A VM program starts with a byte indicating the number of predicates P in the program. Next, there are several components:

- An unsigned integer indicating the number N of nodes to instantiate, followed by $2N$ unsigned integers corresponding to one pair of unsigned integers one for node. The first value is the node ID to use during execution and the second one the ID given by the user.
- A byte indicating the number t of types in the program, followed by the t type specifications.
- An unsigned integer indicating the number of arguments needed to run the program.
- An unsigned integer describing the number of rules R in the program. Followed by R byte regions. Each region contains an unsigned integer, N, indicating the size of the rule and then N bytes with the string for this rule.
- An unsigned integer indicating the number S of constant strings in the program followed by S pairs containing the length of the string and the string itself.
- A byte indicating the number of code constants C and then C bytes for the types of such constants. Finally, there's an unsigned integer describing the code size for computing the constants and the code itself.
- A set of P *predicate descriptors*, with 69 bytes each.
- A set of P byte-code instructions, one for each predicate.

A predicate descriptor consists of the following fields:

- A short integer indicating the size, in bytes, of the corresponding byte-code instructions.
- 1 byte describing the predicate's properties.
- 1 byte indicating the aggregate's type, if any. The high nibble if the aggregate type and the low nibble the aggregate field.
- A byte indicating the predicate's number of fields F.
- 32 bytes with information about the fields' types. Actually, only F bytes are used, and the remaining bytes are zeroes.
- 32 bytes containing the predicate's name representing as a string. As before, unused bytes are left as zeroes.

INSTRUCTION	BYTE FORMAT	ARGS

IF — ARGS: *reg, jump_offset*

0	1	1	0	0	0	0	0
0	0	0	r	r	r	r	r
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j

if *reg* != 0 then process until ELSE and then jump.
if *reg* = 0 then jump to ELSE
(note: IFs may be nested)

ELSE — ARGS: —

0	0	0	0	0	0	1	0

a marker for if blocks

ITER — ARGS: *id, reg, options, options arg, inn*

1	0	1	0	0	0	0	0
0	i	i	i	i	i	i	i
0	0	0	r	r	r	r	r
o	o	o	o	o	o	o	o
a	a	a	a	a	a	a	a
k	k	k	k	k	k	k	k
k	k	k	k	k	k	k	k
k	k	k	k	k	k	k	k
k	k	k	k	k	k	k	k
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j

iterates over all the tuples of type *id* that match
according to the following *matchlist*.
after all matching facts have been processed, use
inner_jump to jump to the first ITER instruction
outer_jump to jump to the next instruction outside the ITER

NEXT — ARGS: —

0	0	0	0	0	0	0	1

return to iter and process next matching fact

SEND — ARGS: reg_1, reg_2

0	0	0	0	1	0	0	0
0	0	0	r_1	r_1	r_1	r_1	r_1
0	0	0	r_2	r_2	r_2	r_2	r_2

sends the tuple in reg_1 along the path in reg_2
if $reg_1 = reg_2$ then the tuple is stored locally

REMOVE — ARGS: *reg*

1	0	0	r	r	r	r	r

delete tuple stored in reg from database

INSTRUCTION	BYTE FORMAT	ARGS

MOVE

$val_1,\ val_2$

0	0	1	1	0	0	0	0
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

copies val_1 to val_2

MVNILFIELD

field

0	1	1	1	0	0	0	0

sets *field* to the nil list

TEST-NIL

$reg_1,\ reg_2$

0	0	0	0	0	0	1	1
0	0	0	r_1	r_1	r_1	r_1	r_1
0	0	0	r_2	r_2	r_2	r_2	r_2

register $r_2 = 1$ if r_1 is nil.
register $r_2 = 0$ if r_1 is not nil.

ALLOC

id, val

0	1	0	0	0	0	0	0
0	i	i	i	i	i	i	i
0	0	v	v	v	v	v	v

allocates a tuple of type *id* and stores it in *val*

RETURN

—

0	0	0	0	0	0	0	0

finished processing the tuple - return

CALL

id, reg, args

0	0	1	0	0	0	0	0
0	i	i	i	i	i	i	i
0	0	0	r	r	r	r	r

call external function number *id* with *args* and store the result in *reg*

CONS

$val_1,\ val_2,\ val_3$

0	0	0	0	0	1	0	0
0	0	0	0	0	0	*t*	*t*
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2
0	0	v_3	v_3	v_3	v_3	v_3	v_3

sets $val_3 = val_1 :: val_2$
t is the list type (00 = int, 01 = float, 02 = addr)

TAIL

$val_1,\ val_2$

0	0	0	0	0	1	1	0
0	0	0	0	0	0	*t*	*t*
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

sets $val_2 = tail\ val_1$
t is the list type (00 = int, 01 = float, 02 = addr)

INSTRUCTION	BYTE FORMAT	ARGS

SELECT — *size,hsize,htable,codeblocks*

0	0	0	0	1	0	1	0

this is a big instruction used to select a specific code block for a node. it is followed by a 4-byte integer indicating the *size* of the whole instruction, then a 4-byte integer indicating the size N of a simplified hash-table. N represents the number of nodes in the system for efficiency reasons. Next, there is N*4-byte integers, where each integer is the offset to a code block of the corresponding node. The offsets start after the end of the hash table. If the offset is 0, this node has no associated code block, so it should use *size* to jump to the next instruction. If the offset is positive, you should subtract one byte from it and then jump to the code block. At the end of each code block, there is a RETURN-SELECT.

RETURN-SELECT — *jump*

0	0	0	0	1	0	1	1

This instruction is followed by a 4-byte integer with a jump offset to the next instruction.

COLOCATED — *n1,n2,dest*

0	0	0	0	1	1	0	0
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2
0	0	0	r	r	r	r	r

sets dest = true if nodes n1 and n2 are on the same machine
sets dest = false otherwise

DELETE — *i,v1*

0	0	0	0	1	1	0	1
0	i	i	i	i	i	i	i
0	0	v_1	v_1	v_1	v_1	v_1	v_1

deletes the tuples of type i with the first argument as value v_1

REMOVE — *reg*

1	0	0	0	0	0	0	0
0	0	0	r	r	r	r	r

deletes tuple reg from the database

RETURN-LINEAR

1	1	0	1	0	0	0	0

linear fact was used, execution must terminate

RETURN-DERIVED

1	1	1	1	0	0	0	0

head of rule was derived, return if some linear fact was used

INSTRUCTION	BYTE FORMAT	ARGS

FLOAT — ARGS: val_1, val_2

0	0	0	0	1	0	0	1
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

sets val_2 = *(float)*val_1

RULE — ARGS: *id*

0	0	0	1	0	0	0	0

rule *id* is gonna be executed

RULE DONE

0	0	0	1	0	0	0	0

rule *id* has been matched

NEW AXIOMS — ARGS: *jump*

0	0	0	1	1	1	1	0
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j
j	j	j	j	j	j	j	j

after the instruction and jump address, there is an arbitrary
number of tuples to be instantiated.
the first byte of each tuple is the predicate code
the tuple arguments follow next.
arguments may be integers, floats, node addresses or serialized lists
if the list is nil, there is a 0x0 byte, if not,
there is 0x1 byte followed by the element
of the list and the rest of the list. the jump argument
indicates the next instruction and where the tuples end

SEND DELAY — ARGS: reg_1, reg_2, *time*

0	0	0	1	0	1	0	1
0	0	0	r_1	r_1	r_1	r_1	r_1
0	0	0	r_2	r_2	r_2	r_2	r_2
t	t	t	t	t	t	t	t
t	t	t	t	t	t	t	t
t	t	t	t	t	t	t	t
t	t	t	t	t	t	t	t

sends the tuple in reg_1 along the path in reg_2
if $reg_1 = reg_2$ then the tuple is stored locally
time indicates the delay (in milliseconds) of the derivation

INSTRUCTION	BYTE FORMAT	ARGS

PUSH

0	0	0	1	0	1	1	0

pushes an unintialized value into the stack

POP

0	0	0	1	0	1	1	1

pops a value from the stack

PUSH REGS

0	0	0	1	1	0	0	0

pushes all registers into the stack

POP REGS

0	0	0	1	1	0	0	1

pops all registers from the stack and restores their values

CALLF — ARGS: *id*

0	0	0	1	1	0	1	0
i	i	i	i	i	i	i	i

calls a Meld function

STRUCT VAL — ARGS: i, v_1, v_2

0	0	0	1	1	1	0	0
i	i	i	i	i	i	i	i
0	0	v_1	v_1	v_1	v_1	v_1	v_1
0	0	v_2	v_2	v_2	v_2	v_2	v_2

sets v_2 with the index i of the struct in v_1

MAKE STRUCT — ARGS: t, v

0	0	0	1	1	1	0	1
t	t	t	t	t	t	t	t
0	0	v	v	v	v	v	v

sets v as a new struct of type t

NOT — ARGS: reg_1, reg_2

0	0	0	0	0	1	1	1
0	0	0	r_1	r_1	r_1	r_1	r_1
0	0	0	r_2	r_2	r_2	r_2	r_2

sets register $r_2 = not\ r_1$

MVINTFIELD — ARGS: *int, field*

0	0	0	1	1	1	1	0

moves an integer to a tuple field (followed by integer and field value)

VALUE	BYTE FORMAT	ARGS
REG	1 r r r r r	*reg*
HOST_ID	0 0 0 0 1 1 refers to the node currently being processed	—
ANY	0 0 1 1 1 1 match value for any value	—
NIL	0 0 0 1 0 0 the empty list	—
NON NIL	0 0 1 1 0 1 match value for the non-empty list	—
LIST	0 0 1 1 1 0 match value for a list: it will contain afterwards the value of the head (may be any) and the value of the tail	—
INT	0 0 0 0 0 1 the next 4 bytes after the current instruction are an immediate integer to which this refers	*int*
FLOAT	0 0 0 0 0 0 the next 4 bytes after the current instruction are an immediate float to which this refers	*float*
ADDR	0 0 0 1 0 1 the next 4 bytes after the current instruction are the address to which this refers	*addr*
FIELD	0 0 0 0 1 0 the next two bytes after the current instruction indicate a field of a register in the following format: X X X X f f f f X X X r r r r r with *reg* indicating a register with a tuple value and *field* indicating the tuple's field number.	*field, reg*
STRING	0 0 0 1 1 0 the next 4 bytes indicate the length of the string which are followed by the string itself	*size, content*

VALUE	BYTE FORMAT	ARGS
ARG	0 0 0 1 1 1 the next byte indicates the argument id	*id*
CONST	0 0 1 0 0 0 the next 4 bytes indicates the constant id	*const id*
STACK	0 0 1 0 0 1 use the element at offset from the top of the stack (4 byte unsigned int)	*offset*
PC-COUNTER	0 0 1 0 1 0 refers to the program counter	
PTR	0 0 1 0 1 1 the next 8 bytes are a machine pointer (in the C sense)	*ptr*
BOOL	0 0 1 1 0 0 the next byte is a bool (either 1 or 0)	*bool*

ARGS	BYTE FORMAT	
VALUE	X X v v v v v v	*value*

MATCHLIST	BYTE FORMAT	
MATCHLIST	f f f f f f f f m m v v v v v v requires that the tuple's field *field* match *value* mm=11 if the match list is empty and mm=01 for the last entry in the list.	*field, marker, value*

AGGREGATE	BYTE FORMAT
none	0 0 0 0
first	0 0 0 1
int max	0 0 1 0
int min	0 0 1 1
int sum	0 1 0 0
float max	0 1 0 1
float min	0 1 1 0
float sum	0 1 1 1
int set_union	1 0 0 0
float set_union	1 0 0 1
int list sum	1 0 1 0
float list sum	1 0 1 1

TYPE	BYTE FORMAT				EXTRA
int	0	0	0	0	
float	0	0	0	1	
addr	0	0	1	0	
list	0	0	1	1	the type of the elements
struct	0	1	0	0	a byte indicating the size of the structure and then the types of the fields
bool	0	1	0	1	
string	1	0	0	1	

PROPERTY	BYTE POSITION
aggregate	1
persistent	2
linear	3
delete	4
schedule	5

NOTES:
All offsets and lengths are given in bytes.