DATA ABSTRACTION, VIEWS AND UPDATES IN RIGEL

by

Lawrence A. Rowe and Kurt A. Shoens

Memorandum No. UCB/ERL M79/5

26 January 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

1. INTRODUCTION

Previous attempts at providing data base access in a programming language were based on embedding data base constructs into an existing language [Allman 76, Chamberlin 76, Date 76, Bratsbergsengen 77, Merrett 77, and Schmidt 77]. This embedding was accomplished by calling subroutines to execute data base functions directly, by using a preprocessor to translate queries into subroutine calls, or by modifying an existing compiler. Although each of these attempts succeeded in providing access to the data base, the resulting programming environment was less than satisfactory. The data base constructs are not integrated into the language because the designers were constrained by the existing languages [Prenner 78].

RIGEL[2] is an experimental general-purpose programming language designed for the development of data base applications. It offers a better programming environment because it was designed from the start with an emphasis on the language mechanisms needed to develop data base applications. These include relations, views, and tuples as built-in data types, a flexible notation for expressing relational queries which is integrated with the iteration constructs in the language, and a data abstraction facility which handles the interface between a program and a data base well. RIGEL is the language component of a sophisticated programming environment which provides interactive program development, intermixed interpretive and compiled execution of program components, and interactive debugging. Such an interactive programming environment gives a programmer powerful tools to support the rapid development of programs [Wegbreit 71]. By the time this paper appears, the first version of the system will be implemented.

Other new languages designed expressly for data base applications are TAXIS [Mylopoulis 78] and PLAIN [Wasserman 78]. TAXIS seeks to make the development of applications easier by limiting the kinds of applications which can be written in the language and by supporting a rigidly structured programming environment. The alternative is to code applications in a good general-purpose programming language using a library of predefined abstractions designed specifically for the class of applications (e.g., procedures to display menus on a video terminal and to process user requests). As with many application-specific languages, it is not certain that TAXIS offers a clear advantage in program development time. In addition, the general-purpose

---

[2] RIGEL (rī jel) is a bright bluish star -- the most luminous in the constellation Orion. We have followed a tradition of naming languages after mathematicians and astronomical bodies. Besides, it seemed like a nice name . . .

DATA ABSTRACTION, VIEWS AND UPDATES IN RIGEL[1]

Lawrence A. Rowe and Kurt A. Shoens
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

Language constructs to support the development of data base applications provided in the programming language RIGEL are described. First, the language type system includes relation, view, and tuples as built-in types. Tuple-values are introduced to provide more flexibility in writing procedures that update relations and views.

Second, an expression that produces sequences of values, called a generator, is defined which integrates relational query expressions with other iteration constructs found in general-purpose programming languages. As a result, relational expressions can be used in new contexts (e.g., as parameters to procedures) to provide new capabilities (e.g., programmer-defined aggregate functions).

Lastly, a data abstraction facility, unlike those proposed for other data base programming languages, is described. It provides a better notation to specify the interface between a program and a data base and to support the disciplined use of views.

All of these constructs are integrated into a sophisticated programming environment to enhance the development of well-structured programs.

Keywords: Programming languages, relational data base systems, data abstraction, generators, views.

---

[1] This work was supported by the National Science Foundation under grant MCS 77-28301.

1. INTRODUCTION

Previous attempts at providing data base access in a programming language were based on embedding data base constructs into an existing language [Allman 76, Chamberlin 76, Date 76, Bratsbergsengen 77, Merrett 77, and Schmidt 77]. This embedding was accomplished by calling subroutines to execute data base functions directly, by using a preprocessor to translate queries into subroutine calls, or by modifying an existing compiler. Although each of these attempts succeeded in providing access to the data base, the resulting programming environment was less than satisfactory. The data base constructs are not integrated into the language because the designers were constrained by the existing languages [Prenner 78].

RIGEL[2] is an experimental general-purpose programming language designed for the development of data base applications. It offers a better programming environment because it was designed from the start with an emphasis on the language mechanisms needed to develop data base applications. These include relations, views, and tuples as built-in data types, a flexible notation for expressing relational queries which is integrated with the iteration constructs in the language, and a data abstraction facility which handles the interface between a program and a data base well. RIGEL is the language component of a sophisticated programming environment which provides interactive program development, intermixed interpretive and compiled execution of program components, and interactive debugging. Such an interactive programming environment gives a programmer powerful tools to support the rapid development of programs [Wegbreit 71]. By the time this paper appears, the first version of the system will be implemented.

Other new languages designed expressly for data base applications are TAXIS [Mylopoulis 78] and PLAIN [Wasserman 78]. TAXIS seeks to make the development of applications easier by limiting the kinds of applications which can be written in the language and by supporting a rigidly structured programming environment. The alternative is to code applications in a good general-purpose programming language using a library of predefined abstractions designed specifically for the class of applications (e.g., procedures to display menus on a video terminal and to process user requests). As with many application-specific languages, it is not certain that TAXIS offers a clear advantage in program development time. In addition, the general-purpose

---

[2] RIGEL (rī jel) is a bright bluish star -- the most luminous in the constellation Orion. We have followed a tradition of naming languages after mathematicians and astronomical bodies. Besides, it seemed like a nice name . . .

language can be used to program a wider range of data base applications.

PLAIN, on the other hand, is a general-purpose programming language which takes a different approach to both the expression of queries and data abstraction than the approach taken in RIGEL. In PLAIN, the programmer specifies how a high-level query is to be processed which means that programs may have to be recoded to take advantage of execution efficiencies resulting from a change to the data base storage structure. A more important difference is that, as will be shown, the data abstraction facility provided in RIGEL is better suited to defining the interface between a program and a data base than that provided in PLAIN.

This paper presents the design of language constructs provided in RIGEL to express data base queries, to specify the interface between a program and a data base, and to use views.

The notation for expressing queries is based on expressions which produce a sequence of values similar to that used in Relational PASCAL [Schmidt 77] and discovered independently by Prenner [Prenner 77]. The use of these expressions has been generalized to allow sequences of values other than tuples (e.g., real) to be specified and to allow their use in contexts other than relation constructors (e.g., as parameters to procedures). This generalization results in a convenient, well-integrated notation for expressing queries which, for example, leads naturally to programmer-defined aggregate functions (e.g., standard deviation).

The virtues of data abstractions as notations to describe interfaces to a data base have been extolled by researchers in programming languages and data base systems [Hammer 76, Prenner 77, Tsichritzis 77, Brodie 78]. Several language proposals exist with a data abstraction facility based on the concept of abstract data types [Furtado 78, Schmidt 78, Wasserman 78, Weber 78]. RIGEL has a data abstraction facility, based on a program structuring concept developed by Wirth [Wirth 77] which provides a better notation for specifying a data base interface that consists of several relations, views, and high-level abstract operations.

Finally, the language supports views as a built-in data type. Both retrieval and update operations are provided on views. The view and data abstraction mechanisms were designed together so that views can be specified in an abstraction in such a way as to separate the view representation as seen by a user of the view from its implementation and to allow updates on views to be specified as high-level abstract operations. This approach enhances data independence because the implementation of view updates is isolated

from the program. Moreover, view updates are specified explicitly so that a programmer knows precisely what set of updates are allowed, eliminating the uncertainty present in approaches based on automatically translating view updates [Stonebraker 75, Astrahan 76, Dayal 78].

These simple, yet powerful, language constructs can be used together to develop a wide range of data base applications. The remainder of this paper presents the details of the language constructs. Section 2 presents the notation for expressing queries. Section 3 describes the update constructs. Section 4 discusses data abstraction and presents an example of an external schema. Finally, section 5 illustrates the use of views in a data abstraction.

Figure 1 shows the declarations for a college data base which is used in all of the succeeding examples. There is a student relation, course relation, professor relation, and a relation expressing the association of a student enrolled in a course. Each relation has a primary key as specified in the key-clause. To simplify the presentation, each course is assumed to be offered only once each term and taught by one professor.

2. GENERATORS

The for-statement found in conventional programming languages is extended in RIGEL to express relational queries by what is called a generator expression which produces a sequence of values. This section illustrates the use of generators to express queries, to define aggregate functions, and to construct new relations.

A for-statement to sum the elements of an array is written:

```
sum := 0;
for i in 1..10 do
    sum := sum + A[i];
end;
```

The body of the for-statement is executed once for each value of i, in this example, 1, 2, ..., 10. The variable i is called an iteration-variable and the expression 1..10 is called a generator because it produces or generates the sequence of values that are to be assigned to the iteration-variable. The expression "i in 1..10" between the for and do is called a bind expression because it specifies a sequence of bindings for the iteration-variable.

The for-statement and bind expressions can be used to specify retrievals from relations. For example,

```
for s in STUDENT where s.level=soph do
    print(s.name, s.major);
end;
```

```
nameType: type = array 1..NAMESIZE of char;
idNumType: type = integer;
titleType: type = array 1..TITLESIZE of char;
gradeType: type = (A, B, C, D, F, I);

COURSE: relation
    c#: idNumType;        /* Course number */
    title: titleType;     /* Course title */
    p#: idNumType;        /* Teaches course */
    key c#;               /* Unique courses */
end;

PROFESSOR: relation
    p#: idNumType;        /* Prof's employee # */
    name: nameType;
    salary: real;
    rank: (lec, asst, assoc, full, special);
    yearsService: integer;
    key p#;
end;

STUDENT: relation
    s#: idNumType;        /* Student number */
    name: nameType;
    major: nameType;
    level: (frosh, soph, jr, sr, grad, other);
    key s#;
end;

/* A many-many relationship that indicates in
   what course each student is enrolled */

ENROLLMENT: relation
    s#: idNumType;
    c#: idNumType;
    grade: gradeType;
    key s#, c#;
end;
```

Figure 1. Example data base.

prints the name and major of all sophomore students. The iteration-variable s is bound in turn to tuples in the STUDENT relation that satisfy the predicate specified in the where-clause. Other data manipulation languages refer to s as a range variable (QUEL [Held 75]) or cursor (SEQUEL [Chamberlin 76]), but in programming languages it is just another iteration-variable. By recognizing this fact,

queries can be integrated with other iteration constructs. Notice that this notation for queries implies that each relation has an implicitly defined generator that produces each tuple in the relation.[3] Although this may seem to imply that retrievals are implemented by scanning the entire relation, query optimizations used to implement high-level query languages can be used to improve execution performance [Astrahan 76, Wong 76].

To specify more complex queries, bind expressions can be generalized to several iteration-variables. For example, to print the name, level, and grade of students in an Economics course, one writes

```
for s in STUDENT, c in COURSE, e in ENROLLMENT
  where c.title="ECON1" and c.c#=e.c# and e.s#=s.s# do
    print(s.name, s.major, s.grade);
end;
```

In these examples, both the binding of iteration-variables and the computation to be performed are specified. Some queries involve passing a sequence of values to a procedure, for example, professor salaries to an averaging function to calculate average salary. The sequence of values can be specified in the following way

```
p.salary : p in PROFESSOR
```

The colon-operator produces a value (specified by the left operand) for each binding of iteration-variables (specified by the right operand). The left operand is an expression; the right operand is a bind expression. The entire expression is a generator. To calculate average salaries, one writes:

```
AVG(p.salary : p in PROFESSOR)
```

This example illustrates how bind expressions can be used to define generators and how generators can be passed to procedures. Because these are general mechanisms in the languages, a programmer has considerable freedom to define arbitrary procedures which take generator arguments and to call those procedures with different sequences of values specified by generator expressions. For instance, programmers can define new aggregate functions as illustrated in a later section.

Generator expressions can also be used to produce a sequence of records by use of a record-constructor. When combined with a relation-constructor, a temporary relation can be created. For example, to create a temporary relation with a tuple for each student in an Economics course one

---

[3] A tuple refers to a "record in a relation." The importance of this distinction is illustrated in the next section.

writes

```
temp :=
  {<s#:s.s#, name:s.name, level:s.level, grade:e.grade>:
    s in STUDENT, c in COURSE, e in ENROLLMENT
    where c.title="ECON1" and c.c#=e.c# and e.s#=s.s#}
```

The expression "{ . . . }" constructs a relation which in this example is assigned to the variable temp.[4] The expression "< . . . >" constructs a record with record-field names that are explicitly specified and expressions that specify the values which are to be assigned to the fields. Record-constructors are like target-lists in QUEL [Held 75].

In these examples, the use of generator expressions to specify sequences of values has been illustrated. Generators can also be specified by a procedure-like routine called a generator. Generator routines are similar to CLU iterators (for details see [Liskov 77] or [Rowe 78]). Regardless of how a generator is specified, they can be used in all of the contexts illustrated here (e.g., in bind expressions or as arguments to procedures).

We have shown that bind expressions and generators are fundamental unifying concepts that provide a good notation for expressing queries. They extend previous work ([Prenner 77, Schmidt 77]) by allowing relational expressions (i.e., generators) to be passed to procedures as arguments and by allowing generators to be specified procedurally.

3. UPDATES

In this section language constructs to specify updates (replaces, deletes, and appends) and data base transactions are presented. Replaces and deletes are specified using an extension to the for-statement which identifies relations to be modified, called an update-statement. Simple statements, used inside the update-statement are provided to carry out replaces and deletes. An additional statement, which may be used anywhere, is available to append tuples to relations. A language construct is also provided to define data base transactions, a sequence of updates that appear to concurrent data base users as an atomic operation. At the end of this section, an extension to the language type system is described which allows updates to be performed in procedures called from the body of an update-statement.

The replace- and delete- statements, when used inside an update-statement, cause the corresponding action to take place. For example, suppose all professors are to be given raises but that the amount of raise is dependent on current

---

[4] RIGEL is a strongly-typed language; thus, all variables must be declared. The declaration constructs have been omitted to simplify the presentation.

salary, rank, and years of service. Assume that a function, called newSalary, is provided that calculates the professor's new salary. To update salaries one writes

```
update p in PROFESSOR do
  replace p by
   p<salary:newSalary(p.salary,p.rank, p.yearsService)>;
end;
```

The semantics of this statement are: open the relation for update, replace each tuple by the constructed record specified in the by-clause, and do the update (i.e., physically change the PROFESSOR relation) after all iterations have been completed (i.e., just before executing the statement following the update-statement). The expression "p< . . . >" constructs a new record by making a copy of p and then changing the specified fields in parallel. This simultaneous change is needed to avoid integrity violations that might arise from constraints that relate two record fields.

To delete tuples, a delete-statement rather than a replace-statement is used in the body of the update-statement. For example, to delete all courses taught by A. Johnson, one writes

```
update c in COURSE for p in PROFESSOR
  where p.name="Johnson, A." and p.p#=c.p# do
    delete c;
end;
```

In this example, the COURSE relation can be updated but not the PROFESSOR relation. Deletes and replaces to different relations can be intermixed in the body of the update-statement. However, a relation can only have one updatable iteration-variable active at a time. This restriction is enforced by a run-time check. The update constructs described here do not solve the problem of non-functional updates.

Records are inserted into a relation by using an append-statement. For example,

```
append
  <s#:1024,name:"Smith, K.",major:"EECS",level:jr>
  to STUDENT;
```

inserts a new student into the STUDENT relation. In contrast to the delete- and replace-statements, the append-statement is not used in the body of an update-statement.

A data base transaction is expressed by using a transaction-block, e.g.,

```
transaction
    updates COURSE,PROFESSOR;
begin
    update p in PROFESSOR
      where p.name="Johnson, A." do
        update c in COURSE where c.p#=p.p# do
            delete c;
        end;
        delete p;
    end;
end;
```

This example deletes Professor Johnson and all courses s/he teaches. The updates list is required because update-statements may occur in called procedures; without it, interprocedural data-flow analysis would be required to determine which relations might be changed. The ability to group together updates in order to maintain data base integrity is a necessary language feature for the development of data base applications.

A major problem in designing the update constructs is to allow a procedure, called in the body of an update-statement, to replace or delete a tuple. For instance, in the salary increase example at the beginning of this section, a procedure updateSalary that takes a PROFESSOR tuple and replaces the salary could have been provided rather than newSalary. The updates would then be coded

```
update p in PROFESSOR do
    updateSalary(p);
end;
```

The problem is what type is the formal argument to updateSalary? Up to this point, an iteration-variable bound to a tuple in a relation has been treated as a record-value. That is, the iteration-variable is actually bound to a copy of the tuple. However, to do updates, a reference to the tuple is needed rather than to a copy. The solution is to introduce a type that denotes "a reference to a tuple in a relation." UpdateSalary can then be coded as follows

```
updateSalary: procedure(a: tuple PROFESSOR'type) =
begin
    . . .
    replace a by a< . . . >;
    . . .
end;
```

The tuple-type specifies the type of relation of which the tuple is a member. In this case, "PROFESSOR'type" specifies the type of the variable PROFESSOR. The suffix "'type" is an example of an attribute function.

There is a problem with introducing values of type tuple into the language. In actuality, tuple-values are

addresses on secondary memory. Consider the following program fragment

```
for s in STUDENT where s.name="Smith, K." do
    x := s;
end;
```

After executing this code, x would reference Smith's tuple in the STUDENT relation. Somewhere later in the program one might try to reference Smith's tuple through x. This cannot be allowed because the tuple might have been moved by a later update from this program or from a concurrent user or by a change in the storage structure used to store the STUDENT relation. This restriction is achieved by not allowing tuple-values to be assigned. In this example, the assignment "x := s" is flagged as a compile-time error.

This section has presented a complete set of update constructs which have been carefully integrated into the retrieval mechanisms presented in the previous section. The introduction of tuple-values allows replaces and deletes to be executed in procedures called from the body of an update-statement, which enhances the power of procedures in the language. This feature is exploited in the view update mechanism discussed below. Before discussing the view mechanisms though, the data abstraction constructs must be presented.

4. DATA ABSTRACTION

In this section the data abstraction facility provided in RIGEL is described. Data abstractions are used to define the interface between a program and a data base, as well as the interfaces between parts of a large program. The abstraction facility is also used to separate the representation of a view from the code which implements it. Thus, the data abstraction facility is extremely important for the development of well-structured data base applications.

Data abstraction mechanisms can be provided in a language by abstract data types (ADT) or by modules. An ADT mechanism allows a programmer to associate primitive operations with one data structure type (e.g., a stack with pop, push, isEmpty, and top operations). The declaration of an ADT adds a new type to the language which can be used, for example, to declare variables of that type (e.g., stack). Thus an ADT mechanism is designed to define one type which can be used to create several instances of the object. For example, ADT's are provided in SIMULA [Birtwistle 73] because simulation programs are concerned with the movement of multiple instances of various types of entities through a system.

The difference between ADT's and modules is that modules do not define a single type. A module mechanism is designed to allow several types, variables, and procedures

to be declared together and to be used in an application program as a unit. A simple example would be an I/O subroutine library which defines types (buffer and file descriptor types), variables (status variables), and procedures (*openFile*, *read*, etc.). A program that uses the I/O module creates only one instance of the objects it defines, called *importing* the module. Modules are provided in MODULA [Wirth 77] because system programs are primarily concerned with interfaces between pieces of hardware and software. These interfaces are comprised of more than one type and are only instantiated once in a program.

ADT's can be used like modules and modules can be used to define ADT's, but, each is more convenient for the use for which it was defined. More extensive comparisons of these two abstraction mechanisms are available elsewhere ([Goos 78, Wirth 77]). Nevertheless, ADT's are best suited to problems which require multiple instantiations of one type while modules are best suited to problems which require one instantiation of a collection of types, variables, and procedures.

The predominant use of data abstraction in data base applications is to define the interface between a program and a data base, called an *external schema* by some data base practitioners. This interface is often quite complex and only one instance is needed. Clearly, modules are the most natural notation for their specification.

A module in RIGEL has three sections: visible, private, and initialization. The visible section defines the objects that can be accessed by the user of the module, called its *client*. The private section implements the objects defined in the visible section. Objects local to the private section cannot be directly accessed by the client. The initialization section is executed when the module is defined so that necessary initialization can be completed.

The visible section specifies the abstract semantics and the private section specifies the implementation. Used correctly, modules provide a means for isolating a client program from the representational details of the abstraction. Figure 3 shows a module definition for an interface to the college data base that might be used by a program that is scheduling professors and classes.[5] The visible section includes all declarations that must be seen by an application programmer to use the relations and procedures. After importing the module, the relations and procedures are

---

[5] The binding of program variables COURSE and PROFESSOR to relations stored in a data base is not shown. Language mechanisms to control this binding were discussed in a previous paper [Prenner 78]; details are also given in the language specification [Rowe 78].

```
scheduleSchema: module =
visible

    COURSE: relation
        c#: idNumType;
        title: titleType;
        p#: idNumType;
        key c#;
    end;

    PROFESSOR: relation
        p#: idNumType;
        name: nameType;
        salary: real;
        rank: (lec, asst, assoc, full, special);
        yearsService: integer;
        key p#;
    end;

    assignProf: procedure(prof: nameType,
      class: titleType);

    scheduleReport: procedure;
    . . .

private

    assignProf: procedure =
    begin
        update c in COURSE for p in PROFESSOR
          where c.title = class and p.name = prof do
            replace c by c<p#: p.p#>;
        end;
    end;
    . . .
end;
```

Figure 3. Example module declaration.

accessed in the usual way. For example, to assign Professor R. Burns to teach PHYSICS 1, one writes

```
assignProf("Burns, R.", "PHYSICS1");
```

Notice that the number of arguments and their types are given in the visible section because a client must know this information to use the procedure. They are not repeated in the private section where the procedure body is specified to avoid needless inconsistency errors and to insure that if

the argument declarations are changed only one change to the program text is necessary.

Modules provide a better notation than ADT's for specifying a program interface to a data base. For example, only those relations and views which a program may access are declared in the visible section. Moreover, high-level abstract operations can be associated with the data by declaring them in the module, thus providing a complete specification of the data base interface.

5. VIEWS

This section illustrates the use of updatable views in RIGEL. A view is an abstract relation which can be used to simplify the access to a data base, to protect parts of a data base, and to enhance data independence [Chamberlin 75]. The definition of a view is comprised of the declaration of the abstract relation which the view simulates and the specification of how a view tuple is constructed from the tuples in the underlying relations (called the base relations). These parts are called, respectively, the abstract relation declaration and the mapping specification. The user of a view should only see the abstract relation declaration because the mapping specification is part of the view implementation. This separation and information hiding can be achieved by putting the abstract relation declaration in the visible section of a module and the mapping specification in the private section of a module. In this way, the use of modules complements the use of views.

For performance and consistency reasons, queries on a view are translated into queries on the base relations rather than being executed directly on a materialization of the view [Stonebraker 75]. This translation is possible for all retrievals but not for all updates. Views and updates exist for which no sequence of operations on the base relations will produce a correct update when the data is accessed through the view (i.e., undefined updates). Other examples exist for which more than one sequence of operations will produce a correct view update but each produces a very different meaning on the underlying data (i.e., ambiguous updates) [Dayal 78]. The problem is to decide how view update facilities should be provided in a language. There are four possibilities:

(1) Views may not be updated.

(2) Views may be updated but the programmer must specify what base relation updates are to be executed, i.e., for those updates that are to be allowed, the programmer must specify the translation of the high-level operation.

(3) Views may be updated only by updates that can be unambiguously translated to base relation updates by some

update translation algorithm.

(4) Views may be updated and language constructs are provided to allow a programmer to specify the translation of ambiguous and undefined updates that are not handled by the translation algorithm, i.e, all updates are allowed.

The first alternative was rejected because it seriously restricts the utility of views.

Several attempts were made to formulate language constructs to allow programmer-defined augmentations to an update translation algorithm, i.e., the fourth alternative. Each attempt introduced significant language complexity and, in most cases, resulted in programs that would be impractical to execute.

The third alternative attempts to support arbitrary view updates by using the standard update constructs defined for relations. However, should one desire an ambiguous update no convenient notation is supplied for the programmer to express that update. Another consequence of adopting this alternative is that a programmer would not know whether a particular view update was legal until it was translated, or in some cases, executed [Stonebraker 75, Astrahan 76, Dayal 78]. Moreover, the update translation algorithms depend on the view mapping specification which the programmer presumably cannot access. The second alternative, on the other hand, acknowledges that only a limited set of view updates are possible and provides a way to associate the appropriate semantics with the view update expressed as a high-level abstract operation. In most applications, because few distinct view updates are required, the burden on the programmer to specify the translation would be insignificant. Moreover, data independence would be enhanced by the discipline of explicitly coding the view updates as high-level operations. Consequently, the second alternative was judged superior to the third alternative.

The remainder of this section presents the language constructs provided in RIGEL to define and use views. The presentation is based on a module that might be used by the professor of a class. The module is comprised of a view of the students enrolled in the class and operations that s/he might access to handle some typical actions: adding and dropping students, counting the number of classes a student is taking, assigning grades, and calculating the grade point average of various groups of students.

Figure 4 shows the visible section of the courseView module. It includes the view for ECON1 and the abstract operations. Assuming that the module has been imported into the current scope, a query to list students with incompletes is written:

```
courseView: module =
visible

    ECON1: view
        s#: idNumType;
        name: nameType;
        grade: gradeType;
    end;

    addStudent: procedure(student: nameType);
    dropStudent: procedure(
      student: tuple ECON1'type);
    assignGrade: procedure(
      student: tuple ECON1'type;
      courseGrade: gradeType);
    countClasses: procedure(
      student: tuple ECON1'type): integer;
    avgGPA: procedure(
      grades: generator:gradeType): real;

private
    . . .
end;  /* courseView */
```

Figure 4. Visible section of courseView module

```
for s in ECON1 where s.grade = I do
    print(s.name);
end
```

To add a student to the course, the addStudent procedure is invoked, e.g.,

```
addStudent("Smith, K.")
```

To assign a grade to each student the following is executed

```
update s in ECON1 do
    assignGrade(s,getGrade(s.name));
end
```

where getGrade is a function that returns a grade for each student (e.g., it might be read from a terminal). Notice that the type of the first argument to assignGrade is a tuple in the view. AssignGrade will perform an update on the student's tuple in the ENROLLMENT base relation.

Other examples to illustrate view use are

```
/* calculate class grade point average */
avgGPA(s.grade: s in ECON1 where s.grade not= I)

/* gpa of students taking more than 4 courses */
avgGPA(s.grade: s in ECON1 where countClasses(s) > 4)
```

These examples illustrate the power and convenience of generator expressions and generators as arguments to procedures. AvgGPA is an example of a programmer-defined aggregate function.

The implementation of this module is shown in Figure 5. Lines 8 to 12 complete the view declaration. The view mapping is specified by a generator expression which produces the tuples in the abstract relation. The particular course for which this module defines a view is specified as a constant in the mapping specification. Obviously, a better way to structure this module would be to make the course a module parameter so that one module could be used for all courses rather than having to define a separate module for each one.

CountClasses takes a view tuple and accesses the ENROLLMENT base relation to find the number of classes in which the student is enrolled. AvgGPA takes a generator argument and calculates the grade point average of the grades returned by the generator. ToGPA is an array that maps grades to the obvious numerical ranking. AddStudent is a procedure which causes a new tuple to be inserted into the view by performing an update on the ENROLLMENT base relation. The implementation of these operations could be expressed directly on the base relations without reference to the base relation tuples that produced a specific view tuple.

DropStudent takes a view tuple as an argument and deletes the tuple in the ENROLLMENT base relation that represents the student's enrollment. The problem is how to access the ENROLLMENT tuple given a view tuple. One solution would be to execute a query to find the tuple, e.g.,

```
update e in ENROLLMENT for c in COURSE
  where student.s#=e.s# and c.c#=e.c# and c.title="ECON1"
  do
    delete e;
end;
```

where student is the argument to the procedure dropStudent. This solution is intolerably inefficient. Additionally, this implementation of the update implies an update to the ENROLLMENT relation for each student dropped rather than deferring the change until all students to be dropped have been processed. In other words, the semantics of the view update are wrong. A better solution would be to allow dropStudent to access the base tuples directly. This solution is achieved by allowing the base relation tuples used

```
courseView: module =
visible
    . . .
private

    /* view mapping specification */

    ECON1: view =
      <s#:e.s#, name:s.name, grade:e.grade>:
      e in ENROLLMENT, s in STUDENT, c in COURSE
      where c.title = "ECON1" and c.c# = e.c#
      and s.s# = e.s#;

    countClasses: procedure =
    begin
        return COUNT(x in ENROLLMENT
          where x.s# = student.s#);
    end;

    /* initialize grade point mapping */

    toGP: array gradeType of real:= [4,3,2,1,0,0];

    avgGPA: procedure =
    begin
        return AVG(toGP[g]: g in grades
          where g not= I);
    end;

    addStudent: procedure =
    begin
        if (s in STUDENT, c in COURSE
          where s.name = student
          and c.name = "ECON1")
        then
            append
              <s#: s.s#, c#: c.c#, grade: I>
              to ENROLLMENT;
        else
            print("No such course or student");
        fi;
    end;

    assignGrade: procedure =
    begin
        replace student'base.e by
          student'base.e<grade:courseGrade>;
    end;

    dropStudent: procedure =
```

```
51          begin
52              delete student'base.e;
53          end;

55      end /* Course view */
```

Figure 5. Private section of courseView module

---

to construct the view tuple to be accessed given a view tuple.

Associated with each view tuple is a record, called the view tuple base record, which has a field that holds the current value for each iteration-variable bound in the mapping specification for the view, e.g., the base record declared for ECON1 view tuples is

```
record
    e: tuple ENROLLMENT'type;
    s: tuple STUDENT'type;
    c: tuple COURSE'type;
end
```

To access this record given a view tuple, say student, one applies the "base" attribute-function to student i.e., "student'base." Thus, to access the ENROLLMENT base relation tuple, one writes

```
student'base.e
```

Because this is just a value of type tuple in a relation, delete is a legal operation. In this way, dropStudent can delete the appropriate base tuple directly. AssignGrade also uses this feature to replace the grade stored in the ENROLLMENT tuple.

This section has illustrated the language constructs provided in RIGEL to support the disciplined use of views. The approach to views taken here is very different from that taken in other languages. An explicit specification is given for the type of the abstract relation that the view simulates and for the update operations that are allowed on the view. Another important difference is that it is possible to separate the view declaration from the view implementation in accordance with the principles of information hiding by defining the view inside a module. Recall that a module is the language construct used to specify the data base interface to a program.

6. CONCLUSION

The main contribution of this paper is to show how various programming language concepts can be generalized and used together to provide a good environment for writing data

base applications. First, the language type system was extended to allow definition of objects of type relation, view, and tuple and to support their disciplined use. Second, generators were introduced as a notation to specify complex retrievals and updates on tuples in views and relations. Finally, the use of modules, along with relations and views, was shown to provide a convenient notation for specifying program interfaces to data bases.

REFERENCES

[Allman 76] Allman, E., Stonebraker, M., and Held, G., "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," Proc. of a Conf. on Data: Abstraction, Definition, and Structure, SIGPLAN Notices, 11, Special Issue, (March 1976), pp. 25-35

[Astrahan 76] Astrahan, M., et al, "System R: Relational Approach to Database Management," ACM Trans. on Database Sys., 1, 2, (June 1976), pp. 97-137.

[Birtwistle 73] Birtwistle, G. M., et al, SIMULA BEGIN, Petrocelli, (1973).

[Bratsbergsengen 77] Bratsbergsengen, K. and Risnes, O., "ASTRAL -- A Structured Relational Applications Language," Proc. SIMULA Users Conf., (September 1977).

[Brodie 78] Brodie, M., and Schmidt, J., "What is the Use of Abstract Data Types in Data Bases?" Proc. 4th Int'l Conf. on Very Large Data Bases, West Berlin, (1978), pp. 140-141.

[Chamberlin 75] Chamberlin, D., et al, "Views Authorization and Locking in a Relational Database System," Proc. AFIPS 1975 NCC, Vol. 44, pp. 425-430.

[Chamberlin 76] Chamberlin, D. D., et al, "SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control," IBM Journal of Research and Development, 20, 6, (November 1976), pp. 560-575.

[Date 76] Date, C., "An Architecture for High-Level Language Database Extensions," Proc. 1976 SIGMOD Conf., (June 1976).

[Dayal 78] Dayal, U., "On the Updatability of Relational Views," Proc. 4th Int'l Conf. on Very Large Data Bases, West Berlin, (1978), pp. 368-377.

[Furtado 78] Furtado, A. L., "A View Construct for the Specification of External Schemas," Series: Monografias em Ciencia da Computacao (M. Challis, ed.), (1978).

[Goos 78] Goos, G., and Kastens, U., "Programming Languages and the Design of Modular Programs," in Constructing Quality Software, (P. Hibbard and A. Schuman, eds.), North-Holland Publishing Company, (1978).

[Hammer 76] Hammer, M., "Data Abstractions for Databases," Proc. of Conf. on Data: Abstraction, Definition, and Structure, SIGPLAN Notices, 11, Special Issue (1976), pp. 58-59.

[Held 75] Held, G., Stonebraker, M, and Wong, E., "INGRES -- A Relational Data Base System," Proc. AFIPS 1975 NCC, Vol. 44, pp. 409-416.

[Liskov 77] Liskov, B., et al, "Abstraction Mechanisms in CLU," CACM, 20, 8 (August 1977), pp. 564-576.

[Merrett 77] Merrett, T., "Aldat -- Augmenting the Relational Algebra for Programmers," Technical Report SOCS-78.1, School of Computer Science, McGill University, (November 1977).

[Mylopoulis 78] Mylopoulos, J., et al, "A Preliminary Specification of TAXIS: A Language for Designing Interactive Information Systems," Technical Report CCA-78-02, Computer Corporation of America, (January 1978).

[Prenner 77] Prenner, C., "A Uniform Notation for Expressing Queries," ERL Memorandum M77/60, Electronics Research Laboratory, U. C. Berkeley, (September 1977).

[Prenner 78] Prenner, C. and Rowe, L., "Programming Languages for Relational Data Base Systems," Proc. AFIPS 1978 NCC, vol. 47, pp. 849-855.

[Rowe 78] Rowe, L., and Shoens, K., "RIGEL: Preliminary Language Specification," Dept. Elec. Eng. and Comp. Sci., U. C. Berkeley, (December 1978).

[Schmidt 77] Schmidt, J., "Some High Level Language Constructs for Data of Type Relation," ACM Trans. on Data Base Sys., 2, 3, (September 1977), pp. 247-261.

[Schmidt 78] Schmidt, J., "Type Concepts for Database Definition," Proc. Int'l Conf. on Data Bases, Haifa, Israel, (August 1978).

[Smith 77] Smith, J. and Smith, D., "Integrated Specifications for Abstract Systems," Technical Report UUCS-77-112, Computer Science Department, University of Utah, (September 1977).

[Stonebraker 75] Stonebraker, M., "Implementation of Integrity Contraints and Views by Query Modification," Proc. 1975 SIGMOD Conf., pp. 65-78.

[Stonebraker 76] Stonebraker, M., et al, "The Design and Implementation of INGRES," ACM Trans. on Database Sys., Vol. 1, No. 3, (September 1976), pp. 189-222.

[Tsichritzis 77] Tsichritzis, D., "Research Directions in Data Base Management Systems," SIGMOD Record, 9, 3, (1977), pp. 26-41.

[Wasserman 78] Wasserman, A., et al, "Report on the Programming Language PLAIN," TR-34, U. C., San Francisco, (1978).

[Weber 78] Weber, H., "A Software Engineering View of Data Base Systems," Proc. 4th Int'l Conf. on Very Large Data Bases, (1978), pp. 36-51.

[Wegbreit 71] Wegbreit, B., "The ECL Programming System," Proc. AFIPS 1971 FJCC, Vol. 39, pp. 253-262.

[Wirth 77] Wirth, N., "Modula: A Language for Modular Multiprogramming," Software -- Practice and Experience, Vol. 7 (1977), pp. 3-35.

[Wong 76] Wong, E, and Youssefi, K., "Decomposition -- A Strategy for Query Processing," ACM Trans. on Database Sys., 1, 3, (September 1976), pp. 223-241.