Generalized Type Specification for Database Systems

David W. Stemple
Computer and Information Science Department
University of Massachusetts
Amherst, Mass. 01003

82-15

Generalized Type Specification for Database Systems

Abstract

A generalized type specification (GTS) technique for defining database sytems is presented. The purpose of GTS is to provide a unified treatment of operation and data type definition in order to specify well-constrained database systems clearly and in a manner which leads to well-structured implementation techniques.

Examples of GTS definitions of data and operation types from both relational and Codasyl models are given. In the process, various concepts of semantic data models are expressed in GTS terms. Among the more complex of these are convoys, cover aggregations, Codasyl sets, and mandatory set membership. It is shown how the concept of a database type leads to explicit statements of such diverse constraints as referential integrity and the global uniqueness of Codasyl database keys.

Operation types are introduced as a means of specifying both transition constraints and the composite operations involved in automatic set maintenance and triggers. The treatment of operation types demonstrates the validity of viewing triggers and certain aspects of Codasyl schemas, such as set occurrence selection, as operation type definition. Though the implementation aspects of GTS are beyond the scope of this paper, it is suggested that GTS offers an approach to data and operation type definition which can lead to the effective implementation of well-constrained database systems.

Generalized Type Specification for Database Systems

Introduction

The goal of this paper is to describe an approach to type definition which facilitates the clear specification and effective implementation of complex database sytems. A specification technique is presented which is based on first order predicate logic and sets. The technique is not limited to any data model and is applied to both the data and operation types in a system being defined. For this reason, the technique is called generalized type specification (GTS).

There are three basic ideas behind generalized type specification. The first is that a type is composed of a set and a defining collection of properties. The collection of type-defining properties can be expressed by a type-defining predicate which must be true for every instance of the type and false for everything else. The second idea is that operations, being functions, are definable as subsets of Cartesian products, and thus should be amenable to a specification technique capable of defining arbitrarily constrained relations. The third idea is that all constraints on a database system, whether in the data model, or specified explicitly as semantic constraints, existence conditions, operational constraints, relational dependencies, automatic set maintenance, or even triggers, should be specifiable as type-defining predicates of the types which constitute the system.

In the following, type will be formally defined and four methods of type construction will be identified and described. The four kinds of construction will then be applied to data type definition, showing in the process how a wide variety of semantic constraints can be incorporated easily into GTS type definitions.

Among the examples is the construction of types which capture the structure of convoys and cover aggregation [CODD79]. Aspects of the Codasyl system are also specified. This leads to a discussion of operation types with examples from both Codasyl and relational models.

Finally, relational dependencies and normal forms are placed in the context of GTS, and it is shown that while dependencies are type defining properties of relation types, normal forms can only be expressed either as predicates on the set of all legal instances of a relation type (not as predicates evaluatable on individual relations) or as theorems involving type definitions, essentially predicates on predicates.

Types

A type is defined as a pair consisting of a set and a predicate of one free variable. The set is called the value set of the type and contains exactly those elements in the universe for which the predicate is true. The predicate is called the type-defining predicate or intension of the type. We denote the type-defining predicate of T by the clause

type[t] is T

Thus type T is defined by

T ::= <VS, type[t] is T> where $t \in$ VS iff type[t] is T

VS is used in the following as a function on a type, and is defined by

VS(T) ::= {t:type[t] is T}

A member of the value set of a type is called an instance of the type.

This definition is not significantly different from that given by Deutsch [DEUT80].

Type construction

Type definition in GTS uses four kinds of type construction. The first construction is simple classification. A type T is defined by simple classification whenever the type definition is of the form

Define type[t] is T by: t ϵ X

where X is an enumerated set or the value set of a previously defined type. In the following we will often leave some types undefined, such as INTEGERS, POSINTS (positive integers), and NATNUMS (natural numbers), and define new types in terms of the undefined types by use of classification and the other constructors defined below.

The second construction is composition/decomposition. Composition refers to forming tuples, and decomposition refers to extracting parts of tuples. A type T is defined by composition when the type definition is of the form:

Define type[t] is T by:
t ϵ VS(A1) X...X VS(An) where Ai is a type for i = 1,n.

In the following, COMP(A1,..., An) will mean VS(A1) X...X VS(An), and

t IS A TUPLE IN COMP(A1,...,An)

will stand for the type-defining predicate of composition, t ϵ COMP(A1,...,An). Composition is equivalent to Smith and Smith's aggregation [SMIT77], except for certain instance level assumptions in the aggregation concept. See the treatment of specialization below for a discussion of these assumptions.

It will sometimes be useful to associate variables with the elements of a tuple. This will be written

t IS A TUPLE(a1,...,an) IN S defined as t ϵ S and t = <a1,...,an>

Other useful notation for tuples and sets of tuples is defined by

For t and A1,...,An defined as above,
Ai(t) ::= ai

```
If X = {Aj1,...,Ajm} ⊆ {A1,...,An} and the ji values ascend with i,
X(t) ::= <Aj1(t),...,Ajn(t)>

If X is as above and ts ⊆ COMP(A1,...,An), then
X(ts) ::= {X(t):t ∊ ts}
```

The last form, X(ts), is relational projection.

Decomposition refers to the use of projection on the value set of one type to form the value set of another type. A type D is defined by decomposition of type T, defined above by composition, if D's definition is of the form

```
Define type[d] is D by:
d IS A TUPLE IN X(VS(T))
where X = {Aj1,...,Ajn} as above.
```

X(VS(T)) is defined since VS(T) is a set of tuples. X(VS(T)) will be written DCOMP(T,X), and if X is enumerated, the set brackets will be dropped. For example,

```
DCOMP(T,X) ::= DCOMP(T,Aj1,...,Ajn)
```

The third construction is <u>grouping</u>. This is basically the same as Brodie's association [BROD81]. A type T is defined using grouping when the type-defining predicate is of the form

```
Define type[t] is T by:
t ⊆ VS(T')
where T' is a previously defined type.
```

VS(T) is thus the power set of VS(T'). In other words, the value set of T, formed by grouping instances of T', consists of all the subsets of the value set of T'. The following notation is used for the above predicate

```
t IS A SET IN SETS-OF(T')
```

SETS-OF is a function on a type and is defined as the power set of the type's value set.

The fourth construction is derivation. It is used to define types whose value sets are subsets, intersections, unions, and differences of the value sets of previously defined types. A type T is defined by derivation when its specification is of the form

Define type[t] is T by: Pred

where

1. Pred is equivalent to a well formed formula of the first order predicate logic,
2. quantifiers may only be over instances of types defined using SETS-OF, and
3. predicates and functions defined on the instances of referenced, predefined types may appear in Pred. (This includes set predicates ($\subseteq$,=) and set functions ($\cup$,$\cap$,-) for types defined using SETS-OF, and arithmetic operators (+,-,*,/) and relational predicates (<,>) for types defined using numbers.)

Derivation may be used to define subtypes. Type S is defined to be a subtype of T (or type T to be a supertype of S) if and only if PS, the type-defining predicate of S, implies PT, the type-defining predicate of T. A sufficient condition for S to be a subtype of T is for PS to be equivalent to PT $\wedge$ P, for some predicate P. Subtype and supertype are obviously related to specialization and generalization [SMIT77], but are not the same (see below).

Derivation predicates will often be combined with composition and grouping predicates to avoid defining intermediate types when only the resulting type is of interest. The LIST example in the next section uses a combination of composition, grouping, and derivation, all in one predicate.

This set of four kinds of construction is based on Brodie's conjecture that his three constructors, aggregation, generalization, and association, may be sufficient to the task of defining any interesting semantic data model [BROD81].

Examples of GTS applied to data

The use of GTS constructions to define data types will now be illustrated by examples. (The clause "IS IN S" is synonomous with "$\in$ S" if S is a set or "$\in$ VS(S)" if S is a type.)

I. Classification

1. Define type[jc] is JOBCLASS by: jc IS IN {'SEC', 'TRUCKER'}

2. Define type[d] is DISTANCE by: d IS IN NATNUMS

II. Composition/decomposition

1. Define type[et] is EMPTUPLE by:
   et IS A TUPLE IN COMP(ENO,DNO,ENAME,SALARY,JCLASS)

   The value set of EMPTUPLE is a set of tuples. An instance of the type is a tuple not a relation.

2. Define type[en] is ENONAME by:
   en IS A TUPLE IN DCOMP(EMPTUPLE,ENO,ENAME)

   The difference between DCOMP(EMPTUPLE,ENO,ENAME) and COMP(ENO,ENAME) is that the DCOMP set is a subset of EMPTUPLE's value set which may have been restricted by derivation predicates. For example, EMPTUPLE's ENO may have to be above 500 for names starting with "Z". ENONAME as defined would inherit this constraint, but would not if defined simply with COMP(ENO,ENAME). However, ENONAME need not have any instance level association with EMPTUPLE (or with EMPS defined below), since ENONAME could be used to form a relation of numbers and names for retired employees who are no longer recorded in the EMPS relation.

III. Grouping

1. Define type[r] is EMPS by: r IS A SET IN SETS-OF(EMPTUPLE)

   This defines the relation type EMPS, each instance of which is a relation of EMPTUPLE tuples. The value set of EMPS is a set of relations, i.e., a set of sets.

2. Define type[list] is LIST by:

```
list IS A SET IN SETS-OF(COMP(POSINTS,ELEMENT,POSINTS))
AND ∃ le1 SUCH THAT le1 IS A TUPLE(1,el1,n) IN list
AND ∃ lem SUCH THAT lem IS A TUPLE(m,elm,m) IN list
AND ((le IS A TUPLE(i,eli,j) IN list) IMPLIES
(((i ≠ j) IMPLIES (∃ EXACTLY ONE TUPLE(j,elj,k) IN le))
AND ((i ≠ 1) IMPLIES (∃ EXACTLY ONE TUPLE(k,elk,i) IN le)))
```

This defines a list type, adapted from [FLEC71]. The first existential quantification clause guarantees a first element, the second, a last element, the third, exactly one successor for all elements but the last (i = j in the last), and the fourth specifies exactly one predecessor for all but the first element.

IV. Derivation

1. Define type[a] is AGE by: a IS IN POSINTS AND a < 200

2. Define type[t] is TRUCKERS by: t IS IN EMPS AND JCLASS(t) IS {'TRUCKER'}

3. Define type[u] is UNIONEMPS by: u IS IN TRUCKERS OR u IS IN SECRETARIES

Constraints

Nearly all constraints of the various database management systems and semantic data models can be expressed as type-defining predicates which specify the types of a database system. Those which cannot be captured in this way include type naming rules, normal forms, and other "meta-constraints". In relational systems, there are four levels of data type definitions, and two of operation type definitions, each of which has a kind of constraint associated with it. In the following, the data constraints of each of the four levels are discussed. The operational constraints are left to the section below on operation types.

The first level of constraint is contained in the domain definitions. Domain definitions in the relational model are of the form

```
Define type[d] is D by:
d IS IN PRIMITIVE AND DOMAIN-CONSTRAINT
```

where DOMAIN-CONSTRAINT is a derivation predicate on the free variable d, and PRIMITIVE is one of the primitive simple types of the system being used.

The next level of constraining is in the form of tuple constraints, which are parts of a tuple type definition. Tuple type definitions are of the form

```
Define type[t] is T by:
t IS A TUPLE IN COMP(A1,...,An)
AND TUPLE-CONSTRAINT
```

where TUPLE-CONSTRAINT is a derivation predicate of the free variable t.

Relation constraints are clauses in relation type definitions. Relation type definitions are of the form

```
Define type[r] is R by:
r IS A SET IN SETS-OF(COMP(A1,...,An))
AND RELATION-CONSTRAINT
```

where RELATION-CONSTRAINT is a derivation predicate on the free variable r.

One important relation constraint is that certain columns be a key of a relation type. For example, suppose that ENO is to be a key of type EMPS defined above. First, for r IN SETS-OF(COMP(A1,...,An)) and K = {Aj1,...,Ajm} as above, define the clause

```
K IS A KEY OF r ::=
FOR ALL ti,tj IN r ((K(ti) = K(tj)) IMPLIES (ti = tj))
```

This defines KEY as a property of a relation. (If K consists of one member, the set brackets will be dropped.) To impose the key constraint on a relation type, the clause must be included in a type definition. For example, the definition of EMPS may be amended to read

```
Define type[es] is EMPS by:
es IS A SET IN SETS-OF(EMPTUPLE) AND ENO IS A KEY OF es
```

The key constraint cannot be specified as a type-defining property of the tuple type. Functional (and other) dependencies can be specified in the same manner and will be discussed separately.

Aggregate constraints are another form of relation constraints. They are simply predicates which use functions on sets of tuples, such as average or count, to refine a relation type.

Interrelational constraints require a type constructed by forming tuples composed of relations. This is the fourth level of data type definition needed for a well-constrained set of relational types.

Consider interrelation constraints wherein certain values in one relation (an instance of one relation type) are constrained to be in some relationship to certain values in a simultaneously occurring instance of another relation type. There is really no way to define such constraints as type-defining properties of either of the two relation types, since such constraints need not have any effect on the value set of either type. As discussed by Hammer and McLeod [HAMM76], a derived relation could be formed, say by a join, and constrained in the manner in which any other relation could be constrained. Sharman [SHAR76] constructs a multilevel graph to facilitate the expression of interrelational constraints. In the following, an approach similar to Sharman's is taken, but no new type constructors are required.

Take, for example, the case in which EMPS is the relation type defined above and DEPTS is defined by

```
Define type[d] is DEPTTUPLE by:
d IS A TUPLE IN COMP(DNO,DNAME,LOC,NUMEMPS)


Define type[ds] is DEPTS by:
ds IS A SET IN SETS-OF(DEPTTUPLE)
```

In order to express the interrelation constraint that the departments referred to in EMPS be exactly those in DEPTS, the following specification may be written

```
Define type[cdb] is COMPANY-DBASE BY:
cdb IS A TUPLE (ds,es) IN COMP(DEPTS,EMPS) AND DNO(es) = DNO(ds)
```

Each instance of COMPANY-DBASE is a tuple containing one relation each of types EMPS and DEPTS. An instance of COMPANY-DBASE is what is normally called a state of the database. Any type-defining property of such types, which will be called relational database types, is a constraint on legal states of the database. Those constraints which cannot be expressed in the type-defining predicates of the constituent types are interrelational constraints and in GTS become database constraints, parts of the database type definition.

Many of the semantic constraints of various semantic data models are easily expressed by clauses in database type-defining predicates. For example, the clause, DNO(es) = DNO(ds), specifies a referential integrity constraint [CODD79]. Existence constraints are similar and are illustrated by a simple example.

Suppose the following two type definitions are added to the company database.

```
Define type[d] is DEPENDENT by:
d IS A TUPLE IN COMP(ENO,NAME,AGE)

Define type[dr] is DEPENDENTS by:
dr IS A SET IN SETS-OF(DEPENDENT)
```

Suppose further that a DEPENDENT tuple is not to be allowed in the database unless there is an EMPS tuple with a matching ENO component currently in the EMPS part of the database. This is specified in the database type definition

```
Define type[cdb] is COMPANY-DBASE by:
cdb IS A TUPLE(ds,es,dr) IN COMP(DEPTS,EMPS,DEPENDENTS)
AND DNO(es) = DNO(ds)
AND ENO(dr) ⊆ ENO(es)
```

The last clause expresses an existence dependence of dependents on employees. Note that such a constraint could only be expressed in the type definition of DEPENDENTS if there were implicit instance level interdependencies already assumed. The difference between aggregation and composition referred to above, and the difference between specialization and subtype lies in the existence of such interdependencies, absent in GTS composition and subtypes, present in aggregation and specialization.

Suppose, for example, SECRETARIES is specified by

```
Define type[ss] is SECRETARIES by:
ss IS A SET IN EMPS AND JCLASS(ss) IS {'SEC'}
```

SECRETARIES is a subtype of EMPS but not a specialization, since in GTS there is no implicit assumption about the coexistence of instances of related types. In order to make SECRETARIES a specialization of EMPS, the database type COMPANY-DBASE could be specified by

```
Define type[cdb] is COMPANY-DBASE by:
cdb IS A TUPLE(ds,es,ss) IN COMP(DEPTS,EMPS,SECRETARIES)
AND ss ⊆ es
```

Generalization, of course, requires a similar interpretation, although both it and specialization need the application of DCOMP if the general type is composed from less constituent types than the specialized type.

Convoys, cover aggregrations, and partitions

To build types which faithfully represent convoys, cover aggregations [CODD79], associations [BROD81] (different from Schmid's associations [SHMI77]), and aggregates [HAMM78] (different from Smith and Smith's aggregates [SMIT77]) no additional constructors are needed. The following illustrates the use of GTS constructions to define types which capture the semantics of ships and convoys. The existence of definitions for the simple types, SHIPID, STYPE, NAME, CONVOYID, FLAGSHIPID(=SHIPID), and NUM(=NATNUMS) is assumed.

```
Define type[s] is SHIP by: s IS A TUPLE IN COMP(SHIPID,STYPE,NAME)

Define type[ss] is SHIPS by: ss IS A SET IN SETS-OF(SHIP)
AND SHIPID IS A KEY OF ss
```

```
Define type[c] is CONVOY by: c IS A TUPLE(cid,fsid,ss,n) IN
COMP(CONVOYID,FLAGSHPID,SHIPS,NUM) AND fsid IS IN SHIPID(ss)
AND n IS COUNT(ss)

Define type[cs] is CONVOYS by: cs IS A SET IN SETS-OF(CONVOY)
AND CONVOYID IS A KEY OF cs
```

This defines two relation types, SHIPS and CONVOYS, whose tuples represent ships and convoys, respectively. Each convoy tuple contains as one component a set of ship tuples. Thus, CONVOYS relations are not in first normal form. CONVOY tuples are different from Hammer and McLeod's aggregates [HAMM78] in that they have attributes. Attributes such as NUM are tied to sets of ships in a manner totally consistent with that used in defining any tuple type.

Another difference between Hammer and McLeod's aggregates and this use of GTS SETS-OF types is that aggregates seem to imply some instance level connection between the aggregate type and its base type. For example, in Hammer and McLeod's SHIPS/CONVOY example there appears to be the implicit assumption that aggregates at any state will be formed from ships in existence in that state. In GTS this must be made explicit.

In the following database type definition, the explicit relationship between instances is defined. First, for S a set and SS a set of sets, define the predicate

```
SS PARTITIONS S ::=
S = UNION(SS) AND
FOR ALL si,sj IN SS ((si ≠ sj) IMPLIES (si ∩ sj = Ø))
```

Using this predicate

```
Define type[ndb] is NAVYDATABASE by:
ndb IS A TUPLE(ss,cs) IN COMP(SHIPS,CONVOYS)
AND SHIPS(cs) PARTITIONS ss
```

In this way the instance level semantic connection between convoys and ships is expressed explicitly. This definition represents the fact that a ship may be in only one convoy, and that every ship must be in a convoy.

It should be clear that a similar treatment could be given cover aggregation in which the partition predicate is replaced by a simple existence constraint as in the EMPS, DEPENDENTS example above.

The CONVOYS relation is clearly not in first normal form. It may be desirable, indeed in some implementations it could be required, that there be a set of "simple" types in any system which constitutes a basis for all component types in the system. A basis is a set of types, the instances of which may be used to derive the simultaneously occurring instances of all other types in the system. In some implementations simple might mean first normal form relations.

If such were a requirement the SHIP type could be amended to

```
Define type[s] is SHIP by:
s IS A TUPLE IN COMP(SHIPID,STYPE,NAME,CONVOYID)
```

and CONVOY to

```
Define type[c] is CONVOY by:
c IS A TUPLE(cid,fsid,ss,n) IN COMP(CONVOYID,FLAGSHIPID,SHIPS,NUM)
AND fsid IS IN SHIPID(ss) AND n IS COUNT(ss) AND CONVOYID(ss) = {cid}
```

The last clause in this definition requires that only ships having the correct convoy id may be in the set of ships of a convoy.

Decomposition (DCOMP) may now be used on the CONVOY type to produce a first normal form relation SCONVOYS which still maintains the semantic integrity of the more complex CONVOYS. SCONVOYS is specified by

```
Define type[scs] is SCONVOYS by:
scs IS A SET IN SETS-OF(DCOMP(CONVOY,CONVOYID,FLAGSHIPID,NUM))
```

This defines SCONVOYS to be equivalent, as a type, to CONVOYS without the ship set column. In order to constrain SCONVOYS to the same instance level semantics as CONVOYS, the database type NAVYDATABASE must be specified by

```
Define type[ndb] is NAVYDATABASE by:
ndb IS A TUPLE(ss,cs,scs) IN COMP(SHIPS,CONVOYS,SCONVOYS)
```

```
    AND SHIPS(cs) PARTITIONS ss AND SCSATTRS(cs) = scs
    where SCSATTRS = {CONVOYID,FLAGSHIPID,NUM}.
```

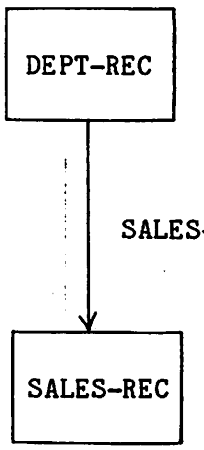
The last clause requires that the simple relation SCONVOYS be linked semantically to CONVOYS at every state of the database. Thus, CONVOYS could be used to derive SCONVOYS, or more importantly, SCONVOYS could be used to derive CONVOYS. The two simple relation types SHIPS and SCONVOYS form a basis for the database type.

It is often convenient to deal in an implementation with simple base types, but use the more robust types to express complex integrity constraints. This is an alternative to the technique of the semantic data model of Hammer and McCleod [HAMM76] in which constraints are expressed by using a query facility to identify sets of constrained data and constraining data. Which of these approaches is better will be determined by their software and human engineering aspects.

Though it will not be discussed here, it should be reasonably clear from this brief discussion, that view definition can be accomplished in GTS by use of decompositions and derivations from database types (and associated operation types).

Codaysl constructs

Models which use constructs equivalent to Codasyl sets (cosets) can be specified in a straightforward manner using GTS constructions. Consider the two record types and coset in the data structure diagram



The following type definitions capture the essential structure of a Codasyl treatment of this diagram.

```
Define type[d] is DEPT-REC by:
d IS A TUPLE IN COMP(DBKEY,DNO,DNAME)

Define type[df] is DEPT-FILE by:
df IS A SET IN SETS-OF(DEPT-REC) AND DBKEY IS A KEY OF df
AND DNO IS A KEY OF df

Define type[s] is SALES-REC by:
s IS A TUPLE IN COMP(DBKEY,DNO,ITEM,VOL)

Define type[sf] is SALES-FILE by:
sf IS A SET IN SETS-OF(SALES-REC) AND DBKEY IS A KEY OF sf

Define type[so] is SVSETOCC by:
so IS A TUPLE IN COMP(DEPT-REC,SALES-FILE)
```

The last definition specifies the structure of coset occurrences relating department records and sets of sales records. Sets of sales records are defined by the SALES-FILE type definition, thus the type appears as an argument to the COMP function. The SVSETOCC definition does not stipulate that the sales records' DNO match the DNO of the department owner. For purposes of illustration, this is left to the definition of the insert type definition in the next section.

To define the type of coset occurrence collections, the following is specified

```
Define type[svs] is SALES-VOL by:
svs IS A SET IN SETS-OF(SVSETOCC) AND DEPT-REC IS A KEY IN svs
AND SALES-FILE(svs) PARTITIONS UNION(SALES-FILE(svs))
```

The last two clauses express the properties which are possessed by any coset, namely that each member occurrence may be related to only one owner occurrence.

Again, a database type is needed to express more complex constraints such as the global keyness of database keys (DBKEY), and mandatory membership of sales records in sales volume cosets.

```
Define type[ds] is type DEPTSTORE by:
ds IS A TUPLE(df,sf,svs) IN COMP(DEPT-FILE,SALES-FILE,SALES-VOL)
AND {DBKEY(df),DBKEY(sf)} PARTITIONS UNION(DBKEY(df),DBKEY(sf))
AND SALES-FILE(svs) PARTITIONS sf.
```

The next to last clause expresses the property of database keys that they be unique across the complete database. The last clause asserts that sales records are mandatory members of the SALES-VOL set.

In order to use the Codasyl set structure as a primitive to avoid the detailed specification each time it is required, a predicate involving the function COSET (a composite constructor) could be defined by

```
cs IS IN COSET(OWNERTYPE,MEMBERTYPE) ::=
cs IS A SET IN SETS-OF(COMP(OWNERTYPE,MEMBERTYPEFILE))
AND OWNERTYPE IS A KEY OF cs
AND VS(MEMBERTYPEFILE) IS SETS-OF(MEMBERTYPE)
AND MEMBERTYPEFILE(cs) PARTITIONS UNION(MEMBERTYPEFILE(cs)).
```

Then the definitions of SVSETOCC and SALES-VOL could be replaced by

```
Define type[svs] is SALES-VOL by:
svs IS A SET IN COSET(DEPT-REC,SALES-REC)
```

Similar treatment could be used to define mandatory membership and other primitives of the Codasyl model. This approach to extending the type constructors is an alternative to splitting types into high-level types of a data model and low-level specifiable types, or type concepts and types[LOCK79].

In order to specify automatic coset membership, GTS needs to be applied to operation types. In the next section, definitions of operation types will be discussed and the automatic membership of sales records will be specified using the composition of Codasyl operations store and insert. This is the same view used in the implementation of Codasyl schemas reported in [STEM76], where it was shown that Codasyl schemas could be used to generate tailored operation implementations which were, in effect, implementations of composed operation types.

Operation types

As stated by Deutsch [DEUT80], type systems may be defined for procedures as well as for data. In the following, GTS is applied to database operations. The motivation for this is twofold. First, it is necessary to build type definitions for operations in order to express operational or state transition constraints [HAMM75] as type-defining properties. Second, certain capabilities of existing schema languages entail, in effect, the specification of augmented operation types. Most notable among these is the automatic set membership feature of Codasyl schemas. In the following, it will be demonstrated that these two kinds of constraining facilities can be specified through use of derivation and composition in the definition of operation types.

Before turning to the more complex case of Codasyl set maintenance, the operation type for inserting tuples into EMPS relations, as defined above, will be specified.

Inserting a tuple into a relation is a function of two variables, the domains of which are the value sets of the relation type and its tuple type. The range of the function is the relation type's value set. This is normally expressed by a statement of the form

```
INSEMP : EMPS X EMPTUPLE -> EMPS
```

This, of course, states that INSEMP is a functional subset of the product EMPS X EMPTUPLE X EMPS where the variables EMPS and EMPTUPLE stand for what, in GTS, would be called the value sets of the types EMPS and EMPTUPLE. If an algebraic/axiomatic approach [LOCK79] were being used, further statements specifying the interrelationships between INSEMP and other operations (types) would be written as axioms. In GTS these two kinds of specification are replaced by a definition of the operation type in set-theoretic terms.

Thus, INSEMP can be specified

```
Define type[ins] is INSEMP by:
ins IS A TUPLE(eri,et,ero) IN COMP(EMPS,EMPTUPLE,EMPS)
AND ero = eri U {et}
```

This states that an insert-into-EMPS operation is a tuple consisting of an input relation, a tuple to be inserted, and an output relation, the latter defined in the last clause as the union of the former two. The value set of the type consists of all such operations.

Suppose that at some point in the life of the system it is decided that new employees must be hired at salaries less than the average of existing employees' salaries. At this point the INSEMP definition could be amended to

```
Define type[ins] is INSEMP by:
ins IS A TUPLE(esi,et,eso) IN COMP(EMPS,EMPTUPLE,EMPS)
AND eso = esi U {et}
AND SALARY(et) < AVERAGE(SALARY(esi))
```

This is the way in which operational or transition constraints are specified in GTS, namely as derivation predicates in operation type definitions.

In the case of Codasyl automatic set membership, a store of a member type record occurrence is redefined to be a store followed by an insert of the record occurrence into the appropriate set occurrence. Using the SALES record type defined above, a store-sales-record-into-sales-file operation type could be defined in essentially the same manner as INSEMP. However, the augmented store operation involves another type, namely, the coset type SALES-VOL. Thus the augmented store is best defined as an operation on the database type. In the following, the coset insert operation is defined first as an insert into a coset occurrence, then as an insert into a collection of cosets, and finally as a database operation. Then, the augmented store is specified. The ordering of coset members is not specified so that the level of detail can be kept manageable.

Suppose that a sales record occurrence must be placed into a set occurrence whose owner DEPT-REC occurrence matches the sales DNO. Then the insert of a sales record into a set occurrence is specified by

```
Define type[so] is INS-SAL-IN-SVOCC by:
so IS A TUPLE(socci,s,socco) IN COMP(SVSETOCC,SALES-REC,SVSETOCC)
AND DNO(s) = DNO(DEPT(socci))
AND socco = <DEPT-REC(socci),SALES-FILE(socci) U {s}>
```

If <socci,s,socco> is in INS-SAL-IN-SVOCC then, in the traditional manner, we define INS-SAL-IN-SVOCC(socci,s) ::= socco

From this, inserts into a set of cosets, INS-SAL-IN-SVSET, can be specified by

```
Define type[ins] is INS-SAL-IN-SVSET by:
ins IS A TUPLE(svsi,s,svso) IN COMP(SALES-VOL,SALES-REC,SALES-VOL)
AND svso = svsi - {socc-old} U {socc-new}
where socc-old IS IN svsi and DNO(DEPT(socc-old)) = DNO(s)
and socc-new = INS-SAL-IN-SVOCC(socc-old,s)
```

And this can be transformed into an operation on the database type DEPTSTORE by

```
Define type[ins] is INS-SVOL by:
ins IS A TUPLE(di,s,do) IN COMP(DEPTSTORE,SALES-RECORD,DEPTSTORE)
AND di = <df,sf,svs> AND do = <df,sf,INS-SAL-IN-SVSET(svs,s)>
```

The store sales operation type is specified as a database operation type by

```
Define type[st] is STORE-SALES by:
st IS A TUPLE(di,s,do) IN COMP(DEPTSTORE,SALES-REC,DEPTSTORE)
AND di = <df,sf,svs> AND do = <df,sf U {s},svs>
```

Although the effect of automatic set selection in Codasyl schemas is to redefine the store verb, in the following, a new operation ENTER-SALES with the semantics of the augmented store is defined. Arguments for the appropriateness of this approach, even in the Codasyl environment, involve the inadvisability of visible side-effects and are beyond the scope of this paper.

The augmented store, ENTER-SALES, can be specified by

```
Define type[es] is ENTER-SALES by:
es IS A TUPLE(di1,s1,do1,di2,s2,do2) IN COMP(STORE-SALES,INS-SVOL)
AND s1 = s2 AND do1 = di2
```

A simpler ENTER-SALES type which associates an input state di1 with an output state do2 in the same way as this definition, but is less explicit in its use of composition, is given by

```
Define type[es] is ENTER-SALES by:
es IS A TUPLE(di1,s,do2) IN COMP(DEPTSTORE,SALES-REC,DEPTSTORE)
AND di1 = <df,sf,svs>
AND do2 = <df,STORE-SALES(sf,s),INS-SAL-IN-SVSET(svs,s)>
```

This is equivalent to (and could be specified as) the decomposition of the previously defined ENTER-SALES on the first, second, and last domains.

This demonstrates the validity of viewing certain Codasyl schema facilities as parts of operation type definitions. Triggers [CHAM76] can be interpreted in a similar manner.

To demonstrate the equivalence of triggers to operation type composition, a composite database operation type MOVE-EMP is defined. This type models the event (type) of moving employees from one department to another. The semantic integrity to be preserved is the value of the NUMEMPS column in the DEPTS relation, defined above, as EMPS changes from one instance to another. Triggers have been given as one implementation of this integrity constraint [CHAM76]. In the following, MOVE-EMP will be defined on the database type COMPANY-DBASE specified above using COMP(DEPTS,EMPS).

As in the case of the coset operations above, a number of primitive operation types are defined before the database operations.

```
Define type[upet] IS UPDATE-DNO-T by:
upet IS A TUPLE(ei,dno,eo) IN COMP(EMPTUPLE,DNO,EMPTUPLE)
AND ei = <eno,dno',ename,sal,jc>
AND eo = <eno,dno,ename,sal,jc>
```

This defines the tuple update which takes an EMPS tuple and changes its DNO component to a new value dno. UPDATE-DNO-T is very specific, the very opposite of a generic procedure [BARO81]. The specificity of this definition is for the purposes of illustration and should not be taken as a recommendation for the style of GTS operation type definition. Generic operation types can and should be defined in the same manner as the generic COSET predicate in the previous section.

UPDATE-DNO-T can be used to define an update on EMPS.

```
Define type[uper] is UPDATE-DNO-R by:
uper IS A TUPLE(eri,eno,dno,ero) IN COMP(EMPS,ENO,DNO,EMPS)
AND ero = (eri -{old-tuple}) U {new-tuple}
AND old-tuple IS IN eri AND ENO(old-tuple) = eno
AND new-tuple = UPDATE-DNO-T(old-tuple,dno)
```

In a similar manner, the relation update UPDATE-NUMEMPS of the NUMEMPS column of a DEPTS relation can be defined. MOVE-EMP will now be defined in terms of UPDATE-DNO-R and UPDATE-NUMEMPS.

```
Define type[m] is MOVE-EMP by:
m IS A TUPLE(dbi,eno,dno,dbo) IN
          COMP(COMPANY-DBASE,ENO,DNO,COMPANY-DBASE)
AND dbi = <dsi,esi> AND dbo = <dso,eso>
AND eso = UPDATE-DNO-R(esi,eno,dno)
AND dso = UPDATE-NUMEMPS(dso',olddno,NUMEMPS(old-dtuple)-1)
AND dso' = UPDATE-NUMEMPS(dsi,dno,NUMEMPS(new-dtuple)+1)
where olddno = DNO(old-dtuple),
old-dtuple IS IN dsi and DNO(old-dtuple) = DNO(etuple),
etuple IS IN esi and ENO(etuple) = eno,
new-dtuple IS IN dsi and DNO(new-dtuple) = dno
```

This specifies MOVE-EMP operations as changing the dno of an EMPS tuple and updating the number of employees in both old and new DEPTS tuples.

Two aspects of this definition are apparent. One is that it seems unduly complex in comparison with the trigger specification. There are two reasons for this. The first is that this specification is more complete than the trigger version, in that it specifies the EMPS update and the "old" and "new" semantics which are defined outside the trigger description.

The second reason for the apparent trigger simplicity is its procedural form. At a certain level of complexity, a procedural style of specification becomes simpler to write and understand than a non-procedural style of comparable abstractness. (See [WELT81] for a discussion of this issue in the context of relational query languages.) For specifying complex operation types, a procedural sublanguage should be used in conjunction with the GTS language used in this paper. Such a procedural adjunct would be defined in GTS terms in exactly the way the relational algebra is defined in terms of the relational calculus [CODD71]. Note that this process has already started by using the algebraic operators of set union and difference in operation type definitions. A language combining the GTS language as used here and a well-designed procedural extension would allow database types and all their admissible operations to be defined succintly and with sufficient abstractness.

It is not the procedurality of triggers which is the essential difference between them and the GTS approach to operational integrity maintenance. It is the difference between producing side effects to primitive operations (the primitive update of EMPS in this example) and defining new higher level operation types (eg., MOVE-EMP) which can carry the augmented semantics required for integrity. The Codasyl example of coset selection and insertion above shows a similar difference between schema-produced side effects (the Codasyl approach) and high level operation type definition in GTS (eg., ENTER-SALES).

It should not be thought that a GTS definition of a database system using the operation type approach to data abstraction would be too difficult to implement efficiently. On the contrary, using a method which compiles tailored database managers which implement all of the integrity logic peculiar to a specific application [STEM76], GTS can lead to efficient implementations with a minimum of run-time interpretation. Exploration of the implementation aspects of GTS are beyond the scope of this paper.

Functional dependencies and normal forms

Functional dependencies can be used as type-defining properties of relation types. For example, consider R defined as a relation type by

```
Define type[r] is R by:
r IS A SET IN SETS-OF(COMP(A,B,C))
```

A functional dependency from A to C in the type R (written FD(A,C) IN R) is specified by appending the following clause to the definition

```
AND (FOR ALL ti,tj IN r ((A(ti) = A(tj)) IMPLIES (C(ti) = C(tj)))
```

This clause, without the initial AND, will be used as the definition of FD(A,C) IN r, where r is a relation. The definitions of both FD predicates are valid when A and C are sets of column domains.

Note that FD IN r is a predicate on a potential instance of a type. To say that the FD holds for the type is to say that it holds for each and every instance of the type. Thus FDs can be parts of type-defining predicates. There are, however, properties of a type which are not properties of individual instances. A trivial instance of such a property is the cardinality of a type's value set.

The various normal forms of relations [DATE81] are properties of types in the above sense and are not type-defining properties of relation types. For example, first normal form refers to an absence of composition and grouping in the definition of the constituent types of a relation's tuple type. Second, third, Boyce-Codd (BCNF),fourth, and fifth normal forms are defined by asserting the presence and/or absence of certain patterns or implications in a relation's type-defining predicate. This will be illustrated below in a definition of BCNF.

Much of the difficulty in understanding normal forms has been due to the lack of distinction between type and instance in the literature dealing with functional dependencies. This lack of distinction leads to little difficulty when asserting FDs, since an FD in the type means an FD in each and every instance of the type. This is not the case when asserting that there is not an FD between columns subsets. For example, if it is asserted that there is not an FD between A and B in the relation type above, this does not mean that the clause

```
AND NOT FD(A,B) IN r
```

should be appended to the type definition. This clause would require that no relation in the type could have the FD(A,B) present. This is clearly not what is meant. In fact, such a requirement would preclude every one-tuple relation from being a legal instance of the type.

What is meant by the lack of an FD in the type is that there exists at least one relation in the value set of the type in which the FD does not hold. This is not specifiable by a predicate which is evaluatable on potential instances of the type. The absence of an FD in the type is specifiable in two ways. The first way is illustrated by the following predicate, in which an existential quantification is made over the value set of the relation type.

(THERE EXISTS r IN VS(R)) SUCH THAT (NOT FD(A,B) IN r)

This asserts a property of the type much like the value set cardinality, and clearly cannot be used to determine if a particular relation is an instance of the type.

The second method is to state a theorem involving the type-defining predicate. The theorem is basically a predicate on the intension of the type. This technique is used in the following definition of BCNF.

Definition of BCNF:

```
Let Z = {A1,...,An} be a set of types
and R be a relation type specified by
Define type[r] is R by:
r IS A SET IN SETS-OF(COMP(A1,...,An)) AND P
where P is a derivation predicate.


R is in BCNF if and only if
R is in first normal form and
FOR ALL X,Y ⊆ Z
(P IMPLIES FD(X,Y) IN R) IMPLIES (P IMPLIES FD(X,Z) IN R)
```

Note that this defines the normal form as local to the relation type. A better approach might be to define it in the context of a database type which could involve interrelational dependencies. In this case, P in the BCNF defining predicate would be replaced by P AND PD where PD is the database type-defining predicate. In fact, a full definition should probably take into account the definition of operations which are used in forming the instances of R. Transition constraints in the insert, delete, and update type definitions could imply FDs in all "reachable" instances of R. The reachable instances in a constrained database type constitute the effective value sets of the constituent types, and thus should be the context in which well-formed types are to be strived for.

Summary

A generalized type specification (GTS) technique for defining database sytems has been presented. The purpose of GTS is to provide a unified treatment of operation and data type definition in order to specify well-constrained database systems clearly and in a manner which leads to well-structured implementation techniques.

Examples of GTS definitions of data and operation types from both relational and Codasyl models were given. In the process, various concepts of semantic data models were expressed in GTS terms. Among the more complex of these were convoys, cover aggregations, Codasyl sets, and mandatory set membership. It was shown how the concept of a database type leads to explicit statements of such diverse constraints as referential integrity and the global uniqueness of Codasyl database keys.

Operation types were introduced as a means of specifying both transition constraints and the composite operations involved in automatic set maintenance and triggers. The treatment of operation types demonstrated the validity of viewing triggers and certain aspects of Codasyl schemas, such as set occurrence selection, as operation type definition. Though the implementation aspects of GTS were beyond the scope of this paper, it was suggested that GTS offers an approach to data and operation type definition which can lead to the effective implementation of well-constrained database systems.

REFERENCES

[BARO81] BAROODY, A.J. and DeWITT, D.J. An Object Oriented Approach to Database System Implementation. ACM Trans. Database Syst. 6, 4 (Dec. 1981), 576-601.

[BROD81] BRODIE, M.L. Association: A Database Abstraction for Semantic Modeling. In Entity-Relationship Approach to Information Modeling and Analysis, P.P. Chen, Ed., 1981.

[CHAM76] CHAMBERLIN, D.D., et al. SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control. IBM J. Res Dev. 20, 6 (Nov. 1976), 560-575.

[CODD71] CODD, E.F. A Database Sublanguage Founded on the Relational Calculus. In Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif., 1971, 35-68.

[CODD79] CODD, E.F. Extending the Database Relational Model to Capture More Meaning. ACM Trans. Database Syst. 4, 4 (Dec. 1979), 397-434.

[DATE81] DATE, C.J. An Introduction to Database Systems, 3rd ed. Addison-Wesley, Reading, Mass. 1981.

[DEUT80] DEUTSCH, L.P. Constraints: A Uniform Model for Data and Control. SIGMOD Record. 11, 2 (Feb. 1981), 118-120.

[FLEC71] FLECK, A.C. Towards a Theory of Data Structures. J. of Comp. and Sys. Sci. 5, 5 (Oct. 71), 475-504.

[HAMM75] HAMMER, M. and McCLEOD, D.J. Semantic Integrity in a Relational Database System. In VLDB 1975.

[HAMM78] HAMMER, M. and McCLEOD, D.J. The Semantic Data Model: a Modeling Mechanism for Database Applications. In Proc. ACM SIGMOD Conf., Austin, Tex., May 31 - June 2, 1978.

[LOCK79] LOCKEMANN,P.C., et al. Data Abstractions for Database Systems. ACM Trans. Database Syst. 4, 4 (Mar. 1979), 30-59.

[SHAR76] SHARMAN, G.C.H. A Constructive Definition of Third Normal Form. In Proc. 1976 ACM - SIGMOD Int. Conf. on Management of Data, Washington, D. C., June 1976, 91-99.

[SHMI77] SCHMID, H.A. An Analysis of Some Constructs for Conceptual Models. In Architecture and Models in Data Base Management Systems, G.M. Nijssen, Ed., 1977.

[SMIT77] SMITH, J.M. and SMITH, D.C.P. Database Abstractions: Aggregations and Generalizations. ACM Trans. Database Syst. 2, 2 (June 1977), 105-133.

[STEM76] STEMPLE, D.W. A Database Management Facility for Automatic Generation of Database Managers. ACM Trans. Database Syst. 1, 1 (March 1976), 79-94.

[WELT81] WELTY, C. and STEMPLE, D.W. Human Factors Comparison of a Procedural and Nonprocedural Query Language. ACM Trans. Database Syst. 6, 4 (Dec 1981), 626-650.