View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by The University of Utah: J. Willard Marriott Digital Library

INTEGRATED SPECIFICATIONS
FOR
ABSTRACT SYSTEMS

by

John Miles Smith
Diane C.P. Smith

UUCS - 77 - 112

September 28, 1977

INTEGRATED SPECIFICATIONS

FOR

ABSTRACT SYSTEMS

by

John Miles Smith
Diane C.P. Smith

Structural specifications define an abstract object as a composition of other abstract objects. Behavioral specifications define an abstract object in terms of its associated operations. Integrated specifications are a combination of structural and behavioral specifications which are more powerful than either used alone. By providing four naming mechanisms, integrated specifications hide the details of how objects are represented and accessed on storage devices. The four naming mechanisms allow objects to be named in terms of the operations previously applied to them, the unique attributes they possess, the relationships they participate in, and the categories they belong to. Integrated specifications can specify the structure of more abstract systems than the relational database model, while also characterizing dynamic properties. Examples are given of integrated specifications for queue, symboltable and expression. These specifications are simple and guide, but do not constrain, the implementor in designing refinements. By exploiting abstract structure in specifications, common aspects of inter-object communication can be suppressed and only salient differences emphasized. Integrated specifications can make a significant contribution to the useability, reliability and efficiency of computer systems.

Keywords and phrases: Specification language, data model, database, abstraction, data type, aggregation, generalization.

CR Categories: 3.65, 4.22, 4.33, 4.34.

This work was partially supported by the National Science Foundation under Grant MCS75-09903

Authors' address: Computer Science Department, University of Utah, Salt Lake City, Utah 84112

into simpler objects. Furthermore, the abstract structure of objects often changes more slowly than their associated operations. As a result, structural specifications are often more stable than behavioral specifications. On the other hand, behavioral specifications are essential to verify that abstract objects remain consistent with system invariants. It follows, therefore, that integrated specifications are more complete characterizations than either behavioral or structural specifications alone.

To extend the "aggregate" and "generic" types to support behavioral specifications, two mechanisms must be provided:

i) a mechanism for defining abstract operations, and

ii) a mechanism for identifying the operands of these operations.

As the first mechanism, we will define abstract operations over the primitive operations "insert", "delete", and "modify" using standard control structures such as conditional, iteration and recursion. We will use a PASCAL-like syntax for expressing these control structures. The second mechanism, for identifying operands, is one of the major contributions of the paper. An example will be useful to motivate the necessary ideas.

Consider a generic class called "car" which has the *attributes* "identification number", "manufacturer", and "color" and which has the associated *operations* "purchase", "sell", and "lease". To apply one of these operations to an individual car, it is necessary to uniquely identify that individual. There are two basic ways in which this can be done. One way appeals to the car's unique attributes, and the other way appeals to operations previously applied to the car. Let's assume that the attribute set {identification number, manufacturer} is sufficient to identify individual cars. A car may then be named as "the *Ford* numbered *437*" or "the *Chevrolet* numbered *623*". We will call this *associative*

1. Introduction

When users interact with a complex, real-world system they give names to abstractions of the system which they can conceive as a whole. Abstractions of the system *state* are conceived as abstract *objects*, while abstractions of state *transformations* are conceived as abstract *operations*. In a computer model of the system, it is important that these abstractions be rigorously specified without introducing extraneous representational or algorithmic detail. Such specifications can be given in two complementary ways. A *structural* specification defines an abstract object as a composition of other abstract objects. A *behavioral* specification defines an abstract object in terms of its associated abstract operations.

In [10, 11] we developed a methodology for the structural specification of abstract objects. We pointed out that *aggregation* (naming relationships) and *generalization* (naming classes) are two fundamental mechanisms for composing an abstract object from other abstract objects. Aggregation and generalization have the effect of embedding abstract objects into a framework of intersecting hierarchies. To define this framework we introduced the data types "aggregate" and "generic". These data types specify the abstract structure of objects without introducing extraneous detail. A generic object is a class of individual aggregate objects. Individuals may be included in, removed from, or changed in a generic by the primitive operations "insert", "delete", and "modify".

In this paper we will develop behavioral specifications which can exploit the previous structural ones. We will show how the two kinds of specification may be dovetailed into a combination called *integrated specifications*. Structural and behavioral specifications each make distinct contributions to the overall characterization of abstract objects. Structural specifications are essential to define the decomposition of complex objects

naming. On the other hand, using information about the order of previous operations a car can be named as "the last car *leased*" or "the second car *bought*". We will call this *operative* naming.

Let's assume that an individual car only appears in the class "car" after it has been bought. In principle then, any individual car can be named as "the nth car *bought*" or "the mth before last car *bought*", for some numbers n and m. However, human beings only find it comfortable to work with "small" values of n and m. Operative names are only convenient when the individual car in question is "the first car *bought*", "the last car *bought*", "the car *bought* after N" or "the car *bought* before N" where N is the known name of another car. When an individual cannot be named operatively in this way, human beings usually switch to associative naming.

We want our "aggregate" and "generic" types to support both associative and operative naming. This implies that a capability for associative search and a capability to retain the temporal order of relevant operator invocations must be built in. These types already support associative naming via the declaration of an "attribute key". An attribute key is a set of attributes whose values are sufficient to uniquely determine individuals of an aggregate type. Individuals with duplicate key values are automatically disallowed. Associative search is provided with respect to the key attributes.

The "aggregate" and "generic" types do not presently support operative naming. Operative naming can certainly be "implemented" over these types by declaring additional attributes and using them to maintain temporal operation orderings. However, such implementations involve extraneous details and are neither convenient for the user nor efficient for

the implementor. We will therefore extend the generic type to support operative naming via the declaration of an "operation key". An operation key is a set of operations whose prior invocations are sufficient to uniquely determine individuals in a generic class. The temporal ordering for invocations of these key operations is automatically retained.

Interestingly, if a generic class has an operation key it is not necessary for it to have an attribute key. For example, if we remove the attribute "identification number" from the "car" generic, the remaining attributes do not form a key - two distinct cars may have the same "manufacturer" and "color". Nevertheless, individual cars can still be distinguished via operative names. Operative naming thus provides a semantic basis for the handling of "identical" representations. Attribute keys are therefore optional when operation keys are declared.

Previously, attribute keys were used for referencing between abstractions in an aggregation/generalization hierarchy. This required copying the attribute key from an individual into its relationships, and also retaining (essentially) the same key at all levels of generalization. Since attribute keys need not exist with the extended types, a new approach to referencing is required. Our approach is to provide the user with two *indirect naming* primitives - one for aggregate components and the other for generic images - and to render their implementation invisible to the user. This means that attribute keys are not copied into relationships and that attribute keys can vary with the level of generalization. The sole use of attribute keys is for associative naming.

Once the aggregate and generic types have been extended in these ways, they can be used to give integrated specifications for abstract systems.

These integrated specifications are at a higher level than the purely behavioral specifications used in programming languages [5, 6, 9]. As a demonstration we will specify the abstractions "matrix" (which uses associative naming), "stack", and "queue" (which, in the absence of attribute keys, require operative naming), "symboltable" (which requires both operative and associative naming) and "expression" (which requires operative naming). These five specifications are notable for their simplicity and clarity. This is not merely the result of excluding implementation detail. Rather, it results from the explicit support of aggregation and generalization abstractions together with suitable naming mechanisms. This allows the common aspects of specifications to be suppressed and only salient differences to be emphasized. Further, without constraining the implementor, these specifications delineate the capabilities which must be provided by lower levels of implementation.

In the database area, specification languages are called "database models". Research into database models has been preoccupied with structural specifications. This reflects the fact that in many database applications the operations which transform the database are quite simple. The importance of aggregation and associative naming is well recognized. Integrated specifications can be thought of as a new database model which captures both the structural and behavioral characteristics of abstract systems. Since it can handle "identical representations", this model can specify the structure of a greater variety of abstractions than the relational model [1]. Moreover, its capability for behavioral specifications is important for databases in applications, such as simulation and control, where dynamic characteristics are critical.

The ideas of several people have been particularly influential in the

present work. Codd [2] was first to show that, when an appropriate naming mechanism for individuals is used, *structure* can be separated from *representation*. His "normalized relations" were the first form of structural specification which supported aggregation abstractions. McCarthy's "abstract syntax" [8] is closely related to our structural specifications. In essence, our structural specifications extend his composition operations of "cartesian product" and "disjoint union" by incorporating suitable naming mechanisms for individuals. Liskov [7] has shown how behavioral specifications can be exploited in the design of a programming language. Guttag [4] has shown how behavioral specifications can be set on an elegant algebraic basis.

Section 2 discusses operative naming and introduces the extended aggregate and generic types. Section 3 considers the specification and maintenance of aggregation hierarchies with the new types. Section 4 covers similar topics for generalization hierarchies. Section 5 demonstrates the advantages of integrated specifications for a variety of data abstractions drawn from the programming language area. Section 6 is a conclusion.

2. Operative Naming

An abstract object type has two external manifestations: i) its set of attributes and ii) its set of operations. A user may need to employ either, or both, of these manifestations to identify an individual of the type. In a "static" environment, such as those used in data processing applications, it is usually the individual's attributes that are important - *associative* naming is therefore common. In a "dynamic" environment, such as those used for simulation and control applications, it is often the history of operations applied to the individuals that is important - *operative* naming is then useful.

The "generic" type [11] already supports associative naming. For example, suppose we declare:

<u>var</u> G: <u>generic</u> <u>of</u> A;

where A is an aggregate type with attribute key k. An insertion into G will not be allowed if it would result in two individuals with the same key values. Furthermore, an individual in G may be named by writing G[k-list], where "k-list" is a list of the individual's key values. For example, assuming the key attributes of the generic "car" are {identification number, manufacturer}, an individual car can be named as "car[437, Ford]".

Operative naming exploits the order in which operations were previously applied to name the individuals of a generic class. As described in section 1, human beings only seem to be comfortable in using operation orders on a "local" basis. In the simplest case, an individual may be named as the first or last individual to which an operation was applied. For example, we may refer to "the first car sold" or "the last car bought". More generally, if N is an individual name, we may talk about the individual operated on before or after N. For example, we may refer to "the car sold before N" or "the car bought after N".

Operative naming may be combined with associative naming so that both operations and attributes serve to identify the desired individual. For example, we may talk about "the last blue car sold" or "the Ford bought after N". More complex forms of *combined* naming, which mix multiple operations, are also possible. For example, we may refer to "the first blue car sold after the last Ford was bought".

Operations such as "buy", "sell", or "lease" are ultimately expressed in terms of the three primitive operations associated with the generic type - namely, "insert", "delete", and "modify". In some cases, these operations map directly to single primitive operations - for example, "buy" may correspond to a single "insert" operation, and "lease" may correspond to a single "modify" operation. In other cases, they will map to several primitive operations - for example, "sell" may correspond to a "modify" operation on "car" together with an "insert" operation on "sales".

We will say that an operation is *associated* with a generic G if that operation is ultimately expressed in terms of primitive operations on G alone. We will say that an operation is *basic* to a generic G, if that operation is associated with G and is expressed as a *single* primitive operation. Since basic operations are necessarily unambiguous with respect to the individual involved, only these operations will be exploited for operative naming. Furthermore, we will assume that an individual is of no further interest once it has been deleted. There is therefore no reason to name an individual as (effectively) "the last individual deleted".

We have decided to provide the following capabilities for operative naming. If "op" is a basic insert or modify operation for a generic G, an individual in G may be named by:

G[<u>first</u>/<u>last</u> oped]

here the slash "/" indicates an alternative and "oped" is the past participle form of "op". Further, if N is an individual name, either associative or operative, then a new name may be formed by writing:

succ/pred oped N.

In this way, we can name the successor (predecessor) "oped" before (after) the individual N.

Some examples of operative names for the generic "car" are:

car[last bought]
car[first leased]
succ bought car[last leased]
pred leased car[437, Ford]

In the third example, notice that two different operations are included in one operative name. The last example may be read as "the car leased before the Ford with identification number 437". This name presumes that the "Ford with identification number 437" was itself leased.

We are assuming that these naming primitives are embedded in some high-level programming language which provides appropriate control structures. It is then possible to implement more powerful operative and combined naming facilities using the primitives provided. It could be argued that the primitives themselves should be more powerful - for example, by providing the capability to directly state:

G[first/last {but n} {where property} oped]

where the curly brackets indicate optional phrases. This construction has two nested iterations built-in - one to check for satisfaction of a property (color = blue) and the other to find the nth such individual. Using our primitives these iterations would have to be expressed explicitly. However, the line must ultimately be drawn somewhere and it seems reasonable to begin with simple primitives.

Let's consider the application of basic operations on a given generic more carefully. A given individual can be inserted, and then deleted, many times. However, for each individual we will only be interested in the *last* time a given basic insert operation was applied to it, without subsequent deletion. Similarly, an individual can be modified several times, possibly with intervening insertions and deletions. However, for each individual we will again only be interested in the *last* time a given basic modify operation was applied to it, without subsequent deletion. As stated before, we will not be interested in deletion operations for operative naming.

We can now outline the implementation requirements for operative naming over a generic G. For each basic insert or modify operation op, a separate total ordering over (some of) the individuals of G must be maintained. This ordering specifies the sequence in which op was *last* applied to each individual. For example, assume op was applied to three individuals in G in the sequence first g_1, then g_2, and finally g_3. At this time G[last oped] names g_3. If op is applied to g_2 again, then the sequence becomes first g_1, then g_3, and finally g_2. Now G[last oped] names g_2. If op is applied concurrently to two different individuals then an arbitrary sequence must be assigned. Whenever an individual is deleted, it must be removed from all orderings.

It is clear that there is a substantial overhead for maintaining operative names - as indeed there is for maintaining associative names. In some applications there may be no need for one of these naming mechanisms. Under such circumstances, the unnecessary mechanism should be "turned off" at the implementation level.* It is important to provide both naming

---

* The type declaration provides the "switch". If no attribute key is declared, associative naming need not be supported. If no operation key is declared, operative naming need not be supported.

mechanisms with the generic type because they support distinct user requirements. The user can express himself simply and naturally, and the implementor can optimize lower levels in accordance with intended usage.

So far we have discussed general issues concerned with operative naming. Now we will consider some specific examples and introduce several syntactic constructs. Let's assume we want to define a generic object call "employee". The attributes of "employee" which interest us are "ID-number", "name", "address", and "salary". From these attributes, "ID-number" alone forms a key. We will also assume that the only operations on "employee" of interest are "HIRE", "TERMINATE", and "PROMOTE".* We can declare the structural characteristics of "employee" as:

```
var employee: generic
              of
              aggregate [ID#]
                    ID#: ID-number;
                      N: name;
                      A: address;
                      S: salary
              end
```

The square brackets following aggregate include the key. We will now discuss the primitive operations on generics, with the objective of ultimately defining HIRE, TERMINATE, and PROMOTE.

To *construct* an individual of a given generic G, we will use the following syntax:

G<component list>.

The angle brackets indicate a construction operation and enclose the components involved. For example, to construct an individual employee we may write:

employee<E23, Brown, NYC, 10>.

---

* It is assumed that a query language is available for information retrieval and so no explicit *retrieval* operations are specified. We will only specify retrieval operations for an abstract object when certain kinds of *restricted* access are a characteristic of that object - e.g. TOP will be specified for stack.

To *insert* an individual I into a generic G, we will use the syntax:

G ⇐ I.

G and I must be compatible types. For example, to insert the previous individual into "employee" we write:

employee ⇐ employee<E23, Brown, NYC, 10>.

As we often want to construct an individual and immediately insert it into its generic, we will shorten the above statement by not repeating "employee":

employee ⇐ <E23, Brown, NYC, 10>.

The type for the constructor is given by the generic on the left.

We can now define HIRE. Essentially HIRE constructs an individual and inserts it into "employee":

HIRE(id, n, a, s): employee ⇐ <id, n, a, s>.

HIRE is therefore a basic insert operation of employee. We will now hire three employees:

HIRE(E23, Brown, NYC, 10);
HIRE(E16, Smith, LA, 12);
HIRE(E4, Brown, SF, 10).

At this time, we can visualize "employee" as the following table:

employee:

ID-number	name	address	salary
E23	Brown	NYC	10
E16	Smith	LA	12
E4	Brown	SF	10

The individual with ID-number "E23" can be named associatively as employee[E23] and named operatively as employee[first hired].

To *delete* an individual I from a generic G we use the syntax:

G ⇒ I.

For example, we can delete the individual "E23" by writing:

employee ⇒ employee[E23].

We can now define TERMINATE as:

TERMINATE(e): employee => e.

In this definition "e" is the name of an employee individual. For example, we can terminate employee "E23" by writing:

TERMINATE(employee[E23])

or: TERMINATE(employee[first hired]).

After "E23" is terminated, employee[first hired] names "E16".

To *modify* an individual I so that its c-component becomes v, we use the syntax:

$I\nabla c \Leftrightarrow v.$

The symbol "∇" forms a "component name" from individual I and selector c.*
For example, to modify the address of "E4" to "SLC" we write:

employee[E4]∇A <=> SLC.

We will assume that an employee's promotion affects only his salary. PROMOTE can then be defined as:

PROMOTE(e, s): e∇S <=> s.

In this definition, e is a name (either associative or operative) for an employee and s is a salary. Let's promote employee "E16" to a salary of $14K:

PROMOTE(employee[E16], 14).

At this time "E16" can be named as either employee[first promoted] or as employee[last promoted]. If "E16" is terminated, both these names become undefined, and "E4" acquires the names employee[first hired] and also employee[last hired]. It is important to keep in mind that an individual's associative name can only change when its key values are modified. However, an individual's operative names may change whenever an operation is applied to an individual in the same generic.

We will now consider the "employee" example again - with the differ-

---

* The reason for not using the familiar "." notation for forming component names will become apparent in the next section.

ence that the only attributes of interest are "name" and "salary". Since individual employees cannot be distinguished by these two attributes alone, the generic "employee" will have no attribute key. The integrated specification of "employee" is now:

```
var employee:  generic
               of
               aggregate
                   N: name;
                   S: salary
               end

operations [HIRE, PROMOTE]
    HIRE(n, s):  employee ⇐ <n, s>
    TERMINATE(e):  employee ⇒ e
    PROMOTE(e, s):  e∇S ⇔ s
end
```

As there is no attribute key, the square brackets following aggregate are omitted. If an erroneous key (e.g. [N, S]) were declared, the generic would violate the requirements for being "well-defined" [10,11]. The square brackets following operations contain the names of the key operations. These are basic insert or modify operations that may be used for naming. Let's hire the same three individuals as before:

```
HIRE(Brown, 10),
HIRE(Smith, 12),
HIRE(Brown, 10).
```

At this time, "employee" can be visualized as the following table:

employee:

name	salary
Brown	10
Smith	12
Brown	10

In this table the representations of two distinct individuals have identical attribute values. These individuals cannot be distinguished by any kind of associative naming. Nevertheless, the individuals can be distinguished by operative naming - one is employee[first hired] and the

other is employee[last hired]. This distinguishability is of critical importance. It will allow one individual to be terminated or promoted without affecting the other. It will be exploited in the next section to allow one individual to participate in different relationships to the other. In short, operative naming provides a semantic basis for the treatment of identical representations.*

When a generic has a declared attribute key, duplicate representations *of the same individual* will be detected and disallowed. However, when a generic has no declared attribute key, this is no longer possible. Such generics should only be used in an environment where the responsibility for preventing duplicate representations is assumed prior to insertion. Such environments seem to occur quite frequently in practice.

We will now consider a third, and final, version of the "employee" example. We will assume that there is no interest in any attributes of "employee" at all. This situation is not unrealistic. It will occur whenever it is the relationships in which an individual participates, and not the details of the individual itself, that are of interest.+ Under these circumstances, we would define "employee" as:

```
var employee:  generic
               of
               aggregate
               end

operations [HIRE, PROMOTE]
    HIRE:  employee <= < >
    TERMINATE(e):  employee => e
    PROMOTE(e): _
end
```

*Since Codd's relational algebra [3] only supports an associative type of naming, all identical representations but one are deleted whenever they are generated by relational operations (e.g. PROJECT or UNION). Semantically, this is not always appropriate.

+Interestingly, this situation occurs in the "symboltable" example of Section 5.

The operation HIRE is implemented by inserting a null aggregate, while PROMOTE is implemented by a "no-op"(_). To hire the same three individuals as before we write:

HIRE;
HIRE;
HIRE.

To promote the first employee hired, we write:

PROMOTE(employee[first hired])

and to terminate the second employee hired, we write:

TERMINATE(succ hired employee[first hired]).

At this time, employee[first promoted] refers to the first employee hired. In essence the last definition of "employee" is providing just a naming capability. This capability is useful when employee relationships are being created and manipulated.

There is a fundamental duality between attributes (nouns) and operations (verbs). For example, one can think of the attribute "name" as the operation "give name to", and the operation "promote" as the the attribute "promotion". One can therefore design a specification language for data types based solely on "attributes" or based solely on "operations". However, within a given context, people find it natural to use the noun form for some concepts and the verb form for others. It is therefore somewhat artificial to insist that all concepts be treated in the noun form or that all concepts be treated in the verb form. Our approach allows the most natural form for a concept to be chosen in each case.

3. Aggregation Hierarchies

Aggregation is used here to mean an abstraction in which a relationship between generic objects (say $O_1,\ldots,O_n$) is regarded as a higher-level generic object (say O). The objects O_i are the *components* of O, and the *names* of these objects are the *attributes* of O. A component O_i may be either *primitive* or *non-primitive*. A non-primitive component is formed by another aggregation abstraction. Primitive objects are *self-naming*, while non-primitive objects may be named *associatively* and/or *operatively*. Aggregation gives rise to a hierarchy of generic objects.

As an example, consider the relationship in which "an instructor is scheduled to teach a course during a particular semester". This relationship may be aggregated as the generic object "class". The components of a class are its instructor, its semester of offering and the course taught. We will assume "semester" is a primitive object and may be named directly as "Fall77" or "Wint78". We will take "instructor" and "course" as non-primitive objects and so they must be named either associatively or operatively. The attributes of job will therefore be "instructor name", "course name", and "semester". To define the structure of "class" we will use the syntax:

```
var class: generic
           of
           aggregate [C, S]
                I: instructor name;
                C: course name;
                S: semester
           end
```

By declaring [C, S] as the attribute key, we are assuming a class can be uniquely identified by its course and semester components.

Let's consider the construction and manipulation of individuals belonging to a non-primitive generic. When an individual is constructed, its components

are named explicitly - these names may be associative or operative. The component names become the attributes of the individual. Even though the component names may change over time, the components of an individual remain the same, unless they are explicitly modified. The attributes of the individual change in accordance with the component names.

It is often necessary to manipulate the components of some individual. For this purpose we will provide an *indirect* naming mechanism for components. If "I" is an individual and "c" is a selector, then "I∇c" names the c-component of I.* It is important to realize that "I∇c" names the component itself and not the attribute. At any point in time, the *value* of the attribute "I.c" is a name for the component "I∇c". Since a component may have many names (associative, operative and now indirect), it is useful to know when two names apply to the same individual. We will therefore provide a predicate for name equivalence. For individual names n and m, "m ≡ n" is true provided n and m name the same individual.

Indirect naming allows us to access a component without knowing any of its "direct" names. The direct names of components become of interest when attributes of an individual are copied into local program variables or used for output. At that time a decision must be made as to which form of direct name to transfer. We will assume that only the associative form need be transfered. From the examples we have seen, this assumption is realistic. Attributes will therefore be copied in the form of a standardized associative name. Attributes names (e.g. I.c) will be legal only in the scope of an assignment or output statement. At all other times indirect names (e.g. I∇c) must be used.

---

*The symbol "∇" is chosen to suggest an arrow pointing *down* the aggregation hierarchy. It may be read as "A-arrow".

With the preceding discussion as background, we will now give a specific example of the definition, construction and manipulation of an aggregation hierarchy. Figure 1 is a schematic form of an aggregation hierarchy involving the abstract objects "class", "instructor", and "course", together with their associated operations. Instructors can be hired, promoted and terminated. Promotion involves a modification of rank. Courses can be created and dropped. Classes can be scheduled, changed and cancelled. However, the only allowed change is to assign a different (or no) instructor to a class. A specification for this aggregation hierarchy is given in Figure 2.

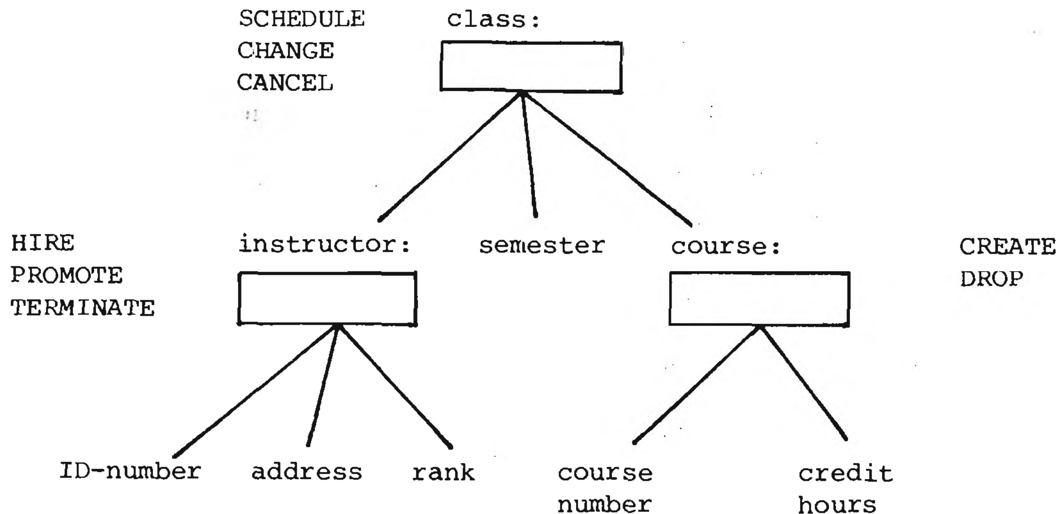

Figure 1: An aggregation hierarchy for "class".

The specifications for "instructor" and "course" are very similar to the specifications of "employee" given previously. A new feature is statements enclosed by curly brackets in the definitions of TERMINATE and DROP. This feature will be discussed later. Before we can construct and manipulate classes, some instructors must be hired and some courses created. Let's do this now:

```
HIRE(I6, 3rd Ave, assnt);
HIRE(I21, 4th Ave., full);
CREATE(MA434, 3);
CREATE(CS231, 4);
CREATE(EE316, 2).
```

```
var class: generic
           of
           aggregate [C, S]
                I: instructor name;
                C: course name;
                S: semester
           end

operations [SCHEDULE, CHANGE]
    SCHEDULE(i, c, s):  class ⇐ <i, c, s>
    CHANGE(cl, i): cl∇I ⇔ i
    CANCEL(cl):  class ⇒ cl
end


var instructor: generic
                of
                aggregate [ID]
                     ID: ID-number;
                      A: address;
                      R: rank
                end

operations [HIRE, PROMOTE]
    HIRE(id, a, r):  instructor ⇐ <id, a, r>
    PROMOTE(i, r):  i∇R ⇔ r
    TERMINATE(i):  instructor ⇒ i; {CHANGE(class)}
end


var course: generic
            of
            aggregate [CN]
                 CN: course-number;
                  H: credit-hours
            end

operations [CREATE]
    CREATE(cn, h):  course ⇐ <cn, h>
    DROP(c):  course ⇒ c; {CANCEL(class)}
```

Figure 2: A specification for the aggregation hierarchy of Figure 1.

The operation SCHEDULE, which requires the construction and insertion of an individual class, can be specified as:

SCHEDULE(i, c, s): class ⇐ <i, c, s>

where i, c and s are an "instructor" name, a "course" name and a semester respectively. The effect of scheduling a class "cl" is that: $cl\nabla I \equiv i$, $cl\nabla C \equiv c$ and $cl\nabla S \equiv s$. Let's schedule the following two classes:

SCHEDULE(instructor[I6], course[last created], FALL78);
SCHEDULE(instructor[last hired], course[CS231], SPRG78).

In these operations, both associative and operative names have been employed for components. Under some circumstances, it may even be useful to employ *indirect* names for components. As an example, we will schedule a third class:

SCHEDULE(class[last scheduled]∇I, course[MA434], WINT78).

In this operation the "instructor" component has been named indirectly via a class previously scheduled.

Now that we have some classes scheduled, let's consider the formation of names for individual classes. Operative names are formed in the usual way (e.g. class[first scheduled]). To form an associative name for a class, values must be given for the C and S attributes. The value of the S attribute is a semester (e.g. SPRG78). The value of the C attribute is the name of a course. This latter name may itself be either associative (e.g. course[CS231]) or operative (e.g. course[last created]). An associative name for a class may thus appear as:

class[course[CS231], SPRG78]

or: class[course[last created], SPRG78]

These associative names can be simplified, without ambiguity, by omitting the generic name "course". Assuming key attributes are listed

in left to right order, this generic name is uniquely determined by the key declaration for "class". The above names would then appear as:

class[[CS231], SPRG78]

and: class[[last created], SPRG78].

Subsequently, we will always omit the generic name from the individual associative and operative names of key components. We will say that a name is *fully associative*, if associative naming is used for key components at all levels. For example, the name "class[[CS231], SPRG78]" is fully associative, while the name "class[[last created], SPRG78]" is not fully associative.

Fully associative names are the standard form used in copying attributes for external usage or output. For example, to output the "instructor" attribute of the last class scheduled, we write:

output class[last scheduled].I

The fully associative form of this attribute is "instructor[I21]" and so this form is output. If an attribute has no fully associative form then copying is illegal. A generic object can be visualized in terms of fully associative names at any point in time. At the present time "class" can be visualized as the table below:

class:

instructor name	course name	semester
I6	EE316	FALL78
I21	CS231	SPRG78
I21	MA434	WINT78

It should be kept in mind that the fully associative name of an individual, like all its names, may change due to subsequent operations. It is always

the *current* name that is output.*

To determine whether the last class scheduled was class[[CS231], SPRG78] we can write:

class[<u>last</u> scheduled] ≡ class[[CS231], SPRG78].

Name equivalence is particularly useful in testing the properties of components. For example, to determine whether instructor "I6" is scheduled to teach EE316 in FALL78, we can write:

class[[EE316], FALL78]∇I ≡ instructor[I6].

Similarly, we can determine whether the first class scheduled is being offered in FALL78:

class[<u>first</u> scheduled]∇S ≡ FALL78.

Notice that the "∇" symbol and not the "." symbol are used to form component names.

The operation CHANGE can be defined as:

CHANGE(cl, i): cl∇I ⇔ i

where "cl" and "i" are names for a class and an instructor respectively. The effect of this operation is that cl∇I ≡ i. To assign instructor "I6" to the class of MA434 given in WINT78, we write:

CHANGE(class[[MA434], WINT78], instructor[I6]).

To simply release an instructor from an assignment to the previous class, without assigning a new instructor, we can write:

CHANGE(class[[MA434], WINT78], -).

The symbol "-" stands for the "null" name.

The operation CANCEL can be defined as:

CANCEL(cl): class ⇒ cl

---

*It is easy to maintain this name "currency" at the implementation level. Non-primitive component names are never actually stored in the individual's internal representation. Instead, pointers are maintained to the individual's components and names are dynamically interpreted before output.

where "cl" is the name of a class. This operation simply deletes the individual "cl" from class. All associative and indirect names equivalent to "cl" will no longer name an individual and will thus become undefined. All operative names equivalent to "cl" will either be transferred to other individuals or also become undefined. To cancel the class previously changed, we write:

CANCEL(class[last changed]).

So far in this section, we have seen how aggregation hierarchies may be defined and manipulated. Now we will consider the aggregation invariants that must be maintained. There are two aggregation invariants for a generic G:

i) if G has an attribute key, then no two individuals in G have the same key components, and

ii) for each individual in G, all its non-null attributes are *defined* names.

The first invariant must be maintained during insert and modify operations. In particular, "null" names must not be allowed in key components. The second invariant can be maintained under insert and modify operations by disallowing undefined names. The most interesting case is the maintenance of the second invariant under delete operations.

When an individual is deleted, attributes of higher-level individuals which name this individual will become undefined. There are two methods for handling this situation. One method is to *delete* all these higher-level individuals, and the other is to *modify* them so that the offending attribute becomes null. The more appropriate method depends on the semantics of the higher-level individual. If the undefined attribute is "essential" to the continued existence of the higher-level individual,

then that individual should be deleted. For example, if the course offered in a class is dropped, then that class is normally deemed to be cancelled. On the other hand, if the undefined attribute is "inessential", then the appropriate action is to modify the higher-level individual. For example, if one terminates an instructor who teaches the class, this does not necessarily imply cancellation of the class. In any case, null names cannot be used for key attributes.

In specifying a generic object, we must define which of these two side-effects on higher-level objects are triggered by a delete operation. This is done by enclosing in curly brackets a modify or delete operation for each higher-level generic. In Figure 2, we have decided that terminating an instructor will only cause his classes to be modified by the CHANGE operation. It is not necessary to define parameters for this operation - all individuals with an undefined attribute are to be modified using the "null" name. We have also decided that dropping a course will cause all its classes to be deleted with the CANCEL operation.

The triggered invocation of defined operations will affect operative names on that operation in just the same way as direct invocation. For example, if we terminate an instructor who is scheduled to teach some classes, then immediately afterwards "class[<u>last</u> changed]" will name one of these classes. In some cases there may be no defined operation which is appropriate to use for the triggered side-effect. A primitive delete or modify must then be specified.

4. Generalization Hierarchies

Generalization is used here to mean an abstraction in which a class of generic objects (say, $\{O_1, \ldots, O_n\}$) is regarded as a higher-level generic object (say, O). The objects O_i are the *categories* of O. The class $\{O_1, \ldots, O_n\}$ is partitioned into *blocks**, so that each block contains disjoint categories. Each category O_i may be either *primitive* or *non-primitive*. A non-primitive category is formed by another generalization abstraction. Generalization gives rise to a hierarchy of generic objects.

As an example, consider the class which contains the generic objects "wind-propelled vehicle", "motorized vehicle", "man-powered vehicle", "land vehicle" and "air vehicle". Each of these objects is a particular category of "vehicle", and so the whole class can be generalized to "vehicle". There are two blocks in the class:

i) {wind-propelled vehicle, motorized vehicle, man-powered vehicle}

ii) {land vehicle, air vehicle}.

The first block contains alternative "propulsion" categories and the second contains alternative "medium" categories.

A structural definition of "vehicle" is shown in Figure 3. The only change to the syntax of [11] is that the category type (e.g. propulsion category), rather than the category selector (e.g. PC), is repeated in a block declaration. This change emphasizes that the category type is completely determined by the block. Each category must also be defined as a generic object in its own right. The attributes PC and MC are called *category* attributes.

A single real-world individual may have representations in several categories. For example, a car will appear as a "vehicle" individual,

---

*In [11] blocks were called clusters.

```
var vehicle:
  generic
    propulsion category =
       (wind-propelled vehicle, motorized vehicle, man-powered vehicle);
    medium category =
       (land vehicle, air vehicle)
  of
  aggregate [ID]
    ID: ident. number;
     A: appraised value;
    PC: propulsion category;
    MC: medium category
  end
```

Figure 3: A definition of "vehicle".

a "motorized vehicle" individual and also as a "land vehicle" individual. We will say that these different representations of the same real-world individual are *images* of each other. When two images have attributes in common, their values must be the same. Since categories are disjoint within blocks, an individual can have an image in at most one category of a block. For example, a "vehicle" individual can have an image in one of the propulsion categories and in one of the medium categories.

When an individual is inserted into one category, it is important that images be "simultaneously" inserted into all other applicable categories. Similarly, when an individual is deleted from a category, images must be "simultaneously" deleted from all applicable categories. Finally, when an individual is modified, its images must be modified "simultaneously" in accordance. This simultaneous activity is provided by operations automatically *triggered* by the original operation. An individual must not be inserted into a category more than once by either a direct or triggered operation. If all categories have declared keys then such duplicate insertions can be prevented, otherwise no guarantees can be made.

We will allow categories to have whatever attributes are necessary for the application on hand. We will not insist that attributes be inherited

from supercategories as this may force the inclusion of extraneous detail. We will allow category attributes (e.g. PC and MC in Figure 2) to take on "null" names. This may mean an individual does not belong in any category, or that the category has not yet been assigned. For example, if we insert a motorboat into "vehicle", its PC attribute will take the value "motorized vehicle", while its MC attribute will take on the "null" name. Only those categories relevant to the application on hand need to be declared as generic objects. Again this does not force the inclusion of extraneous detail. As a result, the union of the subcategories of a category may be a *proper* subset of the category itself.

In [11], where all generics had attribute keys, we insisted (with some minor exceptions) that all subcategories have the same key as their parent category. This allowed the images of an individual to be identified by using associative naming. Now we need a new method for naming images. We will therefore introduce a second indirect naming operator "↓".* If "n" is the name of an individual and "S" is the selector for a category attribute, then "n↓S" is the image of "n" in the category "n∇S". For example, if "v" is an individual vehicle, and v∇PC ≡ "motorized vehicle" then v↓PC is the image of v in "motorized vehicle". Similarly, if v∇MC ≡ "air vehicle" then v↓MC is the image of v in "air vehicle". If n∇S ≡ "null" then n↓S is undefined. Since the category in which an image appears is given by the corresponding category attribute, indirect image naming is usually done inside a "case statement" - for example:

```
case v∇MC of
     air vehicle: with v↓MC do S_a;
    land vehicle: with v↓MC do S_l;
            null: S_n
end
```

---

*The symbol "↓" is chosen to suggest an arrow pointing <u>down</u> the generalization hierarchy. It may be read as "G-arrow".

When an individual is inserted or modified, its *aggregation components* are specified *explicitly* as described in Section 3. In contrast, when an individual is inserted or modifed, its *generalization images* are determined *implicitly* by triggered operations. To understand how this can be done, we must consider the invariants maintained by the triggers. There are three generalization invariants for a generic G.

Let S be a selector of G for categories in the block $\{G_1, \ldots, G_n\}$, then

i) for each individual r_i in G_i, there is an individual r in G such that $r{\downarrow}S \equiv r_i$.

ii) for each individual r in G, if $r\nabla S \equiv G_i$ then there is an individual r_i in G_i such that $r{\downarrow}S \equiv r_i$, otherwise $r\nabla S \equiv$ null and $r{\downarrow}S$ is undefined.

iii) for each individual r in G, if there is an individual r_i in G_i such that $r{\downarrow}S \equiv r_i$ then $r\nabla S \equiv G_i$ and $r\nabla X \equiv r_i\nabla X$ for all common selectors X.

The first invariant ensures G_i is a subset of G. The second invariant ensures that individuals in G, which are categorized as G_i, have an image in G_i. The third invariant ensures that images are consistent with each other. Now we will consider how these invariants can be maintained for insert, modify and delete operations.

Assume an individual r is inserted into G. If there is a supercategory of G, say G', then an image r' will be inserted by a trigger into G' and $r'{\downarrow}S'$ (for the appropriate selector S') will be made equivalent to r. Similarly, if $r\nabla S \equiv G_i$ then an image r_i will be inserted by a trigger into G_i and $r{\downarrow}G$ will be made equivalent to r_i. If $r\nabla S \equiv$ null then no image will be inserted. These triggered insertions may propagate up and/or down the generalization hierarchy always moving away from G. Null names are used as values for attributes that do not also occur in r.

Assume an individual r in G is modified. If no category attribute is changed, then triggers will simply modify the superimages and subimages of r accordingly. However, if a category attribute S of r is modified from G_i to G_j, then the image of r in G_i is deleted and a new image (say r_j) is inserted into G_j. The name $r \downarrow S$ is then made equivalent to r_j. These operations are performed automatically by triggers.

Assume an individual r in G is deleted. Triggered delete operations will then remove all its subimages. There are two ways to handle a superimage (say r') of r. One way is to delete it. The other way is to modify r' so that the relevant category attribute (say S') takes on the "null" name. In this case $r' \downarrow S'$ will become undefined. The appropriate way of handling superimages is determined by semantic considerations and can be selected when the generic G is specified.

We will now consider an example of how indirect image naming is supported by triggers. Figure 4 shows several individuals in the "vehicle" generalization hierarchy. We will assume that the tables are initially empty, and show how the individuals are inserted. Since we have not yet discussed operative naming over generalization hierarchies, we must rely on associative naming. To this end, ID will be taken as an attribute key for each generic.

The first insertion is:

vehicle ⇐ <I6, 347, motorized vehicle, air vehicle>.

This insertion creates the top individual in "vehicle" and, via triggers, the individuals in "motorized vehicle" and "air vehicle". These latter two individuals contain "null" names since neither the "fuel capacity" nor the "wingspan" is determined by the original insertion. As a result of the triggered insertions, the following equivalences hold:

vehicle[I6]↓PC ≡ motorized vehicle[I6]
vehicle[I6]↓MC ≡ air vehicle[I6]

vehicle:

identification number (ID)	appraised value (A)	propulsion category (PC)	medium category (MC)
I6	347	motorized vehicle	air vehicle
I42	63	wind-propelled vehicle	-
I76	-	-	land vehicle
I28	-	man-powered vehicle	-

man-powered vehicle:

identification number (ID)	muscle group (M)
I28	leg

air vehicle:

identification number (ID)	wingspan (W)
I6	-

motorized vehicle:

identification number (ID)	fuel capacity (F)
I6	-

land vehicle:

identification number (ID)	surface type (S)
I76	rail

wind-propelled vehicle:

identification number (ID)	sail area (SA)
I42	-

Figure 4: Individuals in the "vehicle" generalization hierarchy.

The wingspan for "air vehicle[I6]" can be entered by writing:

air vehicle[I6]∇W ⇔ 95.

The second insertion is:

vehicle ⇐ <I42, 63, wind-propelled vehicle, ->.

This insertion creates the second individual in "vehicle", and via triggers, the individual in "wind-propelled vehicle". At this time, the following statements hold:

vehicle[I42]↓PC ≡ wind-propelled vehicle[I42]
vehicle[I42]↓MC is undefined.

The third insertion is:

land vehicle ⇐ <I76, rail>.

This insertion creates the individual in "land vehicle" and, via triggers, the third individual in "vehicle". Notice that this latter individual has the "null" name (-) for propulsion category but "land vehicle" for medium category. The medium category is determined by the original insertion. At this time, the following statements hold:

vehicle[I76]↓PC is undefined.
vehicle[I76]↓MC ≡ land vehicle[I76].

The final insertion is:

man-powered vehicle ⇐ <I28, leg>.

This insertion creates the individual in "man-powered vehicle" and, via triggers, the last individual in "vehicle". Suppose we now modify the last individual in "vehicle" by writing:

vehicle[I28]∇PC ⇔ motorized vehicle.

This modification will cause, via triggers, the deletion of the individual in "man-powered vehicle" and the insertion of a second individual in "motorized vehicle". At this time, the following equivalence holds:

vehicle[I28]↓PC ≡ motorized vehicle[I28].

So far we have seen how the triggered invocation of operations can maintain the generalization invariants and simultaneously support indirect image names. This means that the generics in a generalization hierarchy do not need to have the same attribute keys or even any attribute keys at all. Each generic can now be designed individually without the forced inclusion of certain key attributes. Since the basic triggering mechanism is always the same, it can be built in as a primitive rather than redefined for each generalization hierarchy.

For a particular generalization hierarchy, it is only necessary to specify *which* operations should be triggered. The *time* of triggering and the *parameters* to be used are completely determined by the generalization invariants as previously described. Since the triggered invocation of operations can affect operative naming, the operations to trigger must be decided carefully. Sometimes defined operations must be triggered and other times primitive operations must be triggered. We will give an example to illustrate the specification of triggered operations.

Let's consider a company which employs many *secretaries*. When a secretary is *hired*, she is *assigned* either as a *pool secretary* or as a *personal secretary*. Depending on this assignment, she will either *enter* the secretarial pool or be *engaged* as a personal secretary. A secretary may change from one assignment to the other several times. A secretary may *terminate* her employment at any time. Figure 5 is a generalization hierarchy for "secretary". The asterisks on the two "HIRE" operations indicate that they are never invoked directly - they are only triggered by the higher "HIRE" operation. They are named explicitly for the purposes of operative naming over "pool secretary" and "personal secretary".

A secretary has certain attributes independent of her present assignment (e.g. a social security number and a birthdate). However, a secretary

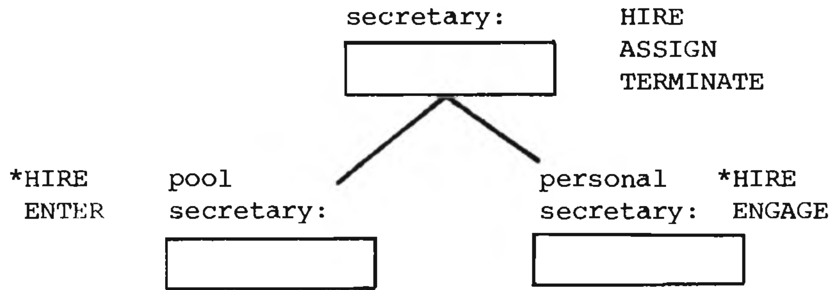

Figure 5: A generalization hierarchy for "secretary".

acquires additional attributes on entering the secretarial pool (e.g. a typing station) and different additional attributes on engagement as a personal secretary (e.g. a private office). We will assume the attribute "social security number", and the operations "HIRE", "ASSIGN", "ENTER" and "ENGAGE", are used to identify individual secretaries. Figure 6 is an integrated specification of "secretary".

When a secretary is hired, a HIRE operation is triggered on either "pool secretary" or "personal secretary" depending on the category assigned. If the category is "pool secretary" then the triggered HIRE will use "-" for the "typing station" attribute. When the secretary actually enters the secretarial pool, "typing station" is updated via the ENTER operation. Notice that an ENTER operation cannot be completed until the secretary is hired into the secretarial pool.

When a secretary is assigned to another category, her "image" must be removed from "pool secretary" and placed in "personal secretary" or vice versa. Since the secretary is not being hired at this time, it is not appropriate to trigger a HIRE operation. Instead a primitive "insert" operation should be used. Primitive "delete" operations are used for

```
var secretary:  generic
                  category = (pool secretary, personal secretary)
                of
                aggregate [N]
                  N: social security number;
                  B: birthdate;
                  C: category
                end

operations [HIRE, ASSIGN]
    HIRE(n, b, c):   secretary ⇐ <n, b, c>
                     {HIRE(pool secretary) / HIRE(personal secretary)}
    ASSIGN(s, c):   s∇C ⇔ c
                     {insert(pool secretary), delete(personal secretary)/
                      delete(pool secretary), insert(personal secretary)}
    TERMINATE(s):   secretary ⇒ s
                     {delete(pool secretary), delete(personal secretary)}
end


var pool secretary:  generic
                     of
                     aggregate [N]
                       N: social security number;
                       T: typing station
                     end

operations [HIRE, ENTER]
   *HIRE(n, t):  pool secretary ⇐ <n, t>
    ENTER(po, t):  po∇T ⇔ t
end


var personal secretary:  generic
                         of
                         aggregate [N]
                           N: social security number;
                           O: office
                         end

operations [HIRE, ENGAGE]
   *HIRE(n, o):  personal secretary ⇐ <n, o>
    ENGAGE(pe, o):  pe∇O ⇔ o
end
```

(s, po and pe are names for a secretary, a pool secretary and a personal secretary, respectively)

Figure 6: A specification for the generalization hierarchy of Figure 5.

removal from a category. If these activities were significant to the user, defined operations would have been declared. Triggered invocation of these operations could then be specified. Since we do not want to force the declaration of "extraneous" operations, we allow both primitive and defined operations to be triggered.

The HIRE operation on "pool secretary" is (by assumption) only triggered by the HIRE operation on "secretary". There is therefore no need to specify any triggered operations for HIRE on "pool secretary". If this operation could be invoked directly, then a triggered HIRE on "secretary" should be specified. The ENTER operation on "pool secretary" modifies an attribute which is unique to this category. There is therefore no need for a triggered modify operation on "secretary". For similar reasons, no triggered operations are specified for HIRE and ENGAGE on "personal secretary".

Now let's consider operative naming over the "secretary" example. Operative names over a generalization hierarchy are often quite ambiguous when stated in English. This is particularly true when individuals can dynamically change categories. The name "pool secretary[last entered]" refers to the pool secretary who last entered the pool - either by being hired into it or by being reassigned to it. The name "pool secretary [last hired]" refers to the pool secretary who was last hired *directly* into the pool. There may be other pool secretaries hired after this one, who entered the pool *following reassignment*. To name the secretary who was hired last of all those in the pool, it is necessary to write:

secretary[last, where C = pool secretary, hired]↓C.

In this name we used the syntax for combined naming mentioned in section 2.

5. Examples of Integrated Specifications

In previous sections we have given several examples of integrated specifications. However, these have been ad hoc examples designed to illustrate certain concepts or techniques. Now we will give some examples which can be used as benchmarks for comparing integrated specifications with other specification techniques. The examples are specifications of standard abstractions from the systems' programming domain - namely, "matrix", "stack", "queue", "symboltable" and "expression".

These abstractions differ from those of previous sections in that a given system may contain several *instances* of each abstraction. For example, a compiler may contain several stacks and an operating system may contain several queues. Each of these instances could in principle be specified separately. However, this would cause an undesirable repetition of detail. A better approach is to give a single *parameterized* specification of each abstraction. For each instance, it is only necessary to specify which parameters are to be used. We therefore introduce a parameterized form of integrated specification which will be called a *template*.

A template differs from a regular specification in only one respect - each generic identifier is a *parameter* rather than a *declared variable*. Figure 7 shows the template for the specification of Figure 6. Notice that the reserved work "var" (variable) has been replaced by "par" (parameter) before each generic identifier. Further, each generic identifier has been listed in the template heading. To specify "secretary" relative to the company IBM, we need only to write:

```
        var (IBMsec, IBMpoolsec, IBMpersec):
                (secretary, pool secretary, personal secretary).
```

```
template (secretary, pool secretary, personal secretary)

    par secretary:  generic
                      category = (pool secretary, personal secretary)
                    of
                    aggregate [N]
                      N: social security number;
                      B: birthdate;
                      C: category
                    end

    operations [HIRE, ASSIGN]

        HIRE(n, b, c):   secretary <= <n, b, c>
                          {HIRE(pool secretary)/ HIRE(personal secretary)}

        ASSIGN(s, c):   s∇C <=> c
                          {insert(pool secretary), delete(personal secretary)/
                           delete(pool secretary), insert(personal secretary)}

        TERMINATE(s):   secretary => s
                          {delete(pool secretary), delete(personal secretary)}

    end


    par pool secretary:  generic
                         of
                         aggregate [N]
                           N: social security number;
                           T: typing station
                         end

    operations [HIRE, ENTER]

       *HIRE(n, t):  pool secretary <= <n, t>

        ENTER(po, t):  po∇T <=> t

    end


    par personal secretary:  generic
                             of
                             aggregate [N]
                               N: social security number;
                               O: office
                             end

    operations [HIRE, ENGAGE]

       *HIRE(n, o):  personal secretary <= <n, o>

        ENGAGE(pe, o):  pe∇O <=> o

    end

endtemplate
```

Figure 7: A specification template for the specification of Figure 6.

Similarly, to specify "secretary" relative to CDC, we may write:

```
var (CDCsct, CDCplsct, CDCprsct):
        (secretary, pool secretary, personal secretary).
```

The effect of these statements is a pairwise substitution of declared variables for parameters in the body of the template. Substitutions therefore occur in *both* the structural and the behavioral parts of the template. The net result is one specification tailored to IBM and another specification tailored to CDC. All the examples in this section will be given in the form of templates.

A specification for "matrix" is given in Figure 8. In essence a matrix is a set of entries; each entry has an associated column and row, and contains some element. Since an entry can be uniquely identified by its row and column, [I, J] is declared as an attribute key. This declaration has the effect of preventing two entries with the same row and column. The operation ENTER constructs and inserts an entry. It will generate an error condition only when an entry at (i, j) already exists. The operation REMOVE deletes the entry named "matrix[i, j]". It will generate an error condition only when this name is undefined -

```
template (matrix)
    par matrix: generic
                of
                aggregate [I, J]
                    I: column;
                    J: row;
                    E: element
                end
    operations
        ENTER(i, j, e):  matrix ⇐ <i, j, e>
        REMOVE(i, j):  matrix ⇒ matrix[i, j]
    retrieval
        FIND(i, j):  matrix[i, j].E
    end
endtemplate
```

Figure 8: A specification for "matrix".

i.e. no entry has been entered at (i, j). The only retrieval operation is FIND(i, j) which returns the E attribute of the entry named "matrix[i, j]". It will generate an error condition when this name is undefined.

Notice that the "matrix" specification does not constrain the implementor in selecting his implementation technique. However, it does delineate the basic mechanisms the implementor must provide. Since neither generalization nor operative naming is utilized, these mechanisms do not have to be supported. It is only necessary to support the aggregation of primitive components and associative naming. The aggregation could be implemented by physical adjacency of components, or by encoding the row and column components as an address. The associative naming could be implemented by hashing, or by using multilists as in a sparse array representation.

A specification for "stack" is given in Figure 9. Essentially a stack is a set of entries; each entry contains some element. Since different entries may contain the same element, stack has no attribute key. Instead entries are named operatively using the operation PUSH. An operation key [PUSH] is therefore declared. The operation PUSH constructs and inserts an entry. It will never generate an error condition. The operation POP deletes the last entry which was pushed. It will generate

```
template (stack)
     par stack: generic
                of
                aggregate
                   E: element
                end
     operations [PUSH]
          PUSH(e):  stack ⇐ <e>
          POP:  stack ⇒ stack[last pushed]
     retrieval
          TOP:  stack[last pushed].E
     end
endtemplate
```

Figure 9: A specification for "stack".

an error condition only when the entry name "stack[last pushed]" is undefined - i.e. when the stack is empty. The only retrieval operation is TOP which returns the E attribute of the last entry pushed. It will generate an error whenever the name "stack[last pushed]" is undefined.

The "stack" specification again guides the implementor without constraining him. In this case, the implementors sole task is to design an operative naming mechanism. There are many ways this could be done using sequential positioning and list structuring techniques. A comparison of the "matrix" and "stack" abstractions indicates that the only fundamental difference between them is the choice of naming mechanism.

A specification for "queue" is given in Figure 10. Intuitively, a queue is very similar to a stack - the basic difference being that one preserves a FIFO discipline and the other preserves a LIFO discipline. If a specification language supports intuitive abstraction mechanisms, then the specifications for stack and queue should explicitly reveal their similarities and differences. A comparison of Figures 9 and 10 indicates that these similarities and differences are indeed revealed. Stack and queue utilize the simplest forms of operative naming. They each have one insert operation, one delete operation and no modify operations - the insert operation is used for naming.

```
template (queue)
     par queue: generic
                of
                aggregate
                   E: element
                end
     operations [ADD]
        ADD(e):  queue ⇐ <e>
        REMOVE:  queue ⇒ queue[first added]
     retrieval
        FRONT:  queue[first added].E
     end
endtemplate
```

Figure 10: A specification for "queue".

A specification for "symboltable" is given in Figure 12. This specification is interesting in that aggregation is employed in a non-trivial way and that both operative and associative naming is used. Our example is taken from Guttag [5], where it is used to demonstrate "algebraic" specifications. The symboltable is designed for the compilation of a block structured language. When a *block* is *entered*, the *identifiers* and their *attributes* (as declared in the block heading) are *added* to the symboltable. When a block is *left*, these entries are removed. As the body of a block is being processed, the attributes of identifiers must be *found* from the symboltable. When the same identifier is declared in several blocks, it is always the attributes of the identifier declared in the closest enclosing block that are found. An identifier should not be declared twice in the same block.

If we think of a symboltable as an abstract system, there are two non-primitive abstractions involved. First, there is the abstraction "block" which has two associated operations - ENTER and LEAVE. A block has no attributes of interest - in most block structured languages, blocks are not even given identifiers. Secondly, there is the abstraction "symbol". A symbol is a relationship between a block, an identifier and some "attributes". The only direct operation on "symbol" is the ADD operation. A symboltable therefore has the aggregation hierarchy shown in Figure 11.

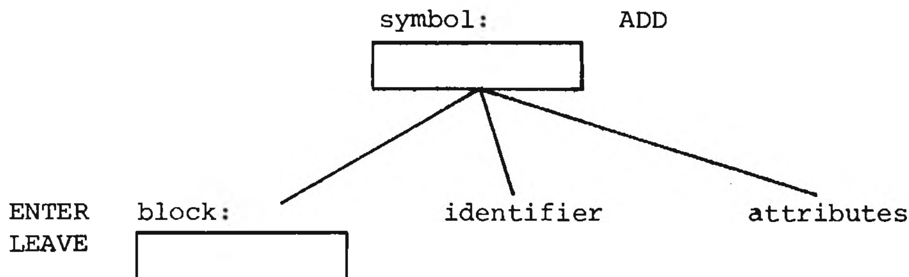

Figure 11: The aggregation hierarchy for "symboltable".

```
template (symbol, block)
     par symbol: generic
                 of
                 aggregate [B, I]
                    B: block name;
                    I: identifier;
                    A: attributes
                 end
     operations
          ADD(i, a):  symbol ⇐ <block[last entered], i, a>
     retrieval
          FIND(i):  symbol[last, where I = i, added].A
     end

     par block: generic
                of
                aggregate
                end

     operations [ENTER]
          ENTER:  block ⇐ < >
          LEAVE:  block ⇒ block[last entered]  {delete(symbol)}
     end
endtemplate
```

Figure 12: A specification for "symboltable".

Let's consider the specification of "symbol" first. The components of "symbol" are "block", "identifier" and "attributes". Since a block and an identifier are sufficient to uniquely determine a symbol, an attribute key [B, I] is declared. The operation ADD constructs and inserts a symbol. The "identifier" and "attributes" for the symbol are supplied as parameters. The "block name" is always the same - "block[last entered]". The ADD operation will generate an error condition on two occasions. One occasion is when a second symbol with the same key values is constructed - i.e. when an identifier is declared twice in the same block. The other occasion is when the name "block[last entered]" is undefined - i.e. when no block has been entered. The only retrieval operation FIND returns the "attributes" of the most recently added symbol with identifier "i". *

* To avoid iteration, we have used the "combined" form of naming mentioned in Section 2.

This operation will generate an error condition when the name

"symbol[<u>last</u>, <u>where</u> I = i, added]"

is undefined - i.e., when the identifier "i" is not declared.

The specification of "block" is very simple. A block has no categories or components of interest. An operation key must therefore be declared. Since blocks are named by the relative order of entry, [ENTER] is declared as operation key. The operation ENTER constructs and inserts a block. This operation never generates an error condition. The operation LEAVE deletes the last block entered. It will generate an error condition when the name "block[<u>last</u> entered]" is undefined - i.e. when no block has been entered. To preserve the second aggregation invariant, a triggered delete operation is specified. This will remove all symbols belonging to the block just left.

The simplicity of the "symboltable" specification arises from the explicit support of aggregation with its two invariants and appropriate naming mechanisms. The decomposition of "symboltable" into the two abstract objects "block" and "symbol" is critical - it means no more to "LEAVE" a symbol than it does to "ADD" a block. The specification suggests to the implementor that associative naming and non-trivial aggregation must be supported in an implementation of "symbol". The implementation of "block" can be quite simple - an integer "counter" would suffice.

A specification of "expression" is given in Figure 14. This specification is interesting because it involves a recursive coupling of aggregation and generalization, together with operative naming. We will consider expressions formed from operators (e.g. +, -, ×, /) and elements (e.g. 3, x, y, 5). There are two categories of expression - those which involve operators (composite expressions) and those which do not (primitive

expressions). A composite expression has three components - a left expression, a right expression and an operator. A primitive expression has one component - an element. Notice that primitive expressions and composite expressions have no components in common. The only component of an expression is its category. Expression (exp), composite expression (comp), and primitive expression (prim) therefore form the aggregation and generalization hierarchies shown in Figure 13.

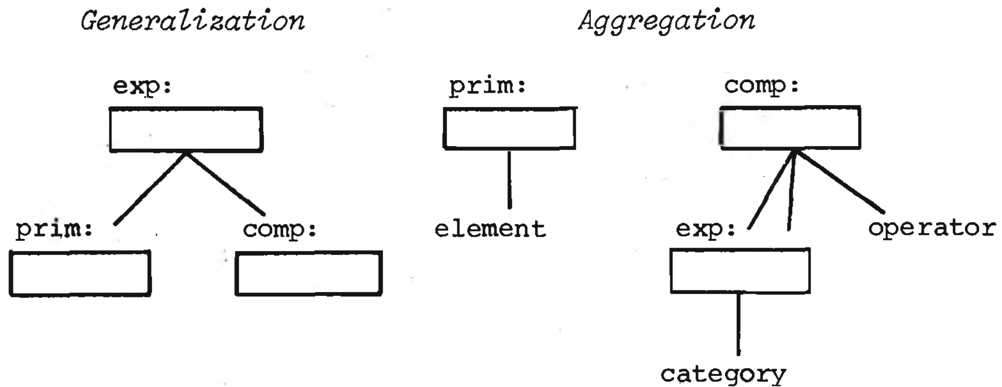

Figure 13: Aggregation and generalization hierarchies for "expression".

Since an exp has only the attribute "category", an exp is not *made* directly. Instead, either a prim or a comp is *made* and this triggers the creation of an exp. An operation MAKE is therefore defined for prim and comp. There are no particular kinds of restricted access which characterize expressions and so no special retrieval operations are defined. Let's consider the specification of expression in Figure 14.

Since a category is not sufficient to uniquely identify an exp, exp has no attribute key. To provide operative naming over exp, we have defined an operation MAKE. As indicated by the asterisk, this operation is triggered by operations on prim and comp - it is not invoked directly. Neither attribute keys nor operation keys are specified for prim and comp.

```
template (exp, prim, comp)
     par exp: generic
                 category = (prim, comp)
              of
              aggregate
                 C: category
              end
     operations [MAKE]
          *MAKE(c):  exp ⇐ <c>
     end

     par prim: generic
               of
               aggregate
                  E: element
               end
     operations
          MAKE(el):  prim ⇐ <el>  {MAKE(exp)}
     end

     par comp: generic
               of
               aggregate
                  L: expression name;
                  O: operator;
                  R: expression name
               end
     operations
          MAKE(l, o, r):  comp ⇐ <l, o, r>  {MAKE(exp)}
     end
endtemplate
```

Figure 14: A specification for "expression".

This means their individuals can only be named indirectly via exp. MAKE operations on prim and comp must trigger MAKE operations on exp to preserve the generalization invariants.

To illustrate possible retrieval operations over expressions, we will write a procedure to print an expression in postfix form. Assume that "e" is an (operative) name for an exp. The image of "e", in either prim or comp, is given by "e↓C". If "e↓C" is in comp, its left and right expressions are named by "e↓C∇L" and "e↓C∇R" respectively.

```
POSTFIX(e):
  case e∇C of
     prim: PRINT(e↓C.E)
     comp: {POSTFIX(e↓C∇R); POSTFIX(e↓C∇L); PRINT(e↓C.O)}
  end
```

6. Conclusion

A generic object in an aggregation/generalization hierarchy has four "external" manifestations: the *operations* which characterize its behavior, and the *attributes*, *relationships*, and *categories* which characterize its structure. Internally, a generic object is a (possibly complex) collection of data structures and algorithms for representing and accessing individuals over storage devices. It is important that the behavioral specification of a generic object be independent of these internal details. Individual operands must be identified only in terms of their external manifestations. In this way, the details of structural representation are factored out of behavioral specifications.

For this approach to be successful, the identification mechanism for individuals must be complete - in the sense that *all* external manifestations can be utilized. We have therefore provided four mechanisms for identifying individuals:

i) operative naming which utilizes the temporal order of operation applications to individuals,

ii) associative naming which utilizes the uniqueness of an individual's attributes,

iii) component naming which utilizes the relationships in which an individual participates, and

iv) image naming which utilizes the categories to which an individual's images belong.

Nevertheless, it is not clear that these naming mechanisms utilize the external manifestations *fully* or in the most *appropriate* way.

We have seen, for example, that the combined form of naming discussed in Section 2 is useful for eliminating some iterations from behavioral specifications. There does not seem to be any *absolute* criterion as to where naming mechanisms should end and behavioral specifications should

begin. It will therefore be useful to establish some *relative* measures of completeness. For example, we hope to show in a forthcoming paper that Guttag's "algebraic specifications" can be mechanically transformed to "integrated specifications" *without introducing any additional representation detail.*

It is clear that a virtual machine can be constructed which will execute integrated specifications as if they were high-level programs. This virtual machine must provide the representation schemata and access functions necessary to implement the four naming mechanisms over storage devices. It is unlikely that *fixed* schemata and functions will be found which are satisfactory in all applications. The virtual machine will probably contain a *menu* from which schemata and functions can be selected for each application. This selection can be supplied by external human input (as is the current practice in database management systems), or by powerful internal optimization features.

We plan to investigate the design of a programming language based on integrated specifications. Our current thinking is that a program in this language would consist of two distinct parts. One part is an integrated specification, and the other part is a selection of appropriate representation schemata and access functions. A user of the program only needs to be aware of the first part. For this purpose, integrated specifications need a fixed set of control structures, and an assertion (or query) language. An appropriate assertion language might be a typed predicate calculus with the operators "∇", "$\downarrow$", and "." and the relation "$\equiv$" built-in.

Acknowledgements:

We are grateful to Bob Keller for several stimulating interactions - he pointed out the connection of our work to "abstract syntax". Gary Lindstrom and Martin Griss made valuable comments about our "programming language" examples.

References:

[1] Codd, E. F., A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377-387.

[2] Codd, E. F., Further normalization of the data base relational model. In *Courant Computer Science Symposia 6: Data Base Systems*, Prentice-Hall, Englewood Cliffs, N.J., May 1971, 33-64.

[3] Codd, E. F., Relational completeness of data base sublanguages. In *Courant Computer Science Symposia 6: Data Base Systems*, Prentice-Hall, Englewood Cliffs, N.J., May 1971, 65-98.

[4] Guttag, J. V., The specification and application to programming of abstract data types. Ph.D. Thesis, University of Toronto, Department of Computer Science, 1975.

[5] Guttag, J. V., Abstract data types and the development of data structures. *Comm. ACM 20*, 6 (June 1977), 396-404.

[6] Liskov, B. H., and Zilles, S. N., Specification techniques for data abstractions. *IEEE Trans. on Software Engineering SE1*, 1 (Mar. 1975), 7-19.

[7] Liskov, B. H., Synder, A., Atkinson, R., and Schaffert, C., Abstraction mechanisms in CLU. *Comm. ACM 20*, 8 (Aug. 1977) 564-576.

[8] McCarthy, J., Towards a mathematical science of computation. Proc. IFIP Cong. 1962, North-Holland Pub. Co., Amsterdam, 1962.

[9] Parnas, D. L., A technique for software module specification with examples. *Comm. ACM 15*, 5 (May 1972), 330-336.

[10] Smith, J. M., and Smith, D. C. P., Database abstractions: aggregation. *Comm. ACM 20*, 6 (June 1977), 405-413.

[11] Smith, J. M., and Smith, D. C. P., Database abstractions: aggregation and generalization. *ACM Trans. on Database Syst. 2*, 2 (June 1977), 105-133.