Software Prototyping by Relational Techniques: Experiences with Program Construction Systems

STEFANO CERI, STEFANO CRESPI-REGHIZZI, ANDREA DI MAIO, MEMBER, IEEE, AND LUIGI A. LAVAZZA

Abstract—**A method for designing and prototyping program construction systems using relational databases is presented. Relations are the only data structures used inside the systems and for interfaces; programs extensively use relational languages, in particular relational algebra. Two large projects are described. The Ada Relational Translator (ART) is an experimental compiler-interpreter for Ada in which all subsystems, including the parser, semantic analyzer, interpreter, kernel, and debugger, use relations as their only data structure; the relational approach has been pushed to the utmost to achieve fast prototyping in a student environment. Multi-Micro Line (MML) is a tool set for constructing programs for multimicroprocessors' targets, in which relations are used for allocation and configuration control. Both experiences confirm the validity of the approach for managing teamwork in evolving projects, identify areas where this approach is appropriate, and raise critical issues.**

Index Terms—**Ada, program construction environments, rapid prototyping, relational programming, target computer description.**

I. Introduction

RELATIONAL programming is a novel approach to the development and rapid prototyping of software applications [6], [26], [30], [31]. The paradigm of this approach is quite simple: data structures are described using the relational model [9]; algorithms are transformations from an initial set of relations to a final one, expressed using relational languages [38], possibly extended, as later argued.

The rationale of this approach is the following.

- Consolidated methods for structuring and normalizing relations can be used in the design of data [25].
- Subsystems become homogeneous in their data structures.
- All interfaces between subsystems are clearly and uniformly defined.
- Data of interest for a subsystem can be extracted as subschema of the database (DB), so that changes and dependences are easy to recognize.
- Relational algebra (or calculus) provides synthetic expressive primitives for operating on all data structures.
- A programming discipline is established, which emphasizes the need for designing data structures before algorithms.

The relational approach was applied to two large software projects, which serve as significant testbeds: a compiler-interpreter (the *A*da *R*elational *T*ranslator, or ART) for the Ada programming language, and a tool set (*M*ulti-*M*icro *L*ine, or MML) for constructing programs for multimicroprocessors' targets. In ART, we relied on enthusiastic but unprofessional students with a very fast turnaround; relational programming has been useful for minimizing the risk of throwing away large mysterious pieces of code without committing the project leader to continuous direct intervention. Similarly, relational programming was exploited in MML to coordinate software developments by distant partners in a national pilot project.

However, a traditional DB system would not always suit our peculiar requirements. In software engineering applications, relations are not defined once and for all by a DB manager and later used, but rather, each subsystem defines and uses a subset of the relations much more dynamically. Furthermore, relations are expected to be of small or moderate size, so that they can be kept in central memory for faster operation. We have designed and implemented an experimental relational data manager to meet these requirements. Other extensions to conventional programming languages have been developed in order to deal with relations, most noticeably Pascal-R [36], Modula-R [39], and DBPL [16]. These three systems constitute an evolution of the same basic approach: to introduce the new type *relation* (with the power of relational calculus or more) within an imperative programming language. Results have been encouraging in several nonconventional applications, including the support of programming descriptions [39].

The use in the project ART of relational queries within computationally intensive loops made us prefer a core, rather than a disk, resident data manager. Other DB sys-

Manuscript received June 15, 1986; revised October 20, 1987. This work was supported in part by Ministero Pubblica Istruzione and in part by CSISEI-CNR. Subsequent developments are also supported by the ESPRIT project 432 "Meteor: An Integrated Formal Approach to Industrial Software Development," and by Rank Xerox University Grant Program.

S. Ceri was with the Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy. He is now with the Dipartimento di Matematica, Università di Modena, 40100 Modena, Italy.

S. Crespi-Reghizzi is with the Dipartimento di Elettronica, Politecnico di Milano, 20133 Milan, Italy.

A. Di Maio and L. A. Lavazza are with TXT Software e Telematica SpA, 20122 Milan, Italy.

IEEE Log Number 8823663.

0098-5589/88/1100-1597$01.00 © 1988 IEEE

tems have been developed specifically for managing software engineering data: IDL [12], DAMOKLES [15], OMEGA [29], and PDB [32]. Their application is mostly to support "programming in the large" by building systems which belong, in our classification, to the *version management subschema* of a program construction system (see Section II).

In addition, software engineering data are more complex than objects in traditional DB applications, and their manipulation sometimes requires recursive queries. Extensions to the relational model and languages are being developed to enhance its expressive power [8], [10], [26], [28], [34]; in the conclusions, we discuss the convenience of extending relational programming in this direction. Interestingly, the need for relational representations has independently arisen in interactive programming environments [22] to supplement the attributed syntax trees used to represent the result of semantic analysis. That paper demonstrates the power and limitations of relational operations and introduces a hybrid system which shows several advantages with respect to a purely relational one, or to a purely attribute-grammar-based approach.

We first propose in Section II a classification of data used by program construction systems in terms of DB relations; relations are grouped into subschemas, and we discuss the timing of their creation, load, and usage. In Section III, we present the ART project, and in Section IV, we present the MML system. In the conclusions, we evaluate the experience and indicate directions for future work.

II. Data Classification for a Program Construction System

Since both applications are program construction systems, we propose a classification of data from the viewpoint of their definition and usage in a *p*rogram *c*onstruction DB (PCDB). The purpose of this classification is also to introduce a terminology for the relevant information structures of a PCDB; extensive examples of some significant cases will be given in Sections III and IV.

At a first classification criterion, we use subschemas[1]; the PCDB includes four subschemas (see Fig. 1).

1) The *version management subschema* describes the development and usage of each software module or document (including intermodule dependences) in the so-called "software life cycle."

2) The *program translation subschema* describes data required for translating a source program into an executable form (typically, the translation involves compilation, linking, code generation, and optimization).

3) The *program execution subschema* describes data used in program execution (including run-time libraries, run-time debugging support, etc.).

4) The *target configuration subschema* describes the target system to which an executable program can be tailored.

[1]A schema describes all the data of the DB; a subschema contains a subset of them, which are used by a well-distinguished set of applications.

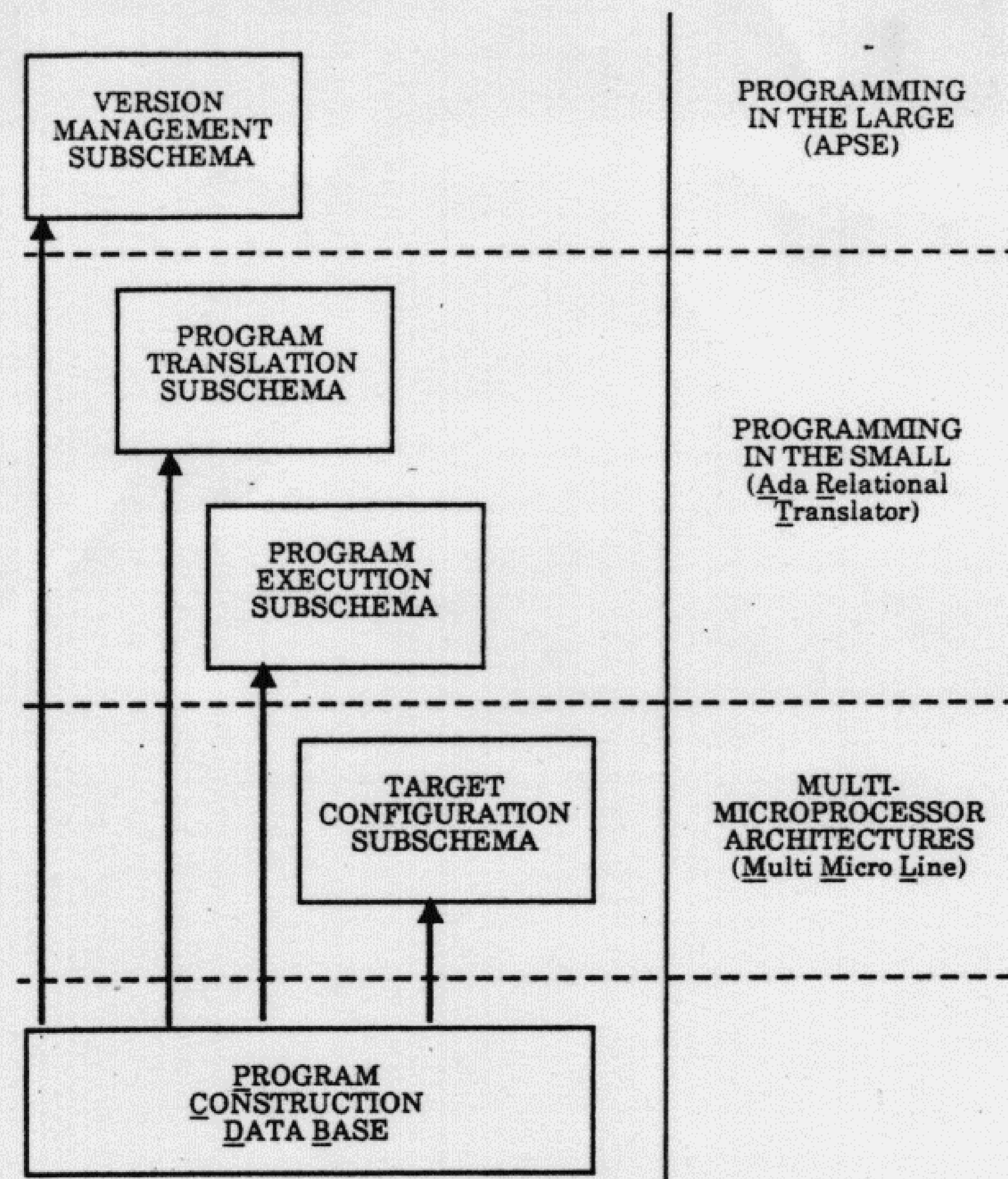


Fig. 1. Subschemas of the PCDB.

The four subschemas are not intended as a partition of the PCDB; rather, they represent different viewpoints from which the PCDB can be seen, corresponding to different uses of the PCDB. In fact, a strong point of the DB approach is the possibility of sharing information among different cooperating subsystems. For instance, a debugger needs information from subschemas 1)–3).

The need of a DB for managing software documentation and modules [subschema 1)] is generally recognized, as witnessed by current Ada programming support environment [2] efforts or the Portable Common Tool Environment (PCTE), an EEC-supported ESPRIT project. Subschema 1) supports activities which are typically referenced as "programming in the large." Extending the DB approach to subschemas 2)–4) represents instead a rather substantial innovation. Subschemas 2) and 3) contain data which are typically used for activities referenced as "language processing." At the very extreme, subschema 3) has to do with the low-level run-time structures accessed by interpreters and kernels. Subschema 4) has less generality, as it is required for tailoring a compiled program to greatly varying target configurations; this happens, for instance, with multimicroprocessor targets, as in [37]. We have widely applied 2) and 3) in the ART project and 4) in the MML project.

Development of DB applications typically distinguishes three times (or phases): "creation time," at which the DB schema is produced; "load time," at which the instances of the DB (or tuples in relational terminology) are generated; and "use time," at which instances of the DB are manipulated or retrieved. A distinguishing feature of PCDB with respect to typical DB applications is that in PCDB there is not just one creation, load, or use time for the entire DB, but rather each relation has its own.

We have developed a general classification of all the data in the PCDB by identifying a hierarchy of the relevant "times" for a program construction system and then indicating at which times relations are created, generated, and used.

Traditionally, for program construction systems we distinguish six "times":

1) *compiler design* (CD) time,
2) *version management system design* (VSD) time,
3) *target architecture design* (TAD) time,
4) *program compilation* (PC) time (includes front-end processing and linking),
5) *program load* (PL) time (includes target-dependent back-end processing of an intermediate text), and
6) *program execution* (PE) time.

The above "times" are shown in Fig. 2, organized in a self-explanatory precedence tree. Fig. 3 summarizes the classifications in this section.

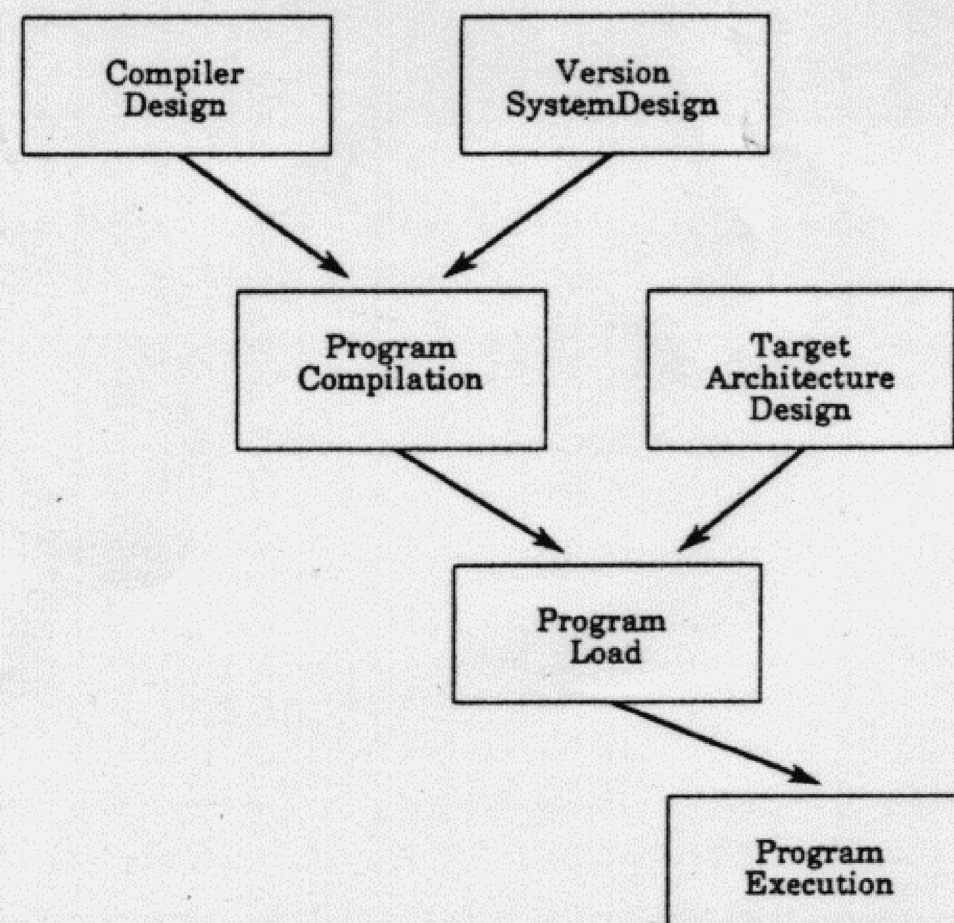


Fig. 2. Partial order of "times" in program construction systems.

Type of data	Schema design time	Instance creation time	Instance usage time
VERSION MANAGEMENT	Version System Design	SW Lifecycle	SW Life Cycle
SOURCE LANGUAGE DESCRIPTION	Compiler Design	Compiler Design	Program Compilation
INTERMEDIATE LANGUAGE DESCRIPTION	Compiler Design	Program Compilation	Program Compilation
SYMBOL TABLE	Compiler Design	Program Compilation	Program Compilation, Program Execution
RELATION BETWEEN COMPILATION UNITS	Compiler Design	Program Compilation	Program Compilation, Program Execution
OBJECT PROGRAM DESCRIPTION	Compiler Design, Program Compilation	Program Compilation	Program Execution
DATA OBJECT DESCRIPTION	Compiler Design, Program Compilation	Program Compilation Program Execution	Program Execution
TARGET HARDWARE DESCRIPTION	Target Hardware Description	Target Hardware Configuration	Target Hardware Configuration
OBJECT PROGRAM TO HARDWARE ALLOCATION	Target Hardware Description, Compiler Design	Program Loading, Program Execution	Program Execution

Fig. 3. Classification of relations in the PCDB.

A. Version Management Subschema

Relations of this subschema maintain information about module history, including author, users, authorizations, creation or revision dates, processing phases passed, tests, etc. [2]. Relations are created at VSD time and used during the software life cycle; instances are manipulated as an effect of editing programs, checking their interfaces, compiling, linking or executing them, making documentation, and so on. This application of DB's is rather established and will not be further covered.

B. Program Translation Subschema

Data in the program translation subschema can be further classified according to their usage.

• *Source Language Descriptions:* These describe the lexical elements of the language and the grammar. Relations are created and loaded at CD time and never updated; they are used by scanners, parsers, and syntax-directed editors.

• *Intermediate Language Descriptions:* These are commonly produced by the various phases of compilation and typically represent an abstract program tree; they are used as input to code generators, optimizers, or interpreters. Relations for intermediate description are created at CD time, while their instances are produced and used at PC time.

• *Symbol Tables:* These describe declared data or program structures. Relations of the symbol table are created at CD time, and their instances are produced and used at PC time; portions of the symbol table can also be used later, at PE time, e.g., for debugging. In some compilers, differences in the lifetimes of parts of the symbol table are carefully exploited for discarding information as soon as possible. Using a DB is convenient for this purpose, as it is possible to select portions of relations during the compilation, store them in secondary memory, and then reconstruct the overall information when needed. Notice that the distinction between intermediate languages and symbol tables is no longer required, as there are recent proposals (such as Diana [21] or ART itself) where they have been integrated.

• *Interrelations Between Compilation Units:* These give the visibility and precedence rules between compilation units and are treated by the linker. Part of these relations may be shared with the version management subschema. The schema of the relations is made at PD time; instances are produced and used by the linker at PC time and also used at PE time.

Examples of relations of this subschema are described in Sections III-A and III-B.

C. Program Execution Subschema

Some of the data of this subschema are also common to the program translation subschema (they are in fact produced during program translation and used during program execution). They are as follow.

• *Object Program Descriptions:* An object program is a sequence of instructions (of a virtual or physical machine) which address memory locations where variables and constants are stored. The schema of relations for describing object programs is designed at CD time; instances of this relation (i.e., actual object programs) are generated at PC time and used at PE time.

• *Data Object Descriptions:* Objects (variables, constants, etc.) are traditionally classified as statically, semistatically, or dynamically allocated [20] according to the time when their size and memory address can be decided. Consistently with the relational approach, we unify object representations by storing them as tuples of relations; however, this distinction leads to different times at which the schema and instances of the relations are created. In most cases, instances are generated and used at PE time. Relations storing static and semistatic objects can be created at PC time. The situation for dynamic objects, such as arrays of variable dimensions, is instead more complex. Consider a dynamic object defined within a procedure: the amount of storage required is different at each execution of the procedure; therefore, it is not possible to create "static" relations for storing it. In a "pure" relational approach, one should create at PE time distinct relations for each dynamic object or use complex representations. The ART relational DB manager supports attributes of type "vector," with a variable number of values; thus, dynamic objects can be stored as values of those attributes. This solution extends the "pure" relational approach in the direction of "non-first-normal-form" relations [26], [28].

• *Process Descriptions:* In languages with concurrency, another run-time structure contains the descriptors of concurrently executing processes. The schema of relations for process description is made at CD time; instances of these relations are generated and used at PE time. Our two projects differ in this respect: Ada run-time support is based entirely on relations, wheras MML is not, for efficiency reasons. We stress that the penalty of a relational representation for run-time structures is unacceptable in practical real-time applications.[2]

Some examples of object program and data object descriptions are shown in Section III-C; examples of process description are in Section III-D.

The *debugger* is also part of the run-time support in a program construction system; the use of the PCDB is highly beneficial for the debugger since 1) all the relevant symbol table information is made available to the debugger, and 2) objects can be inspected and modified easily, allowing monitoring and forcing of program execution.

Much attention has been devoted to an innovative debugger for Ada, which is actually the object of separate research [14], summarized in Section III-D.

[2]However, we recently learned that a telephone switching system ESS5 by AT&T applies a core-resident special relational data manager for line switching.

D. Target Configuration Subschema

This information is used for describing the allocation of resources in a target system with a varying configuration. The situation arises in embedded computer applications based on multimicroprocessor targets and in no-stop reconfigurable machines. We recognize the following.

• *Target Hardware Descriptions:* These describe types and number of processors, types and sizes of memories, interprocessor connections, process-to-memory accesses, peripherals, the operational status of the various components, etc. These relations are created at TAD time; instances are mostly modified when the target system is reconfigured.

• *Object Program Allocation Descriptions:* These describe the allocation of processes to processors and of software modules to storage; they also indicate the connections (e.g., serial line, shared memory, ethernet) used for interprocess communication. Relations of this schema are created statically (at TAD or CD time), and instances are generated at PL time (assuming a static allocation of processes) or ET time (in the case of dynamic allocation of processes).

Target description and allocation are illustrated in Section IV.

III. The ADA Relational Translator

A compiler and interpreter for a large subset of Ada, including tasking, generics, and most critical aspects of the language, have been implemented in a period of about 2 years by 16 graduate students, for a total of about 7 man-years. We did not place any emphasis on performance or on innovative compilation methods since we considered this project a prototype and testbed of the relational methodology. All programs, including the relational data manager, were written in Pascal for a Vax-780 under Unix 4.1.

Next, we briefly describe the novel aspects of the project; a detailed presentation can be found in a series of articles referred to by [6].

The structure of the ART system is shown in Fig. 4; it includes a lexical and syntactical module (LEXSYN), a semantic module (SEM), and an interpreter (EX), which includes run-time support for tasking. The subset of Ada was not established ahead of prototyping, but rather, we encouraged addition of new features as the project proceeded; thus, the treatment of exceptions, generics, and a symbolic debugger (the *h*igh-level *A*da *r*elational *d*ebugger, or HARD) were added in the second year by second-generation participants.

A. Lexical and Syntactical Analysis

This activity belongs to the program translation subschema of Fig. 1 and occurs at program compilation time (Fig. 2).

LEXSYN is driven by a finite-state automaton and an extended BNF grammar, both relationally encoded. The

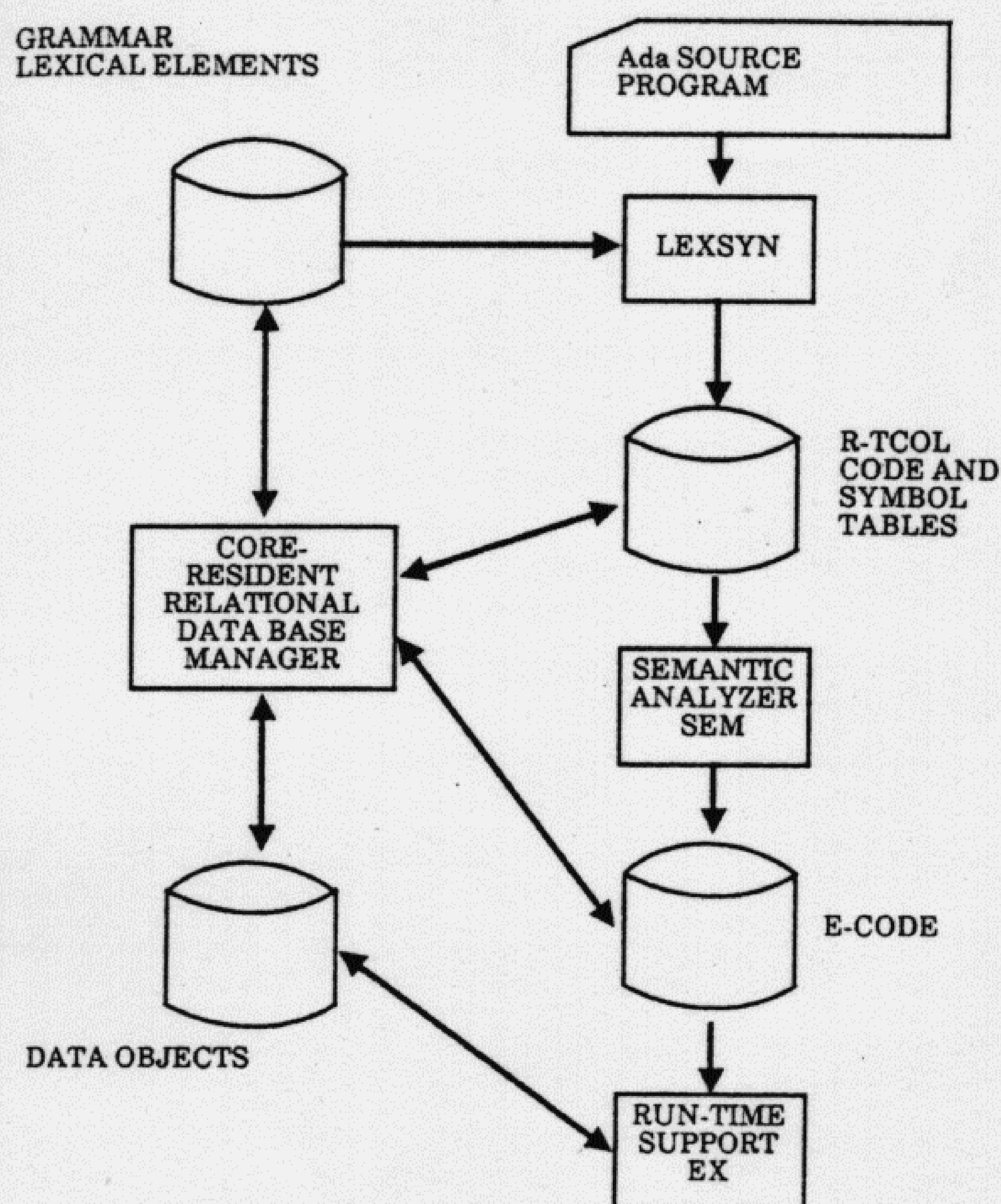


Fig. 4. ART project structure.

```
procedure LEXSYN is
 CT: STRING; -- global variable for the current token;

procedure NEXTSYM is
begin
 -- returns CT as the current token;
end;

procedure PARSER (CLS: in STRING) is
begin
 X <- PROJECT[P#]
    SELECT[GUIDE__SYMBOL = CT and LEFT__SIDE = CLS] LEFT__PART;
  -- determines the current production or productions
  -- for expanding the current left side;
  -- if cardinality is 0 then there is an error;
  -- if cardinality is greater than 1 then there is non-determinism;
 case CARD(X) is
  when 0 = > ERROR;
  when 1 = > L <- PROJECT[LENGTH] SELECT[P# = X]
                   PRODUCTION__LENGTH;
       for i in 1..L loop  -- loops on elements of the right side;
        Y <- PROJECT[ELEMENT,ACTION]
           SELECT[POSITION = i and P# = X] RIGHT__PART;
        if TERMINAL(Y.ELEMENT) then
          if Y.ELEMENT = CT then NEXTSYM; else ERROR;
          -- with terminal elements, the NEXTSYM procedure
          -- is called; mismatch with CT is an error
          end if;
        else PARSER(Y.ELEMENT);
          -- with nonterminal elements, PARSER is called;
        end if;
        if Y.ACTION /= 0 then TRANSLATOR(Y.ACTION);
          -- a translation action might be performed;
        end if;
       end loop;
     when others = > NON-DETERMINISM;
 end case;
end PARSER;

begin
NEXTSYM;          -- reads the first token;
PARSER(AXIOM); -- enters the recursion;
end LEXSYN;
```

Fig. 5. Sketch of the LL(1) parser, which embeds operations of relational algebra, i.e., PROJECT and SELECT.

state transition graph of the finite-state automaton is modeled by the relation

AUTOMATA (<u>PRESENT-STATE</u>, <u>SYMBOL</u>, NEXT-STATE, OP-CODE, KIND)

Attributes forming the primary key are underlined. The triple PRESENT-STATE, SYMBOL, NEXT-STATE is the transition function; the OP-CODE attribute denotes the operations associated with state transitions, while KIND indicates if the transition has recognized a token (e.g., number, identifier, delimiter, or keyword).

Syntactical analysis is performed top-down on LL(1) extended BNF grammars. We exemplify the design of relational data structures representing a grammar as a typical application of "normalization" [25] in this context. A grammar is encoded by the attributes

- LEFT-SIDE the symbol on the left-hand side of a production,
- GUIDE-SYMBOL look-ahead terminal symbol,
- P# production number,
- LENGTH length of a production,
- ELEMENT element in the right-hand side of a production,
- POSITION element's position on the right-hand side,
- ITERATION, OPTIONALITY whether an element is iterated or optional, and
- ACTION identifier of a semantic action to be activated.

We then recognize the functional dependences:

LEFT-SIDE, GUIDE-SYMBOL $\rightarrow$ PRODUCTION-NUM

PRODUCTION-NUM, POSITION $\rightarrow$ ELEMENT, ITERATION, OPTIONALITY, ACTION

PRODUCTION-NUM $\rightarrow$ LENGTH

From these attributes and dependences, we construct the following third normal form relations:

LEFT-PART (<u>LEFT-SIDE</u>, <u>GUIDE-SYMBOL</u>, P#)

RIGHT-PART (<u>P#</u>, <u>POSITION</u>, ELEMENT, ITERATION, ACTION)

PRODUCTION-LENGTH (<u>P#</u>, LENGTH)

The parser is a recursive procedure that implements a variant of the recursive descent algorithm; the parser is data driven, with no procedural knowledge of the grammar embedded in its code. In Fig. 5, the parse algorithm is sketched; the basic structure of the algorithm is not altered by error treatment or nondeterminism (both omitted here), which is solved using semantic information.

A complication is the syntactical ambiguity caused by overloading. An identifier can assume different meanings because of multiple definitions. We have carefully designed the grammar of Ada to confine the sources of ambiguities within alternative productions having the same left part. Then, LEXSYN produces as many intermediate representations as possible interpretations; the solution of the ambiguities is left to the semantic analyzer, to be described later. With this choice, nondeterminism is confined to that part of the tree where overloading causes uncertainty, without giving rise to combinatorial explosion of alternatives.

B. The Intermediate Relational Representation and Semantic Analysis

The language used for intermediate (abstract) representations of Ada programs is R-TCOL, a relational implementation of the language T-COL-Ada [35] (the forerunner of Diana [21]). T-COL-Ada is a directed acyclic graph with the following properties.

- Nodes are associated with the abstract structures of Ada. They are classified into disjoint types, and each node type has several attributes, which describe the properties of the corresponding abstract structure.
- Oriented arcs represent references between abstract structures.

R-TCOL is an implementation of T-COL-Ada consisting of a set of relations obtained by applying the following rules.

- Each type of node in T-COL-Ada is modeled by a relation of R-TCOL; each node instance of a particular type within a T-COL-Ada graph is therefore a tuple of the corresponding R-TCOL relation.
- Each attribute associated with a node type becomes an attribute in the corresponding R-TCOL relation.
- Oriented arcs between nodes are represented explicitly by attributes. If the type of the destination node is fixed, then the arc is modeled by adding in the source relation an attribute referencing the tuple identifier of the destination node; otherwise, the arc is modeled by adding in the source relation a pair of attributes giving the relation name and the tuple identifier of the destination node.

Fig. 6 shows an example of an Ada source program and the corresponding R-TCOL representation. Fig. 7 contains a synthetic description of the meaning of these relations.

Before translating the R-TCOL graph into an executable code for the run-time interpreter, module SEM performs a static semantic verification of the correctness of an R-TCOL graph: the acyclic graph is traversed, and at each node, an appropriate verification action is called. Each node type is associated with a distinct semantic checker. An example of a semantic check is the correctness of jumps (**exit** and **goto**); auxiliary relations describing the environments of the jump and target statements are constructed by SEM recursively ascending the R-TCOL graph until a tuple representing a program unit or an **accept** statement is reached.

The algorithm employed by SEM in order to solve overloading is a relational implementation of the two-pass algorithm [33]. This method is easily integrated within the translation from source programs to R-TCOL and requires the creation of two intermediate relations and a two-pass analysis of the R-TCOL graph. It has been compared to other existing methods, in particular [3] and [18], which are less appealing in the ART environment.

C. The E-CODE Intermediate Language

After passing semantic checks, R-TCOL graphs are translated into a sort of three-address-code, called E-CODE, suitable for interpretation; the schema of relation E-CODE is

```
E-CODE is
    E-CODE-REL (T#,OP-CODE, EXT-REL,
                EXT-T#, A-REL,A#, B-REL,B#,
                C-REL,C#)
```

T# provides tuple identification; OP-CODE specifies the operation code of the E-CODE instruction; EXT-REL and EXT-T# escape to one auxiliary relation and a specific tuple within it (the E-CODE language is complemented by 15 external relations); subsequent attributes reference 3 operands (A, B, and C) by indicating the relation where they are stored and their attribute indentifiers. Compilation units or subunits partition E-CODE-REL into specific E-CODE relations; Fig. 8 shows the E-CODE relation for the program presented in Fig. 6.

The activation records of procedures are also modeled as relations; they include a group of standard attributes (representing the static and dynamic chains, arranged in a cactus-stack organization) and a group of procedure-specific attributes, representing objects used by the procedure (parameters, variables, temporary memory locations). Each procedure is associated with an activation record relation; upon invocation, one new tuple is inserted into the relation. As mentioned in Section II, we store Ada dynamic objects in flexible vector attributes of the DB.

An elegant solution was adopted for *generic units* by indirect addressing; portions of E-CODE describing generic units do not make reference directly to activation records, but rather to intermediate relations containing references to instantiation-specific activation records. With this solution, different instantiations of a generic unit share E-CODE instructions.

D. Execution and Debugging Environment

The EX module contains the E-CODE interpreter and the operating system kernel [23], [24] supporting multitasking. The main kernel functions are to schedule ready tasks for execution, to suspend tasks which wait for future events (e.g., for an *entry* to be open or for the expiration of a delay), to create new tasks and destroy terminating tasks, and to handle exceptions in concurrent operations. In addition, the kernel holds the queues required for scheduling ready tasks for execution (we use round robin with time slicing), for choosing the next task among those waiting for acceptance on the same *entry*, and for managing the tasks waiting for expiration of delays. We outline kernel information representation by relations. TASK-DESCRIPTOR stores relevant information concerning initiated tasks, such as

- STATUS, taking one of the following values: {"active," "suspended," "ready," "delayed," "waiting-for-rendez-vous," "waiting-in-rendez-vous," "completed," "waiting-for-activation-of-dependents"};
- reference to the TASK-CODE (within E-CODE);
- reference to the TASK-DATA (a specific activation record); and

```
task BUFFER is
  entry READ(C: out CHARACTER);
  entry WRITE(C: in CHARACTER);
end BUFFER;

task body BUFFER is
  -- declarations omitted
  begin
   -- some instructions omitted
  select
   when COUNT < POOL-SIZE = > accept WRITE(C: in CHARACTER)
   do -- sequence of statements for entry WRITE
   POOL(IN-INDEX) := C;
   end WRITE;
   IN-INDEX := IN-INDEX mod POOL-SIZE + 1;
   COUNT := COUNT + 1;
  or
   when COUNT > 0 = > accept READ(C: out CHARACTER);
   do -- sequence of statements for entry READ
   C := POOL(OUT-INDEX);
   end READ;
   OUT-INDEX := OUT-INDEX mod POOL-SIZE + 1;
   COUNT := COUNT - 1;
  end select;
   -- some instructions omitted
end BUFFER;
```

(a)

T#	name	formal-seq#	accept-seq#
23	READ	102	103
24	WRITE	104	105

Relation ENTRY-SYM

T#	name	spec-seq#	body
12	BUFFER	101	31

Relation TASK-SYM

T#	kind	sym#	block-seq#	exc#
31	TASK	12	106	0
32	ACCEPT	28	110	0
33	ACCEPT	29	111	0

Relation BLOCK

T#	kind	sym#
46	Entry-sym	23
47	Entry-sym	24

Relation DECL

T#	altern-seq#
3	107

Relation SELECT

T#	ord	rel	num
101	1	Decl	46
101	2	Decl	47
102	1	Varbl-sym	40
103	1	Accept	28
104	1	Varbl-sym	41
105	1	Accept	29
106	n	Select	3
107	1	When	5
107	2	When	6
108	1	Accept	28
109	1	Accept	29

Relation SEQUENCE

T#	entry#	block#
28	23	32
29	24	33

Relation ACCEPT

T#	kind	statm-seq#	ind-seq#
5	or	108	0
6	or	109	0

Relation WHEN

(b)

Fig. 6. (a) Fragment of an ADA program. (b) R-TCOL representation for the fragment of an Ada program in (a).

• reference to MASTER-DATA (the activation record of the master unit).

ENTRY-DESCRIPTOR describes the status of each entry (ENTRY-ID) of each task (TASK-ID); each specific entry has an ENTRY-CONDITION (which can be "open" or "close") and can be IN-RENDEZ-VOUS with other tasks.

Three relations describe queues of tasks. In Fig. 9, we have one queue of "ready" tasks and several queues of tasks waiting on entries (either in rendez-vous or for rendez-vous). Each task can belong to only one queue at any instant. Based on this property, queues are modeled by the QUEUES relation that gives the successor for each task in any queue. Two relations give the first and last

TASK_SYM:
Describes all the tasks declared in the Ada program; in particular, the tuple ith T# = 12 represents the task BUFFER. AttributeSPEC_SEQ# is a reference to the sequence of declarations belonging tothe task specification (see relation SEQUENCE);
attribute BODY refers to the block which encloses the task's body(see relation BLOCK). Both attributes represent arcs of the T-COL-Adagraph (each arc is modeled by a single attribute).

SEQUENCE:
Describes the sequential order of ADA statements.This relation has a double key: attributes with the same T#represent a sequence of references to R_TCOL tuples, ordered by increasing value of the ORD attribute.
The remaining two attributes REL and NUM represent the destination of two arcs of the T_COL_Ada graph.

DECL:
Describes declarative statements; in particular, tuples 46 and 47 represent the declarations of entries READ and WRITE.
Attributes KIND and SYM# are references to tuples of the symbol-table relations; in particular, in this case the relation ENTRY_SYM is referenced.

ENTRY_SYM:
Describes the symbol-table informations about entries. Tuples 23 and 24 describe entries READ and WRITE: attribute FORMAL_SEQ# is a reference to the sequence of entry's formal parameters, while ACCEPT_SEQ# is the list of accept statements referencing the entry.

BLOCK:
Describes block statements, program unit bodies and sequence of statements associated to accept statements.
Attributes KIND and SYM# are a reference to the symbol-table tuple representing the entity owner of the block; BLOCK_SEQ# gives the sequence of instructions contained in the body, while EXC# is a reference to a tuple desribing exception handlers (if any) of the block.

ACCEPT:
Tuples of this relation represent accept statements; this relation links accept instructions to the corresponding body.

SELECT:
Describes statement Select: it indicates the sequence of alternatives.

WHEN:
Describes each alternative: it indicates the nature of the alternative and the associated sequence of statements.

Note: relation VARBL_SYM is not shown

Fig. 7. Description of some R-TCOL relations.

T#	Op-Code	Ext-rel	Ext-T#	A-Rel	A-T#	B-Rel	B-T#	C-Rel	C-T#
24	SELECT	ALTERN-REL	1	nil	3	nil	11	nil	0
25	ACCEPT	ACC-REL	2	WRITE	0	INTEGER	0	nil	0
26	ASSGN	INFOAS	1	BUFFER	12	WRITE	3	nil	0
27	END-ACCEPT	ACC-REL	2	WRITE	0	INTEGER	0	nil	0
28	MOD	nil	0	BUFFER	8	BUFFER	6	BUFFER	10
29	ADD	nil	0	BUFFER		INTEGER	1	BUFFER	8
30	ADD	nil	0	BUFFER		INTEGER	1	BUFFER	7
31	GO-TO	nil	38	nil	0	nil	0	nil	0
32	ACCEPT	ACC-REL	3	READ	0	INTEGER	0	nil	0
33	ASSGN	INFOAS	2	READ	3	BUFFER	12	nil	0
34	END-ACCEPT	ACC-REL	3	READ	0	INTEGER	0	nil	0
35	MOD	nil	0	BUFFER	9	BUFFER	6	BUFFER	10
36	ADD	nil	0	BUFFER	10	INTEGER	1	BUFFER	9
37	SUB	nil	0	BUFFER	7	INTEGER	1	BUFFER	7
38	GO-TO	nil	24	nil	0	nil	0	nil	0

Fig. 8. E-CODE relation for the fragment of a program in Fig. 6.

element of the queue: READY-QUEUE-DELIMITERS is a singleton relation for the "ready" queue, and ENTRY-QUEUES-DELIMITERS has one tuple for each queue established for a particular entry.

Access to elements of queues is initially via the DELIMITER relations (giving the first or last elements) and then by traversing the QUEUE relation (in the appropriate direction). This is an example of an application that would benefit from the introduction of recursion into the algebra (see Section V). Fig. 9 shows an example of some of the

task-id	status	task-code	task-data	master-data
100	wfrv	......	......	......
200	wirv	......	......	......
300	active	......	......	......
400	ready	......	......	......

TASK-DESCRIPTOR

entry-id	task-id	entry-condition	in-rendez-vous
127	300	open	200

ENTRY-DESCRIPTOR

task-id	next
400	nil
200	100
100	nil

QUEUES

first	last
400	400

READY-QUEUE-DELIMITERS

entry-id	first	last
127	200	100

ENTRY-QUEUE-DELIMITERS

Fig. 9. Representation of task information for the kernel.

above relations; we assume that task 300 is active and has accepted a call to its entry 127 by task 200, so that tasks 300 and 200 are in rendez-vous; task 100 has also issued a call to entry 127 and is waiting for acceptance, while other tasks are waiting for execution.

Symbolic run-time **debugging** is a challenge in a concurrent environment, especially for a very complex language like Ada [17]. The need for a debugger was perceived when the ART project was already advanced. Nevertheless, design and development of our high-level debugger were easier than they would have been in a more traditional organization.

The main targets of HARD were [14] the use of Ada features to debug Ada programs, to facilitate use by Ada programmers, and the presence of two usage modes, interactive debugging and unmanned execution for monitoring and analysis. We used the predicate-action approach: *predicates*, commonly referred to as "breakpoints," are Boolean functions defined for each event to be detected. The truth of a predicate triggers an action, which examines or modifies the state of execution. *Actions* can be predefined (e.g., "suspend") or user-defined. The interactive usage mode is useful to remove bugs affecting design requirements, while the unmanned mode (based on user-defined Ada monitoring tasks) allows checking for time-dependent errors and evaluation of performances. The user can insert and remove breakpoints

on a full set of conditions (statement execution, entry call and accept, exception raise, etc.); he can display information about variables, the scope, the history of concurrent execution, the kernel status, and so on; and he is able to manipulate such information in order to modify the future flow of execution.

As a consequence of homogeneity with Ada, some HARD features are as follow.

- Breakpoints are implemented as entry calls, which are inserted into the interpretable E-CODE.
- Predefined actions are implemented through a set of procedures which are written partly in Ada and partly in E-CODE (with some extensions).
- Ada tasks are used to build predicates and actions (in the unmanned mode).

HARD uses data from the *program translation* and *program execution* subschemas (Fig. 1). A major advantage of the relational approach has been experienced in obtaining interfaces which are easy to build and access. Further details on HARD can be found in [14].

IV. Target Configuration Control in a Program Construction System for Multimicroprocessors (MML)

In contrast to ART, which was originally conceived to assess the use of relational methods for prototyping, the MML [4] project was a four-year cooperative effort of academic and industrial institutions. MML is a language derived from Pascal, extended with processes, communication primitives, and low-level I/O, for writing concurrent real-time programs. An innovative feature of MML is that the architecture and configuration of the target are variable and can be changed without affecting the program.

Target systems are categorized at two levels.

- The *architecture* is roughly the family of interconnectable boards out of which a system can be made. Examples are the TOMP architecture [13], allowing from 1 to 64 Zilog Z8001 processors and various combinations of private, local, and global buses and memories, or an architecture based on Motorola M68000 microcomputers linked by a LAN.
- Within a chosen architecture, the *configuration* defines the number and types of processors, the sizes and attributes of memories (ROM/RAM, private/global), the interconnection scheme (shared memory, two-port memory, serial/parallel bus), and the peripherals included in the actual target system.

Fig. 10 shows two configurations of the TOMP architecture.

The *user* of MML must provide a description of the target configuration. Then he decides, at loading time, the hardware allocation of his processes. Notice that after a change in configuration of the target the same source program can be reused without recompilation. The structure of the MML program construction systems, named the *M*ulti *M*icro *D*evelopment *S*ystem (MMDS) is shown in Fig. 11. On the other hand, architectures are defined by

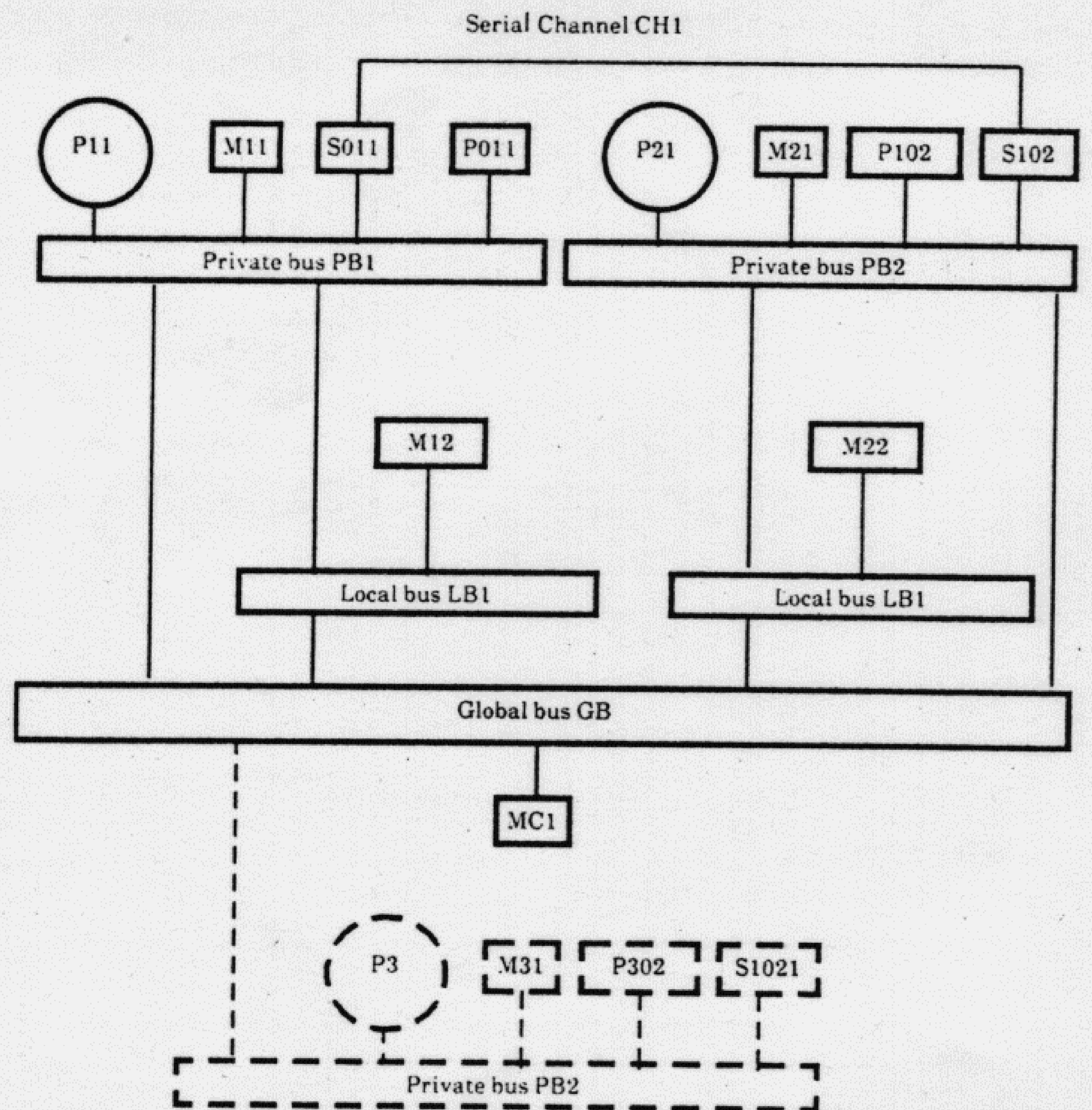


Fig. 10. A configuration of the target system belonging to the architecture TOMP. Adding the dashed blocks, another configuration is obtained.

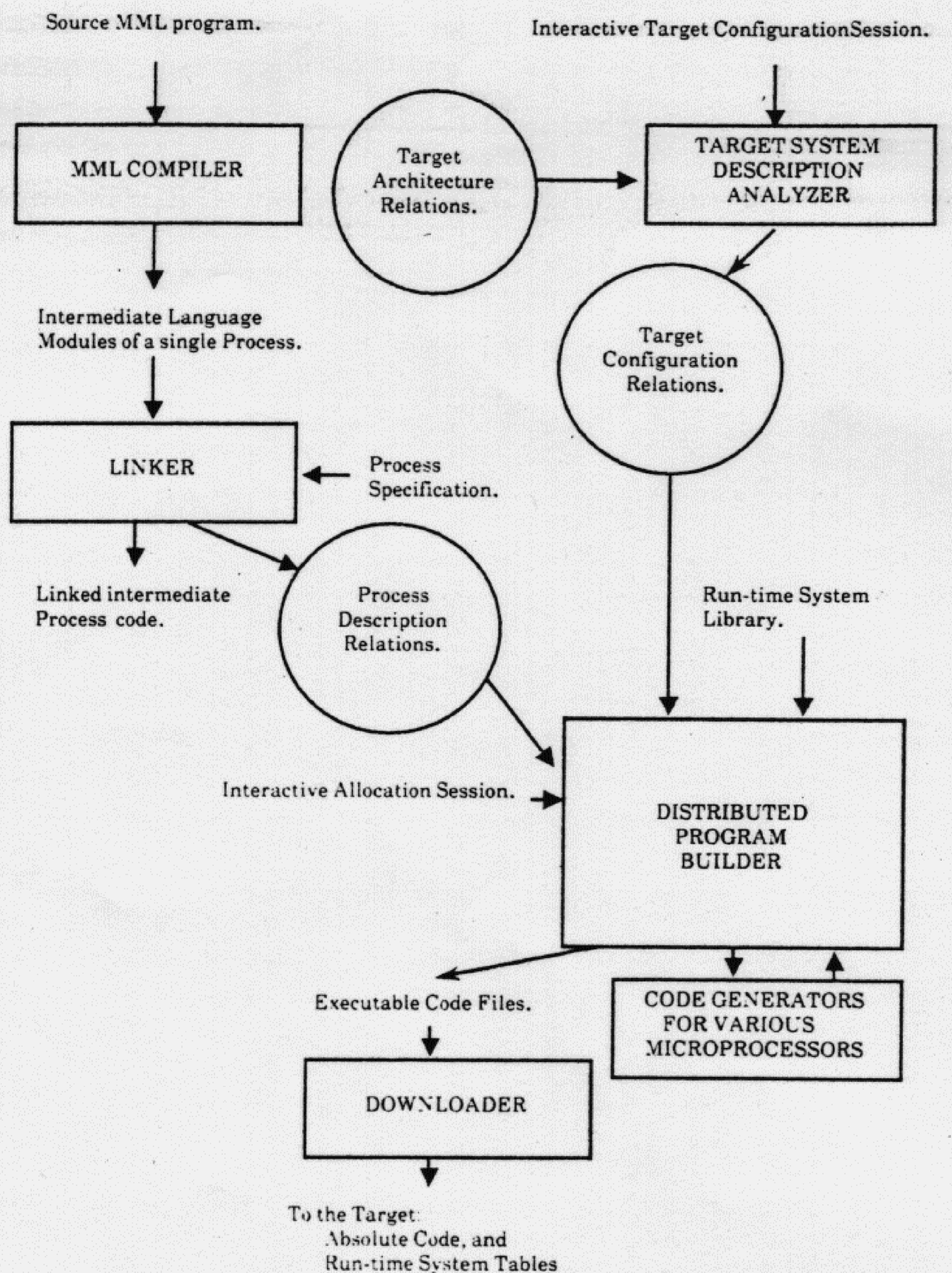


Fig. 11. The MML program construction environment.

the *system designers* of MML since introduction of a new architecture imposes some work: code generation and run-time support must be retargeted.

All the interfaces between the subsystems are relationally described using a specifically designed DBMS derived from the relational DB of ART, but (unlike ART) the MML compiler uses conventional data structures for its internal data. We briefly outline the relations used for architecture, configuration, and allocation description.

Architecture Description: Its purpose is to define the range of architectures available to MML users. A suitable relation describes the properties of each element of the architecture. Examples are omitted for brevity.

Target Configuration. The target configuration is then described interactively, under control of a tool which verifies that the actual values supplied agree with the legal values, specified in the description tables of the chosen architecture. Fig. 12 shows part of the relational description for the target system in Fig. 10 (solid lines). Notice that some of the relations in Fig. 12 represent the topology of the target system (processors, memories, buses, and the connections); others give properties of the target, such as the access rights of processors to memories.

Allocation Description: The next step for MML users is to specify allocation of the source programs to the target resources. Such information is used at load time for targeting the code and computing the addresses; at execution time, it is accessed by the kernel and debugger. In MML, tasks cannot be dynamically created or moved around at run time.

Examples of relations for allocation description are shown in Fig. 13; they provide, in particular, the following information: 1) the mapping of MML tasks to processors, 2) the residency of program modules, 3) the link (including communication protocols) used for communication between any two processes and 4) the physical address corresponding to a certain interrupt signal. Notice that some tuples (not shown in Fig. 13) are preset to indicate the presence of kernels or other run-time libraries on some memories. The loading phase is completed by preparing a summary of relations to be downloaded on the target for future use by the kernel or debugger; e.g., the link to be used for rendez-vous by two processes is one such piece of information. Remember that such relations are represented by specialized data structures for higher efficiency.

MMDS was implemented in Pascal under RSX-11 for PDP-11 machines. Currently, MML is undergoing a complete redesign in order to become a professional tool, but the basic relational philosophy has been retained. It is worth noting that the relational approach has been naturally applied to version management information and that a commercial DB management system has been used instead of the previous ad hoc relational data manager. The change was made possible by the growth of resources of newer Unix host machines. For details on the new MML environment (termed the Multi Micro Program Support Environment), refer to [19].

Processor-Id	Model	I/O Mode
P11	Z8001	IO-Mapped
P21	Z8001	IO-Mapped

MODEL AND I/O MODE OF PROCESSORS

Processor-Id	Memory-Bank-Id	Right
P11	M11	read write
P11	M12	read write
P11	MC1	read only
P21	MC1	read write
P21	M21	read only
P21	M22	read write

ACCESS RIGHTS OF PROCESSORS TO MEMORIES

Bus-Id	Bus-Id
PB1	GB
PB1	LB1
GB	LB1
PB2	GB
GB	LB2
PB2	LB2

BUS TO BUS CONNECTIONS

Bus-Id	Kind
PB1	Private
PB2	Private
LB1	Local
LB2	Local
GB	Global

DESCRIPTION OF BUSSES

Unit-Id	Link-Id
P11	PB1
P21	PB2

PROCESSOR-BUS CONNECTION

Unit-Id	Link-Id
M11	PB1
M12	LB1
M21	PB2
M22	LB2
MC1	GB

MEMORY-BUS CONNECTION

Memory-Bank-Id	Size	Kind
M1	32K	ram
M12	8K	ram
M2	64K	ram

DESCRIPTION OF MEMORY BANKS

Fig. 12. Relations describing the solid line target configuration of Fig. 10 within the MML program construction system.

Processor-Id	Sequence-Id
P11	Consumer
P21	Producer

MAPPING OF MML SEQUENCES (i.e. TASKS) ONTO PROCESSORS

Program-Module-Id	Memory-Bank-Id	Offset
Consumer.Data	M12	0
Consumer.Code	M11	4560
Producer.Data	M22	0
Producer.Code	M21	4800

LOCATION OF PROGRAM MODULES

Fig. 13. Relations describing the mapping of an MML program onto the target configuration of Figs. 10 and 12. We assume that the application consists of two processes, a consumer and a producer, each one split into a data and a code module.

V. Conclusions

Relational programming is an attractive evolutionary paradigm to overcome the present difficulties in software construction [5]. We have applied relational programming in the ART project, where 16 part-time graduate students have been working for about two years with little supervision, and in the MML project, conducted by 6 experienced persons and a project leader. Both experiences confirmed the validity of the relational approach for developing complex evolutionary projects with little central supervision and, in the first case, a fast turnover. We have attempted to analyze the reasons for those successes and to develop further the method and tools that support relational programming. In our opinion, the main advantages are the following.

- In a situation where specifications of the project cannot be laid down completely, the conceptual approach of data modeling, typical of DB applications, avoids early commitment in designing data structures and algorithms. The analysis is focused on identifying relations and their attributes and on organizing them in accordance with functional dependences by applying normalization techniques. This is in contrast with the structured abstract-data-type approach to software specification and design, where enormous attention must be devoted early to an accurate and complete definition of operations and their domains.
- Relational algebra provides expressive and simple facilities for extracting the data view which is needed when new unanticipated functions must be added to the project. Even when data structures are intricate, as with compiler symbol tables, their relational description allows easy entry for newcomers, who can quickly retrieve the relevant components of data. Notice that this approach might lead to inefficient algorithms: when new functions are added to the system, some duplicated processing occasionally takes place if the newcomers do not care to check whether similar processing has already been in the system. However, unanticipated extensions of software projects developed with other conventional approaches often require much more costly respecifications and program revisions.
- As a result, the program structure is totally decoupled from the team structure: addition of manpower for new functions is handled by extracting a suitable view from the DB. Further project documentation is grealy simplified since it coincides with the relational specification of data: alogrithms are of only private interest to each team.
- As Codd states in his Turing lecture [11], relational algebra effectively simplifies procedures by hiding most iterations within the algebraic operators. As loop complexity is a significant measure of program cost in software metrics, it is clear that much effort is thus spared. However, not all loops can be avoided by relational operators, and some extensions to relational algebra are needed, as discussed below.
- A relational prototype might be progressively transformed into a working product. This can hardly be done automatically, but a set of guidelines could be developed for hand-optimizing the system; an experience of this kind is described in [27] for a well-known Ada interpreter prototyped in Setl.

Coming to critical aspects of relational programming, we analyze the current limits and required extensions of the relational approach.

- Current query languages, as typified by SQL or INGRES, are specifically designed for interactive use; even when they provide a program interface, in the form of procedure calls, this tends to be poorly readable. This criticism applies as well to ART's DB manager, which imposes rather unfriendly list manipulations for setting up schema descriptors and tuples. We are currently designing a new relational system [8] with a friendlier interface and more efficient storage management for relations.
- As we mentioned, not all iterative operations can be handled by relational operators; e.g., the ancestor–descendant relation cannot be computed from the parent–son relation, as it would require a transitive closure. In ART, several algorithms including most semantic tree traversal algorithms and the syntax analyzer of Ada source programs (Fig. 5), would benefit from transitive closures. In a current development, we have extended algebraic operators with a closure operator, which computes the fixpoint of a relational transformation. Work on the use of closures for executing recursive Prolog queries on a relational machine is in progress [7].
- Other limitations concern the relational model itself. A remarkable coincidence of research efforts aiming to extend the relational data model can be found in the direction of nested relations (or non-first-normal-form relations) [8], [26] [28], [34]. By means of nested relations, several situations can be more naturally modeled, leading to simpler relational algorithms.
- A major limitation of relational programming is the reduced efficiency of programs with respect to imperative programs, using ad hoc data structures. Indeed, our experiences confirm the fact [29] that a purely relational approach is impractical, at least using current DB technology. This reduces the scope of application of relational

programming to prototype development (as in ART) or forces us finally to design efficient data structures when real-time requirements are heavy (as in MML). However, this limitation is not a direct consequence of the approach itself, but rather of current limitations of main memory DB's, and particularly of the system specifically used in our projects. We observe that if all parts of a complex system are implemented as relational algorithms, it then becomes appealing to investigate special software and hardware for fast execution of relational operations in the central memory, using associative addressing. The performance of such a system should compare favorably to those of high-level artificial intelligence languages.

Acknowledgment

The following persons have participated in the development of the ART project: L. Attuati, P. Lazzarini, A. Millovaz, L. Orlandi, G. Paolillo, M. Pipponzi, A. Rosti, L. Zoccolante, C. Costantini, R. Domenichini, E. Gerlo, P. Gomarasca, P. Locatelli, and G. Morganti. The ART project is primarily an outcome of their work and enthusiasm. The MML project was lead by A. Daprá of TXT, with the participation of other parties, including CISE (L. Zoccolante). MML's database was designed by C. Filippi. The current industrial development of MML is by TXT (S. Gatti) and CISE (L. Zoccolante).

References

[1] *Reference Manual for the Ada Programming Language*, MIL-STD 1815a, U.S. Department of Defense, Jan. 1983.

[2] "Requirements for Ada Programming Support Environments, U.S. Department of Defense, Feb. 1980.

[3] T. P. Baker, "A one-pass algorithm for overload resolution in Ada," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 4, pp. 601–614, Oct. 1982.

[4] M. Boari *et al.*, "Multiple microprocessors programming techniques: MML, A new set of tools," *IEEE Comput.*, vol. 17, no. 1, pp. 47–58, Jan. 1984.

[5] B. W. Boehm and A. T. Standish, "Software technology in the 1990's: Using an evolutionary paradigm," *IEEE Comput.*, vol. 16, no. 11, pp. 30–37, Nov. 1983.

[6] S. Ceri and S. Crespi-Reghizzi "ART: Un compilatore per Ada progettato con tecniche relazionali," *Rivista Informatica*, vol. 15, no. 3–4, pp. 267–277, July 1985.

[7] S. Ceri, G. Gottlob, and L. Lavazza, "Translation and optimization of logic queries: The algebraic approach," in *Proc. VLDB Conf.*, Koto, Japan, Aug. 1986.

[8] S. Ceri *et al.*, "ALGRES: A system for the specification and prototyping of complex data bases," *IEEE Software*, submitted for publication.

[9] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, June 1970.

[10] ——, "Extending the database relational model to capture more meaning," *ACM Trans. Database Syst.*, vol. 4, no. 4, pp. 397–434, Dec. 1979.

[11] ——, "ACM-Turing award lecture: Relational databases: A practical foundation for productivity," *Commun. ACM*, vol. 25, no. 2, pp. 109–117, Feb. 1982.

[12] R. Conradi, T. Didriksen, and A. Lie, "IDL as a data-description language for a programming environment database," EPOS 15, Div. Comput. Sci., July 1986.

[13] G. Conte and D. Del Corso, Eds., *Multi-Microprocessor Systems for Real-Time Applications*. Boston, MA: Reidel, 1985.

[14] A. Di Maio, S. Ceri, and S. Crespi-Reghizzi, "Execution monitoring and debugging tool for Ada using relational algebra," in *Ada in Use, Proc. Ada Int. Conf.*, Cambridge University Press, Cambridge, UK, 1985, pp. 109–123.

[15] K. R. Dittrich, W. Gotthard, and P. C. Lockemann, "Damoken—A database system for software engineering environments," in *Proc. Int. Workshop Adv. Progr. Env.*, IFIP WG 2.4, Trondheim, June 1986.

[16] H. Eckardt *et al.*, "Report on the database programming language DBPL," J. W. Goethe Univ., Frankfurt, West Germany, Tech. Rep., Oct. 1987.

[17] R. E. Fairley, "Ada debugging and testing support environment," *ACM SIGPLAN Notices*, vol. 15, no. 11, pp. 16–25, Nov. 1980.

[18] H. Ganzinger and K. Ripken, "Operator identification in ADA: Formal specification, complexity, and concrete implementation," *ACM SIGPLAN Notices*, vol. 16, no. 2, Feb. 1981.

[19] S. Gatti and L. Zoccolante, "MME: A new integrated programming support environment for distributed embedded systems," presented at the IEEE COMPEURO 88, Apr. 1988.

[20] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*. New York: Wiley, 1982.

[21] G. Goos *et al*, Ed. *DIANA: An Intermediate Language for Ada*. Berlin: Springer-Verlag, 1983.

[22] S. Horwitz and T. Teitelbaum, "Generating editing environments based on relations and attributes," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 577–608, Oct. 1986.

[23] L. Ibsen, "A portable virtual machine for Ada," *Software Practice Experience*, vol. 14, no. 1, pp. 17–29, Jan. 1984.

[24] J. M. Kamrad II, "Runtime organization for the Ada language system programs," *Ada Lett.*, vol. 3, no. 3, pp. 17–29, Nov.–Dec. 1983.

[25] W. Kent, "A simple guide to five normal forms in relational database theory," *Commun. ACM*, vol. 26, no. 2, pp. 120–125, Feb. 1983.

[26] H. F. Korth, "Extending the scope of relational languages," *IEEE Software*, vol. 1b, pp. 19–28, Jan. 1986.

[27] P. Kruchten and E. Shonberg, "Le système ADA/ED: Une experience de prototype utilisant le language SETL," *Tech. Sci. Inform.*, vol. 3, no. 3, pp. 193–200, May 1984.

[28] G. Jaeschke and H. J. Schek, "Remarks on the algebra of non first normal form relations," in *Proc. ACM Symp. Principles Database Syst.*, Los Angeles, CA, Mar. 1982, pp. 124–138.

[29] M. Linton, "Queries and views of programs using a relational database system." Ph.D. dissertation, Dep. Comput. Sci., Univ. Calif., Berkeley, CA, 1984.

[30] R. Lorie *et al.*, "Supporting complex objects in a relational system for engineering databases," in *Query Processing in Database Systems*, W. Kim, D. Reiner, and D. Batory, Eds. New York: Springer-Verlag, 1985.

[31] B. J. MacLennan, "Overview of relational programming," *ACM SIGPLAN Notices*, vol. 18, no. 3, Mar. 1983.

[32] M. Olumi and G. Wiederhold, "Software project databases," IBM Res. Lab., San Jose, CA, Rep. RJ3862, Apr. 1983.

[33] T. Pennello, F. De Remer, and R. Meyers, "A simplified operator identification scheme for Ada," *ACM SIGPLAN Notices*, vol. 15, no. 7–8, July–Aug. 1980.

[34] M. A. Roth, H. F. Korth, and A. Silberschatz, "Theory of non-first-normal-form relational databases," Univ. Texas, Austin, TX, Rep. TR-84-36, Dec. 1984.

[35] B. R. Schatz *et al.*, "TCOL-ADA: An intermediate representation for the DoD standard programming language," Dep. Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, Rep. CMU-CS-79-112, 1979.

[36] J. W. Schmidt, "Some high-level language constructs for data and type relations," *ACM Trans. Database Syst.*, 2, no. 3, Sept. 1977.

[37] M. D. Tedd, S. Crespi-Reghizzi, and A. Natali, *Ada for Multi-Microprocessors*, 2nd ed. Cambridge, UK: Cambridge Univ. Press., 1987.

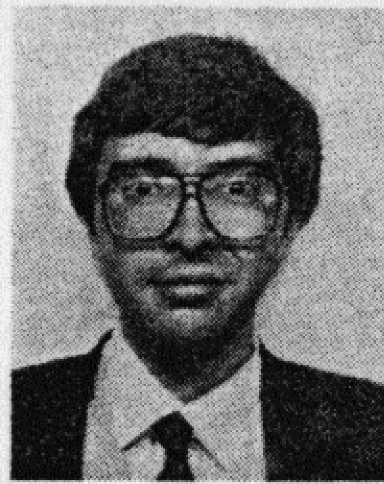
[38] J. Ullman, *Principles of Database Systems*, 2nd ed. Potomac, MD: Computer Science Press, 1983.

[39] C. A. Zehnder, Ed., "Database techniques for professional workstations," ETH, Zurich, Switzerland, Tech. Rep., 1983.

Stefano Ceri is a Professor of Computer Science at the Dipartimento di Matematica Pura ed Applicata, University of Modena, Modena, Italy, and a Visiting Professor at Stanford University, Stanford, CA. While working on this paper, he was at the Dipartimento di Elettronica of the Politecnico di Milano, Milan, Italy. His research interests include distributed databases, database design, and the use of databases in software engineering and logic programming. He is author of numerous articles in these areas and coauthor of the book *Distributed Databases, Principles and Systems* (McGraw-Hill).

Stefano Crespi-Reghizzi graduated in electrical engineering from Politecnico di Milano, Milan, Italy, and received the Ph.D. degree in computer science from the University of California, Los Angeles, in 1970.

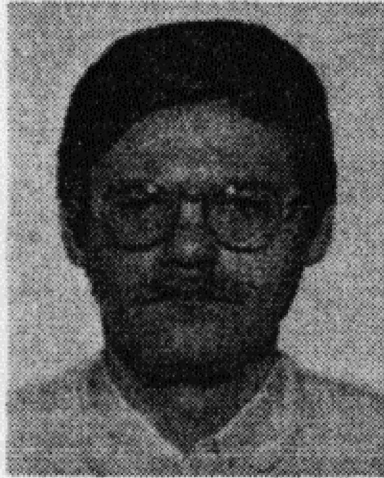
He is a Professor of Computer Science at the Dipartimento di Elettronica, Politecnico di Milano, and the Coordinator of the doctoral program in Computer and Automation Engineering. Previously, he taught formal languages and compilers at the Università di Pisa. His present research interests in research and development focus on the convergence of artificial intelligence, database, and software engineering technologies. He is also engaged in theoretical investigation on formal languages, automata, and semantics.



Andrea Di Maio (S'84–A'85) received the degree in electronic engineering (computer branch) from Politecnico di Milano, Milan, Italy, in 1984.

He is a Software Engineer at TXT SpA, Milan. His interests concern programming languages, design methods and tools, software reuse, and distributed real-time systems.

Mr. Di Maio is a member of the IEEE Computer Society.



Luigi A. Lavazza received the degree in electronic engineering (computer branch) from Politecnico di Milano, Milan, Italy.

He is a member of the research staff at TXT SpA, Milan, currently participating in the ESPRIT project METEOR. His research interests include the study and development of tools for software engineering activities, extensions of relational algebra, and logic programming.