RAPID APPLICATION PROTOTYPING
THE PROQUEL LANGUAGE

Jean-Yves LINGAT
Pierre COLIGNON

THOM'6
33 rue de Vouillé, 75015 Paris, France

Colette ROLLAND

Université Paris I
12 Place du Panthéon, 75005 Paris, France

Abstract

This paper presents PROQUEL, an executable specification language designed for the RUBIS system, an information system development tool.
PROQUEL is at the same time a specification language, data manipulation language, and programming language, making it particularly well suited for prototyping database applications.

After a brief introduction to the RUBIS system, the functionality and the various advantages of the PROQUEL language are detailed.

I Introduction

I.1 Prototyping in Software Development

Prototyping is more and more used in database application development. This is highly justified by the increase in size and complexity of applications. For instance, in our group, we deal with technical software between 500,000 and 1,000,000 lines of code. Such a solution has been chosen in Information System design methodologies (e.g. IDA [BODA83], USE [WASS82]) and in Expert Systems development [HAYE83]. We also made this choice in RUBIS.

The RUBIS environment [LING87a] is composed of a set of tools for application design, specification and prototyping. The applications we are confronted with, are very interactive and resemble real-time systems. That is the reason why RUBIS emphasizes the dynamic aspects of the application and the prototyping of the the future system behavior.

I.2 Analysis of Current Solutions

In most cases, the prototyping language is specific and thus distinct from the development language. Furthermore, the prototyping tool is included in a specific environment which aims to allow the designer to modify his prototype specification. Thus, the designer must manipulate several languages for prototype specification, for the prototype environment and for the target system environment. We think that such diversity impedes efficiency of the designer. For this reason, we have based RUBIS on a single language called PROQUEL.

PROQUEL (PROgramming QUEry Language) is a language which integrates a specification, a data manipulation and a programming language. PROQUEL is fully compatible with SQL and proposes a set of specific statements which extends the query into a more powerful data manipulation request, as is the case for embedded languages (C/SQL, EQUEL...).

PROQUEL is used by the designer during the application specification. It will be seen that this specification is highly modular and can be input in increments. At any time during specification, the designer can query and modify the "specification base" using PROQUEL. The experienced designer can build his own utilities and store them in the "specification base". These utilities can be called from any text within or apart from the specifications.

Prototyping the application permits the progressive refinement of the specification. This leads us to the evolution of the specification into a final solution (as in the USE methodology [WASS82] for instance).

The objective of this paper is to present PROQUEL. Section 2 is devoted to a brief introduction to the architecture and functionality of RUBIS in order to define the context in which PROQUEL is used. Section 3 introduces the principal constructions of the language, and section 4 presents the different uses of PROQUEL during application development. Section 5 details certain technical aspects of the PROQUEL implementation.

II RUBIS Architecture and Functionalities

The architecture of the RUBIS system is presented in figure 1. This displays the three major aspects of the system, which are:
1) The R-Schema (RUBIS-Schema), stored in the Metabase, which describes the database application.
2) The R-Schema design interfaces - the Menu Interface and the Validation Module.
3) The prototyping tools - the Application Monitor, the Event Processor, the Temporal Processor, and the PROQUEL Interpreter.

Each of these three aspects is introduced successively.

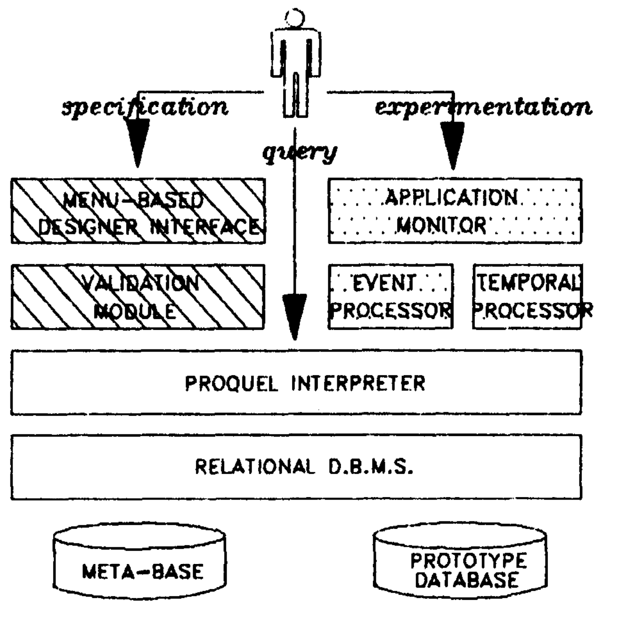


Figure 1: Architecture of the RUBIS system

II.1 The R-Schema

The R-Schema, stored in its relational form in the Metabase, is the focal point of interaction between the application developer and RUBIS. The R-Schema is a modular description of the conceptual schema for the database being developed. This schema is based on the model of the REMORA methodology [ROLL82] [ROLL87], and describes both static aspects (structure) and dynamic aspects (behavior) of the application.

The **static aspects** are modeled using relations representing entities or entity associations in the real world (e.g. client, invoice, loan, etc.).

The **dynamic aspects** are modeled using:
- **Operations** which represent elementary actions on an application object (e.g. add a new client, modify an order, etc.),
- **Events** which represent elementary state changes in the system at which time certain operations must be triggered (e.g. when an order arrives, insert the order into the database, reserve the requested goods, and prepare for delivery). The description of the conditions for the state change is defined in the event predicate. A distinction is made between **external** events (which represent messages received from the real world), **internal** events (which represent elementary state changes of a relation within the database), and **temporal** events (which represent temporal conditions under which certain processing is triggered).

The **temporal aspects** of the application are likewise modeled, using the functions and temporal types of the RUBIS Temporal Model.

The R-Schema is therefore a collection of relations, events and operations defined for an application using PROQUEL specifications. The content of the R-Schema can be illustrated using a graph (fig. 2)

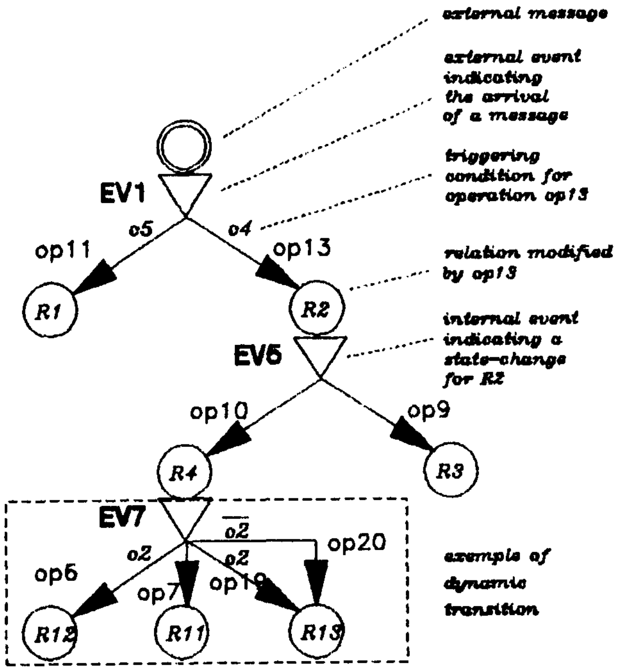

external message
external event indicating the arrival of a message
triggering condition for operation op13
relation modified by op13
internal event indicating a state-change for R2
exemple of dynamic transition
EV1
o5
o4
op11
op13
R1
R2
EV6
op10
op9
R4
R3
EV7
$\overline{o2}$
o2
o2
op20
op6
op7
op19
R12
R11
R13

Figure 2 : Graphic Representation of the R-Schema

Such a representation introduces the **dynamic transitions** of the application, showing their sequence and precedence. A dynamic transition is composed of (1) an event, (2) all the operations triggered by the event, (3) all the relations modified by these operations. This corresponds to an elementary database transaction, since by definition a RUBIS transition is atomic, and must pass the database from one coherent state to another.

II.2 Design Tools

The **Design Menu Interface** allows the insertion, modification, and deletion of the different elements of the R-Schema, using a system of menus which guide the developer during the specification of the application.

The **Validation Module** performs the validation of the R-Schema, detecting the presence of inconsistencies or of specifications which do not meet certain design criteria. Three types of rules are used:
1) **Conformance Rules** which verify the correctness of the specification according to the model, concepts, and language of RUBIS,
2) **Completeness Rules** which insure that there are no isolated or missing objects in the R-Schema,
3) **Accuracy Rules** which detect probable inconsistencies in the R-Schema concerning the accuracy of the specifications as they relate to the actual application. An interactive approach allows the developer to decide whether or not a given situation is actually incorrect.
Certain rules among these are enforced as the specifications are entered (i.e. by the Menu Interface), while others are checked either automatically or on demand at the end of each design session.

II.3 Prototyping Tools

The **Application Monitor** provides the end user interface. It automatically generates data input screens corresponding to each external event defined by the developer. From the specification text, the defined structure of the message to be received serves in the generation of the screen, while the defined predicate represents the condition of message acceptance. During prototype experimentation, the data input screens allow data test cases which can be used to test R-Schema behavior.

The **Temporal Processor** manages all temporal aspects of the application which include:
- handling attributes of type TIME (timestamps, dates, chronological order, calendar conversion, etc.),
- historical processing,
- evaluating expressions of temporal functions and types,
- automatic recognition of temporal events (absolute dates, periodic events, events timed relative to other events, etc.).

The **Event Processor** drives the prototype. It allows the automation of the R-Schema execution by sequencing and synchronizing dynamic cycles that includes:
- handling external and temporal events,
- evaluating operation triggering conditions,
- controlling the execution of operations for which the triggering condition is true,
- recognition of internal events,
- the management of transaction aspects of the application.

The **PROQUEL Interpreter** is used by each of the other modules, in particular by the Event Processor for the execution of operation, condition, and event predicate texts.

The **Relational DBMS** [BOUC81] is the foundation of the RUBIS system. It manages the relations in the prototype database, as well as in the Metabase, which contains the R-Schema.

The detailed operation of the Event Processor and Temporal Processor is described in [LING87b]. The following sections of this paper are devoted to the description of the PROQUEL language and its various uses in database application development.

III Presentation of the PROQUEL Language

We restrict ourselves here to an introduction of the principal syntactic structures of PROQUEL illustrated by several examples. Backus-Naur notation is used to describe certain grammatical structures. The language keywords are noted in upper case.

III.1 The SQL Core of PROQUEL

Each form of standard SQL [SQL86] is valid in the PROQUEL language. In addition, several extensions and facilities have been implemented, notably:

1) The assignment of the result of a SELECT to a relation:
SELECT *projection list* **INTO** *result relation*
FROM *relation list* WHERE *criteria* ;

Three cases may arise:
- if *result relation* does not exist in the database, its schema is deduced from the SELECT <projection list>.
- if *result relation* already exists but its schema is incompatible with the SELECT <projection list>, its schema becomes that of the SELECT result and its previous contents are **replaced** by the SELECT result.
- if *result relation* already exists and its schema is compatible with the SELECT result, the result relation keeps its previous schema but its previous contents are lost. The use of the ADDTO option in place of INTO **appends** the SELECT result to the existing contents.

If the name of the *result relation* begins with "#", PROQUEL considers it to be a temporary relation which automatically disappears at the end of the current transaction. We will see in the next section the utility of this concept.

2) The search for a tuple qualified by its ordinal position in the result of the SELECT:
SELECT [*position* **FROM**] **{FIRST | LAST} TUPLE** *projection list*
FROM *relation list*
[WHERE *criteria*]
[ORDER BY *attribute list*];

For example, to find the name and address of the second to last subscriber in the relation:
SUBSCRIBER (**NAME**, ADDRESS, SUBSCRIPTION_DATE, ...)

the command is:
```
SELECT 2ND FROM LAST TUPLE name, address
  FROM subscriber
  ORDER BY subscription_date;
```

3) The statement **EXISTS** *relation* [WHERE *criteria*]; proves very useful in certain cases, and avoids the need for a SELECT COUNT (*) FROM (*relation*) != 0.

4) In PROQUEL, the operands of any operation are not constrained to be a base relation only, but can rather be any expression which yields table. This makes up for the lack of orthogonality of SQL [DATE86].

III.2 Variables

PROQUEL variables are identified by a name beginning with "$" ($name, $b5, etc.); a variable must be declared before it is used. The assignment of a value to a variable is possible only when the value and the variable are of compatible types; the syntax is that of PASCAL ($name: = "foobar", $b5: = 4*99 + $a).

1) Predefined Types

PROQUEL possesses the types most generally used in database systems INTEGER, REAL, STRING, TEXT, and BOOLEAN. It provides for shortcomings generally encountered for temporal expressions by offering:
- type TIME, which specifies an absolute time based on the gregorian calendar extended to include hours, minutes, and seconds. Absolute time can be expressed at any level of abstraction, from the year (1988) down to the second (1988/04/18:15h22m13s),
- type INTERVAL, which specifies a time interval (e.g. [1988/04-1988/05]),
- type DURATION, which specifies elapsed time between two points in time (e.g. 3hours10minutes),
- type PERIOD, which specifies a periodic interval with its initial starting point (time point) and its frequency (duration).

2) Functions and Operators

PRINT allows the display of variable values and character strings.
e.g. `PRINT "The name of client number ",$num," is:",$name;`

ACCEPT...PROMPT allows the input of variable data following a prompt message.
e.g. `ACCEPT $loan PROMPT "Enter the loan amount:";`

Arithmetic, comparison, and boolean operators (+, -, *, /, DIV, MOD, <, >, < =, > =, =, ! =, NOT, AND, OR).

For character string manipulation:
- LENGTH($c) : returns the length of the character string;
- SEARCH($c1,$c2) : returns the position of the first occurrence of string $c2 in $c1;
- $c[i,j] : extracts the substring from character position i to character position j;
- $c & $d : concatenates two strings;
- $c BW *string* : returns TRUE if $c begins with *string*;
- $c EW *string* : returns TRUE if $c ends with *string*;
- $c IN *string* : returns TRUE if $c is in *string*.
- $c CONTAINS $d : returns TRUE if $c contains $d.

Type conversion is accomplished using a "casting" mechanism the syntax of which is borrowed from the C language.
e.g. `$i: = (INTEGER)"2187" + 33;`

Provided for the temporal types are the operators =, * and +, which are redefined for these types, along with keywords AFTER, FROM, UNTIL, EVERY, AT, and EACH, allowing the construction of temporal expressions [NOBE88].

3) Variable usage in the SQL forms

Apart from their role in algorithmic programming, the principal interest of PROQUEL variables is their power to be used anywhere in a SQL query, including as relation name, attribute name, and result relation. The following is an example:
```
VAR $name: STRING; VAR $res : INTEGER;
ACCEPT $name PROMPT "Find cardinality for which relation ?:";
$res: = SELECT COUNT(*) FROM $name;
PRINT "The result is:",$res;
```

4) The types TUPLE and RELATION

Variables of type TUPLE allow, for example, the capture of the result of a SELECT...TUPLE... in a **single variable**, as opposed to embedded languages (PL1/SQL, C/QUEL,...) which require a unique variable for each field in the result.

The access to the value of **tuple variables** is accomplished by using a dot notation, postfixing the name of the variable with the name of the desired field (as for PASCAL record fields). In addition, the structure of a tuple variable is determined dynamically from the schema of the assigned value, offering a great deal of flexibility.
```
VAR $member: TUPLE;
$member: = SELECT FIRST TUPLE * FROM roster
                      ORDER BY age;
PRINT "The age of the youngest member is ", $member.age;
PRINT "The youngest member is ", $member.name;
```

A **relation variable** contains the name of the database relation to which it refers. This type of variable can be used in place of relation name in any valid statement. Thus the following copies the contents of the relation EMPLOYEES into the relation referred to by $var_rel.
```
VAR $var_rel: RELATION;
$var_rel: = EMPLOYEES;
```

Access to the name of the relation to which the relation variable refers is accomplished by the intermediate use of the unary operator **NAME OF**. Thus the following copies the name "EMPLOYEES" into the variable $var_rel itself, which now refers to the actual EMPLOYEES relation and no longer a copy:
```
VAR $var_rel: RELATION;
NAME OF $var_rel: = "EMPLOYEES";
```

III.3 Algorithmic Structures

As in any high-level programming language, PROQUEL permits the traditional constructions:

IF *boolean expr* **THEN** *statement* [**ELSE** *statement*];
WHILE *boolean expr* **DO** *statement*;

To facilitate the manipulation of relations within the program, PROQUEL includes the construction:

FOR EACH *tuple var* **IN** *relation* [**UNTIL** *bool expr*]
DO *statement*;

This construction can be found in languages such as PASCAL/R [SCHM80], and allows the iteration of processing for each tuple in a given relation (base or calculated relation). The following example illustrates the advantage of this construction:

```
FOR EACH $tup IN members
DO BEGIN
  IF $tup.sex = "m"
  THEN PRINT "Mr. ",$tup.name
  ELSE IF $tup.married = "n"
    THEN PRINT "Miss ",$tup.name
    ELSE PRINT "Mrs. ",$tup.name;
  PRINT "lives at ",$tup.address;
END;
```

III.4 Dynamic Interpretation

The PROQUEL language allows dynamic evaluation of program text. The built-in function **EVAL**(*string*); takes as an argument a character string containing statements to be immediately interpreted.

For example, the automatic display of the result from a selection on the relation CLIENT with the selection criteria input on the fly can be specified as:

```
VAR $crit: STRING;
ACCEPT $crit PROMPT "Selection criteria ?:";
EVAL("PRINT SELECT * FROM client WHERE "& $crit &";");
```

At run time, EVAL interprets its argument as PROQUEL text. Thus variables in the EVAL argument can themselves contain variable names. For instance, using the current example, the following criteria can be entered, assuming $retirement_age and $standard_condition are previously declared and initialized variables: age > $retirement_age AND $standard_condition.

III.5 Procedure and Function

The concepts of procedure and function (a procedure which returns a value) exist in PROQUEL. The syntax for procedure, function and parameter declaration, as well as the rules concerning the visibility of local variables are the same as in PASCAL. Parameters and function return values may be of TUPLE or RELATION types. As in most languages, parameters can be passed by value or by reference.

During the declaration of a procedure, the PROQUEL interpreter verifies the syntax of the body of the procedure. Detected errors produce explanatory error messages. If the definition is valid, the PROQUEL interpreter stores in the Metabase various information along with the text of the procedure. This text is executed at each procedure call. Once stored, this data persists from one session to the next, permitting developers to build their own utility libraries.

III.6 Transactions

During the specification of the application, the developer never manipulates transactions: he just specifies the dynamic transitions which will be managed automatically by the Event Processor as transactions. However, PROQUEL provides statements which allow explicit processing of transactions and may be useful during the construction of utilities or predefined procedures.

In PROQUEL the concept of **transaction** is independent from the procedure or function. The only statements which define a segment of code corresponding to a database transaction are **COMMIT** and **ABORT**. These two statements signal the end of the current transaction, and invoke (1) a write to disk (for a COMMIT) or abandonment (for an ABORT) of all current database work performed during the transaction, (2) the destruction of all temporary relations used during the transaction, (3) the release of all **temporary locks** set by the DBMS on the relations accessed during the transaction (in read or write mode), and (4) the start of a new transaction.
These two statements are forbidden in all texts which are called during a dynamic transition (predicate, condition, operation).

To complement the automatic lock processing within the transaction, PROQUEL includes the statements:

LOCK *relation* **FOR** { SELECT | UPDATE };
RELEASE { *relation* | **ALL** };

These statements allow the management of **persistent locks**, independent of the transactional partitioning of the program.

IV Application Development Using PROQUEL

In this section, we present through examples the various uses and contributions of PROQUEL during the development of database applications.

IV.1 PROQUEL as a Specification Language

The PROQUEL constructions previously introduced are used in particular for the specification of the elements of the R-Schema (e.g. relations, operations and events). Each element of the R-Schema is specified independent of the others, as a module which may be added at any time to the Metabase. The R- Schema description for an application can be built in increments. For example:
- The static subschema can be first input as a collection of relations (using **DEFINE RELATION**),

- The architecture of the dynamic subschema can then be input by specifying the various dynamic transitions (using **DEFINE EVENT**),
- The elements of the dynamic transitions, in particular the operations, can be specified independently (using **DEFINE OPERATION**).

We illustrate this process by considering the framework for an automated subscription-library management system. The examples introduced center around the following relational schema:

```
  BOOK (BOOK#, PUBLISHER, TITLE)
  COPY (BOOK#, COPY#, ACQDATE, PRICE, CP_STATUS)
  SUBSCRIBER (SUBSC#, NAME, ADDRESS, SUBDATE,
              SUBSC_STATUS, NUMCOP)
  REQUEST (REQ#, SUBSC#, REQDATE, REQTYPE,
              BOOK#, REQ_STATUS)
  LOAN (LOAN#, LOANDATE, BOOK#, COPY#, REQ#)
  NOTICE (NTC#, NTC_DATE, LOAN#, SUBSC#)
```

The meaning of the attributes (when not obvious) will be given in the examples.

Figure 3 associates: (1) the current loan agreement in force at the library, (2) the graphical representation of the corresponding dynamic transition, and (3) the PROQUEL specification. The specification is a translation into the PROQUEL language of the dynamic schema, which is itself a natural model of the loan agreement.

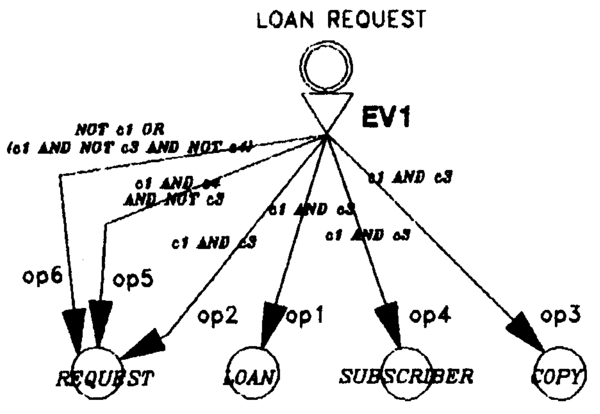
The specification of the external event "loan request arrival" is composed of three parts:
1) The **message** "LOAN REQUEST" is described in **PROP**. It consists of a request number, a book number, the subscriber number, and the request type (immediate, or hold).
2) The **event** predicate is specified in **PRED**. In the example, the "LOAN REQUEST" message is acceptable only if both the requested book and the subscriber are present in the database.
3) The **TRIGGER** specifies the result of the arrival of the loan request. Three cases are possible:
- The request can be satisfied (c3: a copy of the book is available), and the subscriber is in good standing (c1: his subscription is up to date, he has no outstanding late notice, and he has less than three books currently on loan). In this case, a loan is created (op1), the request is accepted (op2), the status for the book is set to "ON LOAN" (op3), and the number of copies on loan for this subscriber is incremented (op4).
- The request cannot be satisfied (there is no available copy), but the subscriber is in good standing and wishes to leave his request on hold (c4). The demand is put on hold (op5).
- The subscriber is not in good standing, or the request cannot be satisfied for an immediate request. In this case, the request is refused, but is still added to the database for statistical purposes (op6).

The specification of an event in PROQUEL defines the structure of the associated dynamic transition. The elements defined in the TRIGGER (conditions and operations) are defined separately. This allows a progressive and modular description of the application. In addition, the same condition or operation may be shared by several dynamic transitions.

TEXT SPECIFICATION

- A subscriber may not borrow more than three books;
- A loan request is receivable if the subscriber is valid (his subscription is up to date and he has no overdue books);
- If a request is type "hold", the unavailability of the requested book causes the request to be put on hold.

GRAPHIC SPECIFICATION



PROQUEL SPECIFICATION

```
DEFINE EVENT ev1 IS request_arrival
 ON MESSAGE
 COMMENT "Arrival of a loan request"
 PROP { num_req: INTEGER;
        num_book: INTEGER;
        num_subsc: INTEGER;
        type: (IMMEDIATE, HOLD); }
 PRED {
  (EXISTS book WHERE book# = CONTEXT.num_book) AND
  (EXISTS subscriber WHERE subsc# = CONTEXT.num_subsc)}
 TRIGGER {
  IF c1 AND c3 THEN { op1 ON loan;
                      op2 ON request;
                      op3 ON copy;
                      op4 ON subscriber; };
  IF c1 AND NOT c3 AND c4 THEN op5 ON request;
  IF NOT c1 OR (c1 AND NOT c3 AND NOT c4)
    THEN op6 ON request; };
```

Figure 3: Dynamic Transition Specification

For example, figures 4 and 5 show the specification of condition c1 and of operation op1.

Each specification is a module independent of the others. The variables declared in a module have scope within that module. All modules receive an implicit call parameter referenced by the keyword **CONTEXT**. This parameter designates the message/tuple for which the arrival/state-change generated the event. In the specifications of ev1, op1 and c1, CONTEXT represents the loan request message.

```
DEFINE CONDITION c1 IS good_standing
COMMENT "The subscription is up to date, no pending late
     notices, and less than three books on loan"
TEXT {
 VAR $status: STRING;
 VAR $ncopy: INTEGER;
 VAR $nolate: BOOLEAN;
 SELECT UNIQUE subsc_status, numcop INTO $status, $ncopy
  FROM subscriber WHERE subsc# = CONTEXT.num_subsc;
 $nolate:= NOT EXISTS notice
              WHERE subsc# = CONTEXT.num_subsc;
 RETURN ($status="VALID" AND $ncopy<3 AND $nolate) };
```

Figure 4: Specification of a Trigger Condition

When a distinction is necessary between the old and new value of the context (in the case of an internal event, for instance), the prefixes **OLD** and **NEW** are used.
The context of a dynamic transition is in fact an implicit parameter passed to all PROQUEL texts which it contains. Not having to declare each passage of the context by parameter clearly simplifies the developer's specifications.

A condition corresponds to a boolean function. Its text is terminated by the RETURN statement, which determines the return value of the condition.

```
DEFINE OPERATION op1 IS ins_loan
 COMMENT "Create loan"
 TYPE insert IN loan
 INPUT ()   (* no explicit parameters *)
 TEXT { VAR $copy, $max: INTEGER;
  $copy:= SELECT FIRST copy# FROM copy
            WHERE book# = CONTEXT.num_book
             AND cp_status = "AVAILABLE";
  $max:= SELECT UNIQUE MAX(loan#) FROM loan;
  INSERT INTO loan ($max+1, current_date,
            CONTEXT.num_book, $copy, CONTEXT.num_req);
 };
```

Figure 5: Specification of an Operation

Each operation has an implicit exit parameter: the two successive values of the tuple it modifies. If the state change generated by the execution of the operation generates an internal event, this parameter serves as the context for the dynamic transition of the generated event.

Figure 4 illustrates the contribution of the combined usage of variables and SQL queries. In the usual application development environment, the developer must resort to a language using embedded SQL, with all the awkwardness this implies.

The Metabase is incremented progressively with the arrival of new specifications. The relational mapping of the R-Schema is thus automatic. We will see in the following paragraph the easiness of the use of such a representation.

IV.2 PROQUEL as (meta) Data Manipulation Language

The specification work for a database application is long and iterative. The "Specification Base" is progressively elaborated. To help the developer with his task, it is essential to provide him with the means of querying and manipulating this Specification Base. Such possibilities are not generally offered by current DBMS, except for the description of the structure of the relations (e.g. ORACLE or INGRES metabase). The recourse to a data dictionary (and thus to a new language) is therefore necessary.
We feel that it is valuable for the developer to have a single language for both specification and manipulation of specifications. In the following paragraphs, we present three aspects of PROQUEL which illustrate (1) query of the Meta-schema, (2) interactive control of the R-Schema and (3) construction of utilities.

1) Query of the Meta-schema

During the specification phase, the developer can use PROQUEL to interactively query the Metabase. For example, he can locate an element of the specification which he has forgotten, or which is supplied by another person.

```
EVENT (EVT_NAME, FULL_NAME, EVT_CATEG,
              RELATED_OBJECT...)
OPERATION (OPE_NAME, FULL_NAME, MODIFIED_OBJECT,
              OPE_TYPE...)
TRIGGER (EVT_NAME, OPE_NAME, COND_NAME,
              FACT_NAME...)
  :
OPE_TEXT (OPE_NAME, TEXT)
  :
INDUCTION (EVT_NAME, IND_EVT)
RESOURCES (COMP_NAME, REL_NAME, ACCES_TYPE)
  :
AUTHOR (COMP_NAME, DESIGNER, BIRTH_DATE,
              LAST_MODIF_NAME, STATUS...)
VALIDATION (VAL_NUM, VAL_DATE, RESULT...)
```

Figure 6: Extract of the structure of the Metabase

In RUBIS, the Meta-schema is a relational base of which figure 6 gives an extract. Globally, the RUBIS Metabase consists of three types of information to which the developer has access:
- information corresponding to the relational mapping of the R-Schema (descriptions of the various components),
- information derived from the R-Schema (resources accessed in read/write mode by each component, list of possible event sequences, etc.),

- informations relative to development progress (list of dynamic transitions completely specified, validation results, etc.).

Some explanations:
- EVT_CATEG gives the category of the event, that is the type of the operation that can generate it (DELETE, UPDATE, INSERT),
- RELATED_OBJECT gives the name of the object of which the event recognizes the state-changes (a relation, CALENDAR, MESSAGE),
- INDUCTION contains the event couples which may be chained.

Using this relational representation of the R-Schema, the developer can formulate interactive queries such as "What are the events that trigger delete operations on the relation NOTICE ?"

```
SELECT evt_name FROM trigger, operation
  WHERE modified_object = "NOTICE"
    AND ope_type = "DELETE"
    AND ope_name = ope_name;
```

Figure 7: Example of Metabase query

This query returns a relation containing the names of the events, and displays them in table form. If the developer would also like the names of the operations and their trigger conditions, he simply adds the attributes ope_name and cond_name to the projection attribute list.

In the same way, still asking questions on the Metabase, the designer will be able to ask questions such as:
"Which events precede the event "newly_available" ?" or
"Who is the developer for operation "insert_loan" ?"

2) Interactive control of the R-Schema

During the development of the schema, the developer often needs to determine the correctness of his description. He can of course use the Validation Module, but he can also pose queries which allow him to detect problems immediately. For example:
- Are all components of the dynamic transition corresponding to the "notification_response" notified ?

```
NOT EXISTS trigger
 WHERE evt_name = "notification_response" AND
  ((ope_name NOT IN SELECT ope_name FROM operation) OR
   (cond_name NOT IN SELECT cond_name FROM condition));
```

- Do any events trigger an insert operation for the NOTICE relation ?

```
EXISTS trigger, operation
 WHERE ope_type = "INSERT"
  AND modified_object = "NOTICE"
  AND ope_name = ope_name;
```

3) Construction of Utilities

In certain cases, the developer may wish to construct specific utilities. He can make use of the concepts of procedure and function in PROQUEL. In this way, he can generalize queries, building them into a library of procedures.
For example, the function modified_by (fig. 8) generalizes the query in figure 7, allowing the search of the events and the triggering conditions of the operations of a given type which modify a given relation. The function returns a relation. Once specified, the function is stored in the Metabase and can be used as a predefined utility within another PROQUEL text.

```
FUNCTION modified_by ($relname, $typ: STRING): RELATION;
 BEGIN
 IF ($typ="INSERT" OR $typ="DELETE" OR $typ="UPDATE")
 THEN RETURN (SELECT evt_name,cond_name,ope_name
         FROM trigger, operation
         WHERE modified_object = $relname
          AND ope_type = $typ
          AND ope_name = ope_name )
 ELSE PRINT "Valid types: INSERT,DELETE and UPDATE";
 END;
```

Figure 8: Example of utility

Note that to facilitate the work of developers, a certain number of builtin functions have been predefined in the RUBIS library:

FUNCTION triggered_by ($ope_name: STRING):RELATION;
which returns a relation containing the names of events and the triggering conditions of the specified operation.

PROCEDURE print_events ($filename: STRING);
which reconstructs in the specified file, all the PROQUEL specifications of the currently specified events.

FUNCTION complete_event ($evt_name: STRING):BOOLEAN;
which validates the completeness of the definition of a dynamic transition by verifying that all objects used directly (conditions, operations, relations accessed by the predicate), or indirectly (relations accessed by the texts of the conditions and operations) have been previously defined.

PROCEDURE trans_closure
 (VAR $rel_res,$rel_init:RELATION;$att1,$att2:STRING);
which carries out the transitive closure of a relation on two of its attributes.

IV.3 PROQUEL as an Integrated Programming Language

To construct more elaborate texts, the developer requires algorithmic possibilities offered by classic programming languages. The integration of the functionality of classical programming languages and data manipulation languages allows PROQUEL to make up for the inability of languages such as SQL to handle problems such as complex calculations, sophisticated document production, and iterative or recursive algorithms. We will illustrate this last aspect of PROQUEL with two examples.

Example 1: Transitive closure.

The first example presents a procedure which performs the transitive closure of a specified relation on two of its attributes (using the "semi-naive" algorithm).

```
PROCEDURE trans_closure (VAR $res, $init:RELATION;
                         $att1, $att2:STRING);
 BEGIN
 #ref:= SELECT $att1, $att2 FROM $init;  (*reference relation*)
 #new:= #ref;      (* contains the newly generated tuples *)
 CLEAR ($res);

 WHILE NOT EMPTY (#new)
 DO BEGIN
   $res:= $res UNION #new;
   #new:= (SELECT #new.$att1,#ref.$att2 FROM #new, #ref
            WHERE #new.$att2 = #ref.$att1) MINUS $res;
   END;
 END;
```

This example calls for several comments about:
- Parameter passing by value or reference for variables of type relation. Here, $init is passed by reference to avoid a useless copy of the initial relation.
- The use of RELATION and STRING variables in SQL statements as name of relation and name of attribute - this allows the coding of queries with truly variable parameters.
- The use of the temporary relations #ref and #new which inherit dynamically the schema of the result relation, and which disappear automatically at the end of the current transaction - this mechanism avoids the explicit creation and destruction of intermediate relations, for which the schema must be known in advance.

All these aspects, along with the trivialization of database calls in the language, the integration of types TUPLE and RELATION with the other data types, and the implementation of specific internal mechanisms, contribute in making PROQUEL a language which is flexible, readable, and concise for the implementation of database applications.

For comparison, the figure below shows the transitive closure algorithm as written for the ORACLE C precompiler [ORAC87].

```
void transitive_closure (res, init, att1, att2)
char *res, *init, *att1, *att2;
{ EXEC SQL BEGIN DECLARE SECTION;
  int c; char ins1 [80], ins2 [80], del1 [80];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL CREATE TABLE REF (A1 NUMBER, A2 NUMBER);
  EXEC SQL CREATE TABLE NEW (A1 NUMBER, A2 NUMBER);
  EXEC SQL CREATE TABLE GEN (A1 NUMBER, A2 NUMBER);
  EXEC SQL CREATE TABLE RES (A1 NUMBER, A2 NUMBER);

  strcpy (ins1,"INSERT INTO REF SELECT ");
  strcat (ins1, att1); strcat (ins1, " , "); strcat (ins1, att2);
  strcat (ins1, " FROM "); strcat (ins1, init);
   /* INSERT INTO #ref SELECT $att1,$att2 FROM $init; */
  EXEC SQL EXECUTE IMMEDIATE:ins1;
  EXEC SQL INSERT INTO NEW SELECT * FROM REF;

  EXEC SQL SELECT COUNT(*) INTO:c FROM NEW;
  while (c)
   { EXEC SQL INSERT INTO RES SELECT * FROM NEW;
    EXEC SQL INSERT INTO GEN
             SELECT NEW.A1, REF.A2 FROM NEW, REF
              WHERE NEW.A2 = REF.A1;
    EXEC SQL DELETE NEW;
    EXEC SQL INSERT INTO NEW SELECT * FROM GEN
                 MINUS SELECT * FROM RES;
    EXEC SQL DELETE GEN;
    EXEC SQL SELECT COUNT(*) INTO:c FROM NEW;
   }

  strcpy (del1, "DELETE "); strcat(del1, res); /* DELETE $res */
  EXEC SQL EXECUTE IMMEDIATE:del1;
  strcpy (ins2, "INSERT INTO "); strcat(ins2, res);
  strcat (ins2, " SELECT * FROM RES");
  EXEC SQL EXEC IMMEDIATE:ins2;
  EXEC SQL DROP TABLE RES;
  EXEC SQL DROP TABLE GEN;
  EXEC SQL DROP TABLE NEW;
  EXEC SQL DROP TABLE REF;
}
```

We don't use here the CONNECT BY clause of SELECT statement proposed by ORACLE as an extend of SQL because this clause is not well suited for implementing the transitive closure problem: because the concision is not better and it generates an execution error when detecting a cycle. Finally, we use the same algorithm in order to facilitate the comparison of the two languages.

The last example leads us to the following remarks on the integration of SQL in the host language.

- We see here that lines destined for the preprocessor must be prefixed by EXEC SQL.

- Variables used in the SQL statements must be declared in a particular fashion. Furthermore, it is impossible to use these variables directly as name of relation or attribute in the SQL queries - this is possible only in the form EXECUTE IMMEDIATE, the use of which is more constrained.

- In the absence of a mechanism similar to that of temporary relations in PROQUEL, explicit declaration and destruction of intermediate relations is necessary, for which the names and types of the attributes must be known in advance. (This has led us in this example to limit the use of the procedure to attributes of type NUMBER to avoid obscuring the code.)

These aspects contribute to the awkwardness of programs written in embedded SQL [CHRI87].

Example 2: Part explosion problem.

The second example illustrates the recursive aspects of PROQUEL.
The use of procedures and functions allows the translation into PROQUEL of most recursive algorithms. To illustrate this aspect, we reproduce one of the examples cited in [ATKI85] and [GAME87], which compared the behavior of different database programming languages for the same problem. The example given is of the processing of an inventory containing:
- base parts, described by name, price, and weight,
- composite parts, described by name, the surcharge and additional weight resulting from assembly, and a list of their subparts.
The schema of the corresponding relational database is:

```
 BASE_PARTS (NAME, PRICE, MASS)
 COMPOSITE_PARTS (NAME, ASMBLY_PRICE, ASMBLY_MASS)
 SUB_PARTS (NAME, SUBNAME, QTY)
```

The following procedure corresponds to a PROQUEL version of one of the request types in the papers mentioned above, and consists of determining the total weight and price of a specified product. It uses a recursive algorithm which traverses the complex product tree to find its weight and price.

```
PROCEDURE price_mass ($name:STRING;
            VAR $price, $mass:REAL);
 VAR $subprice,$submass:REAL;
 VAR $tup_comp,$sub:TUPLE;
 VAR $components:RELATION;
 BEGIN
 SELECT UNIQUE price, mass INTO $price,$mass
    FROM base_parts WHERE name = $name;
 IF $price = NULL     (* it is not a base_part *)
 THEN BEGIN
  SELECT UNIQUE TUPLE * INTO $tup_comp
   FROM composite_parts WHERE name = $name;
  IF $tup_comp = NULL
  THEN PRINT "This part doesn't exist"
  ELSE BEGIN
   $price: = $tup_comp.asmbly_price;
   $mass: = $tup_comp.asmbly_mass;
   $components: = SELECT subname, qty FROM sub_parts
             WHERE name = $name;
   FOR EACH $sub IN $components
   DO BEGIN
    price_mass ($sub.subname, $subprice, $submass);
    $price: = $price + $sub.qty * $subprice;
    $mass: = $mass + $sub.qty * $submass;
   END;
  END;
 END;
END;
```

This procedure illustrates an interesting aspect of the behavior of relation variables: when a result relation is assigned to a relation variable (e.g. $rel3: = SELECT...) before the name of a relation has been assigned to it (using the statement NAME OF $rel3: = ...;), the interpreter generates unique temporary relation name assigned to the variable. This is particularly useful to avoid side effects in recursive procedures using local variables of type RELATION.
In this example, it is in fact impossible to replace the variable $components by a temporary relation because the contents of that relation would be lost at each recursive call. The use of the relation variable $components guarantees the generation of a new local temporary relation at each call.

IV.4 PROQUEL as a Development Language

The language provided by the RUBIS system must permit the rapid prototyping of applications. It is for this reason that we have defined PROQUEL as an interpreted language. We have in effect chosen to focus on the development time, rather than response time which is not the primary criterion for a prototype.

The PROQUEL interpreter provides an interactive mode which executes instructions typed at the keyboard, as well as a batch mode which allows the execution of text files. The conversational interface offers the developer an environment favorable to the development of his application:
- It is possible to trace certain variable and procedure calls. There is a step-mode of execution, to follow precisely the execution of the PROQUEL text. The developer can display, after each instruction, the values of variables which are not being traced systematically, and query or modify the contents of the database.
- For the global development of the application, the Event Processor allows a verbose mode, enabling the developer to follow the execution of his application. At any moment, he can interrupt a given test run to query or modify the database (using the interactive PROQUEL interface), and then resume the execution from the point of interrupt.

IV.5 PROQUEL as a Language for RUBIS Implementation

Certain parts of the RUBIS system are written in PROQUEL. For example, the rules for the Validation Module (cf fig. 1) are coded in the form of PROQUEL procedures which analyze the Metabase.

The ease with which PROQUEL describes the most complex algorithms applied to a relational database proves the capacity of the language to serve as specification language for any type of processing. It offers a desirable alternative to languages of the type embedded SQL (rigid and complex) or of the type "SQL + pseudo-code" (informal and anarchic).
For example, most of the new algorithms introduced into the RUBIS system are now specified in PROQUEL before being implemented (as in the case of the Event Processor).

V The PROQUEL Implementation

The PROQUEL interpreter is written in C under UNIX. The specialized tools LEX and YACC are used to generate the lexical analyzer and parser.

V.1 Principles of Interpretation

The PROQUEL interpreter is based on a two-pass mechanism, which allows the duality of specification/execution and simplifies the implementation of the various functional modes presented above.

The **first pass** performs the lexical analysis and parse of the source text. Incorrect syntax generates error messages, while valid statements are translated into equivalent elementary code which is directly executable. This code substitutes for example procedural statements (conditions, loops) into elementary test-and-jump equivalents.

The **second pass** of the interpreter executes the generated code. The execution is performed by the **Rubis Virtual Machine**, designed to minimize the work of the interpreter.

V.2 The Advantage of Two-Pass Interpretation

This mechanism of two-pass interpretation allows, by focusing on the interaction between the recognition phase and the execution phase, the easy implementation of PROQUEL's different modes of operation.

1) In **interactive mode**, each PROQUEL statement constitutes a unit of execution. The statements which the user types at the keyboard are decoded and executed one by one. The execution of a statement is interrupted on error detection, and an error message is displayed.

2) In **text mode**, the source text resides in a file. The entire text is analyzed and the errors are globally signaled to the user with appropriate error messages. When the source text is free of errors, the code generated by the first pass is taken over by the Rubis Virtual Machine and executed. In text mode the entire text constitutes a unit of execution.

3) The specification of predicates, conditions, operations, and procedures makes use of the **precompile mode**. The interpreter performs in this case the first pass (which performs parsing) but does not execute the code. When the specification of an operation is entered into the Metabase, the interpreter passes from interactive or text mode into precompile mode to validate the operation text. If the analysis is successful, the generated code is admitted into the Metabase with the rest of the operation definition. This improves the performance of the RUBIS system as the lexical and syntactic analysis is performed once during the specification rather than at each invocation.

4) The **execution mode** complements the precompile mode. The interpreter receives not PROQUEL source text, but generated code and this time performs only the second interpreter pass. This mode, therefore, supports the execution of operations, predicates, conditions, and procedures which have been previously precompiled during their specification.

V.3 The Rubis Virtual Machine

The Rubis Virtual Machine (RVM) is organized around a flexible and powerful kernel built with simple internal mechanisms, capable to support both present and future functionality of the language.

The kernel manages in particular **registers** containing the most general system data (user name, current transaction number, execution context system variables, etc.), and a **stack** containing, during execution, dynamic "system data" (loop indices, etc.) and intermediate evaluation results (elementary such as booleans, numbers, strings, or complex such as arrays, lists, sets, tuples, relations, etc.).

The **instructions** are constructed around the RVM kernel, and implement the functions of the PROQUEL language. The instruction set of the RVM is very rich and possesses a number of specialized functions. To preserve acceptable response times, we have not hesitated to code in C the most complex internal functions.

PROQUEL offers a greater tolerance with a less strict type control during the first pass of interpretation. This is highly useful for the preanalysis of queries, as the use of variables as names of relations or attributes is usually forbidden in database calls under rigorous type control (this problem has been discussed in [ALLM76]).

Most of the operators in the RVM are polymorphic. They adapt their behavior to the type of objects to which they are applied. PROQUEL uses this feature to allow automatic type conversion. This permits, for example, to use character string variables as names of relations.

VI Conclusion

We have presented the PROQUEL language, which allows the construction of an executable specification of an application and permits the use of the generated prototype in trial run experimentation.

The total integration of database/specification/programming language aspects makes PROQUEL a language highly adapted for the development of database applications. PROQUEL is in the line of the new Fourth Generation Languages (e.g. INGRES OSL) and offers a viable alternative to the complexity of embedded languages [CHRI87].

An initial version of PROQUEL is running on VAX and SUN systems under UNIX. Current developments are leading towards:

- The evolution of the language to handling more complex data structures (tuple arrays, heterogeneous lists, etc.),
- The creation of interfaces allowing the use of PROQUEL in association with other relational DBMS,
- The provision for the implementation of code generators which translate the PROQUEL specification into a target

"embedded language" (PASCAL/SQL, C/QUEL, etc.). RUBIS will be not only a generator of prototypes, but also a generator of complete applications,

- Allowing PROQUEL to become a real "persistent" programming language [BUNE86] having the possibility to (1) retain the value and structure of a variable beyond the bounds of a program or procedure (such variables will be "meta-variables", accessible to any PROQUEL text as is the case currently for database relations), and (2) maintain, in the same fashion as for variables, the definition of a type. The solution is an integration of PROQUEL data types and domain types of database attributes.
- A study of the possibilities of a "PROQUEL as a Data Type", inspired by the INGRES [STON84] and POSTGRES [STON86] solutions.

References

[ALLM76] ALLMAN E, STONEBRAKER M: "Embedding a Relational Data Sublanguage in a general purpose programming language" ACM SIGPLAN-SIGMOD Conference on Data, Salt Lake City, Utah, March 1976.

[ATKI85] ATKINSON M.P, BUNEMAN O.P: "Database Programming Languages" Technical Report 10-85, Univ. of Pensylvania, 1985.

[BODA83] BODART F, HENNEBERT A.M, LEHEUREUX J.M, MASSON O, PIGNEUR Y: "A System for Requirements Specification, Prototyping and Simulation" Proc. IFIP-TC2 WC on System description methodologies, Kecskemet, Hungary, North Holland 1983.

[BORG85] BORGIDA A.: "Features of Languages for the development of Information Systems at the Conceptual Level" IEEE Software, Vol. 2, Nx1, Jan. 1985.

[BOUC81] BOUCHET et al.: "Databases for Microcomputers : the PEPIN approach" ACM SIGMOD/SIGSMALLS, Orlando, Florida, Oct. 1981.

[BROD82] BRODIE M.L, SILVA E: "Active and Passive Component Modeling" in [CRIS 1].

[BUNE86] BUNEMAN P, ATKINSON M: "Inheritance and Persistence in Database Programming Languages" ACM SIGMOD Conf. on Management of Data, Washington D.C, May 1986.

[CHRI87] CHRISTENSEN A, ZAHLE T.U: "A Comparison of Self-contained and Embedded Database Languages" 13th Conf. on VLDB, Brighton, Sept. 1987.

[CRIS 1] "Information Systems Design Methodologies: a Comparative Review" Ed. OLLE T., North Holland 1982.

[DATE86] DATE C.J.: " A critique of the SQL Database Language" in "Relational Database selected writings", DATE C.J., ADDISON-WESLEY, 1986.

[GAME87] GAMERMAN S, VELEZ F.:"Using a set of database applications to compare Programming Languages" Tech. Rep. ALTAIR nx 12- 87, Oct. 1987.

[HAYE83] HAYES-ROTH F, WATERMAN D.A, LENAT B: "Building Expert Systems" Addison Wesley, 1983.

[LING87a] LINGAT J.Y, NOBECOURT P, ROLLAND C: "Behaviour Management in Database Applications" 13th Conf on VLDB, Brighton, Sept. 1987.

[LING87b] LINGAT J.Y, NOBECOURT P, ROLLAND C: "RUBIS : an Extended Relational DBMS Managing Events" Information and Software Technology Vol.29, Nx 9 & 10, Nov. & Dec. 1987.

[NOBE88] NOBECOURT P, ROLLAND C, LINGAT J.Y.: "Temporal Management in an Extended Relational System" submitted for publication.

[ORAC87] PRO*C User's Guide, Version 1.1, Oracorp 1987.

[ROLL82] ROLLAND C, RICHARD C: "The REMORA Methodology for Information Systems Design and Management" in [CRIS 1].

[ROLL87] ROLLAND C, BENCI G, FOUCAUT O: "Conception de Systemes d'Information: La Methode REMORA", Eyrolles 1987.

[SQL 86] "Information Processing Systems - Database Language SQL" Draft International Standard ISO/DIS 9075, 1986.

[STON84] STONEBRAKER M. et al.: "QUEL as a Data Type" ACM SIGMOD Conf., Boston, MA, June 1984.

[STON86] STONEBRAKER M, ROWE L: "The Design of POSTGRES" ACM SIGMOD Conf. Washington D.C, May 1986.

[SCHM80] SCHMIDT J.W, MALL M: "PASCAL/R Report" Technical Report nx66, Fachbereich Informatik, Univ. Hamburg, Jan 1980.

[WASS82] WASSERMAN A: "The User Software Engineering methodology : an overview" in [CRIS 1].