Eindhoven University of Technology

Faculty of Applied Physics

Department of Measurement and Computer Science (NF)

An Object-Oriented Model for EPEP

Eindhoven Program Editor and Processor

J.C. Nieuwenhuizen
August 24, 1995
NF/FTI 9506

Department of Measurement and Computer Science (NF)

Adviser: dr. ir. G.J.W. van Dijk
Supervisor: prof. dr. ir. K. Kopinga

Contents

List of Figures

1 Introduction

The construction of a software model was an essential step in the process of upgrading and porting EPEP (Eindhoven Program Editor and Processor) to multitasking operating system (MOS) platforms such as UNIX (System V) and the multiprocessor EMPS (Eindhoven MultiProcessor System) platform.

1.1 Automation of physics experiments

At the Department of Physics of the Eindhoven University of Technology, a standardized system was developed for the automation of physics experiments.

Automation of a physics experiment has two aspects, viz. experiment control and data acquisition. These are largely independent tasks that can best be handled in a multitasking environment that has a predictable real time behaviour.

The standardized automating system is composed of:

hardware: a general-purpose Physics Data Acquisition System (PhyDAS),

software: an interpretative development environment, the Eindhoven Program Editor and Processor (EPEP).

PhyDAS integrates the experiment control and data acquisition hardware. It is an assembly of standardized, general-purpose interface modules. Data transport and communication are performed via a specialized bus (PhyBUS). PhyBUS is separated from the computer bus, and hence the measuring hardware and measuring performance are independent of the computer that is used. In addition, the real time behaviour of the system is much better predictable.

EPEP is the combination of an object-based, multitasking operating system, an interactive program development environment, and an interpretative processor. The application language is a function-oriented (procedural) PASCAL-like programming language.

1.2 The object-oriented model

Figure 1.1 shows the schematic of the currently operating multitasking EPEP configuration (standard-EPEP). Because the EPEP System is the lowest software layer it must provide its own operating system. The most common EPEP application is the Program Editor.

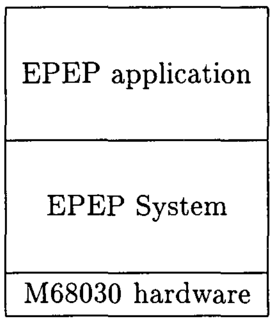

Figure 1.1. A standard-EPEP configuration consists of two software layers: the EPEP System, that runs on the Motorola M68030 processor, and an EPEP application program, usually the Program Editor.

Ongoing developments in multiprocessor systems, e.g., dependable distributed computing, together with the need for more computing power and increasing real time demands in physics experiment control, have led to the design and implementation of the Eindhoven MultiProcessor System (EMPS) and the EMPS multiprocessor executive [12] for distributed computing.

The EMPS platform was, apart from presenting an environment for dependable distributed computing, developed to control the PhyDAS hardware. It was designed to (gradually) replace the currently used single processor M68030-based system. The new multiprocessor implementation of

EPEP, for short called MPEP (Multiprocessor Program Editor and Processor), was primarily designed to run on the multiprocessor EMPS platform. Recent developments however, indicate that commercially available integrated microprocessor systems (RISC architecture) could easily replace the EMPS platform, and might very well present a cost-effective alternative.

The new MPEP implementation will use the external MOS kernel to handle all operating system tasks, which form an integrated part of the (single processor) standard-EPEP software.

Hence, MPEP must have a well defined interface to the MOS kernel. In addition, MPEP must provide multiprocessor primitives. The first step in the development of MPEP is the realization of a clear working model for EPEP.

On the occasions that specific MOS kernel issues must be addressed, the description of the EMPS executive will be used to provide the definitions. The facts that the EPEP design was object based [14], and the description of the EMPS executive is object-oriented, have led to the choice for an object-oriented description of EPEP.

A brief introduction to object-oriented design is presented below, full descriptions can be found in [8] and [9].

1.2.1 Object-oriented design

The object-oriented paradigm offers the combination of data-abstraction and information-hiding as a solution that makes complex models easier to design and verify. It does so by concentrating on the *objects* that are handled, in contrast to the conventional function-oriented paradigm, that focusses on the *tasks* to be performed.

The essence of the object-oriented model is the structured variable type *class*. An object-oriented model consists of class definitions and the relationships between these classes.

Instances of a class are called *objects*. A class is a collection of *features*, where a feature can be an *attribute*, representing a data field of an object, or a *routine*, specifying operations on objects. A class is the definition of one ore more data structures, supplemented by a set of specific operations that can be used to manipulate an object's data.

The general idea of object-oriented design is to create a small but complete set of routines for each class. Attributes and low-level routines are preferably hidden within the class. A feature that is accessible to other classes, is called an *export feature*. From the outside, a class looks like a black box of which only the relevant (i.e. accessible, exported) features can be seen; details of the implementation of the class are hidden. The total set of export features a class provides for use by other classes is called the *interface* of the class.

From a *parent* class *child* classes can be derived. A child class inherits the features from its parent class, but can also add or substitute some features. Constructing a whole family of classes that have the same interface, i.e. look quite similar from the outside, all members of this family can be handled alike, while each individual member will take care of its specific implementation aspects. In this manner a greater level of abstraction is achieved, which helps to concentrate on the essence, not troubled by the details of implementation.

1.3 Overview

The Object-Oriented Model presented for EPEP consists of two separate parts, viz. the Processor (EPEP System) and the Program Editor (User Interface).

The static part of EPEP is modelled in Chapter 2. It gives a new and clear view of the static part of standard-EPEP.

The model for the dynamic, multiprocessor version of EPEP (MPEP) is presented in Chapter 3. This dynamic model was designed as an extension of the static model. It has been the basis for the implementation of the multitasking and multiprocessor aspects of MPEP.

Chapter 4 describes the Program Editor. It provides the user interface for writing and executing EPEP application programs.

The Program Editor is actually an EPEP program that is interpreted by the EPEP System. It is, however, indispensable for the EPEP concept. Using the mechanisms of the EPEP System in an advanced way, it realizes a unique concept of modularity.

Classes and their relationships are represented using the graphical representation method of class diagrams [13]. Class definitions are described in the Eiffel object-oriented programming language [8].

2

EPEP: The Static Model

In this chapter, the model for the static part of standard-EPEP is presented.

The model is based on the object-oriented programming paradigm. It was designed using an internal report [16] and other literature [4, 5, 15, 17] on EPEP. The EPEP sources were consulted and various tests were performed to assure an accurate description. Although EPEP was not implemented in an object-oriented programming language, its design was object-based and hence EPEP is perfectly suitable for object-oriented modelling.

The goal of this first model is to get a clear picture of the internal structure of EPEP, viz. the EPEP application language interpreter, the operating system, the Library, and the user interface. This will form the basis for the multiprocessor model of MPEP.

Standard-EPEP is constructed of two software layers (Fig. 2.1). The lower layer is the EPEP System, that provides its own operating system. It consists of the Library (LIB), the operating system, and the Interpreter (INT). In this model, the operating system primitives are contained within the Library.

The operating system can be split-up into a static part, which includes basic input and output routines (I/O) and other static system primitives (STD), and a dynamic part. The dynamic system primitives (DYN) include memory management (MM), time management (TM), process management (PM), process synchronization (PS), and interrupt handling (INTH). These must, in the new version MPEP, be provided by the MOS kernel, and are not described in the static model for EPEP.

From outside the EPEP System, the only item that can be accessed through the system interface is the Interpreter. In the standard configura-

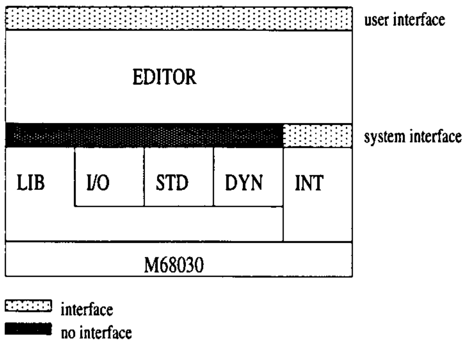

Figure 2.1. Standard-EPEP consists of the Program Editor and the EPEP System. The EPEP System is the lowest software layer.

tion, the only user of this interface is the Program Editor, when it invokes the Interpreter to handle the execution of an application program. Library maintenance and the use of Library primitives can only be performed from within an application program. When an application program is executed, it is handled by the Interpreter, that has access to the Library and thus to the operating system primitives.

In Chapter 4 the possibilities of using the operating system from within the Program Editor are explained by arguing that the Program Editor actually is a special kind of application program.

Configurations other than the one shown in Figure 2.1 would be possible, e.g. with the Program Editor replaced by another application program that has access to the system interface, but these are not very common. Normally, the Program Editor is used to start a subsequent application program. The Program Editor uses the primitives of the EPEP System to implement the 'library mechanism' as a part of the user interface.

2.1 Modular structure of applications

Software development is not confined to the merely correct implementation of an algorithm. Some basic aspects, which in fact help a correct implementation, have to be taken into account:

simplicity/clarity: In spite of the inherent complexity of certain tasks, it must be reasonably easy to verify whether a computer program meets the specified requirements.

reusability: For the performance of similar tasks, it should be possible to use the same program code.

extendibility/maintenance: A computer program should be rather easily adaptable to small changes in the specifications.

A software developing environment does a fine job if it encourages the programmer to write structured programs. This was the main reason to design the EPEP application language [4] based on a PASCAL-like structure [14]. Similar to a PASCAL program, an EPEP application program consists of a declaration part, declaring global routines and variables, and a preferably simple main loop. In addition, a user-extendible library is included within the system. This library holds global routines and variables for use by application programs.

2.1.1 The Library

A 'high-level' software developing environment, such as EPEP, must provide tools for creating modularly structured application programs.

The Library and the 'library mechanism' play an important role in this modularity. The Library is constructed of *units*, that define global routines and variables, which can be used by application programs. The EPEP System supplies the first unit that is stored in the Library, the *system unit*, that is always available. It provides basic input and output routines (I/O), standard mathematical functions and other general-purpose primitives (STD). After this first unit, other, user-defined units may be stored in the Library.

The hardware of the automating system, that is based upon standardized hardware interface modules, is complemented by standardized library units. Although most hardware interfaces are rather simple and easy to control, with several hardware interfaces goes a library unit, supplying standardized low-level control software. In this way, a programmer does not have to know the specific programming of complex hardware modules, and, more importantly, an application program does not 'need to know' how to operate some specific piece of hardware. High and low level tasks are separated, and hardware interfaces can be added or interchanged with a minimum of software redesign.

Applications that are designed to control complex experiments, often consist several standardized interface control units and one or more user-defined units.

2.1.2 Program vs. unit

In the previous section, the unit was introduced as a library building block, supplying global routines and variables for an application program. There exists, however, a close relationship between unit and application program.

When an application program is executed, all external routines and variables that the program uses, are obtained from the Library. Upon execution, the global declarations made in the program, the 'unit part', is split-off as an actual unit, which is subsequently stored in the Library.

Then, as stated above, these globals are available from the Library for any application program. The only candidate for using this new library unit is, of course, the application program itself. When an application program has reached the end of execution, the unit it defined is removed from the Library.

2.1.2.1 Application program vs. library program

The source of a library unit is written in the EPEP application language, using the Program Editor, in quite the same manner as an application program. As a unit is a library element, it is convenient to introduce the name *library program* for the 'source of a unit'. Library programs are used to group more generally reusable, to some extent experiment independent routines, or to structure large application programs.

A library program is executed just like an application program, and its 'unit part' is put in the Library. The difference with respect to executing an application program is that the new library unit must remain in the Library, so that other units may be added, or an application program may be executed, making use of the newly expanded Library. This is achieved by simply preventing the library program from reaching the end of execution, since at the end of execution, the newly stored unit would be removed from the Library. Just before a library program would reach the end of execution it invokes the Program Editor, which offers the user the opportunity to extend the Library once more, i.e. to execute another library program, or to execute an application program.

In Chapter 4 the implementation and the practical use of this feature of the Program Editor, that has become known as the 'library mechanism', are discussed.

2.1.3 The internal structure of a program

Is was mentioned above that the the EPEP application language is function-oriented and has a PASCAL-like structure. It is designed for the implementation and execution of algorithms, is highly structured, and has additional interface control features.

The implementation of a program consists of two parts. The first part defines *global variables* and *routines*. These definitions are called *declarations*. The second part is the routine where program execution starts, known as *main*. A variable represents some specific value or data, e.g., a number or a series of characters. A routine may define *local variables*, and specifies a list of actions. These actions are referred to as *statements*. A single statement can only be used to manipulate a variable or direct the program flow.

The structure of a program is often improved if certain statements are grouped, by forming a lists of statements. A list of statements, optionally preceded by a list of declarations, is called a *block*. Within this framework a routine is a block, and even an entire program can be seen as a block. Another grouping of statements is provided by the *structured statement*. A structured statement contains one or more blocks and may direct program flow to one of these blocks, depending upon certain conditions.

2.1.4 The cell concept

In the object-based design of EPEP, the *cell* is a key concept. For each type of routine and variable of the application language, a specific cell type was designed. At run time, an instance of each identifier is implemented as a cell of the corresponding type. All different cell types can be basically handled in the same way, which is one of the cornerstones of object-based design.

A cell holds the information that is associated with a particular identifier, i.e., the 'value' or 'meaning' of the instance. This may be a simple integer value for the instance of an integer, or a piece of program code for the instance of a procedure. Every operation on an identifier is actually an operation on the cell it represents. Similar to units, that construct the Library, cells are the elementary building blocks of a unit.

2.2 The Classes of EPEP

In the previous sections the main concepts of EPEP were presented without introducing any software classes. In the following sections the software model is presented by describing the main classes and their relationships. Before examining these classes one by one, they are summarized below.

		Internal	**Interface**	**Parameters**
S	*Interpreter*	CELL	PROGRAM	SOURCE
Y		STATEMENT		
S		BLOCK		
T	*Library*	CELL	UNIT	NAME
E		NAME	LIST of UNIT	CELL
M				
Editor			SOURCE	EDITOR_COMMAND

Figure 2.2. The internal and interface classes of static EPEP.

EPEP: The root class of the static model is the class EPEP. EPEP HOLDS THE LIBRARY, THE SYSTEM UNIT AND THE SOURCE OF THE PROGRAM EDITOR

SOURCE: The class SOURCE represents the source of an application program written in the application language.

UNIT: The class UNIT contains global routines and variables. These are made available to library programs or application programs when the UNIT is put in the Library. The class UNIT is a basic element of the class PROGRAM.

PROGRAM: The class PROGRAM is the executable form of SOURCE. A PROGRAM can only be created from a class SOURCE. To be executed it offers the routine **interpret**.

CELL: A cell in EPEP is represented by a class CELL. There is a whole family of CELLs, for each specific type of cell there is a matching CELL. CELL is an elementary building block of the UNIT.

STATEMENT: A statement is represented by a class STATEMENT. A whole family of STATEMENTs exists; for each specific statement of the application language there is a matching STATEMENT.

BLOCK: The class BLOCK represents the block in EPEP. It will be shown that a PROGRAM is a kind of BLOCK, and that a BLOCK handles program execution.

In Figure 2.2 the use of these classes is shown. The interface of the system is formed by the class PROGRAM, which takes the class SOURCE as a parameter. The system can only be used through the system interface. The only direct user of this interface is the Program Editor, when invoking the system to handle the execution of a library program or an application program. The task of the Program Editor is to provide a user interface for writing and executing library programs and application programs, i.e. providing means to construct a SOURCE, 'hand it over' to the Interpreter that creates a UNIT and a PROGRAM, and extends the Library with the UNIT and starts the execution of the PROGRAM.

2.3 The EPEP Root structure

The graphical representation method for classes and their relationships [13] as shown in Figure 2.3 and below, is summarized in Appendix A. The Eiffel object-oriented programming [8] is used to describe the class definitions. The standard class interfaces from the Eiffel library used in this model, are described in Appendix B.

2.3.1 The root class EPEP: EPEP at start-up

Starting EPEP implies the creation of an instance of the class EPEP (see Fig. 2.3). The main object EPEP initially contains the empty LIST **library**, the UNIT **system_unit** and the SOURCE **editor**. The **library** is a generic LIST of UNIT; items to be put in this list must also have the class UNIT as a base class (In this model only UNITs, that may be part of a PROGRAM, are put in the Library), items got from the **library** are always expected to be, and can therefore only be treated as, UNITs. The **library** is a global attribute of the system; every PROGRAM is allowed to use this **list**.

In this static model, the operating system is merely a set of system primitives for the performance of basic input and output (I/O). These system primitives, together with a standard library of mathematical functions and several other general-purpose primitives (STD), are contained within the **system_unit**. This **system_unit** is an instant UNIT, it is the only UNIT that is not created from a SOURCE. The **create** routine of EPEP, that is

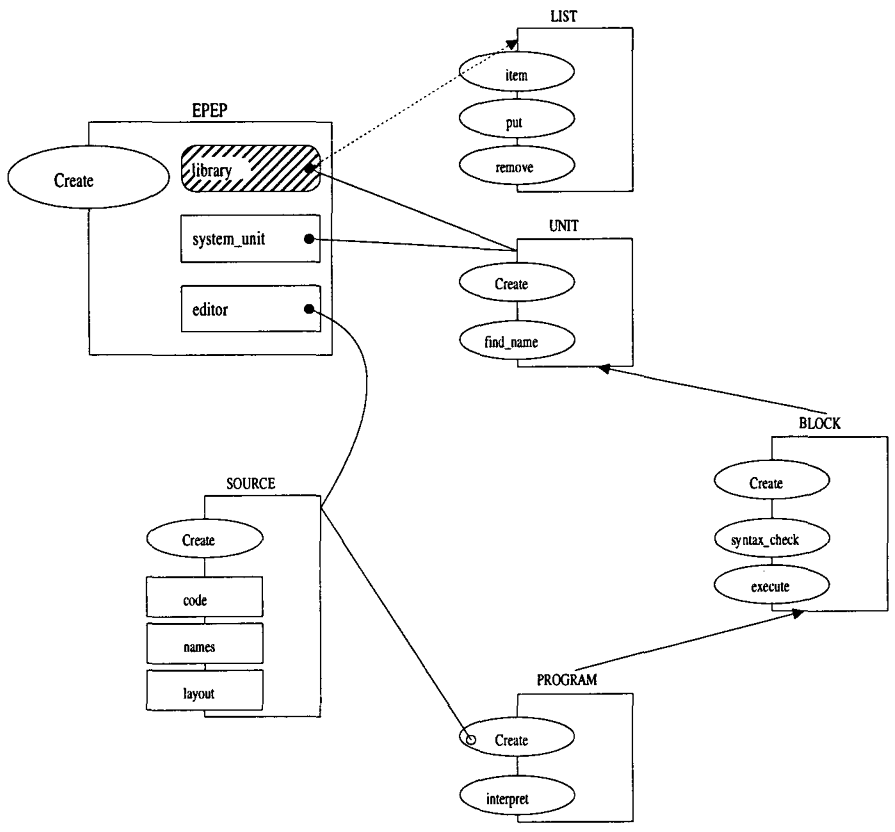

Figure 2.3. The Root of EPEP. Initially, an empty Library, the system unit, and the Program Editor are present in the system. The implementation of the Library, as a global, generic LIST of UNIT, and the use of a SOURCE for interface to create a PROGRAM is clearly shown. The classes UNIT and BLOCK are slightly simplified in this first diagram, for readability reasons.

invoked when creating the root object EPEP, performs four tasks, that quite obviously follow from the figure (see the class definitions, Appendix C):

i. the **system_unit** is put in the LIST **library**, using the routine **library.put**,

ii. from the SOURCE **editor** the PROGRAM **editor_program** is created,

iii. the UNIT part of the **editor_program** is put next in the LIST **library**,

iv. the **editor_program** is started by calling the routine **editor_program.interpret**.

EPEP is now ready for a user.

In the introduction of this chapter it was stated that the goal of the static model for EPEP is to clearly distinguish EPEP's four basic parts. So far the (i) Library, the (ii) operating system and the (iii) user interface were located as separate objects in the system. The missing fourth part, the Interpreter (or Processor), will not be found as one single object. Its functionality is offered by the STATEMENT family.

2.3.2 Class SOURCE: Writing an application

It is common use to write program implementations using one's favorite fully featured text editor. Hence, often the name program text is used for such an implementation. EPEP, however, does not work with plain ASCII program sources. The built-in Program Editor encodes the program text line by line, while it is being typed in. This encoding is fully reversible. The encoded program text produced by the Program Editor is used literally by the system interpreter. Using encoded program sources speeds-up program execution. Furthermore, because the (encoded) source itself is interpreted it must be stored in foreground memory. Using encoded sources that are shared between Program Editor and Interpreter saves substantial amounts of memory.

A high-level application language text can be distinguished into reserved words, special characters, combinations of special characters and identifiers. Encoding a source implies the representation of all these items by simple *tokens*. Each reserved word, special character, and special character combination has a specific token. Every identifier is assigned a special kind of token, a *tag*, numbered in order of appearance in the source.

A program source is principally encoded into two tables (Fig. 2.4):

code : a generic ARRAY of TOKEN. The intermediate program code, TOKENs and TAGs, that can be interpreted,

names : a generic TABLE of TAG to NAME of all identifiers used in the program.

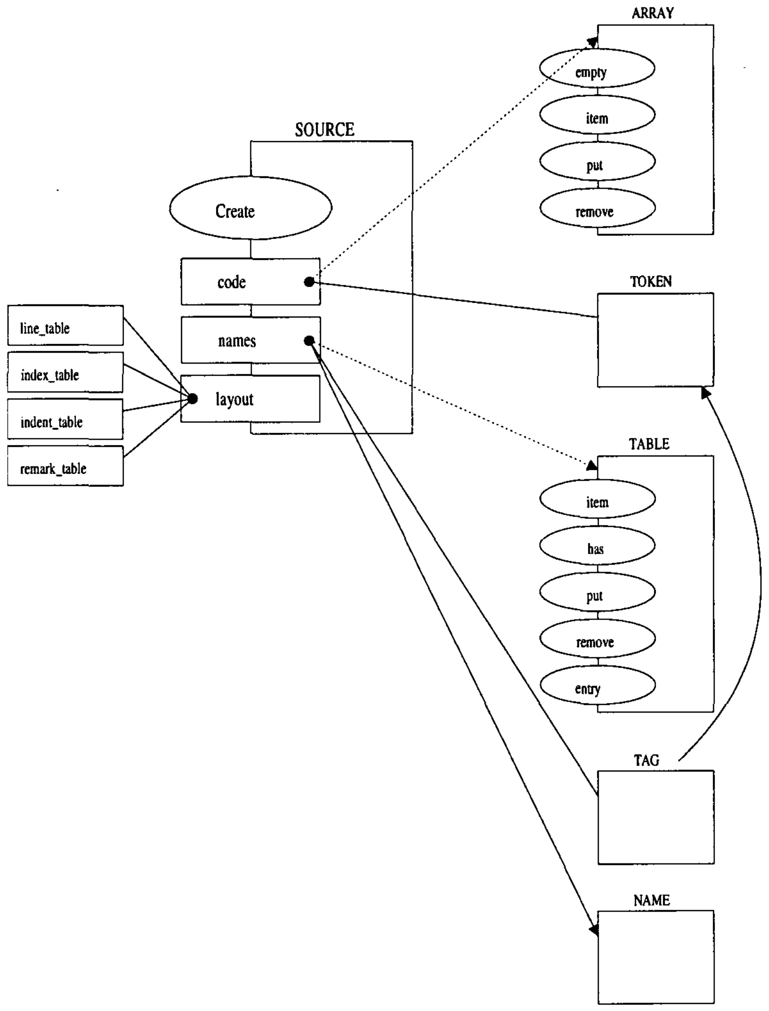

Figure 2.4. The class SOURCE implements the EPEP standard source of application programs. It mainly groups a number of tables. The layout tables are not shown in detail, which improves the readability; they are not discussed further in the model.

Besides these two tables that are needed for execution, a program source may contain four tables of layout information, used by the Program Editor only, to reconstruct the source text so that it may be viewed and re-edited. These are:

line_table: a table of line numbers,

index_table: a table of line breaks,

indent_table: a table registering the horizontal line indentation,

remark_table: a table of all comments.

All six source tables are grouped forming the class SOURCE (Fig. 2.4). The tables are placed on the class interface instead of being hidden within the class. The reason for choosing this solution is that many different operations of diverse origin may be performed on these tables, and on combinations of them. The Program Editor, that creates the SOURCE, has a variety of additional commands for filling and inspecting these tables or parts of them.

2.3.3 Class UNIT: The Library revisited

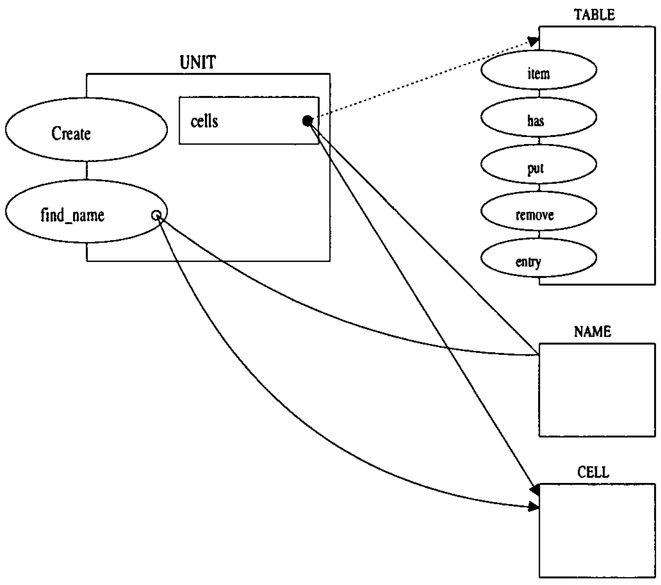

Figure 2.5. The class UNIT. The class NAME covers the name of an identifier; for the moment the explicit implementation is of little importance.

In sections 2.1.1 and 2.1.4 the unit was introduced as an elementary library building block that is consists of cells. The unit is represented in the model by the class UNIT. (Fig. 2.5). The UNIT is a simple class with only two features, the attribute **cells**, a generic TABLE of NAME to CELL and the routine **find_name**, using the classes NAME and CELL as parameters.

To explain the action of the class UNIT and its key position in the model, it is necessary to briefly consider the program execution. Interpreting the program code, which as a whole seems to be an arbitrary series of tags and tokens, implies grouping it into *statements*, small pieces of code that belong together, defining basic tasks that must be performed. A statement defines one, or a combination, of the following elementary operations:

i. reading the value of a constant or variable,

ii. assigning a value to a variable,

iii. evaluating an expression,

iv. comparing two values,

v. directing the program flow.

At a certain point a connection must be made between an identifier tag in the code and the cell it represents. The TAG is used to obtain the NAME of the identifier from the TABLE **names** (Fig. 2.4). From the **library** the most recently added UNIT is fetched (Fig. 2.3). Invoking its routine **find_name** returns the CELL that corresponds to the specific NAME (Fig. 2.5). If the identifier was defined in another, 'earlier' UNIT, the NULL cell is returned, indicating that the CELL of the identifier must be looked up in the next recently added UNIT. It is obvious now, why **cells** is a TABLE of NAME to CELL. By using the **name** of an identifier when searching the successive UNITs, it is avoided that an identifier, defined in an early UNIT must have the same TAG in every level of program code (remember that every UNIT results from a source that was tokenized by the Program Editor).

2.3.4 Class PROGRAM: A glimpse of the Interpreter

The new class PROGRAM, that is to be discussed here, cannot be presented without introducing the class BLOCK as well. The class PROGRAM is an heir of the class BLOCK, adding one feature, i.e., the routine **interpret** (Fig. 2.6). For now it is sufficient to know that BLOCK embodies an executable SOURCE and takes care of its execution. The class BLOCK is discussed later. The PROGRAM's **Create** routine is of great importance.

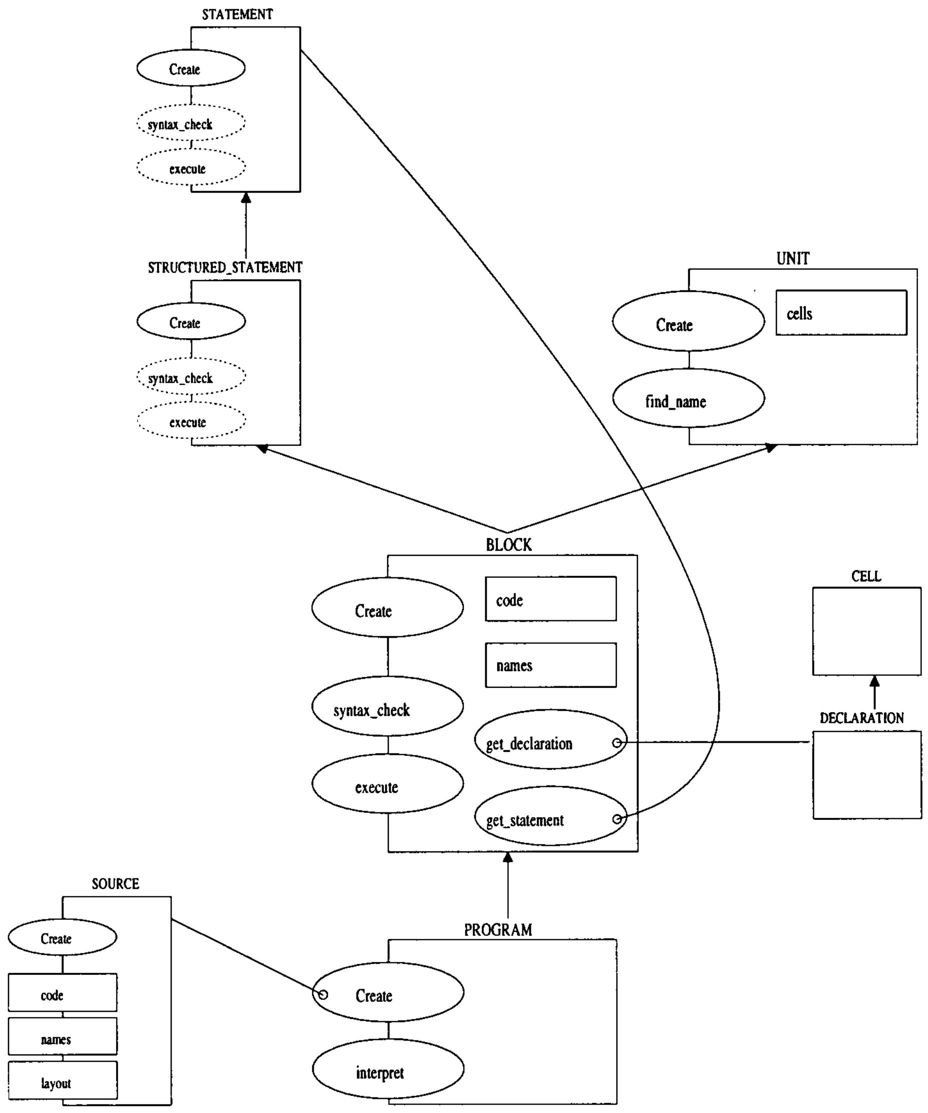

Figure 2.6. The classes BLOCK and PROGRAM.

It shows that from a SOURCE a PROGRAM can be created. The two tables of a SOURCE that are needed for the execution of a PROGRAM, viz. **names** and **code**, appear in BLOCK, where they will be used. Note that the routine **interpret** is added, but, the routines **syntax_check** and

execute, that will be discussed in section 2.4.2, are 'no longer' available on the interface; the only thing one can *do* with a PROGRAM, besides to create and store it, is, as would be expected, to interpret it.

2.4 The Interpreter

The last part of static EPEP that remains to be located, is the Interpreter. It is the most complex of all, because it involves most classes of static EPEP. It was mentioned above that in the object-oriented model, the Interpreter is not visible as a single class, but has a rather fragmented nature. Its functionality is offered by the individual contributions of the members of the STATEMENT family.

2.4.1 Class STATEMENT: the fragmented Interpreter

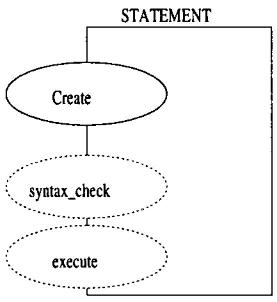

Figure 2.7. The base class STATEMENT offers the deferred routines **syntax_check** and **execute** on the interface. For every statement of the application language there is a specific class, that has the class STATEMENT as parent class, and implements the routines **syntax_check** and **execute**.

As the function-oriented EPEP application language has a PASCAL-like structure, an EPEP program source generally contains a list of declarations and statements. This fact is taken as the basis for the description of the Interpreter.

The Interpreter is modelled using the object-oriented techniques of *inheritance* and *dynamic binding*. The starting point is the class STATEMENT, shown in Figure 2.7. STATEMENT is a *base* class, it is designed only to serve as a parent class for a new group of classes. This 'group of classes' forms the family of STATEMENT-related classes; for every statement of the application language, a matching member class is developed, e.g., the class ASSIGNMENT, the class WHILE_LOOP, the class PROCEDURE_CALL. The only features the class STATEMENT has, are the routines **syntax_check** and **execute**, both 'dummy' routines in the implementation of STATEMENT. These are *deferred* routines, each member of the STATEMENT family provides its own implementation of both routines.

Summarizing, every statement of the application language is represented in the model by a unique class of the STATEMENT family. All STATEMENT classes can therefore be treated similarly, each class providing the routines that handle its specific execution. Examples of statement classes are presented in section 2.4.3.

2.4.2 Class BLOCK: the task list

The program source defines a list of actions to be performed, expressed in the application language. In section 2.3.3 such an action was further specified to be a statement, a small piece of code in the ARRAY **code** of a BLOCK (Fig. 2.6). Besides statements, a program source contains *declarations*. Just like the STATEMENT family, a DECLARATION family of related classes is designed. For every type of constant and variable, there is a specific DECLARATION member. Processing a constant or variable declaration described in the program source results in the creation of a CELL and a NAME, and subsequent storage of CELL in **cells** and NAME in **names**. This is the reason that a DECLARATION has the class CELL as parent class (Fig. 2.6).

The table **code** is parsed creating a matching class DECLARATION for each declaration and a matching class STATEMENT for each statement in the program source.

2.4.2.1 Creating the task list

As already mentioned above, STATEMENTs and DECLARATIONs are created parsing the table **code**. Creating a DECLARATION implies the dynamical creation of a CELL. STATEMENTs use these CELLs and are therefore also created dynamically, that is, each STATEMENT is created just before being executed.

The BLOCK embodies the general unit for executing the table **code** (Fig. 2.6). The table **code** of a BLOCK may represent an entire program, a piece of it, or even a single statement.

It was stated above that the table **code** represents a list of declarations and statements. It is the first task of the class BLOCK to interpret the **code**, creating CELLs and STATEMENTs. This task is performed by the private routines **get_declaration** and **get_statement**. As Figure 2.6 shows, the routine **get_declaration** returns a DECLARATION and the routine **get_statement** returns a STATEMENT, which they create by parsing the **code**.

The execution of a program follows the tracks of its block-structured implementation in the application language. Although discussed as 'application language detail', the concept of the *structured statement* must be mentioned here. A structured statement is defined to be a statement that groups, in any way, a number of statements. The class STRUCTURED_STATEMENT is an heir of the class STATEMENT. It is in turn the base class for the STRUCTURED_STATEMENTs. Just like all STATEMENTs, a STRUCTURED_STATEMENT may be created from the table **code**. The simplest of the STRUCTURED_STATEMENTs is the class BLOCK (Fig 2.6). A BLOCK groups statements, can be syntax-checked and executed. For further information see section 2.4.3.

The parsing of the program code during run time, which includes associating each statement expressed in the application language with specific executable routines just before it will be executed, is the essence of an interpreter-based system.

2.4.2.2 Executing the task list

The interpretation of a PROGRAM is split-up into two parts: syntax check and execution. The syntax of the **code** of a PROGRAM is completely checked before it is executed. This speeds-up the actual execution and, perhaps more importantly, it prevents a measuring or experiment control program from exiting due to a syntax error, which could mean the loss of valuable data or leaving the experiment in an undefined state.

Syntax check The routine **syntax_check** performs a syntax check in a broad sense. Besides the mere checking of the syntax, the **library** is searched for used library routines and variables, whose CELLs are copied (by reference) to the table **cells** of the PROGRAM that is to be executed. These CELLs remain in the table **cells**, ready to be used during execution—they have a global scope in the application program. All other CELLs are created dynamically and added to the table **cells** during execution by the interpretation of identifier declarations. This can save considerable amounts of memory and is a condition for reentrant or recursive routines.

The checking of the syntax of a BLOCK is performed in two steps:

i. The routine **get_statement** (or **get_declaration**) tries to create a STATEMENT (or DECLARATION) from the table **code**. Upon failure, a syntax error is found.

ii. If successful, the syntax of the newly created STATEMENT (or DECLARATION) is checked.

This process is repeated until all DECLARATIONs and STATEMENTs grouped in the BLOCK are syntax checked.

The creation of a SIMPLE_STATEMENT will only succeed from a valid syntax. The second step only has effect for STRUCTURED_STATEMENTs, and is therefore not very interesting if the newly created STATEMENT (i) happens to be a SIMPLE_STATEMENT (see section 2.4.1).

Consider the newly created STATEMENT to be a BLOCK. Checking the syntax of this STRUCTURED_STATEMENT implies the performance of *both* steps that are indicated above; the STATEMENTs grouped in this 'second' BLOCK are to be created and syntax checked (by the second BLOCK) too.

Execute The routine **execute** is quite similar to **syntax_check**. It performs both steps indicated above, the only difference being that instead of **syntax_check**, the **execute** routine on the interface of the newly created STATEMENT is invoked. Examples of specific implementations of **execute** routines are presented in the following sections. As this model shows, the table **code** is being parsed twice and therefore every STATEMENT is created twice (at least, think of loops), once during syntax check and once during execution. The second creation of STATEMENTs during execution takes up unnecessary execution time.

2.4.3 Details of the EPEP application language

The model presented so far is independent of the actual EPEP application language. It was shown how a function-oriented program, analyzed as a list of declarations and statements, is handled and executed by the system, without the necessity of introducing application language specifics.

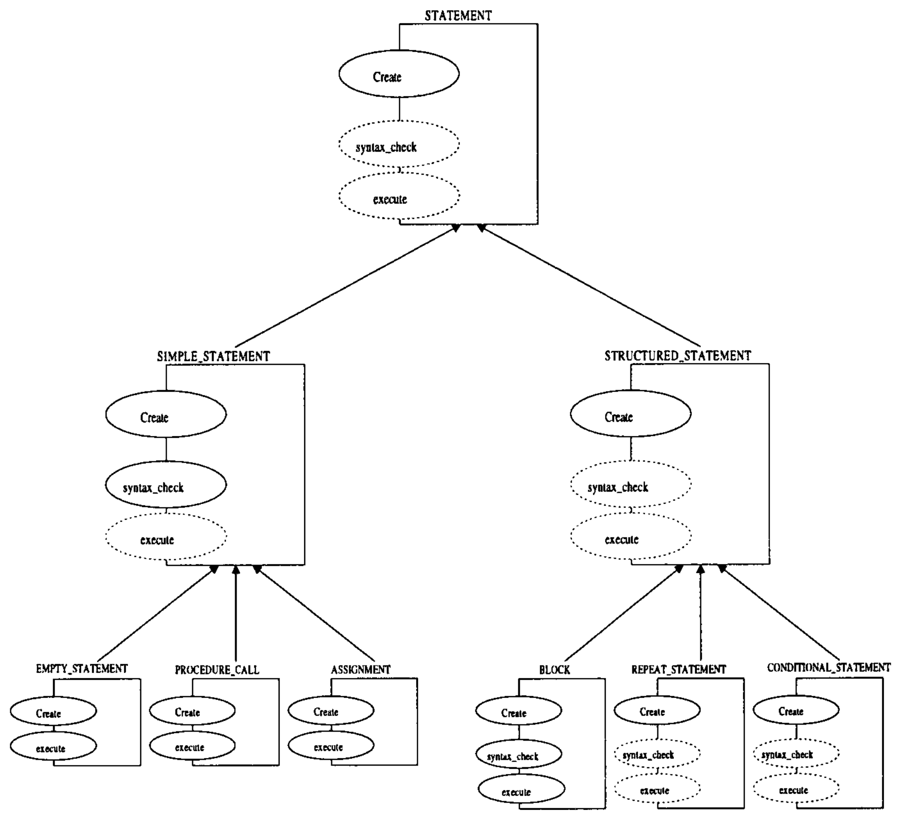

Figure 2.8. The STATEMENT is the base class of all STATEMENTs. The routines **syntax_check** and **execute** are deferred, and are implemented by child classes. The routines **syntax_check** and **execute** of the classes REPEAT_STATEMENT and CONDITIONAL_STATEMENT are still deferred, as both these statements are a base class of respectively the group REPEAT_STATEMENTs and CONDITIONAL_STATEMENTs.

This remarkable fact is a consequence of the adoption of the object-oriented paradigm. Basically, the execution of a program is the sequenced execution of single statements, which obviously are language-dependent. This exactly defines the gap that was left yet unfilled, the explicit execution of specific statements. To round off the description of this static model, the language specific statement details are presented, and examples of statements are given.

Statements are divided in two main subgroups, the simple statements and the structured statements. All STATEMENTs have the class STATEMENT as a base class and can therefore be treated in a quite similar way. The hierarchy of STATEMENTs is shown in Figure 2.8.

2.4.3.1 The simple statements

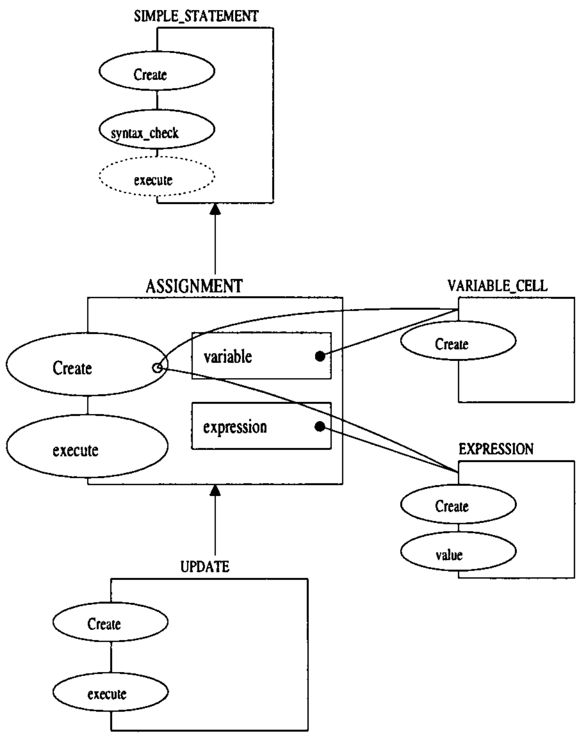

Figure 2.9. The ASSIGNMENT statement. No syntax check routine is provided, for it is already implemented by its base class SIMPLE_STATEMENT. For the execution of an ASSIGNMENT a VARIABLE_CELL and an EXPRESSION are needed. The VARIABLE_CELL is fetched from the table **cells** of the current BLOCK, the EXPRESSION is created parsing the array **code** of the current BLOCK. Executing the ASSIGNMENT implies calculating the value of the EXPRESSION and assigning it to the VARIABLE_CELL.

There are three simple statements, viz. the empty statement, the procedure call, and the assignment. When parsing the table **code**, the structured statements are easily identified by their unique token. This is not the case for simple statements; simple statements do not (cannot) have unique tokens.

The procedure for the identification of simple statements is as follows. If the routine **get_statement** finds the starting token to be a statement delimiter, this indicates an empty statement. If the starting token is an identifier tag, the statement must be either a procedure call or an assignment. These statements are distinguished by looking at the following tokens, e.g., an assignment contains an *assignment operator* followed by an *expression*.

The ASSIGNMENT statement As an example of a simple statement, the ASSIGNMENT statement is presented (Fig. 2.9). The UPDATE statement is very similar to the ASSIGNMENT statement. The only difference is that the old value of the variable, that is to be assigned the new value, is used first to evaluate that new value.

2.4.3.2 The structured statements

There are three groups of structured statements: the block, the repeat statement, and the conditional statement. A structured statement generally groups one or more statement lists. All statements must be syntax checked and executed individually, which implies that a structured statement must be able of creating and executing statements. In all STRUCTURED_STATEMENTS BLOCKs are used to perform these tasks.

The IF statement The creation of a STRUCTURED_STATEMENT from the table **code** involves the creation of one ore more BLOCKs. As an example of a STRUCTURED_STATEMENT, the IF statement is presented (Fig. 2.10).

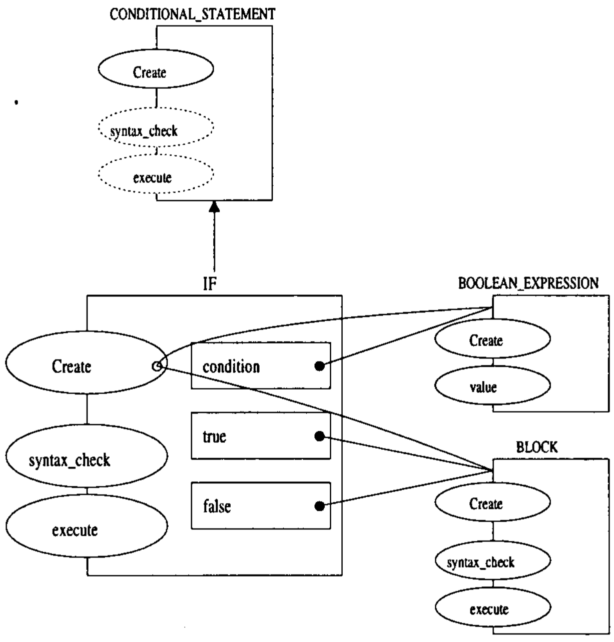

Figure 2.10. The IF statement. Syntax checking an IF statement involves syntax checking both true and false BLOCKS. Executed will only be one of these BLOCKS, depending on what the **condition** evaluates to.

3

MPEP: The Dynamic Model

The dynamic model for MPEP, described in this chapter, is presented as a logical continuation of the static model for EPEP. Besides the use of time and process management primitives, as suggested by the qualifier 'dynamic', this new model handles multitasking and multiprocessor aspects, and hence the name change to MPEP. As suggested in Chapter 1, this dynamic model is used for the implementation of the multitasking and multiprocessor aspects of the MPEP software. The essence of EPEP, the static model, is maintained, and its description given in the previous chapter will be used almost literally. MPEP assumes the target platform to provide elementary multitasking, and optionally multiprocessor, facilities. A list of MOS kernel requirements is presented in Appendix F.

The dynamic model has three layers of software (Fig 3.1). The figure has three interesting starting points. Firstly, the figure does not explicitly show a multi*processor* environment. Although the multiprocessor EMPS system is an important target platform for MPEP (see Appendix D), it should work quite as well in single processor EMPS configurations and single processor systems. This leads to the second point of interest, the MPEP module. An MPEP module operates standalone, while providing means of communication with other MPEP modules. Therefore, MPEP does not have global control over all (MPEP) activities on a MOS platform, as more than one MPEP module may be present. The third point of importance the figure shows, is the fact that MPEP, uses the MOS kernel. Whereas the standard-EPEP software has an embedded operating system, MPEP does not have to provide the lowest layer of software, because MPEP does not 'run directly on the hardware'. The operating system of EPEP is substituted

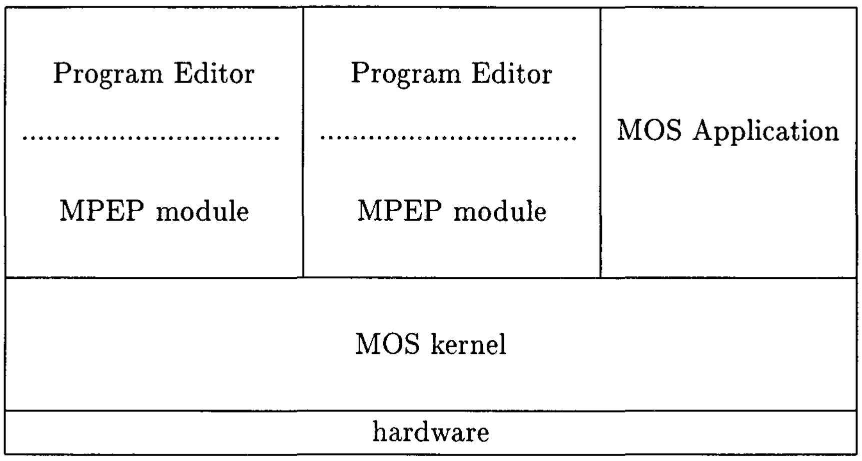

Figure 3.1. An MPEP configuration consists of three software layers. The lowest layer is the MOS kernel, that provides the multitasking (multiprocessor) operating system.

by an operating system interface (OSI), that provides access to the (MOS) operating system. This is a major step towards portability, see Appendix D.

3.1 Multitasking aspects

In the following sections, the definitions involving multitasking aspects are taken from the (object-oriented) description of the EMPS multiprocessor executive [12]. Two reasons for this are that (i) the description is object oriented, and (ii) the EMPS system is an important target platform for MPEP. Although the EMPS platform is a multiprocessor system, it is above all a *multitasking* system. On a multitasking platform several tasks may be executed simultaneously, or, when running on the same processor, quasi simultaneously. Standard-EPEP is multitasking and has therefore several time and process management primitives that are substituted by the MOS kernel, see Figure 2.1. This means that multitasking EPEP and moreover the EPEP application language have certain conventions that must be maintained. On the other hand, upgrading from a single to a multiprocessor or multitasking system requires additional features, especially for interprocess communication, in MPEP as well as in the upward compatible MPEP application language. In the following sections both aspects are considered.

3.1.1 Processes: Concurrent tasks

Separate tasks that may be performed simultaneously are called processes. Simply spoken, in a multitasking system the process replaces the program in a conventional single tasking system. Thus, the multitasking system is capable of executing several programs, now referred to as processes, simultaneously. As a consequence, any separate task may run as a standalone process, instead of being part of a larger program.

3.1.1.1 The EMPS kernel process

Every EMPS application (and every system program) is based on one or more EMPS kernel processes. The execution of a program implies creating and starting a new EMPS kernel process.

A very illustrative example forms the *command line* process. The EMPS kernel provides a command line process (known as *shell* on other platforms) for loading and starting processes. Starting a process is performed similarly to executing a program from the command line in a single tasking environment. The major difference is that just after giving the command that starts the new process, the command line process is ready for the next command. In this way a number of processes can be started to run at the same time.

Processes that run concurrently on the same processor (CPU), must share the real CPU execution time. The process and time management techniques, performed by the EMPS kernel, are described in [12]. Here only the major process scheduling aspects will be mentioned.

A process always is in one of the three major states, viz. CURRENT, READY, and BLOCKED. A process is said to be CURRENT when it is allocated the CPU and is thus actually taking up processor time. A process that is eligible for execution, but is not CURRENT, is in the READY state. A process not ready for execution is said to be BLOCKED.

Every process is assigned a level of priority. The process that is CURRENT will always have a priority that is higher than or equal to all other READY processes. This implies that only processes of equal priority will run truly simultaneously, a process of higher priority will not share but take up all CPU time, a process of lower priority will not get any CPU time as long as higher priority processes are READY. High priority processes are commonly used for quick tasks that must have a short response time (granularity). Such processes are 'sleeping' (state BLOCKED) most of the time, to become 'active' (state READY) only at the moment they have to handle their task.

The EMPS kernel provides the process concept fully transparent with respect to the physical process location. The difference between two processes running on the same CPU and two processes running on separate CPUs, is that in the latter case each process will get more real execution time, and therefore they will run faster.

3.1.1.2 The MPEP process

The MPEP model also defines processes. From here on, an EMPS kernel process will be referred to as kernel process. An MPEP process, or process for short, is created using a kernel process, but offers additional MPEP features.

MPEP supports three related kinds of processes.

The first to be mentioned is the process MPEP itself, the process that is created and started to form an MPEP module. This is a unique process, there is only one process MPEP in an MPEP module.

The other two are processes defined in an application program. One of these matches the processes found in multitasking EPEP. The EPEP application language offers commands to declare and control child processes, for handling separate tasks. Such a child process is, quite like a routine, a block, i.e., a list of statements expressed in the application language. To allow standalone execution of these processes, each process has a Library, similar to the Library of the EPEP system. When a child process is started, the child receives a copy of the entire Library of the parent process, so that it can access all global variables that were declared before.

The remaining kind of process handles an entirely new concept, which does not exist in the currently used form of EPEP, the *remote procedure call*. The remote procedure call is discussed in section 3.1.3.

3.1.2 Semaphores: Process synchronization

For processes that run concurrently on the same processor, the EMPS kernel provides the *semaphore* to ensure mutual exclusion [6]. The multitasking EPEP kernel supported a similar solution for the problem, called the *signal*. In addition, a signal could be associated with a PhyBUS interrupt, providing an easy way of responding to a request of a hardware interface.

In MPEP, the signal will be replaced by the EPMS semaphore, while maintaining its original (EPEP) functionality and (confusing) language syntax.

3.1.3 The dynamic Library: Remote procedure call

So far, solely multitasking aspects of MPEP have been highlighted. This is partly caused by the concept of MPEP, since MPEP is composed of single processor modules that resemble the multitasking implementation of EPEP. To take full advantage of the EMPS multiprocessor system, there is a need for means of communication between independent MPEP modules. The solution is presented in the form of the *remote procedure call* mechanism.

The MPEP application language is equipped to define specific routines as *import* or *export* routines. A routine is enabled to be exported by adding the reserved word **export** to its header. Such a routine can still be invoked in the normal way, but in addition it becomes available to other MPEP modules. An imported routine is declared only as a heading: the type, name and actual parameter list. The reserved word **import** indicates that the routine is implemented in another MPEP module, where it is defined as export.

Although the most interesting case occurs when MPEP modules running on different processors supply such an import/export pair, this is not a requirement. This has the advantage that even multi-module applications are largely independent of the actual hardware configuration.

3.2 The Classes of MPEP

The object-oriented model for MPEP defines, besides the introduction of EMPS kernel interface classes, only three new major classes.

MPEP: The root class MPEP is a redesign of the root class EPEP of the static model. MPEP, like EPEP, holds the Library, the system unit, and the source of the Program Editor, but defines in addition a list of exported routines and is derived from the base class PROCESS.

PROCESS: The class PROCESS is the base class of the new classes of the dynamic model. It is based on a kernel process and simulates the process of multitasking EPEP.

REMOTE_PROCEDURE: The class REMOTE_PROCEDURE handles a remote procedure call.

3.3 The MPEP Root structure

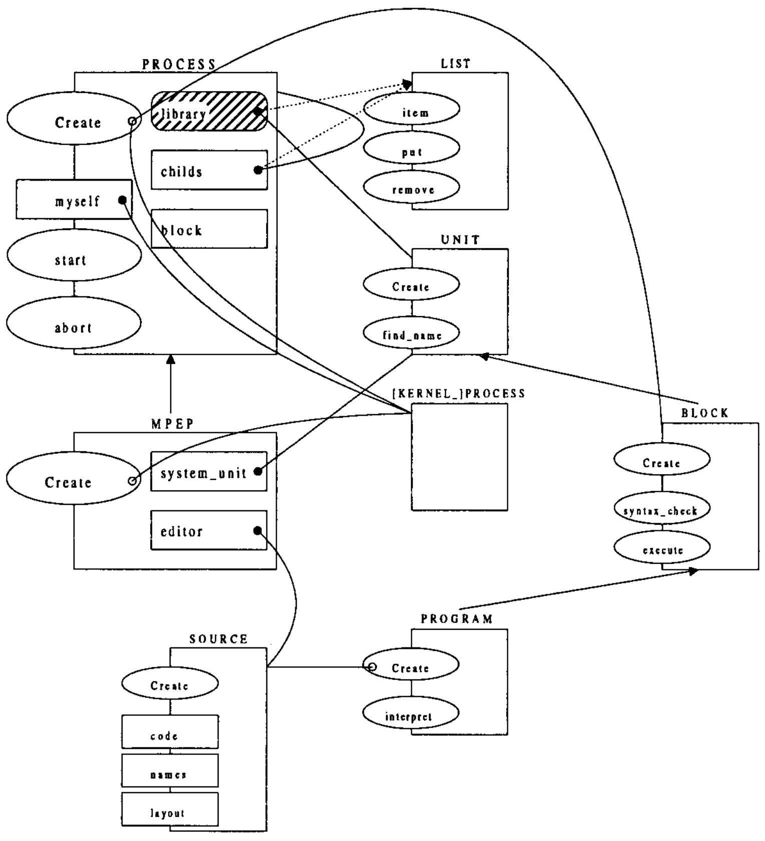

Figure 3.2. The Root of MPEP is an extension of the Root of EPEP. The class PROCESS is introduced, based on the class [KERNEL_]PROCESS. The **library** has been 'moved' from the Root to its base class PROCESS. The EMPS kernel class PROCESS is, in line with the convention assumed in the text, renamed to [KERNEL_]PROCESS.

3.3.1 The root class MPEP: MPEP at start-up

An MPEP module is initiated by creating an instance of the class MPEP (Fig. 3.2). In agreement with EPEP, the main object MPEP initially contains a **library**, a UNIT **system_unit** and a SOURCE **editor**.

The **system_unit** maintains the functionality of its static counterpart, but instead of the implementation of operating system routines, in the dynamic model it contains mainly an interface (OSI) between MPEP and the MOS kernel primitives.

A user starting an MPEP module will not notice any important differences compared to starting EPEP. The MPEP starting procedure is identical to the EPEP starting procedure given in section 2.3.1. The last step is the execution of the Editor, that awaits user input.

The process aspects of MPEP that it inherits from its base class PROCESS are discussed in the following section.

3.3.2 The class PROCESS: Concurrent tasks

The class PROCESS was already presented in Figure 3.2. A PROCESS can be created and started from within an application program. This is done in an indirect way, with the aid of DECLARATIONs and STATEMENTs; the application language is function-oriented and does not handle objects. Similar to a routine declaration, the statement list is assigned to a BLOCK. As mentioned above, each PROCESS has its own Library, that may be expanded for exclusive use by the owner, which probably will never be done, but is supported because of compatibility with multitasking EPEP.

Most statements that operate on a PROCESS, e.g., suspending or reinitiating execution, reading or setting priority, etc., are implemented by direct access of the [KERNEL_]PROCESS **myself** and invoking the EMPS kernel **scheduler** [12]. The **start** and **abort** routines are explicitly defined by the PROCESS itself, as these tasks comprise more than simply readying or killing a [KERNEL_]PROCESS. When starting a PROCESS, the **library** must be copied; a PROCESS that is aborted must clean-up the **library** and abort all child PROCESSes. The details of both routines are described in Appendix C.

3.3.3 The class REMOTE_PROCEDURE: RPC

The remote procedure call is implemented by the class REMOTE_PROCEDURE, which is an heir of class PROCESS (Fig. 3.3). REMOTE_PROCEDURE uses, besides the [KERNEL_]PROCESS **myself**, that is inherited from PROCESS, the EMPS kernel classes MAILBOX and RESPONDER_PORT.

It must be stressed that the class REMOTE_PROCEDURE, that belongs to one MPEP module, the server, implements the execution of a procedure,

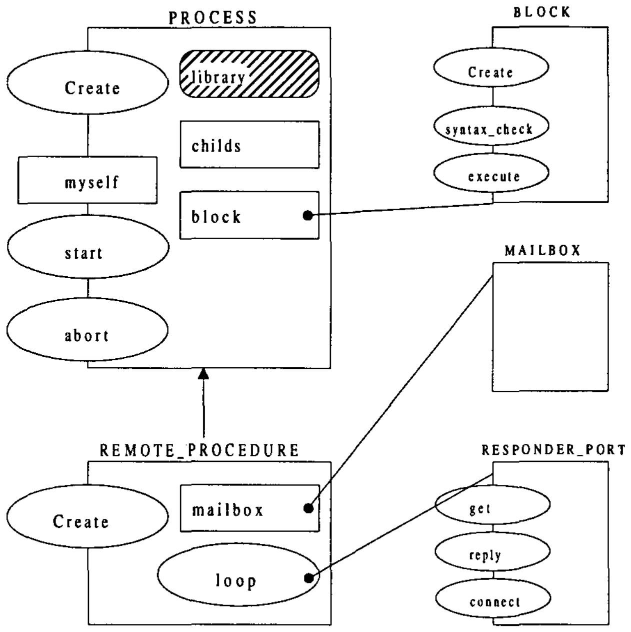

Figure 3.3. REMOTE_PROCEDURE handles the execution of a remote procedure call request. For readability reasons several implementations that were shown in detail in Figure 3.2 are not displayed, and the EMPS kernel class RESPONDER_PORT has been simplified.

in response to a request from another MPEP module, the client. Note that the procedure that is executed is local to the module of the REMOTE_PROCEDURE (server), and that, although REMOTE_PROCEDURE is an heir of PROCESS, the client module does not start a remote process, but only invokes a remote procedure. When a process invokes a remote procedure, it will be BLOCKED until the remote execution has been completed (if only to assign the return parameter). This implies that an MPEP application intended to distribute certain tasks by invoking remote procedures must define separate child processes that do so. In addition, such procedures should be exported by MPEP modules that preferably run on different processors.

3.3.3.1 Handling an RPC request

The REMOTE_PROCEDURE owns two attributes, viz. a MAILBOX and a RESPONDER_PORT that is linked to this MAILBOX. The process of the REMOTE_PROCEDURE is BLOCKED until an RPC request is delivered in the MAILBOX. At that time the MESSAGE_BUFFER that contains the actual parameter list is obtained from the MAILBOX using the routine **get** of the RESPONDER_PORT. Next the **block** is executed and the **reply** routine of the RESPONDER_PORT is invoked with a MESSAGE_BUFFER containing the return parameter. The handling of the RPC request is now completed and the REMOTE_PROCEDURE will be BLOCKED until another request is done.

3.4 Details of the MPEP application language

The MPEP application language is upward compatible with the EPEP application language. This has not been too difficult, as up to now only two new reserved words, viz. **import** and **export**, were introduced. The justification for the present section on language details is that in Chapter 2 all multitasking aspects of EPEP were not considered. These multitasking aspects have implications for the application language that were not discussed in the static model, e.g., the definition and implementation of specific process related STATEMENTs.

3.4.1 The class SIGNAL

The variable type SIGNAL, as defined in the EPEP application language[4], provides means of process synchronization. Signal is an alias for the more commonly used name semaphore. The type SIGNAL has a set of two routines, viz. **wait** and **send**. The **send** routine is an alias for the **signal** operation on a semaphore. (Fig. 3.4).

The **signal** of EPEP is implemented by MPEP using the class SEMAPHORE of the EMPS kernel, described in [12]. In the EPEP application language, the user can create and initialize a SIGNAL at one time, and may decide later to associate this SIGNAL with a specific PhyBUS interrupt. The EMPS kernel, however, does not allow such an implementation. It has a static list of semaphores, one for each PhyBUS interrupt, the PhyBUS semaphores. New semaphores can be created, but these cannot be associated with PhyBUS interrupts.

The MPEP application language therefore defines two kinds of SIGNALs, **signals** that are associated with a PhyBUS interrupt and SIGNALs that are not. The only difference between both kinds occurs in the declaration. If a PhyBUS interrupt number is supplied with the declaration

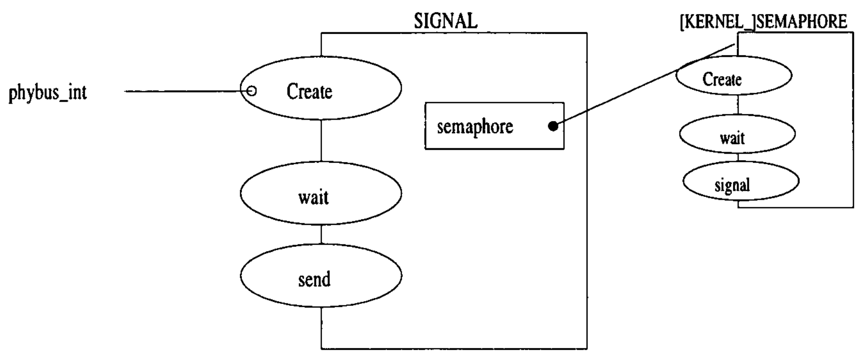

Figure 3.4. SIGNAL provides the functionality of the EPEP signal. If with the declaration of a SIGNAL a (valid) PhyBUS interrupt number is supplied, the SIGNAL is associated with the corresponding EMPS PhyBUS [KERNEL_]SEMAPHORE. It is merely an interface to the [KERNEL_]SEMAPHORE, that is responsible for the implementation.

of a variable of type SIGNAL, this SIGNAL is associated with the corresponding EMPS PhyBUS semaphore. Declaring a variable of type SIGNAL without an interrupt number results in the creation of an EMPS kernel object SEMAPHORE. In both cases, subsequently an object SIGNAL is created and is put in the Library. The SIGNAL provides the interface to respectively the corresponding PhyBUS semaphore, or the newly created [KERNEL_]SEMAPHORE.

3.4.1.1 The signal wait and send operations

Executing a **wait** or **send** operation on an earlier declared SIGNAL is expressed in the application language simply as a procedure call, that has as actual parameter a reference to the specific SIGNAL. The **system_unit** defines these two routines, **wait** and **send**.

Upon calling a **wait** or **send** routine, the specified SIGNAL cell is fetched from the **library** and the matching routine of the corresponding [KERNEL_]SEMAPHORE is invoked.

4

The Program Editor

In this Chapter, a brief description of the Program Editor is presented. The Editor, which itself is an EPEP application program, implements a user interface to EPEP. Reasons for including the Editor in this model are that (i) because the Editor is an EPEP application program, it shows the connection between the EPEP system and EPEP application programs, (ii) the Editor can be considered to be an essential part of EPEP, and (iii) the Editor introduces the concept of modularity in an interesting way (this is the so-called 'library mechanism', that is commonly appreciated as a standard part of EPEP). This means that a more comprehensive discussion of the Editor would not add any really interesting aspects to the model for EPEP.

To find out more about Editor specifics, a user manual of the Editor is presented in [4].

4.1 The Editor characteristics

In an EPEP configuration that is set-up for normal day use, the Program Editor is automatically executed at startup. These are its main features:

- The Editor is used to write, examine, and alter application programs and to 'run' them. Running a program implies the creation of a PROGRAM using the SOURCE, and the subsequent interpreting of this PROGRAM.

- The Editor is command line based, which is common for interpreter based systems. This has the advantage that command execution and entering a program can be interchanged. The program source text must be entered line by line, the user can't 'walk up and down the screen'.

- The Editor offers a transparent and interactive environment. The user can inspect and change Library variables and also use Library routines directly, by entering EPEP statements as if they were Editor commands.
- The Editor offers a number of simple commands and several standard EPEP routines. One of these routines is the 'magical' routine **monitor**, that can be invoked to put the 'library mechanism' into effect.

4.2 The Editor operation

The Editor is driven by *editor commands* (Fig. 4.1). The editor commands form the user interface to the Editor, and to EPEP. The **editor_loop** reads a line of text entered by the user, parses it to create the matching editor command, and executes this command. After that, another line of input is read, and the procedure is repeated.

There are three types of editor commands: the command, the program line, and the direct statement.

- The Editor supplies a range of commands that are useful when creating a source or running a program, such as LIST (display the source currently edited) and RUN (create program and interpret). Program sources that were created with the Editor can be SAVEed to, and LOADed from background memory.
- If a command line begins with a line number, this line is encoded and inserted into the SOURCE that is currently being edited.
- Otherwise, if an entered line is neither a command, nor a program line, the Editor assumes a direct statement. First, the line is enclosed within a BEGIN and an END. Next, a temporary SOURCE is created and the line is encoded into this SOURCE. Finally, using this SOURCE a PROGRAM is created and interpreted. So typing for example

  ```
  WRITE( <library string> ) <enter>
  ```

  will directly display `<library string>` on the screen.

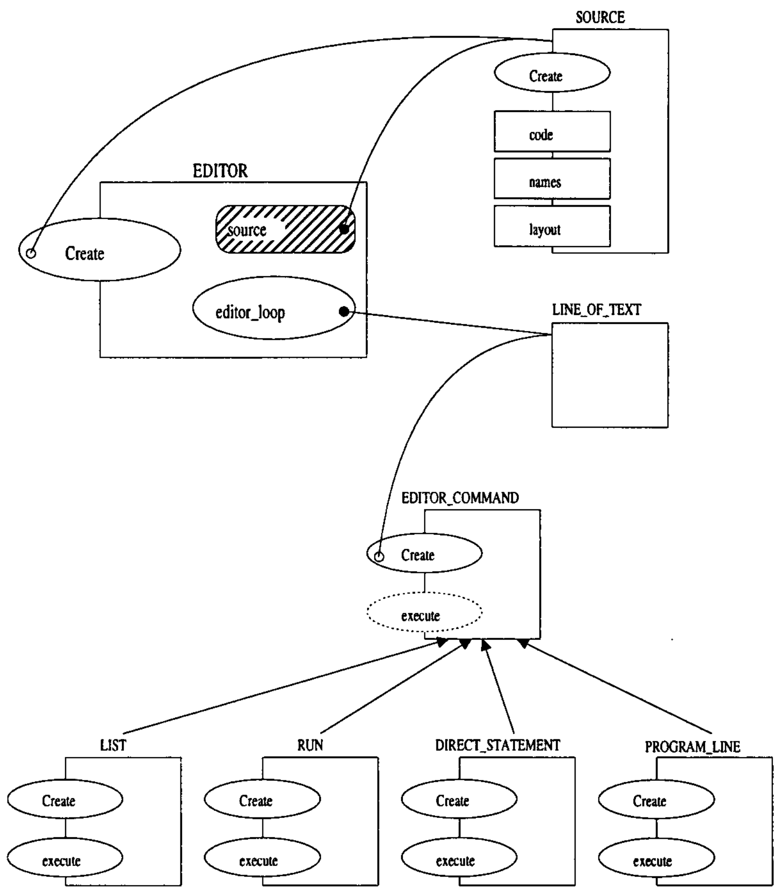

Figure 4.1. The EDITOR is driven by the EDITOR_COMMANDs. There are three different types of EDITOR_COMMAND, viz. the COMMAND, DIRECT_STATEMENT and PROGRAM_LINE. LIST and RUN are typical examples of the type COMMAND.

4.3 The library mechanism

The Editor has a main routine called **monitor** that invokes the command line input handler (the **editor_loop**). When a program is interpreted it will first be put in the Library, that is, its UNIT is added to **library**. The Editor is the first PROGRAM to be interpreted, and is the second UNIT in the Library (first is always the system unit), supplying the routine **monitor** as a library routine.

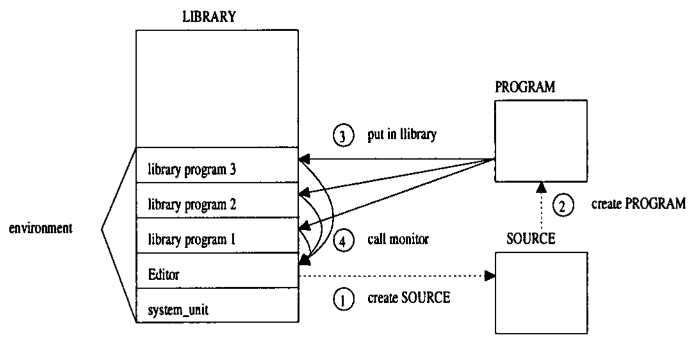

Figure 4.2. The library mechanism provides modularity in an interesting way.

A new source created with the Editor will be added to the Library, when the user enters the command RUN. If this program calls the routine **monitor**, the user finds himself in the command line Editor again; it may seem to him (m/f) that the program was added to the Library and that the Editor is ready for a new source to be entered (Fig. 4.2).

By entering the command UNLOAD the command line routine **monitor** is left. Usually, calling the routine **monitor** is the last statement of a library program, so that when **monitor** is left the execution of this program is completed. The program is then removed from the the Library and program flow returns to the caller of **interpret** on the PROGRAM's interface. If this caller was the routine **monitor**, the user once again finds himself in the command line Editor. The current SOURCE in this case is the source of the executed PROGRAM; from this SOURCE the **program** was created!

4.4 More efficient Program editing

As explained above, the Program Editor encodes program lines immediately after they have been typed in. A program source that is stored in background memory also has this encoded form. Although the Program Editor provides several features that make the command line based input a bit more user-friendly, it is often preferable to write EPEP program sources using an external text editor; an EPEP System charged with dedicated hardware for experiment control is not exactly the most appropriate platform for the mere entering of program sources. Therefore the Program Editor provides the feature of reading and encoding a plain text file.

The procedure of using an external text editor as described above has the disadvantages that (i) the program cannot be easily checked online (not even the syntax), (ii) each new version must be converted before it can be used (executed or checked), which, in addition, makes version control error-prone, and (iii) the Program Editor is still needed for the conversion and execution of the program.

In Appendix D a possible improvement on these problems is presented, offering the additional feature of online (context) help.

Appendix A

Graphical representation

In this Appendix classes and their relationships are represented using the graphical representation method of class diagrams [13]. The symbols and relationships are defined in the figures below.

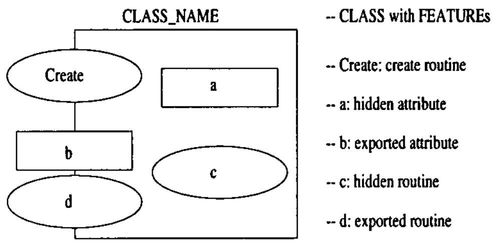

Figure A.1. A *Class* with features.

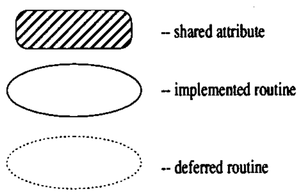

Figure A.2. The *scope* of routines and attributes.

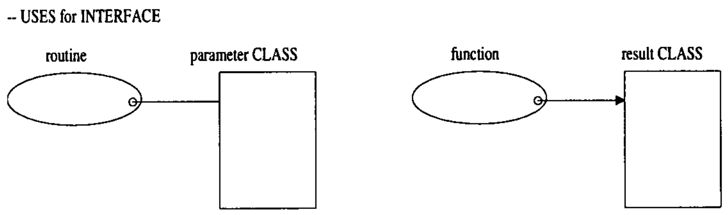

Figure A.3. The *uses for interface* relationship defines the routine parameter specifications, including the function result parameter.

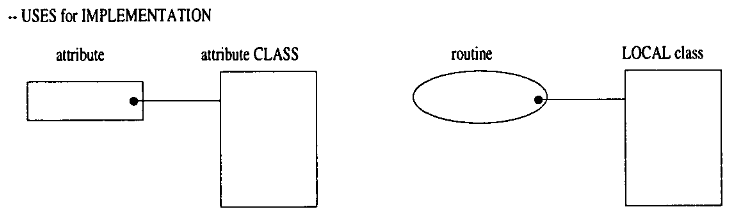

Figure A.4. The *uses for implementation* relationship defines the class type of an attribute, or the class type of an essential local variable.

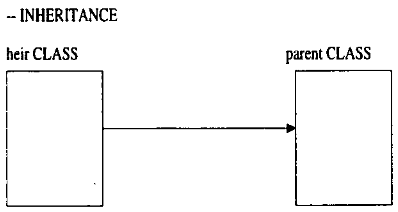

Figure A.5. The *inheritance* relationship shows the base class(es) from which a class is derived.

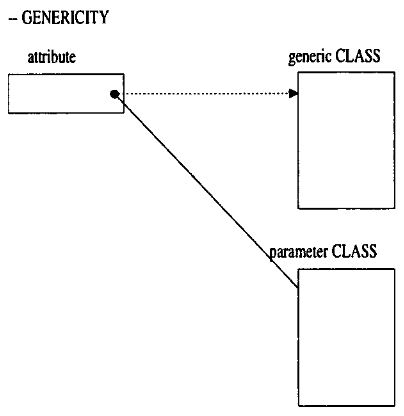

Figure A.6. Representation of *genericity*. Genericity is used to create parameterized types. This involves a generic class (or template) and a parameter class.

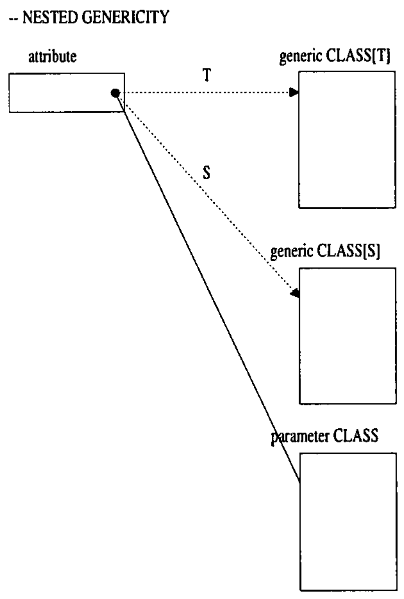

Figure A.7. Representation of *nested genericity*. Nested genericity is used to create parameterized types of a parameterized type. This involves a generic class (or template) taking a generic class (or template) as a parameter class.

Appendix B

Used class interfaces

This Appendix describes the class interfaces of standard Eiffel library types that were used for the creation of generic types, viz. the ARRAY, the LIST, and the TABLE.

B.1 Array

```
class ARRAY[T].
export count, empty, item, put
feature

     count:  INTEGER is
          do
               result :=  - - number of indices in array
          end;

     empty:  BOOLEAN is
          do
               if  - - array is empty
               then
                    result := TRUE
               else
                    result := FALSE
               end
          end;

     item( i:  INTEGER ):T is
```

```
          do
               result :=  -- item at index i
          end;

     put( item:T; i:  INTEGER ) is
           -- insert item at index i
end -- ARRAY
```

B.2 List

```
class LIST[T].
export count, empty, item, put remove
feature

     count:  INTEGER is
          do
               result :=  -- number of items in list
          end;

     empty:  BOOLEAN is
          do
               if  -- list is empty
               then
                    result := TRUE
               else
                    result := FALSE
               end
          end;

     item:  T is
          do
               result :=  -- item at cursor position
          end;

     put( item:T ) is
           -- insert item at cursor position

     remove is
           -- remove item at cursor position
end -- LIST
```

B.3 Table

```
class TABLE[T→ANY, U→HASHABLE].
export entry, item, has, put, remove
feature

      entry( i:  INTEGER ): T is
            do
                  result :=  - - the i-th entry in table
            end;

      has( key:  U ): BOOLEAN is
            do
                  if  - - key in use
                  then
                        result := TRUE
                  else
                        result := FALSE
                  end
            end;

      item( key:  U ): T is
            do
                  result :=  - - item associated with key
            end;

      put( item:T; key:  U ) is
             - - insert item with key

      remove( key:  U ) is
             - - remove item associated with key
end - - TABLE
```

Appendix C

Software design

In this Appendix the software architecture of the static model (EPEP), and also the dynamic model (MPEP) is described.

C.1 The static model

C.1.1 EPEP

```
class EPEP
global library: LIST[UNIT];
feature

    system_unit: UNIT;
    editor: SOURCE;

    Create is
        local editor_program: PROGRAM;
        do
            library.put( system_unit );
            editor_program.Create( editor );
            editor_program.interpret
        end;
end -- EPEP
```

C.1.2 Source

```
class SOURCE
export code, name, layout
```

```
feature
     code:  ARRAY[CODE];
     names:  TABLE[NAME, TAG];
     layout:   - - line_table, index_table, indent_table, remark_table
end - - SOURCE
```

C.1.3 Unit

```
class UNIT
export find_name
feature
     cells:  TABLE [NAME, CELL];

     find_name( name:  NAME ):CELL is
          do
               result := cells.item( name )
          end;
end - - UNIT
```

C.1.4 Statement

```
deferred class STATEMENT
export syntax_check, execute
feature

     syntax_check is deferred end;

     execute is deferred end;
end - - STATEMENT
```

C.1.4.1 Simple statement

```
class SIMPLE_STATEMENT
inherit STATEMENT
export syntax_check, execute
feature

     syntax_check is
          do
          end;
     execute is deferred end;
end - - SIMPLE_STATEMENT
```

C.1.4.2 Structured statement

```
deferred class STRUCTURED_STATEMENT
inherit STATEMENT
export syntax_check, execute
feature

     syntax_check is deferred end;

     execute is deferred end;
end - - STRUCTURED_STATEMENT
```

C.1.5 Block

```
class BLOCK
inherit STRUCTURED_STATEMENT, UNIT
export syntax_check, execute, find_name
feature

     code: ARRAY [TOKEN];
     names: TABLE [TAG, NAME];
     cells: TABLE [NAME, CELL];

     Create( block: BLOCK ) is
          do
               cells := block.cells;
               code := block.code;
               names := block.names
          end;

     get_declaration:DECLARATION is
          local declaration: DECLARATION;
          do
               - - parse code
               declaration := - - create specific declaration
               result := declaration
          end;

     get_statement:STATEMENT is
```

```
        local statement:  STATEMENT;
        do
             -- parse code
            statement :=  -- create specific statement
            result := statement
        end;
    syntax_check is
        local declaration:  DECLARATION;
        statement:  STATEMENT;
        do
            declaration := get_declaration;
            from
            until not declaration
            do
                -- put declaration in names and cells
            end;
            statement := get_statement;
            from
            until not statement
            do
                statement.syntax_check
            end -- empty cells except for static declarations
        end;
    execute is
        local declaration:  declaration;
        statement:  statement;
        do
            declaration := get_declaration;
            from
            until not declaration
            do
                -- put declaration in names and cells
            end;
            statement := get_statement;
            from
            until not statement;
            do
                statement.execute
            end -- empty cells
        end;
end -- BLOCK
```

C.1.6 Program

```
class PROGRAM
inherit BLOCK
export interpret
feature

    Create( source:  SOURCE ) is
        do
            code := source.code;
            names := source.names
        end;

    interpret is
        do
            syntax_check;
            execute
        end;
end -- PROGRAM
```

C.1.7 Empty statement

```
class EMPTY_STATEMENT
inherit SIMPLE_STATEMENT
export syntax_check, execute
feature

    execute is
        do
        end;
end -- EMPTY_STATEMENT
```

C.1.8 Assignment statement

```
class ASSIGNMENT
inherit SIMPLE_STATEMENT
feature
```

```
     variable: VARIABLE_CELL;
     expression: EXPRESSION;

     execute is
          do
               variable.setvalue( expression.value )
          end;
end - - ASSIGNMENT
```

C.1.9 If statement

```
class IF
inherit STRUCTURED_STATEMENT
feature

     condition: BOOLEAN;
     true: BLOCK;
     false: BLOCK;

     syntax_check is
          do
               true.syntax_check;
               false.syntax_check
          end;

     execute is
          do
               if condition then
                    true.execute
               else
                    false.execute
               end
          end;
end - - IF
```

C.2 The dynamic model

C.2.1 Process

```
class PROCESS
export myself, start, abort
```

```
global library: LIST[UNIT];
feature

     myself: [KERNEL_]PROCESS;
     childs: LIST[[KERNEL_]PROCESS];
     block: BLOCK;

     Create is
          local cell: PROCESS_CELL;
          do
               cell.Create( process_size + unit_chain_size );
               library.put( cell );
               if syntax then
                    block.syntax_check;
               else
                     - - copy unit chain
               end
          end;

     abort is
          do
               if  - - process was started
               then
                    myself.abort;
                     - - abort all childs
                     - - cleanup
               end
          end;

     start is
          do
               if not syntax then
                    myself.Create( block.execute, ... );
                     - - put patch in effect
                     - - initialize PEP process
                    myself.start
               end
          end;
end - - PROCESS
```

C.2.2 MPEP

```
class MPEP
inherit PROCESS
feature

    system_unit: UNIT;
    editor: SOURCE;

    Create is
        local editor_program: PROGRAM;
        do
            library.put( system_unit );
            editor_program.Create( editor );
            editor_program.interpret
        end;
end - - MPEP
```

C.2.3 Remote procedure

```
class REMOTE_PROCEDURE
inherit PROCESS
feature

    mailbox: MAILBOX;

    Create is
        do
            process.Create;
            if not syntax then
                myself.Create( loop, ...);
                myself.start
            end;
            mailbox.Create
        end;

    loop is
        local client: RESPONDER_PORT;
        parameters: MESSAGE;
        function_result: MESSAGE;
        do
            port.connect( mailbox );
            from
            until FALSE
            do
                parameters := client.get;
                 - - put parameters on stack
                block.execute;
                function_result :=  - - result cell
                client.reply( function_result );
                 - - get parameters from stack
            end;
        end;
end - - REMOTE_PROCEDURE
```

C.2.4 Signal

```
class SIGNAL
feature

    semaphore:  [KERNEL_]SEMAPHORE;

    Create( phybus_int:  INTEGER ) is
        do
            if phybus_interrupt then
                semaphore := get_semaphore( phybus_int )
            else
                semaphore := create_semaphore( 1 )
            end
        end;

    send is
        do
            semaphore.signal
        end;

    wait is
        do
            semaphore.wait
        end;
end - - SIGNAL
```

Appendix D

Implementation aspects

D.1 Introduction

The first implementation of EPEP dates from the late 1970s. It was based on the Digital PDP 11 and LSI 11 systems, and was written in assembly language. In the 1980s, a new implementation was written for the M68000 microprocessor, still in assembly language. With the M68000, the increase in enabled the introduction of new features, viz. multitasking, and later, multiuser facilities. The M68000 version was designed to operate directly on the M68000 hardware, and therefore it had to provide its own operating system and multitasking kernel. It still defines the standard EPEP implementation (standard-EPEP).

In the early 1990s, a start was made to reimplement EPEP in the C programming language. This would enhance portability and simplify, if not allow, 'software maintenance'. The first goal was to implement the static part of EPEP, i.e. without multiprocessing or multiuser aspects, which would follow later. This static C version of EPEP has now the state of a stable beta release and is approaching completion. It is commonly referred to as CPEP.

The reimplementation process of EPEP was divided into two stages, (i) the static stage, or (static) CPEP, and (ii) the dynamic stage, or multitasking/multiprocessor MPEP (which, of course, is also implemented in the C programming language). The second stage of implementation is performed after the model described in Chapter 3.

D.2 Portability

Although MPEP was primarily intended to be designed uniquely for the multiprocessor EMPS platform, target platforms now potentially include any multitasking operating system (MOS) platform. Apart from this, it proves to be quite convenient to have large programs, such as MPEP, run on different platforms. The static part (CPEP) was developed on other platforms than the target EMPS platform. Practical reasons for this were, besides cost aspects, the lacking of a C compiler and operational debugger for the EMPS platform, and the very time consuming data transport to the EMPS platform.

D.2.1 Software aspects

As mentioned above, the assembler version of EPEP provides its own operating system. If reimplemention of EPEP in the C programming language would has to support portability, all low-level operating system routines must be substituted by standard C language primitives.

Static CPEP should be portable to any C platform. MPEP however, expects the operating system to provide, besides a number of static C primitives, basic multitasking primitives. These can be divided into memory management (MM), time management (TM), process management (PM), process synchronization (PS), interrupt handling (INTH), and interprocess communication (IPC). If MPEP is run on a single-processor MOS platform, this does not imply loss of functionality, as a multiprocessor MPEP application consists of multiple stand-alone modules that communicate with eachother (see Chapter 3). The practical use for multiple module MPEP applications on single-processor platforms, is however, questionable. Only a reduced version of MPEP, may be ported to static (or single-tasking) platforms, presenting the functionality of static CPEP.

For full lists of static and dynamic operating system requirements see Appendix F.

D.2.2 Hardware aspects

From a hardware point of view, CPEP is not very portable. This had to be expected because EPEP as well as CPEP were developed for real time, multitasking experiment control and data-acquisition using the PhyDAS hardware. The EMPS hardware makes use of two computer busses, viz.

a VME bus and a VSB bus, and a VME/PhyBUS converter to control the PhyDAS hardware. A computer module that is VME compatible can therefore be used to control the PhyDAS hardware. The real time aspect will remain an issue open for discussion.

D.3 The implementation of static CPEP

CPEP is written in ANSI C, without using any C++ features. As stated above, CPEP is in the beta release state. The platform on which it is being developed and tested, is an INTEL 386+ using the the WATCOM C++ compiler. The WATCOM C++ compiler offers the programmer a linear memory addressing mode (LAM) on one hand, and on the other hand a library of operating system related routines, based on MS-DOS system calls, that is compatible in functionality with the TURBO C++ library. In addition, the WATCOM C++ compiler package contains an extensive debugger with, e.g., the possibility of protected mode debugging. To avoid confusion, this 'host' platform of CPEP, will from here on be called MS-DOS LAM. The normal MS-DOS platform will referred to as plain MS-DOS.

D.3.1 The EMPS platform

One of the most interesting systems to port CPEP to, is the EMPS multiprocessor system. Although the EMPS system is also based on the M68000 microprocessor family, it would be quite an understatement classifying this operation as 'software maintenance'. For a start, all operating system aspects must be handled by the EMPS kernel.

The CPEP source code is compiled on a remote system, and sent to the EMPS system using the Motorola S-format. The remote system is a SUN Sparc-solaris station, the compiler used is the Oasys gc68000 x-compiler [1]. The Oasys x-compiler provides a full ANSI C library. However, low-level operating system routines, e.g., for file I/O and system time, should be supplied/modified by the user to satisfy the conditions of the target system's (in this case the EMPS) multitasking operating system.

For the programmer's convenience, the EMPS kernel provides a library that implements all these low-level, and even some higher-level, routines. To be able to access these routines, the CPEP object code must be linked with a small assembler file, that provides the necessary interface to the EMPS kernel primitives.

There have been quite some difficulties before CPEP did run on the EMPS platform. The EMPS kernel interface was completed and adapted for more standardized use. The first test runs showed that the interpreter seemed to work, interpreting at least a few thousand lines of EPEP code, successfully loading two subsequently interpreted EPEP code files from disk. It then failed, however, to recognize the correctness of the syntax of the Program Editor (ED1.CPC), flagging the error 'identifier declared before'.

Although, when familiar with CPEP, it has a quite clear internal structure, the process of debugging and getting acquainted to the program began to take too much time.

The over 25.000 lines of C source code, are not very self-documenting, which is mostly due to the use of cryptic names and abbreviations, and are only documented on routine level, if documented at all.

Oasys failed to deliver a working x-debugger for the EMPS system and data transport was very slow. The problem was postponed, also because of promising test runs on a UNIX platform. After the successful port of CPEP to a UNIX platform, that is discussed in the following section, simultaneous debugging on the UNIX and the EMPS platform ultimately resulted in a working CPEP version on the EMPS platform.

D.3.2 The UNIX platform

The description given in the previous section, i.e. invoking a cross-compiler for the CPEP C sources on a MOS platform (in this case a SUN Sparc-solaris station) to try the portability of CPEP to another MOS platform (the EMPS platform) may seem a bit taking two steps at a time. The reason for trying the port to the EMPS platform first, was that there had no C compiler been installed on the SUN Sparc station.

After CPEP failed to perform properly on the EMPS platform, it was successfully ported to ULTRIX on a Digital workstation, DEC 2100. CPEP was compiled using the GNU C compiler (gcc). To allow CPEP to run on this UNIX platform, several operating system interface calls had to be implemented. These include explicit file path conversion, and two simple routines that handle character I/O, in relation with the control of terminal settings, such as echoing, translation of carriage return, line feed and break (Control C). Apart from the GNU compiler being somewhat fussier about ANSI C type conversions, the CPEP sources compiled smoothly. Features that imply user directory searching were not implemented.

D.3.3 The plain MS-DOS platform

As the host platform (MS-DOS LAM) already is a MS-DOS machine, but provides the feature of linear addressing and so wiping away all segment-and-offset troubles, including the 64kB data limit, CPEP should be made to run under MS-DOS without major problems. For testing, CPEP may be compiled with TURBO C++ (versions 1.0, 3.0), which works as long as the EPEP workspace is kept unacceptably small. No attempts were made to introduce huge pointers for the EPEP workspace, as this platform has not very high priority, if priority at all.

D.4 The implementation of MPEP

Although EPEP has been multitasking for quite some years, multitasking had not yet been introduced to CPEP. Here 'introduced' is not an euphemism for 'implemented'; it is probably not favorable to have multitasking fully embedded in CPEP. In the assembler version, EPEP handled multitasking aspects such as time slicing, scheduling, and process synchronization itself.

There are three ways of introducing multitasking CPEP:

i. Writing an in CPEP embedded kernel, that takes care of all multitasking aspects, ignoring any multitasking services provided by the operating system,

ii. Writing an in CPEP embedded kernel, but using operating system services if available,

iii. Relying only on operating system services, possibly polished-up by supplementary routines in CPEP. Whenever the operating system fails to provide a needed service, it should be provided for by writing an external (i.e. not in CPEP embedded) driver.

The first option had been chosen in the M68000 assembler implementation, led by the fact that this was the only target platform for EPEP, and EPEP would be the only application to run on that platform. As MPEP may run on different platforms in the future, the third option seems the only one to achieve a somewhat universal tackling of the multitasking aspects.

One of the new platforms will be the EMPS multiprocessor system, that offers a complete set of multitasking facilities. In the future, more and more platforms will offer some kind of multitasking services, e.g., UNIX/LINUX, OS/2, WINDOWS-NT/-95. Should no multitasking services be provided, the system will probably be non-complicated, so that these services may relatively easy be implemented as an extension of the operating system, see section D.4.6.

For further discussion of the implementation aspects of multitasking, it is necessary to introduce a few definitions. The definitions suggested in [12] are used, which follow in section D.4.1.

D.4.1 Tasks and processes in the EMPS system

Although the EMPS system is a multiprocessor system, reaching beyond merely multitasking systems, it must in the first place handle all multitasking aspects. It is a transparent system, which means that any description and any mode of operation will hold for a true multiprocessor configuration as well as for a single-processor configuration. The definitions concerning multitasking given in [12] are therefore perfectly suitable to describe single-processor multitasking systems as well.

A *task* is defined as a set of class definitions and a set of *process* definitions. A set of class definitions stands for the routines, data, and heap in a function-oriented programming environment. Inside one task, multiple processes may therefore share the same address space for code, data and heap, the *task address space*. A process is sometimes referred to as a *thread* in other systems.

Each process has a private address space for its stack, the *process address space*. So each task has private memory in the form of its task address space, which only is truly private if the task contains just one process definition. Private to a process are its current CPU register values and its stack; in general a process does not have additional private memory. Processes that want to have any true private memory, that is, apart from their stack, must therefore be separate tasks and hence cannot share code or heap.

D.4.2 The CPEP C sources

The static part of EPEP is almost completely covered by the *de facto* CPEP source code. No multitasking aspects were implemented, but CPEP was said to be 'prepared for multitasking', partly because the implementation had the the multitasking M68000 assembler version of EPEP as an example.

The Interpreter works with a data structure that contains all process related information about the currently interpreted code. This *main structure* (mn_str) contains data items as the code table, and location pointer, name table, cell table, cell heap, etc. Whenever a process is created in EPEP, a new main structure is set up, so for each EPEP process such a main structure exists. Putting these main structures in a ready list, the Interpreter was supposed to perform a context switch by simply getting the next main structure in that list. The pointer *Current Process* (mn_str*) should always point to the main structure of the EPEP process that is being interpreted.

The above description fits the M68000 assembler version as well as the 'multitasking preparation' of the CPEP version. There are, however, two differences between the two implementations:

i. In the assembler implementation, the pointer Current Process is always stored in the CPU register A5 [16], whereas in the CPEP implementation Current Process is a global pointer value, which implies that its access scope has changed, i.e., from private (to a process) to shared,

ii. The assembler implementation would run on the M68000 system, and use an embedded process scheduler, whereas the MPEP implementation will use operating system process scheduling facilities (see introduction of section D.4). This implies that, where an EPEP process could previously be handled by the embedded scheduler as a special entity with its own characteristics, in the MPEP implementation an EPEP process must meet the conditions of an operating system process; the operating system scheduler cannot make any distinctions between EPEP- and non-EPEP processes.

From the above observations it can be concluded that the *de facto* CPEP sources required a process to have both private memory, to store the pointer Current Process, and shared memory, where the (shared) cell tables, code tables and name tables must be stored.

As will be shown in the following sections, these are most unusual, if not conflicting, demands on most platforms.

D.4.3 The host MS-DOS LAM platform

On the host platform no attempts to introduce multitasking were made. Foreseen problems and solutions are expected to be quite similar to those discussed for the plain MS-DOS platform, section D.4.6.

D.4.4 The EMPS platform

Implementation of multitasking on this platform must deal with the fact that an EMPS process cannot have both private memory (apart from the stack), and shared memory (see section D.4.1 and [12]). The first alternative is that all EPEP processes run within one task, and share all data address space, including the pointer Current Process. However, this pointer should be private to a process, because for it should always point to the process' main structure. This solution therefore precludes the implementation of multitasking MPEP. As a second alternative, each EPEP process is a complete task, so that the creation of a new EPEP process involves copying all code and data, which in turn makes it impossible for EPEP processes to access each other's global variables. This is in conflict with the EPEP Application Language definition [4].

Of three possible solutions, (i) providing the operating system (EMPS kernel) with additional knowledge about the peculiarities of EPEP processes, (ii) providing the operating system (EMPS kernel) with facilities to let tasks share memory, and (iii) changing the access scope of the pointer Current Process back from shared to private, only the third option was considered (The first option is unacceptable, whereas the second option would still imply copying all code and data for each new EPEP process, which hardly seems to be an elegant solution).

Rejecting the options (i) and (ii) above, the access scope of a variable, in particular the access scope of Current Process, could be made private in two ways:

i. Exclusive storage in one of the CPU registers,

ii. Storage on stack.

D.4.4.1 The blunt C solution

The ANSI C standard does not support global register variables; the storage specifier *register* is only available for variables declared in a block, and for formal arguments [2]. This leaves only the possibility of storage on stack. The regular way to achieve this in the C programming language, is by passing Current Process as a parameter (the first) to all routines of the Interpreter. This would imply the changing of all formal parameter lists, as well as all actual parameter lists; an enormous operation.

D.4.4.2 The elegant C++ solution

If there were plans for future implementation of CPEP in the C++ programming language (which should be considered as a possibility, as on all target platforms already hybrid C/C++ compilers are used), the object-oriented nature of the language could elegantly cope with the problem of the global pointer Current Process. Current Process would be made a *class member* of the class INTERPRETER. As such it would be private to a process, and at the same time available to all member functions of the INTERPRETER. In this way, no formal parameter lists, nor any actual parameter lists would have to be changed to achieve the passing of Current Process between INTERPRETER routines.

A change from C to C++ will also be quite an operation. Although C is a subset of C++, C++ is more than ‘C with classes’; it is a *different* language. Whereas most C code will remain usable without change, and hybrid code would do fine for an intermediate phase, a true C++ implementation involves library design and class modelling.

Incidentally, in the object-oriented paradigm that was adopted to model dynamic MPEP, these kind of access scope problems can be avoided in an elegant way. The class concept allows the routines of a class to share private class data. The programmer does not need to pass the private data explicitly over the stack. This makes the object-oriented approach especially suitable for dealing with multitasking environments (see Chapter 3 and [10]).

D.4.4.3 The patch

So far, no decision for a specific solution was made. Because all options mentioned above involve too much additional work, the problem has been patched, temporarily. The pointer Current Process is declared with an extra level of indirection, i.e., it is now a pointer to a pointer to a main structure (mn_str**).

The basic reason for this patch to work, lies in the fact that the EMPS kernel uses virtual memory addressing, and memory mapping for kernel processes. Thus, it proved to be possible, using M68000 assembler statements, to cheat the effective Current Process pointer value onto the stack, making it private to processes after all. Because this patch is clearly not portable, and should as such only be used temporarily, for testing, it will not be explained in detail. The patch is documented quite thoroughly in the MPEP C sources.

D.4.4.4 Practical EMPS application problems

MPEP would be the first major application for the EMPS platform. Therefore, apart from the above 'EPEP related problems', the EMPS platform still presented some other development areas: e.g., performance, utilities, debugging, and how and when to redirect terminal I/O. Because an MPEP configuration with a Program Editor cannot operate without user input (as can hardly any serious application), a UNIX-like solution for the redirection of terminal I/O was chosen and implemented.

EMPS applications can now be loaded and started by simply entering the name of the executable (without extension) in the EMPS kernel command interpreter. If an application is started in this way, the EMPS kernel will redirect all terminal I/O to this application. Once the application is started, the user can switch (redirect) his (m/f) terminal I/O to any process that requests user input. by typing Control Z. One of these processes, of course, is the command interpreter. When the process associated with terminal I/O is stopped or killed, terminal I/O is redirected to the process that is awaiting user input for the longest time.

D.4.5 The UNIX platform

On the UNIX platform (see section D.3.2), the first attempt to the implementation of multitasking was made.

The CPEP sources were expanded with two modules (about 2500 lines): one to handle the declaration and starting of processes, and another implementing the semaphore interface.

The main structure was expanded with four additional fields: the parent's main structure, the child's main structure, a location pointer to the (optional) expression of the requested process size and (once calculated) the requested process size. In this preliminary code

- each process can only have one child, and
- the only operation on processes that is available is the *start* routine.

According to the definitions give above, the specific UNIX platform does not support processes; it only supports tasks (see section D.4.1). On the UNIX platform, child tasks are created with the system call *fork*() [11]. A fork() is commonly explained as a routine that is invoked once, but returns twice, the parent task with a nonzero return value (1 on success, -1 on failure) and the child task with zero return value. Because not only a new process (or thread) is created, but in fact an entire new task, upon a fork() all data, static and heap, is copied (Depending on the specific implementation, UNIX might copy all program code as well).

Contrary to the EMPS platform, under UNIX the problem that occurs when assigning a complete task to each new EPEP process, can be solved. A task can apply for *shared memory* by issuing a UNIX system call. The cell tables, code tables, and name tables, etc. are stored in this shared memory, so that EPEP child processes can indeed use global EPEP variables.

After fork()ing, the child task must issue another system call to get access to the shared memory segment.

Interprocess communication (IPC) including the remote procedure call (see section 3.1.3), was implemented using the UNIX message primitives.

D.4.6 The MS-DOS platform

On the MS-DOS platform the first real multitasking version of MPEP was implemented. The implementation on the UNIX platform of a simple test of true multitasking MPEP seemed to be cumbersome, and no expertise on this subject was available nearby. Because even static CPEP did not yet operate on the EMPS system, the MS-DOS platform was chosen to give a quick answer to the question whether multitasking could be easily introduced to CPEP, or any major changes had to be made.

Following alternative iii given in section D.4, a small stand-alone kernel (about 5000 lines) was implemented, that must be loaded before EPEP is started. It consists of a timer, a simple scheduler, and a user interface, and provides means of process synchronization. This kernel is a fully object-oriented, almost literal implementation of [12] (as far as the small implementation goes). It was written in the C++ programming language (and small parts in assembler).

The scheduler maintains four ready lists, one for each EPEP priority, and performs time slicing by executing a context switch each 18.2 ms, i.e., each clock tick. Communication with the scheduler is handled by the class PROCESS.

A kernel interface was designed for the communication between user and scheduler. A request for a certain service may be placed by generating a MS-DOS software interrupt with the appropriate service number in the AH register (a CP/M / MS-DOS convention). This kernel interface provides services to create, block, and kill a process to add or remove it from the ready list, to obtain a process's id (pid) or a process's parent's id (ppid), and to preempt the current process.

To allow the expanded CPEP sources to offer real multitasking, the scope of the pointer Current Process has to be be forced from shared to private. The kernel achieves this by supporting the feature of *virtual private memory*. A process may designate a certain physical memory area as private memory. The kernel ensures that a process will always find the contents of this private memory area just as when the process left it. Thus, EPEP processes use four bytes of virtual private memory; the memory locations in which the pointer Current Process is stored.

With the aid of this simple MS-DOS kernel, a simple multitasking EPEP application program was successfully executed. The program consisted of a process declaration and a main loop. Both program blocks contain a loop that prints an alive message and then issues a delay. After the main program block has start(...)ed the child process, both processes keep printing their alive messages as expected.

D.5 An integrated environment for EPEP

As mentioned in Chapter 4 an attempt was made to present an example of a more efficient and user-friendly way to enter, edit and check EPEP application programs.

As an example, an integrated desktop environment (IDE), for EPEP program editing and EPEP program execution was implemented (6000 lines). It uses the object-oriented Borland Turbo Vision text-screen based window library for the MS-DOS platform. This EPEP IDE was named XPEP.

XPEP provides a full-screen text editor, a turn-key interface to the CPEP interpreter, online (context) help and an automatic indentation feature.

Appendix E

Future work

Most urgent future work involves standardization of CPEP sources and EMPS kernel sources. This is due to the facts that CPEP is still under development while the additional MPEP sources are not yet incorporated into the CPEP sources EPEP is the first serious EMPS application, and the EMPS kernel is still being reorganized and customized.

Although two or more programmers work on the project simultaneously, no means of version control is used; that is why source code must be manually merged.

E.1 The EMPS kernel

The EMPS kernel has several areas open for improvement, here only the urgent CPEP related problems are summarized.

- Standardization of header files of the EMPS kernel (and CPEP). Temporarily, parts of EMPS kernel header files are copied and used by MPEP because both cannot yet be included.

- Standardization of EMPS kernel services, functionality and names.

- Offering of kernel services using standard headers and library file.

- Full use and support of customizable ANSI C libraries for EMPS applications, e.g., **getchar** instead of **read(1, &c, 1)**. A number vital ANSI C functions for EPEP were implemented in customized EMPS library files.

- Control-C detection and support of user defined Control-C handlers. The implementation of UNIX **signals** might be a good idea.
- Providing of **delete_semaphore** routine.
- Providing of **semaphore_inspect** routine.
- Fixing **term_proc**, and offering of functionality under name **exit**.
- Provide primitives to turn the time-slicing mechanism on/off. EMPS kernel processes run at software level 0. A lock could therefore be performed by a simple piece of assembly code **mov 2700, sr**.
- Real time clock.
- Command interpreter:
  - progress indicator
  - kill process (and other operations) by name e.g. **kill cpep** instead of **kill 2000015**.

E.2 CPEP

Static CPEP is still developing further. There are, however, some important tasks that should be handled before CPEP and MPEP can be merged. These are:

- Standardization of header files of CPEP and (EMPS kernel), see previous section.
- Choice for and implementation of one of the options discussed in section D.4.4, to allow a sound basis for MPEP processes, rather than the currently used patch.
- When major changes in the CPEP sources should be made (see previous item), it would be wise to consider the development and implementation of a naming convention for CPEP sources. In addition, the use of good and explicit names, instead of single letter variables and cryptic abbreviations would make CPEP a lot easier to read, debug and develop.

E.2.1 CPEP naming convention

The following sections on the CPEP naming convention, could be considered to stand somewhat besides the scope of this thesis. They are presented here because (i) CPEP seems in want for such a discussion and a lot of interest on the subject was expressed, and (ii) in this Appendix all other suggestions for future work and changes to CPEP are summarized. Every programmer that is working (or going to work) on CPEP will be faced with this matter. Seen from this viewpoint this Appendix seems to be an appropriate place to discuss the CPEP naming convention. Aspects of object-oriented naming are also considered.

E.2.1.1 User defined types

CPEP defines and uses integer types such as `BYTE`, `SBYTE`, `WORD`, `SWORD`, instead of using the predefined c-types `(unsigned) char`, `int`. The reasons for this are obscure. There are, of course, some specific variables that should have a defined size. These include, e.g., the location pointer (`unsigned char*`) and the fields of cells. User defined types are not only used for specific CPEP tasks, but also strictly as parameter to ANSI C routines. Not always the corresponding user defined type is chosen. A most peculiar example is formed by the ANSI C string functions that take the type `char*`. Often the type `BYTE* (unsigned char*)` is used instead of `char*`.

E.2.1.2 Cryptic names

Similar to most conventional C programs, CPEP tends to use cryptic names for structures, variables and fields where possible. This may save the programmer some key-strokes, but it makes the program much harder to understand for an outsider or newcomer. It is a known fact that variable names that based upon some kind of abbreviation, often only seem logical to the programmer of the code himself (abbreviations may not seem so logical anymore when one is confronted with scarcely documented code that was written some time (say two years) ago—did I write this?). The use of good names really is harder than it looks, but it makes the program much better readable.

E.2.1.3 Names of structures and fields

Most structures in CPEP are typedefed with a name of the form `XXX\_STR`. If instead the `struct` namespace would be used, a structure variable declaration could be identified by its prefix `struct`. The postfix `_STR` is made redundant and these four extra characters could be used instead for a clear name. In addition, if names of structure variables were chosen carefully, it would not be necessary to echo the name of the involved structure in its field names, as seems to be the convention in CPEP.

In CPEP we may find something similar to this:

```
typedef struct pu {
  ...
  BYTE* pu_nt;
  ...
} PU_STR;

{
  PU_STR* pu_ptr;

  pu_prt->pu_nt = nt;
}
```

Compare this to:

```
struct Unit {
    ...
    BYTE* nameTable;
    ....
};

{
    struct Unit* unit;

    unit->nameTable = nameTable;
}
```

In the second example, one can immediately understand what is happening. Note that the only differences between both examples result from the choice of names.

E.2.1.4 Suggestions for a naming convention

There are several naming conventions commonly used for writing C code. It is not so important which convention is used, because readability is to some extent a question of personal taste and habit, as long as there is a clear definition that is universally applied by all programmers that work on the project, so that all code has a similar appearance, and programmers do not have to choose whether to use capitals, underscores or both, every time.

With the advent of the object oriented programming techniques and C++, the naming problem easily gets out of hand. It often is convenient to have an object and an instance that are described best by the 'same' name (e.g. `String string`; where `String` is the object and `string` is an instance).

Borland defines a naming convention that handles all these problems in a simple way [3]:

- From the mere spelling (capitals and lowercase) of a name, one can tell if it a a class or structure, a constant, or a class member (or shared variable).

Borland spelling convention [3]

type	*spelling*	*example*
class, struct	first letter is capital only	`String`
generic class	concatenation of class names	`StringList`
class member	first word lowercase, subsequent words start with capital	`iCantReadThis`
constant, define	all letters uppercase, words joined with underscores	`LINE\_LENGTH`

- Only full names are used, except for obvious counters in simple loops e.g. `for ( int i = 0; i < MAXIMUM; i++ ){ ... }`.

- Pointer variables end with `Ptr`.

E.3 MPEP

The essential multitasking and multiprocessor primitives are implemented and tested. Several, most EPEP-related dynamic primitives however, remain to be implemented. This should be a rather straightforward job, because MPEP provides the necessary primitives on C source level. The main reason for leaving this task open is the fact that these primitives imply some simple but essential changes in CPEP, while MPEP and CPEP sources were not yet merged. Implementation would have meant double work.

- Implementation of (connecting of) the process related EPEP primitives **abort**, **ask_priority**, **error_cause**, **main**, **myself**, **set_priority**, **show_ec**.
- The lock/unlock mechanism. The EMPS kernel must offer primitives to turn the time-slicing mechanism on/off.

Appendix F

Kernel requirements

The requirements that MPEP imposes upon the operating system are presented in two parts, viz. static primitives and dynamic primitives. For a (dynamic) UNIX implementation the static C platform primitives are also required.

In a number of tables the names of UNIX / EMPS functions are given, with a brief description of their functionality.

For detailed descriptions of static (C) primitives see [2] or [11]. Detailed descriptions of dynamic UNIX primitives can be found in [11], EMPS primitives are described in [7].

F.1 Static primitives

F.1.1 I/O primitives

F.1.1.1 File primitives

Name	Platform	Description
close	C EMPS	close file
open	C EMPS	open file for reading or writing
read	C EMPS	read from file
unlink	C EMPS	delete file
rename	C EMPS	rename file
write	C EMPS	write to file

F.1.1.2 Stream primitives

Name	Platform	Description
fclose	C EMPS	close stream
fopen	C EMPS	open stream
fprintf	C EMPS	formatted output to stream
fread	C EMPS	read data from stream
fscanf	C EMPS	formatted input from stream
fseek	C EMPS	position file pointer of stream
fwrite	C EMPS	write to stream
perror	UNIX	system error messages
printf	C EMPS	formatted output to stdout
sprintf	C EMPS	formatted output to string
scanf	C EMPS	formatted input from stdin
sscanf	C EMPS	formatted input from string

F.1.2 Mathematical primitives

Name	Platform	Description
atan	C EMPS	arc tangent
atan2	C EMPS	arc tangent of y / x
atof	C EMPS	convert string to floating point
atol	C EMPS	convert string to long
cos	C EMPS	cosine
exp	C EMPS	calculate power of e
fabs	C EMPS	absolute value of floating point
floor	C EMPS	round down
log	C EMPS	logarithm ln(x)
log10	C EMPS	logarithm, base 10
sin	C EMPS	sine
sqrt	C EMPS	calculate square root

F.1.3 Memory management

Name	Platform	Description
AllocateMemory	EMPS	allocate memory
malloc	C EMPS	allocate memory
malloc	EMPS	allocate memory
malloc_type	EMPS	allocate memory
getrlimit	C	available memory

F.1.4 String operations

Name	Platform	Description
memcmp	C EMPS	compare two strings of n bytes
memcpy	C EMPS	copy string of n bytes
memset	C EMPS	set n bytes of string
strcpy	C EMPS	copy string
strlen	C EMPS	calculate length of string
strncmp	C EMPS	compare at most n characters of two strings
strncpy	C EMPS	copy at most n characters of string
tolower	C EMPS	translate character to lowercase
toupper	C EMPS	translate character to uppercase

F.1.5 Time management

Name	Platform	Description
clock	C EMPS	number of clock ticks since program start
localtime	C EMPS	convert date and time to tm structure
time	C EMPS	get time of day

F.1.6 Error handling

Name	Platform	Description
longjmp	C EMPS	perform nonlocal goto
setjmp	C EMPS	set up nonlocal goto

F.2 Dynamic primitives

The multitasking platforms currently supported, viz. the EMPS platfrom and the UNIX system V platform, offer similar dynamic operating system features, but with a different set of primitives. For future MPEP platforms, only the functionality of either one is required.

F.2.1 Interprocess communication

F.2.1.1 IPC creation

Name	Platform	Description
Connect	EMPS	connect port to mailbox
CreateMailBox	EMPS	create mailbox
msgctl	UNIX	message queue control

F.2.1.2 IPC deletion

Name	Platform	Description
RemoveMailBox	EMPS	remove mailbox
msgctl	UNIX	message queue control
msgget	UNIX	set up message queue

F.2.1.3 IPC operation

Name	Platform	Description
ReceiveFromPort	EMPS	get message from mailbox
msgrcv	UNIX	get message from message queue
SendToPort	EMPS	put message in mailbox
msgsnd	UNIX	put message in message queue

F.2.2 Process management

F.2.2.1 Process creation

Name	Platform	Description
CreateProcess	EMPS	set up new process
StartProcess	EMPS	start process
fork	UNIX	start new process

F.2.2.2 Process deletion

Name	Platform	Description
KillProcess	EMPS	terminate process
exit	EMPS	terminate process

F.2.2.3 Process identification

Name	Platform	Description
getpid	EMPS	get process identifier
getpid	UNIX	get process identifier

F.2.2.4 Process scheduling

Name	Platform	Description
delay_process	EMPS	block process for amount of time
sleep	UNIX	block process for amount of time

F.2.3 Process synchronizaton

F.2.3.1 Semaphore creation

Name	Platform	Description
GetSEM	EMPS	get PhyBUS semaphore
create_semaphore	EMPS	get new semaphore
semctl	UNIX	semaphore control
semget	UNIX	set up semaphore

F.2.3.2 Semaphore deletion

Name	Platform	Description
– –	EMPS	delete semaphore
semctl	UNIX	semaphore control

F.2.3.3 Semaphore operation

Name	Platform	Description
signal	EMPS	perform signal operation on semaphore
wait	EMPS	perform wait operation on semaphore
semop	UNIX	perform operation on semaphore

F.2.4 Shared memory

Name	Platform	Description
shmat	UNIX	map shared memory
shmctl	UNIX	shared memory control
shmdt	UNIX	delete shared memory

F.2.5 Signal handling

Signal handling is not supported by the EMPS kernel.

Name	Platform	Description
signal	UNIX	specify signal-handling actions
sigaction	UNIX	handle signal
system	UNIX	execute shell command

Bibliography

[1] Oasys / Green Hills 68K. "Cross developement guide, version 1.8.6". Prentice/Hall Int., 1993.

[2] Mark Williams Company. "ANSI C a lexical guide". Prentice/Hall Int., Englewood Cliffs, NJ, 1988.

[3] S. R. Davis. "Hands-on Turbo C++". Addison-Wesley, Amsterdam, 1991.

[4] W. M. Dijkstra. "PEP Programmeertaal voor de MicroGiant". Technical Report BL 87-02, Eindhoven University of Technology, April 1987.

[5] J. H. Emck, J. H. Voskamp, and A. J. van der Wal. "EPEP: An operating system designed for experiment-control". Technical report, Eindhoven University of Technology, 1985.

[6] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft. "Operating systems – advanced concepts". The Benjamin/Cummings Publishing Company, Inc., 1987.

[7] R. Marissen. "The real-time EMPS-kernel: Memory management and suitability for EPEP / PhyDAS". Eindhoven University of Technology, October 1994.

[8] Bertrand Meyer. "Object-oriented software construction". Prentice/Hall Int., 1988.

[9] B. Stroustrup. "The C++ programming language". Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[10] B. Stroustrup. "The design and evolution of C++". Addison-Wesley, Reading, Massachusetts, 1994.

[11] R. Thomas, L. R. Rogers, and J. L. Yates. "Advanced programmer's guide to UNIX System V". Osborne McGraw-Hill, Berkeley, California, 1986.

[12] G. J. W. van Dijk. "The design of the EMPS multiprocessor executive for distributed computing". PhD thesis, Eindhoven University of Technology, March 1993.

[13] O. van Roosmalen. "A hierarchical diagrammatic representation of class structures". To be published.

[14] P. W. E. Verhelst and N. F. Verster. "PEP: an interactive programming system with an algol-like programming language". *Software-Practice and Experience*, **14**(2), 119–133, February 1984.

[15] N. F. Verster. "Verschillen tussen EPEP en CPEP, deel 1". Discussiestuk Werkgroep FTL, October 1992.

[16] J. H. Voskamp. "An object-oriented approach in the software design for experimental physics". Internal report, 1987.

[17] J. H. Voskamp and A. J. van der Wal. "A database approach for the control of technical processes". Technical report, Eindhoven University of Technology, 1985.