Denotational Model and Implementation of Scalable Virtual Machine in CPDev

Jan Sadolewski
Department of Computer and Control Engineering
Rzeszow University of Technology,
al. Powstańców Warszawy 12, 35-959 Rzeszów, Poland
Email: js@kia.prz.edu.pl

Bartosz Trybus
Department of Computer and Control Engineering
Rzeszow University of Technology,
al. Powstańców Warszawy 12, 35-959 Rzeszów, Poland
Email: btrybus@kia.prz.edu.pl

Abstract—Denotational semantic model and its implementation in C/C++ are presented for a virtual machine executing programs written in the CPDev development environment according to IEC 61131 standard. Programs written in IEC ST language are compiled to control-oriented intermediate language designed specifically for the machine. Architecture of the machine and its operation are represented by formal semantic model which assigns abstract algebraic objects to denote machine behaviour. Execution of intermediate language instructions is described in details by denotational semantic equations followed strictly by C/C++ implementations to assure reliability of the machine.

I. INTRODUCTION

The concept of virtual machines as platforms for software execution had a significant impact on computer science for almost half a century [1], [2]. A virtual machine (VM) is understood as a kind of processor with a certain instruction set and data types, which is implemented by software on particular hardware platforms. A VM processes an intermediate code generated by a compiler from a source program. The concept of VMs has been gaining importance due to the widespread use of the Java [3] and the .NET [4], [5]. Solutions based on VMs have some important advantages, namely a) source program and intermediate code are independent of target platforms, b) one compiler is sufficient, c) programs are executed in safe environments. The disadvantages include slower execution of the intermediate code and the need to develop a runtime environment suitable for the target platform.

This paper deals with the development of a runtime environment for control programs written according to the standard IEC 61131 [6]. The IEC standard defines the programming languages: Structured Text (ST), Instruction List (IL), Ladder Diagram (LD), Function Block Diagram (FBD) and Sequential Function Chart (SFC). Here, employing the VM concept appears to be particularly justified in order to cope with the large variety of target platforms. The CPDev engineering environment [7] uses this concept to program controllers according to the IEC standard. It consists of a compiler translating ST to intermediate code and a VM-based runtime system written in C. Initially, small and medium-scale controllers were considered [8]. Recently, however, motivated by applications with extensive calculations, arose the need to extend the CPDev compiler and its VM. Therefore, some additional assumptions were imposed, namely:

- to develop a semantic model of the machine and its intermediate language followed by a C implementation,
- to achieve scalability of the machine depending on the particular hardware and application requirements,

The model formalizes the VM description as an interpreter of the intermediate code, including instruction and operand decoding, and low-level operations while executing the instructions. Denotational semantics [9], [10] appropriate to formally describe programming languages are applied [11], [12]. For denotations the λ-notation is adequate and, therefore, applied [9], [13].

II. VIRTUAL MACHINE ARCHITECTURE

The architecture of the VM includes [14]: code and data memories, stacks and registers. The instruction processing module fetches successive instructions from Code memory and executes them acquiring values of operands either from Data or Code memory. Results are stored in Data memory.

Registers: The program counter is kept in the CodeReg register. The data base register DataReg is set by calls to and returns from subprograms, including function blocks and functions. When entering a subprogram the current values of CodeReg and DataReg are pushed onto Code stack and Data stack. The machine also includes the Flags register with status flags signaling errors or unusual situations.

VMASM intermediate language: The virtual machine operates as an interpreter of assembly code called VMASM (VM Assembler). The syntax is:

[:label] instruction [operand1][,operand2]...

Instruction set: Functions and system procedures are two kinds of virtual machine instructions. Examples are shown in Table I.

The compilation of the simple ST instruction

\[ \text{MOTOR:=}(\text{START OR MOTOR}) \text{ AND NOT STOP}; \]

is presented in Listing 1. At first, the variables START and MOTOR are ORed and the result is stored in a temporary variable ?LR?A03. If it is zero, JZ jumps to r>A02, where MCD sets MOTOR to 0 interpreted as FALSE. If ?LR?A03 is not zero, the function NOT performs logical negation of STOP storing the result in ?LR?A05. If it is zero, JZ jumps to the

<table>
<thead>
<tr>
<th>Table I. VMASM Intermediate Language Instruction Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>MOV</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>SUB</td>
</tr>
<tr>
<td>MUL</td>
</tr>
<tr>
<td>DIV</td>
</tr>
<tr>
<td>JNZ</td>
</tr>
<tr>
<td>JZ</td>
</tr>
<tr>
<td>CALL</td>
</tr>
<tr>
<td>RET</td>
</tr>
<tr>
<td>IFE</td>
</tr>
<tr>
<td>EXIT</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 1: ST Instructions Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
</tr>
<tr>
<td>OR</td>
</tr>
<tr>
<td>MOTOR</td>
</tr>
<tr>
<td>AND</td>
</tr>
<tr>
<td>NOT STOP</td>
</tr>
<tr>
<td>JZ</td>
</tr>
<tr>
<td>r&gt;A02</td>
</tr>
<tr>
<td>MCD</td>
</tr>
<tr>
<td>?LR?A03</td>
</tr>
<tr>
<td>FALSE</td>
</tr>
<tr>
<td>?LR?A05</td>
</tr>
<tr>
<td>STOP</td>
</tr>
</tbody>
</table>

IEEE Catalog Number: CFP2285N-ART ©2022, PTI 587
TABLE I
SELECTED FUNCTIONS AND PROCEDURES

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Addition</td>
<td>+ (arithm.)</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtraction</td>
<td>– (arithm.)</td>
</tr>
<tr>
<td>GT</td>
<td>Greater</td>
<td>&gt;</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>=</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>NOT</td>
<td>Negation</td>
<td>– (unary)</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
<td>&amp;</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
</tr>
<tr>
<td>JZ, JNZ</td>
</tr>
<tr>
<td>CALB</td>
</tr>
<tr>
<td>RETURN</td>
</tr>
<tr>
<td>MCD</td>
</tr>
</tbody>
</table>

:?A02 as before to set MOTOR to 0. If not, the first MCD sets MOTOR to 1 (TRUE). This is followed by JMP to :?E08 from which another code begins.

Listing 1. Example of VMASM mnemonic code

```xml
<deny-type name="USINT"/>
<type name="USINT" implement="alias">
  <alias name="BYTE"/>
</type> ...

<function name="ADD" vmcode="01" return="INT"/>
```

**Functions:** The definition of one in the group of ADD functions is presented in Listing 3. The virtual machine code vmcode consists of two bytes, with the first one 01 identifying the group, whereas the second *2 indicates a flexible number of inputs (+) and identifies the data type (2) processed. The two components of vmcode are called group and type identifier, and are denoted by ig and it. By choosing an appropriate it, type-specific functions such as ADD:SINT, ADD:INT, etc. are defined.

Listing 3. Function definition

```xml
<function name="ADD" vmcode="01*2" return="INT"/>
```

Procedures: All system procedures are identified by ig=1C. The second byte it of vmcode indicates a particular procedure. The definitions of JNZ and CALB are shown in Listing 4. JNZ executes a conditional jump to :gclabel, CALB calls the subprogram at :gclabel.

Listing 4. Procedure definitions

```xml
<sysproc name="JNZ" vmcode="1C01">
  <op no="0" name="cond" type="BOOL"/>
  <op no="1" name="clbl" type="gclabel"/>
</sysproc>
<sysproc name="CALB" vmcode="1C16">
  <op no="0" name="inst" type="rdlabel"/>
  <op no="1" name="clbl" type="gclabel"/>
</sysproc>
```

**Operand types:** The following types are available:
- :gclabel :gdlabel – global pointer (address) to Code/Data memory.
- :rdlabel :rdlabel – address relative to actual content of code/data register.
- :imm – immediate value (direct, constant).

**III. SCALABILITY**

The data types and instructions of the VMASM language are defined in XML-formatted library configuration files (LCF).

**Types and instructions:** A portion of type definitions is shown in Listing 2. By applying deny-type one can restrict some data types. Aliases to existing types and special types not specified in the IEC standard can be defined, too.

Listing 2. Type definition

```xml
<deny-type name="LREAL" />
<type name="USINT" implement="alias">
  <alias name="BYTE"/>
</type> ...
```

**Functions:** The definition of one in the group of ADD functions is presented in Listing 3. The virtual machine code vmcode consists of two bytes, with the first one 01 identifying the group, whereas the second *2 indicates a flexible number of inputs (+) and identifies the data type (2) processed. The two components of vmcode are called group and type identifier, and are denoted by ig and it. By choosing an appropriate it, type-specific functions such as ADD:SINT, ADD:INT, etc. are defined.

Listing 3. Function definition

```xml
<function name="ADD" vmcode="01*2" return="INT"/>
```

**Procedures:** All system procedures are identified by ig=1C. The second byte it of vmcode indicates a particular procedure. The definitions of JNZ and CALB are shown in Listing 4. JNZ executes a conditional jump to :gclabel, CALB calls the subprogram at :gclabel.

Listing 4. Procedure definitions

```xml
<sysproc name="JNZ" vmcode="1C01">
  <op no="0" name="cond" type="BOOL"/>
  <op no="1" name="clbl" type="gclabel"/>
</sysproc>
<sysproc name="CALB" vmcode="1C16">
  <op no="0" name="inst" type="rdlabel"/>
  <op no="1" name="clbl" type="gclabel"/>
</sysproc>
```

**Operand types:** The following types are available:
- :gclabel :gdlabel – global pointer (address) to Code/Data memory.
- :rdlabel :rdlabel – address relative to actual content of code/data register.
- :imm – immediate value (direct, constant).

**IV. SEMANTIC MODEL**

Semantic models provide formal descriptions of programming languages [11], [15]. In case of VMASM, the model consists of domains describing the virtual machine’s states, memory functions, value interpreters relating memory to VMASM types, limited range operators, and a universal semantic function.

**Semantic domains:** The domain BasicTypes consists of four sets reflecting the memory sizes of the VMASM types. The domain Address specifies 16- or 32-bit implementation. The general domain Memory is a function mapping Address to Byte1. Stack models a sequence (*) of Address domains (Kleene closure).

BasicTypes = Byte1 + Bytes2 +
           + Bytes4 + Bytes8

Address = if AddressSize = 2 then
          Bytes2 else Bytes4

Memory = Address → Byte1

CodeMemory = Memory

Stack = Address*

CodeStack = Stack

CodeReg = Address

Flags = Bytes2

State: The purpose of program execution is to change the current state into a new one. The state of the VM is a Cartesian product of memory domains, stacks, registers and flags, i.e.

State = CodeMemory × DataMemory ×
\[ \times \text{CodeStack} \times \text{DataStack} \times \times \text{CodeReg} \times \text{DataReg} \times \text{Flags} \]

Model functions: The functions presented below model low-level operations executed on memory, stacks and flags.

- Get data from memory (read)
  \[ G1BM = (\text{Address} \times \text{Memory}) \to \text{Byte1} \]
  \[ G2BM = (\text{Address} \times \text{Memory}) \to \text{Byte2} \]
  \[ G4BM = (\text{Address} \times \text{Memory}) \to \text{Bytes4} \]
  \[ G8BM = (\text{Address} \times \text{Memory}) \to \text{Bytes8} \]

- Get address from memory
  \[ \text{GetAddress} = (\text{Address} \times \text{Memory}) \to \text{Address} \]

- Memory move (copy)
  \[ \text{MemMove} = (\text{Address} \times \text{Memory} \times \text{Byte1}) \to \text{Memory} \]

The two addresses represent source and target, respectively with Byte1 denoting number of bytes being moved.

- Stack functions
  \[ \text{Push} = (\text{Stack} \times \text{Address}) \to \text{Stack} \]
  \[ \text{Pop} = \text{Stack} \to (\text{Address} \times \text{Stack}) \]

Value interpreters: The following sample functions provide numerical interpretations of memory chunks.

- Boolean
  \[ \text{BoolOf} = \text{Byte1} \to \text{BOOL} \]

- Integers
  \[ \text{FromInt} = \text{INT} \to \text{Bytes2} \]
  \[ \text{DIntOf} = \text{DINT} \to \text{Bytes4} \]
  \[ \text{LIntOf} = \text{LINT} \to \text{Bytes8} \]

Limited range operators: The virtual machine executes arithmetic operations in limited ranges, dependent on the particular types. For signed integers addition \( \oplus \) is defined by

\[ a \oplus b = \text{if } (a + b) \geq 0 \]
\[ \text{then } (a + b) \mod (-\text{MinRange}(a)) \]
\[ \text{else } (a + b) \mod (\text{MinRange}(a) + 1) \]

where \( \text{MinRange} \) for SINT, INT, DINT and LINT means \(-128, -32768, -2^{31} \) or \(-2^{63}, \) respectively. For unsigned integers USINT, UINT etc. we have

\[ a \oplus b = (a + b) \mod (\text{MaxRange}(a)) \]

where \( \text{MaxRange} \) means \( 256, 65536 \) etc.

Unification: The operator \( \vdash \) used in expression unifies both sides. If the right side is an expression, then the left side is a variable with the value of the right side (assignment). If the left side is a tuple and the right side a variable, then the variable is split into the tuple’s components.

Universal semantic function: To jointly express the concept of decoding group and type, followed by execution of a particular instruction, one may define a universal function covering all instructions

\[ U[\text{any_instruction}] = \text{State} \to \text{State} \quad (1) \]

Internally, after decoding \( \lambda s \) and \( \mu t \), this function calls a specific function of the form

\[ C[\text{instruction}] = \text{State} \to \text{State} \quad (2) \]

Instruction decoding: Instruction decoding can formally be expressed by the denotational semantic equation shown in Listing 5. According to [9] or [13], the \( \lambda \)-expression has the form of \( \lambda s.\text{body} \), where \( s \) denotes the current state and \( \text{body} \) determines the value returned by the function. The \( \text{body} \) consists of a sequence of operations, the first of which splits current state \( s \) into a tuple composed of model components. The other operations decode the values of identifiers \( \lambda s \) and \( \mu t \) and, by means of \( \lambda s.\text{match ... with} \) statements, call particular \( C \) functions. The result provided by \( C \) defines the new state \( s_1 \) returned by the function \( \lambda s \).

Listing 5. Denotational equation for the function \( \lambda s \)

\[ \lambda s.\text{match } \lambda s \text{ with } \begin{cases} \text{ig} & \vdash \lambda s.\text{G1BM}(cr, cm) \\ \mu t & \vdash \lambda s.\text{G1BM}(cr, cm) \end{cases} \]
\[ s_1 = \text{match } \lambda s \text{ with } \begin{cases} 01 \to \text{match } \mu t \text{ with } \begin{cases} 12 \to C[\text{ADD:INT:r:op1:op2}] \\ (cm, dm, cs, ds, cr, dr, flg) \end{cases} \\ 32 \to C[\text{ADD:INT:r:op1:op2:op3}] \\ (cm, dm, cs, ds, cr, dr, flg) \end{cases} \]
\[ s_1 \]

V. DENOTATIONS AND IMPLEMENTATIONS

Denotational equations modeling the VMASM instructions have the common form \( \lambda s.\text{body} \) where the dots on the left side are replaced by the descriptor of a particular instruction. Splitting current state \( s \) into components through unification

\[ (cm, dm, cs, ds, cr, dr, flg) \vdash s \quad (3) \]

is the first operation in \( \text{body} \).

Assume that while calling a particular function \( C \) by the universal function \( U \), the code register \( cr \) points to the first operand. (actually \( cr_2 \) in Listing 5). If the operand is a variable
or label, then its value, i.e. address, is acquired from code memory \( cm \) by

\[
\text{operand} := \text{GetAddress}(cr, cm)
\]

(4a)

In case of a global variable or label, \( \text{operand} \) stands for a direct address in data or code memory. If, however, the operand is a local variable of a subprogram, then the value \( \text{operand} \) means an address relative to the current value of data base register \( dr \), which was set earlier by a subprogram call. Therefore, the address of a local variable is obtained by adding

\[
\text{operandaddr} := dr \oplus \text{operand}
\]

(4b)

The value of a variable, here shown for a Boolean, in data memory \( dm \) is read out and interpreted by composition

\[
\text{BooLof}(\text{GBM}(\text{operandaddr}, dm))
\]

(5)

If an instruction has another operand, the code register \( cr \) is incremented to point to the next memory location by

\[
\text{cr}_1 := cr \oplus \text{AddressSize}
\]

(6)

Defining the new state \( s_1 \) as the tuple

\[
s_1 := (cm, ...) \text{ and (4b) (with the content } dr \text{ of the data base register equal to zero in case of a global variable). The code register is incremented to } \text{cr}_1 \text{ to obtain the address } clbl \text{ and, then, to } \text{cr}_2 \text{ pointing to the next instruction. The Boolean value } ctl \text{ controlling execution is determined as in (5). Depending on } ctl \text{, the code register of } s_1 \text{ includes either } clbl \text{ or } cr_2."

The first operand of CALB is the label of an instance in data memory for which the subprogram beginning at the label \( clbl \) is executed. The instance address \( iad \) and the subprogram address \( clbl \) are determined as before. \( cr_2 \) points to the next instruction. Since the contents of \( cr_2, dr \) must be remembered for the subprogram return, they are pushed onto corresponding stacks.

Listing 8 shows C implementation of the JNZ and CALB. All system procedures having the common group identifier 1C are handled by a single general function \( \text{IG_SYSCProc}_1C \), with type identifier it as its parameter. The command \( \text{switch} \) selects a particular procedure. Each of the code segments sets \( \text{codeReg} \) to a new value depending on the respective meaning. \( \text{CALB} \) also modify \( \text{dataReg} \).

### Listing 8. Implementations of JNZ and CALB procedures

```c
void \text{IG_SYSCProc}_1C(BYTE it) {
    switch(it) {
        case 0x01: /* JNZ conditional jump */ {
            ADDRESS cnaddr = dataReg + GetCodeAddress();
            ADDRESS clbl = GetCodeAddress();
            BOOL ctl = BOOLOF(GBMData(cnaddr));
            if (ctl) codeReg = clbl;
            break;
        }
        case 0x16: /* CALB call a function block */ {
            ADDRESS iad = dataReg + GetCodeAddress();
            ADDRESS clbl = GetCodeAddress();
            push_CodeStack(codeReg);
            push_DataStack(dataReg);
            dataReg = iad;
            codeReg = clbl;
        }
        ... /* other procedures */
        default: /* unknown code */
            flags |= FAULT;
            break;
    }
}
```

### Selected functions:

The semantics of function \( \text{NOT} \) presented in Listing 9 negates the value stored at \( \text{op1} \). The addresses \( \text{raddr}, \text{op1addr} \) are determined as before, followed by the Boolean value \( bv \) obtained as in (5). By means of the function \( \text{GBM} \) (Sec. IV) the value at \( raddr \) in data memory \( dm \) is then updated. That value is determined from \( bv \) by the \( \text{FromBool} \) and \( \text{match ... with} \) construct. \( \text{um} \) denotes the new state of the data memory. The C implementation of the \( \text{NOT} \) function presented in Listing 10 corresponds directly to its semantics.

In the case of the function \( \text{EQ} \) (Listing 11) two LINT operands \( \text{op1}, \text{op2} \) are checked for equality. The Boolean value \( \text{cmp} \) follows from comparison \( (=) \) of the LINT numbers determined by \( \text{LIsEq} \). The updated data memory in \( s_1 \) is the result of invoking \( \text{GBM} \). The byte stored at \( raddr \) is given by \( \text{FromBool}(\text{cmp}) \). The function \( \text{IG_EQ}_12 \) from Listing 12 implements the comparison \( \text{EQ} \) for all relevant data types (group) via a parameterized macrodefinition \( \text{EQ}_\text{TYPE} \).
The value of an operand of a particular TYPE is determined in EQ_TYPE by the function TYPE##Of with given sizeof(TYPE).

Listing 9. Denotation of NOT function

```c
C[NOT::r::op1] = λs. (cm, dm, cs, ds, cr, dr, flg) := s
r := GetAddress(cm)
raaddr := dr ⊕ r
cr1 := cr ⊕ AddressSize
op1 := GetAddress(cm)
op1addr := dr ⊕ op1
cr2 := cr1 ⊕ AddressSize
bv := BOOLOf(GBMData(op1addr));
um := U1BM(raaddr, FromBOOL(cmp));
U1BM(raaddr, FromBOOL(bv with
| true → false
| false → true end))
s1 := (cm, um, cs, cr, dr, flg)
s1
```

Listing 10. Implementation of NOT function

```c
ADDRESS radaddr = dataReg + GetCodeAddress();
ADDRESS op1addr = dataReg + GetCodeAddress();
BOOL bv = BOOLOf(GBMData(op1addr));
U1BM(radaddr, FromBOOL(
    bv ? FALSE : TRUE ));
```

Listing 11. Denotation of EQ function

```c
C[EQ::LINT::r::op1::op2] = λs. (cm, dm, cs, ds, cr, dr, flg) := s
r := GetAddress(cm)
raaddr := dr ⊕ r
cr1 := cr ⊕ AddressSize
op1 := GetAddress(cm)
op1addr := dr ⊕ op1
cr2 := cr1 ⊕ AddressSize
op2 := GetAddress(cm)
op2addr := dr ⊕ op2
cmp := LIntOf(GBMData(op2addr, dm))
= LIntOf(GBMData(op2addr, dm))
s1 := (cm, U1BM(raaddr, dm),
FromBOOL(cmp)), cs, ds, cr, dr, flg)
s1
```

Listing 12. Implementation of EQ function

```c
#define EQ_TYPE(TYPE) \  
switch (it & 0x0F) {  }  
EQ_TYPE(SINT);  
EQ_TYPE(INT);  
EQ_TYPE(DINT);
```

VI. CONCLUSION

Architecture, VMASM intermediate language, and denotational semantic model have been presented for a virtual machine executing IEC control programs. The machine’s scalability covers the address size, available data types and instructions. A semantic model was developed to enhance software quality. Denotational equations modeling VMASM instructions are directly followed by C implementations.

ACKNOWLEDGMENT

This project is financed by the Ministry of Education and Science of the Republic of Poland within the “Regional Initiative of Excellence” program for years 2019–2022. Project number 027/RID/2018/19, amount granted 11 999 900 PLN.

REFERENCES