Der folgende Beitrag ist die Niederschrift eines Vortrags, auf der Prozeßrechnertagung '81 im März 81 in München.

Comparison of Languages (Coral, PASCAL, PEARL, Ada)

H. Sandmayr

Abstract. The facilities of some languages used for realtime applications are summarized and compared. It is not intended to give a recommendation for the use of one of these languages. Instead a set of different approaches is presented which provides an overview.

SOME REMARKS ON LANGUAGE DEVELOPMENT

Before discussing details of the languages considered in this paper some remarks on language development seem to be appropriate. The development of each of the languages was influenced by the state of the art at the time of their design.

In the development of programming languages three phases can be distinguished [We 76]:

- discovery and description of programming concepts and basic implementation techniques in the 1950's,
- elaboration and analysis of this concepts, development of models, abstractions, and theories concerning languages in the 1960's, and
- emphasis on the engineering approach in the software development technology (in the 1970's).

In the first phase languages were regarded as tools to facilitate the formulation of programs;this phase includes the development of FORTRAN, ALGOL 60, COBOL, and many other languages.

The languages development in the second phase, e.g. PL/I, SIMULA 67, and ALGOL 68 are elaborations or generalizations of earlier languages. PL/I, for example, combines features of FORTRAN, ALGOL 60, and COBOL and attempts to replace different languages by one. ALGOL 68 is a systematic generalization of the features of ALGOL 60.

These attempts to achieve greater power of expression led to excessively elaborated and very complex languages. In the third phase we encounter a return to the essentials, to simple languages which support structured programming, modularity, and verification efforts. Examples of such methodology-oriented languages are PASCAL, and MODULA.

The development of real-time languages is embedded in the above mentioned development. CORAL an PEARL are languages developed in the second phase mentioned above. CORAL (1964, 1966) is an attempt to combine features of ALGOL 60, FORTRAN and macroassembly languages into an efficient language suited for realtime applications on small machines. PEARL (1971) follows the ideas of ALGOL 68 and of PL/I, and adds further multiprogramming facilities. PASCAL (1971) is a product of the third phase whereas ADA (1979) is an attempt to unify, elaborate and generalize the features of languages of the third phase.

DESIGN GOALS

In this section a short summary of the design goals of the different languages is given, Some goals were never stated explicitly but were implied by the time of the design.

CORAL has been designed for the implementation of systems on small, dedicated computers to replace machine code in this type of systems. The specific requirements were:

- compilers must be small enough to run in the production systems or standby system, and
- the language must allow to make full use of individual machine hardware and any other special facilities provided for example by an operating system. At the same time, the implementation must be possible on a wide range of machines.

PASCAL was designed and implemented with the following principal aims [Wi 71]:

- To make a notation available in which the fundamental concepts and stuctures of programming are expressible in a systematic, precise and appropriate way.
- To make a notation available which takes into account the various new insights concerning systematic methods of program development.

- To demonstrate that a language with a rich set of flexible data and program structuring facilities can be implemented by an efficient and moderately sized compiler.

System programming aspects were only considered in so far as necessary for compiler developments.

PEARL has been designed as high level language for industrial process control applications. It should provide multiprogramming facilities tailored to the particular application area, and facilities for a suitable description of the interaction between processes and environment. The syntax of PL/I has been adopted for the algorithmic part of the language.

In contrast to CORAL, machine independence and protability are considered more important than efficiency

ADA has been designed with three overriding concerns [AD 79]:
- a recognition of the importance of program reliability and maintenance,
- a concern for programming as human activity, and efficiency.

The intended application range are "embedded computer systems", i.e. software systems which are embedded into an existing physical environment, comparable to process control applications. The language should be used by application prgrammers.

PROGRAM STRUCTURES AND COMPILATION UNITS

A CORAL program consistes of segments and communicators. Communicators allow communication between segments and allow access to items which exist outside the program.

```
program_name
'COMMON' name (specific. of data items, labels,
              switches, procedures, segments and
              overlays);
'LIBRARY' (specification and eventual renaming of
          library routines used in program);
'EXTERNAL' external_symbol_name (List data items);
'ABSOLUTE' (specification of absolute adresses of
           data items);

segment_name
'BEGIN' segment_declarations;
   statement_sequence
'END';

further_segments
'FINISH'
```

Fig. 1: Structure of a CORAL Program

Independent compilation of segments is possible. The start address of a program can be any segment or label in a segment defined in a COMMON. It must be specified explicitly by means of an 'ENTER' definition.

CORAL has adopted the ALGOL 60 block structure, the scope rules and visibility rules for identifiers, except for macro identifiers whose definitions are valid until they are deleted explicitly.

The structure of a PASCAL program is shown in the following figure.

```
PROGRAM name (file_parameters);
  LABEL label_definitions
  CONST constant_definitions
  TYPE  type_definitions
  VAR   variable_declarations
  Procedure_and_function_declarations
BEGIN statement_sequence
END.
```

Fig. 2: Structure of a PASCAL program

In PASCAL the program is the compilation unit; however many implementations allow independent compilation of procedures.

Blocks are bound to procedures, functions, and programs. There exist no anonymous blocks as in ALGOL 60.

A PEARL program consistes of a set of modules; modules as show in Fig. 3 are compilation units and cannot be nested. A module consists of a system part and/or a problem part. The system part defines the relation of the program to elements of the computer systems and of the technical process. The problem part contains algorithms solving the given problem.

```
MODULE (name);
  SYSTEM; description_of_configuration
  PROBLEM; specification_of_imported_objects
           declaration_of_objects
           declaration_of_tasks/procedures
MODEND;
```

Fig. 3: Structure of a PEARL Module

The scope of objects is a module or a block. The scope of objects declared on the module level can be extended to other modules by declaring such objects as global and specifying them in other modules as imported objects.

Modules cannot be nested in contrast to procedures and tasks.

An ADA program can be composed of program units:subprograms and modules. Modules are either tasks, task types, or packages. A package is a set of logically related types, objects, and operations. Units can be nested, i.e. a task can contain subtasks and packages, and a package can contain local tasks as well as packages.

```
 PACKAGE
{        }    module_name  IS
 TASK
  declaration of objects and operations
  visible to environment
PRIVATE
  declaration of structural details
  of exported objects
END module_name;

 PACKAGE
{        }   BODY module_name  IS
 TASK
  declaration of types, objects, and operations
BEGIN statement_sequence
EXCEPTION list_of_exception_handlers
END module_name;
```

Fig. 4: Structure of an ADA Module

In general, a unit consists of two parts, the specification and the body. The entities declared in the specification part are visible outside the unit and can be used by outer units.

Structural details of some declared types or objects may be irrelevant to their use outside a module. Declaring them in the private section prevents other units to make use of this information. Thus, the scope of an entity declared in the declaration part of a program unit is the range from the entity´s declaration to the end of the scope of the program unit containing the declaration.

The scope of an entity declared in the body of a unit or in a block is the respective program unit, or more precisly, the range between an entity´s declaration and the end of the unit containing the declaration.

There exists no explicit feature or restriction for the import of entities which are defined in an outer program unit. Every object whose name is visible at the point of the unit´s declaration is implicitly imported and can be used within the unit, unless the name is hidden by a local redefinition. However, in contrast to the usual scope rules redefinition of an identifier in an inner block does not necessarily hide its definition in the outer block. An identifier denoting more than one entity is said to be overloaded. When using such an overloaded identifier the context must allow to determine which definition is to be used.

Units of compilation are module declaration, module bodies, subprogram declarations, and subprogram bodies. PRAGMAs allow to control the compilation process, e.g. specification of configuration, or optimization criteria. By means of a context specification the set of units visible by the compilation unit can be specified.

Subprograms

Subprograms (procedures and functions) can be declared in all the languages. PEARL and ADA allow to specify whether a subprogram should be expanded inline at each call or whether the usual subroutine mechanism is to be used. In Ada inline subprograms can be used to included assembly code in a program.

Subprograms can have parameters in all the languages. In CORAL, PEARL, and PASCAL, variables can be passed either by value or by reference. In ADA, three parameter modes are provided:

IN or constant,
OUT or result, and
IN OUT or update parameters.

For IN parameters default values can be defined.

Different objects are accepted as parameters:

CORAL: values (represented by expressions)
variables (arrays and tables by reference only)
procedures

PEARL: every object except tasks and modules

PASCAL: values, variables, and subprograms

ADA: values and variables only,
(however variables can be of task types)

Actual parameters are associated to the formal ones by their positional order. In addition, ADA also permits an association by name, i.e. formal and actual parameters are explicitly associated in the actual parameter list.

TYPES AND STRUCTURES

The following tables show a summary of the types and structures provided by the compared languages.

Table 1: CORAL 66 and PASCAL

	CORAL 66	PASCAL
basic types	INTEGER FLOATING FIXED (+ scale)	INTEGER REAL BOOLEAN CHAR enumeration types subrange types pointer
structures	ARRAY (static) TABLE	ARRAY (static) RECORD (variants) SET FILE
remarks	structures cannot be nested one or two dimensional arrays only	structures can be nested
allocation of variables	in common or stack at address determined by compiler or specified in program overlays possible	in heap or stack at address determined by runtime system or compiler resp.
access to variables	by name or absolute address aliasing possible (parameter,overlay, anonymous reference)	by name or reference if dynamically allocated aliasing possible (parameter)

Table 2: PEARL and ADA

	PEARL	ADA
basic types	FIXED FLOAT BIT CHAR REF CLOCK, DURATION SEMA, BOLT	INTEGER (RANGE) FLOAT (DIGITS) fixed point (DELTA) BOOLEAN CHARACTER enumeration derived types subtypes ACCESS DURATION
structures	array (dynamic) STRUCT bit chain DATION	ARRAY (dynamic) RECORD (variants) STRING
remarks	structures can be nested	structures can be nested types can be parameterized
representation specification	no. of bits for numerical values	range, absolute and relative accuracy for numerical values repr. of enumeration types record types
allocation of variables	at addr determined by compiler, RESIDENT attribute indicates fast access	at addr determined by compiler or runtime system for dynamically allocated objects; explicit address and spec possible
access to variables	by name or reference aliasing possible (parameters and references)	by name or reference if dynamically allocated no aliasing (except for dynam. alloc. variables)

The type concept provided by CORAL is rather poor. There are only numeric types and two structures. Arrays are restricted to vectors and matrices of numerical values. Tables, the equivalent to a vector of records, require references to the internal representation of data for the definition of fields.

In contrast to the remaining languages new types cannot be named except by the general macro facilitiy provided in the language.

PEARL adheres to the type concept of PL/I and adds some simple types for multiprogramming purposes and time specifications. Particular features are provided to define the interface to computer and process peripherals (DATION).

PASCAL and ADA have a strong type concept; the type of any object is determinable during translation and therefore the set of applicable operations is known. New types can be defined and named. Type equivalence is related to name equivalence, a solution which is not totally realized in PASCAL.

PASCAL´s subrange types are elaborated to subtypes in ADA. The derived type in ADA even allows to distinguish types with formally identical set of values and operations (but eventually different representation).

The PASCAL set structure is not available in ADA. Files are provided in ADA in a predefined package.

References to variables exist in all languages in some form. CORAL and PEARL allow references to any variables with the inherent problem of references to objects in a no longer existing block. In PASCAL and ADA there exist only references to dynamically allocated and (explicitly) deallocated objects.

STATEMENTS

Overview

The following tables list the statements provided in the different languages. The most detailed version is shown for statements allowing several variants.

Table 3: List of CORAL 66 and PASCAL statements

CORAL 66	PASCAL
location := expression GOTO identifier procedure call ANSWER expression	variable := expression GOTO number procedure call
BEGIN declarations statements END	BEGIN statements END
IF condition THEN simple_statement ELSE statement	IF boolean_expression THEN statement ELSE statement
	CASE expression OF constant: statement; ... END
FOR variable := expr STEP expr UNTIL expr DO statement	FOR variable := expr {TO / DOWNTO} expr DO statement
FOR variable := expr WHILE condition DO statement	WHILE boolean_expr DO statement
	REPEAT statements UNTIL boolean_expr
CODE BEGIN assembler_statements END	WITH record_variable DO statement
i/o not defined	predefined i/o procedures

Neither CORAL nor PASCAL provide facilities for multiprogramming. However, tasks can be represented by programs and the procedure call mechanism can be used to access operating system functions, especially functions allowing inter-process (interprogram)communication.

Table 4: List of PEARL and ADA Statements

PEARL	ADA
variable := expression; GOTO identifier;	variable := expression; GOTO identifier; EXIT loop_identifier WHEN condition;
CALL identifier; RETURN(expression); INDUCE signal identifier	proc_call RETURN expression; RAISE exception;
BEGIN declarations statements END;	DECLARE declarations BEGIN statements EXCEPTION exc_handler END;
ON signal_id: statement;	
IF condition THEN statements ELSE statements FIN;	IF boolean_expression THEN statements ELSIF boolean_expression THEN statements ELSE statements END IF;
CASE expression ALT statements ... OUT statements FIN;	CASE expression OF WHEN choice => statements ... END CASE;
FOR variable FROM expr BY expr TO expr WHILE condition REPEAT declaration_list statements END;	{FOR variable IN range / WHILE boolean_expr} LOOP statements EXIT WHEN boolean_expr; statements END LOOP;
OPEN, CLOSE, PUT, GET, TAKE, SEND + formatting facilities, READ, WRITE	predefined packages defining i/o types and operations
statements for multiprogramming see next section	

The statements which control the sequential flow of instructions in the different languages provide almost identical possibilities and differ only respect to their syntax. This difference can however influence the style of a program; note for example the difference between the overloaded loop statement in PEARL and the set of simple loop statements in PASCAL, or the difference between the not very readable CASE statement in PEARL and its counterpart in ADA.

Input/Output-Facilities

CORAL follows ALGOL 60 and gives no definition of input-output facilities. This allows an implementor to use directly the mechanisms provided by an underlying operating system. This solution can be very efficient but does certainly not enhance portability of a program.

PASCAL bases its i/o on the file structure and a set of predefined procedures. The procedures for text i/o are treated by the compiler in a special way. They accept an optional file parameter, a varying number of parameters of different types, and a special field width seperator. The file structure with the basic procedures PUT and GET requires in general a simple runtime interface to the underlying system. Initialisation of this interface is assumed to be implicit. Experience shows that many PASCAL implementations provide further procedures allowing access to special file system facilities, e.g. random access. This is the main source of difficulties when moving a PASCAL program from one installation to another.

Low level- or process-i/o is not defined in the language. It can only be provided by language extensions or the use of operating system procedures.

PEARL provides the most comprehensive (and complex) set of i/o facilities. The basic elements are data stations (DATION). They are either system defined (e.g. terminals, disc, or a sensor) or user defined. I/O operations read or write data structures from or to such data stations. There are facilities for formatted i/o (PUT/GET + format specifications), for i/o in internal representation (READ, WRITE), and process i/o in form of bit sequences (TAKE, SEND).

A complex set of attributes allows specification of all kinds of data station characteristics but requires a sophisticated runtime system for the support of the different i/o operations.

A totally different approach is taken by ADA. No attempt is made to define special features covering the large range of input-output applications. The language facilities are designed in a way which allows the development of input-output packages without the definition of special features.

Three standard packages are predefined in the language:

INPUT_OUTPUT for general user level input-output operations,

TEXT_IO for text input-output, and

LOW_LEVEL_IO for operations dealing low level input-output.

This solution has the advantage that not every user and every translator must handle the additional complexity; however, a solution realized within the language can be realized in a much more flexible way than by using standard language features, e.g. lists of a varying number of output elements could be supported.

Exception Handling

In PEARL and ADA exceptions can be treated explicitly; however, different solutions are provided.

In PEARL, exceptions are considered to be infrequent events but not necessarily errors. Thus an exception can provoke execution of some actions and then control may return to the point where the exception interrupted the normal execution of a task. It is assumed that the exeption handler has performed some repair actions and normal execution can be resumed. (However, the exception handler can decide to branch to an other point of the program).

Exceptions are related to signals and occurrence of an exception activates an exception handler if present. Exception handlers are statements of the form

```
ON signal_id : statement
```

The scope of an exception handler is the task, procedure, repetition or block containing its declaration. Its scope includes all nested units which do not provide a handler for the particular exception.

Thus, these handlers behave like subroutines which can be anonymously activaded at any point of execution. This undetermined behaviour poses almost unsolvable problems for the verification of program units containing exception handlers.

In ADA , exceptions are restriced to events which can be considered as erros or at least termination conditions. Therefore exception handlers can be declared at the end of a subprogram body, module body, or block, e.g.

```
BEGIN  statements
EXCEPTION
  WHEN exception_id =   statements
  ...
  WHEN OTHERS =   statements
END;
```

Exceptions can be raised implicitly or explicitly (by means of the RAISE statement). When an exception is raised, normal program execution is suspended and the appropriate local handler is activated and replaces execution of the remainder of the current unit. If no local handler is provided execution of the current unit is terminated and the exception is reraised in the outer unit (for a subprogram the outer unit is the unit containing its call). An exception is propagated in this way until a handler is encountered or the body of a task is reached and the task is terminated.

MULTIPROGRAMMING FACILITIES

Multiprogramming facilities are only provided in PEARL and Ada. Both language allow the declaration of tasks; in PEARL they have a structure similar to that of a subprogram, in ADA that of a module. Table 5 lists the operations available to control execution and synchroniuation of tasks.

Table 5: Multiprogrammirg Facilities

PEARL	ADA
extended_time_specification ACTIVATE task; TERMINATE task; SUSPEND task; time_spec CONTINUE task; time_spec RESUME task; PREVENT task;	tasks are activated implicitly upon task creation ABORT task; RAISE task.FAILURE; DELAY expression;
operations on semaphores: REQUEST, RELEASE operations on bolt varibles: RESERVE, FREE, ENTER, LEAVE operations on interrupts: DISABLE, ENABLE, TRIGGER WHEN interrupt_identifier task_control_statement operations on signals: ON signal: statement; INDUCE signal;	rendezvous: ACCEPT entry_descr DO statements END; SELECT WHEN boolean_expr => select_alternative OR select_alternative ELSE statements END SELECT; entry_call_statement SELECT entry_call ELSE statement(s) END SELECT; SELECT entry_call statement(s) ELSE delay_statement statement(s) END SELECT;

In both languages tasks can be created, and deleted, activated, suspended, and aborted. Whereas ADA only provides a minimal set of basic operations, PEARL follows a more application oriented approach. Some of its operations can be combined with elaborated timing specifications, e.g.

AT 16:00:30 RESUME task;

WHEN interrupt _id AFTER 10 SEC EVERY 20 MIN
UNTIL 15 :20:00 ACTIVATE task_id;

This powerful mechanism requires substantial runtime support and it may be difficult to map it on an underlying operating system. It is even doubtful whether features are to be included in a language or whether they belong to the problem set and should be realized by simpler tools provided in the language.

Synchronization Concepts

Symchronization and mutal exclusion must be performed in PEARL with semaphores and bolt variables. Bolt variables are extended semaphores and are in one of the three states "free", "blocked", and "occupied". Table 6 shows the effect of the bolt operations.

Table 6: Bolt operations

RESERVE: {free} -> blocked}
FREE: {blocked}-> free}

ENTER: {free, occupied} -> {occupied}
LEAVE: {occupied} -> {free (iff #LEAVEs=#ENTERs), occupied}

Thus, bolt operations provide the mechanism to achieve exclusive access or simultaneous access to shared objects.

Semaphores and bolt variables are simple and easy to unterstand; however their use tends to be unstructured and prone to error: the respective operations must occur in pairs but no automatic checks for correct use are possible.

The rendezvous concept in ADA tries to circumvent these difficulties. A rendezvous is an (asymmetric) interaction between two tasks. One task issues a request to an "entry" in the second task. The second task performs the interaction when it is ready to accept the request. Entries are declared in a form similar to a subprogram declaration and requests have the form of a subprogram call.

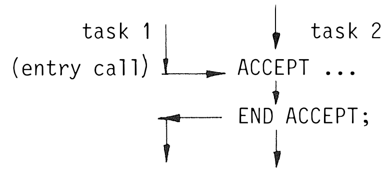

Fig. 5: Rendezvous

The accept statement specifies the actions to be performed during a rendezvous, i.e. when the corresponding entry is called (by task 2).The task arriving statement is reached (by task 2). The task arriving first has to wait for the other.

A select statement combines several accept and delay statements, thus making selective wait and timeout conditions possible. Two other forms of the select statement allow the caller of an entry to issue a conditional entry call, i.e. the entry call is only issued if the redezvous is immediately possible; the timed entry call allows the secification of a maximal delay for the acceptance of the entry call.

The rendezvous concept is an attempt to unify process communication and mutual conclusion. It allows synchronous process communication via the parameter list of an entry. This synchronous communication technique allows any other synchronization or communication concept to be modelled; however, in most cases auxiliary processes are required.

FINAL REMARKS

The following summarizing remarks on each of the languages do not consider the availability of compilers, although availability and quality of a compilers can be the determining factor when a language is to be chosen.

CORAL certainly fulfills the design criteria stated above. It is a simple language, easy to implement and allows efficient access to hardware and operating system facilities. However, the definition leaves many details to a particular implementaion, e.g. I/O. In addition assembly code insertions and usage of anonymous references, i.e. absolute addresses, reduce the portability (and probably also the maintainability) of programs.

PASCAL provides a set of simple control structures and a large variety of data types. The concept of strong typing (although not totally waterproof) allows the detection of many errors at compile time. However, PASCAL does not provide modules, multiprogramming facilities, and support for seperate compilation. There are many languages extending PASCAL in this respect which maintain its original simplicity. Two examples are MODULA [Wi 78] and PORTAL [Na 79].

PEARL provides a large set of facilities for real-time programming. However, the language is very complex and baroque. Furthermore, its design is not very consistent, e.g. interrupts exist besides the very elaborated timing specification possibilities, and a primitive case statement together with powerful input-output statements. The input-output system and the multi-tasking model require an elaborated run time support. In many cases it is very difficult to map PEARL features on an underlyng operating system in an efficient manner.

ADA also provides a large set of facilities. In comparison with PEARL, the elements are kept on a lower level. For example, timing specifications for process scheduling are not provided but can be realized with the given features. ADA has a consistent typing concept which is stronger than that of PASCAL. Since every single feature is elaborated in a detailed way (e.g. type, subtype, and derived type are distinguished)the whole language becomes rather complex. Since many restrictions ore rules which are only understandable when the underlying concepts are known, the language is difficult to instruct.

REFERENCES

[AD79] --: Preliminary ADA Reference Manual, acm, Sigplan Notices, 14,6 1979

--: Rationale for the Design of the ADA Programming Language, acm, Sigplan Notices, 14,6 1979

[AD80] --: Reference Manual for the Ada Programming Language United States Department of Defence, July 1980

[ES79] --: ESL, Report on Main Task I and II. Siemens, Munich and CII Honeywell Bull, Paris Oct. 1979

[Na79] Naegeli H.H.: Programming in PORTAL: Publication of Landis & Gyr, Zug, Switzerland

[PE77] --: Basic PEARL Language Description. Gesellschaft fuer Kernforschung mbH, Karlsruhe PDV-Report KfK-PDV 120, 1977

--: Full PEARL Language Description. Gesellschaft fuer Kernforschung mbH, Karlsruhe PDV-Report KfK-PDV 130

[We76] Wegner P.: Programming Languages - The First 25 Years. IEEE Transactions on Computers, Vol. C-25, 12, 1976

[Wi71] Wirth N.: The Design of a PASCAL Compiler. Software-Practice and Experience, Vol. 1,1971

[Wi77] Wirth N.: Modula: A Language for Modular Multiprogramming. Software, Vol. 7, 3-35, 1977

[Wi78] Wirth N.: MODULA-2. Report of the Institut fuer Informatik, ETH Zurich, No. 27, Dec. 1978

[Wo70] Woodward P.M.: Official Definition of CORAL 66. Her Majesty´s Stationery Office, 1970

[WW78] Werum, W. and Windauer H.: PEARL, Process and Experiment Automation Realtime Language. Vieweg Braunschweig 1978