GMD MODULA SYSTEM MOCKA
User Manual
Vincenz–Prießnitz–Straße 1, D–76131 Karlsruhe

Universität Karlsruhe
Institut für Programm- und Datenstrukturen
Vincenz-Prießnitz-Strasse 1
D-76131 Karlsruhe, Germany
Phone: *-49-721 / 608-6088
Fax: *-49-721 / 69-14-62
Contact:
Helmut Emmelmann, Jürgen Vollmer
Email: emmel@ira.uka.de / vollmer@ira.uka.de

March 28, 1995

Contents

1 Introduction

1.1 Pre-Preliminary

If you are not interested in the technical data and the special features of the MOCKA system prefering a direct approach to the system you should skip to Appendix A, in which a short sample session gives you a basic idea of how to use the system.

1.2 Preliminary

MOCKA (Modula Compiler Karlsruhe) is a complete, high performance implementation of the language Modula-2 as defined in [Wir85]. The MOCKA system consists of a compiler, a binder, and a simple environment (including a make-facility) supporting interactive program development. The compiler translates Modula-2 source programs into binary machine code or assembler code. For some systems (e.g. HP 9000) - instead of machine or assembly code - C code is produced, which is then passed on to a C compiler. The binder checks the object files generated by the compiler for consistency and links them to an executable program. All these components may be used separately in a command mode (e.g. in shell scripts), but also integrated under supervision of the interactive programming environment (in the so-called session mode). Additionally, this environment provides a mechanism similar to the Unix-make, but works interactively and makes the maintenance of a description file (makefile) superfluous.

1.3 Supported machines

MOCKA may be used on the following systems:

target machine	operating system	processor
DEC Station	ULTRIX	R3000, R2000 (MIPS)
Silicon Graphics	IRIX	R3000, R2000 (MIPS)
Sony NEWS	News	MC 68020 with 68881
SUN3	SUN OS	MC 68020 with 68881
SUN4	SUN OS	SPARC
SUN4	Solaris2.x/SunOS 5.0	SPARC
PCS Cadmus	UNIX	MC68020 with 68881
NeXT	Mach/Unix	MC68020 with 68881
HP9000/300	HPUX	MC68020 with 68881
HP9000/700	HPUX	using C backend
RS6000	AIX	using C backend
VAX	UNIX (BSD) / ULTRIX	VAX
Transputer	Without own O.S. but access via UNIX from a host computer	T800, T414
IBM PC	Linux	80386,80486
IBM PC	386BSD	80386,80486
C	UNIX	C

1.4 Compilation Speed

MOCKA compiles up to 33000 lines of source per minute (on a DEC 3100 station).

1.5 Optimizer

An optimizer is available which can be used to speed up the execution of the compiled program. For some systems the resulting code is up to 20% faster than the non-optimized. For the Intel 80386/486 the optimizer is not yet supported.

1.6 Debugger

For some target systems, the 'dbx' and 'gdb' debugger is supported. 'adb' may be used on all (non-VMS) systems.

1.7 Libraries

The Modula-2 source and binary files may be distributed into libraries. These libraries are mapped to the file system of the underlying operating system. A standard library is provided containing MathLib, (un)formatted and raw byte I/O utilities, storage management utilities, and access to the operating system.

1.8 Acknowledgements

MOCKA was designed and implemented by Friedrich-Wilhelm Schröer, Franz Engelmann, Dirk Schwarz-Hertzner, Helmut Emmelmann and Jürgen Vollmer at the former *GMD Forschungsstelle an der Universität Karlsruhe, German National Research Center for Computer Science.* MOCKA is now maintained at the University of Karlsruhe. This paper serves as a guide how to work with the MOCKA system.

2 MOCKA Reference Manual

2.1 The Compiler

There are three kinds of compilation units in Modula-2:

Definition Modules specify interfaces which may be imported by other modules. Files containing definition modules must end with the suffix ".md".

Implementation Modules provide the implementation of the corresponding interface. Files containing implementation modules must end with the suffix ".mi".

A Program Module constitutes the main program. A program module file must also end with the suffix ".mi".

Module name and file name prefix must be identical.
To compile the definition module contained in file *source.md*, invoke the compiler with the shell command (let '$' be the prompt of the operation system):

```
$ mc -s source [ options ]
```

If the definition module is free of errors, a symbol file containing the interface description of the module is created.
To compile an implementation module or a program module contained in file *source.mi*, invoke the compiler with the shell command:

```
$ mc -c source [ options ]
```

If the module does not contain any errors, an object file is created that contains the binary code of the module. Definition modules imported by a compilation unit must have been compiled previously. Their interface description is searched in the current directory, in the directories specified at system invocation and in the installation specific system library, in this order.
A definition module must be compiled before the corresponding implementation module. If a definition module is recompiled the corresponding implementation module and all units importing the module must be recompiled. The system checks consistency of configurations. In session mode, i.e. using the interactive environment, the system invokes recompilation automatically.
The compiler invocation may be parameterized with the following two classes of options:

- compiler pragmas
- options

Compiler pragmas and options are described in section 2.5 and section 2.4 below.

2.2 The Binder

The binder performs the modula-specific preparations for the system linker call. E.g., in Unix systems the system linker is *ld* . The binder collects the transitive closure of the imported modules, provides a consistency check on the object files of the modules that together constitute the program, generates an initialization routine that is executed when the program is loaded (to initialize the module bodies), and invokes the system linker (with a complete argument list).
To link the object modules of the program module *program* , invoke the system with

```
$ mc -p program [option]
```

If the object files of the program module and of the (transitive closure of the) modules imported by the program module were found and consistent, an executable program named *program* is generated.

2.3 Interactive Environment and Make Facility

The MOCKA system components may be used separately in the command mode as described above, or integrated in an interactive session mode under supervision of a special programming environment. The interactive environment reduces the number of commands to be typed in, and provides a make facility to support the development and maintenance of large programs consisting of many compilation units. So, if the MOCKA system is used in session mode, the user needs not worry about the correct compilation order.
To ease the first steps with the MOCKA system, see for Appendix A. Appendix B describes a sample session where the capabilities of the MOCKA system are used in a more sophisticated way.
To invoke the system in session mode, enter the shell command

`$ mc [` *`options`* `]`

Again, the options available are of two classes:

- compiler pragmas
- options

Compiler pragmas and options are described in section 2.5 and section 2.4 respectively.
In session mode, the user can enter commands to edit or compile compilation units, to specify goals that the system tries to reach automatically, to overwrite options given at system invocation, or to execute Unix commands. Goals are such as "generate an executable program for program module xy". In this case the user needs neither enter commands by hand to compile the program module and each module (both definition and implementation) imported by the program module (in an correct order) nor to bind the objects. All necessary actions to reach the specified goal are performed by the MOCKA system.

The commands available in session mode are:

- `d` *module*
- `i` *module*
- `s` *module*
- `c` *module*
- `p` *module*
- `<empty>`
- *-pragma*
- `q`
- *unixcommand*

To edit the definition module *module* stored in source file *module*.md enter the `d` command. To edit the implementation/program module *module* stored in source file *module*.mi enter the `i` command. If the name *module* is omitted, the name used in the previous `d` or `i` command is taken.
The user can specify goals with the `s, c` or `p` command. The system tries to reach this goal automatically by compiling (and/or recompiling) modules. The correct compilation order is computed by the system itself, the user needs not specify the dependencies between compilation units in a description file (makefile).
If an error is detected during compilation of a module a listing is generated and the editor is invoked automatically. Errors may be corrected in the listing. After leaving the editor the system writes the listing back to the source file (without the error messages).
Give the `s` command to specify the goal: "compile and generate a symbol file for the definition module *module*. Create symbol files for all (transitively) imported modules if they are missing or obsolete".

Give the `c` command to specify the goal: "compile and generate a code file for the implementation/program module *module* and symbol files for all modules imported (transitively) by *module*, if they are missing or obsolete".
Give the `p` command to specify the goal: "compile and generate a code file for the program module *module* and code and symbol files for all modules imported (transitively) by *module*, if they are missing or obsolete. Create executable program *module* if missing or obsolete".
Give the empty command <empty> to continue processing after editing a file. The goal is that specified in the previous `s`, `c` or `p` command.
For the use of pragmas see section 2.5.
Give the `q` command to leave the system.
All other commands entered in session mode are assumed to be Unix commands and passed to the Unix shell.

2.4 System Installation Parameters

The MOCKA system can be installed in various configurations depending on the actual installation parameters given. A default installation (including default values for the options) is given in the shell script `mc` that is used to invoke the system. Installation parameters may also be given as options at system invocation time in order to overwrite the default values.
Installation parameters available are:

`-d` *directory*

`-D` *directory*

`-link` *script*

`-edit` *script*

`-asm / -cvt` *script*

`-cc` *script*

`-O`

`-list` *script*

`-syslib` *directory*

Libraries are directories containing compiled modules. If the module to be compiled contains imports from other modules, the actual directory is inspected first (by default). The `-d` option may be repeated and allows import from modules in library *directory*. The libraries are inspected in the order specified, but after the actual directory. Finally, the installation-specific system library (given as an argument of the `-syslib` option) is inspected.
The `-D` option allows you to specify a directory where the compilation results are put. On default the current directory is chosen.
A *script* is the name of an executable file containing Unix shell commands.
The *script* given in the `link` parameter is used by the MOCKA system to invoke the system linker. When a `p` *module* was entered in session mode, the system collects all modules imported by *module*, checks them for consistency, creates a root module containing the initializations for the module bodies, and then invokes the link *script* with *module* and a list of object files as arguments (*script module objectfile* ...).
The *script* given in the `edit` parameter is used by the MOCKA system to invoke the editor. When a `d` *module* or a `i` *module* was entered in session mode, the system calls *script* with the source file as argument (*script sourcefile*) .
The `-asm` and `-cc` allow you to use your own scripts to invoke an assembler or a C-Compiler respectively. Note that the NeXT-Version of the MOCKA Compiler does not know the `-asm` option, because the MOCKA system produces native code for this machine. The resulting code file, however, must be transformed to suit NeXT standards. A script can be specified for this task by the `-cvt` option.

The `-O` option switches on the Optimizer.
The *script* given in the `list` parameter is used by the MOCKA system to generate listings. When an error is detected in session mode during a compilation this *script* is called with the name of the sourcefile (where the error occurred) as argument. The *script* contains commands to generate the listing with the error messages interspersed, to invoke the editor, to remove the error messages from the corrected listing, and to copy the listing to the sourcefile.
Appendix D shows how implementations of such scripts used as options could look like.

2.5 Compiler Pragmas

Pragmas are instructions to the compiler and can be used to control code generation for runtime checks. Pragmas may be given at system invocation time, or during a session, or inside a Modula-2 source program. The pragmas described here are a speciality of the MOCKA system and therefore depend on this special Modula-2 implementation. Do not expect them in other implementations.
Pragmas available are

- range / norange
- index / noindex
- Stack_Test / noStack_Test
- S / noS
- static / nostatic
- g / nog
- O / noO
- blip / noblip
- info

The `Stack_Test / noStack_Test`, `S / noS`, `g / nog`, and `O / noO` pragma is not supported on all target systems.
The `range / norange` pragma causes the compiler to generate / not to generate additional code to check at runtime whether a value to be assigned is out of the range of the variable it will be assigned to, e.g. for the assignment statement or parameter passing. If the value is out of bounds, an error message will be emitted by the runtime system and the program execution will be aborted. Default value is `range`.
The `index / noindex` pragma causes the compiler to generate / not to generate additional code to check at runtime whether array indices are out of bounds, i.e. out of the range specified by the array's index type. If the index is out of bounds, an error message will be emitted by the runtime system and the program execution will be aborted. Default value is `index`.
The `Stack_Test / noStack_Test` pragma controls generation of additional code for workspace violations by procedure calls. Has to be set, if `NEWPROCESS` or `TRANSFER` are used. With programs that do not use coroutines this pragma is useless.
The `S` option prevents the MOCKA Compiler from deleting the file *module.s*, which contains the assembler code for the module *module.mi*. Default is `noS`.
The `g` option supports debugging. Default is `g`.
The `static` option will link the resulting executable statically. Default is `nostatic`.
The `blip` option switches on counting processed procedures during the semantic analysis. Default is `blip`.
By specifying `O` or `noO` you may enable or disenable the Optimizer. For MOCKA versions with a C-BackEnd this is the C-Optimizer.
Concluding `info` shows the current settings of the system installation parameters.
The syntax for pragmas used at system invocation time or during a session is
-*pragma*
The syntax for pragmas located in a Modula-2 source is:
%␣*pragma*
A pragma may be located in any declaration part of a module or a procedure, the scope of the pragma extends from the point where it is specified up to the end of the block containing the specification (cf. example 5).

2.6 Files and File Name Conventions

The commands of the MOCKA system expect module names instead of file names. To enable the MOCKA system to relate a module name and the name of the file that contains the module, the file names have to follow special conventions, i.e. the file name consists of the concatenation of the module name and a suffix that describes the purpose of the file.
The conventions are:

***module*.md** source file with definition module *module*.

***module*.mi** source file with implementation/program module *module*.

***module*.d** symbol file for definition module *module*. Used for inter module type checking, created as the result of the compilation of the correct definition module *module* .

***module*.i** symbol file for implementation/program module *module*. Used for debugging, created as the result of the compilation of a correct implementation/program module *module*.

***module*.r** reference file for implementation/program module *module*. Used for linking, created as the result of the compilation of a correct implementation/program module *module*.

***module*.s** contains the assembler code for the module *module*. It is removed after the assembler has produced executable code, unless the `S` option was specified.

***module*.o** code file for implementation/program module *module*. Created as the result of the compilation of a correct implementation/program module *module*.

module executable program for program module *module*.

3 The Modula-2 Implementation

The MOCKA system implements the language Modula-2 as defined in [Wir85] , with a few minor extensions.

3.1 Extensions of Modula-2

underscores in identifiers : The underscore character is considered as a letter and may be used in identifiers.

additional basic types LONGINT and LONGCARD : Two additional basic types `LONGINT` and `LONGCARD` are introduced. `LONGINT` comprises the integers between `MIN(LONGINT)` and `MAX(LONGINT)`. `LONGCARD` comprises the integers between 0 and `MAX(LONGCARD)`.

additional basic types SHORTINT and SHORTCARD : Two additional basic types `SHORTINT` and `SHORTCARD` are introduced. `SHORTINT` comprises the integers between `MIN(SHORTINT)` and `MAX(SHORTINT)`. `SHORTCARD` comprises the integers between 0 and `MAX(SHORTCARD)`.

type BYTE : The type `BYTE` exported from module `SYSTEM` represents an individually accessible storage unit. It is implemented as synonym to WORD. No operation except assignment is defined on this type. However, if a formal parameter of a procedure is of type `BYTE`, the corresponding actual parameter may be of any type that uses one storage unit. If a formal parameter has the type `ARRAY OF BYTE`, its corresponding actual parameter may be of any type.

3.2 Foreign Modules: Accessing Procedures written in other Languages

The MOCKA system allows that procedures not written in Modula-2 may be called in Modula-2 programs. Such procedures have to be declared in so-called *Foreign Modules* which act as definition modules. There is no corresponding implementation module. Instead the object file is created by e.g. the C compiler or the assembler. Nevertheless, the binder may be used as before.
Foreign modules have the same syntax as definition modules except that the keyword `DEFINITION` is replaced by `FOREIGN`. They may contain type, constant and procedure declarations, but they may not define global variables.
The compiler of the MOCKA system follows the C calling conventions, i.e. the parameters are passed from right to left. There is no specific conversion of Modula-2 parameters. The representation of Modula-2 objects is described in a section below. Table 1 illustrates the correspondence of actual Modula-2 parameters with formal C parameter types. Size and layout of Modula-2 types are described below.

formal C parameter type	actual Modula-2 argument
char	`CHAR`
int	`SHORTINT` or `LONGINT` (machine-dependent)
short int	`SHORTINT`
long int	`LONGINT`
unsigned	`SHORTCARD` or `LONGCARD` (machine-dependent)
unsigned short	`SHORTCARD`
unsigned long	`LONGCARD`
float	`REAL`
double	`LONGREAL`
pointer type	pointer type
enum type	enumeration type
struct	address of `RECOD` of corresponding size and layout
array	address of `ARRAY` of corresponding size and layout
open array, int	open array

Table 1: Modula-2 to C parameter passing

Complex structure like arrays (ARRAY), structs (RECORD) may be passed as VAR parameters to the corresponding constructs in C. Arrays and Records are passed as addresses, while open arrays are passed as address and last index (= HIGH(array)). For an example see Appendix C.

The MOCKA system expects an object file for each foreign module in the same directory where the corresponding symbol file is located. The object file has to be named according to the MOCKA file name conventions. The C functions that correspond to the Modula-2 procedures declared in a foreign module usually are located in some C library, maybe even in different ones. Compile these C sources, link the object files created by the C compiler without resolving the relocation information (*pre-linking* , cf. your Unix system linker manual for the corresponding options), and instruct the system linker to create as output an object file whose name corresponds with the foreign module's name. Move that object file into the directory where the corresponding symbol file is (or will be) located. The pre-linked object file has to be existent in that directory before the Modula-2 program that imports the corresponding foreign module is linked. Example 3 illustrates the use of foreign modules.

3.3 Storage Representations of Modula-2 Objects

The representation of Modula-2 objects in storage is machine- and implementation-dependent. MOCKA allocates Modula-2 objects as specified in table 2. Class 1 comprises Sun-3, pcs and NeXT computers, Class 2 VAX, SPARC, 80386 and MIPS.

Modula-2 type	size	alignment Class 1	alignment Class 2
`BOOLEAN`	1	byte address	byte address
`CHAR`	1	byte address	byte address
enumeration[1]	1 (or 2)	byte (or even) address	byte (or even) address
`SHORTINT`	2	even address	even address
`SHORTCARD`	2	even address	even address
`LONGINT`	4	even address	four byte boundary
`LONGCARD`	4	even address	four byte boundary
`INTEGER`	4	even address	four byte boundary
`CARDINAL`	4	even address	four byte boundary
subranges	size base type	align base type	align base type
`BYTE`	1	byte address	byte address
`WORD`	1	byte address	byte address
`REAL`[2]	4	even address	four byte boundary
`LONGREAL`[2]	8	even address	eight byte boundary
`BITSET`	4	even address	four byte boundary
`ADDRESS`	4	even address	four byte boundary
pointer type	4	even address	four byte boundary
opaque type	4	even address	four byte boundary
procedure type	4	even address	four byte boundary
set type	4	even address	four byte boundary

Table 2: storage representation of Modula-2 objects

3.4 Type Compatibility Rules

The Wirth report states about type compatibility (pp. 148):

> "A type T1 is said to be *compatible* with a type T0, if it is declared either as T1 = T0 or as a subrange of T0, or if T0 is a subrange of T1, or if T0 and T1 are both subranges of the same (base) type."

And on page 155:

> "A string of length 1 is compatible with the type `CHAR`."

[1] **enumeration types** If the number of literals in the enumeration type declaration does not exceed 256, variables of this type are stored in one byte. Otherwise, if the number of literals does not exceed 32767, such variables occupy two bytes.

[2] **`REAL`,`LONGREAL`** Variables of type `REAL` or `LONGREAL` are stored in IEEE floating-point format, except for the VAX system.

Those type compatibility rules are implemented as stated but with some relaxations concerning the new `SHORT` and `LONG` types. This means, two types that are compatible according to the report are also compatible in the MOCKA implementation.
`CARDINAL` and `LONGCARD` are implemented as synonyms, and also `INTEGER` and `LONGINT`. `BYTE` and `WORD` are synonyms either. Additionally, integer types (`INTEGER`, `SHORTINT`, `LONGINT`) are compatible with each other. The same holds for cardinal types (`CARDINAL`, `SHORTCARD`, `LONGCARD`).
Table 3 shows the *compatibilities* between arithmetic types as implemented in the MOCKA system. The "+" in position (x,y) indicates compatibility between type x and type y. Subrange types inherit the compatibilities of their base types. Other types not mentioned here are compatible only if they are equal.

	INTEGER	SHORTINT	LONGINT	CARDINAL	SHORTCARD	LONGCARD	REAL	LONGREAL	ADDRESS
`INTEGER`	+	+	+						
`SHORTINT`	+	+	+						
`LONGINT`	+	+	+						
`CARDINAL`				+	+	+			+
`SHORTCARD`				+	+	+			
`LONGCARD`				+	+	+			+
`REAL`							+		
`LONGREAL`								+	
`ADDRESS`				+		+			+

Table 3: type compatibility

The Wirth report states about assignment compatibility (pp. 155):

> "Operand types are said to be *compatible*, if either they are compatible or both are `INTEGER` or `CARDINAL` or subranges with base types `INTEGER` or `CARDINAL`.

Those type compatibility rules are implemented in the MOCKA system as stated in the report but with the relaxation that each integer or cardinal type is assignment compatible with each other integer or cardinal type. Table 4 shows the *assignment compatibilities* between types as implemented. The "+" in position (x,y) indicates that a variable of type x may be assigned a value of type y. Subrange types inherit the assignment compatibilities of their base types. Other types not mentioned here are assignment compatible only if they are equal.

	INTEGER	SHORTINT	LONGINT	CARDINAL	SHORTCARD	LONGCARD	REAL	LONGREAL	ADDRESS
`INTEGER`	+	+	+	+	+	+			
`SHORTINT`	+	+	+	+	+	+			
`LONGINT`	+	+	+	+	+	+			
`CARDINAL`	+	+	+	+	+	+			+
`SHORTCARD`	+	+	+	+	+	+			
`LONGCARD`	+	+	+	+	+	+			+
`REAL`							+	+	
`LONGREAL`							+	+	
`ADDRESS`				+		+			+

Table 4: assignment compatibility

3.5 Operations on Types

Table 5 shows which operations may be applied on objects of the various types (cf. [Wir85]). The operations available for subranges are identical with those for their base types.
The types of the right hand side and the left hand side in an assignment must be assignment compatible. The operands of binary operations must be compatible.
The operands of arithmetic binary operations may be of different, but compatible (base) types. If the operand types are identical, the result is of the operand type. Otherwise, the result is of the larger operand type (e.g. the result of the addition of a `SHORTINT` and a `LONGINT` variable is of type `LONGINT`). If one operand is a constant expression, the value of the expression has to be in the range of the other operand's type.

type	operations
`BOOLEAN`	:=, AND, OR, NOT, =, #
`CHAR`	:=, =, #
`INTEGER`	:=, +, -, *, DIV, MOD, =, #, <=, <, >, >=
`SHORTINT`	:=, +, -, *, DIV, MOD, =, #, <=, <, >, >=
`LONGINT`	:=, +, -, *, DIV, MOD, =, #, <=, <, >, >=
`CARDINAL`	:=, +, -, *, DIV, MOD, =, #, <=, <, >, >=
`SHORTCARD`	:=, +, -, *, DIV, MOD, =, #, <=, <, >, >=
`LONGCARD`	:=, +, -, *, DIV, MOD, =, #, <=, <, >, >=
`REAL`	:=, +, -, *, /, =, #, <=, <, >, >=
`LONGREAL`	:=, +, -, *, /, =, #, <=, <, >, >=
enumeration types	:=, =, #
`BITSET`	:=, IN, =, #, <=, >=
set types	:=, IN, =, #, <=, >=
array types	:=, indexing
record types	:=, selection
pointer types	:=, =, #, dereferencing
opaque types	:=, =, #
procedure types	:=,
`ADDRESS`	:=, =, #
`WORD`	:=, =, #
`BYTE`	:=, =, #

Table 5: operations availabe on types

3.6 Floating-Point Arithmetic

On MC68020/MC68881 and Intel80386 processor configurations the compiler supports the floating-point coprocessor by default.

3.7 Implementation Restrictions

The MOCKA system implements the language Modula-2 as specified in [Wir85] with the restrictions listed below.

code size of procedure and module bodies The code size of a procedure or module body must not exceed 32 KByte, because on MC680xx and MIPS processors the displacement of a branch instruction is restricted to a signed word.

function results The result type of a function procedure must not be an array nor a record type.

absolute addressing in variable declarations This non-standard facility (cf. [Wir85] page 167) is not supported. The compiler returns an error message if this feature is used.

module priorities The specification of a priority in a module heading (cf. [Wir85] report page 168) has no effect.

`IOTRANSFER` The procedure `IOTRANSFER` is not implemented.

constant declarations The right-hand side of a constant declaration may contain calls of built-in functions (standard procedures, functions exported from module `SYSTEM`).

set types Set types must contain elements whose ordinal numbers are in the range 0..31.

type *`WORD`* Wirth's report states that the type `WORD` represents an individually accessible storage unit. Bytes are individual accessible storage units. So, the type `WORD` represents a byte, not a machine word.

standard procedure *`ABS`* `ABS` returns a value whose type is the type of the argument.

standard procedure *`ORD`* If the argument type is of an integer type, the argument must have a non-negative value.

arithmetic overflow Checks on arithmetic overflow are not generated by the compiler.

`WITH` ***nesting depth*** Up to 16 `WITH` statements may be nested.

`CASE` ***labels*** The ordinal numbers of the smallest and and greatest case label used in a `CASE` statement must not differ more than 2048, i.e. `ORD`(greatest label) - `ORD`(smallest label) <= 2048. E.g., if the ordinal value of the smallest case label is 10000, the other labels may be in the range 10001 .. 12047. If the case label range exceeds this range for some reasons, use an IF-ELSIF construct or split the case statement into two or more.

type transfer functions The size of the argument of the type transfer function and the size of the destination type have to be equal.

passing of string constants as VAR-parameters It is allowed to pass a string constant to a VAR-parameter.

OC-Handling of strings String Constants are 0C terminated by the compiler, although [Wir85] does not explicitly call for this. The 0C character is not considered part of the string, so having passed it to an open array accessing it will cause an index check error. Therefore it is recommended to use the built-in function HIGH in order to determine the end of the string when using open array parameters.

4 The System Library

4.1 Description

The MOCKA system includes a small system library (including sources) that gives enough support to implement Modula-2 programs for simple applications. The library comprises buffered I/O for standard input/output devices *(InOut)*, and formatted/unformatted I/O via files *(TextIO, ByteIO)*, interfaces to the Unix system *(SysLib, ErrNumbers, Arguments, Clock, Signals)* , modules to allocate data structures dynamically *(Storage, MemPools)*, a module with mathematic functions *(MathLib)*, modules for conversions *(NumConv, RealConv, LREAL)*, a module for simple string handling *(Strings1)*, and a simple exception mechanism (not all target systems) *(Exceptions)*. The user should have in mind that there exists currently no standard library for Modula-2, and that the modules listed in the appendix of the Wirth report are only meant as proposals (some modules are meaningful only on Wirth's Lilith computer).

4.2 Definition Modules

The following list of annotated definition modules is intended to aid proper use of the system library.

4.2.1 Unix interfaces

```
FOREIGN MODULE Arguments;

   TYPE ArgTable =
      POINTER TO ARRAY [0..999] OF
      POINTER TO ARRAY [0..999] OF CHAR;

   PROCEDURE GetArgs (VAR argc: SHORTCARD; VAR argv: ArgTable);
      (* GetArgs provides access to arguments of the shell command that *)
      (* invoked the current program. After a call GetArgs (argc, argv) *)
      (* argc is the number of arguments and argv^[i] is a pointer to *)
      (* the i-th argument *)
      (* (i in [0 .. argc-1]; argv^[0] is the command name). *)

   PROCEDURE GetEnv (VAR env: ArgTable);
      (* GetEnv provides access to Unix environment variables. *)
      (* After a call GetEnv(env) env^[i] is a pointer to the i-th *)
      (* variable definition (which is a string of the form name=val) *)
      (* (i in [0 .. n] , where env^[n] is the first entry = NIL *)

END Arguments.
```

Note that the transputer version of `Arguments` is a normal definition module with corresponding implementation module.

```
DEFINITION MODULE Clock;

   PROCEDURE ResetClock;

   PROCEDURE UserTime() : INTEGER;
      (* Return user time since last 'ResetClock' *)

   PROCEDURE SystemTime() : INTEGER;
      (* Return system time since last 'ResetClock' *)

END Clock.

FOREIGN MODULE ErrNumbers;

   PROCEDURE ErrNo () : SHORTCARD;
      (* Returnes the error number set by system calls *)

   CONST
      ePERM    = 1;         (* Not super-user *)
      eNOENT   = 2;         (* No such file or directory *)
      eSRCH    = 3;         (* No such process *)
      eINTR    = 4;         (* interrupted system call *)
      eIO      = 5;         (* I/O error *)
      eNXIO    = 6;         (* No such device or address *)
      e2BIG    = 7;         (* Arg list too long *)
      eNOEXEC  = 8;         (* Exec format error *)
      eBADF    = 9;         (* Bad file number *)
      eCHILD   = 10;        (* No children *)
```

```
      eAGAIN   = 11;        (* No more processes *)
      eNOMEM   = 12;        (* Not enough core *)
      eACCES   = 13;        (* Permission denied *)
      eFAULT   = 14;        (* Bad address *)
      eNOTBLK  = 15;        (* Block device required *)
      eBUSY    = 16;        (* Mount device busy *)
      eEXIST   = 17;        (* File exists *)
      eXDEV    = 18;        (* Cross-device link *)
      eNODEV   = 19;        (* No such device *)
      eNOTDIR  = 20;        (* Not a directory *)
      eISDIR   = 21;        (* Is a directory *)
      eINVAL   = 22;        (* Invalid argument *)
      eNFILE   = 23;        (* File table overflow *)
      eMFILE   = 24;        (* Too many open files *)
      eNOTTY   = 25;        (* Not a typewriter *)
      eTXTBSY  = 26;        (* Text file busy *)
      eFBIG    = 27;        (* File too large *)
      eNOSPC   = 28;        (* No space left on device *)
      eSPIPE   = 29;        (* Illegal seek *)
      eROFS    = 30;        (* Read only file system *)
      eMLINK   = 31;        (* Too many links *)
      ePIPE    = 32;        (* Broken pipe *)
      eDOM     = 33;        (* Math arg out of domain of func *)
      eRANGE   = 34;        (* Math result not representable *)
      eNOMSG   = 35;        (* No message of desired type *)
      eIDRM    = 36;        (* Identifier removed *)
      eCHRNG   = 37;        (* Channel number out of range *)
      eL2NSYNC = 38;        (* Level 2 not synchronized *)
      eL3HLT   = 39;        (* Level 3 halted *)
      eL3RST   = 40;        (* Level 3 reset *)
      eLNRNG   = 41;        (* Link number out of range *)
      eUNATCH  = 42;        (* Protocol driver not attached *)
      eNOCSI   = 43;        (* No CSI structure available *)
      eL2HLT   = 44;        (* Level 2 halted *)
      eNOSWP   = 45;        (* Out of swap space *)
      eXPATH   = 46;        (* Path continues onto another machine *)
      eXREDO   = 47;        (* Unison request for a retry *)
      eDEADLK  = 48;        (* Record locking deadlock *)
      eNOUARP  = 49;        (* Could not resolve IP addr, host down *)
      eNOUGW   = 50;        (* No available gateway in route table *)
      eLOOP    = 51;        (* symbolic links form endless loop *)

END ErrNumbers.

FOREIGN MODULE Signals;

  FROM SysLib IMPORT
     SIGNED;

  CONST
     SIGHUP  = 1;      (* hangup *)
     SIGINT  = 2;      (* interrupt *)
     SIGQUIT = 3;      (* quit *)
     SIGILL  = 4;      (* illegal instruction *)
     SIGTRAP = 5;      (* trace or breakpoint *)
     SIGIOT  = 6;      (* IOT instruction *)
     SIGEMT  = 7;      (* ETT instruction *)
     SIGFPE  = 8;      (* floating exception *)
     SIGKILL = 9;      (* kill, uncatchable termination *)
     SIGBUS  = 10;     (* bus error *)
     SIGSEGV = 11;     (* segmentation violation *)
     SIGSYS  = 12;     (* bad argument to system call *)
     SIGPIPE = 13;     (* end of pipe *)
     SIGALRM = 14;     (* alarm clock *)
     SIGTERM = 15;     (* software termination signal *)
     SIGADDR = 16;     (* odd address error *)
     SIGZERO = 17;     (* zero divide *)
     SIGCHK  = 18;     (* check error *)
     SIGOVER = 19;     (* arithmetic overflow *)
     SIGPRIV = 20;     (* privilege violation *)
     SIGUSR1 = 21;     (* user defined signal 1 *)
     SIGUSR2 = 22;     (* user defined signal 2 *)
     SIGCLD  = 23;     (* death of a child (old signal) *)
     SIGPWR  = 24;     (* power-fail restart *)

  TYPE
     Signal        = SIGNED [1 .. 24];
     SignalHandler = PROCEDURE (Signal);
```

```
  PROCEDURE signal
     (sig  : Signal; func : SignalHandler) : SignalHandler;
     (* Defines the handler procedure for the specified signal 'sig' *)
     (* as 'func'. Returns the previous (or initial) handler procedure *)
     (* of the particular signal. Upon recept of the signal 'sig' the *)
     (* handler procedure 'func' is executed. Before entering 'func', *)
     (* the handler procedure of 'sig' is reset to it's initial value. *)
     (* Upon return from 'func', execution continues at the interrupted *)
     (* point. *)

END Signals.
```

The module SysLib is highly machine-dependent and thus exists in many different versions, of which only the mips-version is printed here.

```
FOREIGN MODULE SysLib;

   FROM SYSTEM IMPORT ADDRESS;

   TYPE
      SIGNED    = INTEGER;
      UNSIGNED  = INTEGER;

      inoT      = CARDINAL;
      offT      = INTEGER;
      devT      = SHORTINT;
      timeT     = INTEGER;

      Stat =
         RECORD
            stDev    : devT;
            stIno    : inoT;
            stMode   : SHORTCARD;
            stNlink  : SHORTINT;
            stUid    : SHORTINT;
            stGid    : SHORTINT;
            stRdev   : devT;
            stSize   : offT;
            stAtime  : timeT;
            stSpare1 : INTEGER;
            stMtime  : timeT;
            stSpare2 : INTEGER;
            stCtime  : timeT;
            stSpare3 : INTEGER;
            stBlksize: INTEGER;
            stBlocks : INTEGER;
            stSpare4 : ARRAY [0..1] OF INTEGER;
         END;
      clockT = LONGINT;

      tms =
         RECORD
            utime  : clockT;
            stime  : clockT;
            cutime : clockT;
            cstime : clockT;
         END;


   CONST

      (* flags for open *)

      oTRUNC   = 01000B;    (* open with truncation *)
      oAPPEND  =   010B;    (* append, i.e writes at the end *)
      oRDWR    =    02B;    (* open for reading and writing *)
      oWRONLY  =    01B;    (* open for writing only *)
      oRDONLY  =     0B;    (* open for reading only *)

      (* file access permisson flags (for create and umask) *)

      pXUSID   = 04000B;    (* set user ID on execution *)
      pXGRID   = 02000B;    (* set group ID on execution *)
      pSTEXT   = 01000B;    (* save text image after execution *)
      pROWNER  =  0400B;    (* read by owner *)
      pWOWNER  =  0200B;    (* write by owner *)
      pXOWNER  =  0100B;    (* execute by owner *)
      pRGROUP  =   040B;    (* read by group *)
      pWGROUP  =   020B;    (* write by group *)
```

```
    pXGROUP  =   010B;    (* execute by group *)
    pROTHERS =    04B;    (* read by others *)
    pWOTHERS =    02B;    (* write by others *)
    pXOTHERS =    01B;    (* execute by others *)
    pEMPTY   =     0B;    (* no flag set *)

    (* file access check flags (for access) *)

    cREAD    = 04H;       (* check if readable *)
    cWRITE   = 02H;       (* check if writable *)
    cEXEC    = 01H;       (* check if executable *)
    cEXISTS  =  0H;       (* check existance *)

  PROCEDURE umask (cmask : SIGNED) : SIGNED;

  PROCEDURE access (path  : ADDRESS; amode : SIGNED) : SIGNED;

  PROCEDURE creat (path  : ADDRESS; cmode : SIGNED) : SIGNED;

  PROCEDURE open (path : ADDRESS; oflag : SIGNED) : SIGNED;

  PROCEDURE close (fildes : SIGNED) : SIGNED;

  PROCEDURE unlink (path : ADDRESS) : SIGNED;

  PROCEDURE read (fildes : SIGNED; buf : ADDRESS; nbyte : UNSIGNED) : SIGNED;

  PROCEDURE write (fildes : SIGNED; buf : ADDRESS; nbyte : UNSIGNED) : SIGNED;

  PROCEDURE sbrk (incr : SIGNED): ADDRESS;

  PROCEDURE malloc (size : UNSIGNED) : ADDRESS;

  PROCEDURE free (ptr : ADDRESS);

  PROCEDURE stat (path: ADDRESS; VAR buf: Stat) : SIGNED;

  PROCEDURE fstat (fd: SIGNED  ; VAR buf: Stat) : SIGNED;

  PROCEDURE time (VAR t : INTEGER);

  PROCEDURE times (VAR buffer: tms);

  PROCEDURE system (string : ADDRESS) : SIGNED;

  PROCEDURE exit (n: SIGNED);

  PROCEDURE abort ();

END SysLib.
```

4.2.2 Storage, MemPools

```
DEFINITION MODULE Storage;

  FROM SYSTEM IMPORT ADDRESS;

  PROCEDURE ALLOCATE (VAR a : ADDRESS; size : CARDINAL);
     (* Allocates an area of the given size 'size' and returns it's *)
     (* address in 'a'. If no space is available, 'a' becomes 'NIL'. *)

  PROCEDURE DEALLOCATE (VAR a : ADDRESS; size : CARDINAL);
     (* Frees the area of size 'size' starting at address 'a'. *)
     (* Upon return 'a' is set 'NIL' *)

END Storage.

DEFINITION MODULE MemPools;

  FROM SYSTEM IMPORT ADDRESS;

  TYPE
    MemPool;

  PROCEDURE NewPool(VAR pool: MemPool);
  (* Does create a new [empty] MemPool. *)

  PROCEDURE PoolAllocate(VAR pool: MemPool; VAR ptr: ADDRESS; want: CARDINAL);
  (* Allocates want bytes of memory out of pool MemPool.        *)
  (* ptr's alignment will be suitable for all types.    *)

  PROCEDURE KillPool(VAR pool: MemPool);
```

```
  (* Destroys the pool. *)

END MemPools.
```

4.2.3 Strings, Strings1

```
DEFINITION MODULE Strings;

   TYPE String = ARRAY [0..255] OF CHAR;

   PROCEDURE EmptyString (VAR str: ARRAY OF CHAR);
      (* str := "" *)

   PROCEDURE Assign (VAR dst, src: ARRAY OF CHAR);
      (* assign string 'src' to string 'dst'. 'src' must be terminated by 0C *)

   PROCEDURE Append (VAR dest, suffix: ARRAY OF CHAR);
      (* append 'suffix' to 'dest', only significant characters. *)

   PROCEDURE StrEq (VAR x, y: ARRAY OF CHAR): BOOLEAN;
      (* x = y , only significant characters. *)

   PROCEDURE Length (VAR str : ARRAY OF CHAR) : CARDINAL;
      (* returns the number of significant characters. *)

   PROCEDURE Insert
      (substr: ARRAY OF CHAR; VAR str: ARRAY OF CHAR; inx: CARDINAL);
      (* Inserts 'substr' into 'str', starting at str[inx] *)

   PROCEDURE Delete (VAR str: ARRAY OF CHAR; inx, len: CARDINAL);
      (* Deletes 'len' characters from 'str', starting at str[inx] *)

   PROCEDURE pos (substr: ARRAY OF CHAR; str: ARRAY OF CHAR): CARDINAL;
      (* Returns the index of the first occurrence of 'substr' in 'str' or *)
      (* HIGH (str) + 1 if 'substr' not found. *)

   PROCEDURE Copy
      (str: ARRAY OF CHAR; inx, len: CARDINAL; VAR result: ARRAY OF CHAR);
      (* Copies 'len' characters from 'str' into 'result', *)
      (* starting at str[inx] *)

   PROCEDURE Concat
      (s1, s2: ARRAY OF CHAR; VAR result: ARRAY OF CHAR);
      (* Returns in 'result' the concatenation of 's1' and 's2' *)

   PROCEDURE compare (s1, s2: ARRAY OF CHAR): INTEGER;
      (* Compares 's1' with 's2' and returns -1 if s1 < s2, 0 if s1 = s2, *)
      (* or 1 if s1 > s2 *)

   PROCEDURE CAPS (VAR str: ARRAY OF CHAR);
      (* CAP for the entire 'str' *)

END Strings.
```

About the OC termination of strings see Chapter 3.7 dealing with implementation features. The module Strings1 contains the same types, procedures, etc. as module Strings. This is needed to avoid name clashes with the reuse library. Please use Strings1 instead of Strings if you use the reuse library.

4.2.4 BasicIO, ByteIO, TextIO

```
DEFINITION MODULE BasicIO;

   FROM SYSTEM IMPORT ADDRESS;

   TYPE File = INTEGER;

   PROCEDURE OpenInput (VAR file : File; VAR name : ARRAY OF CHAR);
   (* Open file 'file' for input. Use External name 'name' *)

   PROCEDURE OpenOutput (VAR file : File; VAR name : ARRAY OF CHAR);
   (* Open file 'file' for input. Use External name 'name' *)

   PROCEDURE Close (file: File);
   (* Close file 'file' *)

   PROCEDURE Erase (VAR name : ARRAY OF CHAR;
                    VAR ok   : BOOLEAN);
   (* erase file with external name 'name' *)

   PROCEDURE Read (file : File; x : ADDRESS; n : INTEGER; VAR read : INTEGER);
```

```
   (* Read n bytes from file 'file' into area starting at address 'x' *)
   (* On exit 'read' denotes the number of bytes actually read *)

   PROCEDURE Write (file : File; x : ADDRESS; n : INTEGER);
   (* Write 'n' bytes of area starting at address 'x' to file 'file' *)

   PROCEDURE Accessible (VAR name       : ARRAY OF CHAR;
                            ForWriting : BOOLEAN) : BOOLEAN;
   (* Return true if the file with external name 'name' is accessible for
      reading (ForWriting = FALSE) resp. for writing (ForWriting = TRUE).
    *)

   VAR DONE : BOOLEAN;  (* READ ONLY variable *)
END BasicIO.

DEFINITION MODULE ByteIO;

FROM SYSTEM IMPORT
     (* TYPE *) BYTE;

IMPORT BasicIO;
TYPE File = BasicIO.File;

(* for all variants of the mocka system *)

PROCEDURE OpenInput (VAR file : File; VAR name : ARRAY OF CHAR);
(*****************************************************************)
(* Open file 'file' for input. Use External name 'name'          *)
(* Any sequential file any record format may be opened.          *)
(*****************************************************************)

PROCEDURE OpenOutput (VAR file : File; VAR name : ARRAY OF CHAR);
(*****************************************************************)
(* Open file 'file' for input. Use External name 'name'          *)
(* A sequential file with undefined record format is opend.      *)
(*****************************************************************)

PROCEDURE Close (file: File);
(* Close file 'file' *)

(*=== Byte IO ===*)

PROCEDURE GetByte (file : File; VAR x : BYTE);
(* Read the next byte of file 'file' into 'x' *)

PROCEDURE GetBytes (file : File; VAR x : ARRAY OF BYTE; n : CARDINAL);
(* Read the next 'n' bytes of file 'file' into 'x' *)

PROCEDURE GetItem (file : File; VAR x : ARRAY OF BYTE);
(* Read the next n bytes of file 'file' into 'x' *)
(* where n = HIGH(x)+1                            *)

PROCEDURE PutByte (file : File; x : BYTE);
(* Write the byte 'x' to file 'file' *)

PROCEDURE PutBytes (file : File; VAR x : ARRAY OF BYTE; n : CARDINAL);
(* Write the first 'n' bytes of 'x' to file 'file' *)

PROCEDURE PutItem (file : File; VAR x : ARRAY OF BYTE);
(* Write the first n bytes of 'x' to file 'file' *)
(* where n = HIGH(x)+1 *)

(*=== Misc. ===*)

PROCEDURE UndoGetByte (file: File);
(*************************************************************************)
(* The last 'GetByte' is undone, i.e The same byte is returned with the next  *)
(* 'GetByte'.                                                            *)
(* NOTICE: no other 'Get...' procedure is allowed to be called between   *)
(*        'GetByte' and 'UndoGetByte'!!!                                 *)
(* 'Done ()' signals success.                                            *)
(*************************************************************************)

PROCEDURE Done () : BOOLEAN;
(*************************************************************************)
(* last operation successfully                                           *)
(* (notice: if during reading a file the end-of-file is reached, then 'Done'  *)
(*         returns FALSE).                                               *)
(*************************************************************************)

PROCEDURE PutBf (file : File);
(* Emit buffer contents immediately *)
```

```
PROCEDURE EOF (file : File) : BOOLEAN;
(* TRUE iff next read will read beyond end of file *)

PROCEDURE Accessible
 (VAR name : ARRAY OF CHAR; ForWriting : BOOLEAN) : BOOLEAN;
(* returns true if the file with external name 'name' is *)
(* accessible for writing ('ForWriting' is true)        *)
(* or for readig ('ForWriting' is false).               *)

PROCEDURE Erase (VAR name : ARRAY OF CHAR; VAR ok : BOOLEAN);
(* Erase the file wirh external name 'name'. *)
(* On exit 'ok' indicates success.           *)

END ByteIO.

DEFINITION MODULE TextIO;

IMPORT BasicIO;

   TYPE File = BasicIO.File;

   (*=== Open/Close ===*)

PROCEDURE OpenInput (VAR file : File; VAR name : ARRAY OF CHAR);
(* Open file 'file' for input. Use External name 'name' *)

PROCEDURE OpenOutput (VAR file : File; VAR name : ARRAY OF CHAR);
(* Open file 'file' for input. Use External name 'name' *)

PROCEDURE Close (file: File);
(* Close file 'file' *)

(*=== Formatted IO ===*)

PROCEDURE GetChar (file : File; VAR x : CHAR);
(* Read the next char of file 'file' into 'x' *)

PROCEDURE GetString (file : File; VAR x : ARRAY OF CHAR);
(* Read the next string of file 'file' into 'x'      *)
(* A string is a non empty sequence of characters    *)
(* not containing blanks nor control characters.     *)
(* Leading blanks and control characters are ignored *)
(* Input is terminated by any character <= ' '       *)

PROCEDURE GetCard (file : File; VAR x : CARDINAL);
(* Read the next string from file 'file' *)
(* and convert it to cardinal 'x'        *)

PROCEDURE GetInt (file : File; VAR x : INTEGER);
(* Read the next string from file 'file' *)
(* and convert it to integer 'x'         *)

PROCEDURE GetReal (file : File; VAR x : REAL);
(* Read the next string from file 'file' *)
(* and convert it to real 'x'            *)

PROCEDURE GetLongReal (file : File; VAR x : LONGREAL);
(* Read the next string from file 'file' *)
(* and convert it to long real 'x'       *)

PROCEDURE PutChar (file : File; x : CHAR);
(* Write the char 'x' to file 'file' *)

PROCEDURE PutString (file : File; VAR x : ARRAY OF CHAR);
(* Write the string 'x' to file 'file' *)

PROCEDURE PutCard (file : File; x : CARDINAL; n : CARDINAL);
(* Convert the cardinal 'x' into decimal representation     *)
(* and write it to file 'file'. Field width is at least 'n' *)

PROCEDURE PutOct (file : File; x : CARDINAL; n : CARDINAL);
(* Convert the cardinal 'x' into octal representation       *)
(* and write it to file 'file'. Field width is at least 'n' *)

PROCEDURE PutHex (file : File; x : CARDINAL; n : CARDINAL);
(* Convert the cardinal 'x' into hexadecimal representation *)
(* and write it to file 'file'. Field width is at least 'n' *)

PROCEDURE PutInt (file : File; x : INTEGER; n : CARDINAL);
(* Convert the integer 'x' into decimal representation      *)
(* and write it to file 'file'. Field width is at least 'n' *)
```

```
PROCEDURE PutReal (file : File; x : REAL; n : CARDINAL; k : INTEGER);
(* Convert the real 'x' into external representation       *)
(* and write it to file 'file'. Field width is at least 'n'. *)
(* If k > 0 use k decimal places.                            *)
(* If k = 0 write x as integer.                              *)
(* If k < 0 use scientific notation.                         *)

PROCEDURE PutLongReal (file : File; x : LONGREAL; n : CARDINAL; k : INTEGER);
(* Convert the long real 'x' into external representation   *)
(* and write it to file 'file'. Field width is at least 'n' *)
(* If k > 0 use k decimal places.                           *)
(* If k = 0 write x as integer.                             *)
(* If k < 0 use scientific notation.                        *)

PROCEDURE PutLn (file : File);
(* Write the end of line character to file 'file' *)

PROCEDURE PutBf (file : File);
(* Emit buffer contents immediately *)

(*=== Misc. ===*)

PROCEDURE UndoGetChar (file: File);
(******************************************************************************)
(* The last 'GetChar' is undone, i.e The same char is returned with the next  *)
(* 'GetChar'.                                                                 *)
(* NOTICE: no other 'Get...' procedure is allowed to be called between        *)
(*         'GetChar' and 'UndoGetChar'!!!                                     *)
(* 'Done ()' signals success.                                                 *)
(******************************************************************************)


PROCEDURE Done () : BOOLEAN;
(******************************************************************************)
(* TRUE if the last action on the file succeeds, FALSE else                   *)
(* (notice: if during reading a file the end-of-file is reached, then 'Done'  *)
(*          returns FALSE).                                                   *)
(******************************************************************************)

PROCEDURE EOF (file : File) : BOOLEAN;
(******************************************************************************)
(* TRUE  if it is tried to read beyond the last record in the file.           *)
(* FALSE else.                                                                *)
(******************************************************************************)

PROCEDURE Accessible
      (VAR name : ARRAY OF CHAR; ForWriting : BOOLEAN) : BOOLEAN;
(* returns true if the file with external name 'name' is *)
(* accessible for writing ('ForWriting' is true)        *)
(*  or for readig ('ForWriting' is false).              *)

PROCEDURE Erase (VAR name : ARRAY OF CHAR; VAR ok : BOOLEAN);
(* Erase the file wirh external name 'name'. *)
(* On exit 'ok' indicates success.          *)

END TextIO.
```

4.2.5 Exceptions

```
FOREIGN MODULE Exceptions;

  FROM SYSTEM IMPORT
    WORD;

  TYPE
    Exception = ARRAY [1 .. 300] OF WORD; (* PRIVATE *)

  PROCEDURE SetJump (VAR exc : Exception) : LONGINT;
     (* Setjump saves the current process state for later use by LongJump. *)
     (* It always returnes zero. Warning: if SetJump is called on parameter *)
     (* position, absolute chaos is guaranteed. *)


  PROCEDURE LongJump (VAR exc : Exception; val : LONGINT);
     (* LongJump restores the process state saved by the last call of SetJump *)
     (* After LongJump is completed, program execution continues as if the *)
     (* corresponding call of SetJump had returned 'val'. If 'val' is 0 upon *)
     (* a call of LongJump, SetJump will return 1. *)

END Exceptions.
```

Note that the Type Exception is maschine dependent. Only the mips version is printed here.

4.2.6 InOut

```
DEFINITION MODULE InOut;

   PROCEDURE Read (VAR x : CHAR);
   (* Read the next character from std input into 'x' *)


   PROCEDURE ReadString (VAR x : ARRAY OF CHAR);
   (* Read the next string from std input into 'x'. *)
   (* Leading blanks are ignored.                   *)
   (* Input is terminated by any character <= ' '   *)

   PROCEDURE ReadCard (VAR x : CARDINAL);
   (* Read the next string from std input and *)
   (* convert it to cardinal 'x'.             *)
   (* Syntax : digit {digit}                  *)

   PROCEDURE ReadInt (VAR x : INTEGER);
   (* Read the next string from std input and *)
   (* convert it to integer 'x'.              *)
   (* Syntax : ['+'|'-'] digit {digit}        *)

   PROCEDURE ReadReal (VAR x : REAL);
   (* Read the next string from std input and convert it   *)
   (* to real 'x'.                                          *)
   (* Syntax : ['+'|'-'] digit {digit} ['.' digit {digit}] *)
   (*          ['E'['+'|'-'] digit {digit}]                 *)

   PROCEDURE ReadLongReal (VAR x : LONGREAL);
   (* Read the next string from std input and convert it   *)
   (* to long real 'x'.                                     *)
   (* Syntax : ['+'|'-'] digit {digit} ['.' digit {digit}] *)
   (*          ['E'['+'|'-'] digit {digit}]                 *)

   PROCEDURE Write (x : CHAR);
   (* Write character 'x' onto std output *)

   PROCEDURE WriteString (x : ARRAY OF CHAR);
   (* Write the string 'x' onto std output *)

   PROCEDURE WriteCard (x : CARDINAL; n : CARDINAL);
   (* Convert the cardinal 'x' into decimal representation and *)
   (* write it onto std output. Field width is at least 'n'.   *)

   PROCEDURE WriteOct (x : CARDINAL; n : CARDINAL);
   (* Convert the cardinal 'x' into octal representation and *)
   (* write it onto std output. Field width is at least 'n'. *)

   PROCEDURE WriteHex (x : CARDINAL; n : CARDINAL);
   (* Convert the cardinal 'x' into hexadecimal representation   *)
   (* and write it onto std output. Field width is at least 'n'. *)

   PROCEDURE WriteInt (x : INTEGER; n : CARDINAL);
   (* Convert the integer 'x' into decimal representation and *)
   (* write it onto std output. Field width is at least 'n'.  *)

   PROCEDURE WriteReal (x : REAL; n : CARDINAL; k : INTEGER);
   (* Convert the real 'x' into external representation and  *)
   (* write it onto std output. Field width is at least 'n'. *)
   (* If k > 0 use k decimal places.                         *)
   (* If k = 0 write x as integer.                           *)
   (* If k < 0 use scientific notation.                      *)

   PROCEDURE WriteLongReal (x : LONGREAL; n : CARDINAL; k : INTEGER);
   (* Convert long real 'x' into external representation and  *)
   (* write it onto std output. Field width is at least 'n'.  *)
   (* If k > 0 use k decimal places.                          *)
   (* If k = 0 write x as integer.                            *)
   (* If k < 0 use scientific notation.                       *)

   PROCEDURE WriteLn;
   (* Write the end of line character onto std output *)
   (* Emit buffer contents immediately                *)

   PROCEDURE WriteBf;
   (* Emit buffer contents immediately *)
```

```
   PROCEDURE Done () : BOOLEAN;
   (* last operation ok *)

   PROCEDURE EOF () : BOOLEAN;
   (* EOF at standard input *)

END InOut.
```

4.2.7 NumConv, RealConv, LREAL

```
DEFINITION MODULE NumConv;

  CONST
    MaxBase = 16;

  TYPE
    tBase = [2..MaxBase];


  PROCEDURE Str2Num(VAR num: LONGCARD; base: tBase;
                    str: ARRAY OF CHAR;
                    VAR done: BOOLEAN);
    (* Convert 'str' to 'num' using 'base' *)

  PROCEDURE Num2Str(num: LONGCARD; base: tBase;
                    VAR str: ARRAY OF CHAR;
                    VAR done: BOOLEAN);
    (* Convert 'num' to 'str' using 'base' *)

  PROCEDURE AdjustWidth(VAR str: ARRAY OF CHAR; width: INTEGER; filler: CHAR);
    (* make str at least abs(width) chars long
       width >= 0: insert filler chars to the left if necessary
       width < 0 : append filler chars if necessary
     *)

END NumConv.

DEFINITION MODULE RealConv;

  (* This procedures only work if maschine use IEEE Standard for REALs *)

  PROCEDURE IsLongRealInfinityOrNaN (x: LONGREAL) : BOOLEAN;
  PROCEDURE IsRealInfinityOrNaN     (x: REAL)     : BOOLEAN;
  PROCEDURE IsLongRealInfinity      (x: LONGREAL) : BOOLEAN;
  PROCEDURE IsRealInfinity          (x: REAL)     : BOOLEAN;


  PROCEDURE Str2Real(s: ARRAY OF CHAR; VAR done: BOOLEAN): REAL;
  (* Converts the string 's' to real 'x'.                      *)
  (* s has to be of the form:                                  *)
  (* real   = num ['.' {digit}] ['E' num].                     *)
  (* num    = ['+'|'-'] digit {digit}.                         *)

  PROCEDURE Str2LongReal(s: ARRAY OF CHAR; VAR done: BOOLEAN): LONGREAL;
  (* Converts the string 's' to real 'x'.                      *)
  (* s has to be of the form:                                  *)
  (* real   = num ['.' {digit}] ['E' num].                     *)
  (* num    = ['+'|'-'] digit {digit}.                         *)

  PROCEDURE Real2Str (x : REAL; n : CARDINAL; k : INTEGER;
                      VAR s: ARRAY OF CHAR; VAR done: BOOLEAN);
  (* Convert real 'x' into external representation.    *)
  (* IF k > 0 use k decimal places.                    *)
  (* IF k = 0 write 'x' as integer.                    *)
  (* IF k < 0 use scientific notation.                 *)

  PROCEDURE LongReal2Str (x : LONGREAL; n : CARDINAL; k : INTEGER;
                          VAR s: ARRAY OF CHAR; VAR done: BOOLEAN);
  (* Convert long real 'x' into external representation.*)
  (* IF k > 0 use k decimal places.                    *)
  (* IF k = 0 write 'x' as integer.                    *)
  (* IF k < 0 use scientific notation.                 *)
END RealConv.

FOREIGN MODULE LREAL;
   PROCEDURE LTRUNC(x: LONGREAL): LONGINT;
   PROCEDURE LFLOAT(x: LONGINT): LONGREAL;
END LREAL.
```

4.2.8 MathLib

```
DEFINITION MODULE MathLib;

   PROCEDURE sqrt (x : REAL) : REAL;
      (* calculates the square root of 'x' *)

   PROCEDURE sqrtL (x : LONGREAL) : LONGREAL;
      (* calculates the square root of 'x' *)

   PROCEDURE exp (x : REAL) : REAL;
      (* calculates 'e' to the power of 'x', 'e' Euler's number *)

   PROCEDURE expL (x : LONGREAL) : LONGREAL;
      (* calculates 'e' to the power of 'x', 'e' Euler's number *)

   PROCEDURE ln (x : REAL) : REAL;
      (* calculates natural logarithm of 'x' *)

   PROCEDURE lnL (x : LONGREAL) : LONGREAL;
      (* calculates natural logarithm of 'x' *)

   PROCEDURE sin (x : REAL) : REAL;
      (* calculates sine of 'x' *)

   PROCEDURE sinL (x : LONGREAL) : LONGREAL;
      (* calculates sine of 'x' *)

   PROCEDURE cos (x : REAL) : REAL;
      (* calculates cosine of 'x' *)

   PROCEDURE cosL (x : LONGREAL) : LONGREAL;
      (* calculates cosine of 'x' *)

   PROCEDURE arctan (x : REAL) : REAL;
      (* calculates arc tangent of 'x' *)

   PROCEDURE arctanL (x : LONGREAL) : LONGREAL;
      (* calculates arc tangent of 'x' *)

   PROCEDURE real (x : INTEGER) : REAL;
      (* converts 'x' to type 'REAL' *)

   PROCEDURE realL (x : INTEGER) : LONGREAL;
      (* converts 'x' to type 'LONGREAL' *)

   PROCEDURE entier (x : REAL) : INTEGER;
      (* calculates the largest integer <= 'x' *)

   PROCEDURE entierL (x : LONGREAL) : INTEGER;
      (* calculates the largest long integer <= 'x' *)

END MathLib.
```

A Example 1: First Steps with the MOCKA System

To ease the first steps with the MOCKA system, we present here a demo session that involves editing, compiling, binding, and executing of a program module that has no imports (besides an import of module *InOut* which is located in the system library).
First, to invoke the MOCKA system in session mode, type (let '$' be the prompt of the operating system):

```
$ mc
```

The MOCKA system responds with the version number and then enters the session mode. Now you can enter one of the commands that are available for session mode (or a Unix command). We want to create a program module named *hello*, so we enter (let '>>' be the prompt of the MOCKA system):

```
>> i hello
```

If a source file with the name *hello.mi* already exists in the actual working directory, this source file would be loaded into the editor. In our example we assume that there is no such file, so the editor is loaded with the newly created empty file. You can now type in a Modula-2 program, e.g.:

```
MODULE hello;
  FROM InOut IMPORT WriteString, WriteLn;
BEGIN
  WriteString("hello world");
  WriteLn;
END helo.
```

The last line of this program contains an misspelt `hello`. After having typed in this program, leave the editor. You are then back in the MOCKA system and you can compile the program module by typing:

```
>> c hello
```

The compiler detects the error and generates a listing which is loaded into the editor:

```
@ LISTING
MODULE hello;
  FROM InOut IMPORT WriteString, WriteLn;
BEGIN
  WriteString("hello world");
  WriteLn;
END helo.
@   ^
@ Col 5: module identifiers do not match
```

You can find the position of an error messages in the listing by searching for the character in the first column of the first line of the listing. If you are using the vi editor invoked by the standard edit script you may just type `v` to move to the next error message in the listing. Correct the error, by changing `helo` to `hello`.
Let us assume this was the only error, so no further corrections are necessary. Leave the editor and resume the compilation by entering the empty command. After the compilation is finished an object file for the program module *hello* exists. To create an executable program, enter

```
>> p hello
```

The binder links the object files, in this case the object files of the program module *hello*, and the module *InOut*, the root module (not visible to the user), and the runtime system. Execute the created program by entering

>> `hello`

It should be noted that the above `c hello` command was redundant, because the make facility would have detected that there was no object file for *hello* when the `p hello` command was entered, and therefore would have invoked the compiler automatically.

B Example 2: Using the Make Facility

In this example we explain the usage and effects of the make facility implemented in the MOCKA system.
In Modula-2, a definition module imported by a compilation unit must have been compiled previously. A definition module must be compiled before its corresponding implementation module. The recompilation of a definition module invalidates the corresponding implementation module and all units importing that module. Recompilation of an implementation or program module invalidates no other units, but the program has to be linked again. Invalidated compilation units must be recompiled in an order corresponding to the dependencies between those units. The dependencies are introduced through imports. The make facility computes those dependencies and initiates the necessary recompilations automatically.
To illustrate the behaviour of the make facility let us consider a sample program consisting of several compilation units. Beyond the purpose to introduce module dependencies the sample program has no meaning (so the sources contain neither declarations nor statements). The sources of the sample program are:

```
DEFINITION MODULE black;
END black.

IMPLEMENATION MODULE black;
END black.

DEFINITION MODULE brown;
END brown.

IMPLEMENATION MODULE brown;
END brown.

DEFINITION MODULE blue;
  IMPORT black;
END blue.

IMPLEMENATION MODULE blue;
END blue.

DEFINITION MODULE green;
END green.

IMPLEMENATION MODULE green;
  IMPORT black;
  IMPORT brown;
END green.

MODULE red;
  IMPORT blue;
  IMPORT green;
BEGIN
  (* ... *)
END red.
```

Figure 1 shows the dependency graph of the sample program (forget about the thin entities for the moment). Compilation units are represented as boxes. The box representing an implementation module is partially hidden by the box representing the corresponding definition module. This means that the declarations made in a definition module are visible in the corresponding implementation module and therefore a dependency is introduced implicitly. Explicit import is represented by arrows. E.g., an arrow from the box representing the implementation module *green* to the box representing the definition module *black* means that the implementation module *green* imports module *black*.
Each implementation module depends on the corresponding definition module. The implementation module *green* depends on the definition modules (symbol files) of *black* and *brown*. The definition module *blue* (and because of the transitivity the corresponding implementation module either) depends on the definition module *black*. The program module *red* depends directly on the definition modules *blue* and *green*, and (because of

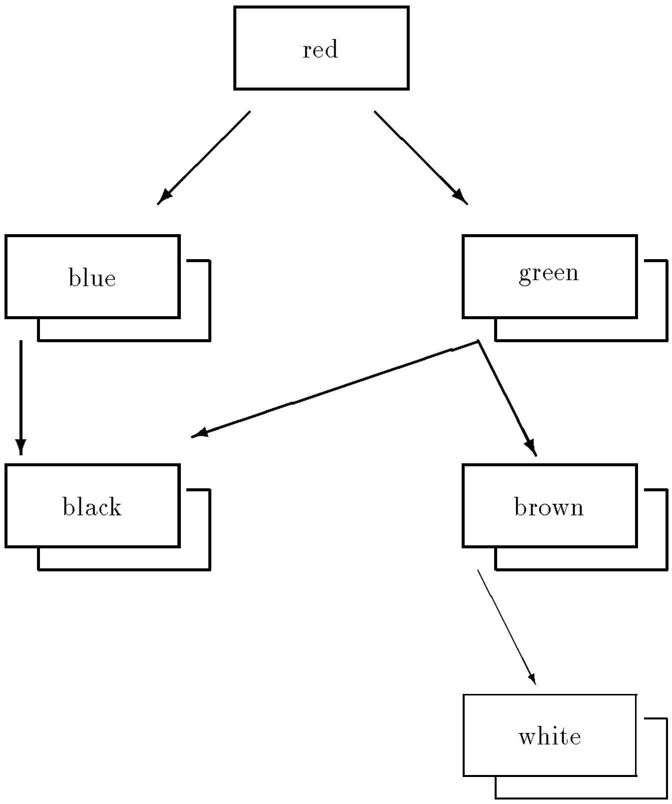

Figure 1: module dependencies of a sample program

the transitivity) on *black* and *brown*. Suppose that the actual working directory contains all the sources of the sample program. The corresponding symbol and object files are not present neither in the working directory nor in one of the libraries nor in the system library. So the invocation of the MOCKA system

```
$ mc
```

and the specification of the goal

```
>> p red
```

results in the compilation of each unit that is part of the sample program in an order computed automatically. Note that there are several orders possible, e.g.

```
definition module green
definition module brown
implementation module brown
definition module black
implementation module black
implementation module green
definition module blue
implementation module blue
program module red.
```

After completion of compilation and binding an executable program named *red* exists.
During the further development of the program modules will be modified, e.g. the definition module *black*:

```
>> d black
```

Modify *black.md*, e.g. by insertion of

```
  CONST c=0;
```

After leaving the editor the program is no longer consistent, because the still existing symbol file of *black* is the result of the compilation of an older version of definition module *black*. The implementation module *black* is invalidated, and also the definition module *blue* and the implementation module *green*. Furthermore, the transitive closure of invalidated modules contains the implementation module *blue* and the program module *red*, because they depend on modules that were invalidated. So, resuming of processing (enter the empty command) will lead to recompilation of

```
definition module black
implementation module black
implementation module green
definition module blue
implementation module blue
program module red
```

e.g. in this order.
During the development of a program system it often happens that new modules have to be incorporated, in our example the module *white*:

```
DEFINITION MODULE white;
END white.

IMPLEMENATION MODULE white;
END white.
```

Let the implementation module *brown* import objects exported by *white*, thus insert `IMPORT white;` Figure 1 (now including the thin entities) illustrates the new situation.
Resuming of processing will start the compilation of the definition and implementation module *white* (because neither symbol nor object file for *white* existed before) and the recompilation of the implementation module *brown*, because the source was modified and therefore the corresponding object file has become invalid.

C Example 3: Foreign Modules

The example presented here gives an idea how foreign modules can be used to access procedures not written in Modula-2. The example implements (parts of) a simple numeric I/O based on the C library. Let us consider the following C source:

```
/* implementation NumericIO.c */

void WriteCard (unsigned long card)
{
printf("%d",card);
}

void WriteLn ()
{
printf("\n");
}

void WriteCardArray (unsigned long cardarr[], int high)
{
while(high>=0) {
   WriteCard(cardarr[high--]);
   WriteLn();
   }
}
```

The Modula-2 declarations of those C functions are contained in a foreign module say with the name *NumericIO.md.*

```
FOREIGN MODULE NumericIO;
   PROCEDURE WriteCard(c : CARDINAL);
   PROCEDURE WriteLn();
   PROCEDURE WriteCardArray(VAR a : ARRAY OF CARDINAL);
END NumericIO.
```

Note, how parameters are passed. MOCKA passes an open array as address and last index (= HIGH(a)). This has to be considered in *NumericIO.c*, where WriteCardArray has an additional parameter `high`. To use the FOREIGN MODULE *NumericIO.md* you just have to compile the C source file and place the resulting object file in a path where MOCKA looks for it (the current directory or directories specified by the `-d` or `-D` option). The command sequence might look like this :

```
cc -c NumericIO.c
```

You can now use the module *NumericIO* like an ordinary MOCKA module.

D Example 4: A System Installation

The options can be used to embed the MOCKA system into various environments. An example is given here that usually fits into each Unix environment. Note that pathnames and command arguments might differ from system to system.
Let the executable code of the MOCKA system be contained in the file */usr/local/lib/mocka/sys/Mc*. The Unix shell script *mc* that invokes the MOCKA system with default parameters could look like this:

```
MOCKADIR = /usr/local/lib/mocka
COMPILER = Mc

export MOCKA_DIR
$MOCKADIR/sys/$COMPILER      \
  -link \$MOCKADIR/sys/link  \
  -edit \$MOCKADIR/sys/edit  \
  -list \$MOCKADIR/sys/list  \
  -asm  \$MOCKADIR/sys/asm   \
  -syslib \$MOCKADIR/lib     \
  -index                     \
  -range                     \
  $*
```

The link script is contained in the file *$MOCKADIR/sys/link* and could contain the shell script:

```
program=$1 ; shift ; modules="$*"
/bin/ld \
   -g  \
   -o $program \
   /usr/lib/crt0.o \
   $MOCKADIR/sys/M2RTS.o  $modules \
   -lc
```

/bin/ld contains the system linker *ld*, */lib/crt0.o* the C runtime system, *M2RTS.o* the root module generated by the MOCKA binder (and deleted immediately after linking is finished), the shell variable **$program** contains (during execution of the script) the name of the executable object file to be created by the linker, the shell variable **$modules** contains the names of the object files of the modules that constitute the program. *-g*, and *-lc* are arguments for the system linker *ld*.
The edit script is contained in file *$MOCKADIR/sys/edit* and could contain the shell script:

```
vi $1
```

The file *vi* contains the system editor that is invoked when a `d` or `i` command is entered in session mode.
The list script is contained in file *$MOCKADIR/sys/list* and could contain the shell script:

```
$MOCKADIR/sys/Lister $1
vi LISTING
$MOCKADRIR/sys/Unlister $1
```

The file *$MOCKADIR/sys/Lister* contains the listing generator that is invoked during session mode if errors are detected during a compilation. The listing generator merges the source file and the file containing the error messages (file *modulename.md_errors* resp. *modulename.mi_errors* located in the actual working directory) and generates the listing file *LISTING* (also located in the actual working directory). The editor stored in file *vi* is invoked with the listing file so that the errors can be corrected in the listing. Afterwards the unlister contained in file *$MOCKADIR/sys/Unlister* removes the error messages from the listing file and copies the listing to the source file.

E Example 5: Use of Compiler Pragmas in a Source Program

The following example shows how to use compiler pragmas to control special aspects of compilation. It should be noted that pragmas should be used very carefully.
The first pragma enables code generation for runtime range checking for the whole compilation unit, because the pragma is declared at level 0. Therefore, range checking is enabled for the module body and the body of *p1*. The second and third pragma make the first pragma inefficious and disable range checking in *p2* and *p3*. The fourth pragma makes the third pragma inefficious, so range checking is enabled for the body of *q1*.

```
MODULE PragmaDemo;
% range                          (* 1st pragma *)
  VAR x: CARDINAL; y: INTEGER;
  PROCEDURE p1;

  BEGIN

    x := y;                      (* => range check *)

  END p1;
  PROCEDURE p2;

  % norange                      (* 2nd pragma *)

  BEGIN

    x := y;                      (* => no range check *)

  END p2;
  PROCEDURE p3;

  % norange                      (* 3rd pragma *)
    PROCEDURE q1;

    % range                      (* 4th pragma *)

    BEGIN (* q1 *)

      x := y;                    (* => range check *)

    END q1;
  BEGIN (* p3 *)

    x := y;                      (* => no range check *)

  END p3;
BEGIN (* PragmaDemo *)

  x := y;                        (* => range check *)

END PragmaDemo.
```

References

[Wir85] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, Berlin, Heidelberg, New York, Tokyo, third, corrected edition, 1985.