

MOBIL(-P)
Intermediate Compiler Languages for
(Explicit Parallel) Imperative Languages

Internal Report

Jürgen Vollmer¹
FriWi Schröer²
Gesellschaft für Mathematik und Datenverarbeitung

August 20, 1992³

¹GMD Research Group at the University of Karlsruhe, Vincenz-Prießnitz-Straße 1, D-7500 Karlsruhe 1,
Phone: +/49/721/6622-14, email: vollmer@karlsruhe.gmd.de

²GMD FIRST, D-1199 Berlin, Rudower Chaussee 5, +49/30/6704-4343, email: friwi@first.gmd.de

³Rev: 2.1, Status: released

Abstract

Mobil and its extension *Mobil-P* are the low level intermediate compiler languages of the *Mocka / Mocka-P* *Modula-2 / Modula-P* compilers developed at the GMD in Karlsruhe. The programming language *Modula-P* a superset of *Modula-2* offers explicit parallelism on the language level, based on Hoare's *communication sequential processes*. *Mobil* and *Mobil-P* form the interface between the compiler front end (syntactic / semantic analysis and transformation) and the code generator for a specific target processor. The semantics of *Mobil* and *Mobil-P* are given in terms of an interpreter for the *Mobil / Mobil-P* instructions.

Contents

1	Introduction	3
1.1	Characteristics of Mobil and Mobil-P	3
1.2	Introduction into Mobil and Mobil-P	3
1.2.1	Operators	4
1.2.2	Attributes	4
1.2.3	Arguments and result	4
1.2.4	Machine data types	5
1.2.5	The Mobil memory model	5
1.2.6	Views of Mobil	6
2	Definition of Mobil	7
2.1	The Mobil interpreter	7
2.2	The Mobil instructions	7
2.2.1	Declarations	8
2.2.2	General operations	8
2.2.3	The Mobil Constants	10
2.2.4	Structured Constants	10
2.2.5	Address computation	10
2.2.6	Compiler generated variables	13
2.2.7	Memory access	13
2.2.8	Memory access and arithmetic	13
2.2.9	Integer arithmetic	14
2.2.10	Real arithmetic	15
2.2.11	Set arithmetic	15
2.2.12	Misc conversions	16
2.2.13	Comparisons	17
2.2.14	Control flow	18
2.2.15	Procedure call and parameter passing	20
2.3	Generating Mobil by MOCKA	21
2.3.1	Mobil Grammar	21
2.3.2	Procedure call	22
2.3.3	Parameter passing	23
2.3.4	Procedure nesting	23
3	Definition of Mobil-P	24
3.1	The Mobil-P interpreter	24
3.2	The Mobil-P instructions	25
3.2.1	Declarations	25
3.2.2	Channel instructions	26
3.2.3	Timer instructions	27
3.2.4	Parallel statements	28
3.2.5	Replication	29
3.2.6	The ALT statement	30
3.3	Generating Mobil-P by MOCKA-P	31

3.3.1	The Mobil-P grammar	31
3.3.2	Parallel statements	32
3.3.3	The ALT statement	32
3.4	Transputer machine instructions	32
	Bibliography	33

Chapter 1

Introduction

1.1 Characteristics of Mobil and Mobil-P

Mobil-P [Vollmer 89a] is the low level intermediate language of the *Modula-P* [Vollmer 89b, Vollmer *et al* 92] compiler system *Mocka-P*. *Mocka-P* itself is based on the *Modula-2* [Wirth 85] compiler system *Mocka*¹ [Schröder 88a], with the intermediate language *Mobil*² [Schröder 88b].

The compilation process of *Mocka* is divided into three phases: The parser constructs an abstract syntax tree, using the intermediate language *ASTA*. After the semantic analysis has taken place, this is transformed into the intermediate language *Mobil*³. A code generator translates a *Mobil* program into an assembly language program.

Several code generators for *Mobil* and *Modula-2* has been produced, either by hand writing (MC68K, VAX, Transputer) or generating using the *BEG* code generator generator tool [Emmelmann *et al* 89] (MIPS, SPARC, MC68K, Transputer). The flexibility of the *Mobil* intermediate language has been proven by extending it to *Mobil-P* and by adding *Mobil* back ends to a different (*Pascal*) front end.

Mobil has three characteristics, it is:

1. low level,
2. machine independent,
3. fully typed.

Low level means that the block structure of the source language is flattened, loops are translated into (un) conditional jumps, boolean expressions are mapped into jump cascades for implementing the short cut evaluation semantics of *Modula-2*. The access path for variables and memory access is made completely explicit. Local modules are also flattened and the body is transformed to an ordinary procedure, called automatically when the body of the enclosing module is called. All bodies of implementation modules are called according to the *Modula-2* rules, when initiating the main program.

Machine independence is achieved by using machine instructions in a three address format and having only the notion of data and address operands, which may be seen as an unbound set of abstract registers. Also a simple but general storage model is used.

Typed is *Mobil* in the sense that it knows several scalar machine types like short and long signed quantities, but their actual sizes is not specified by *Mobil* (but by supporting procedures). Also for each operand of an operator the type must be given.

1.2 Introduction into Mobil and Mobil-P

A *Mobil* module is a sequence of procedure definitions enclosed in *BeginModule* and *EndModule*. A procedure definition is a sequence of instructions enclosed in *BeginProcedure* and *EndProcedure*.

¹Modula-2 COmpiler KARlsruhe

²Modula-2 Backend Intermediate Language

³sometimes *Mobil* stands for both: *Mobil* and *Mobil-P*.

A *Mobil* instruction has the form:

OPERATOR ATTRIBUTES ARGUMENTS RESULT.

All items except the first may be missing.

A *Mobil* instruction is issued by the compilers front end using a procedure call

OPERATOR (ATTR₁, ..., ATTR_m, ARG₁, ..., ARG_n, RESULT).

1.2.1 Operators

The *OPERATOR* together with *ATTRIBUTES* the define the action to be performed.

There are two kinds of operators:

Declarations introduce unique identifications for program objects like modules, procedures, code labels, string addresses, etc. These unique identifications are used subsequently in the *Mobil* program to access these objects. Each object must be declared before it is used.

Actions Are the instructions of the abstract machine. They either return a single result (output) or not. In the latter case they are called *Mobil statements*.

1.2.2 Attributes

Attributes specify properties of the *Mobil* operators and operands, like their type, external value, or size of data entities.

There are several kinds of attributes:

Mode specifies the type of an operand.

Integers and **cardinals** are used to specify things like sizes and alignments of storage entities.

Relation is used to specify the relation of a comparison operator.

```
TYPE Relation = (RelEqual
                ,RelUnequal
                ,RelLess
                ,RelLessOrEqual
                ,RelGreater
                ,RelGreaterOrEqual
                );
```

ModuleKind TYPE ModuleKind = (ProgramModule, ImplementationModule, ProcessModule), specifies the kind of module currently compiled.

Labels are used as symbolic addresses of a specific piece of code.

ModuleIndex , **ProcedureIndex**, and **StringIndex** are used to identify modules, procedures and string constants.

and some more. The attributes and their meaning are described together with the operator that uses them.

1.2.3 Arguments and result

ARGUMENTS is a list of operands computed as output of previous instructions and used here as input. *RESULT* is the operand computed by the instruction. Each operand is defined exactly once as a result and used exactly once as an argument. Hence, *Mobil* instructions for expressions are called in postfix order by the front end.

Mobil distinguishes between two classes of operands: *Data operands* and *Address operands*. Data operands are used to hold values of different machine types that are fetched from memory using a *Content* instruction or that are result from some computation.

Address operands provide access to memory locations. Pointer values are considered as data, they can be retyped to serve as addresses.

1.2.4 Machine data types

Mobil knows the following machine data operands types, also called *Modes*:

- UnsignedByte
- UnsignedWord
- UnsignedLong
- SignedByte
- SignedWord
- SignedLong
- FloatShort
- FloatLong

Each data object, whether scalar or structured, has two special properties:

1. its size, and
2. an alignment.

Size gives the number of bytes needed to store a value of this type. This usually also defines the range of values representable by that data type. The alignment specifies the kind of addresses, at which a value may be stored in memory. The alignment requirement is usually a small number out of the set $\{1,2,4,8,16\}$. An alignment of 4, for example, specifies, that these data objects may be stored in memory only on addresses, which are divisible by 4 without of a remainder. More formally: $MemAdr(x) \text{ MOD } Alignment = 0$, where x is of type *Mode*.

The alignment requirement for structured source language types is usually the maximum of the component types.

The mapping from source language types to machine language types, size and alignment is specified by a set of procedures and constants, which are part of a code generator description for a given hardware. For example, the *Modula-2* front end and the Transputer (T 800) back end perform the following mapping:

Modula data type	value range	<i>Mobil</i>	Transputer	
			size	alignment
BOOLEAN	0 = FALSE 1 = TRUE	UnsignedByte	1 byte	1
CHAR	0 .. 255 coded as ASCII	UnsignedByte	1 byte	1
enumeration	0 .. 255 elements	UnsignedByte	1 byte	1
	256 .. $2^{16} - 1$	UnsignedWord	4 byte	2
	$2^{16} .. 2^{32} - 1$	UnsignedLong	4 byte	4
BITSET	0 .. 31	UnsignedLong	4 byte	4
SHORTCARD	0 .. $2^{16} - 1$	UnsignedWord	4 byte	4
CARDINAL	0 .. $2^{32} - 1$	UnsignedLong	4 byte	4
LONGCARD	0 .. $2^{32} - 1$	UnsignedLong	4 byte	4
SHORTINT	$-2^{15} .. 2^{15} - 1$	SignedWord	4 byte	4
INTEGER	$-2^{31} .. 2^{31} - 1$	SignedLong	4 byte	4
LONGINT	$-2^{31} .. 2^{31} - 1$	SignedLong	4 byte	4
POINTER	$-2^{31} .. 2^{31} - 1$	SignedLong	4 byte	4
ADDRESS	$-2^{31} .. 2^{31} - 1$	SignedLong	4 byte	4
REAL		FloatShort	4 byte	4
LONGREAL		FloatLong	4 byte	4

1.2.5 The Mobil memory model

The memory of the abstract *Mobil* machine is partitioned into *frames*.

For each separately compiled (source language) module there is a *module frame* (*STATICFRAME [module]*). The frame contains the static variables of the module. Objects in that frame are accessed by a *StaticVariable* instructions which specifies the frame of the module and offset of the object in the frame.

For each procedure there is a *local variable frame* (*VARFRAME*). The frame contains the local variables of the procedure. Objects in that frame are accessed by a *LocalVariable* instruction which specifies the offset in the

frame. If the variable belongs to a frame of a procedure different from the current one (i.e. a procedure statically surrounding the current one in the original source program), it is accessed by a *FrameBase* instruction, which specifies the static nesting level of the procedure and returns the address of that frame. The *GlobalVariable* instruction uses the returned variable frame address and the variable's offset in that frame to return the address of the variable.

For each procedure there is a *parameter frame* (*PARAMFRAME*). The frame contains the parameters passed to the procedure. It is established by the caller and filled using *Pass* instructions. Objects in that frame are accessed using *LocalParam* instructions (corresponding to *LocalVariable* instructions). If the object belongs to the parameter frame of a procedure different from the current one is, it is accessed using a *ParamBase* instruction (corresponding to the *FrameBase* instruction) and *GlobalParam* instruction using the frame address. Compiler created variables are called *tempos*. There are two classes of *Tempos*: *DataTempos* may be used to store data, *AddressTempos* to store addresses. Their scope is bound to the procedure, they are declared in. Structured values of the source language are mapped to sequences of scalar machine types. For each field the alignment requirements of the fields type must be fulfilled⁴. The entire data object is accessed by the address of its first field. The other fields are accessed relative to this first object. These offsets are either computed at compile time (records, and arrays with constant indices) by *AddOffset* or at runtime (arrays) by *Subscript*.

1.2.6 Views of Mobil

Two viewpoints may be taken concerning *Mobil*. First, it may be seen as an instruction set of an abstract machine having an arbitrary number of registers. Second, a *Mobil* program is a forest (sequence) of expression trees. Operands represent edges, instructions are the nodes of the tree. *Mobil* statements form the root of a tree. Instructions without arguments from the tree leaves.

The first view leads to a very easy implementation of a code generator: just expand each *Mobil* instruction into some target processor instructions and map operands to real target registers, which are allocated "on the fly". The second view is more appropriate for a *Mobil* optimizer, which transforms a *Mobil* program into a "better" one, or for a more sophisticated code generator (like *BEG*) which does tree pattern matching to generate less expensive code for several tree nodes (*Mobil* instructions) together.

⁴There are no "packed" data types.

Chapter 2

Definition of Mobil

2.1 The Mobil interpreter

The semantics of instructions is described by specifying an interpreter for *Mobil*. The interpreter uses the following data structures:

PC (program counter) refers to the *Mobil* instruction to be executed next.

VARBASE [i] is the base address of the VARFRAME for the procedure at static nesting level *i*.

PARMEBASE [i] is the base address of PARAMFRAME for the procedure at static nesting level *i*.

CALLBASE is the address of a parameter frame that is currently used to pass parameters.

FUNRES is the result of the last function call.

NEST is the static nesting level of the current procedure.

M is the untyped memory of the machine. $M [adr;n]$ denotes a slice of *n* bytes starting at address *adr*. The operation *ALLOCATE (adr, size)* creates a new slice in *M* with *size* and returns its address in *adr*.

STACK is used to save and restore administration data. *PUSH* and *POP* operations refer to the stack.

D[i] is the *i*-th *Data tempo* of the current procedure.

A[i] is the *i*-th *Address tempo* of the current procedure.

The *Mobil* machine knows two kinds of registers classes, data and address registers. The data registers may hold values of several types according to the *Mobil Modes*. There are an arbitrary number of such registers. They are single assignment and single use registers. *op.mode* describes, that the register is used with the given *mode*. Additionally *op.adr*, *op.pointer* specifies that the value is an address, *op.bitset* is interpreted as a *BITSET* value, mapped to some machine type.

The *Mobil* machine knows the usual (un)signed integer and floating arithmetic. Some operators are specified by describing them in terms of the corresponding *Modula-2* functionality.

2.2 The Mobil instructions

IN (OUT) specifies that this is an input (result) operand. *ATTR* indicates an attribute. The keyword *PROCEDURE* is used to mark *Mobil* declarations from the other instructions. Since they don't return a data or address operand, their results are marked with *VAR*. The inputs of declarations are not specially marked.

Attributes and arguments of *Mobil* instructions marked with a † are used only if the front end compiles a *Modula-P* program. For *Modula-2* they are not needed.

2.2.1 Declarations

All results of a declaration are unique for the current compiled compilation unit, except for the declaration of tempos. The scope of a tempo is bound the procedure declared it.

PROCEDURE DeclareModule	
extern:	BOOLEAN
† kind:	ModuleKind
CompUnitName:	ARRAY OF CHAR
VAR ref:	ModuleIndex
extern = TRUE, <i>iff the module is another imported compilation unit.</i>	

PROCEDURE DeclareProcedure	
extern:	BOOLEAN
isFunction:	BOOLEAN
ProcName:	ARRAY OF CHAR
ProcNumber:	SHORTCARD
module:	ModuleIndex
level:	SHORTCARD
father:	ProcIndex
VAR ref:	ProcIndex
extern = TRUE, <i>iff the procedure is imported from an other compilation unit.</i> isFunction = TRUE, <i>iff the procedure is a function.</i> Each procedure of an compilation unit has a unique ProcNumber. module specifies the module, the procedure is declared in. level specifies the nesting level of the procedure. Global procedures get level 0. If the procedure is declared local to another, father specifies that procedure.	

PROCEDURE DeclareString	
length:	SHORTCARD
string:	ARRAY OF CHAR
VAR ref:	StringIndex
length gives the number of significant characters of the string.	

PROCEDURE DeclareLabel	
VAR lab:	Label
<i>Labels are used for symbolic addresses of a piece of code.</i>	

PROCEDURE DeclareDataTempo	
mode:	Mode
VAR tempo:	DataTempo
<i>The front end may introduce temporary storage for compiler generated variables. The scope of a data tempo is bound to the current compiled procedure.</i>	

PROCEDURE DeclareAddressTempo	
VAR tempo:	AddressTempo
<i>The front end may introduce temporary storage for compiler generated variables. The scope of a address tempo is bound to the current compiled procedure.</i>	

2.2.2 General operations

BeginModule	
ATTR ModuleName:	ARRAY OF CHAR
ATTR VarSize:	LONGINT
ATTR † kind:	ModuleKind
<i>Indicates the beginning of a module. ModuleName is the name of the module. VarSize is the size of the module frame (in bytes).</i>	

EndModule	
<i>Indicates the end of module.</i>	

BeginProcedure	
ATTR index:	ProcIndex
ATTR level:	SHORTCARD
ATTR VarSize:	LONGINT
ATTR ParamSize:	LONGINT
<i>Indicates the beginning of a procedure. index is the index of the procedure as defined by a DeclareProcedure directive. level is the static nesting level of the procedure. VarSize is the size of procedure frame (in bytes). ParamSize is the size of the procedures parameter frame (in bytes).</i>	
<pre>PUSH (NEST); PUSH (PARAMBASE [level]); PUSH (VARBASE [level]); NEST := level; PARAMBASE [level] := CALLBASE; ALLOCATE (VARBASE [level], VarSize);</pre>	
EndProcedure	
<i>Indicates the end of a procedure.</i>	
CopyOpenArray	
ATTR DataOffset:	LONGINT
ATTR HighOffset:	LONGINT
ATTR elemsize:	LONGINT
ATTR † IsGlobalProcess:	BOOLEAN
<i>Initial treatment of "open array" value parameters (the instructions is issued for each open array value parameter at the beginning of the procedure). Open arrays are passed as as two parameters:</i>	
<i>(1) the address of the data vector and</i>	
<i>(2) the value of the HIGH function applied to the array.</i>	
<i>DataOffset is the offset of parameter (1) and HighOffset is the offset of parameter (2) in the parameter frame of the actual procedure. The instruction creates a copy of the data vector and changes the first parameter such that it points to the copy.</i>	
<i>†IsGlobalProcessBody = TRUE, iff this instruction is emitted in the body procedure of a PROCESS MODULE.</i>	
<pre>high, size : CARDINAL; source, target : address; high := M [PARAMBASE [NEST] + HighOffset; size (address)]; size := (high+1) * elemsize; source := M [PARAMBASE [NEST] + DataOffset; size (address)]; ALLOCATE (target, size); M [target; size] := M [source; size]; M [PARAMBASE [NEST]+ DataOffset; size (address)] := target;</pre>	
Mark	
ATTR line:	SHORTCARD
ATTR col:	SHORTCARD
<i>Passes the current source position to the Mobil program.</i>	
SkipData	SkipAddress
IN op: DataOperand	IN op: AddressOperand
<i>No action, "eats" not needed data values. Ignore the value of op.</i>	<i>No action, "eats" not needed address values. Ignore the value of op.</i>
SKIP;	SKIP;

2.2.3 The Mobil Constants

ShortCardConstant ATTR c: SHORTCARD OUT result: DataOperand <i>Returns the SHORTCARD constant c.</i> result.shortcard := c;	LongCardConstant ATTR c: LONGCARD OUT result: DataOperand <i>Returns the LONGCARD constant c.</i> result.longcard := c;
ShortIntConstant ATTR c: SHORTINT OUT result: DataOperand <i>Returns the SHORTINT constant c.</i> result.shortint := c;	LongIntConstant ATTR c: LONGINT OUT result: DataOperand <i>Returns the LONGINT constant c.</i> result.longint := c;
RealConstant ATTR c: REAL OUT result: DataOperand <i>Returns the REAL constant c.</i> result.real := c;	LongRealConstant ATTR c: LONGREAL OUT result: DataOperand <i>Returns the LONGREAL constant c.</i> result.longreal := c;
CharConstant ATTR c: CHAR OUT result: DataOperand <i>Returns the CHAR constant c.</i> result.char := c;	BoolConstant ATTR val: BOOLEAN OUT result: DataOperand <i>Returns the BOOLEAN constant c.</i> result.boolean := c;
SetConstant ATTR c: BITSET OUT result: DataOperand <i>Returns the BITSET constant c.</i> result.bitset := c;	NilConstant OUT result: DataOperand <i>Returns the POINTER constant NIL.</i> result.pointer := c;
ProcedureConstant ATTR index: ProcIndex OUT result: DataOperand <i>Returns a reference to the procedure given by index. This reference may be used to call the procedure or to assign it to a procedure variable or parameter.</i> result.label := PROCSTART (index);	

2.2.4 Structured Constants

StringAddr ATTR index: StringIndex OUT result: AddressOperand <i>Returns the address of a string constant designated by index.</i> result := STRINGADDR (index);
--

2.2.5 Address computation

StaticVariable ATTR module: ModuleIndex ATTR offset: LONGINT OUT result: AddressOperand <i>Returns the address of a variable located in a module frame. module is the corresponding module. offset is the offset of the variable in the frame.</i> result := STATICBASE [module] + offset;
--

<p>FrameBase</p> <p>ATTR proc: ProcIndex ATTR level: SHORTCARD OUT result: AddressOperand</p> <p><i>Returns a reference to a procedure frame. In the original source program the procedure was declared at nesting level level and enclosed the current one. Used to access a GlobalVariable.</i></p> <p>result := VARBASE [level];</p>
<p>ParamBase</p> <p>ATTR proc: ProcIndex ATTR level: SHORTCARD OUT result: AddressOperand</p> <p><i>Returns a reference to the parameter frame of a procedure. In the original source program the procedure was declared at nesting level level and enclosed the current one. Used to access a GlobalValueParam, GlobalVarParam or GlobalOpenArrayValueParam.</i></p> <p>result := PARAMBASE [level];</p>
<p>LocalVariable</p> <p>ATTR offset: LONGINT OUT result: AddressOperand</p> <p><i>Returns the address of a variable located in the frame of the current procedure. offset is the offset of the variable in the frame.</i></p> <p>result := VARBASE [NEST] + offset;</p>
<p>GlobalVariable</p> <p>ATTR offset: LONGINT IN frame: AddressOperand OUT result: AddressOperand</p> <p><i>Returns the address of a variable located in the procedure frame indicated by frame. offset is the offset of the variable in the frame.</i></p> <p>result := frame + offset;</p>
<p>LocalValueParam</p> <p>ATTR offset: LONGINT OUT result: AddressOperand</p> <p><i>Returns the address of a value parameter. The Parameter is located in the parameter frame of the current procedure. offset is the offset of the item in the frame.</i></p> <p>result := PARAMBASE [NEST] + offset;</p>
<p>LocalVarParam</p> <p>ATTR offset: LONGINT OUT result: AddressOperand</p> <p><i>Returns the address of a var parameter. The address is located in the parameter frame of the current procedure. offset is the offset of the item in the frame.</i></p> <p>result := M [PARAMBASE [NEST] + offset; size (address)];</p>
<p>GlobalValueParam</p> <p>ATTR offset: LONGINT IN base: AddressOperand OUT result: AddressOperand</p> <p><i>Returns the address of a value parameter. The parameter is located in the parameter frame frame. offset is the offset of the item in the frame.</i></p> <p>result := frame + offset;</p>

GlobalVarParam ATTR offset: LONGINT IN frame: AddressOperand OUT result: AddressOperand <i>Returns the address of a var parameter . The address is located in the parameter frame frame. offset is the offset of the item in the frame.</i> result := M [frame + offset; size (address)];
LocalOpenArrayValueParam ATTR offset: LONGINT OUT result: AddressOperand <i>Returns the address of a data vector passed as an "Open Array". The address is located in the parameter frame of the current procedure. offset is the offset of the item in the frame.</i> result := M [PARAMBASE [NEST] + offset; size (address)];
GlobalOpenArrayValueParam ATTR offset: LONGINT IN frame: AddressOperand OUT result: AddressOperand <i>Returns the address of a data vector passed as an "Open Array". The address is located in the parameter frame frame. offset is the offset of the item in the frame.</i> result := M [frame + offset; size (address)];
AddOffset ATTR offset: LONGINT IN BaseOp: AddressOperand OUT result: AddressOperand <i>Returns the address of a subobject. BaseOp is the address of the containing object. offset is the offset of subject inside the containing object.</i> result := BaseOp + offset;
Subscript ATTR IndexMode: Mode ATTR LwbMode: Mode ATTR UpbMode: Mode ATTR ElemSize: LONGINT IN BaseOp: AddressOperand IN IndexOp: DataOperand IN LwbOp: DataOperand IN UpbOp: DataOperand OUT result: AddressOperand <i>Returns the address of an array element. BaseOp is the address of the array, IndexOp is the index (with mode IndexMode). LwbOp and UpbOp specify lower and upper bound of the array (with modes LwbMode and UpbMode). ElemSize is the size of the array elements (in bytes).</i> result := BaseOp + (IndexOp.IndexMode - LwbOp.LwbMode) * ElemSize;
UsePointer IN op: DataOperand OUT result: AddressOperand <i>Returns the POINTER value op as address.</i> result := op.pointer;

2.2.6 Compiler generated variables

AssignDataTempo ATTR mode: Mode ATTR tempo: DataTempo IN op: DataOperand <i>Stores the value given by op in the temporary tempo. mode is the mode of op.</i> D [tempo] := op.mode;	AssignAddressTempo ATTR tempo: AddressTempo IN op: AddressOperand <i>Stores the address given by op in the temporary tempo.</i> A [tempo] := op;
UseDataTempo ATTR mode: Mode IN tempo: DataTempo OUT result: DataOperand <i>Returns the value stored in the temporary tempo. mode is the mode of tempo.</i> result.mode := D [tempo];	UseAddressTempo ATTR tempo: AddressTempo OUT result: AddressOperand <i>Returns the address stored in the temporary tempo.</i> result.mode := A [tempo];

2.2.7 Memory access

Assign ATTR mode: Mode IN lhs: AddressOperand IN rhs: DataOperand <i>Assigns the value given by rhs to the storage location given by lhs. mode specifies the mode of the value.</i> M [lhs; size (mode)] := rhs.mode;	AssignLong ATTR size: LONGINT IN lhs: AddressOperand IN rhs: AddressOperand <i>Assigns the value stored at the address given by rhs to the storage location given by lhs. size specifies the length of the value (in bytes).</i> M [lhs; size] := M [rhs; size];
Content ATTR mode: Mode IN op: AddressOperand OUT result: DataOperand <i>Returns the value stored at the address given by op. mode specifies the mode of the value.</i> result.mode := M [op; size (mode)];	

2.2.8 Memory access and arithmetic

Incl ATTR mode: Mode IN addr: AddressOperand <i>The value at the storage location given by addr is incremented by one. mode is the mode the object at address addr.</i> x : mode; x := M[addr; size(mode)]; M[addr;size(mode)] := x+1;	Inc2 ATTR mode: Mode IN addr: AddressOperand IN val: DataOperand <i>The value at the storage location given by addr is incremented by val. mode is the mode the object at address addr.</i> x : mode; x := M[addr.address; size (mode)]; M[addr.address;size(mode)] := x+val.mode;
---	---

Decl ATTR mode: Mode IN addr: AddressOperand <i>The value at the storage location given by addr is decremented by one. mode is the mode the object at address addr.</i> x : mode; x := M[addr.address; size(mode)]; M [addr.address; size(mode)] := x-1;	Dec2 ATTR mode: Mode IN addr: AddressOperand IN val: DataOperand <i>The value at the storage location given by addr is decremented by val. mode is the mode the object at address addr.</i> x : mode; x := M[addr.address; size (mode)]; M [addr.address; size(mode)] := x-val.mode;
Incl ATTR mode: Mode IN addr: AddressOperand IN val: DataOperand <i>mode IN Number. 0 < val < 31, all other is an error, is not checked.</i> <i>The value val is included into the BITSET value at the storage location given by addr. mode specifies the mode of val.</i> x : bitset; x := M[addr; size(bitset)]; M [addr; size(bitset)] := x+val.mode;	Excl ATTR mode: Mode IN addr: AddressOperand IN val: DataOperand <i>mode IN Number. 0 < val < 31, all other is an error, is not checked.</i> <i>The value val is excluded from the BITSET value at the storage location given by addr. mode specifies the mode of val.</i> x : bitset; x := M[addr; size(bitset)]; M [addr; size(bitset)] := x-val.mode;

2.2.9 Integer arithmetic

FixedNegate ATTR mode: Mode IN op: DataOperand OUT result: DataOperand <i>Returns the negated (unary minus) value of op. mode is the mode of the argument and the result.</i> result.mode := - op.mode;	FixedPlus ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the sum of op1 and op2. mode is the mode of the arguments and the result.</i> result.mode := op1.mode + op2.mode;
FixedMinus ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the result of subtracting op2 from op1. mode is the mode of the arguments and the result.</i> result.mode := op2.mode + op1.mode;	FixedMult ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the result of multiplying op1 and op2. mode is the mode of the arguments and the result.</i> result.mode := op1.mode * op2.mode;
FixedDiv ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the result of dividing op1 by op2. Integer division is used. mode is the mode of the arguments and the result.</i> result.mode := op1.mode DIV op2.mode;	FixedMod ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the remainder of dividing op1 by op2. mode is the mode of the arguments and the result.</i> result.mode := op1.mode MOD op2.mode;

FixedAbs
ATTR mode: Mode IN op: DataOperand OUT result: DataOperand <i>Returns ABS(op). mode is the mode of the argument and the result.</i> result.mode := ABS(op.mode);

2.2.10 Real arithmetic

FloatNegate	FloatPlus
ATTR mode: Mode IN op1: DataOperand OUT result: DataOperand <i>Returns the negated (unary minus) value of op. mode is the mode of the argument and the result.</i> result.mode := - op.mode;	ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the sum of op1 and op2. mode is the mode of the arguments and the result.</i> result.mode := op1.mode + op2.mode;

FloatMinus	FloatMult
ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the result of subtracting op2 from op1. mode is the mode of the arguments and the result.</i> result.mode := op2.mode + op1.mode;	ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the result of multiplying op1 and op2. mode is the mode of the arguments and the result.</i> result.mode := op1.mode * op2.mode;

FloatDiv	FloatAbs
ATTR mode: Mode IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand <i>Returns the result of dividing op1 by op2. mode is the mode of the arguments and the result.</i> result.mode := op1.mode / op2.mode;	ATTR mode: Mode IN op1: DataOperand OUT result: DataOperand <i>Returns ABS(op). mode is the mode of the argument and the result.</i> result.mode := ABS(op.mode);

2.2.11 Set arithmetic

SetUnion	SetDifference
IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand $x \in A \cup B \Leftrightarrow x \in A \vee x \in B$ <i>Returns the union of the BITSET operands op1 and op2.</i> result.bitset:=op1.bitset+op2.bitset;	IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand $x \in A - B \Leftrightarrow x \in A \wedge x \notin B$ <i>Returns of subtracting the BITSET operands op1 and op2.</i> result.bitset:=op1.bitset-op2.bitset;

SetIntersection IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand $x \in A \cap B \Leftrightarrow x \in A \wedge x \in B$ <i>Returns the intersection of the BITSET operands op1 and op2.</i> result.bitset:=op1.bitset*op2.bitset;	SetSymDifference IN op1: DataOperand IN op2: DataOperand OUT result: DataOperand $x \in A/B \Leftrightarrow (x \in A \wedge x \notin B) \vee (x \notin A \wedge x \in B) \Leftrightarrow x = A \text{ xor } B$ when A, B interpreted as bit vectors. <i>Returns the set symmetrical difference of the BITSET operands op1 and op2.</i> result.bitset:=op1.bitset/op2.bitset;
SetPlusSingle ATTR ElemMode: Mode IN SetOp: DataOperand IN ElemOp: DataOperand OUT result: DataOperand <i>Returns a BITSET, which is obtained by including the element ElemOp into the BITSET SetOp. ElemMode is the mode of ElemOp.</i> result.bitset:=SetOp.bitset+{ElemOp.ElemMode};	
SetPlusRange ATTR LwbMode: Mode ATTR UpbMode: Mode IN SetOp: DataOperand IN LwbOp: DataOperand IN UpbOp: DataOperand OUT result: DataOperand <i>Returns a BITSET, which is obtained by including the elements in the range [LwbOp ... UpbOp] into the BITSET SetOp. LwbMode is the mode of LwbOp, UpbMode is the mode of UpbOp.</i> result.bitset := SetOp.bitset + { x : LwbOp.LwbMode ≤ x ≤ UpbOp.UpbMode};	

2.2.12 Misc conversions

Cap IN op: DataOperand OUT result: DataOperand <i>lower case letters to upper case letters</i> <i>Returns CAP(op).</i> result.char := CAP(op.char);	Float IN op: DataOperand OUT result: DataOperand <i>converts CARDINAL value to a REAL value</i> <i>Returns FLOAT(op).</i> result.FloatShort:=FLOAT(op.UnsignedLong);
Trunc ATTR opmode: Mode ATTR resultmode: Mode IN op: DataOperand OUT result: DataOperand <i>converts REAL value to a CARDINAL value</i> <i>Returns FLOAT(op).</i> <i>opmode is the mode of op, resultmode is the mode of the result.</i> result.UnsignedLong:=FLOAT(op.FloatShort);	Adr ATTR arg: AddressOperand OUT result: DataOperand <i>Returns the address defined by op as DataTempo.</i> result.pointer := op

Coerce	
ATTR	premode: Mode
ATTR	postmode: Mode
IN	op: DataOperand
OUT	result: DataOperand
<i>Returns the value given by op, which has mode premode, converted into a representation with mode postmode.</i>	
result.postmode := op.premode;	
Check	
ATTR	IndexMode: Mode
ATTR	LwbMode: Mode
ATTR	UpbMode: Mode
ATTR	CheckLwb: BOOLEAN
ATTR	CheckUpb: BOOLEAN
IN	IndexOp: DataOperand
IN	LwbOp: DataOperand
IN	UpbOp: DataOperand
OUT	result: DataOperand
<i>Checks (LwbOp ≤ IndexOp) if CheckLwb is TRUE. Checks (IndexOp ≤ UpbOp) if CheckUpb is TRUE. Returns IndexOp as result.</i>	
IF CheckLwb AND NOT (LwbOp.LwbMode ≤ IndexOp.IndexMode) THEN ABORT END;	
IF CheckUpb AND NOT (IndexOp.IndexMode ≤ UpbOp.UpbMode) THEN ABORT END;	
result.IndexMode := IndexOp.IndexMode;	

2.2.13 Comparisons

FixedCompare	FloatCompare
ATTR mode: Mode	ATTR mode: Mode
ATTR rel: Relation	ATTR rel: Relation
IN op1: DataOperand	IN op1: DataOperand
IN op2: DataOperand	IN op2: DataOperand
OUT result: DataOperand	OUT result: DataOperand
<i>Compares op1 and op2 according to relation rel. Returns a BOOLEAN value indicating the result. mode is the mode of the arguments.</i>	<i>Compares op1 and op2 according to relation rel. Returns a BOOLEAN value indicating the result. mode is the mode of the arguments.</i>
result.boolean := op1.mode rel op2.mode;	result.boolean := op1.mode rel op2.mode;
PointerCompare	TestOdd
ATTR rel: Relation	ATTR mode: Mode
IN op1: DataOperand	ATTR cond: BOOLEAN
IN op2: DataOperand	IN op1: DataOperand
OUT result: DataOperand	OUT result: DataOperand
<i>Compares the POINTER values op1 and op2 according to relation rel. Returns a BOOLEAN value indicating the result</i>	<i>Tests whether ODD(op) evaluates to cond. Returns a BOOLEAN value indicating the result.</i>
result.boolean :=	result.boolean := ODD(op.mode) = cond;
op1.pointer rel op2.pointer;	

SetCompare	
ATTR	rel: Relation
IN	op1: DataOperand
IN	op2: DataOperand
OUT	result: DataOperand
<i>Compares the BITSET operands op1 and op2 according to relation rel. Returns a BOOLEAN value indicating the result. mode is the mode of the arguments.</i>	
<i>Meaning of "rel" attribute (A rel B) A, B : BITSET</i>	
<i>"=" : A equal B</i>	
<i>"#" : A not equal B $\Leftrightarrow (A - B) \neq 0$</i>	
<i>"<" : $A \subset B \Leftrightarrow (A \cap B) = A$</i>	
<i>">" : $B \subset A \Leftrightarrow (A \cap B) = B$</i>	
<i>"<=" : NOT (A > B) $\Leftrightarrow (A \cap B) \neq B$</i>	
<i>">=" : NOT (A < B) $\Leftrightarrow (A \cap B) \neq A$</i>	
result.boolean := op1.mode rel op2.mode;	
TestMembership	
ATTR	ElemMode: Mode
ATTR	cond: BOOLEAN
IN	elem: DataOperand
IN	set: DataOperand
OUT	result: DataOperand
<i>If cond is TRUE, it is tested whether the value given by elem is contained in the BITSET operand set. If cond is FALSE, it is tested whether elem is not contained in set. ElemMode is the mode of elem. Returns a BOOLEAN value indicating the result.</i>	
IF cond THEN result.boolean := elem.ElemMode IN set.bitset	
ELSE result.boolean := NOT (elem.ElemMode IN set.bitset)	
END;	

2.2.14 Control flow

PlaceLabel	Goto
ATTR lab: Label	ATTR target: Label
<i>Defines the current location as the target of a branch to the label lab.</i>	<i>Branches to target.</i>
	PC := target;
Switch	
ATTR mode: Mode	
ATTR lwb: LONGINT	
ATTR upb: LONGINT	
ATTR CaseLabels: ARRAY OF Label	
ATTR DefaultLabel: Label	
IN op: DataOperand	
<i>If the value given by op is in the range lwb .. upb the entry with index op - lwb of table CaseLabels is selected and a branch occurs to that label. Otherwise a branch occurs to the label DefaultLabel. mode is the mode of op.</i>	
IF lwb <= op.mode <= upb THEN PC := CaseLabels [op]	
ELSE PC := DefaultLabel END;	

<p>TestAndBranch</p> <p>ATTR cond: BOOLEAN ATTR target: Label IN op: DataOperand <i>Branches to target , if the value of the BOOLEAN operand op is equal to cond.</i> IF op.boolean = cond THEN PC := target END;</p>	<p>FixedCompareAndBranch</p> <p>ATTR mode: Mode ATTR rel: Relation ATTR target: Label IN op1: DataOperand IN op2: DataOperand <i>Compares op1 and op2 according to relation rel. mode is the mode the arguments. Branches to target if the test yields TRUE.</i> IF op1.mode rel op2.mode THEN PC := target END;</p>
<p>FloatCompareAndBranch</p> <p>ATTR mode: Mode ATTR rel: Relation ATTR target: Label IN op1: DataOperand IN op2: DataOperand <i>Compares op1 and op2 according to relation rel. mode is the mode the arguments. Branches to target if the test yields TRUE.</i> IF op1.mode rel op2.mode THEN PC := target END;</p>	<p>SetCompareAndBranch</p> <p>ATTR rel: Relation ATTR target: Label IN op1: DataOperand IN op2: DataOperand <i>Compares the BITSET operands op1 and op2 according to relation rel. Branches to target if the test yields TRUE. rel has the same meaning as in SetCompare.</i> IF op1.bitset rel op2.bitset THEN PC := target END;</p>
<p>PointerCompareAndBranch</p> <p>ATTR rel: Relation ATTR target: Label IN op1: DataOperand IN op2: DataOperand <i>Compares the POINTER values op1 and op2 according to relation rel. Branches to target if the test yields TRUE.</i> IF op1.pointer rel op2.pointer THEN PC := target END;</p>	<p>TestOddAndBranch</p> <p>ATTR mode: Mode ATTR cond: BOOLEAN ATTR target: Label IN op: DataOperand <i>Tests whether ODD(op) evaluates to cond. Branches to target if the test yields TRUE.</i> IF ODD(op.mode) = cond THEN PC target END;</p>
<p>TestMembershipAndBranch</p> <p>ATTR ElemMode: Mode ATTR cond: BOOLEAN ATTR target: Label IN elem: DataOperand IN set: DataOperand <i>If cond is TRUE , it is tested whether the value given by elem is contained in the BITSET operand set. If cond is FALSE , it is tested whether elem is not contained in set. ElemMode is the mode of elem. Branches to target if the test yields TRUE.</i> IF cond THEN IF elem.ElemMode IN set.bitset THEN PC := target END; ELSE IF NOT (elem.ElemMode IN set.bitset) THEN PC := target END; END;</p>	

2.2.15 Procedure call and parameter passing

<p>PreCall</p> <p>ATTR ParamSize: LONGINT</p> <p><i>Begins a procedure or function call operation. Initializes a parameter list (parameter frame). This list is called the actual list until a corresponding PostCall instruction follows. The list is extended by Pass instructions. The instructions surrounded by PreCall and PostCall may contain a nested call sequence. Inside the enclosed sequence the actual parameter list is defined by that sequence. ParamSize specifies the total size of data (in bytes) passed to the routine.</i></p> <p>PUSH (CALLBASE); ALLOCATE (CALLBASE, ParamSize);</p>
<p>PassValue</p> <p>ATTR mode: Mode</p> <p>ATTR offset: LONGINT</p> <p>IN op: DataOperand</p> <p><i>Copies the value of op to the actual parameter list. mode is the mode of the value. offset is the offset of the parameter in the parameter frame.</i></p> <p>M [CALLBASE + offset; size(mode)] := op.mode;</p>
<p>PassLongValue</p> <p>ATTR size: LONGINT</p> <p>ATTR offset: LONGINT</p> <p>IN op: AddressOperand</p> <p><i>Copies a long value to the actual parameter list. size specifies the size of the value (in bytes), offset is the offset of the parameter in the parameter frame. op denotes the address of the value.</i></p> <p>M [CALLBASE + offset; size] := M [op; size];</p>
<p>PassOpenArrayValue</p> <p>ATTR offset: LONGINT</p> <p>IN op: AddressOperand</p> <p><i>Copies the address of a data vector to the actual parameter list. offset is the offset of the address in the parameter frame. (Although the data vector is passed as a value parameter it is not here but inside the called procedure using a CopyOpenArray instruction.)</i></p> <p>M [CALLBASE+offset; size(address)] := op;</p>
<p>PassStringValue</p> <p>ATTR SourceLength: LONGINT</p> <p>ATTR TargetLength: LONGINT</p> <p>ATTR offset: LONGINT</p> <p>IN op1: AddressOperand</p> <p><i>Copies a string value to the actual parameter list. SourceLength specifies the length of argument, TargetSize specifies the length expected by the procedure. If SourceLength is less than TargetLength the argument string has to be extended. offset is the offset of the parameter in the parameter frame. op denotes the address of the string.</i></p> <p>M [CALLBASE + offset; TargetLength] := M [op; max(SourceLength, TargetLength)];</p>
<p>PassAddress</p> <p>ATTR offset: LONGINT</p> <p>IN op: AddressOperand</p> <p><i>Copies the address given by op to the actual parameter list. offset is the offset of the address in the parameter frame.</i></p> <p>M [CALLBASE + offset; size(address)] := op;</p>
<p>Call</p> <p>IN proc: DataOperand</p> <p><i>Invokes the procedure or function given by proc using the actual parameter list.</i></p> <p>PUSH(PC); PC := proc;</p>

SysCall	
ATTR sysproc: SysProc <i>Invokes a procedure of the Run Time System using the actual parameter list. sysproc specifies the procedure.</i> PUSH (PC); PC := SYSPROCSTART (sysproc);	
PostCall	
ATTR ParamSize: LONGINT <i>Ends a procedure or function call operation. ParamSize specifies the total size of data (in bytes) passed to the routine.</i> POP (CALLBASE);	
FunctionResult	
ATTR mode: Mode OUT result: DataOperand <i>Returns the result of the immediately preceding function call. mode is the mode of the function result.</i> result.mode := FUNRES;	
Return	ReturnValue
ATTR ParamSize: LONGINT <i>Exit from the current procedure. ParamSize specifies the total size of data (in bytes) passed to the routine.</i> POP (VARBASE [NEST]); POP (PARAMBASE [NEST]); POP (NEST); POP (PC);	ATTR mode: Mode ATTR ParamSize: LONGINT IN op1: DataOperand <i>Exit from the current function. Let op be the result of the function call. mode specifies the mode of the result. ParamSize specifies the total size of data (in bytes) passed to the routine.</i> FUNRES := op.mode; POP (VARBASE [NEST]); POP (PARAMBASE [NEST]); POP (NEST); POP (PC);

2.3 Generating Mobil by MOCKA

The following sections shows some specifics of the *Mocka* compiler, generating *Mobil* instruction for a *Modula-2* program.

2.3.1 Mobil Grammar

The following is a context free grammar, of how the *Modula-2* front end calls the *Mobil* instruction procedures. Notice that the instruction procedures are called in postfix order for expressions. **terminal** denotes terminal symbols, i.e. *Mobil* operators. [symbol] denotes zero or one occurrence, {symbol} zero or more occurrences of *symbol*, (and) are used for grouping symbols. The nonterminal symbols *Expr* denotes a *DataOperand* while *Adr* denotes an *AddressOperand*. The attributes of the operators are not shown.

```

MobilProgram ::= BeginModule {GlobalDecl | Procedure} EndModule.
GlobalDecl  ::= DeclareModule | DeclareProcedure.
Procedure   ::= BeginProcedure {CopyOpenArray} {Decl | Stmt} EndProcedure.
Decl        ::= DeclareDataTempo | DeclareAddressTempo | DeclareLabel | DeclareString.
Stmt        ::= Mark | PlaceLabel |
               Expr Adr Assign | Adr Adr AssignLong |
               Expr AssignDataTempo | Adr AssignAddressTempo |
               Adr Inc1 | Expr Adr Inc2 | Adr Dec1 | Expr Adr Dec2 |
               Expr Adr Incl | Expr Adr Excl |
               Expr SkipData | Adr SkipAddress |
               Goto | Expr TestAndBranch | Expr Switch |
               Expr Expr FixedCompareAndBranch | Expr Expr FloatCompareAndBranch |
               Expr Expr SetCompareAndBranch | Expr Expr PointerCompareAndBranch |
               Expr Expr TestMembershipAndBranch | Expr TestOddAndBranch |
               CallSequence | Return | Expr ReturnValue .
CallSequence ::= PreCall {PassParam} ProcCall PostCall.
ProcCall     ::= Expr Call [FunctionResult] | SysCall.
PassParam    ::= Expr PassValue | Adr PassLongValue | Adr PassOpenArrayValue |
               Adr PassStringValue | Adr PassAddress .
Expr         ::= ShortCardConstant | LongCardConstant | ShortIntConstant |
               LongIntConstant | RealConstant | LongRealConstant |
               CharConstant | BoolConstant |
               SetConstant | NilConstant | ProcedureConstant |
               UseDataTempo | Adr Content |
               Expr FixedNegate | Expr Expr FixedPlus | Expr Expr FixedMinus |
               Expr Expr FixedMult | Expr Expr FixedDiv | Expr Expr FixedMod |
               Expr FixedAbs |
               Expr Expr FloatPlus | Expr Expr FloatMinus | Expr Expr FloatMult |
               Expr Expr FloatDiv | Expr FloatAbs |
               Expr Expr SetUnion | Expr Expr SetDifference | Expr Expr SetIntersection |
               Expr Expr SetSymDifference | Expr Expr SetPlusSingle |
               Expr Expr Expr SetPlusRange |
               Expr Cap | Expr Float | Expr Trunc | Expr Adr | Expr Coerce |
               Expr Expr FixedCompare | Expr Expr FloatCompare | Expr Expr SetCompare |
               Expr Expr PointerCompare | Expr Expr TestMembership | Expr TestOdd |
               CallSequence .
Adr          ::= StringAdr | LocalVariable | Adr GlobalVariable | StaticVariable |
               LocalValueParam | LocalVarParam | LocalOpenArrayValueParam |
               Adr GlobalValueParam | Adr GlobalOpenArrayValueParam |
               Expr UsePointer | FrameBase | ParamBase | Adr AddOffset |
               Adr Expr Expr Expr Subscript | UseAddressTempo.

```

2.3.2 Procedure call

The *Mocka* compiler front end generates the *Mobil* instructions of a function procedure call intermixed with the expression's instructions.

For example, the *Mobil* instructions for the assignment and function call `x := 1 + f (2)`; are generated in the following sequence:

```

LongintConstant (1,op1);
PreCall (...);
LongintConstant (2, op2);
PassValue (op2,...);
ProcedureConstant ("f", op3);
Call (op3);
FunctionResult (... ,op4);
PostCall (...);
FixedPlus (... ,op1, op4, op5);
LocalVariable ("x", op6);
Assign (... ,op6, op5);

```

Having the forest of expression tree view of *Mobil*, a function procedure call in an expression is side effect free,

in the sense that the *Mobil* code for the function procedure call is not contained the expression's *Mobil* tree. The function's value is used through the *FunctionResult* instruction. The forest of the above example look like¹:

```

PreCall
PassValue
  LongintConstant (2)
Call
  ProcedureConstant ("f")
PostCall
Assign
  LocalVariable ("x", ..)
  FixedPlus
    LongintConstant (1)
    FunctionResult

```

In the tree view of *Mobil*, this fact has some bad consequences: If in an expression several functions are called, it is not clear, which *FunctionResult* refers to which function call. Hence the *Mobil* code must be rewritten, if a tree is constructed out of the in postfix order generated *Mobil* instructions: The operand *op1* generated by the *FunctionResult* (*..,op1*) instruction is replaced (while constructing the tree view) by the operand *op2* generated by the following instruction sequence:

```

DeclareDataTempo (.., t1);
AssignDataTempo (.., t1, op1);
UseDataTempo (.., t1, op2);

```

The operand *op2* is now used in the tree, instead of the original *op1*, returned by *Functionresult*.

This replacement is not needed in the postfix view since the *FunctionResult* operator follows immediately the *Call* instruction of the function which produces this result.

2.3.3 Parameter passing

The *Mocka* compiler generates the *Pass* instruction in a "right to left" fashion. The right most parameter of a source program procedure call is passed first, the left most is passed as the last parameter.

2.3.4 Procedure nesting

If procedure *Q* is declared local to procedure *P*, *Q* is processed by *Mocka* before *P*. This fact may be used for example, for not executing the interpreters *PUSH / POP (PARAMBASE [level])* and *PUSH / POP (VARBASE [level])* instruction in the *BeginProcedure / Return / Return Value* instructions, if *Q* doesn't use variables declared in *P*.

¹The root is printed on the left margin, tree children are printed below its parent with some indentation.

Chapter 3

Definition of Mobil-P

3.1 The Mobil-P interpreter

The abstract *Mobil-P* machine is a generalization of the *Mobil* interpreter. Its main extensions are the notion of *processes* and *channels*. The parent process of a process P is that process, which created P . The main program of a *Modula-P* program forms the “first” process of a program, it has obviously no parent process.

The *Mobil-P* interpreter uses the following data structures and operations:

PID Each process has a unique *process identification (PID)*. Each PID has several attributes, described below.

ME is a special PID: the PID of the process executing ME. Most data structures and interpreter operations have an parameter or qualifier PID. Omitting this parameter or qualifier always refers to the “current” process, i.e. reads as $pid = ME$.

pid.PARENT is the parent PID of the process pid ¹.

pid.PC refers to the *Mobil-P* instruction to be executed by process pid next.

Process pid executes the statements denoted by the PC , as long as $PC \neq 0$. A process stops its execution if $PC = 0$. Another process may assign it a value $\neq 0$ which causes the execution of that instruction (and possibly following instructions).

pid.REP_VAL denotes the value of the replicator variable, used by the replicated process pid . If the process pid is not replicated, this value is undefined.

CREATE_NEW_PROCESS returns a new unique PID and assigns it to ME of that new process. It also initializes the PC of the new process to 0, i.e. no statements are executed.

pid.CHILD denotes the PID returned by the last call to *CREATE_NEW_PROCESS* done by process pid .

pid.WAIT_FOR denotes the number of child processes of the process pid that have to be terminated, before it may execute the instruction following its *EndParallel* instruction.

pid.VARBASE[i], **pid.PARAMBASE[i]**, **pid.STATICBASE [module]**, **pid.CALLBASE**, **pid.FUNRES**, **pid.NEST**, **pid.D[i]**, **pid.A[i]** are defined as for the *Mobil* interpreter. These data structures are now local to each process. Omitting the i or *module* parameter denotes the entire data object used by that process.

pid.R[i] There is an additional class of compiler generated variables, called *replicator tempos*. Their scope is bound to the procedure declaring it. $pid.R[i]$ denotes the i -th replicator tempo. A replicator tempo consists of two fields: *count* and *value* ($pid.R[i].CNT$ and $pid.R[i].VAL$, respectively).

pid.ALT_SKIP is a boolean flag, set to true, iff all boolean expressions of an **ALT** statement are evaluated to *FALSE*.

¹the phrase *process pid* is an abbreviation for: *the process with PID pid*.

pid.ALT_GUARD_READY is a boolean flag, set to true, iff a guard of the **ALT** statement has become ready.

pid.ALT_WAITING is a boolean flag, set to true, if the process is now waiting for a guard to become ready, i.e. a *WaitForReadyGuard* instruction has been executed.

pid.ALT_SELECTED is a code address, referring to the alternative selected for execution.

M is the untyped memory of the machine. Notice: *M* as entire entity it is not local to any process.

CID Each channel has a unique *channel identification (CID)*. Each CID has several attributes, described below.

cid.READY is a boolean flag: *cid.READY = TRUE* specifies that a process has reached a communication statement for this channel and is now waiting until another process wants to communicate.

cid.ALT is a boolean flag: *cid.ALT = TRUE* specifies that a process (containing channel *cid* as a channel guard) is executing an **ALT** statement and waits for communication.

cid.MSG is the message send by a process. *cid.MSG* is undefined if *cid.READY = FALSE*. A process may send a message iff *cid.READY = FALSE* and read a message iff *cid.READY = TRUE*. Reading also implies *cid.READY := FALSE*.

cid.PID specifies the PID of the process reaching a communication statement (for channel *cid*) first. The first process suspends itself and is activated by the second one. *cid.PID* is defined iff *cid.READY = TRUE*.

cid.NEXT specifies the instruction of process *cid.PID* to be executed after activating it, i.e. the instruction after the communication statement, which caused its suspension. *cid.NEXT* is defined iff *cid.READY = TRUE*.

OPEN returns a new CID.

CLOCK is a clock (implemented outside of the interpreter). It has several attributes, described below.

CLOCK.SYSTIME returns the current system time. Notice, this must not be a global time for all processes.

CLOCK.ALT [pid] is a boolean value is true, iff process *pid* uses a time guard in an **ALT** statement.

CLOCK.DELAY [pid] is a boolean value is true, iff process *pid* has suspended itself for some time.

CLOCK.TIME [pid] if **CLOCK.ALT [pid]** or **CLOCK.DELAY [pid]** is **TRUE**, specifies that process *pid* has to be activated if **CLOCK.SYSTIME** is later than *CLOCK.TIME [pid]*.

CLOCK.NEXT[pid] if **CLOCK.ALT [pid]** or **CLOCK.DELAY [pid]** specifies the instruction to be executed, if the clock activates process *pid*.

The set of interpreter instructions given of one *Mobil-P* instruction are atomic, in the sense that at one time only one process may access attributes of the data structures, for example the channel attributes.

Notation example: *cid.PID.ALT_READY_GUARD* specifies the attribute *ALT_READY_GUARD* of the process *cid.PID*.

Some *Mobil* instructions are now in some aspect “process local”, e.g. *StaticVariable* now refers to *ME.STATICBASE [module] + offset*. But *ME.STATICBASE [module]* may be copied from the parent process (in case of a local *Modula-P* process) or is new allocated (in case of a global *Modula-P* process).

3.2 The Mobil-P instructions

3.2.1 Declarations

PROCEDURE DeclareReplicatorTempo

VAR tempo: ReplicatorTempo

<i>Declares a new replicator temporary variable. Its scope is bound to the current compiled procedure.</i>
--

3.2.2 Channel instructions

OpenChannel	
IN channel: AddressOperand <i>Opens the channel for communication.</i> cid : CID; cid := OPEN; cid.READY := FALSE; cid.ALT := FALSE; M[channel;size(CHANNEL)] := cid;	
Receive	ReceiveLong
ATTR mode: Mode IN channel: AddressOperand IN dest: AddressOperand <i>Receive a value with mode mode from channel channel and stores it in memory starting with address dest. If another process waits for sending a message, activate it. If no other process want to send a message, suspend ME and signal "receiver ready".</i> cid : CID; cid := M[channel;size(CHANNEL)]; IF cid.READY = TRUE THEN (* partner is ready and suspended *) cid.PID.PC := cid.NEXT; (* activate*) ELSE (* suspend me and signal *) cid.NEXT := CODEADDR (L); cid.READY := TRUE; PC := 0; (* suspend ME *) END ; L: M[dest;size(mode)] := cid.MSG; cid.READY := FALSE;	IN channel: AddressOperand IN dest: AddressOperand IN size: DataOperand <i>Receive size bytes from channel channel and stores them in memory starting with address dest. If another process waits for sending a message, activate it. If no other process want to send a message, suspend ME and signal "receiver ready".</i> cid : CID; cid := M[channel;size(CHANNEL)]; IF cid.READY = TRUE THEN (* partner is ready and suspended *) cid.PID.PC := cid.NEXT; (* activate *) ELSE (* suspend me and signal *) cid.NEXT := CODEADDR (L); cid.READY := TRUE; PC := 0; (* suspend ME *) END ; L: M[dest;size.longcard] := cid.MSG; cid.READY := FALSE;

Send	SendLong
<pre> ATTR mode: Mode IN channel: AddressOperand IN value: DataOperand <i>Sends a value with mode over the channel.</i> <i>If another process waits for receiving a message, activate it.</i> <i>If no other process want to receive a message, suspend ME and signal "sender ready".</i> cid : CID; cid := M[channel;size(CHANNEL)]; cid.MSG := op.mode; IF cid.READY = TRUE THEN (* partner is ready and suspended *) cid.PID.PC := cid.NEXT; (* activate *) ELSE (* suspend me and signal *) cid.READY := TRUE; IF cid.ALT = TRUE THEN (* check for waiting ALT *) cid.PID.ALT_READY_GUARD := TRUE; IF cid.PID.ALT_WAITING = TRUE THEN cid.PID.PC := cid.NEXT; (* activate *) END; END; cid.NEXT := CODEADDR (L); PC := 0; (* suspend ME *) END; L: (* next instruction *) </pre>	<pre> IN channel: AddressOperand IN source: AddressOperand IN size: DataOperand <i>Sends size bytes starting in memory from address src to the channel. If another process waits for receiving a message, activate it. If no other process want to receive a message, suspend ME and signal "sender ready".</i> cid : CID; cid := M[channel;size(CHANNEL)]; cid.MSG := M[source;size.longcard]; IF cid.READY = TRUE THEN (* partner is ready and suspended *) cid.PID.PC := cid.NEXT; (* activate *) ELSE (* suspend me and signal *) cid.READY := TRUE; IF cid.ALT = TRUE THEN (* check for waiting ALT *) cid.PID.ALT_READY_GUARD := TRUE; IF cid.PID.ALT_WAITING = TRUE THEN cid.PID.PC := cid.NEXT; (* activate *) END; END; cid.NEXT := CODEADDR (L); PC := 0; (* suspend ME *) END; L: (* next instruction *) </pre>

3.2.3 Timer instructions

GetSysTime	Delay
<pre> IN dest: AddressOperand <i>Reads the system time and stores it in memory.</i> M[dest;size(TIME)] := CLOCK.SYSTIME; </pre>	<pre> IN time: DataOperand <i>Delays the current process until the system time is later than the time specified by time.</i> CLOCK.DELAY[pid] := TRUE; CLOCK.TIME[pid] := time.time; CLOCK.NEXT[pid] := CODEADDR (L); PC := 0; L: CLOCK.DELAY[pid] := FALSE; </pre>

3.2.4 Parallel statements

<p>BeginParallel</p> <p>ATTR NextInstr: Label</p> <p><i>Indicates the beginning of a parallel statement. NextInstr indicates the statement to be executed after all child processes have been terminated.</i></p> <p>ME.WAIT_FOR = 1;</p>	<p>EndParallel</p> <p>ATTR NextInstr: Label</p> <p><i>Indicates the end of a parallel statement. The current process is suspended, until all child processes have terminated.</i></p> <p>ME.WAIT_FOR := ME.WAIT_FOR - 1;</p> <p>IF ME.WAIT_FOR > 0</p> <p>THEN (*there are non terminated children*)</p> <p> ME.PC := 0; (* stops execution now *)</p> <p>ELSE (* all children are terminated *)</p> <p> ME.PC := CODEADDR (NextInstr);</p> <p>END;</p>
<p>BeginProcess</p> <p><i>Indicates the start of the process body.</i></p>	
<p>EndProcess</p> <p>ATTR NextInstr: Label</p> <p><i>Indicates the end of the process body. nextInstr specifies the instruction to be executed, by the parent process after all child processes of it has been terminated.</i></p> <p>ME.PARENT.WAIT_FOR := ME.PARENT.WAIT_FOR - 1;</p> <p>IF ME.PARENT.WAIT_FOR = 0 THEN (* all children have terminated *)</p> <p> ME.PARENT.PC := NextInstr; (* activates parent *)</p> <p>END;</p> <p>ME.PC := 0;</p>	
<p>StartProcess</p> <p>ATTR processLab: Label</p> <p>ATTR replicated: BOOLEAN</p> <p>ATTR RepTempo: DataTempo</p> <p><i>Starts a child process. If it is a replicated process, the replicator variable of the child process gets its value assigned.</i></p> <p>ME.WAIT_FOR := ME.WAIT_FOR + 1;</p> <p>CHILD := CREATE_NEW_PROCESS;</p> <p>CLOCK.DELAY[CHILD] := FALSE;</p> <p>CLOCK.ALT[CHILD] := FALSE;</p> <p>CHILD.VARBASE := ME.VARBASE; (* copies var base *)</p> <p>CHILD.PARAMBASE := ME.PARAMBASE;</p> <p>CHILD.STATICBASE := ME.STATICBASE; (* for all modules *)</p> <p>IF replicated THEN CHILD.REP_VAL := R [RepTempo].VAL END;</p> <p>CHILD.PC := processLab; (* starts of child execution now *)</p>	

StartGlobalProcess	
IN	proc: DataOperand
IN	processorNr: DataOperand
ATTR	ParamSize: LONGINT
<i>proc denotes a ProcedureConstant, which denotes the body procedure of a process module</i>	
<i>The earliest point the all imported modules are known is linking or interpreting time.</i>	
<i>processorNr codes an information used by a runtime system supporting the real execution of Mobil-P program. Its intended meaning is the number of a processor the code has to be executed, or a strategy how to determine this number.</i>	
FORALL m \in {module (transitively) imported by process module <i>proc</i> } DO	
ALLOCATE (STATICBASE [m], static_var_size (m));	
END;	
PUSH(PC);	
ALLOCATE (CHILD.CALLBASE, ParamSize);	
CHILD.CALLBASE := ME.CALLBASE;	
PC := proc;	

3.2.5 Replication

InitReplication	
ATTR	RepTempo: DataTempo
ATTR	EndLab: Label
IN	lwb: DataOperand
IN	upb: DataOperand
<i>Initializes the replicator temporary variable.</i>	
R[RepTempo].CNT := ORD(upb.mode) - ORD(lwb.mode) + 1;	
R[RepTempo].VAL.mode := lpb.mode;	
R[RepTempo].CNT < 0 THEN PC := EndLab END;	

DoReplication	
ATTR	RepTempo: DataTempo
ATTR	StartLab: Label
ATTR	EndLab: Label
<i>Implements loop for replication.</i>	
R[RepTempo].CNT := R[RepTempo].CNT - 1;	
INC (R[RepTempo].VAL.mode);	
IF R[RepTempo] = 0 THEN PC := EndLab ELSE PC := StartLab; END;	

UseProcessRepVal	
ATTR	level_diff: CARDINAL
OUT	RepValue: DataOperand
<i>Access the value of the replicator variable of a replicated process. level_diff = 0: means the value of the replicator variable of this process. level_diff = n means the replicator variable with level_diff = n-1 of the parent, i.e. the "level_diff" grand parent.</i>	
RepValue.mode := ME.PARENT...PARENT.REP_VAL.mode;	
(* where PARENT occurs level_diff times. *)	

3.2.6 The ALT statement

<p>BeginAltInput</p> <p>ATTR ContainsTimer: BOOLEAN <i>Indicates the beginning of of a ALT statement. ContainsTimer is TRUE, iff an input of the TIMER is part of a guard of this ALT statement.</i></p> <p>ME.ALT_GUARD_READY := FALSE; ME.ALT_WAITING := FALSE;</p>	<p>EndAltInput</p> <p>ATTR ContainsTimer: BOOLEAN <i>Indicates the end of the enable / wait / disable sequence. ContainsTimer is TRUE, iff an input of the TIMER is part of a guard of this ALT statement. Jumps to the instructions of the selected alternative.</i></p> <p>PC := ME.ALT_SELECTED;</p>
<p>CheckBoolGuard</p> <p>ATTR check_tempo: DataTempo IN bool_val: DataOperand OUT result: DataOperand <i>The check, that at least one boolean expression of an ALT statement is true, is done incrementally, by a 'OR' of the value stored in check_tempo and the bool_val.</i></p> <p>D[check_tempo].bool := D[check_tempo].bool OR bool_val.bool; result.bool := bool_val.bool;</p>	<p>CheckAlt</p> <p>ATTR check_tempo: DataTempo ATTR else_label: Label <i>If the value stored in check_tempo is FALSE, then the code marked with else_label will be executed, after all alternatives are disabled. This includes, that the following alternative input statements are skipped. If this value is TRUE, the next instruction is executed.</i></p> <p>IF check_tempo.bool = FALSE THEN PC := CODEADDR (else_label); END;</p>
<p>WaitForReadyGuard</p> <p>ATTR WaitLab: Label ATTR ContainsTimer: BOOLEAN <i>If no guard is already ready, suspends the process and waits for a guard to become ready. If a guard is ready, execute the instruction marked with WaitLab. ContainsTimer is TRUE, iff an input of the TIMER is part of a guard of this ALT statement.</i></p> <p>IF ME.ALT_GUARD_READY = FALSE THEN ME.ALT_WAITING := TRUE; ME.PC := 0; ELSE ME.PC := CODEADDR (WaitLab); END;</p>	
<p>EnableSkip</p> <p>ATTR WaitLab: Label IN bool_expr: DataOperand <i>This guard is ready, if the bool_expr is true. WaitLab specifies the instruction to be executed, if the processes is activated (here it may be not needed).</i></p> <p>IF bool_expr.bool = TRUE THEN ME.ALT_GUARD_READY := TRUE; END;</p>	<p>DisableSkip</p> <p>ATTR target: Label IN bool_expr: DataOperand IF bool_expr.bool = TRUE THEN ME.ALT_SELECTED := CODEADDR (target); END;</p>

<p>EnableChannel</p> <p>ATTR WaitLab: Label IN bool_expr: DataOperand IN channel: AddressOperand</p> <p><i>If bool_expr evaluates to TRUE, informs the channel channel that it is used in an ALT statement; otherwise nothing happens. WaitLab specifies the instruction to be executed, if the processes is activated by a sender.</i></p> <pre>cid : CID; cid := M[channel;size(CHANNEL)]; IF bool_expr.bool = TRUE THEN cid.NEXT := CODEADDR (WaitLab); cid.ALT := TRUE; IF cid.READY = TRUE THEN (* partner is suspended*) ME.ALT_READY := TRUE; END; END;</pre>	<p>DisableChannel</p> <p>ATTR target: Label IN bool_expr: DataOperand IN channel: AddressOperand</p> <p><i>If bool_expr evaluates to TRUE and a sender is waiting the corresponding alternative will be selected. If no sender is ready, informs channel channel that it is no longer used in a guard. IF the bool_expr evaluates to FALSE, nothing happens.</i></p> <pre>cid : CID; cid := M[channel;size(CHANNEL)]; IF bool_expr.bool = TRUE THEN IF cid.READY = TRUE THEN ME.SELECTED := CODEADDR (target); END; cid.ALT := FALSE; END;</pre>
<p>EnableTimer</p> <p>ATTR WaitLab: Label IN bool_expr: DataOperand IN time_expr: DataOperand</p> <p><i>If the bool_expr evaluates to TRUE, and the system time becomes later than time_expr.time then this guard becomes ready. WaitLab specifies the instruction to be executed, if the processes is activated by the clock.</i></p> <pre>IF bool_expr.bool = TRUE THEN CLOCK.ALT[ME] := TRUE; CLOCK.NEXT[ME] := CODEADDR (WaitLab); CLOCK.TIME[ME] := time_expr.time.time; END;</pre>	<p>DisableTimer</p> <p>ATTR target: Label IN bool_expr: DataOperand IN time_expr: DataOperand</p> <p><i>If bool_expr evaluates to TRUE and the system time is later than the time time_expr than the corresponding alternative will be selected. IF the bool_expr evaluates to FALSE, nothing happens.</i></p> <pre>IF bool_expr.bool = TRUE THEN IF CLOCK.SYSTIME > time_expr.time THEN ME.SELECTED := CODEADDR (target); END; CLOCK.ALT := FALSE; END;</pre>
<p>BeginAlternative</p> <p><i>Indicates the beginning of an alternative</i></p>	<p>EndAlternative</p> <p>ATTR NextInstr: Label</p> <p><i>Indicates the end of the alternative. Jumps the instruction marked by NextInstr.</i></p> <pre>PC := CODEADDR (NextInstr);</pre>

3.3 Generating Mobil-P by MOCKA-P

The following sections shows some specifics of the *Mocka-P* compiler, generating *Mobil* instruction for a *Modula-P* program.

3.3.1 The Mobil-P grammar

Some syntax rules are extended:

```

Decl      ::= ... | DeclareReplicatorTempo.
Expr      ::= ... | UseProcessRepVal.
Stmt      ::= ... | Adr OpenChannel | Adr Adr Receive | Adr Adr Expr ReceiveLong |
              Expr Adr Send | Adr Adr Expr SendLong |
              Adr ReadTimer | Expr Delay |
              BeginAlt {Enable} Wait {Disable} EndAlt {Alternative} PlaceLabel |
              BeginParallel {CreateProcess} EndParallel {ProcessBody} PlaceLabel |
Enable     ::= [StartReplication] Expr CheckBoolGuard
              (EnableSkip | Adr EnableChannel | Expr EnableTimer)
              [EndReplication].
Wait       ::= CheckAlt WaitForReadyGuard.
Disable    ::= [StartReplication] Expr
              (DisableSkip | Adr DisableChannel | Expr DisableTimer)
              [EndReplication].
Alternative ::= BeginAlt [Receive | ReceiveLong] {Stmt} EndAlt.
CreateProcess ::= [StartReplication] StartProcess [EndReplication].
ProcessBody ::= PlaceLabel BeginProcess ({Stmt} | GlobalProcess) EndProcess.
GlobalProcess ::= PreCall {PassParam} StartGlobalProcess PostCall.
StartReplication ::= DeclareReplicatorTempo Expr Expr InitReplication PlaceLabel.
EndReplication  ::= DoReplication PlaceLabel.

```

3.3.2 Parallel statements

For a **PAR** statement the compiler front end emits the following additional instructions: For each process body and for the instruction following the **PAR** statement a *label* is declared. Replicated processes are implemented by surrounding *StartProcess* by a loop. For each replicated process a replicator tempo and two labels (for start and end of the loop) are declared.

3.3.3 The ALT statement

The **ALT** statement is translated into a sequence of *Enable* instructions. They inform the channel or timer, that they are used in a guard. Then the process must wait, until a guard is ready. After the process returns from waiting, the guards must be *disabled*, to inform them that they are no longer part of a guard. During disabling it is decided which alternative out of the set of ready ones is selected for execution. After the enable - wait - disable instructions the instructions of the alternative bodies is emitted. An Alternative ends by branching to the instruction following the **ALT** statement.

The interpreter forces a specific strategy (the last disabled alternative), but the real implementation may choose the alternative arbitrary.

For a **ALT** statement the compiler front end emits the following additional instructions: For each alternative body, for the instruction following the **ALT** statement, and for the *ELSE* part a *label* is declared. If the *ELSE* part is missing, instructions for calling the *AltError* system procedure are generated.

For checking the boolean expressions of guards one data tempo is declared and assigned to *FALSE*, it is used by the *CheckBoolGuard* and *CheckAlt* instructions. If the boolean expression is omitted in the source program, the front end inserts a *BooleanConstant* with value *TRUE*.

Replicated alternatives are implemented by surrounding the *Enable* / *Disable* instructions by a loop. For each replicated alternative a replicator tempo and two labels (for start and end of the loop) are declared.

The arguments of the (non-replicated) *Enable* / *Disable* instructions are computed once and then stored in tempos, which are created by the front end. For replicated alternatives the arguments of the *Disable* instruction are recomputed each time.

The first action of an alternative with a channel guard is to read message from the channel, then the code for the alternative follows.

3.4 Transputer machine instructions

To implement for a Transputer [INMOS 88a] based system runtime system efficiently, some of the basic Transputer instructions should be directly available in a *Modula-P* program. Since the *Mocka-P* system doesn't has an inline assembler, these Transputer instructions are provided by a module, known to the compiler, like

the *SYSTEM* module. The semantics of these Transputer instructions is defined in the Transputer manuals [INMOS 88a, INMOS 88b].

Transputer_OUT IN link: DataOperand IN size: DataOperand IN src: DataOperand <i>implements the out Transputer instruction.</i>	Transputer_OUTB IN link: DataOperand IN val: DataOperand <i>implements the outb Transputer instruction.</i>
Transputer_OUTW IN link: DataOperand IN val: DataOperand <i>implements the outw Transputer instruction.</i>	Transputer_IN IN link: DataOperand IN size: DataOperand IN dest: DataOperand <i>implements the in Transputer instruction.</i>
Transputer_MOVE IN source: DataOperand IN size: DataOperand IN dest: DataOperand <i>implements the move Transputer instruction.</i>	

Bibliography

- [Emmelmann *et al* 89] Helmut Emmelmann, F. W. Schröder, and Rudolf Landwehr. Beg – a generator for efficient back ends. *ACM SIGPLAN NOTICES*, 24(7):227–327, July 1989.
- [INMOS 88a] INMOS, editor. *The Transputer instruction set - a compiler writers' guide*. Prentice–Hall, Inc., 1988.
- [INMOS 88b] INMOS. *The Transputer reference manual*. Prentice–Hall, Inc., 1988.
- [Schröder 88a] F.W. Schröder. Das GMD Modula-2 Entwicklungssystem. *GMD-Spiegel*, 1/1988.
- [Schröder 88b] F.W. Schröder. Mobil: An intermediate language for portable optimizing compilers. draft of an unpublished internal paper, 1988.
- [Vollmer 89a] Jürgen Vollmer. Kommunizierende sequentielle Prozesse in Modula-2; Entwurf und Implementierung eines Transputer – Entwicklungssystems. Master's thesis, Universität Karlsruhe, May 1989.
- [Vollmer 89b] Jürgen Vollmer. Modula-P, a language for parallel programming. *Proceedings of the First International Modula-2 Conference October 11-13, 1989, Bled, Yugoslavia*, pages 75–79, 1989.
- [Vollmer *et al* 92] Jürgen Vollmer and Ralf Hoffart. Modula-P, a language for parallel programming; definition and implementation on a transputer network. In *Proceedings of the 1992 International Conference on Computer Languages ICCL'92, Oakland, California*, pages 54–64. IEEE, IEEE Computer Society Press, Los Alamitos, California, April 1992.
- [Wirth 85] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, Heidelberg, New York, third, corrected edition, 1985.