Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 587–591

DOI: 10.15439/2022F236
ISSN 2300-5963 ACSIS, Vol. 30

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

```
MOTOR:=(START OR MOTOR) 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 `:?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 I
SELECTED FUNCTIONS AND PROCEDURES

Mnemonic	Meaning	Operator
Functions		
ADD	Addition	+ (arithm.)
SUB	Subtraction	− (arithm.)
GT	Greater	>
EQ	Equal	=
NE	Not equal	<>
NOT	Negation	− (unary)
AND	Logical and	&
System procedures		
JMP	Unconditional jump	
JZ, JNZ	Conditional jumps	
CALB	Subroutine call	
RETURN	Return from subroutine	
MCD	Initialize data	

:?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
```
  OR ?LR?A03, START, MOTOR
  JZ ?LR?A03, :?A02
  NOT ?LR?A05, STOP
  JZ ?LR?A05, :?A02
  MCD MOTOR, #01, #01
  JMP :?E08
:?A02
  MCD MOTOR, #01, #00
:?E08
```

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
```
<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 $\star 2$ indicates a flexible number of inputs ($\star$) 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
```
<function name="ADD" vmcode="01*2" return="INT">
  <operands>
    <op no="*" name="a*" type="INT"/>
  </operands>
</function> ...
```

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
```
<sysproc name="JNZ" vmcode="1C01">
   <op no="0" name="cnd" 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*,
- :rclabel :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).

$$\begin{aligned}
BasicTypes &= Byte1 + Bytes2 + \\
&\quad + Bytes4 + Bytes8 \\
Address &= \textbf{if } AddressSize = 2 \textbf{ then} \\
&\quad Bytes2 \textbf{ else } Bytes4 \\
Memory &= Address \to Byte1 \\
CodeMemory &= Memory \\
Stack &= Address^* \\
CodeStack &= Stack \\
CodeReg &= Address \\
Flags &= Bytes2
\end{aligned}$$

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.

$$State = CodeMemory \times DataMemory \times$$

$$\times\, CodeStack \times DataStack \times$$
$$\times\, CodeReg \times DataReg \times Flags$$

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

- Get data from memory (read)

$$G1BM = (Address \times Memory) \to Byte1$$
$$G2BM = (Address \times Memory) \to Bytes2$$
$$G4BM = (Address \times Memory) \to Bytes4$$
$$G8BM = (Address \times Memory) \to Bytes8$$

- Get address from memory

$$GetAddress = (Address \times Memory) \to Address$$

- Memory update (write)

$$U1BM = (Address \times Memory \times Byte1) \to Memory$$
$$U2BM = (Address \times Memory \times Bytes2) \to Memory$$
$$U4BM = (Address \times Memory \times Bytes4) \to Memory$$
$$U8BM = (Address \times Memory \times Bytes8) \to Memory$$

- Memory move (copy)

$$MemMove = (Address \times Memory \times Address \times$$
$$\times Memory \times Byte1) \to Memory$$

  The two $Addresses$ represent source and target, respectively with $Byte1$ denoting number of bytes being moved.
- Stack functions

$$Push = (Stack \times Address) \to Stack$$
$$Pop = Stack \to (Address \times Stack)$$

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

$$BoolOf = Byte1 \to BOOL$$
$$FromBool = BOOL \to Byte1$$
$$IntOf = Bytes2 \to INT$$
$$FromInt = INT \to Bytes2$$
$$DIntOf = Bytes4 \to DINT$$
$$FromDInt = DINT \to Bytes4$$
$$LIntOf = Bytes8 \to LINT$$
$$FromLInt = LINT \to Bytes8$$

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

$$a \oplus b = \textbf{if } (a+b) >= 0$$
$$\textbf{then } (a+b)\ mod\ (-MinRange(a))$$
$$\textbf{else } (a+b)\ mod\ (-MinRange(a)+1)$$

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

$$a \oplus b = (a+b)\ mod\ (MaxRange(a))$$

where $MaxRange$ means 256, 65536 etc.

Unification: The operator $:=$ 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

$$\mathcal{U}[\![\text{any\_instruction}]\!] = State \to State \tag{1}$$

Internally, after decoding `ig` and `it`, this function calls a specific function of the form

$$\mathcal{C}[\![\text{instruction}]\!] = State \to State \tag{2}$$

Instruction decoding: Instruction decoding can formally be expressed by the denotational semantic equation shown in Listing 5. According to [9] or [13], the λ-expression has the form of $\lambda s.body$, where s denotes the current state and $body$ determines the value returned by the function. The $body$ 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 ig and it, update the code register to cr_2 and, by means of **match** ... **with** statements, call particular $\mathcal{C}$ functions. The result provided by $\mathcal{C}$ defines the new state s_1 returned by the function $\mathcal{U}$.

Listing 5. Denotational equation for the function $\mathcal{U}$

```
$\mathcal{U}[\![\text{any\_instruction}]\!] =$
 $\lambda s.(cm, dm, cs, ds, cr, dr, flg) := s$
 $ig := G1BM(cr, cm)$
 $cr_1 := cr \oplus 1$
 $it := G1BM(cr_1, cm)$
 $cr_2 := cr_1 \oplus 1$
 $s_1 :=$ match $ig$ with
        | 01 → match $it$ with
                | 22 → $\mathcal{C}[\![$ADD:INT:r:op1:op2$]\!]$
                        $(cm, dm, cs, ds, cr_2, dr, flg)$
                | 32 → $\mathcal{C}[\![$ADD:INT:r:op1:op2:op3
                        $]\!](cm, dm, cs, ds, cr_2, dr, flg)$
                | ...
                end
        | ...
        end
 $s_1$
```

V. Denotations and implementations

Denotational equations modeling the VMASM instructions have the common form $\mathcal{C}[\![...]\!] = \lambda s.body$ where the dots on the left side are replaced by the descriptor of a particular instruction. Splitting current state s into components through unification

$$(cm, dm, cs, ds, cr, dr, flg) := s \tag{3}$$

is the first operation in $body$.

Assume that while calling a particular function $\mathcal{C}$ by the universal function $\mathcal{U}$, 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

$$operand := GetAddress(cr, cm) \tag{4a}$$

In case of a global variable or label, $operand$ stands for a direct address in data or code memory. If, however, the operand is a local variable of a subprogram, then the value $operand$ 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

$$operandaddr := dr \oplus operand \tag{4b}$$

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

$$BoolOf(G1BM(operandaddr, dm)) \tag{5}$$

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

$$cr_1 := cr \oplus AddressSize \tag{6}$$

Defining the new state s_1 as the tuple

$$s_1 := (cm, ...) \tag{7}$$

is the last operation in $body$, with the dots being replaced by new values of the data memory (if updated), stacks, etc.

Basic procedures: The denotational equation of the unconditional jump JNZ is presented in Listing 6 and subprogram call CALB in Listing 7.

Listing 6. Denotation of JNZ procedure

```
$\mathcal{C}[\![\texttt{JNZ:cnd:clbl}]\!] = \lambda s.$
 $(cm, dm, cs, ds, cr, dr, flg) := s$
 $cnd := GetAddress(cr, cm)$
 $cndaddr := dr \oplus cnd$
 $cr_1 := cr \oplus AddressSize$
 $clbl := GetAddress(cr_1, cm)$
 $cr_2 := cr_1 \oplus AddressSize$
 $ctl := BoolOf(G1BM(cndaddr, dm))$
 $s_1 :=$ **match** $ctl$ **with**
         | **true** $\rightarrow (cm, dm, cs, ds, clbl, dr, flg)$
         | **false** $\rightarrow (cm, dm, cs, ds, cr_2, dr, flg)$ **end**
 $s_1$
```

Listing 7. Denotation of CALB procedure

```
$\mathcal{C}[\![\texttt{CALB:inst:clbl}]\!] = \lambda s.$
 $(cm, dm, cs, ds, cr, dr, flg) := s$
 $inst := GetAddress(cr, cm)$
 $iad := dr \oplus inst$
 $cr_1 := cr \oplus AddressSize$
 $clbl := GetAddress(cr_1, cm)$
 $cr_2 := cr_1 \oplus AddressSize$
 $s_1 := (cm, dm, Push(cs, cr_2), Push(ds, dr), clbl, iad, flg)$
 $s_1$
```

The equation of JNZ has two operands, the conditional variable `cnd` in data memory and the code label `clbl` as before. The address $cndaddr$ is determined according to (4a) and (4b) (with the content dr of the data base register equal to zero in case of a global variable). The code register is incremented to cr_1 to obtain the address $clbl$ and, then, to cr_2 pointing to the next instruction. The Boolean value ctl controlling execution is determined as in (5). Depending on ctl, the code register of s_1 includes either $clbl$ 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 `IG_SYSCPROC_1C`, with type identifier `it` as its parameter. The command `switch` selects a particular procedure. Each of the code segments sets `codeReg` to a new value depending on the respective meaning. CALB also modify `dataReg`.

Listing 8. Implementations of JNZ and CALB procedures

```
void IG_SYSPROC_1C(BYTE it) {
 switch(it) {
  case 0x01: /* JNZ conditional jump */ {
   ADDRESS cndaddr = dataReg + GetCodeAddress();
   ADDRESS clbl = GetCodeAddress();
   BOOL ctl = BOOLOf(G1BMData(cndaddr));
   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;
  } break;
  ...     /* other procedures */
  default: /* unknown code */
   flags |= FAULT;
   break; }
```

Selected functions: The semantics of function NOT presented in Listing 9 negates the value stored at `op1`. The addresses $raddr$, $op1addr$ are determined as before, followed by the Boolean value bv obtained as in (5). By means of the function $U1BM$ (Sec. IV) the value at $raddr$ in data memory dm is then updated. That value is determined from bv by the $FromBool$ and **match** ... **with** construct. um denotes the new state of the data memory. The C implementation of the NOT function presented in Listing 10 corresponds directly to its semantics.

In the case of the function EQ (Listing 11) two LINT operands `op1`, `op2` are checked for equality. The Boolean value cmp follows from comparison ($=$) of the LINT numbers determined by $LIntOf$. The updated data memory in s_1 is the result of invoking $U1BM$. The byte stored at $raddr$ is given by $FromBool(cmp)$. The function `IG_EQ_12` from Listing 12 implements the comparison EQ for all relevant data types (group) via a parameterized macrodefinition `EQ_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

$$
\begin{aligned}
&\mathcal{C}[\![\texttt{NOT:r:op1}]\!] = \lambda s.\\
&(cm, dm, cs, ds, cr, dr, flg) := s\\
&r := GetAddress(cr, cm)\\
&raddr := dr \oplus r\\
&cr_1 := cr \oplus AddressSize\\
&op1 := GetAddress(cr_1, cm)\\
&op1addr := dr \oplus op1\\
&cr_2 := cr_1 \oplus AddressSize\\
&bv := BoolOf(G1BM(op1addr, dm))\\
&um := U1BM(raddr, dm, FromBool(\textbf{match } bv \textbf{ with}\\
&\mid \textbf{true} \rightarrow \textbf{false}\\
&\mid \textbf{false} \rightarrow \textbf{true end}))\\
&s_1 := (cm, um, cs, ds, cr_2, dr, flg)\\
&s_1
\end{aligned}
$$

Listing 10. Implementation of NOT function

```
ADDRESS raddr = dataReg + GetCodeAddress();
ADDRESS op1addr = dataReg + GetCodeAddress();
BOOL bv = BOOLOf(G1BMData(op1addr));
U1BM(raddr, FromBOOL(
    bv ? FALSE : TRUE ));
```

Listing 11. Denotation of EQ function

$$
\begin{aligned}
&\mathcal{C}[\![\texttt{EQ:LINT:r:op1:op2}]\!] = \lambda s.\\
&(cm, dm, cs, ds, cr, dr, flg) := s\\
&r := GetAddress(cr, cm)\\
&raddr := dr \oplus r\\
&cr_1 := cr \oplus AddressSize\\
&op1 := GetAddress(cr_1, cm)\\
&op1addr := dr \oplus op1\\
&cr_2 := cr_1 \oplus AddressSize\\
&op2 := GetAddress(cr_2, cm)\\
&op2addr := dr \oplus op2\\
&cr_3 := cr_2 \oplus AddressSize\\
&cmp := LIntOf(G8BM(op1addr, dm))\\
&\quad = LIntOf(G8BM(op2addr, dm))\\
&s_1 := (cm, U1BM(raddr, dm,\\
&FromBool(cmp)), cs, ds, cr_3, dr, flg)\\
&s_1
\end{aligned}
$$

Listing 12. Implementation of EQ function

```
#define EQ_TYPE(TYPE) \
case IT_EQ_##TYPE & 0x000F: {\
 ADDRESS raddr = dataReg + GetCodeAddress(); \
 ADDRESS op1addr = dataReg + GetCodeAddress(); \
 ADDRESS op2addr = dataReg + GetCodeAddress(); \
 TYPE op1 = TYPE##Of(GetMemData(op1addr, sizeof(TYPE)
     )); \
 TYPE op2 = TYPE##Of(GetMemData(op2addr, sizeof(TYPE)
     )); \
 BOOL cmp = op1 == op2; \
 U1BM(raddr, FromBOOL(cmp));\
break;

void IG_EQ_12(BYTE it) {
  switch (it & 0x0F)  {
    EQ_TYPE(SINT);
    EQ_TYPE(INT);
    EQ_TYPE(DINT);
    EQ_TYPE(LINT);
    ...   /* other types */
 default:  /* unknown code */
    flag |= FAULT;
 }
 return; }
```

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 Minister 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

[1] B. Venners, *Inside the Java Virtual Machine*. McGraw-Hill Companies, 1997.

[2] R. F. Stärk, J. Schmid, and E. Börger, *Java and the Java Virtual Machine*. Berlin: Springer Heidelberg, 2001.

[3] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java® Virtual Machine Specification*. Oracle America, Inc., 2013.

[4] T. L. Thai and L. H., *.NET Framework Essentials*. O'Reilly Media, 2001.

[5] ECMA-335 Standard, *Common Language Infrastructure (CLI)*. Geneva: ECMA, 2012.

[6] IEC 61131-3 Standard, *Programmable Controllers. Part 3. Programming languages*. International Standard: IEC, 2013.

[7] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Developing a multiplatform control environment," *JAMRIS*, vol. 13, no. 4, p. 73–84, 2019. [Online]. Available: https://doi.org/10.14313/JAMRIS/4-2019/40

[8] B. Trybus, "Development and Implementation of IEC 61131-3 Virtual Machine," *Theoretical and Applied Informatics*, vol. 23, no. 1, pp. 21–35, 2011.

[9] M. Gordon, *The Denotational Description of Programming Languages*. New York: Springer-Verlag, 1979.

[10] J. Stoy, *Denotational Semantics: The Scott–Strachey approach to programming language theory*. Massachusetts: Massachusetts Institute of Technology, 1979.

[11] K. Slonneger and B. L. Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach*. Addison-Wesley Publishing Company, Inc, 1995.

[12] N. S. Papaspyrou, "Denotational semantics of ANSI C," *Computer Standards & Interfaces*, vol. 23, pp. 169–185, 2001. doi: 10.1016/S0920-5489(01)00059-9

[13] H. Barendregt and E. Barendsen, *Introduction to Lambda Calculus*. online: ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf, 2000.

[14] J. Sadolewski and B. Trybus, "Compiler and virtual machine of a multiplatform control environment," *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 70, no. 2, p. e140554, 2022. doi: 10.24425/bpasts.2022.140554

[15] D. Schmidt, *Denotational Semantics: A Methodology for Language Development*. Kansas State University, Manhattan: Department of Computing and Information Sciences, 1997.