The Ultimate Data Support System

Karl Soop

Updated 2010-12-06

Abstract

This paper reports a number of results, issues, and decisions that emerged in the 1978 design of a relational data-support system for engineering usage and suggests topics for future research. A number of the decisions were aimed at drastically simplifying the underlying concepts (e.g. basing the theoretical model solely on mathematical logic), which allowed a very straightforward implementation. Applications and practical experience with the support system have proven the soundness of the approach.

Acknowledgment

Messrs. Gérard Duwat, IBM Marseille, France and Alain Massabo, Aerospatiale Marignane, France formed with the author at IBM Nordic Lab, Sweden, the original design team of the relational prototype. Ideas and results reported here represent the effort of the whole team.

Table of Contents

1. Introduction
2. The Binary Relation
3. The Basic Object
4. Basic Operators
    - 4.1 Set Operators
    - 4.2 Pair Operators
    - 4.3 Conversion Operators
5. Expressions
6. Statements
7. Abstract Relations
    - 7.1 The Tester
    - 7.2 Double Composition
    - 7.3 The Accessor
    - 7.4 The Generator
    - 7.5 Universal & Fundamental Relations
    - 7.6 Summary
8. Relational Programs
9. Evaluability
    - 9.1 Set Operators
    - 9.2 Composition
10. Grouping Operations
    - 10.1 Relation Names as Objects
    - 10.2 The Grouping Operator
    - 10.3 Computational Relations
    - 10.4 Set-Property Composition
    - 10.5 Group Composition
11. Classes of Objects
    - 11.1 Semantic Classes
    - 11.2 Domains
    - 11.3 Roles
    - 11.4 Semantic Checking
12. Implementation Issues
    - 12.1 The Vocabulary
    - 12.2 Relational Models
    - 12.3 The Ternary Model
    - 12.4 The Cylindrical Memory
13. Conclusion

References

1. Introduction

Characteristic of a sophisticated engineering environment is the amount, variety, and complexity of data, and the operations upon them, coupled with frequent demands for unplanned access. At the data-base level this requires a very general and flexible data structure, allowing rapid and dynamic re-structuring and unrestricted access paths. In the study reported here, this led to the choice of relational data support.

At the same time, the system must be interactive and highly responsive. Even if the prototype developed in the joint project leaves something to be desired in this respect, it was felt that performance problems could be overcome in the future by hardwiring the basic support functions. Therefore, a number of design decisions aimed at an utterly pure and simple *data model*, making a later implementation in relational hardware at least theoretically feasible.

The bulk of this paper describes the Model, whereas implementation points, including the prospect of a relational machine, are discussed in the concluding sections.

The support system was thoroughly validated in a number of application studies. Although a fascinating topic in its own right, the goal of the present paper is not a detailed report on our practical experience (see [11,12]). In summary, we found (1) application-development productivity improved by at least a decimal order of magnitude over conventional methods, and (2) a blurring of the borderline between application development and execution, both as user activities and as Model concepts. The latter result is especially important, bearing in mind that engineering applications often need to be gradually corrected and refined while being run. This property proved, in fact, to be shared with many non-engineering applications validated in the study.

The reader is assumed to be conversant with the fundamentals of set theory and the relational model of data.

2. The Binary Relation

The binary relation is one of the simplest structures (perhaps the simplest) that permits any other structure to be built from it. Therefore an early decision was made to use it as the basic structure in the Model.

It may be argued that the n–ary relation is a more general concept, but many facts spoke in favour of trading generality for simplicity in this instance. To begin with, we were pushed in this direction by the early influence of Abrial's work [1] on our study. A second reason was the obvious fact that the simpler the structure, the simpler the operations performed upon it, which leads to very generalised execution. This in turn increases the chances that one might eventually come around the performance problems that have plagued all previous relational efforts, provided the binary relations are not extensively used as a low-level base for n–ary relations.

In our experience they are not. A *relation* in the Model sense is closely connected with the intuitive notion of a *relationship* between two objects, as opposed to many other systems, where a relation is connected with the notion of a table[1]. Experience with describing (developing)

[1] One can probably argue forever which is more "natural" to humans. It seems likely, however, that an affinity for tabular arrangement, as found in the telephone directory or in a tax return, is not something one is born with, whereas relationships are fundamental to the way we think.

applications with the support system also bore out this observation. Many times the engineers already thought — sometimes unconsciously — in terms of such relationships.

Our users further pointed out that most true relations that occur in nature are binary, and one rarely encounters higher degrees than three (a ternary relation may of course be reduced to two or three binary ones). Indeed, the very fact that users of n-ary relations tend to give names to the columns supports this observation. Named columns are mostly equivalent to named binary relations (with the primary-key column). In the mind of the user, these named relationships are often more germane to his application than the artificial arrangement of the columns into an n-ary table (which may, in its turn, come from earlier association with fields in punched cards or the like). The fact that a set of binary relations may be neatly *presented* or *entered* in tabular form is a convenience, rather than a fundamental property, of the application.

Our decision also ruled out *unary* relations as components of the Model. This choice was reached after much debate, due to the greater uniformity, and therefore simplicity, achieved by having only two basic data types (objects and binary relations). When required, a set (unary relation) can be emulated, for instance by a binary relation with a constant in one of the columns. For operations like display, this constant can be chosen as an object that prints as a blank.

Another early decision was to allow non-normalised relations, in the sense that certain objects in the relation are themselves names of other relations. This, in turn, means that the distinction between an object and a relation becomes blurred; any object is potentially (the name of) a relation, and its role is determined by its occurrence in the data base, rather than by an artifact like a relational index. This aspect is further pursued in Section 11.

3. The Basic Object

An *object* `Q` in our terminology is the constituent of relations, a relation being a named set of object pairs:

$$R = \{<Q1,Q2> \mid Q1,Q2 \in V\}$$

where `V` is a set of objects.

An early issue was the possibility of an inner object structure. On one hand, one can represent such a structure as new relations between the constituents of the objects. This approach leads eventually to unstructured objects and may be regarded as "pure". On the other hand, one may accept an object structure that is unknown to the basic relational operators, but is interpreted by the accessing programs. This brings an inner object structure close to the concepts of "category" and "domain" [1], or "role" [2].

At this stage we consider only the simplest possible view of an object: a bit string of arbitrary length. The basic relational operators must be defined so they can handle such objects without regard to the possible meaning (structure!) vested by the user in the bit pattern. All the operators can decide is, in fact, whether two objects are identical or not, where identity is defined as string-length and bit-for-bit equality.

In implementation an object cannot be infinitely long, but in our experience the limit should be set very high. In some cases it even proved useful to store entire documents as objects!

The bit pattern of an object need not represent printable characters; it may be convenient, for example, to store a 4-character object, that when looked upon as a 32-bit quantity, is the floating-point representation of a number. The fact that this object, whose actual value is 5, say, may be regarded in certain applications as equivalent to the 1-character object '5' or to the 3-character

object '5.0' (or to the object 'V' in Roman notation) is irrelevant at this level. Since these three objects form different bit strings, they are different as far as the basic operators are concerned.

The approach obviates support for different object types (e.g. numbers and character strings). Any information about an object, for example that it is a decimal integer in the range 0-100, must be represented by additional relations. Such information makes it possible to convert, say, '5', '5.0', or 'V' to a common base in the processing program, when meaningful. In another application '5.0' may mean "street number five and ground floor" for instance, and conversion to a number has no meaning.

To summarise, the basic Model consists of:

1. **Binary relations** between **objects**, where an object is a bit string of arbitrary length, entirely devoid of attributes; and

2. A mechanism to retrieve a relation, given an object (the **name** of the relation).

4. Basic Operators

There have been several studies of relational operators, stressing aspects like compactness, completeness, general usefulness, and so forth [4,5]. In our study we have been concerned with a *minimal* yet *complete* set, this being in line with the objectives stated earlier.

The task of defining such a set of operators is greatly simplified by the fact that only binary relations are present. It is interesting to note that at an earlier stage of investigation, when also unary relations were included, the number of operators was 30% higher. Another criterion was that we wanted to limit ourselves to monadic and dyadic operators returning single results, as this simplifies the syntax of relational expressions. For example, this rules out Codd's restriction operator [5], since it is considered triadic.

4.1 Set Operators

The *set operators* are fundamental for any kind of queries to the data. They can be chosen in several ways that are equivalent from a functional viewpoint. If, for instance, union, intersection, and symmetric difference are chosen, the latter can be replaced by asymmetric (normal) difference or by monadic complement (provided a universal set is defined — see Section 7.5). We have chosen union, intersection, and (normal) difference as probably being the more useful operators.

The three operators all consider the binary pair as an indivisible unit (hence the name "set operator"), and pair equality is the joint equality of the two objects in the ordered pair. The operators can be described by first-order predicate expressions. Consider the set operation:

```
R3 = R1 • R2
```

where `R1`, `R2` and `R3` are relations; the operators "•" can then be defined by:

$$\forall\ T \mid (T \in R1 \wedge P1) \vee (T \in R2 \wedge P2),\ T \in R3$$

where `T` is an object pair, and the predicates `P1` and `P2` are given by:

Set Operator "•"	P1	P2
∪ Union	T ∉ R2	*true*
∩ Intersection	T ∈ R2	*false*
- Difference	T ∉ R2	*false*
+ Symmetric Difference	T ∉ R2	T ∉ R1

As seen, we have used two asymmetric predicates to reflect the sequential nature of practical evaluation. We can note that the results are always purged of duplicates whenever the operands are.

4.2 Pair Operators

The basic operators that regard pairs as consisting of two objects can also be chosen in several ways. We have first included (monadic) ***inversion***, which is a special case of projection, derived from the concept of the inverse of a mathematical function. It is defined thus:

```
R2 = ~R1
∀ <Q1,Q2>∈R1, <Q2,Q1>∈R2
```

The inverse is, of course, purged if the operand is purged. It exhibits the identity:

```
R1 ≡ ~~R1
```

Since we want the result to be a binary relation, we have chosen ***composition*** rather than join:

```
R3 = R1 * R2
```

A general form of composition is given by:

```
 ∀Q1, (∀Q2 | <Q1,Q2>∈R1, Q2∈S2),
(∀Q4, (∀Q3 | <Q3,Q4>∈R2, Q3∈S3)^P, <Q1,Q4>∈R3)
```

where the predicate `P` depends on the predicate variables `S2` and `S3`, which are sets of objects: `S2=S2(Q2)` and `S3=S3(Q3)`, functions of the inner objects of the joining pairs. We have the following fundamental cases:

Suggested Notation	S2,S3	P	Composition
R1 * R2		S2∩S3 ≠ ∅	object-equality (normal)
R1 ¬* R2		S2∩S3 = ∅	set-exclusion
R1 ⊆* R2		S2∩S3 = S2	set-inclusion (or S2⊆S3)
R1 =* R2		S2 = S3	set-equality
R1 ≠* R2		S2 ≠ S3	set-inequality
R1 ** R2		*true*	quadratic

Note that object-inequality composition is not covered by the above predicate expression.

These operators are not independent of each other, e.g. quadratic composition is the union of the inclusion and exclusion compositions, and set-inequality composition can be formed in a similar way. Furthermore, quadratic, as well as object-inequality composition can be formed with abstract relations (Section 7.5).

In summary, a minimal set of operators need only contain *three* of the above compositions to be complete, e.g. object-equality, set-equality, and set-inclusion. In the prototype we have only included object-equality (normal) composition among the basic operators. Our work indicates that an attempt to include general set-property composition by means of abstract relations alone, meets with theoretical difficulties. These have their root in the problem of defining "accessors" with sets as arguments (cf. [1] and Section 3). Our solution, described in Section 9, lies in exploiting the inner structure of *objects*.

4.3 Conversion Operators

The remaining operators deal with the conversion between an object and a relation. We define two operators of opposite effect:

Formation, denoted by the operator "`,`", of two objects yields as result a relation containing the single pair formed by the operands, or:

```
Q1,Q2 = {<Q1,Q2>}
```

It has been argued that monadic formation is more fundamental than dyadic. But this introduces a new concept, namely a null or blank object, and is therefore not more economical.

Extraction, denoted by the operator "#", has the basic effect of extracting an object from a relation and regarding it, in its turn, as (the name of) a relation:

```
for R = {<R1,Q2>},
    #R = R1
```

In the more general case, the object may be a relational expression, and the operand may have more than one pair:

$$\textit{for}\ \mathtt{R} = \{\langle \mathtt{R_i}, \mathtt{Q_i} \rangle,\ \mathtt{i}=1,2,\ldots n\},$$

$$\#\mathtt{R} = \bigcap_{\mathtt{i}=1}^{n} \mathtt{R_i}$$

The definition with intersection is justified by the use of extraction in domain definitions, but other operators can be similarly defined on e.g. union, and be generally useful.

We can note that the two conversion operators are connected by the identity:

```
Q1 ≡ #Q1,Q2
```

The basic use of extraction is to get a relation, whose name appears as an object in another relation. Suppose, in a list-processing application, a relation is a list of names of other lists (at left) with some index (at right). Then entry number 8 in list `L` is the list:

```
#L * <8,>
```

(where the angular brackets are used only for clarity).

The facility of writing a relational expression as an object makes it possible to store and exploit *defined* relations. Suppose, e.g. that the relation `grandparent` contains the sole pair:

parent * parent	

where `parent` is a relation containing the parents of a number of persons, then the relation:

```
#grandparent
```

would contain the grandparents of the same persons. This relationship is maintained, irrespective of changes in the relation `parent`.

Finally, it is important to note that the extraction operator complicates the evaluation of relational expressions to the extent that the identity of all ingredient relations is not predictable.

5. Expressions

We conclude the discussion on operators by listing the syntax of basic relational expressions. This syntax is later extended by the so-called second-order operators (Section 10).

A relational *expression* is:

	Remarks
relexpr :=	
object	relation named by the object
object, *object*	formation
~*relexpr*	inversion
#*relexpr*	extraction
relexpr op relexpr	

where the basic dyadic relational operators are:

	Remarks
op :=	
∪	union
∩	intersection
-	difference
*	composition

and *object* is any character string, whose exact specification is unimportant in the present context.

The algebra contains only *seven* operators, and the result of an expression is always a relation (it can never be an object). The syntax does not show the traditional use of parentheses to signal evaluation precedence, nor the angular brackets often used redundantly to highlight the formation operation. The only operator precedence rules, chosen for syntactic convenience, are that "," and "~" have the two highest binding strengths. The question of direction of evaluation is discussed in the next section.

Although limited in the area of set-property composition, the seven operators form an almost complete and very compact set that allows the majority of queries to be formulated in practical

cases, as shown by our study. In this, the operators are considerably enhanced by the support of abstract relations, discussed in Section 7.

The syntax is semantically closed, in the sense that all syntactically correct expressions have a meaning. No error situations can occur in this context. A contributing factor is that any object, stated in an expression or resulting from extraction, that does not name a stored relation, is nevertheless considered the name of an empty relation.

6. Statements

The next step towards a relational language is to introduce *statements*. On the basic level, only a few statements are of significance. They are ***assignment*** and ***list***, where one might consider *list* as a special case of "assignment to the paper", as does e.g. APL. There has been some debate as to whether assignment is to be regarded as a dyadic operation whose result is a relational expression that can continue at left. It was finally decided, however, that assignment is not an operator, but a statement. This leaves us with two basic statement forms:

> *object* ← *relexpr*;
> `LIST` *relexpr*;

where `LIST` is used for output on whatever medium is used.

A number of other statement forms have been introduced for permanent or temporary assignment, and for defining macros. These refinements, although essential in practice, are not needed in the present context.

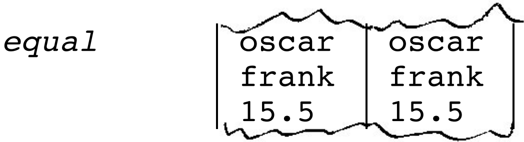
Assignment has the effect of *replacing* a relation. Other basic operations, such as adding or deleting a pair and emptying a relation, can be expressed by assignment. They may be recognised as special cases and their execution optimised by the system.

Note that we do not talk here about the creation or deletion of relations, which has no meaning at this level, and no statements to this effect have been introduced. An object is the name of a non-empty relation if and only if it has been assigned to a non-empty result.

It is certain that the syntax of relational expressions and statements is somewhat "hieroglyphic" and not suited to untrained users. In the study we developed several macro facilities (not reported here) that allowed the engineers to create and manipulate relations comfortably. Most users were able to learn the concepts and methodology in a couple of days.

7. Abstract Relations

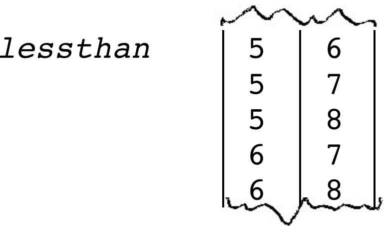
The case for *abstract relations* arose early in the project, they being central to Abrial's theory [1]. Many relationships between data can be described not by an explicit tabular list of pairs (a *concrete* relation), but rather by an *algorithm*. Let us take the simple example of selecting the part of a relation `R`, where the two objects in each pair are equal. Rather than inventing a new operator for doing this, we imagine an infinite relation `equal`, consisting of all objects in our universe at left, paired with the same objects at right. A section of that relation might look:



The desired query can then be expressed as a simple intersection with *equal*:

R ∩ *equal*

This syntactic use of abstract relations conforms with the intuitive, as well as the mathematical, notion of equality as a relationship between two objects. To further emphasise this point, let us select from R all pairs where the objects at left are smaller than those at right over the domain of integers. For this purpose we introduce a new abstract relation, exhibiting the desired relationship:



The query is then expressed as:

R ∩ *lessthan*

and the query selecting the less-than-or-equal pairs as:

R ∩ (*lessthan* ∪ *equal*)

Note, that if R contains objects other than integers, the expression is not in error: the corresponding pairs are simply not included in the result. One may, of course, broaden the domain of *lessthan* to real numbers, character strings, and so on.

In practical cases the universe of stored data is, of course, finite. But if the concept of abstract relations permits us to introduce sets of objects like "integers", the knowledge of the data base, i.e. the effective Universe, becomes truly infinite through its semantic properties. Therefore the members of the study coined the term *knowledge bank* (banque de connaissance) rather than talk about a "data base".

7.1 The Tester

It is apparent that infinite abstract relations can be operated upon by the basic relational operators, but that they must always be combined with concrete relations to yield a finite result. The difference occurs at evaluation: an abstract relation cannot be accessed in the relational memory, but must be invoked as a *program*. In the intersection case above, each pair of the concrete relation R will be given in turn to the program, and the program will test whether the objects in the pair satisfy the stated relationship, i.e. equality or inferiority. The pair will be added to the result only if the test is satisfied. Because of this mechanism, the algorithm or program that represents the abstract relation is called the *tester* of the relation[2].

We shall say that ***an abstract relation*** A ***has a tester*** T(A) ***if*** T ***is a finite procedure that for any pair*** <Q1,Q2> ***can decide whether*** <Q1,Q2> ∈ A.

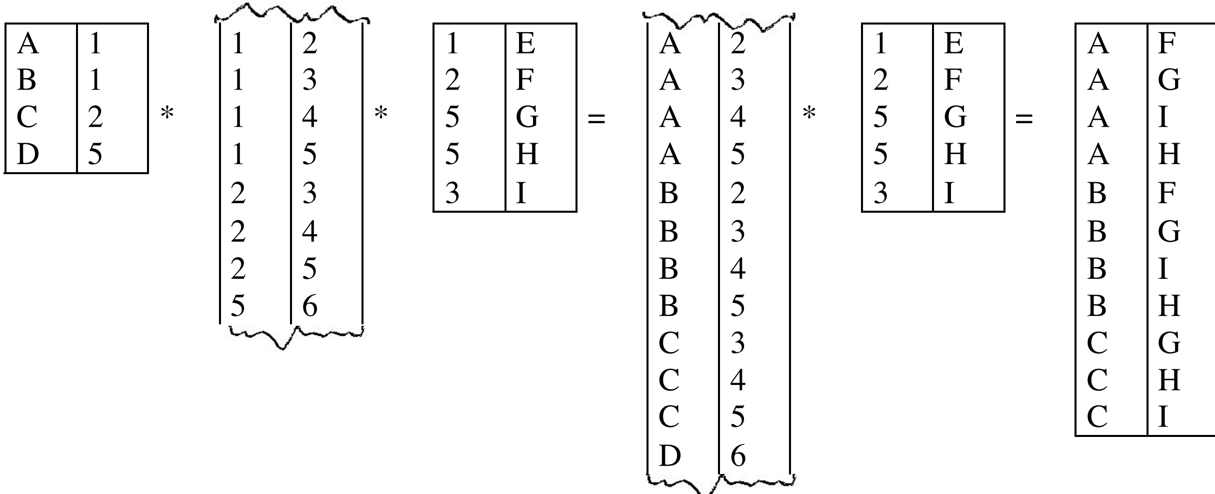
[2]This discussion is in terms of sequential execution. However, the definition of the tester (and of the other relational programs) holds equally well if a parallel-executing machine accepts all pairs simultaneously.

It is a useful exercise to ignore the practical implementation, and imagine an actual union and intersection to take place on the infinite relations, using sections of them as illustrated above (we shall use a jagged box in these examples).

Also the inversion operator is easy to handle with testers. Given `T(A)`, the same tester program `T` can be used for `~A`, provided each pair is inverted before it is presented to `T`.

7.2 Double Composition

In a previous section it was mentioned that it is possible to generalise the composition operator through abstract relations. As an example of this, let us take our previous relation *`lessthan`* to perform a composition under the predicate of object *inferiority*, without changing the definition of "*":



A	1
B	1
C	2
D	5

*

1	2
1	3
1	4
1	5
2	3
2	4
2	5
5	6

*

1	E
2	F
5	G
5	H
3	I

=

A	2
A	3
A	4
A	5
B	2
B	3
B	4
B	5
C	3
C	4
C	5
D	6

*

1	E
2	F
5	G
5	H
3	I

=

A	F
A	G
A	I
A	H
B	F
B	G
B	I
B	H
C	G
C	H
C	I

The reader can verify that the operation, which is written:

```
R1 * lessthan * R2
```

yields the same result when executed from right to left, thus preserving the associative law for composition also for abstract relations. We have, in other words, expanded the repertoire of operations into a new form of composition through an abstract relation, rather than adding a new operator.

The new operations are termed *double composition*, and the middle relation is called its *filter*. As can be easily verified, it is the *tester* of the filter that is invoked in double composition; however, this time the tester program is not given two objects from one pair, but one object from an `R1` pair and one object from an `R2` pair. In this way, the tester program works without change for both double composition and intersection.

Single (normal) composition can be regarded as a double composition with the filter *`equal`*.

7.3 The Accessor

One can see from the above illustration that the intermediate result:

```
R1 * lessthan
```

is not finite. There is, however, a class of abstract relations that *can* be evaluated in single composition. They are typified by (single or multi-valued) *functions* in the mathematical sense.

Imagine an abstract relation `sqrt`, which for any object `Q` over the domain of positive integers yields √`Q` and −√`Q`:

sqrt

7	2.646
7	-2.646
8	2.828
8	-2.828
9	3
9	-3

Composed at right with a relation `R`, it will replace the right-hand objects of `R` with their square roots (adding a new pair for the negative root):

```
R * sqrt
```

and composed at left, it will compute the squares of the left-hand objects.

The program that represents the abstract relation `sqrt` is different from a tester in nature. In one form it will be given a number from `R` and asked to return the two square roots; such a program is termed the *accessor* of the abstract relation. `sqrt` is also represented by a second program which yields the square of the object, when the relation is composed at left or inverted right. This second program is the *inverse accessor*, so termed because it represents mathematically the inverse function. In fact `sqrt` might also be represented by a third program, a tester, which verifies that the right object is the square root of the left.

We shall say that ***an abstract relation*** `A` ***has an accessor*** `B(A)`, ***if*** `B` ***is a finite procedure that produces, for a given*** `Q1`, ***all*** `Q2` ***such that*** `<Q1,Q2>` ∈ `A`. Similarly it may have an ***inverse accessor*** `I(A)` that produces `Q1` if `Q2` is given. As will be seen from several examples below, a relation with an accessor need not have an inverse accessor, and vice versa.

A relation represented by an accessor need not, of course, be a mathematical function. An example is the relation `last` between an object of any kind, and its last character. If `R` contains names of persons over a domain where female names end with "a" (as is, e.g., often the case in Spanish), the women can be selected by:

```
R * last * <a,>
```

Our abstract relation *`equal`* may also be represented by an accessor. Because this program returns the same object as is given to it, any concrete relation can be composed on either side with *`equal`* without change:

```
R * equal ≡ equal * R ≡ R
```

Therefore *`equal`* is the *identity relation* for composition, and its accessor and inverse accessor are one and the same program. We then have:

```
~equal ≡ equal
```

and *not* the relation *`unequal`*. In fact *`unequal`*, although infinite like *`equal`*, is nevertheless more difficult to "tame", and is in this respect similar to *`lessthan`*, none of which can be singly composed with a relation to yield a concrete result. This is the same as stating that neither *`lessthan`* nor *`unequal`* can have an accessor. It is apparent that the query "Which objects are

not equal to `Oscar`?" can not, in contrast to "Which objects are equal to `Oscar`?", give a finite answer. These properties are formalised in Section 9.

On the other hand, the following identity holds:

~lessthan ≡ greaterthan

obviating a special tester program for superiority.

7.4 The Generator

The reason for abstract-ness was the infinite cardinality of the relations in our previous examples. This need not always be the case, however. Every time there is some *systematic* way to define the objects, it is practical to express also a finite relation with a program rather than as a stored table. The program that is able to generate such a relation is called the *generator* of the abstract relation. An example is the relation `digits`, whose generator produces pairs of all integers from 0 to 9.

We shall say that ***an abstract relation*** `A` ***has a generator*** `G(A)`, ***if*** `G` ***is a finite procedure that can produce all pairs*** `<Q1,Q2>`, ***such that*** `<Q1,Q2>` ∈ `A`.

Generators can be defined to solve the task of *data entry*. It is often convenient to enter data from a keyboard (or other medium) in tabular form, and a statement:

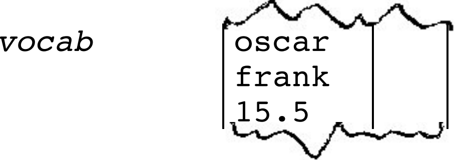
`X ← term;`

may mean "enter a number of pairs from the keyboard and store them in `X`". `term` is in this case an abstract relation, whose generator accepts one line after another from the workstation and stores them as pairs in a temporary relation. The generator program will handle the parsing of lines into pairs and objects and convert the latter into bit strings.

Unfortunately the concept of an abstract relation does not lend itself easily to the inverse operation: output of a relation. Our work indicates that no general meaning can be attached to an assignment whose target is an abstract relation (except, of course, replacing it by a concrete one).

7.5 Universal and Fundamental Relations

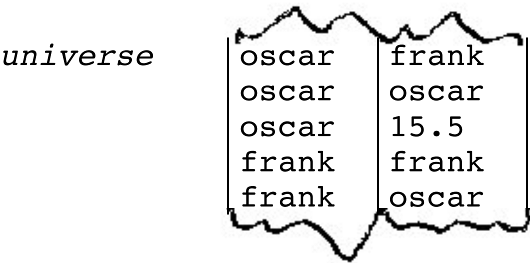
We now have material for stating the universes of our Model in terms of abstract relations. We define the relation *vocab* as containing all objects at left, and nulls at right:



vocab is therefore the *vocabulary* of all objects known to the knowledge bank, including all infinite sets, like numbers, used by other abstract relations. *vocab* is an abstract relation and has an accessor that always returns a null. It as no inverse accessor.

universe is defined to contain all pairs that can conceivably be formed:

universe



oscar	frank
oscar	oscar
oscar	15.5
frank	frank
frank	oscar

universe is an abstract relation with a tester that always returns *true*. Obviously these two *universal relations* are connected by the identities:

```
universe ≡ vocab * ~vocab
vocab ≡ universe * <Q,>
```

where Q is any object.

universe is the identity relation for intersection:

```
R ≡ R ∩ universe
```

and it can also be used to define a monadic *complement* operator:

```
-R = universe - R
```

Abstract complements can be expressed in this way; e.g. the relation *unequal* is not explicitly needed, since:

```
unequal ≡ universe - equal
```

The abstract relation *empty* may be represented by a generator that generates nothing (or a tester that always returns *false*). It is the identity relation for union:

```
R ≡ R ∪ empty
```

(Of course, *empty* may also be defined as concrete).

We may now express the *projection* operators in terms of abstract relations. The nearest we can come to one-column projection (since we bar unary relations) is to pack the remaining column with nulls. This is directly achieved by composition with *vocab*.

Columns may be duplicated with:

```
<1,1>%R = (R * ~R) ∩ equal
<2,2>%R = (~R * R) ∩ equal
```

where "%" stands for the (dyadic) projection operator. In the study we found many situations where it was practical to use these projections to represent *unary* relations, rather than packing with null (cf. Section 11). They can also be used to construct operators similar to *restriction*. For example, the query "Replace all right-hand objects by Oscar" is expressed by:

```
R ← R * vocab * <,Oscar>;
```

This is not a case of double composition, and it can easily be verified that associativity of composition is conserved also in this case, although actual evaluation must start with R.

Finally, a noteworthy result is that object-inequality and quadratic composition can now be expressed by double composition:

```
R1 ~* R2 = R1 * unequal  * R2
R1 ** R2 = R1 * universe * R2
```

respectively.

We may note an interesting property of the four abstract relations `universe`, `empty`, `equal` and `unequal`. They are similar to concrete relations in the respect that they operate without regard to the *meaning* of the objects. Any two objects, irrespective of their domains, can be compared for equality in `equal` and `unequal`. On the other hand, the comparison in the tester of `lessthan` restricts the objects to the domain of real numbers (unless one somehow broadens the meaning of "less than" for other domains). The testers of `universe` and `empty` return `true` and `false`, respectively, without even looking at the objects to be tested.

These four relations are therefore *fundamental* in the context of abstract relations, and they are moreover the *only* fundamental ones. All other testers and accessors must assign some meaning to the objects, however trivial, as e.g. in the above relation `last`. In fact, the reason for an abstract relation is quite often to introduce the properties of a new domain.

From an economical point of view, it is certain that the saving of projection and restriction operators is somewhat offset by the necessity of certain abstract relations. But these abstract relations have proven useful in a host of other situations, and we have found that the compactness of a very small set of operators together with a general mechanism for handling abstract relations is the correct trade-off. In fact only *two* abstract relations are necessary in order to produce all fundamental and universal relations, namely `vocab` and `equal`, which is certainly not too high a price to pay for the considerably increased power of the Model.

7.6 Summary

Abstract relations permit us to extend the notion of relationships between objects to embrace intuitive and algorithmic relations of infinite (or finite) size. Though we have concentrated on those representing mathematical and logical relationships, many of the abstract relations we used in practical applications perform problem-oriented tasks. Examples are `inertia` (between a geometric shape and a number) and `totalcost` (between a project and a dollar figure).

Our prototype supports the use of concrete and abstract relations in the same way: their behaviour in expressions is transparent to the user. We can say abstract relations represent procedural knowledge in the knowledge bank, whereas concrete relations represent factual knowledge.

8. Relational Programs

The programs that represent abstract relations must be finite procedures. We see immediately that starting with a generator, we can always construct an accessor, an inverse accessor and a tester for the same relation, simply by making the relation explicit. Furthermore, it is always possible to construct a tester from an accessor (straight or inverse). On the other hand, due to finiteness requirements, all relations do not have an accessor because they have a tester (e.g. `lessthan`). Nor do they have an inverse accessor because they have an accessor. A few examples will illustrate this:

- Let `factor` be the relation between an integer and all its integral factors; it has an accessor, but no inverse accessor, since there are infinitely many multiples of any integer.
- `arcsine` over the domain of real numbers has an inverse accessor (sine) but no accessor, unless it is restricted to the principal value (which would similarly restrict its inverse accessor).

- Let `trunc` be the relation between a real number and the nearest smaller integer; it has only an accessor.

The absence of an inverse accessor does not, of course, preclude the existence of an inverse *relation*.

We can now grade all relations, depending upon the existence of the following *relational programs*:

1. generator
2. accessor *and* inverse accessor
3. accessor *or* inverse accessor
4. tester

where level 3 can be split into two types:

3A. accessor exists
3I. inverse accessor exists

Level number 1 through 4 will be termed the *evaluability* of the relation. We see that a program at any level covers the programs at all higher levels, and it is therefore unnecessary to provide, say, both an accessor and a tester to represent a relation of evaluability 3. It would indeed be an error to do so, since even a small inconsistency between the two programs would ruin the integrity of the knowledge bank.

To further unify the Model we regard a concrete relation as a special case of an abstract relation of evaluability 1. Its generator turns to the physical data base for the generation of its data, and the corresponding relational program is therefore the access method, into which all idiosyncrasies of the current implementation can be built.

Our prototype provides a simple PL/I-like syntax for coding the relational programs. The program header is ACCESSOR, TESTER, etc. rather than PROCEDURE, to signify the evaluability. A generator is invoked once to return all pairs of the relation. An accessor is invoked once for each object, and is supposed to return zero or more "accessed" objects. A tester is invoked once per pair and will return either ***`true`*** or ***`false`***. No particular order is assumed between these invocations.

The methodology used to develop applications in our study encourages *all* procedural code to be specified as relational programs. Normally, therefore, there was little programming in conventional terms. Programming takes place in the syntax mentioned, and the substance of the application will be a mixture of concrete and abstract relations.

9. Evaluability

It is possible to establish rules for how to calculate the evaluability of any expression not involving the extraction operator, or where all extracted relations are known to be concrete. Restricting ourselves to such expressions, it is possible to decide before processing whether they are evaluable or not. If not, the expression is in error, and does not yield a well-defined result. (This is in fact the only semantic error that can occur within the framework of the Model.)

By *evaluable* is meant that the evaluability of the result relation is *one*. The present prototype uses an evaluation algorithm that ensures that any evaluable expression is correctly executed. This process uses known properties of the basic operators, such as their distributivity and associativity laws, in order to re-arrange the expression for easier handling. It also applies some

limited optimisation of the expressions, and performs some special-casing. Although an interesting and challenging topic in its own right, a description of this process would be somewhat lengthy and of less interest in the present context. More sophisticated work in the area has been reported elsewhere (e.g. [8]).

On the other hand it is interesting to investigate whether the five evaluabilities, defined in the previous section, represent all possible forms of abstract relations. As will be shown in the following subsections, this is *not* so. Composition of some evaluabilities leads to new abstract relations that can not be handled by algorithms of the required kind. Expressions leading to these cases are therefore in error in the present prototype, and further research is needed to resolve this highly challenging point in the theory.

9.1 Set Operators

It is easy to see from the predicates defining them, that only a tester is required by the set operators. Space does not permit the complete analysis, but it is not difficult to prove that if R(*n*) means a relation of evaluability *n*, we have:

$$R(n) \cup R(m) \rightarrow R(\max(n,m))$$
$$R(n) \cap R(m) \rightarrow R(\min(n,m))$$
$$R(n) - R(m) \rightarrow R(n)$$
$$\sim R(n) \rightarrow R(n)$$

where "→" means "yields". It can be seen that the above operations involving evaluability three preserves the type 3A or 3I, except that inversion interchanges 3A and 3I.

In conclusion, the set operators combine to yield the evaluability of one of the operands, and no new level emerges in the process.

Note that we *may* get a finite result in special cases. For instance, if the relations `lt9` and `lt6` consist of all pairs <Q,Q>, where Q is an integer less than nine and six, respectively, we have:

`lt9 - lt6 =`

6	6
7	7
8	8

although the result has evaluability 4. Such cases can not be treated in a general way, and we therefore do not consider them in the theory.

9.2 Composition

The situation is far more complicated for composition. Since the accessor (or its inverse) is involved in single composition, the problem can be explored in terms of what comes out of the accessor when various sets of objects are presented as input. In an expression:

```
R1 * R2 * R3 * R4 * ...
```

a middle relation may be presented either from the left or from the right with sets of objects from the neighbouring operand, and produces as the result of the composition operation a new set of objects, which is passed further along the line. In this way one can associate a relation of any evaluability with a virtual accessor F and its inverse F^{-1}:

$$Y_1 = F(X_1)$$
$$Y_2 = F^{-1}(X_2)$$

yielding an output set Y_i from an input set X_i. The different cases of evaluability now depend on whether the sets X_i and Y_i are finite or infinite, and our task becomes one of combinatorics. Denoting by $C(X)$ the cardinality of a set X, finity by "$\circ$" and infinity by "∞", we can in principle construct 4 × 4 = 16 types of virtual accessors. These correspond to:

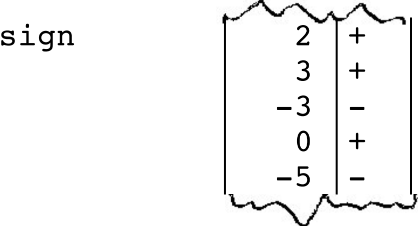
$$C(Y_1) = \circ \text{ or } \infty, \text{ for } C(X_1) = \circ \text{ or } \infty$$
$$C(Y_2) = \circ \text{ or } \infty, \text{ for } C(X_2) = \circ \text{ or } \infty$$

independently.

In a first cut, we shall exclude the obviously unrealistic cases where the accessor yields a finite output for infinite input, yet infinite output for finite input. This brings us down to nine types of accessors, characterised by $C(Y_i)$. It is now possible to derive the types of all 81 possible combinations under composition. This is a lengthy process, but it can be shown that seven of the nine types form a semi-group under composition, where evaluability 2 is the unity element. Further analysis shows that the seven types of the semi-group obey the associative law under composition (the commutative law does not hold, of course, since it does not hold for the relations themselves).

The two remaining evaluabilities can therefore never be formed by combining any of the members of the semi-group, and will never be relevant in the Model, unless abstract relations exhibiting these types can be found in nature. So far we have been unable to find an example.

Excluding these evaluabilities in the second cut, we find that five of the seven types are our previously defined evaluabilities. What are the two remaining types? They can be formed by the composition of other types, and examples are easily found in nature. A typical one is the relation `sign`, over the domain of integers:



Examination of the kind of accessor that would have to be programmed to represent such a relation reveals that no finite algorithm can handle its evaluability. Therefore we decided to treat the remaining two odd types as evaluability 3 (they are in fact slightly easier to "tame"). From this follows that some expressions that our prototype classifies as non-evaluable, are in reality evaluable. The `sign` accessor, for example, even when presented with an infinite set of integers, should never be able to produce more than two objects (+ and –).

After the last cut the following table gives the evaluability of composition:

p in R(n)*R(m) → R(p)

n	m= 1	2	3A	3I	4
1	1	1	1	(3I)	(3I)
2	1	2	3A	3I	4
3A	(3A)	(3A)	3A	4	4
3I	1	3I	4	3I	4
4	(3A)	4	4	4	4

The associative law for composition does *not* hold any longer for the reduced group of types. This has the consequence that certain operations yield the odd evaluabilities. Those, parenthesised above, are considered an error.

As can be seen, an evaluability is invariant when combined with itself, and has in general no inverse. Also the result is evaluable only if at least one of the evaluabilities is 1. It is interesting to note, however, that before the final cut the two odd types in the semi-group combined to yield evaluability one, which is the only case where two infinite relations generally combine with any kind of operator into a concrete relation.

It is evidently possible to refine even more the analysis of accessor types and evaluability of composition. Introducing the notion of domains of objects, one may distinguish between enumerable and non-enumerable infinite sets, and find that a wider class of expressions are in fact evaluable. We have left these Cantorian excursions for future research.

10. Grouping Operations

Our defined relational operators make it possible to access certain type of information from the knowledge bank. The information is limited in several ways, e.g. a result is always a relation, and the relational operators do not in principle create *new* objects. Even though generators and accessors can be said to create objects, their base relations always build on defined domains, whose existence is a priori assumed. So far, new objects are always introduced through the formation operator.

An important class of results is thereby excluded by the relational language, namely where one wishes to *compute* something from the pairs. The classical case in point is to try to derive the average of a population using some relational algebra. The issues and difficulties are well known: How does one keep duplicate values of the population in a purged relation? The averaging function can be realised as an abstract relation, but between what and what? And how are we to represent the results of the calculation? If they are to be the population size, the average and the standard deviation, it is not evident what is going to be paired with what.

The fundamental problem here is that operations that involve columns rather than pairs are not easily adapted to the notion of relationships between objects, and are therefore not directly supported by the basic relational operators. For example, by combining the mechanisms of a tester and a generator, one can essentially produce *pair functions*. But such a "relational" program would hardly represent a meaningful relationship between the left and right objects, but rather some kind of relationship between two *relations*: the old pairs and the new, transformed ones.

The grouping facilities described in the next few sections are not necessarily the final Model answer to relations of relations, operations on sets of objects, and other "second-order" issues. To further streamline the theory in this respect is one of the many interesting tasks for further research.

10.1 Relation Names as Objects

In order to support generators and accessors that receive or produce relation names, we have introduced a special statement, *SCAN*, to be used in relational programs. It causes a given relation to be scanned in a loop, pair by pair, until exhausted. In contrast to the accessor mechanism, initiation and termination code can be inserted before and after the loop. This makes it possible to implement `average` and similar computational procedures as the (abstract) relation between a relation name and a number. This also fits with the intuitive notion of a population as a whole (relation) having certain properties (relationships), such as average, in contrast to the properties of its constituent objects. If `R1` is the average of `R2` in this sense, we have:

```
R1 ← <,R2> * average;
```

where `average` has an accessor with a *SCAN* statement.

Another example is the abstract relation `close`, representing transitive closure under composition. It relates the name of one relation, `R1`, exhibiting some hierarchical relationship, with the name of another, `R2`, which contains all instances of nodes in the unravelled hierarchy of `R1`. The name of `R2` is in this case invented by the accessor of `close` as the composition takes place. Thus, if `R1` represents the relationship "consists of" between components and sub-components in a machine assembly, then:

```
R2 ← #~(<R1,R1> * close);
```

is the "exploded view" of `R1`, relevant e.g. in a bill-of-material application.

Although powerful enough to cope with computational and similar tasks, *SCAN* itself has no clear theoretical backing at present, and must be regarded as a necessary programming convenience (it may, moreover, be used to implement the relational operators themselves).

10.2 The Grouping Operator

SCAN allows any desired processing to be carried out on relations. In many cases, this processing involves structures of objects that are not necessarily meaningful in terms of relations belonging to the application. In support of these cases, we have implemented another extension, which may have more merit than *SCAN* from a theoretical viewpoint.

It builds upon the fundamental premise, stated in Section 3, that any object may have an *inner structure*. This structure, e.g. a floating-point number (characteristic + mantissa), is not known to the basic operators — it is at most interpreted inside a relational program (e.g. in an accessor), or by the user when entering or viewing the object. One can say that the inner structure is an escape mechanism, allowing n-ary relations, ordered collections of data items (e.g. vectors and matrices), and hierarchies to be represented in any user-defined way.

In the present prototype the inner structure is specified by some user-defined syntax on the object name. The structure may be a permanent feature of particular domains of objects (e.g. matrices), or it may be transitory if the user chooses to make it explicit through new relations at a later stage of development (outer structure).

Among the "users" that may impose an inner structure on objects is the relational system itself. In this vein, we have introduced an inner-object syntax (hidden from the real user), and operators and abstract relations that support this syntax. The result is that a host of previously identified issues have found solutions, although perhaps not in the theoretically most elegant manner. Among those are computations (e.g. `average`), set-property composition and other set operations.

The monadic operator "/", ***grouping***, is defined thus:

$$R2 = /R1$$

$$\forall\ Q_i,\ (\forall\ Q_j \mid <Q_i,Q_j> \in R1,\ Q_j \in S_i),\ <Q_i,f(S_i)> \in R2$$

where S_i is a set, and the mapping function $f(S_i)$ forms an object from S_i using a system-defined inner structure. Thus, we may, for example, group suppliers by their products:

supplier

Alain	butter
Alain	sugar
Gérard	sugar
Monique	cream
Monique	sugar
Monique	butter

/supplier

Alain	$f(S_1)$
Gérard	$f(S_2)$
Monique	$f(S_3)$

where:

S_1 = {butter, sugar}
S_2 = {sugar}
S_3 = {cream, butter, sugar}

10.3 Computational Relations

The grouping of a relation does not, by itself, perform a useful task, except possibly if the result is listed in a format that brings out the system-defined inner structure. The composite objects $f(S_i)$ are handled internally as any other objects by the basic operators, so it is possible to, say, intersect two grouped relations, the result still being grouped.

More importantly, a host of abstract relations may now be defined to perform useful functions, in particular computations, upon the composite objects. With a program element to unravel the inner structure, one can code the accessor of a `sum` relation, such that:

meas

A	2
A	3
A	6
B	4
B	6

(/meas) * sum

A	11
B	10

Similarly, `average` would produce averages of the right-hand objects of `meas`, grouped by the measurer (left-hand objects). Note, however, that these facilities do not alone solve the problem of how to represent duplicate objects (e.g. measurements).

The basic abstract relation, *expand*, is defined as the inverse of the grouping operator:

$$R \equiv (/R) * expand$$

Its accessor returns all objects of the group. Some reflection will show that the grouping operator itself can not be replaced by an abstract relation.

10.4 Set-Property Composition

In the previous section, accessors were involved. In the same way, abstract relations may be defined whose testers examine the composite objects in double composition. Among those, filters that test for set properties are of particular interest. For instance, an application may define a filter relation `com2`, which composes under the condition that there are exactly two common objects in the sets:

```
(/R1) * com2 * (~/R2)
```

All set-property compositions of Section 4.2 can now be handled. For example, the query "which suppliers satisfy my need to manufacture which products?":

```
which ← (/product) * incl * (~/supplier)
```

where the filter `incl` handles set inclusion. With the `supplier` example of Section 10.2, a bakery with the following products may receive the answer:

product

cake	cream
cake	butter
pizza	butter
pizza	sugar

which

cake	Monique
pizza	Alain
pizza	Monique

Set-equality composition is had simply by:

(/R1) * *equal* * (~/R2)

which is, of course, the same as:

```
(/R1) * (~/R2)
```

10.5 Group Composition

The abstract relations performing computations like `sum` and `average` represent relationships between a set of objects and a property of such a set. But one often wishes to transform the objects in the set prior to any computation; for example, square all measurements before summing, as a step in a least-squares fit. One way out would be to use a special abstract relation, whose accessor works on the composite object directly. But this would mean a duplication of function if (as is usually the case) a "traditional" abstract relation `square`, which works on the elemental objects, is already available. This is all the more important when one wishes to compose the objects with existing concrete relations before applying the sum.

In order to exploit existing relations, we have introduced a new operator, ***group composition***, denoted "$\underline{*}$".

The example mentioned could now be coded:

((/meas) $\underline{*}$ square) * sum

This operation does not eliminate duplicates when forming the group. Therefore, each grouped object in `meas` has a distinct corresponding element in the composite object after group composition (e.g. 3 and -3 would give rise to two instances of 9 in the above expression).

This property of the composite objects solves the problem of duplicate measurements and similar cases of populations stored in relations. Now `meas` would be the relation between the measurer and a uniquely assigned measurement number (such as a time stamp), and another relation, say `mno`, would relate the number to the actual measurement, of which there may be duplicates. Sums of the measurements, grouped by measurer, would then be given by:

```
((/meas) *̲ mno) * sum
```

whereas:

```
((/meas) * mno) * sum
```

would yield purged (erroneous) sums.

11. Classes of Objects

In the previous discussion, one can discern a bias in favour of the binary *relation*, at the expense of the *object* as item of importance. This was first suggested by the equivalence between an object and a relation; any object being a potential relation name. If no pairs are associated with the object, it is still the name of an, albeit empty, relation. The point was emphasised by the extraction operator, making in essence a relation available from the contents of another relation. We shall now spell out the fundamental role of an object in our Model; a principle that has been implicit in most of what was earlier said:

> *An object has a meaning only in its relationships to other objects.*

This principle has its origin in the human user's intuitive perception of an object in the engineering — and probably other — problem environments. It states, in other words, that an object by itself is meaningless: each object must be seen together (relations!) with other objects to gain relevance.

An object is never rejected by the knowledge bank, but unless used in a relation, it will eventually be forgotten (purged from the vocabulary). (This remark may be true for the human memory as well.) We can express this by saying that the system "knows *about* all objects", but does not "know all objects". This is especially striking in the presence of abstract relations: here the system implicitly knows about all objects in the domains (e.g. all integers) over which the abstract relations are defined.

In some other models (e.g. [1]) the system must support concepts like "exists" or "constitutes" or similar facts that imply that the object is stored in the knowledge bank, and perhaps belongs to a "category" (domain, class, role ...). In our Model, the user may define such "pseudo-properties" if he wishes; they would then be represented by relations named *`is, constitutes, property, category`*, or the like, but this is not *required* by the Model.

For example, the statement "Oscar" has no meaning in our Model; "Oscar is" or "Oscar exists" is trivially true if `Oscar` is used at all in the knowledge bank, otherwise false. (Knowledge about an object can, of course, exist outside the system, e.g. in the mind of the engineer.) On the other hand, the statement "Oscar is a person" involves perhaps the relation `species` or *`category`*, which may have some merit in the application. But eventually such definitions become circular or trivial, in that one ends up by trying to define *`category`* or even *`relation`*, and the information is no longer pertinent to the semantics of the application. In summary, the Model supports *application information*, not (meta-)information about its own internal functionality.

In the following sections we shall close the object-relation circle by showing how an object can be approximately defined in terms of relations. This is done via the concept of a *class*.

11.1 Semantic Classes

The first derivation of objects from a binary relation arises when we collect all objects that appear in the left-hand (or alternatively right-hand) column of a relation R, a process traditionally called *projection*. Such a set of objects will be called a *semantic class* of R (other terms e.g. domain, range or role, have been used elsewhere). Each relation has two semantic classes, defined with the symbols of Section 7.5 as:

```
C1 = <1,1>%R
C2 = <2,2>%R
```

In special cases, C1 and C2 for a given relation R, can be regarded as the value domains from which objects can be drawn to form new pairs in R. But in general C1 and C2 will change as pairs are added or subtracted in R, and only occasionally will a semantic class contain all possible values of objects for that column, i.e. form a true domain. Take, for example, the relation ccomp, which contains the colours of the components used in manufacturing a product:

ccomp

stud	red
bolt	ochre
cover	lightpink

Now it is highly unlikely that the right projection will span the full domain of colours available for painting the product. Only when looking at the semantic class at the peak of its cardinality, i.e. when all necessary components have been added, does one stand a fair chance of catching the full spectrum. It might here be more useful to look elsewhere, say, in a relation formula which defines a mixing formula for paints, or in a relation stpaint that describes the storage place for all paints. Which is the best one to use? Recognising that a paint without both a mixing formula and a storage place will be of little use in the manufacture, the semantic-class intersection:

```
<1,1>%formula ∩ <1,1>%stpaint
```

is probably a more accurate representation of a domain. This form allows, say, a new colour to be defined by a formula on a trial basis, without actually giving it a storage place. Sticking strictly to the intersected domain one can ensure that this new paint will not be applied to a component until the stpaint relation has been properly updated.

The example indicates that, in general, the notion of a domain builds upon the joint usage of the objects in a *number* of relations. Just as we said that the meaning of an object comes from its use in relations, a set of objects has a meaning only as part of semantic classes.

11.2 Domains

This leads to the formal definition of a domain:

Let a knowledge bank B consist of *n* relations R_i:

$$B = \{R_i \mid i=1,2...n\}$$

We can index all semantic classes in B with the set M of ordered pairs:

$$M = \{\langle R_i, j\rangle \mid i=1,2...n;\ j=1,2\}$$

A *domain* D(N) with respect to a subset N of M is defined as:

$$D(N) = \bigcap_{<R_i,j> \in N} <j,j>\%R_i$$

One should note that a domain is a subset of the abstract relation *equal*.

Not all domains are interesting. Many D(N) will be empty, since certain of the semantic classes involved will have no objects in common. Only when the cardinality of a domain approaches the cardinality of its semantic classes can we expect the domain to correspond to a real-world supply of values. We can use this property as a rough measure of the time-dependent state of *completeness* of the relations using a given domain.

A domain can be said to play a role in our Model, equivalent to a Category in Abrial's. The key difference is that Abrial requires each object to belong a priori to a category. In our case, the user may introduce relations that behave like Abrial's categories, e.g. a relation colour that contains all colours, and he may even define the category *category* of all categories. He is, however, not bound to do so by the system, and an object may be a member of zero, one, or more defined domains, according to the need, whereas Abrial requires exactly one membership. It is significant that in our Model the domains are derived from *how the objects appear in relations* on an ad-hoc basis, rather than through their formal introduction to the system as category members.

11.3 Roles

We shall now turn to the case of small domains. Defining a sequence of domains $D(N_1)$, $D(N_2)$, ..., where:

$$N_1 \subset N_2 \subset \ldots$$

we see from the definition that:

$$D(N_1) \supseteq D(N_2) \supseteq \ldots$$

as each new domain in the sequence is produced by further intersection with new semantic classes. By judicially choosing the sets N_i, we may eventually arrive at a domain $D(N_k)$, whose cardinality is one, and $D(N_k)$ will then contain a single pair:

$$D(N_k) = \{<Q_k,Q_k>\}$$

We may intuitively say that the sequence *defines* the object Q_k by stepwise refinement through a number of semantic classes. This is as near as we can come to defining an object from relations.

Now, there are in general other sequences that also define Q_k, so they arrive at the same final domain. One of these sequences must have the property that it leads to a domain which is the intersection of more semantic classes than any other. It follows that this particular domain is defined in terms of all relations where the object Q_k appears, including the parity information (i.e. whether it appears in the right or the left column). This joint information, N_k, is called the *role* of the object Q_k in the knowledge bank B.

From the above definition it follows that a role is a set of pairs, with the relation name at left and the parity information at right. A role is therefore itself formally a relation, but represents a

property, rather than a constituent, of the knowledge bank. Moreover, it has no semantic significance as a relation.

It is not always possible to find a sequence that defines a single object. In many cases we arrive at a minimal *set* of objects, upon which any further intersection with a new domain will empty the set. We shall then say that the objects in this minimal set play *the same role* in the knowledge bank, i.e. there is no way of isolating any particular one of the objects as behaving differently from the others. By "behaving" we mean how the object appears in all relations in B (at left, at right, or not at all). (This is not to say that they are *synonymous*, since they may be paired differently in R_i.) This leads to the formal definition of the role ρ of an object Q_k in the knowledge bank:

$$\rho(k) = \{<R_i,1> \mid \exists Q_r,\ <Q_k,Q_r> \in R_i,\ R_i \in B\} \cup \{<R_i,2> \mid \exists Q_r,\ <Q_r,Q_k> \in R_i,\ R_i \in B\}$$

A set of objects that play the same role ρ' in B is by definition a domain in B, namely $D(\rho')$. Such a set is termed a *role domain* in B, and it follows that:

$$D(\rho') = \{<Q_k,Q_k> \mid \rho(k)=\rho'\}$$

We shall make some final comments on the structuring of the knowledge bank. From the definition, there is a many-to-one correspondence between an object and its role, which means that the role domains of a knowledge bank B are all disjoint. Moreover, if the roles in B are ρ_1, ρ_2... ρ_p, we have:

$$\bigcup_{1 \le i \le p} D(\rho_i) \equiv \mathit{equal}$$

Therefore the role domains completely span the vocabulary of B, and conversely the vocabulary can always be split in a unique way into sets, namely the role domains. Note that certain role domains may be infinite, although the roles themselves are usually finite. This is, of course, due to the presence of abstract relations in B. (A role can be infinite only if there is an infinity of relations, which is not excluded in our Model. If this is the case, the subset of non-empty relations must be finite in the implementation.)

11.4 Semantic Checking

Domains make it possible for the user to group objects into sets that reflect their intended usage in the knowledge bank. So far the new notions have at most served him in his thought process when designing and running an application. We shall now proceed to show how domains can be made to perform more practical tasks.

One such task was hinted at in the example of component colours in the manufacture of a product. Let the domain D1 define all colours that may be used in the manufacture:

D1

red	red
ochre	ochre
green	green
grey	grey

Then the update of the relation:

```
ccomp ← ccomp ∪ <cover,lightpink>;
```

would be contrary to the intent of the user. The user may instead choose to make his updates in the form:

```
ccomp ← ccomp ∪ (<comp,col> * D1);
```

where *comp* is any component and *col* any colour, which would prevent the above pair from being added. The domain has served as a *check* on the semantics of the updating process, allowing only valid colours to be used.

Similarly, if the user wants to check also on components, he might refer to a second domain D2, perhaps derived from a relation catalog, containing a complete list of components. Updating will now take place through:

```
ccomp ← ccomp ∪ (D2 * <comp,col> * D1);
```

This process assumes that the domains were explicitly stored in D1 and D2 at a time when complete knowledge about colours and components was available. However, as we saw in the example, it may so happen that a new colour, say lightpink, is added to the manufacturing process at a later stage (by updating the relations formula and stpaint), which makes the explicit relation D1 out of date. To prevent this, the relational expressions defining D1 and D2 will have to appear in the updating statement. This makes the latter rather unwieldy, especially in the case of multiple semantic classes.

To cope with this difficulty we may introduce a macro:

```
UPDATE R1,R2;
```

which expands into an assignment statement causing R2 to be added to R1 under semantic checking. But in order to function for any R1, the macro must invoke a mechanism that automatically fetches the associated domains. Such a mechanism is the following:

Let the relations *domain* and *range* contain, for every relation subjected to checking, the names or expressions defining the semantic classes used in the left-hand and right-hand check, respectively. In our previous example they would contain:

domain

...	...
ccomp	<1,1>%formula
...	...
ccomp	<1,1>%stpaint
...	...

range

...	...
...	...
ccomp	<2,2>%catalog
...	...
...	...

We then let the *UPDATE* macro perform the equivalent of:

```
R1 ← R1 ∪ ((#~(,R1*domain)) * R2 * (#~(,R1*range)));
```

which handles all cases with a two-sided semantic check. If no check is to be performed, or only on one side, the relation must be entered in *domain* and/or *range* with *equal* as the default domain.

In many cases it becomes apparent after a while that there will be no more modifications to the domains used in checking a given relation. The user may then replace all corresponding expressions in *domain* (or *range*) by the name of a fixed domain that he creates, like D1 above,

especially for this purpose. This *fixing* of a domain is a well-defined and generalisable procedure, and the system may admit another macro for the purpose:

```
FIX  R1 [domain(name1)] [range(name2)];
```

Fixing of domains marks stages in the progress of an application, where certain knowledge has accumulated, and therefore typifies important decision points. The name of each new fixed domain can be entered in a common relation with a time stamp, yielding a record of application evolution.

Domain fixing also has a consequence for the roles of the objects therein. We can divide the roles of any object into two disjoint subsets: the roles ρ_f used in fixing one or more domains, and the "unfixed", or flexible roles ρ_u used for checking relations:

$$\rho = \rho_f \cup \rho_u$$

These two roles can be said to correspond, respectively, to declarative and procedural knowledge about the object, and be made available for inspection through other macros.

In many cases, domains are fixed from the start and need never be defined through semantic classes. Typical cases are the decimal digits, the alphabet, and the signs (plus and minus). Non-mathematical examples are the days of the week, the given (frozen) departments in a company, the political parties in a nation, and the signs of the Zodiac. It is often necessary or useful to express domains by abstract relations, which must then have at least an accessor, since they are used in composition in the checking. Examples are integers, real numbers, and all objects starting with the letter *x*.

Furthermore, since checks are made in terms of composition, we need not limit the checking agent to domains, as we have defined them (i.e. subsets of *equal*). We may use the *UPDATE* macro to actually *convert* the objects before they are added, or to supply defaults when the object is not within the domain. Examples are conversion from numeric to character, and from month name to number.

UPDATE is only one example of many macros that can make use of domains. It approaches the Abrial formalism, where objects belong to categories, and membership is automatically checked at update time. Another Abrial feature is the *updator*, which performs a check on the actual pair, once it is clear that the objects, taken separately, pass scrutiny. An updator may, in our case, be the tester of an abstract relation T, and the *UPDATE* macro would essentially be expanded to perform the equivalent of:

$$R1 \leftarrow R1 \cup ((D1*R2*D2) \cap T);$$

Again, to generalise the macro, T would be found in a common relation *updator*, of the same nature as *domain* and *range*. The tester relation can perform simple tests, like *lessthan*, or embody more complex tasks, like security checking. In its extreme, T may be designed as an interactive process that asks the user for permission, possibly with a password.

The examples given in this section merely hint at the possibilities offered by the basic relational language. It is relatively easy to extend the kit of macro facilities into data management, security, integrity checking, and session control.

12. Implementation Issues

Most existing implementations of relational data bases (e.g. [1,2,6]) build upon standard data-base methods, using indexed access within a "flat" file, and the relational properties emulated by

software. Our prototype is based upon the RAM [3] data-base support, which is an exponent of this method. All such implementations suffer from degraded performance due to the emulation, and the systems are not serious competitors to hierarchical data bases in areas where the latter structure is pre-planned, fixed, and convenient to the application.

Good performance has always been at odds with generality. Traditionally, performance problems have been countered already in the design by conforming to the machine architecture (e.g. sequential access), or later in implementation (e.g. coding in machine language). If worst comes to worst, one compromises on the user needs.

Our objectives in the study called for a different plan:

(a) First design a Model that was *proven* to satisfy *any* data structuring and access needs;

(b) Implement (a) without compromising on generality;

(c) Then, if the performance was not adequate, use our experience with the Model to influence further research into relational machinery.

The fact that the Model is based on set theory made it possible to prove (a) with strict mathematical logic. In this we were helped by the extreme economy of Model concepts: binary relations, equal roles of concrete and abstract relations, small set of operators, and attribute-free data objects, that may in turn be relation names. Moreover, due to this economy, the implementation (b) on RAM was fast, elegant, and easily checked out. Also, in the end it turned out that the performance was astonishingly good, even with moderately large knowledge banks.

We believe, however, that a hardware implementation (in which we include firmware) is both feasible and justified, in order to improve the performance further. Conversely, due to the higher cost of hardware, we think that for any such implementation to be worth the while, the provability of the architecture, in the sense of (a) above, must be ensured. This concluding section is intended to give some ideas, in line with our objective (c), on how to proceed in that direction.

12.1 The Vocabulary

With RAM it is possible to assign a unique numeric identifier, here called *token*, to each object in the knowledge bank. The important characteristic of tokens is that they are all of the same length. If objects are represented by their tokens in all relations, this may improve access performance, especially if the tokens fit the registers of the executing machine. Moreover, if the tokens are shorter than most objects, substantial saving in space can be achieved.

RAM defines (1) an entity space, in which all objects known to the system are stored, and (2) a relation space where binary relations are stored. The entity space can therefore be regarded as the *vocabulary* of the system, and hence an embodiment of the concrete part of the universal relation `vocab`.

It is important that the entity space maintain a one-one correspondence between objects and their tokens. It is possible to enforce this property with RAM, which in turn means that an object is stored exactly once if known to the system. However tempting, this one-one correspondence can not, of course, be implemented as a physical relation, although it is formally possible to describe it as such, as shown by Abrial.

RAM allows any integer to be stored in a relation, irrespective of its role as a token or not. However, our prototype restricts relations to contain only tokens. If it did not, a relational operator might erroneously treat a token 15 and an intended object `'15'` as equal. The increased overhead of having to refer to the vocabulary is less important than one might think. The

conversion between object and token is performed only upon data entry and exit from the system, and within the system only in the non-fundamental relational programs (cf. Section 7.6).

The implementation of the Vocabulary is a topic of its own, worthy of future research. It will not be further discussed here, except noting that for efficiency reasons any implementation of the Model should probably contain a separate Vocabulary of the described type. This is assumed to be the case in the following discussion, and an "object" is usually represented by its token. We can note, however, that for purity and simplicity reasons, at least one other system [6] has no separate vocabulary, and actually stores objects as full character strings in all relations. In a sense, such a system is therefore nearer to our Model than RAM.

12.2 Relational Models

In a certain sense, there is no question that an n-ary relational Model is more general than a binary one. An n-ary knowledge bank, A, consists of N_1 relations A_k, each represented by a two-dimensional array of objects Q_{ijk}:

$$A = \{A_k \mid k = 1,2,...N_1\}$$

$$A_k = \{<Q_{i1k}, Q_{i2k},..., Q_{in_kk}> \mid i = 1,2,...m_k\}$$

where m_k and n_k are the cardinality and degree of relation number k. By fixing the degree n_k to *two* for all relations, we arrived at a knowledge bank, B, of N_2 ($>>N_1$) binary relations B_k:

$$B = \{B_k \mid k = 1,2,...N_2\}$$

$$B_k = \{<Q_{i1k}, Q_{i2k}> \mid i = 1,2,...m_k\}$$

with considerable gain in uniformity of representation and simplicity of operation. On the other hand, the user view of the data may have suffered in clarity, to the extent he handles relations other than binary (including unary). Those must be emulated by binary relations. (At this point one may note that a further reduction to solely unary relations is possible only for very trivial knowledge banks.)

We may now notice that the knowledge bank B is not fully homogeneous, as the relations still differ in their cardinalities. There is also a lack of uniformity in that an object, which is the name of a non-empty relation, must be associated with what amounts to the "address" of the table of pairs that makes up the relation in some memory (this association may itself be stored as a relation). The connotation with address is one of the fundamental drawbacks that the relational concept seeks to overcome, and it will be understood that the subdivision of the knowledge bank into distinct tables of pairs will always introduce addresses in one form or another.

As has been noted in several earlier studies (e.g. [9,10]), it is possible to take a second and last step towards total homogeneity. The knowledge bank, C, is then defined as *one ternary* relation:

$$C = \{<Q_{i1}, Q_{i2}, Q_{i3}> \mid i = 1,2, ... N_3\}$$

which now represents all binary relations in B in such a way that for any relation B_k:

$$<Q_{i2}, Q_{i3}> \in B_k \quad \textit{iff} \quad <B_k, Q_{i2}, Q_{i3}> \in C$$

Since C is a *set*, no specific order is implied between the triplets, and nothing requires triplets belonging to a relation to follow in sequence in any sense. The addressing $i = 1,2,...N_3$ in the above definition is therefore spurious and is merely a notational convenience (we might instead write $i \in N$, where $N = \{1,2,...N_3\}$).

12.3 The Ternary Model

C is completely devoid of any addressing, both internally and from the outside, and can therefore be said to be a true model of an *associative memory*. All information in C is contained in the ternary associations between objects. In particular, an object Q_i is a relation name (or, more precisely, names a non-empty relation) if and only if:

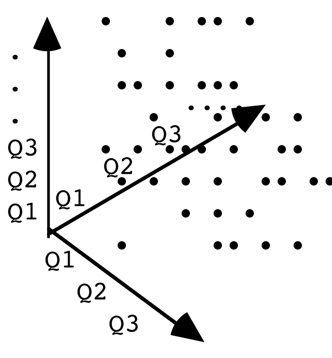
$$\exists \langle Q_j, Q_k \rangle \mid \langle Q_i, Q_j, Q_k \rangle \in C$$

A corollary of the above is that there is no specific "start" or "end" point in C, which makes its illustration as a $3 \times N_3$ table misleading. A better way of depicting the model is as a *cylindrical matrix*:

Q11 Q12 Q13
Q21 Q22 Q23
Q31 Q32 Q33
... ...
... ...

which gives the clue to a possible hardware solution, although physically revolving devices are not considered in a modern implementation [13].

Another interesting representation, suggested by a colleague, consists of a three-dimensional array of dots. All objects in the vocabulary are ticked off (in some known order) along each of the three axes of a coordinate system, and a dot is painted in space at the position $\langle Q_i, Q_j, Q_k \rangle$ if and only if that triplet belongs to C:



This representation becomes more economical than the cylindrical matrix when a significant percentage of all possible triplets are contained in C for a given vocabulary $\{Q_i\}$. Queries correspond to points, lines and planes (parallel to the axes) in this space. (It was even speculated that there might be a natural connection between the ternary relation, as the simplest homogeneous way of perceiving relationships between objects, and our perception of space as three-dimensional.[3])

Though an electro-mechanical implementation of the 3D dot array seems out of the question, the possibility of holographic generation was briefly considered [7]. The idea was abandoned, or at least tagged for future exploration, due to signal/noise limitations even for relatively small $\{Q_i\}$.

---

[3] Cf. the conjecture in [14] that the Universe must be 3D to harbour living organisms.

Moreover, the hologram does not lend itself to easy modification. The representation may of course be stored as a (compressed) bit string in a conventional memory.

12.4 The Cylindrical Memory

The cylindrical relation space brings up a number of interesting implementation points, which will only be briefly hinted at. It has an obvious implementation as a drum storage, provided the triplets are of the same length, which in practice means that tokens will be used. Since the memory is usually not full, there must be a representation of a vacant triplet, and one may reserve a token (e.g. zero) for this purpose.

Given a suitable read/write mechanism, a number of important operations can be carried out during *one* revolution of the drum. Moreover, there is no reason to start the operation at any particular position so the traditional "seek" or "search" steps are meaningless. Such a revolution without a particular starting position may be said to correspond to one *machine cycle*. We have investigated a convenient instruction set based upon this cycle, and found that the basic Model operators can be satisfied by some ten machine instructions.

Although physically revolving devices are perhaps not expected to be an actual implementation medium, we may get an idea of the response time by considering some direct-access technologies. With a rotation time (cycle time) of 17 ms, a simple but typical query involving 15 instructions would take less than 0.3 seconds, *independently* of the size of the data base up to its maximum.

13. Conclusion

The data-support system reported in this paper has been extensively validated as part of the study. Experience has been gained from a set of widely different application areas, ranging from commercially-oriented with minimal explorative usage to fully-fledged design of complex three-dimensional shapes with interactive graphics. Without exception this validation has shown that the original design objectives and decisions were correct.

By not compromising on technical soundness we have ensured that the full range of applications could be successfully implemented. On the other hand, the limited performance was sometimes felt in production running of the developed applications. Very generally, the key reason for this can be stated as follows: We have on one hand the user's way of thinking — his experience and application problem — in terms of relationships, essentially involving parallel processing, which has been closely modelled in the support system. On the other hand we have a sequentially addressed machine on which the model has to run. (One can say that the computer maintains one relation, namely between a data item and its address, and does this extremely well!) The mismatch between the two mechanisms slows down the processing of even simple queries — simple to the mind of the user. In fact the relational operations are intrinsically simple, straightforward, and general, even more so than the conventional computer's own instruction set.

What our study has shown in this respect is that very basic relational support appears to lie much closer to the real world of computer applications than perhaps earlier realised. It is therefore our contention that the implementations briefly outlined in the previous section, and in particular the ternary cylindrical memory, hold significant promise as candidates for future knowledge banks.

References

1. J.R. Abrial: "Data Semantics", University of Grenoble (1973)

2. S. Todd: "The Peterlee relational test vehicle", IBM Sys J, 15,4 (1976)

3. R. Lorie & A.J. Symonds: "A schema for describing a relational data base", IBM G320-2059 (1970)

4. P. Hitchcock: "Fundamental operations on relations", IBM UKSC 0051 (1974)

5. E.F. Codd: "A relational model of data for large shared data banks", Comm ACM 13,6 (1970)

6. M.G. Notley: "Peterlee IS/1 system", IBM UKSC 0018 (1972)

7. J.A. Jordan & P.M. Hirsch, IBM Palo Alto Scientific Center, priv. comm.

8. P.A.V. Hill: "Optimisation of a single relational expression in a relational data base system", IBM UKSC 0076 (1975)

9. A.J. Symonds: "Auxiliary-storage associative data structure for PL/I", IBM Sys J, 7 p.229 (1968)

10. J.A. Feldman: "Aspects of associative processing", MIT Lincoln Lab. Techn. Note 1965-13 (1965)

11. G. Duwat: "Développement d'applications dans un bureau d'étude", Point de l'Informatique, IBM France

12. K. Soop: "Data aspects of graphical applications, experience from an engineering joint study", Intl conf on data base techniques for pictorial applications, Florence 1979.

13. A survey of such implementations is given in Datamation, Jan 1979, p.146.

14. J. Barrow & F. Tipler: ”The Anthropic Cosmological Principle”, Oxford Univ. Press p. 269 & 274 (1986)