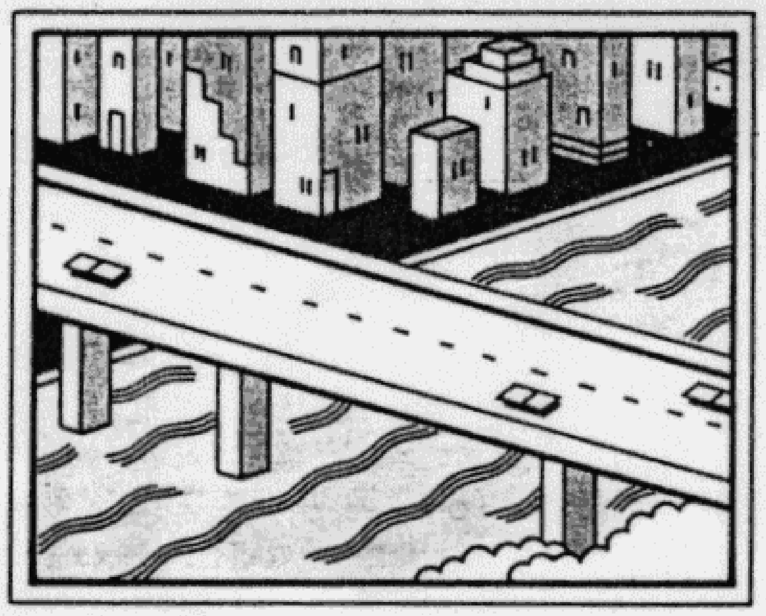


Algres: An Advanced Database System for Complex Applications

Stefano Ceri, *Universitá di Modena*
Stefano Crespi-Reghizzi *and* **Roberto Zicari**, *Politecnico di Milano*
Gianfranco Lamperti *and* **Luigi A. Lavazza**, *TXT SpA*

This relational programming environment extends the relational model to handle complex objects and operations and integrates the logic programming paradigm.

The cost and complexity of large software-development projects has spurred research into innovative development environments. Algres is an advanced relational programming environment for the development of data-intensive applications that perform complex operations over complex data structures.

Algres is designed to be used to develop knowledge bases, software-engineering systems, office-automation systems, and computer-aided design and manufacturing databases.

Algres originated from the Art project,[1] which gave us positive results in using a core-memory-resident relational database to realize working prototypes. However, Art also revealed two major limitations of pure relational algebra as a prototyping tool:

- The relational model cannot directly model nested structures; it requires that you flatten structures and introduce artificial entities such as references or pointers.
- Because relational algebra is not computationally complete, it must be embedded in a traditional language.

Algres overcomes these limitations, yet stays as close to the relational approach as possible.

Algres is based on a rigorous formalization of a data model, which is an extension of the relational model, and operations, which constitute an extended relational algebra.[2-4] Algres operations can manage complex objects (similar to non-first-normal-form relations[2]) and express recursive algebraic expressions.[5] Therefore, Algres integrates two important research areas: extending the relational model to handle complex objects and integrating database technology and logic programming.

0740-7459/90/0700/0068/$01.00 © 1990 IEEE

Algres components

The Algres project is a composite project that incorporates the results of several research efforts:

- We developed a data model that could support the definition of complex objects through the type constructors record, set, multiset, and sequence. Algres lets you define (to a finite depth) complex objects that directly model common hierarchical data structures.

The dynamic part of our model is expressed through an algebraic language, Algres-Prefix, designed to limit loop complexity and suppress linked-list structures, two major sources of programming costs. Algres-Prefix extends relational algebra with restructuring operations for nesting and unnesting objects, direct representation of ordered sets and multisets, and tuple and aggregate functions. It also supports a closure operator, which lets you define recursive or deductive queries, as advocated by proponents of Prolog-like query languages.

Together, these constructions make a very expressive formalism that we have experimented with in several applications, including a database for a tool that supports algebraic formal specifications and parts of a computer-integrated-manufacturing system.

Algres-Prefix is a complex and novel language, so we have provided two ways to formally specify the semantics of the language:

- We have given an interpretation of a complex object's schema as a context-free grammar with regular right parts. In this way, we define the valid instances of Algres objects as strings that the grammar generates and define algrebraic operations as grammar transformations. Once transformed, the new grammar generates a string for the resulting Algres object. This approach, derived from syntax-directed translator theory, provides a very abstract attribute grammar as a foundation for the Algres compiler and interpreter.
- We used Rap, a tool for axiomatic abstract-data-type specifications developed at the University of Passau, West Germany, to formally specify Algres-Prefix.[6] From the executable Rap specifications, we obtained a preliminary prototype of the Algres environment, which we recoded in Prolog to improve efficiency. The Prolog prototype has let us evaluate critical Algres constructs early in its development and influenced the final system's implementation.
- Although we integrated Algres with a commercial relational database system (Informix), we did *not* implement it on top of the commercial system. We did this for efficiency: Operations involving nested loops, nested relations, or fixpoint computations would be very expensive if executed in a conventional database system. We assume that Algres will be applied exclusively to medium-sized complex data structures, rather than to very large databases.

Figure 1 shows the Algres environment. Algres runs on Sun 3/60 and 3/80 workstations and Digital Equipment Corp. VAX computers under Unix. The environment's core is the translator from Algres-Prefix to RA (relational algebra) object code, which is read by the RA machine. The RA abstract machine provides runtime support and includes a compiler, command interpreter, and memory-management unit. The Informix database-management system stores relations.

As Figure 1 illustrates, Algres is best described as an operational environment for complex data manipulation, rather than as a database system, which sets it apart from other projects that have augmented a relational database system with support for nested relations. The RA machine is very different from the runtime support for other dynamic languages (like Lisp) because its basic operations are

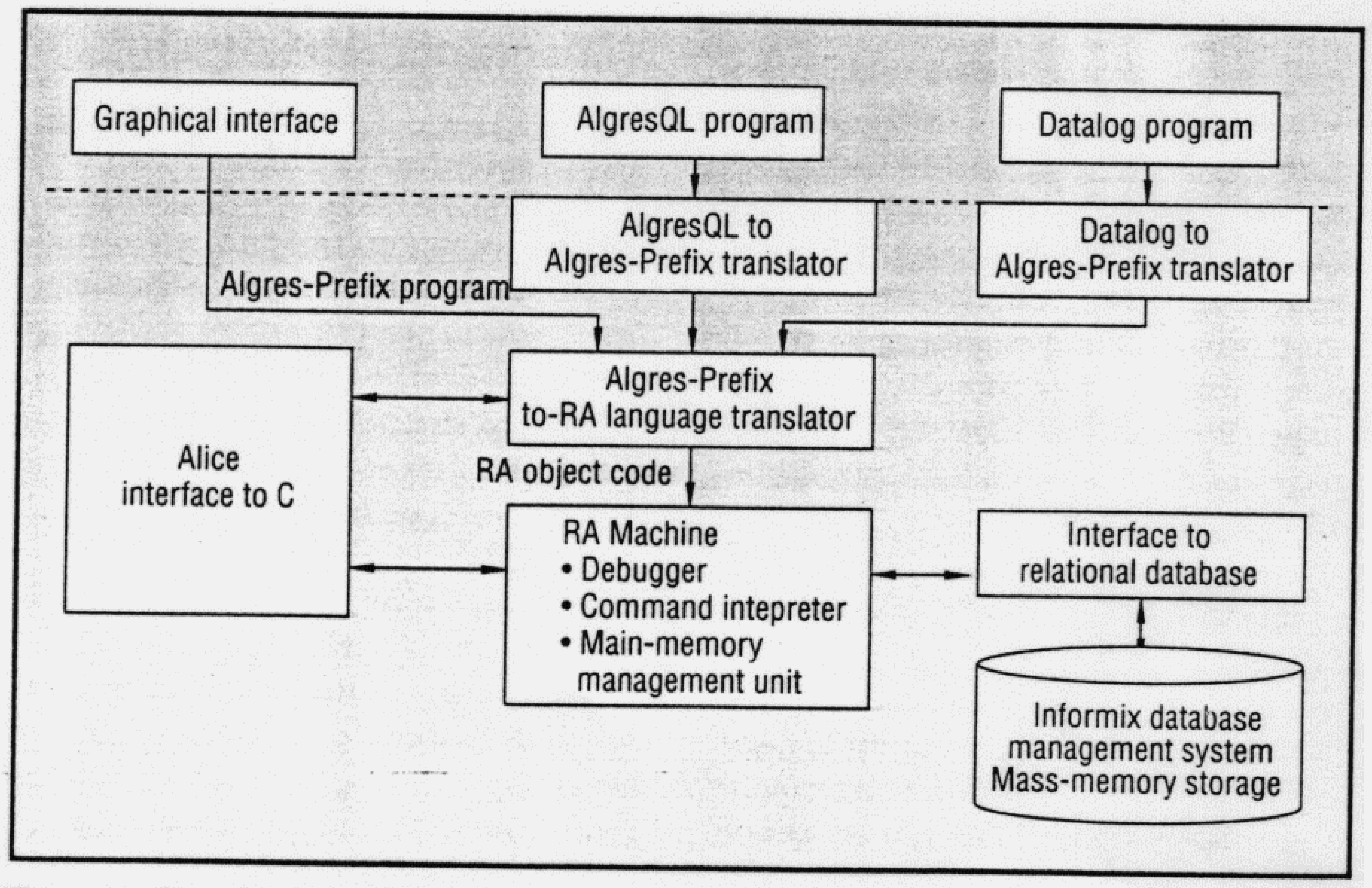

Graphical interface
AlgresQL program
Datalog program
AlgresQL to Algres-Prefix translator
Datalog to Algres-Prefix translator
Algres-Prefix program
Algres-Prefix to-RA language translator
Alice interface to C
RA object code
RA Machine
• Debugger
• Command intepreter
• Main-memory management unit
Interface to relational database
Informix database management system Mass-memory storage

Figure 1. The Algres environment.

relational and its memory allocation is compact (no lists or garbage collection).

Algres programs operate on main-memory data structures, which are loaded from mass-memory storage in an external database using explicit statements. The RA machine can load entire relations or portions of them (tuples in particular) using selective and projective load operations.

The design of the system is based on the assumption that a typical Algres user will require only a fraction of the mass-memory database. We thus assume that main memory will be sufficient to accommodate Algres databases and that speed — not memory — will be the limiting factor.

Algres objects are stored, both in mass memory and in main memory, as conventional relations. Algres-Prefix programs are translated into the RA intermediate, relational-algebra language. RA provides traditional algebraic operations on normalized relations, as well as special features. We designed it to provide object code that can be executed efficiently in the main-memory RA machine.

Algres's runtime efficiency does not compete with traditional imperative Algol-like solutions, but it does compare favorably with other tools and high-level languages that have been used successfully for rapid prototyping, such as Setl and Prolog.

- We have developed two user languages you can use with Algres instead of using Algres-Prefix directly, which is like the assembly language for the RA machine and is *not* easy to use. We designed Algres Query Language, a multiparadigm, easy-to-use programming environment that is an extension of Structured Query Language. Programs in Algres Query Language are translated into Algres-Prefix before execution.

We have also used Datalog[7-9] to graft a logical style (à la Prolog) onto Algres. Datalog is a clause-based logic language developed in the database community that is syntactically similar to Prolog and designed to retrieve information from flat relational databases. Although much work remains to be done to integrate the relational and logic paradigms, we have developed some practical algorithms to translate Horn clauses into algebraic expressions, at least for basic recursive queries. Datalog programs are also translated into Algres-Prefix before execution.

- We have developed Alice, an Algres-to-C interface to exploit existing libraries and to make Algres facilities available to C programs. Space limitations prevent us from describing Alice, which is considerably more advanced than current database-to-language interfaces.
- Finally, we designed a graphical interface to display and create the complex objects and the schemas of their relations. A full description of this window-oriented interface is outside this article's scope.

Algres data model

The Algres data model incorporates standard elementary types (like character, string, integer, real, and Boolean) and the type constructors record, set, multiset,

Algres's runtime efficiency does not compete with traditional imperative Algol-like solutions, but it does compare favorably with other tools and high-level languages that have been used successfully for rapid prototyping.

and sequence. A record is a type constructor for building tuples, which can be collected into sets, multisets, and sequences.

In Algres, an object is a pair consisting of a schema and an instance. The schema is a hierarchical structure of arbitrary (but finite) depth that you build with the type constructors. The instance must be type-compatible with the schema.

An Algres-Prefix program has two instruction types, which you can interleave arbitrarily: data definitions, to define objects, and algebraic operations applied to objects.

Object definition. To use a practical example, suppose you wanted to model a production-control system. First you would define a set of products, each of which is composed of a set of componen in a given quantity. To do so, you woul define an object consisting of two neste sets. The first set is the set of all product the second is the set of all the product components. We believe this model is ve natural because it does not require any a tificial objects (like pointers) that do n correspond to any real entity.

In this example, the Algres-Prefix dat definition instruction is

```
DEF Product-Set: SET OF (
  Product-Code: string;
  Description: string;
  Elements: SET OF (
    Component-Code: string;
    Component-Quantity: integer)).
```

This defines the schema of the objec Product-Set. Description is an elementar attribute of Product-Set and Elements is complex attribute (or subobject) of Product-Set because it contains the set-of con structor. You name all objects and thei attributes through such data definition — you must give distinct names to all attributes of objects or subobjects.

You can represent an Algres object' schema in a graphical tree structure, a Figure 2 shows. A tree's internal node represent complex attributes and are labeled □ for records, {} for sets, [ ] for multisets, and <> for sequences. A tree's leave represent elementary attributes and are labeled with their type. Figure 2a show the Product-Set object's schema.

You can retrieve objects from a database, create them by entering definitions, or define them with an assignment statement like this one, which defines the object Product-Set:

```
Product-Set <- { ( P1 Bench  { ( P2 4 )
                               ( P3 8 ) } )
                 ( P2 Leg    { } )
                 ( P3 Stick  { } )
                 ( P4 Table  { ( P2 4 )
                               ( P5 1 ) } )
                 ( P5 Top    { ( P3 4 )
                               ( P6 2 ) } )
                 ( P6 Drawer { } ) }
```

In assignment statements, () enclose tuples (records), {} enclose sets, [] enclose multisets, and <> enclose a sequence. You denote empty sets, multisets, and sequences simply by leaving the space between their respective enclosure symbols empty.

To complete our example production-control system, we must also define objects that correspond to a daily sequence of manufacturing operations. We define each manufacturing operation as a multiset of assembling operations in which a given quantity of products is assembled on a given assembly line. Furthermore, each assembly line uses a machine, is subject to a controller, and requires a given amount of assembly time.

The data-definition statements for these manufacturing operation objects are

```
DEF Assembly-Line (
  Machine: string;
  Controller: string;
  Assembly-Time: integer ).
DEF Manufacturing: MULTISET OF (
  Product-Code: string;
  Assembly-Line: VS;
  Quantity: integer ).
DEF Planned-Production: SEQUENCE OF (
  Day: string;
  Manufacturing: VS ).
```

Assembly-Line is a record with three elementary attributes. Manufacturing is a multiset that includes Assembly-Line as a complex attribute (VS — for *"vide supra"* — refers to a preceding object definition). Planned-Production is a sequence of days during which each Manufacturing object takes place. Figures 2b, 2c, and 2d represent the schemas of objects Assembly-Line, Manufacturing, and Planned-Production.

A simple instance of Planned-Production is:

```
Planned-Production <-  < ( 1.1.87 [
  ( P3 ( plank John 3 ) 30 )
  ( P5 ( plank John 3 ) 30 )
  ( P2 ( router David 5 ) 45 ) ] )
                       ( 1.2.87 [
  ( P2 ( plank John 3 ) 50 ) ] ) >.
```

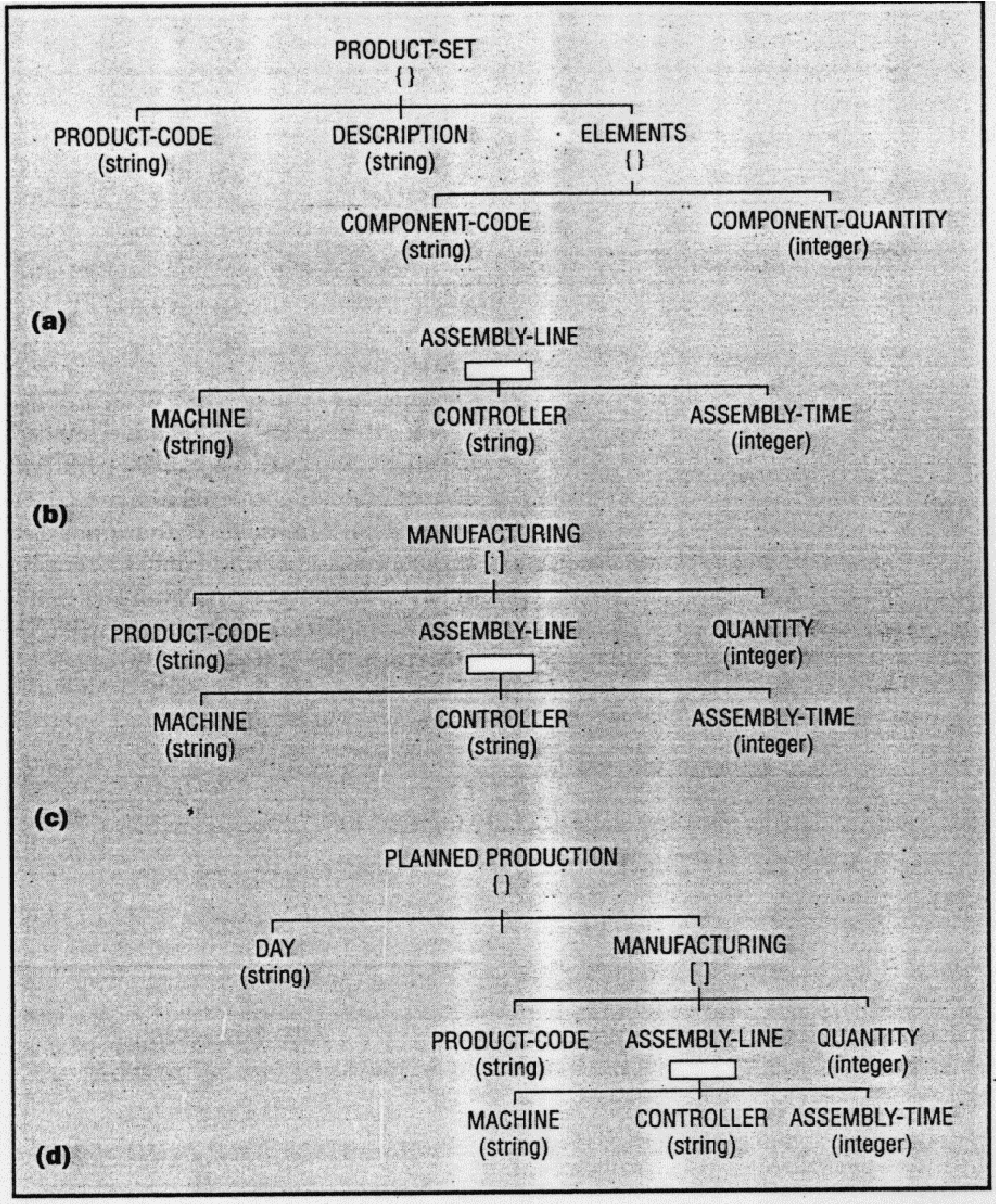


Figure 2. The Algres object schemas for **(a)** Product-Set, **(b)** Assembly-Line, **(c)** Manufacturing, and **(d)** Planned-Production represented as graphical tree structures.

Operations. The Algres-Prefix language includes:

- The classical algebraic operations of selection, projection, Cartesian product, union, difference, and derived operations like join, suitably extended to deal with type constructors and multilevel objects.
- The restructuring operations nest and unnest, which modify an object's structure.[2]
- Operations to evaluate tuple and aggregate functions over objects and subobjects.
- Operations to transform types.
- A closure operator, which iteratively evaluates an algebraic expression until the result reaches a fixpoint as a termination condition.[5]

Each Algres-Prefix operation has a prefix operation code. This code is either unary or binary and may have a specification part enclosed in brackets. Each operation constructs an object. Operations can be combined to form expressions.

For our example, we use the classic bill-of-material problem, which involves evaluating the number of elementary components in a product. It is well known that this problem cannot be solved by standard relational algebra.

First, you must determine the end products, which are those products that are not components of other products. In Algres, you program this in two steps.

The first step is to determine the set (Component-Set) of all existing components. You do this by writing an expression with the algebraic operations Project and Unnest. The Algres-Prefix instructions and the resulting object are

```
Component-Set <-
  UNNEST [Elements]
  PROJECT [Component-Code]
    Product-Set.

Component-Set =
  { ( P2 ) ( P3 ) ( P5 ) ( P6 ) }.
```

The first operation is a projection on the Component-Code attribute of the Product-Set object. A projection eliminates the attributes not mentioned in the specification part, while it preserves the hierarchical object structure. Figure 3a shows the resulting schema of the intermediate object from this projection, a set of sets. The

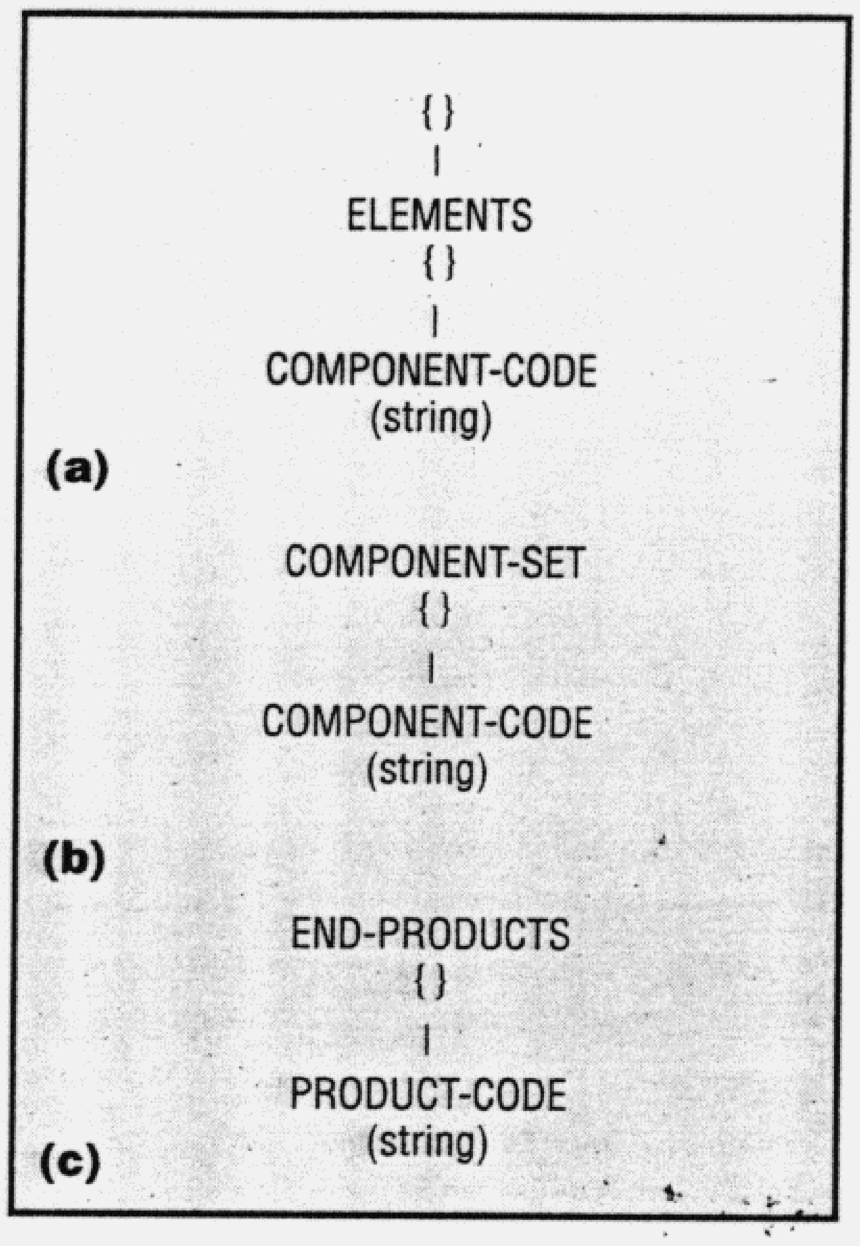


Figure 3. (a) Schema of an intermediate object that is the result of a projection on the Component-Code attribute of the Product-Set object; **(b)** result of Unnest operation on the set of sets, the new Component-Set object; and **(c)** result of projection over Product-Code, the object End-Products.

second operation, Unnest, transforms a set of sets into a simple set.[3,6] Figure 3b shows the result of Unnest, the new Component-Set object (new objects are created with the assignment operator).

The second step in determining the end products is to evaluate those products that are not components. To do this, you use the Selection operation, which eliminates the tuples of an object or subobject that do not satisfy a selection predicate.

```
End-Products <–
  PROJECT [Product-Code]
  SELECT [Product-Code NOT-IN
    Component-Set]
    Product-Set

End-Products = { ( P1 ) ( P4 ) }.
```

In this case, you test that Product-Code is not in Component-Set, which has already been evaluated and is used here as a constant object. A projection over Product-Code reveals the result; Figure 3c shows the schema of object End-Products. This example shows that the compiler can infer the schema — you need not declare it.

Now that you have determined the end products, you must face the more difficult task of evaluating the bill of material. For example, if P1 is made of four units of P4 and P4 is made of three units of P6, a bill-of-material computation evaluates that P1 is made of 12 units of P6. You must apply this construction recursively to obtain the desired result.

The Closure operator plays a fundamental role in evaluating bill-of-material. Closure is a unary operator that, in the simplest case, can be defined as

```
CLOSURE [Expression] Argument
```

where Closure is applied to a set/object

The classic bill-of-material problem, which involves evaluating the number of elementary components in a product, cannot be solved by standard relational algebra.

called Argument. Argument may also occur in the expression in the specification part; this is required to obtain the fixpoint of nonlinear expressions.

At each iteration, the Closure operation evaluates the expression over the current result, which initially is equal to Argument. The result is united to the current result, yielding the next operand of the expression. The new value of Argument is substituted to the operand as well as to all occurrences in the expression. The iteration terminates when the result remains identical for two consecutive iterations; the result of Closure is then the last value of Argument. Obviously, the correctness of Closure requires that the expression's result be type-compatible with Argument.

To understand the meaning of this operator, consider that if the expression is monotonic (with respect to set inclusion) in its Argument, the Closure operation evaluates the unique minimal fixpoint of the algebraic equation

$$X = \text{Expression}(X) \cup \text{Argument}$$

In this case, Closure terminates in a finite number of iterations.

However, termination is not guaranteed if the expression is not monotonic (for example, if it uses a set-difference operation). Therefore, to program with the Closure operator, you must first understand how to structure Argument so it leads to an iterative evaluation.

In this example, a convenient structure is a triple, $<X, Q, Y>$. Each component X is made up of Q instances of component Y. We chose the End-Products set as the initial Argument value, initialized so each end product has the quantity 1 — itself. We obtained this by applying the operation Tuplextend twice to the End-Products object, which has been evaluated previously.

```
Argument <–
  TUPLEXTEND [Arg-Component:=
    Arg-Product]
  TUPLEXTEND [Arg-Quantity:=1]
  RENAME [Arg-Product:= Product-Code]
    End-Products.

Argument = {  ( P1,1,P1 )
              ( P4,1,P4 ) }.
```

Tuplextend extends an object's schema and instance; its specification indicates the new attribute's name and the function required to evaluate it. We used the Rename operation here to change the name of one elementary attribute.

Now you can apply the Closure operator to Argument to produce the bill of material. The difficult part is to generate a suitable expression whose result is type-compatible with Argument. The expression we used to do this contains the Unnest operation on Elements applied to the Product-Set object (Unnest transforms a set of sets into a set). The resulting object is joined to Argument; a join is simply a selection over a Cartesian product, as in standard relational algebra. Then a Tuplupdate operation evaluates, as an update to the Arg-Quantity attributes, the quantities of each subcomponent in a given component. Finally, the expression projects the result to obtain triples that are type-compatible with Argument.

```
Bill-of-material <-
  CLOSURE
    [ PROJECT [Arg-Product,
      Arg-Quantity, Component-Code]
     TUPLUPDATE [Arg-Quantity:=
      Arg-Quantity * Component- Quantity]
     JOIN [Product-code=Arg-Component]
       UNNEST [Elements] Product-Set ]
         Argument.

         Bill-of-Materials = { ( P1, 1, P1 )
                               ( P1, 4, P2 )
                               ( P1, 8, P3 )
                               ( P4, 1, P4 )
                               ( P4, 4, P2 )
                               ( P4, 1, P5 )
                               ( P4, 4, P3 )
                               ( P4, 2, P6 ) }.
```

In this expression, the Unnest operation applied to Product-Set in the iterative Closure expression can be moved out of the loop because it does not change at each iteration. You can do this by preevaluating an Unnested-Product-Set object:

```
Unnested-Product-Set <-
  UNNEST [Elements] Product-Set.

Unnested-Product-Set ={
  ( P1, Bench, P2, 4 )
  ( P1, Bench, P3, 8 )
  ( P2, Leg, NULL , NULL )
  ( P3, Stick, NULL, NULL )
  ( P4, Table, P2, 4 )
  ( P4, Table, P5, 1 )
  ( P5, Top, P3, 4 )
  ( P5, Top, P6, 2 )
  ( P6, Drawer, NULL, NULL ) }.
```

Then you rewrite the Closure expression so you join Unnested-Product-Set directly to Argument. This example shows that unnesting empty sets generates null elements.

Other features. Our production example also illustrates other Algres-Prefix features. For example, we used the Aggrfun operation to evaluate all assembly times on each day of production. Aggrfun does this by evaluating the aggregate function Sum over Assembly-Time in the Planned-Production object:

```
Assembly-Time1 <-
  PROJECT [Day,Sum-Assembly-Time]
  TUPLEXTEND [Sum-Assembly-Times:=
    AGGRFUN
      [SUM/(Assembly-Time*Quantity)]]
  Planned-Production.

Assembly-Time1 =
{ ( 1.1.87, 405 ) (1.2.87, 150 ) }
```

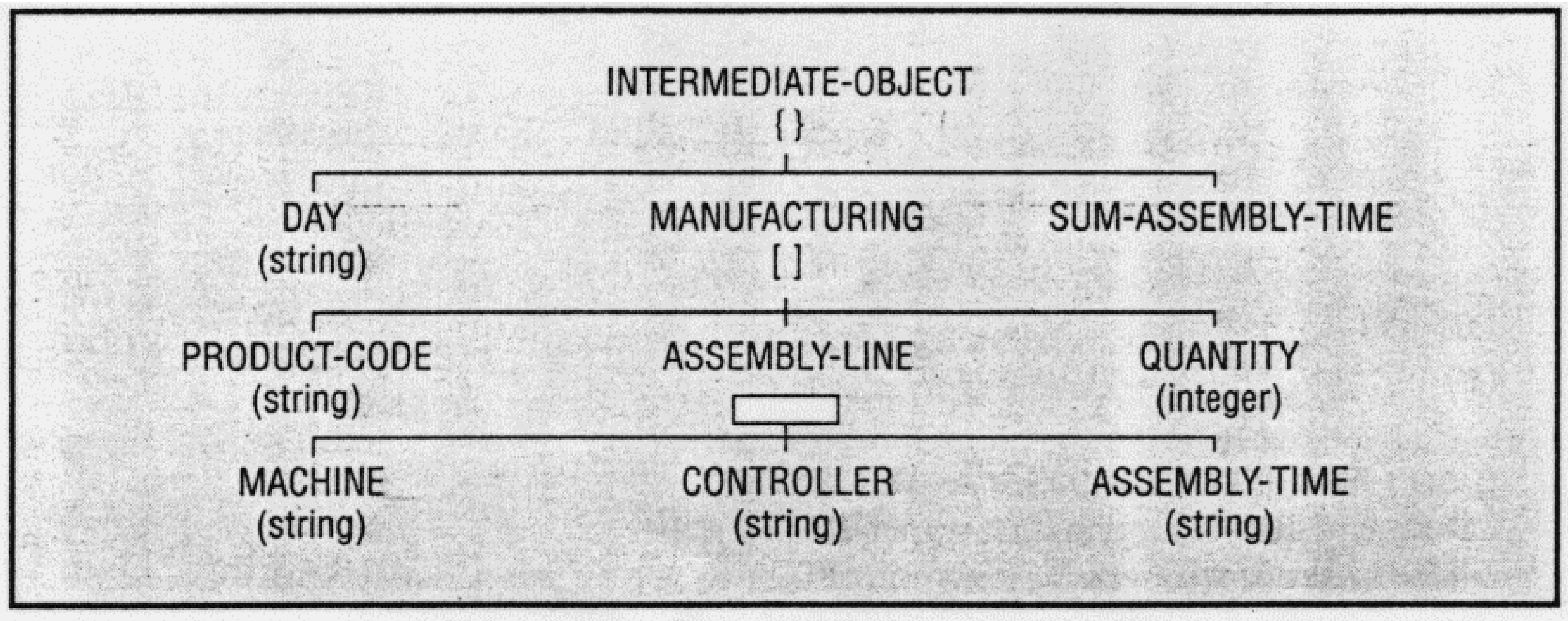


Figure 4. Schema of the intermediate object Tuplextend, showing a new Sum-Assembly-Time attribute to store the results of the aggregate function.

Figure 4 shows the schema of the intermediate object Tuplextend [Sum-Assembly-Times:= Aggrfun/Sum/(Assembly-Time*Quantity)]] Planned-Production. This schema shows that a new attribute, Sum-Assembly-Time, has been added to the hierarchical level immediately above the attributes Assembly-Time and Quantity. This attribute lets us store the result of the aggregate function.

Next, you transform the Assembly-Time1 object into a sequence, sorted by day. To do this, we used the type-coercion operator Makeseq, which generates a sequence; its specification indicates an ascending order:

```
Assembly-Time2 <-
  MAKESEQ [ASC Day] Assembly-Time1

Assembly-Time2 =
  < ( 1.1.87, 405 ) (1.2.87, 150 )>
```

RA abstract machine

Algres objects are stored, both in mass memory and in main memory, as conventional relations. For example, the RA machine translates the Algres object shown in Figure 5a into two RA relations shown in Figure 5b.

The RA machine maps Algres objects to RA relations by mapping each set, multiset, and sequence object or subobject to a distinct relation. Subobjects are linked to their parent objects through attribute pairs, called pointer and pointed respectively. For example, if the Family object equals 01 and the Car object equals 02, the main-memory representation of 01 requires a pointer attribute and the representation of 02 requires a pointed attribute to model the one-to-many relationship between the tuples of 01 and

```
DEF Family : SET OF (  Name: string,
                       Age: integer,
                       Cars: MULTISET OF (Brand: string)).

Family <-{    ( john       46   [ ( buick ) ( ford ) ( ford ) ] )
              ( mary       45   [ ( GM ) ] )
              ( david      17   [ ( datsun ) ] )
              ( deborah    15   [] ) }.
```
(a)

```
Family    (Name: string, Age: integer, Car: pointer).
Car       (Family: pointed, Occ: integer, Brand: string).
```

FAMILY	Name	Age	Car-ptr	CAR	Family-ptd	Occ	Brand
	john	46	1		1	1	buick
	mary	45	2		1	2	ford
	david	17	3		2	1	GM
	deborah	15	0		3	1	datsun

(b)

Figure 5. (a) Algres object and **(b)** the results of its translation by the RA machine into RA relations.

```
DEF Library: SET OF (          Subject: string,
                               Books: SET OF (  First-Author: string,
                                                Title: string,
                                                Year: integer)).

DEF All-Book-Authors: SET OF(  Books: SET OF (  Author: string))
```
(a)

Algres-Name	Algres-Id
Library	1
All-Book-Authors	2

(b)

Algres-Id	RA-Id
1	1
1	2
2	1
2	2

(c)

Algres-Id	RA-Id	AT-Id	AT-Name	Type	Ref	Offset	Mark
1	1	1	Subject	string	NULL	1	NULL
1	1	2	Books	pointer	2	2	NULL
1	2	1	Subject	pointed	1	1	NULL
1	2	2	First-Author	string	NULL	2	NULL
1	2	3	Title	string	NULL	3	NULL
1	2	4	Year	integer	NULL	4	NULL
2	1	1	Books	pointer	2	1	NULL
2	2	1	All-Book-Authors	pointed	1	1	NULL
2	2	2	Author	string	NULL	2	NULL

(d)

Figure 6. (a) Algres objects Library and All-Book-Authors and their **(b)** Algres-Descriptor, **(c)** RA-Descriptor, and **(d)** AT-Descriptor directory relations.

02. In implementating pointer and pointed attributes, we used explicit (integer) values, but we are considering using physical pointers instead to improve efficiency.

In the example in Figure 5, the names of RA relations and attributes are symbolic, but the Algres-Prefix-to-RA translator and RA machine use the actual numeric identifiers, which were generated automatically. You can store these RA relations in the Informix mass-memory database without conversion by using two system-controlled attributes. To represent multisets, you store each identical tuple only once and use the system-controlled attribute Occ to give the number of occurrences of identical elements within the multiset. Similarly, instead of representing sequences by the physical ordering of tuples, sequence objects have a system-controlled attribute Pos, which gives each tuple's position in the sequence.

RA machine organization. The RA machine is a hierarchy of three virtual machines: the debugger, the command interpreter, and the memory-management unit.

The debugger provides step-by-step execution, tracing, and visualization of intermediate results. The command interpreter supplies procedures to visualize Algres objects and to execute RA programs. The memory-management unit implements memory management for relations, tuples, and attributes, while hiding their physical organization.

The memory-management unit stores each Algres object as a sequence of contiguous RA relations, each RA relation as a sequence of contiguous tuples, and each tuple as a sequence of contiguous attribute values. It implements all memory blocks as arrays of bytes, which means that all database values are represented internally as byte strings and are type-converted for presentation and computation. The type-conversion facilities of C (the implementation language) ease these conversions. Finally, because the memory required is not known in advance, fixed-length memory blocks are allocate dynamically.

In main memory, RA relations ar stored in two opposite instance stacks which grow against each other. The firs stack is segmented into permanent an temporary areas. The permanent are stores objects that must be retained throughout an Algres session. Permanen objects include objects imported from the mass-memory resident database and objects obtained by assignment instructions in Algres-Prefix. The temporary area stores objects that can be discarded as soon as they have been used. Temporary objects contain the intermediate results of a computation. The second stack contains only temporal objects.

We chose to organize the main memory into two opposite stacks to avoid fragmenting the storage area, thus reducing the need for expensive garbage collection. We developed a special algorithm for stack management that guarantees that, whenever we compute an algebraic operation involving one or two Algres objects as operands, all the operands are allocated on the same stack and the result is written to the opposite stack. The algorithm also guarantees that permanent objects that are evaluated by any expression are always allocated to the first stack.

This means that you can include permanent objects in the permanent area of the first stack simply by changing the value of the stack pointer that separates the permanent and temporary areas. In this way, the permanent area grows by enclosing permanent objects in the order in which they are evaluated. On the other hand, temporary areas can be discarded (again by manipulating stack pointers) as soon as their objects have been used in the computation because they cannot be referenced by other parts of the program.

Some RA operations that are applied to temporary results are performed by Mark attributes, which are invisible to users. For example, to perform selections and projections over temporary objects you mark the selected tuples or projected attributes. This lets you execute some operations directly on the stack where the operand is allocated without copying objects from one stack to the other.

```
DEF Library: SET OF (       Subject: string,
                            Books: SET OF (    First-Author: string,
                                               Title: string,
                                               Year: integer)).

Library <– { ( English Literature, {      (Shaw,Pygmalion,1914)
                                          (Shakespeare,Hamlet,1601)
                                          (Dickens,Oliver Twist,1838)
                                          (Shakespeare,King Lear,1606)
                                          (Shakespeare,Macbeth,1606) } ),
             (American Literature, {      (Hemingway,A Farewell to Arms,1929)
                                          (Steinbeck,Of Mice and Men,1937)
                                          (Hemingway,For Whom the Bell Tolls,1940)
                                          (Hemingway,The Old Man and the Sea,1952)
                                          (Bellow,Humboldt's Gift,1975)} )}

DEF All-Book-Authors: SET OF(   Books: SET OF (   Author:string ))

All-Book-Authors = {      (     {(Shaw) (Shakespeare) (Dickens)} )
                          (     {(Hemingway) (Steinbeck) (Bellow)} ) }
```

Figure 7. Algres object definitions for Library and All-Book-Authors.

Directory. The RA machine maintains a directory that describes the mapping of an Algres object to a RA relation. Each mapping consists of three RA relations:

```
Algres-Descriptor
  ( Algres-Name, Algres-Id )
RA-Descriptor
  ( Algres-Id, RA-Id, Stack,
    Start-Addr, End-Addr, Tuple-Length,
    Cardinality)
AT-Descriptor
  ( Algres-Id, RA-Id, AT-Id, AT-Name,
    Type, Ref, Offset, Mark)
```

The first, Algres-Descriptor, contains the mapping from Algres-Prefix names to internal identifiers.

The second, RA-Descriptor, contains one tuple for each RA relation. It indicates the identifier of the RA relation, the identifier of the corresponding Algres object, the stack number, the start and end addresses in that stack, and the length and number of tuples.

The third, AT-Descriptor, contains one tuple for each attribute in an RA relation. It indicates the Algres and RA identifiers, the attribute identifier, the name, the type, a reference to RA relations (used if the attribute is of type pointer or pointed), the offset of the attribute within the tuple, and a Mark, which in temporary objects indicates whether the attribute has been projected.

Candidate keys for Algres-Descriptor are both Algres-Name and Algres-Id, the unique key of RA-Descriptor is the pair <Algres-Id, RA-Id>, and the unique key of AT-Descriptor is the triple <Algres-Id, RA-Id, AT-Id>. The RA machine implements directory relations as arrays of records, which are stored in two opposite directory stacks. These stacks are managed like the data stacks.

For example, the Algres objects Library and All-Book-Authors, defined in Figure 6a, would have the directory relations shown in Figure 6b, 6c, and 6d.

RA object code. RA object-code instructions, produced by the translator, have this format:

```
<operating code>  <operand1>
                  <operand2> <result>
```

where <operating code> identifies an RA operation and <operand1>, <operand2>, and <result> indicate either an RA relation or an attribute of an RA relation (some instructions have only one operand). A translator transforms each Algres-Prefix operation into a sequence of RA instructions:

```
ENTER Stack-Number
< instructions to generate the schema
  of the result object>
< instructions to generate the instance
  of the result object>
END Num-Rel
```

where Stack-Number indicates in which of the two stacks the result must be written, and Num-Rel is the number of operand relations to be deleted from the opposite stack.

An example of how an Algres-Prefix operation is executed is a projection

```
PROJECT [A1...Am] Object
```

where *Ai* are names of attributes, possibly at different hierarchical levels of Object. In the execution, first the list *A*1 to *Am* is transformed into a larger list, *A*1 to *An*, which includes the ancestors of all attributes in the list *A*1 to *Am*, even if they are not mentioned in the list. You must include them because you must mantain all intermediate levels between projected attributes and the root in the schema of the resulting Algres object.

Next, to perform the projection you access the RA relations in suitable order and mark attributes that do not appear in the list *A*1 to *An*. If an RA relation corresponds to a set, you first detect and delete replicated tuples generated by the projection; this is in fact achieved directly by using the Mark attribute. If an RA relation corresponds to a multiset, you modify the Occ attribute. If an RA relation corresponds to a sequence, you do not perform an additional operation. These operations are executed on the current stack; the marked tuples and attributes of RA relations are then copied to the other stack.

Consider again the Algres Library object and the Algres-Prefix projection:

```
All-Book-Authors <–
  PROJECT [First-Author] Library
```

The Library and All-Book-Authors objects (schemas and instances) are shown in Figure 7.

The RA code obtained by translating the above Algres-Prefix projection is

```
ENTER 1
  PJSEQ 1 2
    FIELD 1 2 1
    FIELD 1 2 2
  ENDPJ
  REMDUP 1 2
  COPY 1 1 2 1
  COPY 1 2 2 2
END 2
```

This code assumes that the result is copied to the first stack, while operations can be performed directly on the operand relation on the second stack.

- Enter is the heading of an Algres-Prefix operation and its parameter (1 in this example) identifies the stack on which the result must be written.
- End is the end of an Algres-Prefix operation and its parameter gives the number of temporary RA relations to be deleted from the stack.

```
SET Product-set
  ITEM Product-Code: STRING
  ITEM Description:  STRING
  SET  Elements
      ITEM Component-code: STRING
      ITEM Component-quantity: INTEGER
  END
END ;

RECORD Assembly_line
    ITEM Controller: STRING
    ITEM Machine : STRING
END ;

MULTISET Manufacturing
      ITEM  Product-Code: STRING
      RECORD Assembly_line: VS
      ITEM Quantity: INTEGER
END ;

LIST Planned_Production
    ITEM day: STRING
    MULTISET Manufacturing: VS
END ;
```

Figure 8. AlgresQL program for computing the bill-of-material problem.

• Pjseq describes the schema resulting from the projection by specifying the attributes to be preserved. Its parameter indicates the RA relation on which the projection is performed, indicated by the pair <Algres-Id, RA-Id>. Each projected attribute is introduced by the operating code Field, and is indicated by the triple <Algres-Id, RA-Id, AT-Id>.

• Endpj concludes a sequence of Fields, indicates the attribute list is finished, and executes projection.

• Remdup operates on RA relations by removing duplicate tuples after projections.

• Copy is a two-operand operation that copies one RA relation into another one. In this case, it transfers to the opposite stack the result of the projection operation performed on a temporary object (only marked attributes are copied). Each operand is a pair, <Algres-Id, RA-Id>.

User languages

The Algres programming environment is multiparadigm. You can interact with the system with two languages, Algres Query Language and Datalog.

AlgresQL extends Structured Query Language to deal with all features of the Algres data model in a query-language programming style. AlgresQL is computationally equivalent to Algres-Prefix, but is easier for those accustomed to SQL to use and read.

Datalog, while it deals only with a subset of the Algres data model's features, lets you program in a logic-programming style. The computational power of Datalog is included in that of Algres-Prefix. You can write part of your program in the programming style you prefer and translate the AlgresQL and Datalog portions into Algres-Prefix before execution.

```
Component_Set :=
    UNNEST Elements
    FROM (SELECT Component_Code
        FROM Product_set) ;

End_Products :=
          SELECT Product_Code
          FROM Product_Set
          WHERE [Product_Code NOT_IN Component_Set] ;

Argument :=
        SELECT Arg_Product = RENAME Product_Code,
        [NEW] Arg_quantity = 1,
        [NEW] Arg_component = Arg_Product
        FROM End_Products;

Bill_of_Material :=
  CLOSE Argument ON
    ( SELECT Arg_product,
          [NEW] New_quantity = Arg_quant * Comp-quantity,
          Component_Code
      FROM (UNNEST Elements
          FROM Product_set),
          Argument
      WHERE Product_Code = Argument_Component );
```

Figure 9. AlgresQL program to build the Component-Set, End-Products, Argument, and Bill-of-Material objects progressively.

AlgresQL. To SQL we have added language constructs to express explicity the Closure, Nest, and Unnest operators and to deal with multisets and sequences (including type transformation).

We extended SQL's basic block structure, which is based on Select-From-Where clauses, by supporting the use of block structures recursively within the Select and From clauses. You can nest queries in both Select and From clauses.

The definition of Algres-Prefix operators dictated our SQL extensions, which are comparable to those proposed in the AIM-II project at IBM Heidelberg, West Germany,[2] and by Hans Sheck and Mark Scholl.[4] The AIM-II language, however, supports neither the multiset and list data types nor a closure operator; Shek and Scholl's language does not have a closure operator.

Like SQL, AlgresQL is composed of data-definition, query, and data-manipulation statements. In addition, database-interface statements let your program interact with the external database.

Figure 8 shows the AlgresQL program for computing the bill-of-materials problem. Data-definition statements define the Product-Set, Assembly-Line, Manufacturing, and Planned-Production objects. These AlgresQL definitions are equivalent to Algres-Prefix's data-definition instructions.

You then build the Component-Set, End-Products, Argument, and Bill-of-Material objects progressively, using the same strategy as in Algres-Prefix, as shown in Figure 9. In fact, each AlgresQL statement can be easily translated into its corresponding Algres-Prefix expression.

Datalog. Datalog is a logic language designed specifically to query a relational database. A Datalog program is simply a collection of Horn clauses — logic formulas with a particular structure. Horn clauses are one of three types:

• Facts are assertions about the real world. Conventional relations are collections of facts with the same structure.

• Rules express the behavior of the real world. Rules have a head (also called a conclusion) and a body (also called the premise). Intuitively, a rule means that if the premise holds then the conclusion holds.

• Goals express queries on the real world. A Datalog processor uses facts and rules to answer all the queries of a goal.

Syntactically, Datalog is similar to Prolog. However, it is much simpler than Prolog because it does not include special predicates (like Cut) and function symbols and because it partially restricts negation. Datalog's semantics is set-oriented. The retrieval of all possible answers to a goal uses a breadth-search strategy (instead of Prolog's depth search). Further, the result of a query is independent of the order of clauses in the program or of predicates in clauses (Prolog is order-sensitive).

Datalog programs formulate logic queries over potentially large collections of facts that are stored through conventional relations. Returning to the bill-of-material problem, in Datalog you would describe the product-component relationship simply as a set because Datalog does not deal with complex objects.

```
DEF Product-Set: SET OF (
  Product-Code: string;
  Component-Code: string;
  Quantity: integer).
```

Computing the bill-of-materials problem in Datalog is similar to doing so in Algres-Prefix and AlgresQL. You need rules to evaluate

• the components set (products whose code appears as the second attribute of the Product-Set object),

• the End-Products set (products whose code does not appear in the set of component codes),

• the start (nonrecursive) definition of the Bill-of-Material object for each End-Product, stating that each end product consists of itself in quantity 1, and

• the recursive definition of Bill-of-Material for components belonging to an arbitrary depth level in the part-component tree.

Algres provides a novel approach to very high-level programming and an open laboratory for the integration of relational and logical techniques.

The Datalog rules to model this problem are

```
Component(X):– Product-Set(_,X,_).
                              /* rule 1 */
End-Product(X):– Product-Set(X,_,_),
  not Component(X).     /* rule 2 */
Bill-Of-Material(X,1,X) :– End-Product(X)
                              /* rule 3 */
Bill-Of-Material(X,Y,Z) :–
  Bill-Of-Material(X,Q1,W),
  Product-Set(W,Z,Q2),
  Times(Q1,Q2,Y).          /* rule 4 */
```

Rule 1 simply says that a Component is any part appearing as second argument of the predicate Product-Set; this predicate is assumed to be stored in mass memory.

Rule 2 says that a finite product is any product that does not also appear as a component. This use of negation is acceptable because negation is safe.

Rule 3 evaluates the nonrecursive part of a bill of material, by stating that each finite product is made of one unit of itself.

Rule 4 evaluates the recursive part of a bill of material. It asserts that — if X is made of $Q1$ components W and W itself is made of $Q2$ components Z — X is made of $Q1 * Q2$ components Z.

It is easy to translate Datalog programs with one recursive predicate to a standard relational algebra that has been enriched with a fixpoint operator.[9] Thus, relational algebra extended by the Closure operation is a suitable environment to execute Datalog queries. In fact, we do not use much of the expressive power of the Algres data model. We are considering Datalog extensions to deal with complex objects, as LDL-1[7] and similar languages do.

However, it is more difficult to translate rules with mutually recursive predicates,[8] and we are now evaluating extensions to the Closure operation to do this efficiently.

We believe that Algres provides a novel approach to very high-level programming and an open laboratory for the integration of relational and logical techniques. It also provides an alternative to Lisp and Prolog runtime environments.

All the components of the Algres environment are operational; the system is written entirely in C. We are now expanding the graphical interface to handle Algres expressions and extending Datalog as a query language. We will make a distribution kit of the Algres system available for nonprofit use at a nominal fee.

More work is needed to develop a design methodology and an effective style for Algres-based developments.

We are using Algres in the ESPRIT Phase 2 Stretch project (Project 2443) to rapidly prototype an extended database system — Logres — that integrates the object-oriented data-modeling paradigm and the rule-based approach for the specification of queries and updates.[10] ❖

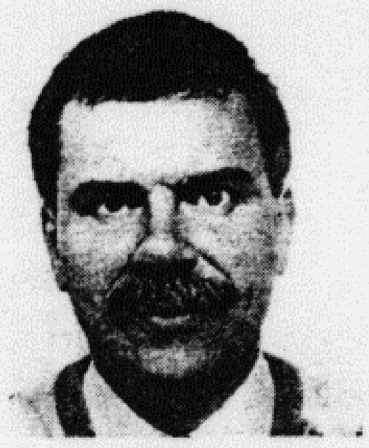
Acknowledgments

We thank everyone who has contributed to the Algres project, especially Alberto Daprá, Stefano Gatti, and O. Zaffaroni of TXT; Georg Gottlob of the University of Vienna; Letizia Tanca of Politecnico di Milano; and M. Antonetti, S. Aliverti, G. Bossi, Filippo Cacace, F. Cesani, P. Dotti, Marco Ferrario, M. Giudici, D. Milani, P. Nasi, A. Pastori, A. Patriarca, M. Patriarca, G. Pisani, M. Riva, and P. Vagnozzi. We also thank the anonymous referees for their comments.

The Algres Project is mainly supported by ESPRIT Project 432 (Meteor). We also acknowledge partial support of the Italian Research Council, the Italian Ministry of Education's 40% research program, and the Rank Xerox University Grant Program.

References

1. S. Ceri et al., "Software Prototyping by Relational Techniques: Experiences with Program Construction Systems," *IEEE Trans. Software Eng.*, Nov. 1988, pp. 1,597-1,609.
2. S. Abiteboul and N. Bidoit, "Non-First-Normal Form Relations to Represent Hierarchically Organized Data," *Proc. Third ACM SIGMOD-SIGACT Symp. Princ. Database Systems*, ACM, New York, 1984.
3. P. Dadam et al., "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies," *Proc. ACM SIGMOD*, ACM, New York, 1986.
4. H.J. Shek and M.H. Scholl, "The Relational Model with Relation-Valued Attributes," *Information Systems*, 1986.
5. A. Aho and J. Ullman, "Universalities of Data-Retrieval Languages," *Proc. Sixth ACM Symp. Princ. Programming Languages*, ACM, New York, 1979.
6. L. Lavazza and S. Crespi-Reghizzi, "Algebraic ADT Specifications of an Extended Relational Algebra and Their Conversion into a Running Prototype," in *Workshop Algebraic Methods, Theory, Tools, and Applications*, W. Bergstra and N. Wirsing, eds., Springer-Verlag, New York, 1987.
7. C. Beeri et al., "Sets and Negation in Logic Data Language," *Proc. Sixth ACM SIGMOD-SIGACT Symp. Princ. Database Systems*, ACM, New York, 1986.
8. S. Ceri and L. Tanca, "Optimization of Systems of Algebraic Equations for Evaluating Datalog Queries," *Proc. Very Large Databases*, Morgan Kaufmmann, Brighton, England, 1987, pp. 31-41.
9. S. Ceri, G. Gottlob, and L. Tanca, "Relational Databases and Logic Programming," *Surveys in Computer Science*, Springer-Verlag, New York, 1990 (to appear).
10. F. Cacace et al., "Integrating Object-Oriented Data Modeling with a Rule-Based Programming Paradigm," in *Proc. ACM SIGMOD*, ACM, New York, 1990.



Stefano Ceri is professor of computer science at the Dipartimento di Matematica Pura ed Applicata, Universitá di Modena, Italy, and a visiting professor at Stanford University. His research interests include distributed databases, database design, and the use of databases in software engineering and logic programming.

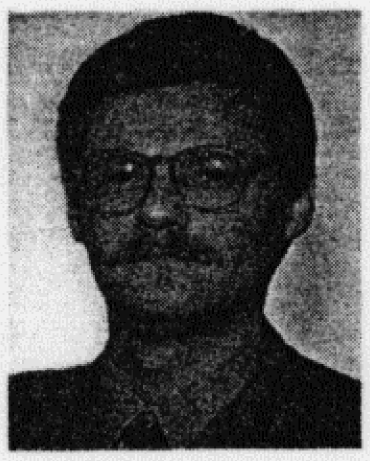
Ceri received an MS in computer science from Stanford University and a doctor degree in electrical engineering (computer science) from the Politechnico di Milano.

Stefano Crespi-Reghizzi is professor of computer science at the Dipartmento di Elettronica, Politecnico di Milano and the coordinator of the doctoral program in electronics and computer science. His research interests include formal languages, automata, semantics, and the convergence of artificial intelligence, databases, and software-engineering technologies.

Crespi-Reghizzi received a doctor degree in electrical engineering (computer science) from Politecnico di Milano and a PhD in computer science from the University of California at Los Angeles.

Gianfranco Lamperti is a member of the research staff at TXT SpA. His research interests include the study and development of software-engineering activities and extensions of relational algebra.

Lamperti received a doctor degree in electrical engineering (computer science) from the Politecnico di Milano.



Luigi A. Lavazza is a member of the research staff at TXT SpA. His research interests include the study and development of software-engineering activities and extensions of relational algebra and logic programming.

Lavazza received a doctor degree in electrical engineering (computer science) from the Politecnico di Milano.

Roberto Zicari is an associate professor of computer science at the Politecnico di Milano. His research interests include the definition of advanced database systems for complex applications, database theory, and office automation.

Zicari received a doctor degree in electrical engineering (computer science) from Politecnico di Milano.

Address questions about this article to Zicari at Politecnico di Milano, Dipartimento di Elettronica, Piazza Leonardo da Vinci 32, I-20133 Milano, Italy; Bitnet relett15@imipoli.bitnet