JUSTIFICATIONS and APPLICATIONS of the SET CONCEPTUAL MODEL

Paul C. Gilmore

Technical Report 87-9
April 1987

Abstract: In an earlier paper, the SET conceptual model was described, along with the domain graph method of table design. In this paper a justification for the method is provided, and a simple condition shown to be sufficient for the satisfaction of the degree constraints of a set schema. The basis for the consistency of the model is also described. Applications of the SET model to the full range of data processing are suggested, as well as to the problems raised by incomplete information.

The research reported on in this paper has been supported by grants of the Natural Sciences and Engineering Research Council of Canada

1. Introduction

In the paper [Gil87b] it was argued that the conceptual orientation of the entity-relationship (ER) model [Chen76,77] permits it to avoid both the excessive implementation concerns of the hierarchical and network models, and the restrictive presentation concerns of the relational model. The weakness of the ER model, on the other hand, is its lack of a sound foundation upon which a management system might be based. The primary motivation for the development of the purely set-based data and conceptual model SET and its specification/query language DEFINE described in [Gil87b] was to provide such a foundation. Five reasons were offered as to why this is necessary:

1. For the unified view of data proposed in [Chen76] to be fully achieved, a database is needed that is capable of recording a high level conceptual model of an enterprise and at the same time of providing the tables for a relational database schema as a defined user view in its specification/query language.
2. The ER modelling process requires a greater discipline than is now possible.
3. A provably sound foundation is needed for databases that can reference and describe themselves.
4. A fully unified model of an enterprise is needed that at the same time can give a conceptual view of the enterprise, a user's view of data as it is presented, a data administrator's view of data as it is stored, and a programmer's view of the processing of the data.
5. Sound foundations are needed for knowledge base systems capable of dealing with incomplete information.

The domain graph method of table design described in [Gil87b] translates the set schema obtained from the modelling of an enterprise using SET, into a table schema in which each table is a defined user view declared as a set in DEFINE. But for one initial step, the method is fully automated. A theorem stated in the paper asserts that the method will always result in correct tables; that is, tables that are free from any anomalies. But no proof was provided for the theorem. The first purpose of this paper is to remedy that deficiency, so that (1) can be offered as an advantage of the SET model. That proof is provided in section 2.

A tentative beginning was made in [Gil87b] in providing a basis for some of the decisions that must be made while modelling, so that (2) can be offered as an advantage of the SET model as well. Another purpose of this paper is to demonstrate that DEFINE can be used as a query language for the SET model, and that the model also satisfies the demands (3)-(5). While doing this, applications and extensions of the SET model will be described in section 3. The final purpose of this paper, accomplished in section 4, is to sketch the basis for the consistency of the model and its integrity constraints.

Familiarity with the paper [Gil87] is presumed.

2. Correctness of the Domain Graph Method of Table Design

The domain graph method of table design described in [Gil87b] translates the set schema obtained from the modelling of an enterprise using SET, into a table schema in which each table is a defined user view declared as a set in DEFINE. The method was described in terms of operations on the augmented domain graph of the set schema. The steps of the method are:

1. Each edge of the augmented domain graph of the set schema is labelled with the lower and upper degrees that have been declared or calculated for it.
2. 1-connected subgraphs of the augmented domain graph are determined by selecting only edges that have been labelled with the lower degree 1. The resulting subgraphs are simplified by eliminating all nodes labelled with undeclared sets, and by replacing directed paths through such nodes with a single edge connecting nodes labelled with declared sets.
3. Each undirected cycle of a subgraph determined in 2 is broken by removing an edge with tail a bottom node of the cycle. The result of this step is a forest of trees.
4. Each tree obtained in 3 is extended with new nodes and edges to form its identifier extension.
5. From the identifier extension of each tree obtained in 4, a declaration of a table as a defined set is constructed.

The construction in (4) of the identifier extension of a tree needed in (3) was described in 3.4 of [Gil87]. The following lemma expresses a fundamental property of identifier extensions:

Lemma 1: Let Tdg be any tree obtained in step 3, and let Tr be its identifier extension obtained in step 4. Let nd_0, nd_1, ... , nd_p, be an undirected path of Tr for which nd_0 is a node of Tdg. Let the edge from nd_i, to nd_{i+1}, have lower degree 0. Then the edge has head nd_i and tail nd_{i+1}.

Proof of lemma 1: An edge of lower degree 0 is not an edge of Tdg, but has been added in making a node arity predecessor complete or in adding a pair of nodes labelled with an identifier for a primitive base set and with a value set for the identifier. The former must point towards the node that is arity predecessor incomplete without it, while the latter must point towards the node labelled with the identifier.

End of proof of lemma 1

Consider now any set schema Sch. Let S be a set declared in Sch of interest to a user. S labels exactly one node of the domain graph of Sch, and therefore exactly one node nde of the augmented domain graph. Let Tdg be the single tree obtained in step 3 of which nde is a node. Let Tr be the identifier extension of Tdg obtained in step 4, and let T(Tr) be the table obtained in step 5. If the domain graph method is correct, then it should be possible for a user to determine the membership of S from the table T(Tr).

The bottom nodes of Tr are labelled with value sets or primitive base sets only, while all other nodes are labelled with nonprimitive sets. The declaration of T(Tr) makes use of an assignment of variables to the bottom nodes of Tr, with a distinct variable assigned to each node. Every other node nd of Tr is then assigned a nested tuple tp of the variables assigned to the bottom nodes; tp is a tuple of the tuples assigned to the nodes that are immediate predecessors of nd. Associated with each node of Tr is therefore an assertion tp:SS, called the **assertion of the node**, where tp is the variable or tuple assigned to the node, and SS is the set that labels the node. Join(Tr) is an assertion of DEFINE consisting of the conjunction of all such assertions for nodes that are not bottom nodes of Tr. The declaration of T(Tr) is then:

T(Tr)={ v_1:V_1, ... , v_n:V_n | [**For some** bv_1:BS_1, ... , bv_m:BS_m] Join(Tr) | }.

Here V_1, ... , V_n are all the value sets that label bottom nodes of Tr in some order with repititions if necessary, and v_1, ... , v_n are the variables assigned to those nodes; BS_1, ... , BS_m are all the primitive base sets that label bottom nodes of Tr in some order with repititions if necessary, and bv_1, ... , bv_m are the variables assigned to those nodes.

Consider now how a user determines the membership of S from T(Tr). First the columns of T(Tr) that identify members of S must be known to the user. These columns are determined as follows: Let tup be the tuple assigned to the node nde that S labels. The variables occurring in tup are among the variables v_1, ... , v_n and bv_1, ... , bv_m, since these are all the variables assigned to bottom nodes of Tr. By reordering the columns of T(Tr) it can be assumed that $v_1, \ldots, v_j$ are the variables among $v_1, \ldots, v_n$ that occur in tup. Without loss of generality it may be assumed that $bv_1, \ldots, bv_b$ are those variables among $bv_1, \ldots, bv_m$ that occur in tup. Join(Tr) necessarily includes a conjunction

bv_1:IBS_1:vbv_1 **and** ... **and** bv_m:IBS_m:vbv_m,

where $IBS_1, \ldots, IBS_m$ are identifiers for $BS_1, \ldots, BS_m$, and $vbv_1, \ldots, vbv_m$ are among the variables $v_1, \ldots, v_n$. None of the variables $vbv_1, \ldots, vbv_b$ occur among $v_1, \ldots, v_j$ since Tr is a tree. Therefore by a futher reordering of the columns they may be assumed to be $v_{j+1}, \ldots, v_{j+b}$. The columns of T(Tr) that are used to identify members of S are therefore those for the variables $v_1, \ldots, v_j, v_{j+1}, \ldots, v_{j+b}$.

With knowledge of the columns of T(Tr) that identify members of S, a user determines the members of S as follows: From a selected row of T(Tr), a user can determine a member of V_1x ... x V_{j+b}. From knowledge of the identifiers $IBS_1, \ldots, IBS_b$ a user can therefore determine a j+b tuple that is a member of V_1x ... x V_jxBS_1x ... xBS_b. This j+b tuple is a flattened version of a member of a S, provided that the table T(Tr) is correct. No matter whether T(Tr) is correct or not, it is a flattened version of a member of a set SU that will be declared after one more definition is given to make more precise what is meant by "flattened".

The **bottom domain** of a set labelling a node of Tr is defined recursively as follows: The bottom domain of the value set or primitive base set that labels a bottom node of Tr is the set itself. Let SS label a node nd that is not a bottom node, and let SS_1x ... xSS_k be the domain of the arity domain of SS. Necessarily there are exactly k nodes that are tails of edges with head nd and these nodes are labelled with $SS_1, \ldots, SS_k$. Let $BDSS_1, \ldots, BDSS_k$ be the bottom domains of $SS_1, \ldots, SS_k$, respectively. Then the bottom domain of SS is $BDSS_1$x ... x$BDSS_k$.

Let BDS be the bottom domain of S. Then the set SU, called the **user form** of S, is declared:
SU={ tup:BDS | [**For some** v_{j+1}:$V_{j+1}, \ldots, v_n$:V_n] (<$v_1, \ldots, v_n$>:T(Tr) **and** <bv_1, v_{j+1}>:IBS_1 **and** ... **and** <bv_b, v_{j+b}>:IBS_b) | }.

The following theorem justifies the domain graph method of table design.

Theorem: Assume that all declared and defined degree constraints labelling edges of an augmented domain graph are satisfied by the membership of the declared sets. Let S be a declared set labelling a node of a 1-connected, not necessarily maximal, subtree of the domain graph, and let SU be the user form of S declared for the table obtained from the subtree. Then S and SU have the same extension.

Proof of theorem: Let Sch, S, nde, Tdg, Tr, and, T(Tr), be as described. To avoid clashes of bound variables, the variables $bv_1, \ldots, bv_b$ in the declaration of T(Tr) will be replaced below by the distinct variables $bv'_1, \ldots, bv'_b$ that are distinct from any variables assigned to nodes of Tr. Join(Tr) is the join assertion for Tr for the given variable assignment. Join'(Tr) results from Join(Tr) by replacing occurrences of $bv_1, \ldots, bv_b$ by $bv'_1, \ldots, bv'_b$ respectively.

To prove the theorem it is sufficient to prove that the assertions

1. [**For all** s:SU] s:S, and
2. [**For all** s:S] s:SU,

are assigned **true** whenever all declared and defined degree constraints labelling edges of the augmented domain graph are satisfied by the membership of the declared sets.

By a **variable binding** BVar is meant a binding of some or all of the variables $v_1, \ldots, v_n$, $bv_1, \ldots, bv_m$, and $bv'_1, \ldots, bv'_b$ to members, or internal surrogates of members, of the sets that label the bottom nodes of Tr. A variable v_i is bound to a member of V_i, and a variable bv_i or bv_i', is bound to an internal surrogate of a member of BS_i. A variable binding BVar' is an **extension** of BVar if the variables bound by BVar are all bound to the same entities in BVar'.

Consider assertion (1). Should the node nde that S labels be a bottom node, then that (1) is

Let s be bound to the internal surrogate of a member of SU. Necessarily that internal surrogate takes the form of tup with its variables bound by a variable binding BVar, for which the assertion

3. tup:SU

is assigned **true**. It is sufficient to show that the assertion

4. tup:S

is also assigned **true** under BVar.

From (3) and the declaration of SU it follows that the assertion

5. tup:BDS **and [For some** v_{j+1}:V_{j+1}, ... , v_n:V_n]

(<v_1, ... , v_n>:T(Tr) **and** <bv_1,v_{j+1}>:IBS_1 **and** ... **and** <bv_b,v_{j+b}>:IBS_b)

is also assigned **true** under BVar. Therefore the following assertion is also assigned **true**:

6. **[For some** v_{j+1}:V_{j+1}, ... , v_n:V_n]

(<v_1, ... , v_n>:T(Tr) **and** <bv_1,v_{j+1}>:IBS_1 **and** ... **and** <bv_b,v_{j+b}>:IBS_b)

Necessarily there is an extension BVar' of BVar, in which the variables v_{j+1}, ... , v_n are bound to members of V_{j+1}, ... , V_n, under which the following assertion is also assigned **true**:

v_{j+1}:V_{j+1}**and** ... **and** v_n:V_n **and**

<v_1, ... , v_n>:T(Tr) **and** <bv_1,v_{j+1}>:IBS_1 **and** ... **and** <bv_b,v_{j+b}>:IBS_b.

From the declaration of T(Tr) the following assertion must also be assigned **true**:

v_{j+1}:V_{j+1}**and** ... **and** v_n:V_n **and**

[For some bv'_1:BS_1, ... , bv'_b:BS_b, bv_{b+1}:BS_{b+1}, ... , bv_m:BS_m] Join'(Tr) **and**

<bv_1,v_{j+1}>:IBS_1 **and** ... **and** <bv_b,v_{j+b}>:IBS_b.

Therefore there is an extension BVar" of BVar', in which the variables bv'_1, ... , bv'_b, bv_{b+1}, ... , bv_m are assigned to internal surrogates of members of BS_1, ... , BS_b, BS_{b+1}, ... , BS_m, under which the following assertion is assigned **true**.

v_{j+1}:V_{j+1}**and** ... **and** v_n:V_n **and**

bv'_1:BS_1**and** ... **and** bv'_b:BS_b**and** bv_{b+1}:BS_{b+1}**and** ... **and** bv_m:BS_m **and**

Join'(Tr) **and** <bv_1,v_{j+1}>:IBS_1 **and** ... **and** <bv_b,v_{j+b}>:IBS_b.

Since the assertion Join'(Tr) includes a conjunction

<bv'_1,v_{j+1}>:IBS_1 **and** ... **and** <bv'_b,v_{j+b}>:IBS_b,

and since the degree constraints are assumed to be satisfied, the internal surrogates to which bv'_1, ... , bv'_b have been bound are the same as the internal surrogates to which bv_1, ... , bv_b are bound. This follows from the fact that each of IBS_1, ... ,IBS_b has upper degree 1 on its value set. Necessarily, therefore, the assertion

7. v_{j+1}:V_{j+1}**and** ... **and** v_n:V_n **and**

bv_1:BS_1**and** ... **and** bv_b:BS_b**and** bv_{b+1}:BS_{b+1}**and** ... **and** bv_m:BS_m **and**

Join(Tr) **and** <bv_1,v_{j+1}>:IBS_1 **and** ... **and** <bv_b,v_{j+b}>:IBS_b

is also assigned **true** under BVar". Since Join(Tr) includes (4) as one of its conjuncts, (4) is also assigned **true** under BVar", and therefore under BVar as well.

Note that in this half of the proof the only use made of the assumption on the satisfiability of the degree constraints concerned the upper degrees of IBS_1, ... ,IBS_b on their value sets. That these are all 1 are the only degree assumptions necessary to show that (1) is assigned **true**.

Now consider the assertion (2). Let s be bound to the internal surrogate of a member of S. Again that internal surrogate takes the form of tup with its variables bound by a variable binding BVar. Under BVar the assertion (4) is assigned **true**. It is sufficient to prove that (3) is assigned **true** also. The following lemma is required for the proof.

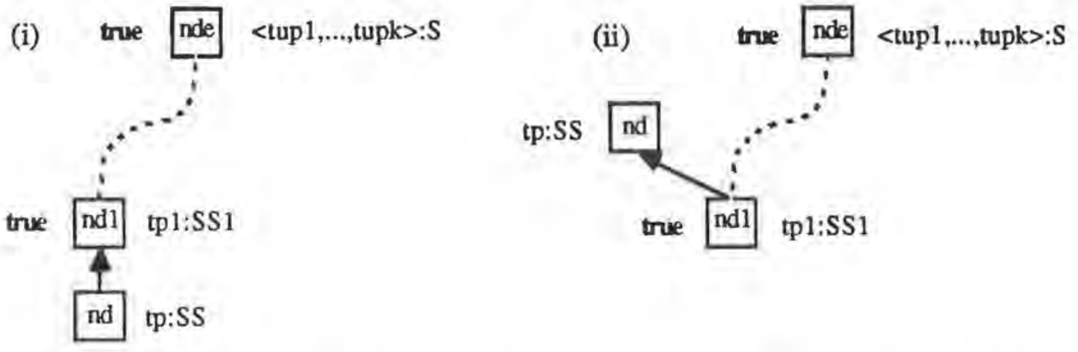
Lemma 2: There is an extension BVar" of BVar for which (7) is assigned **true**.

Proof of lemma 2: Recall that the assertion of a node nd of Tr is the assertion tp:SS, where SS labels nd and tp is assigned to nd. A node of Tr will be said to be true for a binding of variables, if the assertion of the node is assigned **true** for the binding. Each conjunct of (7) is the assertion of a such paths that there is an extension BVar" of BVar for which the end node of any path beginning at nde is true. Since the node nde is true for BVar, the result is true for paths of length 0.

Consider now a path nde, ... , nd_1, nd, where nd_1 may be nde. Assume that there is an extension BVar' of BVar for which all of the nodes nde, ... , nd_1 are true. It is sufficient to show that there is an extension BVar" of BVar' for which nd is also true.

Let tp:SS be the assertion of nd, and tp_1:SS_1 the assertion of nd_1. The latter is assigned **true** under the variable assignment BVar' since nd_1 is true by the induction assumption. The edge of the path connecting nd_1 and nd may have head nd_1 or head nd. The two possibilities are illustrated in figure 2.1.

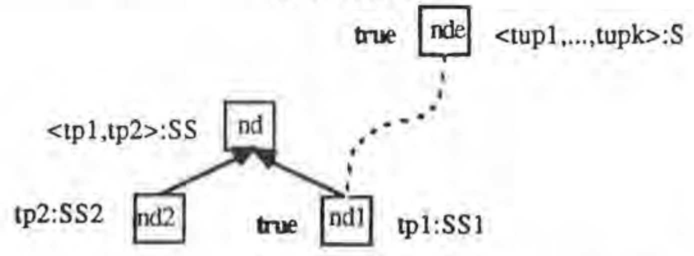
FIGURE 2.1



Consider (i) first. In this case SS is an immediate predecessor of SS_1, or of the arity domain of SS_1. Therefore tp_1 must either be tp, or of the form <... , tp, ...>. Since tp_1:SS_1 is assigned **true**, tp_1 must be in the domain of SS_1, so that tp:SS is assigned **true** also.

Now consider (ii). By lemma 1, the edge <nde_1,nd> necessarily has lower degree 1. Let nd_1, ... , nd_d, $d \geq 1$, be the nodes of Tr for which <nd_i, nd> is an edge of Tr, and let tp_i:SS_i be the assertion of nd_i. Therefore tp is tp_1 if d=1, and is otherwise <tp_1, ... , tp_d>. The situation is illustrated in figure 2.2 for d=2.

FIGURE 2.2



Since the edge <nd_1, nd> has lower degree 1, SS has lower degree 1 on SS_1. The theorem being proved assumes that all degree constraints are satisfied by the memberships of the declared sets. Therefore there must be an extension BVar"of BVar', binding all the variables occurring in tp, under which the assertion of nd is assigned **true**.

End of proof of lemma 2

In the first part of the proof of the theorem an argument was given that concluded that if (3) is assigned **true** under a variable binding BVar, then there exists an extension BVar" under which (7) is assigned **true** also. In this second part, the lemma establishes that if (4) is assigned **true** under a variable binding BVar, then there is an extension BVar" of BVar under which the assertion (7) is assigned **true**. Now consider the reverse of the argument used in the first part of the theorem. It is elementary to establish that (6) is necessarily assigned **true** under BVar", and therefore under

End of proof of theorem.

In the first half of the proof of the theorem, the only degrees that were required to be satisfied are the upper degrees of 1 of $IBS_1, \ldots, IBS_b$ on their value sets. It is possible, therefore, to relax the definition of identifier by requiring it to have only the degrees <1,*> on the set it identifies, rather than <1,1>. But efficiency considerations often dictate that there should be a single identifying string for each member of a primitive base set.

The more interesting conclusion from this observation is, however, that SU is a subset of S, no matter what node S labels, or no matter from what subtree of the domain graph Tr was constructed. The worst that can happen is that SU loses members of S. The theorem establishes sufficient conditions that SU does not lose members. Therefore, rather than saying T(Tr) is correct for S when SU has the same extension as S, the tradition of the relational model could be followed and T(Tr) could be said to be **lossless** for S.

The assumption of the theorem concerning the satisfiability of the degree constraints will be discussed in 5.1.

3. Extensions and Applications

In this section applications of the SET model and DEFINE will be described. To accomplish some of the applications, the model and the language must be extended from the simple form described in [Gil87]. In 3.1 DEFINE will be presented as a query language. In 3.2 a kernel schema is sketched that is intended to meet the demand (3) of section 1. In 3.3 DEFINE is extended to admit recursively defined sets. In 3.4 a beginning is made on showing that the SET model can satisfy the demand (4) of section 1, and to that end object oriented programming is briefly discussed. In 3.5 it is argued that the purposes of the universal relation model are better served by the SET model. In 3.6 a beginning is made on showing that the SET model may also satisfy the demand (5) of section 1.

3.1. DEFINE as a Query Language

DEFINE as a specification language for a set schema based on the SET model has been amply demonstrated. But clearly it can also be used as a query language provided that the management system supporting the language and based on the model can respond to a request for listing the members of a set. A command **list** S results in the system listing the identifiers of members of S, no matter whether S has been declared as base or defined. For example, should S be E, the system responds by listing the employee numbers of E, while should S be TE, the system responds by listing the tuples that are its members.

Some sets will, of course, just be declared for the purposes of a single query and declarations of them should not be construed as additions to the set schema of an enterprise. In this case S need not be the name of a declared set but can take the form { domain assertions | intension } of the domain and intension declaration of a defined set. For example, a listing of the courses taught by managers of academic departments results from the following query:

list {w:C | [**For some** <x,y>:MA] x:ICT:w }

At the same time additional temporary declarations of sets can be made in a **where** clause for the **list** command. For example, a listing of members of academic departments who are not instructors results from the query:

list { x:E | [**For some** y:AD] (x:ED:y **and not** x:I) } **where** AD={ y:D | [**For some** w:VC#] <y,w>:C | }, I={ x:E | [**For some** w:C] x:ICC:w }.

In 2.3 of [Gil87], a functional notation was introduced that can be conveniently used at times in queries. For example, a listing of the members of the English department is obtained from the following query:

list { x:E | x:ED:{:DN:'ENGLISH'} }.

The functional notation {:DN:'ENGLISH'} stands for the set with member the set DN associated with 'ENGLISH'. In the form {:NM:v} it is a function which for a value of v in VN will return the empty set if v is not the name of a department, and will other wise return the singleton set containing the department. The notation can also be used without arguments as in the following query:

list { x:E | x:ED:{:AD} }

where AD={ y:D | [**For some** w:VC#] <y,w>:C | }

that lists the members of academic departments.

The functional notation provides a substitute for the **GROUP BY** and the **GROUP BY** with **HAVING** features of the SQL language.[Date82] It has the benefits of the functional notation of [Ship81, LyKt86] but provides a simpler notation for the two functions that can be defined from any binary association; for example, {:ED:y} is the set of employees assigned to the department y, while {x:ED:} is the set of departments to which the employee x has been assigned. The semantics for the notation is provided in the manner of the functional notation of [Gil77].

An essential variation of the concept of set is that of **ordered set**. The distinction between a set and an ordered set is that the former has members that are all distinct, while the latter can have elements that are the same, since associated with an element is its position in the set. The tuples that have been used extensively are examples of ordered sets with elements that have been stated by enumeration. However, just as it is possible to declare a defined set with an intension that is a condition for membership of the set, so must it be possible to declare ordered sets in a similar fashion. For example, < x:E | **true** > is a tuple consisting of the members of E without an order specified. The notation can be simple extended to permit the specification of an order if that is desired.

With a notation for ordered sets available, it is possible to introduce associations with tuples as one of their immediate domain predecessors. For example, a count association on tuples that gives the number of elements in a tuple is needed. The notation used in the following query is commonly used for this association:

list { n:INT | < x:E | **true** >:COUNT:n }

The set declared in the query has a single member consisting of the integer that is the number of members of the set E.

The functional notation introduced can be used with ordered sets as well. For example, the query

list { y:D, n:INT | <:ED:y>:COUNT:n }

results in a listing of the pairs <y,n> for which y is a department and n is the number of employees assigned to the department. The functional notation <:ED:y> used in this context denotes, for each department y, the tuple of employees ED associated with the department. The notation can be nested as in the following query:

list { y:AD, n:INT | <{:M:y}:ICT:>:COUNT:n }
    where AD={ y:D | [**For some** w:VC#] <y,w>:C | }

that provides a list of those pairs <y,n> for which n is the number of courses being currently taught by the manager of the academic department y.

Another essential use of ordered sets arises in queries concerning totals. For example, assume that an association SAL with domain ExINT has been declared that associates a unique salary with each employee. Then a query as to the total salaries of employees by department can be expressed:
list { x:D, n:INT | <{:ED:x}:SAL:>:SUM:n },
where SUM is an association between tuples of integers and a single integer, the latter being the sum of the elements of the former. Thus n is the sum of the integers appearing in the tuple <{:ED:x}:SAL:> for a given department x.

The substantial computations that can sometimes be required for the processing of queries, can be reduced through methods of query optimization [Ull80]. The methods described there for relational algebra queries can be translated into methods for DEFINE [Krey87].

Another issue raised in [Ull80] is that of safe queries; these are queries that request the listing of such a number of tuples as it is reasonable to expect a management system to process. A query requesting a listing of INT or STR, for example, would not be a safe query. Because of the necessity of declaring the range of all variables, it is less likely that a user will inadvertently pose an unsafe query in DEFINE than in the relational calculus, although, of course, one can be posed. Since the detection of safe queries is undecidable [VaTo87], sufficient conditions for safe queries must be sought to assist in protecting the management system from the errors and misunderstandings of its users.

As a query language DEFINE does satisy the dual demands of precision and completeness. Whether it is sufficiently transparent for general purpose querying raises issues that cannot be dealt with here. Queries posed in a language that is intended to be comprehensible to unsophisticated users could be translated into DEFINE, if that is desired. Such queries may even be abbreviated, with the management system using the domain graph to disambiguate when necessary. However, as stressed in [LuKl86], it must be recognized that the responsibility for posing queries that involve essentially complicated definitions cannot be removed from the user, the user can only be assisted.

3.2. A Kernel Schema

The SET model can provide a provably sound foundation for databases that can reference and describe themselves. Such a base will be sketched here, while the basis for its justification as "provably sound" will be sketched in 4.1.

In figure 2.9 of [Gil87] a summary of the declarations of all the sets of the schema for Simple University was given in a table. That table should be defined from a set schema, called a **kernel schema**, in the same manner that the tables for Simple University were defined from its set schema. The kernel schema can be thought of as the schema that will support the enterprise of set schema design. It must be capable, therefore, of storing the information contained in any set schema, including itself. The design of such a kernel schema, as a set schema that is at the same time a schema for its own "data dictionary", was one of the orginal motivations for developing the SET model. An incomplete kernel schema has been found to be useful in the teaching of the entity-relationship approach in undergraduate and graduate courses in database design since 1979 [Gil86b]. The equivalent of such a schema for relations is described in [Mark85].

The rudiments of a kernel schema are evident from figure 2.9 of [Gil87]. First the set of declared sets must be declared:
DSET={ DSET || all declared sets }.
DSET is a primitive base set that has as its members all declared sets. Since DSET is itself a declared set, it will be a member of itself. The first four sets declared for Simple University, namely STR, INT, L, and $\leq$, are also in the kernel schema, and are members of DSET.

The identifier for DSET is the declaration attribute DEC that associates the declaration of a set with the set. The declaration of the value set VDEC for the attribute requires the definition of the full syntax of assertions of DEFINE as well as of declarations, and is beyond the scope of this paper Assuming the declaration of VDEC, however, the declaration of DEC can be given:
DEC={ DSET, VDEC | <1,1>, <0,1> | the declaration attribute }.

Although the declaration of a set is its ultimate identifier, some abbreviation of the declaration is necessary for a reasonable syntax, and the name of a set, which is part of its declaration, is so used. In the set schema for SU, each set has a unique name so that the name attribute of sets could be used as an identifier in that model. But for practical modelling it is essential that some duplication of names be permitted. For example, there are likely to be several attributes on different sets all with the same value set VN, and it should be possible to call them all NAME.

When duplicate names are allowed, the name of a set appearing in isolation need not uniquely identify the set. In the context of an assertion of DEFINE, however, the name of a set should identify the set, or at least narrow its identification sufficiently to permit the system to request clarification from the user, or make intelligent guesses. The following rules for naming sets, using definitions given in 2.10 of [Gil87], appear to satisfy this requirement:

1. Each primitive set must have a name distinct from the name of any other set.
2. Sets of the same arity, with a common arity predecessor that is not a value set, must have distinct names.

The need for the first rule is transparent. The second rule reflects the fact that members of value sets are generally not entities about which information is recorded, but are used in human-machine communication to record and retrieve information about other sets. Thus a variety of attributes all with the same value set VN, but on different sets, may have the same name NAME. But different attributes of the same set, whether inherited from the arity domain of the set or not, must have different names. If in a kernel schema it is necessary to declare attributes on sets of strings, for example on the set VDEC, then distinct names can be given to the attributes although by rule 2 they are not required to be distinct.

From the DEC attribute it is possible to declare the immediate domain predecessor, the immediate define predecessor, and the immediate predecessor associations as defined sets with domain DSETxDSET. The names given to them are assumed to be IDMP, IDFP, and IP, respectively. The set of primitive sets can then be declared as a defined set:
PSET={ x:DSET | **not** [**For some** y:DSET](x:IDMP:y **or** x:IDFP:y) | }.
The need for the assertion x:IDFP:y will be apparent shortly.

The ontology of sets, described in 2.7 of [Gil87] for a simple form of the SET model, admitted as primitive defined sets only the primitive value sets such as STR and INT. Other primitive defined sets, however, are needed for a kernel schema. They are not members of PSET because, although they do not have an immediate domain predecessor, they do have an immediate define predecessor. The most important of these is the set of entities that are members of members of PSET:
UV={ x:UV | [**For some** y:PSET] x:y | }.
UV is a defined set since its intension is stated in DEFINE, but it is a primitive set since there is not a previously declared set from which its extension can be drawn, since its members are drawn from the extensions of all the primitive sets. The members of UV form the basic universe of the model for the enterprise described in a set schema. All other entities are nested tuples of members of UV.

One of the important ways in which SET differs from the similarly motivated models of [LyKe86, BCP86] is that UV is a primitive defined set in SET, while in the other models it is a primitive base set. It is possible to declare UV as a primitive base set in SET, although that would offend the first principle of conceptual modelling stated in 2.14 of [Gil87], since an identifier for UV cannot be declared apart from the identifiers for the primitive sets.

The effect on the language DEFINE of admitting sets such as UV is profound. In the simple form of the model each entity that was a member of a set could be uniquely "typed" as a member of the arity domain of the set. The language DEFINE is monomorphic without such sets, and polymorphic with them, since a member of UV is also a member of some other primitive set [CaWe85, Ing86].

A second related effect concerns the arity of sets. Each set that can be declared in the simple form of the SET model has as members only tuples of length the arity of the set. That is no longer the case in the extended model. For example, the union of any two sets of different arity can be declared as a primitive defined set. By definition the set will have arity 1, although its members are not tuples of length 1, or even tuples all of the same length.

3.3. Recursively Defined Sets

It is necessary to declare as sets of the kernel schema some of the associations that were defined informally in section 2 of [Gil87]. For example, domain predecessor, the transitive closure of immediate domain predecessor, must be declared as a recursively defined set:
DMP={ DSET, DSET | **[For all** x:IDMP] x:DMP **and**
[For all u,v,w:DSET] **if** (<u,v>:DMP **and** <v,w>:DMP) **then** <u,w>:DMP |
the domain predecessor association }.

Although DMP is a defined set, its domain declaration does not declare variables. The declaration can be recognized as recursive from the fact that the machine readable portion of it between the two vertical bars is not a degree declaration, but rather an assertion of DEFINE.

The declaration appears to offend the requirement that the predecessor association be acyclic, since DMP is used in its own declaration. But when the declaration is properly interpreted, this is not so. The meaning of such a definition is that DMP is the smallest transitively closed subset of DSETxDSET that includes IDMP. A formal expression of this meaning requires a second order set theory of the kind introduced in [Gil86a]. For example, using second order variables, DMP can be declared:

```
DMP={ z:DSETxDSET | [For all X⊆DSETxDSET]
        if ( [For all y:IDMP] y:X and
             [For all u,v,w:DSET] if (<u,v>:X and <v,w>:X) then <u,w>:X )
        then z:X | }.
```

Note the quantifier for X ranges over subsets of DSETxDSET. The fact that this range is not a previously declared set, but rather all possible subsets of a previously declared set, makes it a second order variable that cannot be replaced by a first order variable without explicitly declaring every possible subset.

The form of the DMP declaration is one that has been widely used in logic for defining recursive sets. The assertions used in the **if ... then** clauses take the form of pure Horn clauses, that now form the basis for the programming language PROLOG. Restricting the intension of recursively defined sets to the use of pure Horn clauses avoids the complications that are necessary when PROLOG is extended to include nonHorn clauses [ABW86]. Given the ability to use any assertion of DEFINE in the intensions of nonrecursively defined sets, no loss of expressive power results.

The set TUP of all tuples of members of the set UV can be defined recursively in the usual way. With that set available, COUNT can also be defined recursively with domain TUPxINT, as can also SUM.

3.4. Updates and Data Processing

Once a set schema has been declared, a user must be able to add members to any declared set, and a command must be available for doing this. The form of a suitable command is:
Add tup **to** S **where** assert
where the variables occurring in tup occur unbound in the assertion assert; the latter provides an appropriate description of the entity to be added to the declared set S. A companion command removes an entity from a set:
Drop tup **from** S **where** assert.

In the most elementary form of the **Add** and **Drop** commands, S must be restricted to being base, and the meaning of the commands when S is defined must be reduced to the elementary form. For humans are responsible only for the membership of base sets, while the system is responsible only for the membership of defined sets. A command to add or drop a member of a defined set must, therefore, be interpretable as one or more commands to alter the membership of base sets. But how such commands are to be interpreted requires more research; the equivalent problem for the relational model is the updating of virtual relations, or views [Date83, Kell85].

In a kernel schema, the declaration of a set is equivalent to adding the set to the primitive base set DSET. The sets declared in the kernel schema itself are assumed predeclared and therefore have members determined by the schema. But when a member is added to DSET by a user, the declaration attribute DEC must be updated. During such a transaction, the system must ensure that cycles are not introduced into the immediate predecessor association; it is sufficient for the system to ensure that the immediate predecessors of a set are declared before the set can be declared.

In 1.1 of [Gil87] it was argued that a fully unified model of an enterprise is needed that at the same time can give a conceptual view of the enterprise, a user's view of data as it is presented, a data administrator's view of data as it is stored, and a programmer's view of the processing of the data. A commonly encountered perception of the latter is that the dynamic nature of data processing prevents the representation of a programmer's view in a "static" model such as SET.

In as much as the commands **Add** and **Drop** are extensions to DEFINE, the perception is soundly based. For example, it is not possible to represent the addition of a new employee to the primitive base set E as an association in the same sense that the assignment of an employee to a department is represented by ED: First, the new employee, before being added to the set E, is not a member of any set, so that a domain for **Add** is not available; and second, the effect of adding the employee to E is not to change its intension, but only its extension. When an entity that is a member of the domain of a nonprimitive base set is added to the set, a change of extension is the result. Such an application of **Add** can be regarded as an association between two states of the extensions of the declared sets. But that requires treating the extensions of all the declared sets as a tuple of tuples, something theoretically possible but often impractical.

A recognition of the need for **Add** and **Drop**, and the form of them given above, is one of the contributions of [Morr].

On the other hand, each defined set can be regarded as a "program" that determines the membership of the set from the membership of its immediate predecessors. In this sense, therefore, the perception of the SET model as only supporting static descriptions is not soundly based. Typical data processing requires in part the use of the **Add** and **Drop** commands, sometimes imbedded in assertions, but also a substantial number of defined sets.

The two cornerstones of object-oriented programming, encapsulation and inheritence [Cox86], are central features of the SET model. These are the features also of the object-oriented data modelling methodology for information systems described in [LyKe86]. The objects of the SET model are the members of declared sets. The **list**, **Add**, and **Drop** commands provide the procedural element, as the comparable commands do in [LyKe86]. In the LORE approach to object oriented programming[BCP86], the procedural element is introduced through message passing, traditionally a part of object-oriented programming.

3.5. The Universal Relation Model

The SET model may shed some light on a controversy involving the universal relation model [Knt81, Ull82, Ull83, Knt83]. A universal relation for a relational schema is a user view that is intended as an assistance to users in the formation of queries in a relational model, as a tool for a database managment system for the resolution of ambiguous queries, and as a foundation for relational dependency theory. A question naturally raised by the model is whether universal relations can be constructed for an arbitrary relational schema.

The domain graph method of table design will construct one table for all the declared sets that label nodes of a 1-connected subtree of a domain graph for a set schema. If it were possible to have every set of interest label nodes of a single such subtree, then the method would result in a single "universal" relation. So the trick needed to produce a universal relation for a set schema is a way of converting lower degrees of 0 into lower degrees of 1. The introduction of pseudo entities into some of the primitive base sets and of special strings into some of the value sets is such a trick.

Consider, for example, the selected <1,1>-connected subtrees illustrated in figure 3.4 (i). Restoring the edge from D to ED and the edges connecting C to EDC.C to ICC appearing in figure 2.5, results in three 1-connected subgraphs. An edge from D to C can be restored if every department is made responsible for at least one course. Such a course can be invented for every nonacademic department by reserving a special number for it. Every member of a nonacademic department can be assumed to be competent to teach the fictious course of the department. With the edge from D to C restored, the number of subtrees is reduced to two. That number can be reduced to one by inventing, for each department, an employee competent to teach all the courses of the department and currently teaching any course not being taught by real members of the department.

Therefore a universal relation for Simple University is possible. Although the universal relation need only exist in a user's mind, the necessity of having to invent a number of fictious entities raises questions about its value. Furthermore, the three purposes for the universal relation model may be better served by the SET model.

Query formation in the SET model enjoys the advantage of depending upon a conceptual schema modelling an enterprise, rather than upon a relational schema modelling a presentation of data. The domain graph of a set schema can be used in place of the hypergraphs of the universal relation model to assist in the disambiguation of queries which use ambiguous names of sets. Finally a dependency theory for tables based on the SET model takes a particularly simple form since it can make use of the trees from which the tables are defined, and since there is a unique undirected path from any node of a tree to any other node. Dependencies are determined from the degrees labelling the edges of the trees.

3.6. Incomplete Information and Defaults

A set schema specifies what data should be available. Null values provide a means for recording that data is not available. It is necessary, however, to distinguish between two kinds of null values, the not-applicable null value **N/A** and the don't-know null value **D/K**. For the former is used to record that a value cannot be known; for example, the fictitous entities introduced in 3.5 can be regarded as not-applicable null values. The don't know null values, on the other hand, are used to indicate that a value should be known, but is not; it can be regarded as a default value that the system provides in the absence of a user-specified value.

Defaults have been the subject of numerous papers, with proposals for treatment that are not encouraging of confidence [McC80]. The thesis [Ethn86] provides a good summary. Their treatment in a model such as SET that recognizes the distinction between defined and base sets, on the other hand, is quite simple and captures exactly the semantics described in the previous paragraph.

Let the primitive defined set D+ be declared

D+={ x:D+ | x:D **or** x=**D/K** | }.

Let DED, "defined ED", be declared:

DED={ x:E, y:D+ | x:ED:y **or** (y=**D/K and not [For some** w:D] x:ED:w) | }.

If DED is used in place of ED in the declaration of TE, for example, then a row of TE for a given employee will display **D/K** if a user has not specified a department for that employee. When the ED association is updated by a user to assign that employee to a department, the default value **D/K** will automatically be replaced by the department's name, since both DED and TE are defined sets with membership maintained by the system.

The simplicity of this solution for defaults suggests that the use of the SET model to deal with other problems of incomplete information may be worth exploring. For example, a proper treatment of identity allows for the introduction of distinct internal surrogates for which insufficient information is available to permit their identification. An insufficiently specified update of the following kind'

Add <x,y> **to** ED **where** x:E#:1234 **and** (y:DN:'ENGLISH **or** y:DN:'PHYSICS')

can then be accepted as stating limiting conditions on the internal surrogate introduced for y. A full treatment of this proposal to be provided elsewhere requires a careful treatment of identity [KhCo86].

4. Consistency and Integrity

In the last section a sketch was given of a kernel schema that can reference and describe itself. It remains to be shown however that such a schema is a provably sound foundation for databases. A basis for a proof of this is given in 4.1. In the satisfaction and maintenance of the integrity constraints of SET are discussed.

4.1. Consistency

The meaning given to "consistency" in chapter 2 of [Date83] is at variance with the meaning given here. "Consistency" is used here in the sense employed in logic and mathematics: A theory is consistent if it is not possible to conclude that both a sentence and its negation are true for the theory. The meaning given in [Date83] is included in the meaning of "integrity" used here.

In a consistent database in which all integrity constraints have been satisfied, it is not possible to conclude that a sentence and its negation are both true. But a consequence of the high level of abstraction tolerated in kernel schemas, is that such schemas might satisfy very strong integrity constraints but neverthless not be consistent, unless care is exercised in its implementation. Consider, for example, the following declared set:
R={ x:DSET | **not** x:x | };
the members of R are those declared sets that are not members of themselves. The set E for example, is a member of R, while the set DSET is not.

There is nothing inherently wrong with the declaration of R, and it is even conceivable that it might prove useful for some purposes, but the set has a notorious past: It is the basis for the Russell paradox that shook the foundations of mathematics at the turn of the century. The paradox arises when one asks whether R is a member of itself. Using naive reasoning it can be concluded that R is a member of itself, and that it is also not a member of itself. Thus a database system that permits the declaration of R, or some similar set, and that permits naive reasoning will be inconsistent, even though it enforces very strong integrity constraints. This is what [Blac85] demonstrated for the semantic networks described in [Sowa84].

Since the discovery of the Russell and other paradoxes, a number of set theories have been invented that maintain consistency by restricting the declaration of sets in a variety of ad hoc fashions. In [Gil86a] it is argued that it is the reliance on naive reasoning, rather than the definition of R, that is the source of the paradoxes.

An assumption employed in naive reasoning is that every sentence is either true or false. A careful examination of this assumption shows, however, that it can only be made for atomic sentences, or what is called in theorem proving, ground sentences. The truth value for a complex sentence must be reduced to the truth values of simpler sentences, and be ultimately expressed in terms of the truth values of atomic sentences. If it is not possible to ground a truth value for a complex sentence in atomic sentences, then the sentence receives no truth value. It is this provably sound resolution of the paradoxes that is used to maintain the consistency of the SET model.

This simple resolution of the paradoxes has an important consequence. Some sentences, such as R:R, cannot be grounded in atomic sentences and therefore have no truth value assigned to them. On the basis of the semantics, the correct answer is "no" to a query as to whether R is a member of itself, since R:R is not true; it is also the correct answer to a query as to whether R is not a member of itself, since R:R is not false. Negation cannot therefore be understood in the sense of "negation as failure" as suggested in [Clar78] and criticized in [Flan86]; the negation of an assertion is assigned a truth value if and only if the assertion has been assigned a truth value, and it then receives the opposite truth value. In this sense a negated assertion derives its meaning from the assertion that can be obtained from it by driving negations down to the level of assertions of membership in base sets. All assertions expressing membership in defined sets must be replaced with their intensions, with membership in recursive sets requiring "recursive" treatment. Once a query has been grounded in this sense, a truth value can be determined for it, but not before. Therefore a management system can respond to a query as to whether R is a member of itself, with a report of failure to ground the query.

4.2. Integrity in the SET Model

Two kinds of integrity constraints are expressed in the domain and degree declarations. The maintenance of these constraints presents different problems to a management system.

The maintenance of domain constraints is a relatively simple matter. There are no constraints on adding new members to primitive base sets such as E or D. Domain constraints affect users actions only when a new member is to be added to a nonprimitive base set such as ED; then the new member must be selected from the domain of the set.

The degree constraints on base sets present problems of a different character. There is first the question as to whether the degree constraints can be satisfied at all. Consider, for example, a base association R with domain PxQ and degrees <1,1> and <1,1>, where
P={ x:INT | x=1 }, and Q={ x=INT | x=1 **or** x=2 | }.
Clearly the degrees can never be satisfied since they require that P and Q have the same number of members. The satisfaction problem for degree constraints is unsolvable when a sufficiently rich form of DEFINE is available for defining sets; for example, the halting problem for a Turing machine can be expressed as the problem of determining whether two defined sets have the same number of members. But degree constraints satisfying one simple condition can always be satisfied.

A **membership specifying path** in a domain graph is a directed path in which the first node is labelled with a primitive defined value set, and every edge <nde1, nde2> for which nde2 is labelled with a base set has lower degree 1. For example, the nodes labelled with INT, P, and R form a membership specifying path, as do the nodes labelled with INT, Q, and R.

Theorem: Consider a set schema for which no base set labels a node of a membership specifying path. Then the degrees of the set schema are satisfied if every set that does not label a node of a membership specifying path is empty.

Proof: By induction on the maximum length of directed paths in the domain graph. If that length is 0, then every set that does not label a node of a membership specifying path is necessarily primitive base and can therefore be empty. Consider now a schema in which the maximum length of directed paths is greater than 1, and let S be a nonprimitive set labelling a node that is not in a membership specifying path. Should S be defined, then each immediate domain predecessor of S may be assumed to be empty, so that S may be assumed to be empty also. Should S be base, then each immediate domain predecessor of S that does label a node on a memberships specifying path, may be assumed to be empty. The lower degree of any edge from a node labelled with an immediate domain predecessor of S to the node labelled with S is necessarily 0 if the immediate domain predecessor labels a node of a membership specifying path. Therefore S may in this case also be assumed to be empty.
End of proof

Assuming that the degree constraints of a set schema can be satisfied, they nevertheless present special problems to a management system that must maintain them. The lower degree constraints are the most difficult since lower degrees of 1 may require that new members be added to an association such as ED in response to the addition of a new member to E or a new member to D. Combined with upper degree constraints, lower degree constraints may even compel the addition of a new member to a primitive base set. For example, adding a new member to D will compel at the very least a changing of the membership of ED, and could compel the adding of a member to E. Since there is no limit on the size of 1-connected subgraphs, there is no limit on the number of **Add** and **Drop** commands that must be executed in a transaction that transforms a database state in which all degree constraints are satisfied, to another such state. However, the existence of these subgraphs permits transactions to be checked for degree preservation before being committed. The provision of assistance to users in the management of degree preserving transactions is an interesting research problem. Aspects of the problem are addressed in [ApPu87].

5. Conclusions

The SET model described in [Gil87], when extended in the fashion described in this paper is suitable for modelling all aspects of data processing as well as for handling the complexities of incomplete information. It is an advance over similarly motivated models in as much as the consistency of the model can be demonstrated.

BIBLIOGRAPHY

[ABW86] APT, K., BLAIR, H., AND WALKER, A. Towards a Theory of Declarative Knowledge. *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*. Washington, D.C. 546-629. 1986.

[ApPu87] APT, KRYSZTOF 9. AND PUGIN,JEAN-MARC. Maintenance of Stratified Databases Viewed as a Belief Revision System. *Proc. Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 136-145. March 1987.

[BCP86] BENOIT, CHRISTOPHE, CASEAU, YVES, AND PHERIVONG, CHANTAL. The LORE Approach to Object Oriented Programming Paradigms. Memo C29.0, Laboratoires de Marcoussis, Centre de Recherches de la C.G.E. April 15, 1986.

[Blac85] BLACK, MICHAEL JULIAN. Naive Semantic Networks. Final Paper, Directed STudy in Computer Science. Dept of Comp. Sci., Univ. of B.C. Jan 22, 1985.

[CaWe85] CARDELLI, LUCA, AND WEGNER, PETER. On understanding types, data abstraction, and polymorphism. *ACM Comp. Surveys* 17, 4 (Dec. 1985), 471-522.

[Chen76] CHEN, PETER PIN-SHAN. The Entity-Relationship model - toward a unified view of data. *ACM Trans. Data Base Syst.*, 1, 1 (March 1976), 9-36.

[Chen77] CHEN, PETER PIN-SHAN. The Entity-Relationship model - A basis for the enterprise view of data. AFIPS Conference Proceedings, Vol. 46, 1977 NCC.

[Clar78] CLARK, K.L. Negation as Failure. H. Gallaire and J. Minker (eds.), Logic and Data Bases, Plenum, New York, 1978.

[Cox86] COX, BRAD J. Object-Oriented Programming. Addison-Wesley, 1986.

[Date81] DATE, C.J. An Introduction to Database Systems, Vol.I, 3rd ed. Addison-Wesley, 1981.

[Date83] DATE, C.J. An Introduction to Database Systems, Vol.II. Addison-Wesley, 1983.

[DKM86] DE TROYER, O., KEUSTERMANS, J., AND MEERSMAN, R. How Helpful is an Object-Oriented Database Model?. [DiDa86]. 124-132.

[DiDa86] DITTRICH, KLAUS, AND DAYAL, UMESHWAR. (Eds) Proc. International Workshop on Object-Oriented Database Systems. ACM and IEEE. Sept 23-26, 1986.

[Ethn86] ETHERINGTON, DAVID WILLIAM. Reasoning with Incomplete Information: Investigations of Non-Monotonic Reasoning. PhD Thesis, Dept. Comp. Sci., Univ. of B.C. April 1986.

[Flan86] FLANNAGAN, TIM. The Consistency of Negation as Failure. *Journal of Logic Programming*, 2 (1986), 93-114.

[Gil77] GILMORE, PAUL C. Defining and computing many-valued functions. Parallel Computers - Parallel Mathematics. FEILMEIER, M. (ed.), North-Holland (1977), 18-23.

[Gil86a] GILMORE, PAUL C. Natural deduction based set theories: a new resolution of the old paradoxes. *J. Symb. Logic*, 51, 2 (June 1986), 393-411.

[Gil86b] GILMORE, PAUL C. Class notes for CPSC 404. Dept of Computer Science, Un. of B.C. August 11, 1986.

[Gil87] GILMORE, PAUL C. The SET Conceptual Model and the Domain Graph Method of Table Design. Dept of Computer Science Tech. Report 87-7, Un. of B.C. March 1987.

[Kell85] KELLER, ARTHUR M. Updating Relational Databases Through Views. Stanford Computer Science Department Tech. Report STAN-CS-85-1040. Feb 1985.

[Knt81] KENT, WILLIAM. Consequences of Assuming a Universal Relation. *ACM Trans. Database Syst.*, 6, 4 (Dec 1981), 539-556.

[Knt83] KENT, WILLIAM. The Universal Relation Revisited. *ACM Trans. Database Syst.*, 8, 4 (Dec 1983), 644-648.

[KhCo86] KHOSHAFIAN, SETRAG N. AND COPELAND, GEORGE P. Object Identity. [Meyr86]. 406-416.

[Krey87] KREYKENBOHM, MICHAEL. Optimizing DEFINE Queries. Term Project, Dept of Computer Science, Un of B.C. March 1987.

[LuKl86] LUK, W.S. AND KLOSTER, STEVE. ELFS: English Language for SQL. *ACM Trans. Database Syst.*, 11, 4 (Dec 1986), 447-472.

[LyKt86] LYNGBACK, PETER, AND KENT, WILLIAM. A Data Modelling Methodology for the Design and Implementation of Information Systems. [DiDa86]. 6-17.

[Mark85] MARK, LEO. Self-describing database systems - formalization and realization. Technical Report - #1484. Dept. Comp. Sci. Un. Maryland. April, 1985.

[McC80] Circumscription - A Form of Non-monotonic Reasoning. *Artificial Intelligence* 13, 295-323. 1980.

[Meyr86] MEYROWITZ, NORMAN. (ed.) Proc. Object-Oriented Programming Systems, Language and Applications. ACM Sigplan Notices, 21, 11 (Nov 86).

[Morr] MORRISON, RODERICK. Implementating a Set Based Data Model and its Data Definition/Manipulation Language. PhD thesis, Department of Computer Science, Un. British Columbia. *In progress.*

[Ship81] SHIPMAN, DAVID W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6, 1 (March 1981), 140-173.

[Sowa84] SOWA, J.F. Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, 1984.

[TrLo87] TRYON, D.C.; AND LOYD, D.G. Information Resource Depository: History, Current Issues, and Future Directions. Pacific Bell, A Pacific Telesis Company. Presentation to Canadian Information Processing Society, Vancouver, Canada, February 1987.

[Ull80] ULLMAN, JEFFREY D. Principles of Database Systems. Computer Science Press, 1980.

[Ull82] ULLMAN, J.D. The U.R. Strikes Back. *Proc. ACM Symp. Principles of Databse Systems.* 1982, 10-23.

[Ull83] ULLMAN, J.D. On Kent's "Consequences of Assuming a Universal Relation. *ACM Trans. Database Syst.*, 8, 4 (Dec 1983), 637-643.

[VaTo87] VAN GELDER, ALLEN AND TOPOR, RODNEY W. Safety and Correct Translation of Relational Calculus Formulas. *Proc. Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* 313-327. March 1987.