A" - "Z" ] $ < 5 "-" & [ "A" - "Z" ] $ < 8  ))

( data-value-class PHONE-NUM
   type: STRING
   size: 12
   form: ( & [ "0" - "9" ] $ 3 "-" & [ "0" - "9" ] $ 3 "-"
           & [ "0" - "9" ] ))

( data-value-class SSNUM
   type: STRING
   size: 9
   form: ( & [ "1" - "9" ] $ 9 ))
```

```
( data-value-class TIME-V
    type: STRING
    size: 4
    form: ( [ [ "1" - "9" ] | "10" | "11" | "12"]
            [ "A" | "P" ] "M" ))

( object-class AGE
    representative: ( AGE-V ))

( object-class CLASS-LIMIT
    representative: ( CLASS-LIMIT-V ))

( object-class COURSE
    representative: ( TOKEN )
    definition: ( IS-COURSE )
    names: ( HAS-TITLE ))

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

( object-class EMPLOYEE
    representative: ( TOKEN )
    definition: ( IS-EMPLOYEE )
    superclasses: ( PERSON )
    names: ( IS-PERSON-NAME HAS-NAME ))

( object-class GPA
    representative: ( GPA-V ))

( object-class GRAD-STUDENT
    representative: ( TOKEN )
    definition: ( IS-GRAD-STUDENT )
    superclasses: ( STUDENT )
    names: ( IS-STUDENT-NAME ))

( object-class GRADE
    representative: ( GRADE-V ))

( object-class HOUR
    representative: ( TIME-V ))

( object-class INSTRUCTOR
    representative: ( TOKEN )
    definition: ( IS-INSTRUCTOR )
    superclasses: ( PERSON )
    names: ( IS-PERSON-NAME HAS-NAME ))

( object-class NAME
    representative: ( PERSONAL-NAME-V ))
```

```
( object-class OFFERING
    representative: ( TOKEN )
    definition: ( IS-OFFERING )
    names: ( HAS-TITLE ))

( object-class PERSON
    representative: ( TOKEN )
    definition: ( IS-PERSON )
    names: ( IS-PERSON-NAME HAS-NAME ))

( object-class PERSON-NAME
    representative: ( PERSONAL-NAME-V ))

( object-class STAFF-STUDENT
    representative: ( TOKEN )
    definition: ( IS-STAFF-STUDENT )
    superclasses: ( STUDENT EMPLOYEE )
    names: ( IS-STUDENT-NAME ))

( object-class STUDENT
    representative: ( TOKEN )
    definition: ( IS-STUDENT )
    names: ( IS-STUDENT-NAME ))

( object-class UNDERGRAD-STUDENT
    representative: ( TOKEN )
    definition: ( IS-UNDERGRAD-STUDENT )
    superclasses: ( STUDENT )
    names: ( IS-STUDENT-NAME ))

( situation ADMITTED-TO-GRAD-SCHOOL
    participants: (( agent x PERSON ))
    cardinalities: (( 1 x ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

( situation CAN-TEACH
    participants: (( agent x PERSON ) ( object y COURSE ))
    definition: ( PRIMITIVE )
    extension: ( OPEN-WORLD ))

( situation COURSE-GRADE
    participants: (( agent x STUDENT ) ( object y COURSE )
                   ( value z GRADE ))
    definition: ( and ( GRADE-FOR  ( agent x ) ( object  w )
                                   ( value z ))
                      ( OFFERING-OF ( agent y ) ( object  w ))))
( situation COURSE-GRADE-2
    participants: (( agent x STUDENT ) ( object y COURSE )
                   ( value z GRADE ))
    definition: ( and ( GRADE-FOR ( agent x ) ( object w )
                                  ( value z ))
                      ( OFFERING-OF ( agent y ) ( object w ))))

( situation FINAL-GPA
    participants: (( agent x STUDENT ) ( value y GPA ))
    cardinalities: (( 1 x y ))
    definition: ( PRIMITIVE ))

( situation FLUNKED
   participants: (( agent x STUDENT ) ( object y COURSE ))
   definition: ( and ( OFFERING-OF ( agent y ) ( object z ))
                     ( GRADE-FOR ( agent x ) ( object z )
                                 ( value w "F" ))))

( situation GRADE-FOR
    participants: (( agent x STUDENT ) ( object y OFFERING )
                   ( 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 HAS-AGE
    participants: (( agent x PERSON ) ( value y AGE ))
    cardinalities: (( 1 x ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

( situation HAS-NAME
    participants: (( agent x PERSON ) ( value y NAME ))
    cardinalities: (( 1 x y ))
    definition: ( PRIMITIVE ))

( situation HAS-TITLE
    participants: (( agent x COURSE ) ( object y COURSE-NAME ))
    cardinalities: (( 1 x ))
    definition: ( PRIMITIVE ))

( situation INTERESTING
   participants: (( agent x PERSON ) ( result y BOOLEAN ))
   cardinalities: (( 1 x ))
   definition: ( PRIMITIVE ))
```

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

( situation IS-COURSE-NAME
    participants: (( agent x COURSE ) ( object y COURSE-NAME ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

( situation IS-EMPLOYEE
    participants: (( agent x PERSON ))
    definition: ( or ( IS-INSTRUCTOR ( agent x ))
                     ( IS-STAFF-STUDENT ( agent x ))))

 ( situation IS-FACULTY
    participants: (( agent x EMPLOYEE ))
    definition: ( PRIMITIVE )
    necessary: ( FINAL-GPA ( agent x ))
    required: ( or ( TEACHES-OFFERING ( agent x ) ( object z ))
                   ( CAN-TEACH ( agent x )))
    extension: ( OPEN-WORLD ))

( situation IS-GRAD-STUDENT
    participants: (( agent x GRAD-STUDENT ))
    definition: ( and ( ADMITTED-TO-GRAD-SCHOOL ( agent x ))
                      ( IS-STUDENT ( agent x )))
    necessary: ( ADMITTED-TO-GRAD-SCHOOL ( agent x ))
    required: ( HAS-AGE ( agent x )))

( situation IS-GRAD-STUDENT-2
    participants: (( agent x STUDENT ))
    definition: ( ADMITTED-TO-GRAD-SCHOOL ( agent x ))
    necessary: ( IS-PERSON ( agent x ))
    required: ( ADMITTED-TO-GRAD-SCHOOL ( agent x )))

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

( situation IS-OFFERING
    participants: (( agent x OFFERING ))
    definition: ( OFFERING-OF ( object x )))

( situation IS-PERSON
    participants: (( agent x PERSON ))
    definition: ( PRIMITIVE ))

( situation IS-PERSON-NAME
    participants: (( agent x PERSON ) ( object y PERSON-NAME ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))
```

```
( situation IS-PRESIDENT
    participants: (( agent x EMPLOYEE ))
    definition: ( PRIMITIVE ))

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

( situation IS-STAFF-STUDENT
    participants: (( agent x STUDENT ))
    definition: ( PRIMITIVE ))

( situation IS-STUDENT-NAME
    participants: (( agent x STUDENT ) ( object y PERSON-NAME ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

( situation IS-UNDERGRAD-STUDENT
    participants: (( agent x STUDENT ))
    definition: ( and ( IS-STUDENT ( agent x ))
                      ( not ( IS-GRAD-STUDENT ( agent x )))))

( situation LIMIT
   participants: (( agent x OFFERING ) ( value y CLASS-LIMIT ))
   cardinalities: (( 1 x ))
   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 ))
                        ( IS-STUDENT ( agent x ))
                        ( not ( COURSE-GRADE ( agent x )
                                             ( object z )))))))
   required: ( IS-OFFERING ( agent y ))
   definition: ( PRIMITIVE ))

( situation MEETING-TIME
    participants: (( agent x OFFERING ) ( value y HOUR ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

( situation OFFERING-OF
    participants: (( agent x COURSE ) ( object y OFFERING ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))
```

```
( situation PREREQUISITE-FOR
   participants: (( agent x COURSE ) ( object y COURSE ))
   definition: ( PRIMITIVE )
   extension: ( CLOSED-WORLD ))

( situation TAKES-COURSE
    participants: (( agent x STUDENT ) ( object y COURSE ))
    definition: ( and ( TAKES-OFFERING ( agent x ) ( object z ))
                      ( OFFERING-OF ( agent y ) ( object z ))))

( situation TAKES-OFFERING
    participants: (( agent x STUDENT ) ( object y OFFERING ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

( situation TEACHES-COURSE
    participants: (( agent x INSTRUCTOR ) ( object y COURSE ))
    definition: ( and ( TEACHES-OFFERING ( agent x ) ( object z ))
                      ( OFFERING-OF ( agent y ) ( object z ))))

( situation TEACHES-OFFERING
    participants: (( agent x INSTRUCTOR ) ( object y OFFERING ))
    cardinalities: (( 1 y ))
    necessary: ( and ( OFFERING-OF ( agent z ) ( object y ))
                     ( CAN-TEACH ( agent x ) ( object z )))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

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

( computation AVERAGE-OF
   participants: (( domain x extension-of ( s ))
                   ( measure y role-of ( s ))
                   ( result z real ))
   definition:
     ( DIVIDE
       ( domain-1  ( SUM ( domain x ) ( measure y )
                         ( result r )))
       ( domain-2  ( COUNT-OF ( domain x ) ( measure y )
                              ( result t )))))
( computation COUNT-INTERESTING
   participants: (( domain x  object-of ( A ))
                  ( result y INTEGER ))
   definition:
        ( ACCUMULATE
            ( mapping  "SUM" )
            ( domain
               ( MAP ( domain z )
                     ( mapping  ( sigma ( z )
                       ( INTERESTING ( agent x ) ( result z ))))
                     ( result w )))
              ( mapping   "PLUS" )
              ( result y )))

( computation COUNT-OF
   participants: (( domain x extension-of ( A ))
                  ( result y INTEGER ))
   definition:
           ( ACCUMULATE
              ( domain
                ( MAP ( domain x )
                      ( mapping   "INSTANCE-VALUE" )
                      ( result u )))
              ( mapping   "PLUS" )
              ( result y )))

( computation COUNT-UNIQUE
   participants: (( domain x extension-of ( s ))
                  ( measure y role-of ( s ))
                  ( result z INTEGER ))
   definition:
     ( COUNT-OF
       ( domain ( PROJECT ( domain x ) ( measure y )
                          ( result r )))
       ( measure y )
       ( result z )))

( computation ENROLLMENT
   participants: (( domain x COURSE ) ( result y INTEGER ))
   definition:
       ( COUNT-OF
           ( domain
             ( sigma ( z )
                ( TAKES-COURSE ( agent z ) ( object x )
                               ( result y ))))))
( computation GPA-OF
   participants: (( agent x STUDENT ) ( result y GPA ))
   definition:
        ( AVERAGE-OF
             ( context z )
             ( domain
                 ( sigma ( z )
                  ( and ( COURSE-GRADE ( agent x ) ( value w ))
                        ( GRADE-VALUES ( agent  w )
                                       ( value z )))))))

( computation INSTANCE-VALUE
   participants: (( domain x instance-of ( A ))
                   ( result y INTEGER ))
   definition: ( CONSTANT ( domain x ) ( result y 1 )))

( computation SUM
   participants: (( domain x vector-of ( INTEGER ))
                   ( result y INTEGER ))
   definition:
     ( ACCUMULATE ( domain x ) ( mapping "PLUS" ) ( result y )))

( action CANCEL
   participants: (( object x OFFERING ))
   results: ( and ( not ( OFFERING-OF ( object x )))
                  ( not ( TAKES-OFFERING ( object x )))))

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

( action ENROLLS-IN
   participants: (( agent x STUDENT ) ( object y OFFERING ))
   prerequisites:
     ( and ( OFFERING-OF ( agent z ) ( object y ))
           ( MAY-TAKE ( agent x ) ( object z )))
   results: ( TAKES-OFFERING ( agent x ) ( object y )))

( action GRADUATES
   participants: (( agent x STUDENT ) ( object y GPA ))
   results: ( FINAL-GPA ( agent x ) ( value y )))
```

Appendix I. Schema for Figure 16

```
( data-value-class EVENT-NAME-V
    type: STRING
    size: 10
    form: ( & [ "A" - "Z", "-" ] $ ))

( object-class EVENT-NAME
    representative:  ( EVENT-NAME-V ))

( object-class EVENT
    representative:  ( TOKEN )
    definition:  ( IS-EVENT )
    names: ( HAS-TITLE ))

( situation IS-EVENT
    participants: (( agent x EVENT ))
    definition:  ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

( situation IS-EVENT-NAME
    participants: (( agent x EVENT ) ( object y EVENT-NAME ))
    definition:  ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

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

( object-class COURSE
    representative:  ( TOKEN )
    definition:  ( IS-COURSE )
    names: ( HAS-TITLE ))

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

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

```
( situation HAS-TITLE
    participants: (( agent x COURSE ) ( object y COURSE-NAME ))
    cardinalities: (( 1  x ))
    definition: ( PRIMITIVE )
    extension:  (CLOSED-WORLD))

( data-value-class TIME-V
    type: STRING
    size: 4
    form: ( [ [ "1" - "9" ] | "10" | "11" | "12"]
            [ "A" | "P" ] "M" ))

( object-class HOUR
    representative: ( TIME-V ))

( situation MEETING-TIME
    participants: (( agent x OFFERING ) ( value y HOUR ))
    definition: ( PRIMITIVE )
    extension:  ( CLOSED-WORLD ))

( situation IS-NOON-MEETING
    participants: (( agent x EVENT ))
    necessary: ( HAS-TITLE ( agent x ) ( object "MEETING" ))
    required:  ( MEETING-TIME ( agent x ) ( value "12AM" ))
    definition:  ( PRIMITIVE )
    extension:  ( OPEN-WORLD ))

( object-class OFFERING
    representative:  ( TOKEN )
    definition:  ( IS-OFFERING )
    names: ( HAS-TITLE ))

( situation OFFERING-OF
    participants: (( agent x COURSE) ( object y OFFERING ))
    definition: ( PRIMITIVE )
    extension: ( CLOSED-WORLD ))

( situation IS-OFFERING
    participants: (( agent x OFFERING ))
    definition:    ( OFFERING-OF ( object  x )))

( situation IS-SEMINAR
    participants: (( agent x OFFERING ))
    necessary: ( IS-COURSE ( agent x ))
    required:  ( HAS-TITLE ( agent x ) ( object "SE-400" ))
    definition:  (and ( IS-OFFERING ( agent x ))
                      ( LIMIT ( agent x ) ( value 12 )))
    extension:  ( CLOSED-WORLD ))
```

```
( data-value-class CLASS-LIMIT-V
   type: INTEGER
   minval: 10
   maxval: 100 )

( object-class CLASS-LIMIT
   representative: ( CLASS-LIMIT-V ))

( situation LIMIT
   participants (( agent x OFFERING ) ( value y CLASS-LIMIT ))
   cardinalities: (( 1 x ))
   definition: ( PRIMITIVE )
   extension: ( CLOSED-WORLD ))
```

Appendix J Departures from the SIDUR Manual

Numbers -- Objects of type INTEGER and REAL are handled as Franz Lisp numbers.

BNF -- Simplified as shown in Appendix G.

'empty' -- Has same effect as 'not' when used in update operations.

'MENTIONS' -- Returns a list of construct names that appear in the specified slot of a construct.

Schema operations -- Must not involve data operations and can only handle one construct. The name of the construct must be a constant in the operation.

'not' -- Any open sigma expression may appear within a 'not' connective.

'not' -- Only one computation may appear within the scope of a 'not' connective

Appendix K. Notation Used In This Paper

'sidur' *assert	Single quote marks or underlines are used to denote identifiers, reserved words, and items which are to be taken literally.
<expression>	The expression in angle brackets can be replaced by any well-formed expression of the type named inside of the brackets.
REFLECT	Names and reserved words which are capitalized in the OSIRIS literature are also capitalized is this paper.
< comment >	Denotes a comment within an expression or BAGAL procedure.