MODULA-2 IN EMBEDDED SYSTEMS

Christian Vetterli
Hiware AG
Basel, Switzerland

Claude Vonlanthen
Hiware AG
Basel, Switzerland

Christian Vetterli studied under Niklaus Wirth, writing his thesis on the object-oriented expandable document editor OPUS (Object-Oriented Publishing System) written in Modula 2. Dr. Vetterli is responsible for the Modula-2 tool MacMETH (loader and libraries for Apple Macintosh), libraries for Modula-2 development systems on IBM-RT (RISC), and the code generator for the Modula-2 compiler. He is responsible for developing embedded systems software for Hiware.

Claude Vonlanthen is responsible for applications software at Hiware. A graduate engineer from Eigenossische Technische Ilochschule, Zurich, Switzerland, Claude was responsible for developing a Modula-2 embedded system at a large machine factory before joining Hiware.

Modula-2 in Embedded Systems

Dr. Christian Vetterli
Claude Vonlanthen

PART 1

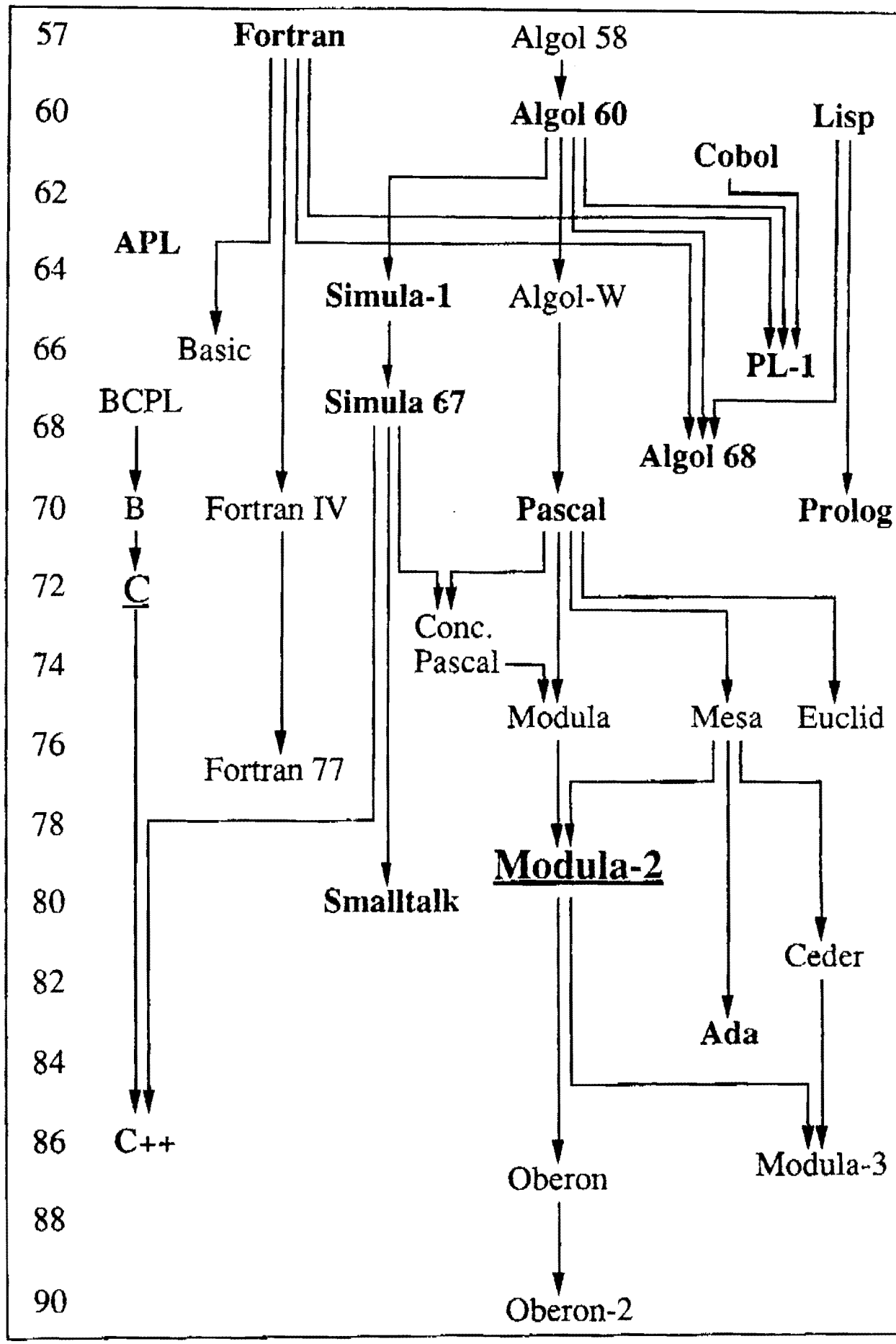
In the years 1977-1981 Prof. N. Wirth developed Modula-2 [1] as a further member of the family of the Algol, Pascal and Modula programming languages (Fig. 1). "Whereas Pascal had been designed as a general purpose language and after implementation in 1970 has gained wide usage, Modula had emerged from experiments in multiprogramming and concentrated therefore on relevant aspects of that field of application.

In 1977, a research project with the goal of designing a computer system (hardware and software) in an integrated approach, was launched at the Institut für Informatik of the ETH Zürich. This system (later to be called Lilith [2]) was to be programmed in a single high-level language which therefore had to satisfy requirements of a high-level system design as well as those of low-level programming of parts which interact closely with the given hardware. Modula-2 emerged from careful design deliberations as a language which includes all aspects of Pascal and extends them with the important module concept and with those of multiprogramming."

Some of the main features of Modula-2 are:

- separate compilation of program modules,
- strong type checking,
- comprehensive runtime tests,
- structured data types,
- dynamic data types,
- nested program structures,
- procedure types,
- the support of parallel processes (co-routines) and
- system-dependent language properties (low-level facilities).

First of all, we turn our attention to the most important concept of Modula-2, the one which has also given the language its name - the modular concept. Then we discuss briefly the other properties of Modula-2, especially with regard to applications in Embedded Systems. Finally we present a approach of an operating system for distributed real-time applications and a related development method.



The modular concept

The concept of the module has long been in use in the engineering field. We find it for example in the implementation of electronic devices where we find slide-in modules which contain complete functional units. The advantages are obvious, the modules can be developed, produced, tested and even repaired as self-contained units. The use of modules also enables substantially reduced costs because it is possible to refer to proved solutions for certain functions.

The same motivations which promoted the introduction of modules in engineering also apply in the field of software development. Because, today, software is no longer created by an elite group of programming wizards, but within an industrial framework, it is absolutely essential to produce the software in a properly structured and modularized form.

A certain amount of modularization is already to be found in most commonly used programming languages in the procedure. Procedures already enable a problem to be subdivided into smaller part problems. The parameter mechanism permits an exact specification of the interface for solving a part problem. The implementation of the solution remains "hidden" in the individual procedure. Because, however, the data defined within the procedure exists only during the execution of the procedure, commonly used so-called *global data* must remain visible for all program parts. It cannot be guaranteed therefore that a special implementation (definition of the data structure) of the global data will not be used within a procedure. In large projects this leads to incomprehensible complexity of the system because changes to the implementation of the global data have effects in many (unexpected) places.

An effective solution to the problem is created here in Modula-2 by the concept of the *module* and the associated concept of *data encapsulation*. A module makes it possible to hide both data and procedures and their implementation. In this way it is possible to combine individual part problems of a larger system. Each module has an external interface which precisely defines all operations of the module. Once this interface has been defined, the module can be used independently of the actual implementation because the details are not visible to the user. We speak in this context of *information hiding*.

We can differentiate between various classes of modules:

(1) *Function modules* make available to the user a series of functions, but themselves do not contain their own (internal) data objects.

(2) A so-called *data module* (abstract data structure) contains its own data objects and makes procedures available for manipulating these objects. The objects are of course only accessible from outside via these procedures. A data module contains only a copy of the data object.

(3) An *abstract data type* is defined by the definition of a type name and the operations required to process the type. The operations also embrace here the generation of new copies of the data type. All the operations are parametrized with the respective copy of the type.

Modules in Modula-2

The language Modula-2 supports the above-described model thanks to a *strict separation* of interface and implementation of a module. A distinction is made between the definition of the interface (definition modules) and the implementation (implementation module). A module can export both procedures as well as types and variables, i.e. make them available to other modules. Each module, in turn, can import objects from other modules. The definition of the interface contains all the procedures and data types which the module exports. Procedures are described by the procedure head, type and variables are described by a declaration. When the definition module is translated, a so-called *symbol file* is created which contains all the information about the interface. The symbol file is used for the compilation of modules which use the interface. The implementation of the procedures listed in the interface takes place in the implementation module. It can be developed independently of the interface, but must comply with the specifications laid down in the interface. This means in particular that it is very easy to prepare different implementations for the same interface. Thus, machine-dependent parts of a larger program can also be readily combined in one place. This permits simple extraction of the machine concerned. The importing/exporting of a program consists then in adapting the implementation of the machine-dependent module to the new machine.

The scheme of *separate* compilation implemented in Modula-2 is to be distinguished from *independent* compilation which is implemented for most programming languages. In the case of independent compilation the compiler does not have the facility to check the consistent use of an interface. Errors only become noticeable therefore when a program is linked or, even later, when it is run. With the method used in Modula-2, errors relating to the use of an interface are discovered as early as the compilation because the compiler has all the important information available in the form of symbol files. The big advantages of type testing are not therefore lost in this way.

Some short examples in Modula-2 are given below. An example is given for each of the above-listed classes of modules.

Function module

The following module is listed in the language description for Modula-2 [1] as the standard module for mathematical functions. It combines certain operations, namely the most important mathematical functions, but does not make a special data type available nor does it have its own data manipulated by functions.

```
DEFINITION MODULE MathLib0;

   PROCEDURE sqrt(x: REAL): REAL;
   PROCEDURE exp(x: REAL): REAL;
   PROCEDURE ln(x: REAL); REAL;
   . . . .
END Mathlib0.
```

Data module

In the following example, a stack which can store INTEGERS is to be implemented as an abstract data structure.

```
DEFINITION MODULE Stack;

   PROCEDURE Push(x: INTEGER); (* pushes the value 'x' onto the
stack *)
   PROCEDURE Pop(): INTEGER;  (* pops the top element of the stack *)

   PROCEDURE Empty(): BOOLEAN;     (* return the state of the stack *)
   PROCEDURE Full(): BOOLEAN;

END Stack.
```

The definition of the interface shows that precisely one stack is defined. The stack is to some extent an implicit parameter of all operations. Details about the representation of the stack are not visible. An implementation of the module looks like this:

```
IMPLEMENTATION MODULE Stack;

   CONST MaxStack = 20; (* max. 20 values on the Stack *)

   VAR   stack: ARRAY[1..MaxStack] OF INTEGER;
         stackPtr: INTEGER;
   . . . .

   PROCEDURE Push(x: INTEGER);
   BEGIN
      IF stackPtr = MaxStack THEN (*stack overflow *)
      ELSE INC(stackPtr); stack[stackPtr] := x
      END
   END Push;
   . . . .

BEGIN (* initialization *)
   stackPtr := 0
END Stack.
```

Abstract data type

If you want to use several copies of the same abstract data structure, this can no longer be achieved with a data module. The *abstract data type* is available for this. When applied to the preceding example of the stack, the definition module then looks as follows:

```
DEFINITION MODULE Stack;

   TYPE Stack;     (* hidden type *)

   PROCEDURE NewStack(VAR s: Stack): BOOLEAN;
      (* assigns a new stack to 's' returning TRUE upon success *)
   PROCEDURE DisposeStack(s: Stack);
      (* returns the stack 's' to be reused *)

   PROCEDURE Push(s: Stack; x: INTEGER);
   (* pushes the value 'x' onto the stack 's' *)
   PROCEDURE Pop(s: Stack): INTEGER;
```

```
   (* pops the top element of the stack 's' *)

   PROCEDURE Empty(s: Stack): BOOLEAN;
   PROCEDURE Full(s: Stack): BOOLEAN;
   (* return the state of the stack 's' *)

END Stack.
```

The module is first of all extended by two further operations which can generate or destroy stack objects. In addition, the 'Stack' type is exported, The above form of export is also known as *opaque* because the details of the data type are not visible. Because the operations have now to be universal, the stack to be processed must be included as an explicit parameter.

```
IMPLEMENTATION MODULE Stack;

   CONST MaxStack = 20;

   TYPE  Stack       =  POINTER TO StackDesc;
         StackDesc =    RECORD
                        stackPtr: INTEGER;
                        stack: ARRAY[1..MaxStack] OF INTEGER;
                     END;

   PROCEDURE NewStack(VAR s: Stack; size: CARDINAL): BOOLEAN;
   BEGIN
      ALLOCATE(s, SIZE(StackDesc));
      IF s = NIL THEN
         RETURN FALSE
      ELSE
         s^.stackPtr := 0; RETURN TRUE
      END
   END NewStack;

   PROCEDURE DisposeStack(s: Stack);
   BEGIN
      DEALLOCATE(s, SIZE(StackDesc));
   END DisposeStack;

   PROCEDURE Push(s: Stack; x: INTEGER);
   BEGIN
      IF s^.stackPtr = MaxStack THEN (*stack overflow *)
      ELSE INC(s^.stackPtr); s^.stack[s^.stackPtr] := x
      END
   END Push;

   . . . .
END Stack.
```

In the present implementation the stacks are represented in each case by an array. Because of the fixed limits of an array this, of course, restricts the length of the stack. However, it would be very readily possible by the use of a module to produce another implementation for the same interface which, for example, uses concatenated lists for the representation.

The language Modula-2

The syntax is largely the same as that of Pascal. There is also a strong similarity in the elemental data types, the static data structures and the types of statement. In this section we shall present only the most important properties and concepts of Modula–2; more detailed information will be found in the textbooks listed at the end. For comparison, we juxtapose the Modula-2 constructs with the corresponding C-definitions.

Standard Data Types

Standard data types are pre-declared types. Their range of values depends among other things on the basic machine and/or compiler. The types INTEGER and CARDINAL are signed and unsigned numbers normally the size of a machine word. BOOLEAN variables can have the values TRUE (=1) or FALSE (=0). The type BITSET enables quantity operations in the quantity {0..N-1} where N is the width of a machine word.

M2-Declaration	M2-Usage	C-Declaration	C-Usage
`i, j: INTEGER;`	`j := i + 5;`	`int i, j;`	`j = i + 5;`
`li: LONGINT;`	`li := 1000000;`	`long li;`	`li = 1000000;`
`c, d: CARDINAL;`	`d := c DIV 2;`	`unsigned int c, d;`	`d = c / 2;`
`r, s: REAL;`	`s := r / 1.5E-2;`	`float r, s;`	`s = r / 1.5E-2;`
`lr: LONGREAL;`	`lr := 1.5D+30;`	`double lr;`	`lr = 1.5E+30;`
`b, l: BOOLEAN;`	`b := NOT l;`	`int b, l;`	`b = ~l;`
`ch: CHAR;`	`ch = "y";`	`unsigned char ch;`	`ch = 'y';`
`bts: BITSET;`	`bts := {0, 5, 8..13};`		`unsigned int bts;`
	`bts = 1<<0 | 1<<5 | ..;`		

Apart from the standard data types, Modula-2 also offers the unstructured data types enumeration, subrange and set.

	Declaration	Usage
Enumeration:	`color: (red, green, blue);`	`color := red;`
Subrange:	`subrange: [2..32];`	`subrange := 16;`
Set:	`colors: SET OF (red, green, blue);`	`colors := colors + red;`

Operators

In Modula-2 we see a strong type linkage. The operands of an operator must usually be of the same type, and the result, in turn, has a exact defined type. If the programmer has to by-pass this rule, e.g. when adding a CARDINAL or INTEGER number, type conversion functions are available.

Type of Operands	M2-Operators	C-Operators
INTEGER, CARDINAL, LONGINT	+ - * DIV MOD	+ - * / %
REAL, LONGREAL +	- * / +	- * / %
BOOLEAN	OR AND NOT	|| && !
BITSET	+ - * /	| & ^
all types (in boolean expressions)	= # < > <= >=	== != < > <= >=

Modula-2:
```
TYPE NotifyProc = PROCEDURE (Window, INTEGER);

PROCEDURE OpenWindow(....; redraw: NotifyProc; ...): Window;
BEGIN
   ...
   redraw(win, parameter);
   ...
```

C:
```
typedef (*NoifyProc)(Window, int);

Window OpenWindow(..., NotifyProc redraw, ...)
{
   ...
```

The parameter 'redraw' is a pointer to a call-back function. By calling up this function the window handler can prompt the application to redraw the window contents.

In Embedded Systems a vector table has often to be maintained. In Modula-2 the following simple declaration defines such a table in the form of an array the basic type of which is the Modula-2 standard type PROC:

```
VAR vectorTable[0]: ARRAY [0..63] OF PROC;

PROCEDURE SetVector(nr: INTEGER; interruptHandler: PROC);
BEGIN
   vectorTable[nr] := interruptHandler;
END SetVector;
```

PROC corresponds to a parameterless procedure:

```
TYPE PROC = PROCEDURE();
```

Hardware-oriented programming

For machine-oriented programming the strong type checking of Modula-2 is often a handicap. Modula-2 therefore supports the hardware-oriented (low-level) programming with a few simple constructs. The data types and procedures defined for this are exported from the (pseudo-) module 'SYSTEM'.

```
DEFINITION MODULE SYSTEM;

   TYPE  BYTE;
         WORD;
         ADDRESS = POINTER TO BYTE;

   PROCEDURE ADR(x: AnyObject): ADDRESS;

   PROCEDURE VAL(CastType, x: AnyType): CastType;

   . . . .

END SYSTEM.
```

The data types BYTE and WORD are uninterpreted types with a width of a byte or a machine word. Variables of these types can be assigned values of any other types of the same size (without any checks whatsoever). They usually appear as parameters of procedures which are to be universally applicable, e.g.:

```
FileSystem.WriteWord(f:File; w: WORD);
```

As an open-array parameter the type BYTE or WORD has a special meaning because values of any size can be assigned to such parameters.

```
PROCEDURE WriteBytes(f: File; data: ARRAY OF BYTE);
```

Within the procedure the data is regarded as ARRAY [0.HIGH(data)] OF BYTE.

Variables of the ADDRESS type can be assigned addresses of objects (variables, procedures and constants). They are allocation-compatible with pointers. Arithmetic functions are restricted mostly to addition and subtraction. The function ADR() supplies the address of the object which is listed as the parameter.

The function VAL acts as a Type-Cast and converts the type of 'x' to the new type (first parameter). This makes it possible to by-pass the strong type linkage.

Often the 'SYSTEM' module contains further functions which, however, are very machine-dependent and are not therefore available in all implementations of Modula-2.

With the construct [aa] it is possible to allocate an absolute address to global variables during the declaration (see also the 'vectorTable' example). This facility can be used advantageously for efficient memory-mapped I/O.

```
TYPE
   CtrlRegType = SET OF (RIE, TC1, TC2, WS1, WS2, WS3, CD1, CD2);
   StatRegType = SET OF (IRQ, PE, OVRN, FE, CTS, DCD, TDRE, RDRF);
VAR
   UARTData [0FF81H]: CHAR;
   UARTControl [0FF85H]: CtrlRegType;
   UARTStatus [0FF85H]: StatRegType;
   . . . .
   UARTControl := CtrlReg{RIE, TC2, WS2 .. CD1};
   . . . .
   WHILE ~RDRF IN UARTStatus DO (* Wait *) END;
   receivedCh := UARTData;
```

Co-routines

The programming language Modula-2 makes it possible to define co-routines for the formulation of parallel processes. The operations required for this are also made available by the (pseudo-) module SYSTEM.

```
DEFINITION MODULE SYSTEM;
   . . . .

PROCEDURE NEWPROCESS( p: PROC;
                      workspace: ADDRESS; wspSize: CARDINAL;
                      VAR coroutine: ADDRESS);

PROCEDURE TRANSFER(VAR from, to: ADDRESS);

   . . . .
END SYSTEM.
```

C: `void EvalSize(char *ch, char *buffer);`

Modula-2: `PROCEDURE EvalSize(VAR ch: CHAR; ARRAY[0..N] OF CHAR);`

If an array is concerned, it is also not defined how big it is to be. The C-compiler is not able to generate range-checks nor has the debugger the facility to show the programmer the data in the correct representation.

Books on Modula-2

[1]	N. Wirth	Programming in Modula-2, 3rd editon	Springer, 1985
[2]	N. Wirth	The Personal Computer Lilith	ETH Zurich, Institute of Informatics, Internal Report No. 40, 1981
[3]	R.S. Wiener and R. Sincovec	Software Engineering with Modula-2 and Ada	Wiley, 1985
[4]	G.A. Ford and R.S. Wiener	Modula-2: A software development approach	Wiley, 1985
[5]	N. Wirth	Algorithms and Data Structures	Prentice-Hall, 1985
[6]	B.K. Walker	Modula-2: Programming with data structures	Wadsworth Publ. Co., 1985
[7]	Ed.J. Joyce	Modula-2: A seafarer's manual and shipyard guide	Addison-Wesley, 1985
[8]	G. Pomberger	Software Engineering and Modula-2	Prentice-Hall International, 1986
[9]	E. Knepley and R. Platt	Modula-2 Programming	Reston Publ. Co., 1986
[10]	A. Sale	Modula-2: Discipline and Design	Addison-Wesley, 1986
[11]	P. Messer and I. Marshall	Modula-2: Constructive Program Development	Blackwell Sci. Publ., 1986
[12]	T.A. Ward	Advanced Programming Techniques in Moudla-2	Scott, Foresman and Co., 1987
[13]	J.B. Moore, K.N. McKay	Modula-2 Text and References	Prentice-Hall, 1987
[14]	H. Schildt	Modula-2 made easy	McGraw-Hill, 1987
[15]	H. Schildt	Advanced Modula-2	McGraw-Hill, 1987
[16]	P.D. Terry	An introduction to programming with Modula-2	Addison-Wesley, 1987

PART 2

Foreword

The high-level language Modula 2 is suitable for generating the software for embedded systems. A complete development, testing and maintenance environment has been developed for this language. This tool can be used in different host computers and generates code for different target processors. In order to be able to make the development of real-time software even more efficient, an inquiry has been made about what is needed in this environment. An attempt has been made to formulate a comprehensive approach which will not only meet the demands of a real-time operating system but also the requirements for an appropriate design methodology.

Introduction

Using the high-level languages the engineer develops on a linguistically high abstraction level. In order that the debugging can also take place at this level, the code from editing to debugging must adopt certain forms. The development takes place in the high-level language and is translated by a compiler into a form which the processor understands. In order to be able to monitor the operation of the processor for the runtime at the level of the high-level language, the development tool must be able to refer back from the code to the level of the high-level language. The forms which the information assumes are found at different abstraction stages. According to Dr. Vetterli this can be represented as follows:

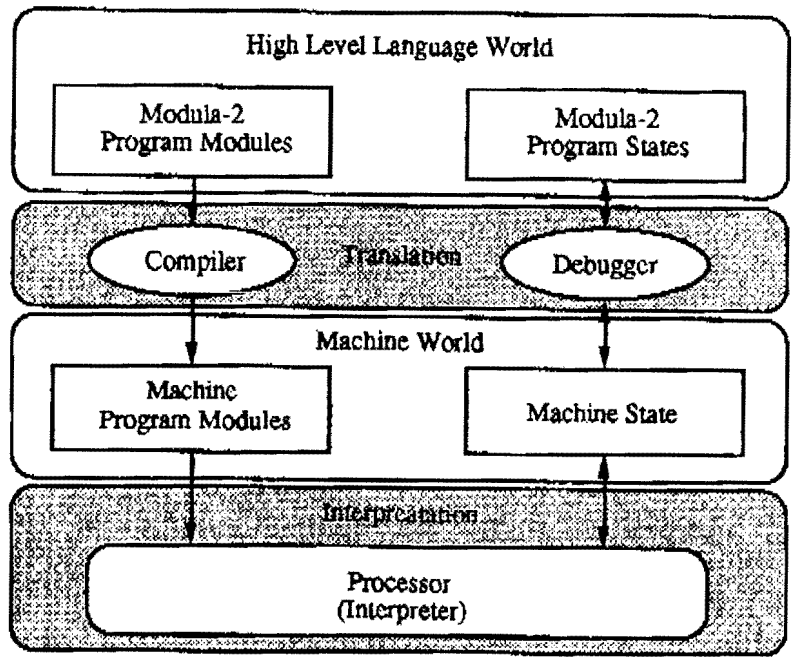


Fig.1

Formulation of the Solution

The formulation takes into account the requirements described above.

The basic thinking behind it is the fact that processes in an application can be understood and formulated as a set of services. Services are supplied and requested. The interplay between the offering and requesting of services forms the essence of the application and 'makes it work'.

What is a Service?

Services contain activities and actions which may be requested. The requests, in turn, arise out of other services.

Examples

In a chemical plant, such a service might be the "closing of emergency valve 23'. This is an example of a concrete service which might arise out of a "pressure monitoring" service when excess pressure is detected. Another example of a service might be the "manufacture and filling of a chemical agent".

The Idea behind SOOM

Top Down...

Services occur in different forms. The above examples represent services which are at different abstraction stages. The services of a lower level are contained in those of a higher level. This could well be imagined in the above example of the chemical process. This suggests a Top Down procedure in the design. Starting from an original, abstract service, concrete services are found in a Top Down procedure. By concrete we mean that these services would be able to implemented and requested by means of the language and the operating system. In addition, the service would need to be rational from the point of view of the application.

... Bottom Up

Where is a number of elementary services present, then these should be able to be combined on the basis of certain features. For example, the services "open emergency valve" and "close emergency valve" should be able to be brought together in an "operating emergency valve" group. Groups of services can be joined together to form families of services. At this stage, libraries can be created in this way.

Capacity for Distribution

These service structures are to be embedded in objects (teams) which make the necessary data and processes available for their implementation. A team is to be indivisible, i.e. it resides at one computer node. However it must be able to be used at any identical node without changes having to be made to its code. On the basis of the service names and the teams it must be possible to achieve a capacity for distribution by means of the operating system.

Encapsulation

The core of a service, its data structures and actions must not be known to the requesting object. They are therefore also to be encapsulated. The request can be made available to the user in the form of a procedure.

Realization of the Solution

A real-time operating system has been developed on the basis of the requirements discussed. A method has been devised for the design of applications which are to be implemented with the system. A tool to accompany this method is to be produced in the near future. In what follows, we give a brief description of the operating system, the structures which the system makes available and the method which helps the developer to find the structures.

The Operating System (the Kernel)

This is a preemptive real-time operating system with process priorities. The preemptive scheduling mechanism can be initiated at any priority stage by a timeslice method. In the implementation, care has been taken to ensure a high efficiency while also making easy-to-use interfaces available to the user. For instance, the operating system calls are available in the form of Modula 2 procedures whereas the dispatching is implemented in the assembler of the computer used.

The communication between closely coupled processes, i.e. processes on the same computer node, can be effected by their process identities. For the execution of distributed applications, however, a loose coupling is advantageous. The mechanism implemented is based then on the structures of the service, the service group and the service family as described below.
The operating system provides the developer with a set of functions in the form of Modula 2 functions and procedures. Some of these will be illustrated later.

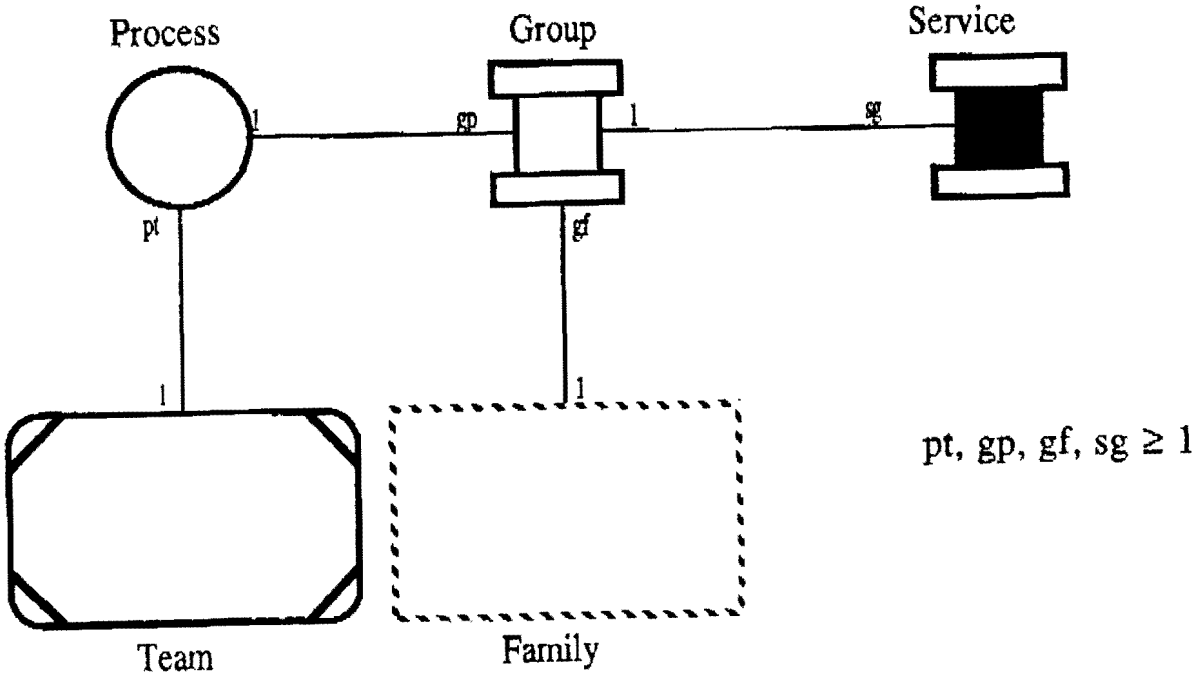


Fig.4

A possible line-up is shown in the following figure:

Example:

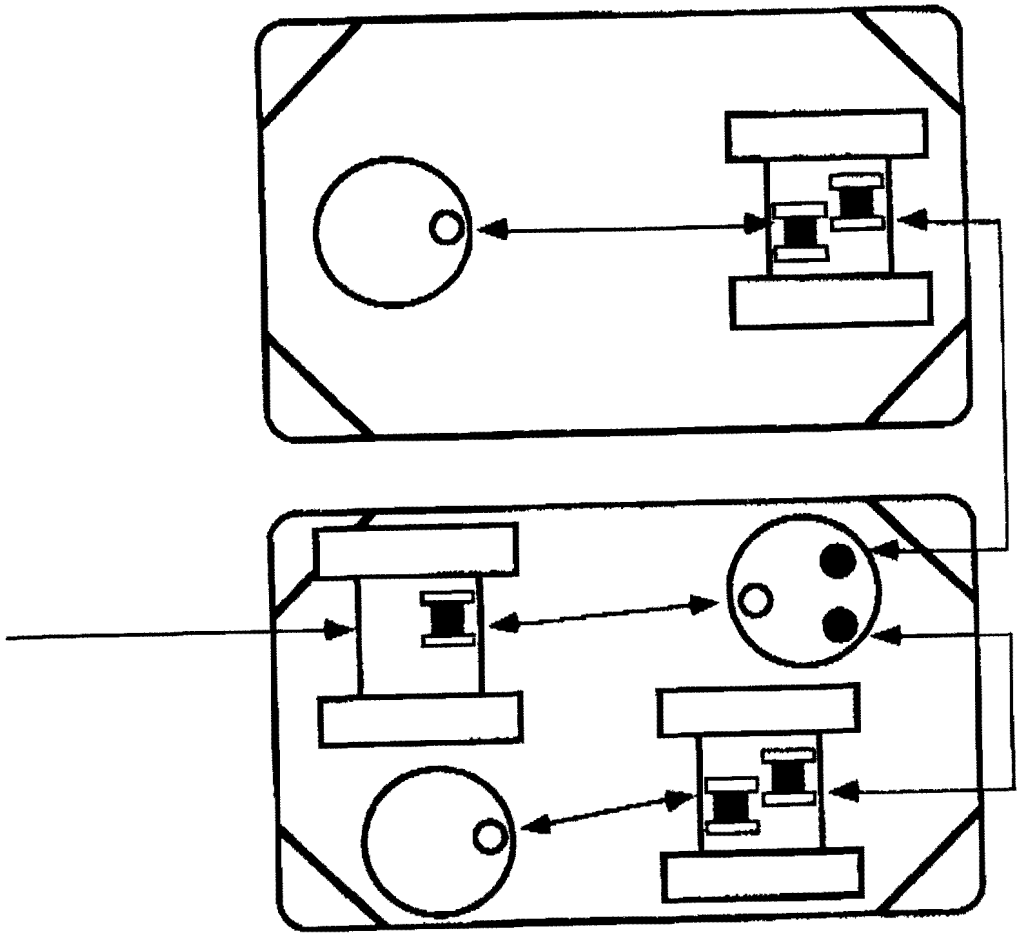


Fig.5

The Use of Modula 2

In the following we shall show the conversion of the structures discussed into Modula 2. Certain calls of the operating system are also illustrated.

Modular Structure of a Team

The team is the vessel of all the structures and codes appertaining to the service. Because the principle of information hiding is also to be applied at the services stage, as many as possible of the services offered are to be encapsulated. It is recommended therefore that a subdivision be made into different Modula 2 modules. The following module structuring has proved successful for a team:

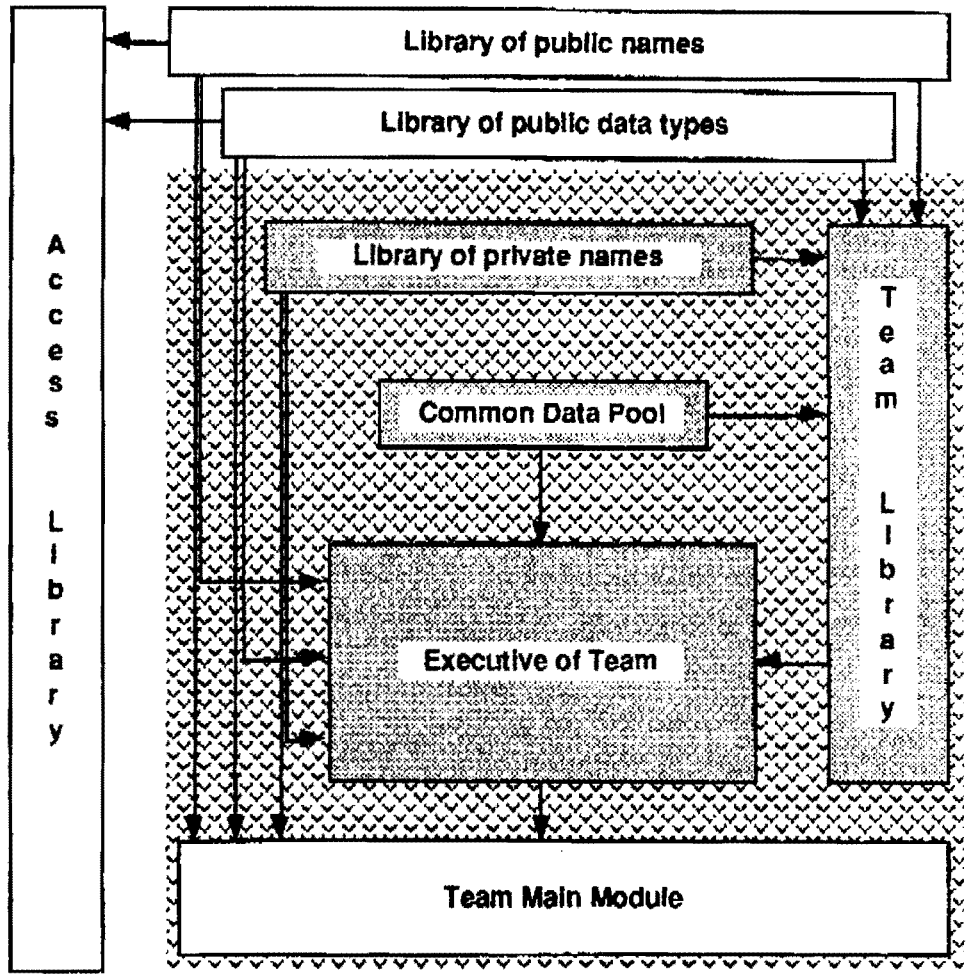


Fig.6

In the "Public Names" module all the names of the services, service groups and service families made available by the team are declared at Modula 2 level. For this purpose it is advantageous to use enumeration types. The following extract from a definition module will illustrate this:

```
        GETDATAFRAME (region, dataLength,
                      accessRight, requestType, valuePt);
        valuePt^ := Sensor;
        REPLYREQUEST(PressGri, reName, region, valuePt, sysRep);
                                .
                                .
                                .
     ELSE
                                .
                                .
                                .
     END;
   END; (* loop *)
END PressureManagerPr;

END PressEX.
```

Fig.10

Request for a service

The code of the process also reveals how a service is requested from another team. This request might also be packaged in an access procedure and made available by the provider team in an access library.

The mechanism of a service request

Behind every service request there is generally a communication between processes which has to be made possible by the operating system. As illustrated above, the processes do not have to "know" one another; the requester and the service provider remain anonymous. They are loosely coupled. Two forms of communication are available. One is asynchronous without data transmission. It can initiate a service by an event message. There is a control flow between the processes via the service names. The second form of communication is synchronous with data transmission. Here there is, in addition, a data flow between the two processes. The two forms are also possible between processes of the same team via process addressing. The following figure illustrates all the possible forms of communication/synchronization:

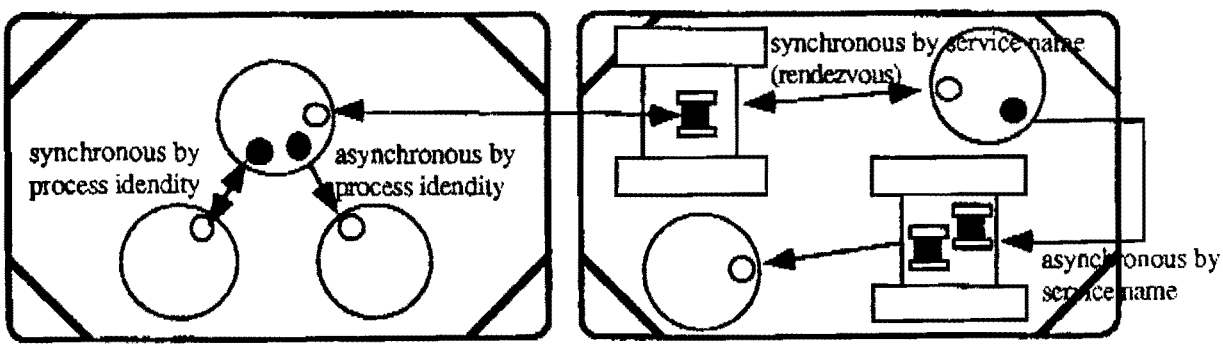


Fig.11

The Design

As already mentioned, the fundamental idea behind the design method is as follows: "Sequences in an application can be understood to be a set of services provided'. These services do not simply float in space, but are allocated to objects which are responsible for their implementation. A concrete service is provided, for example, by a team (= standardized SW object). Inside this team, a responsible process accepts the requests for the service and undertakes the necessary actions to enable it to provide this service. Services are supplied by objects for the benefit of the application and are made available to one another. An important point is also the fact that services can be considered at different abstraction stages. This permits a Top Down procedure when designing a project. The aim is to master the great complexity of a service by subdividing it into a set of services of lesser complexity. An attempt should be made here to carry out the subdivision in such a way that the services obtained are as far as possible universal and the objects to which they appertain are reusable.

The method can be broken down into three parts:

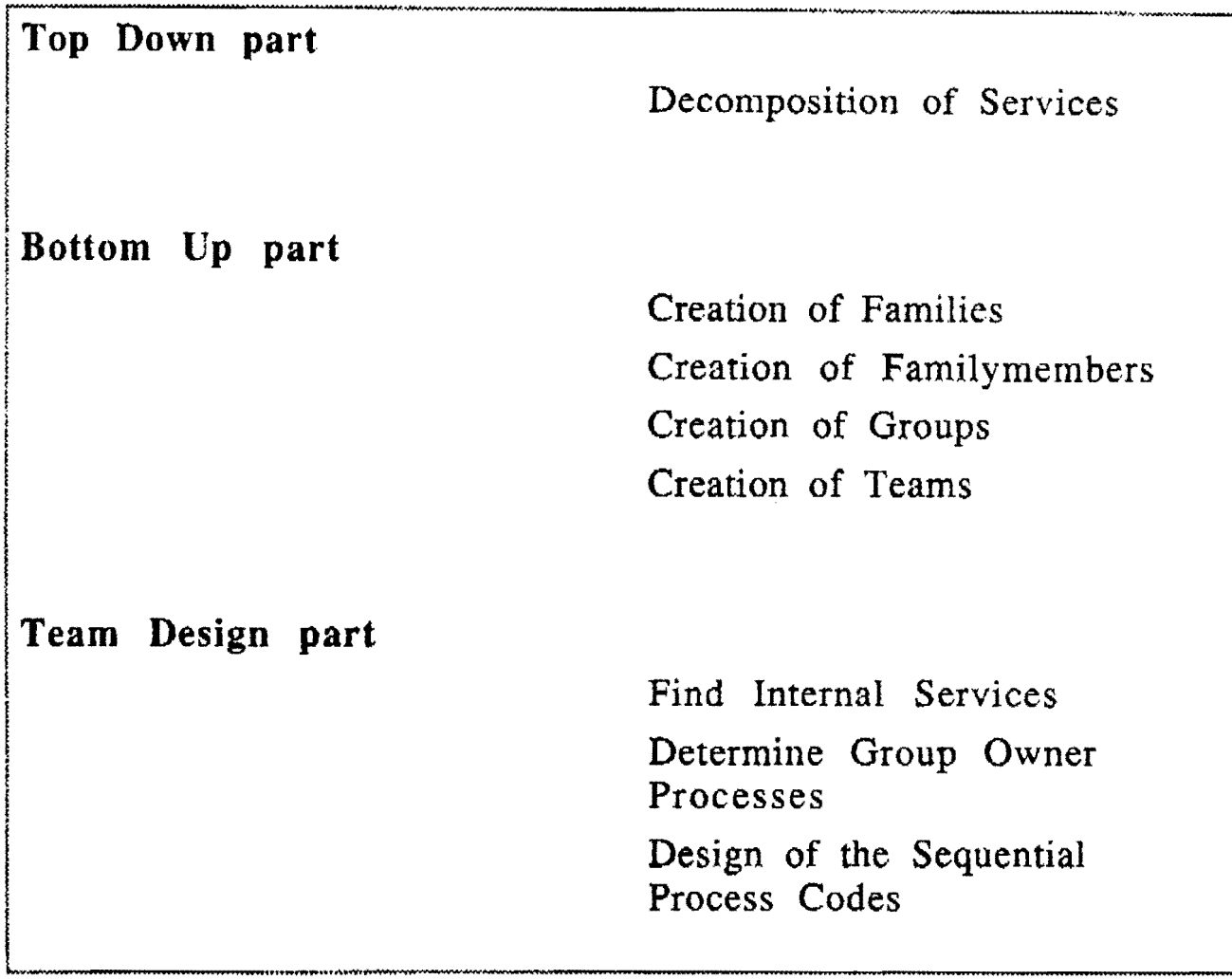


Fig.12

Capacity for Distribution

The communication via names enables the teams to be loosely coupled together. The operating system implements the link with the responsible process. The developer does not need to bother about the distribution of the teams among the computer nodes when designing and implementing the teams. The operating system establishes the absence of a service at a node and then automatically undertakes the necessary steps to guide the request to the correct node. For this purpose it must be able to access the information about the locality of the services. For this, families and their service groups are mapped at physical addresses. The developer generates a transmitter process and a receiver process which ensure access to the transmission medium and the protocol used respectively.

Other performances of the operating system

In the following, some of the operating system calls will be listed **point by point**. They illustrate, on the one hand, the power of the system and, on the other, they show the interfaces in the form of Modula 2 procedures

Time Management

Various calls are available for the purpose of delaying processes or waking up sleeping processes periodically.

All communications mechanisms can be connected to a timeout.

Memory Management

Processes can reserve for themselves parts of the global or private heap. Storage not needed any more is returned.

Interrupt Handling

The occupancy of an interrupt vector by an interrupt vector can be reported to the operating system.

Exception Handling

An exception process can be defined for each process. This becomes active when an exception occurs to the runtime of the normal process.

Exceptions are reported by the hardware or from processes.

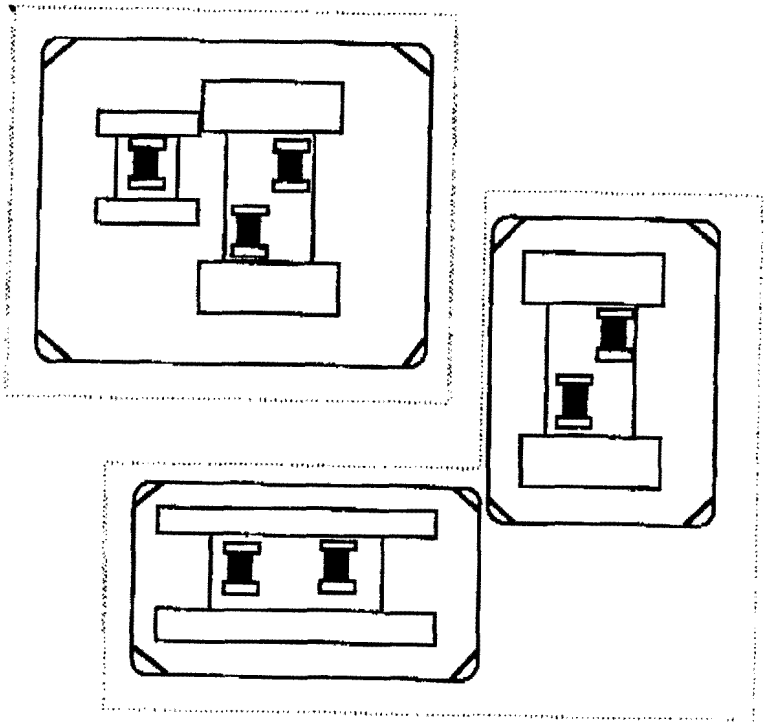


Fig. 14/2

In the Team Design part, processes which are responsible for the acceptance of the service requests are defined in the teams. At this stage their code is designed.

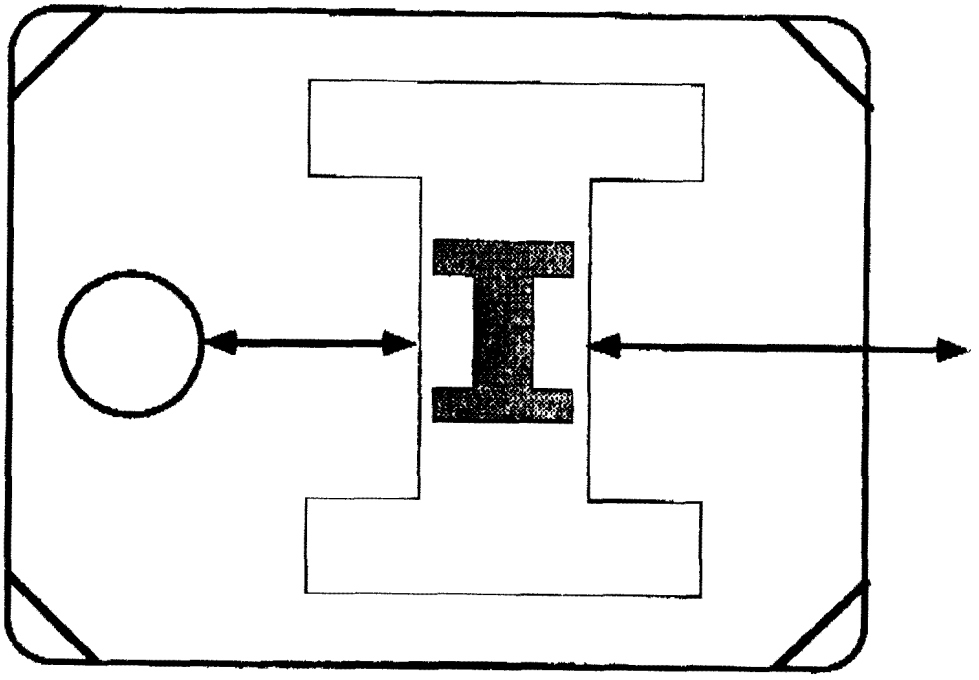


Fig.15

Each part has a check-list the purpose of which is to help the developer to ask himself the right questions. Decisions will be possible on the basis of the questions.