DOI: 10.15439/2023F5651
ISSN 2300-5963 ACSIS, Vol. 35

Exception Handling in Programmable Controllers with Denotational Model

Jan Sadolewski
0000-0001-7370-9027
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
0000-0002-4588-3973
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*—The paper introduces a customized approach to handle failures in IEC 61131-3 programmable controllers. The solution assumes the utilization of a virtual machine as a runtime environment to execute control code in an isolated manner. A formal model of the runtime is presented, employing denotational semantics. Subsequently, the model is expanded by incorporating new procedures that enable the handling of runtime exceptions using ST code constructs. This formal model serves as the foundation for implementing the exception infrastructure in the CPDev development environment. The research presented in the paper, driven by industry demands, aims to facilitate the development of more reliable and resilient control systems, capable of effectively dealing with failures.**

I. Introduction

PROGRAMMABLE Logic Controllers (PLC) and Programmable Automation Controllers (PAC) have established their position in the modern world. Their applications are wide-ranging, including industrial production, energy, transportation, and Internet of Things solutions. They control processes, often using advanced algorithms, and are expected to be reliable and operate in real-time mode.

One characteristic that allows for the flexibility of these devices is the ability to program them, where an engineer creates their own control algorithm and places it in the controller. From this perspective, such a controller is versatile as it can be programmatically tailored to specific applications and its functionality can be extended or modified when changes are required in the controlled object.

In some controllers, programming is done using typical languages, most commonly C/C++. However, there are standardized mechanisms and programming solutions specifically designed for control devices. The most significant of these are the international standards IEC 61131-3 [1] and IEC 61499 [2]. Particularly, the former has become a recognized standard adopted by many manufacturers. It allows, among other things, the transfer of program code between devices from different vendors and introduces specialized programming languages such as structured text (ST), instruction list (IL), graphical block diagram (FBD), ladder diagram (LD), and sequential function chart (SFC). The language structures align well with control system programming paradigms. Hence, this article considers a system that complies with the IEC 61131-3 standard.

The stable and predictable operation of PLC and PAC controllers is a feature resulting from their applications, where errors and unexpected reactions can have serious consequences. To minimize the risk of such situations, developers employ various solutions. The mere use of the standard's languages can reduce potential problems, particularly by avoiding programming mechanisms related to manipulating pointers and dynamic memory allocation. Another solution involves constructing an isolated runtime environment for user programs. In such cases, the effects of programmer errors will not propagate beyond that runtime, allowing the device to remain operational and enabling controlled handling of exceptional situations.

One concept for creating such an isolated environment is a virtual machine [3]. In general terms, a virtual machine (referred to as VM) is understood here as a type of processor with its own instruction set and data types, implemented through software on specific hardware platforms. This means that when processing code designed for a VM, appropriate software mechanisms execute it using the native resources of the target platform, such as a specific CPU and memory. The VM processes code, typically referred to as intermediate code, which is generated by a compiler from a source program. The concept of virtual machines has gained prominence in information technology due to the widespread use of platforms such as the Java Runtime Environment [4] and the .NET Framework [5], [6].

Solutions based on virtual machines offer several important advantages. Firstly, the source program and intermediate code are independent of the target hardware platforms. This means that only one compiler for the source language is required, rather than separate cross-compilers for different platforms. Additionally, programs are executed within secure environments with memory protection, preventing potential errors from propagating beyond the designated boundaries.

However, there are also disadvantages to consider. Execution of intermediate code tends to be slower compared to executing native code on the target processor. This is because the instructions and operands of the intermediate language need to be decoded by software, whereas a standard CPU utilizes hardware decoders and pipelining. Consequently, implementing even a simple intermediate instruction requires multiple native instructions.

When designing a virtual machine as a runtime environment prepared to handle error and exceptional situations, several aspects need to be taken into account. The first aspect is the compatibility of the machine's operation with its specification. For this purpose, the authors have proposed a formal model of operations performed by the virtual machine using denotational semantics [7]. This model enables the implementation of these operations in accordance with the assumptions, for example, in languages like C/C++. In this article, the model has been expanded to include functions related to exception handling.

Another task is to supplement the languages of the IEC 61131-3 standard with additional constructs related to exception handling. This is necessary due to the lack of dedicated solutions in the standard. Hence, there is a need to introduce them. The authors take into account the extensions to the ST language available in the CODESYS package but propose their own implementation of these extensions.

II. Programming and runtime environments

The engineering environment CPDev (Control Program Developer) allows to program controllers according to the IEC standard [8]. It consists of ST/IL/FBD/LD/SFC editors, a compiler translating programs to the intermediate code [9] and a VM-based runtime system written in C/C++ [10].

The architecture of the VM is shown in Fig. 1. It includes the following components:

- code and data memories,
- code and data stacks,
- registers and pointers,
- instruction processing module.

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 directly in *Data memory*.

The machine does not utilize an accumulator, however it maintains other registers. The instruction pointer, also known as the program counter, is stored in the *CodeReg* register. VM increments the *CodeReg* register every time after fetching an instruction code or an operand address. The *DataReg* register is used for managing the data base addresses and is set during subprogram calls and returns, including function blocks and functions. This allows the executed code to access variables in different areas of the *Data memory* and handle multiple instances of subprograms. When entering a subprogram, the current values of *CodeReg* and *DataReg* are pushed onto the *Code stack* and *Data stack* respectively. Upon returning, the contents of these registers are popped from the stacks. This stack mechanism enables nested function blocks. Additionally, the machine includes the *Flags* register, which contains status flags that signal errors or unusual situations such as an array

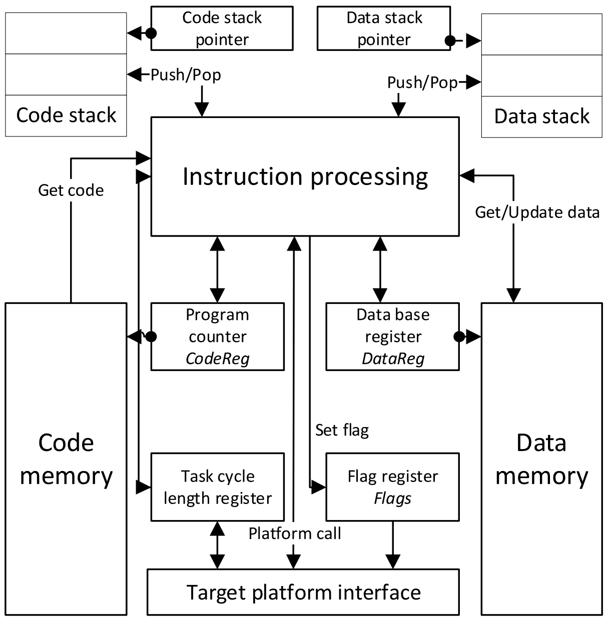


Fig. 1. Architecture of the virtual machine

index outside the valid range, an unknown instruction code, or a cold start.

The virtual machine, as designed specifically for execution of control programs, can handle all IEC 6131-3 data types. The number of bytes required to store each such type in the data memory is given in Table I.

There are two kinds of virtual machine instructions:

- functions,
- system procedures.

Examples of some functions are shown in Table II with decreasing priority. The functions return one value each to be written into the variable being the first operand (as said, an accumulator does not exist in this VM) and may have up to 15 other operands. Note that such order is different than in *Static Single Assignment* of dataflow graphs used in typical compilers [11]. Arithmetic operations are executed in limited ranges, depending on the type. In case of integers the ranges are (−128, 127) for SINT, (−32768, 32767) for INT, etc.

Contrary to functions, system procedures do not return values or return more than one. Table III shows typical examples. The procedures control program flow, handle memory, call

TABLE I
Data types and number of bytes

Types	Bytes
BOOL, BYTE, SINT, USINT	1
INT, UINT, WORD	2
REAL, DINT, UDINT, DWORD	4
DATE, TIME, TIME_OF_DAY	4
LREAL, LINT, ULINT, LWORD	8
DAY_AND_TIME	8
STRING, WSTRING	var

TABLE II
FUNCTIONS OF THE VIRTUAL MACHINE

Mnemonic	Meaning	Operator
EXPT	Power	**
NEG	Negation	− (unary)
MUL	Multiplication	*
DIV	Division	/
ADD	Addition	+ (arithm.)
SUB	Subtraction	− (arithm.)
CONCAT	String join	+ (text)
GT	Greater	>
GE	Greater or equal	>=
LE	Less or equal	<=
LT	Less	<
EQ	Equal	=
NE	Not equal	<>
AND	Logical and	&

subprograms, etc. From the programmers' viewpoint there is no major difference in using functions or procedures.

TABLE III
SYSTEM PROCEDURES OF THE VIRTUAL MACHINE

Mnemonic	Meaning
JMP	Unconditional jump
JNZ	Conditional jump
JR	Unconditional relative jump
JRN	Conditional relative jump
CALB	Subroutine call
RETURN	Return from subroutine
MEMCP	Copy memory block
MCD	Initialize data
FPAT	Fill memory block
GARD	Copy global memory to local area
GAWR	Copy local memory to global area

To accommodate exception handling, the architecture of the virtual machine needed to be expanded. The modifications mainly revolved around the protected code stack, which will be thoroughly explained in Section IV.

III. Runtime formal model

A formal model has been developed to specify the operation of the virtual machine using denotational semantics [12], [13]. The fundamental aspects of the model were outlined in the previous publication [14]. In this context, we now expand upon the description of the model components. The model consists of various domains that define the states, memory functions, value interpreters, limited range operators, and a universal semantic function that invokes specific functions representing individual instructions.

The domains within the model encompass abstract data types that represent the values processed by different components of the virtual machine (Fig. 1). One such domain, denoted as $BasicTypes$, comprises four sets that correspond to the memory sizes of the basic data types outlined in Table I. Another domain, named $Address$, specifies the size of addresses associated with data or code memory. The domain $Memory$, maps $Address$ to $OneByte$. Both $CodeMemory$ and $DataMemory$ are aliases for the $Memory$ domain. The domain $Stack$ represents a sequence (indicated by *) of $Address$ domains, with $CodeStack$ and $DataStack$ serving as aliases for specific stack types. The other domains are defined similarly.

$$\begin{aligned} BasicTypes &= OneByte + TwoBytes + \\ &\quad + FourBytes + EightBytes \\ Address &= FourBytes \\ Memory &= Address \to OneByte \\ CodeMemory &= Memory \\ DataMemory &= Memory \\ Stack &= Address^* \\ CodeStack &= Stack \\ DataStack &= Stack \\ CodeReg &= Address \\ DataReg &= Address \\ Flags &= TwoBytes \end{aligned}$$

Broadly speaking, the goal of program execution is to transition the current state of the computer into a new state. In the context of the virtual machine, the state is represented as a Cartesian product of various domains including memory, stacks, registers, and flags. More specifically, the domain denoted as $State$ can be understood as a collection of tuples $(cm, dm, cs, ds, cr, dr, flg)$, where each element corresponds to a value within its respective domain.

$$\begin{aligned} State &= CodeMemory \times DataMemory \times \\ &\quad \times CodeStack \times DataStack \times \\ &\quad \times CodeReg \times DataReg \times Flags \end{aligned}$$

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

- Get data from memory
  $Get1BMem = (Address \times Memory) \to Byte$
  $Get2BMem = (Address \times Memory) \to TwoBytes$

  $Get4BMem$, $Get8BMem$, etc. are defined similarly.
- Get address from memory
  $GetAddress = (Address \times Memory) \to Address$

  The function returns the value stored at the given $Address$ in $Memory$ which is another $Address$. Since the VM has no accumulator, it operates directly on addresses, and the function $GetAddress$ is essential for the model. $Address$ domain means $TwoBytes$ or $FourBytes$.
- Memory update

  $$\begin{aligned} Upd1BMem &= (Address \times Memory \times \\ &\quad \times OneByte) \to Memory \\ Upd2BMem &= (Address \times Memory \times \\ &\quad \times TwoBytes) \to Memory \end{aligned}$$

  Similarly for $Get4BMem$, $Get8BMem$, etc.

- Memory move
$$\begin{aligned}MemMove = (&Address \times Memory \times \\ &\times Address \times Memory \times \\ &\times OneByte) \to Memory\end{aligned}$$
The source and target $Addresses$ of code or data $Memory$ should be provided. The number of bytes being moved ranges from 0 to 255 ($OneByte$).
- Stack functions
$$Push = (Stack \times Address) \to Stack$$
$$Pop = Stack \to (Address \times Stack)$$
The functions execute stack operations needed by subprograms. Note that Pop returns a pair, viz. $Address$ and new $Stack$.
- Flag operations
$$\begin{aligned}ClearFlag = (&TwoBytes \times TwoBytes) \to \\ &\to TwoBytes\end{aligned}$$
$$\begin{aligned}SetFlag = (&TwoBytes \times TwoBytes) \to \\ &\to TwoBytes\end{aligned}$$
The $Flags$ domain is an alias to $TwoBytes$. The successive $TwoBytes$ above denote actual flags, bits to be set or reset, and new flags.
- Value conversions
$$ByteToWord = OneByte \to TwoBytes$$
$$WordToByte = TwoBytes \to OneByte$$
For a value without a sign, $ByteToWord$ places zero bits into the more significant byte of $TwoBytes$, otherwise the byte is filled with the sign bit. $WordToByte$ reduces the value by removing the most significant bits.

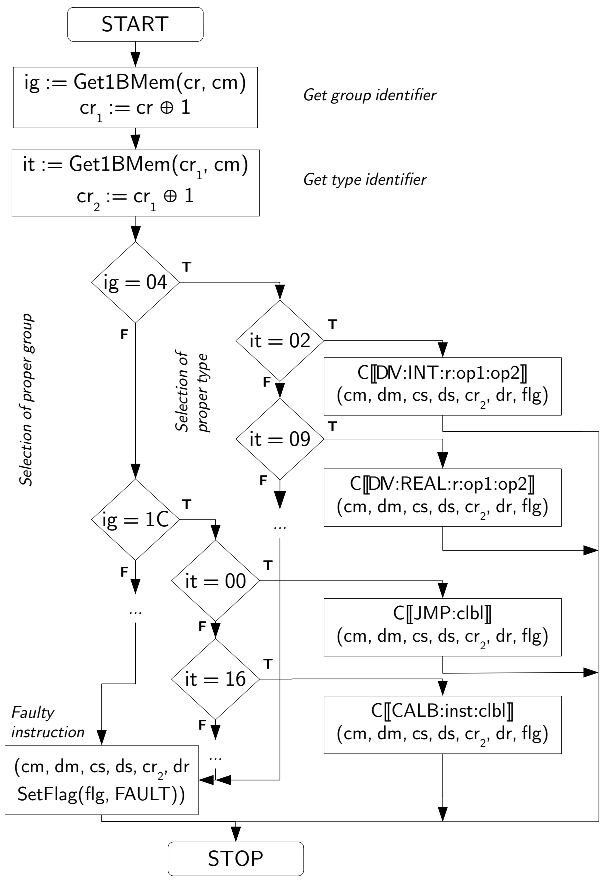
The following sample functions provide numerical interpretations of $OneByte$, $TwoBytes$ and two other memory chunks.

$$\begin{aligned}BoolOf &= OneByte \to BOOL\\ FromBool &= BOOL \to OneByte\\ IntOf &= TwoBytes \to INT\\ FromInt &= INT \to TwoBytes\\ DIntOf &= FourBytes \to DINT\\ FromDInt &= DINT \to FourBytes\\ LIntOf &= EightBytes \to LINT\\ FromLInt &= LINT \to EightBytes\end{aligned}$$

Other types are interpreted analogously.

The numeric identifiers of VM instructions consist of the identifier of a group `ig` and the identifier `it` of a particular data type or procedure. In this way type-specific instructions or procedures may be selected. For some functions `it` also indicates the number of inputs.

To collectively represent the concept of decoding a group and type, followed by the execution of a specific instruction, a universal function $\mathcal{U}$ has been defined. The algorithm of the function $\mathcal{U}$ is presented in Fig. 2. It is assumed that the code register `cr` initially points to the group identifier `ig` in code memory `cm`. The algorithm starts by fetching `ig` (one byte) and incrementing `cr` to $\mathtt{cr}_1$. Then `it` is acquired and the code register incremented to $\mathtt{cr}_2$. At this moment the state of the VM is described by the tuple $(cm, dm, cs, ds, cr_2, dr, flg)$ involving memories, stacks, registers and flags. For instance, for `ig=04` and `it=02` the DIV function is called with the two operands `op1`, `op2` of type INT, whereas `it=09` means operands of type REAL. The last group `ig=1C` consists of system procedures, including JMP and CALB.



START
ig := Get1BMem(cr, cm)
cr₁ := cr ⊕ 1
Get group identifier
it := Get1BMem(cr₁, cm)
cr₂ := cr₁ ⊕ 1
Get type identifier
ig = 04
T
F
Selection of proper group
it = 02
T
F
Selection of proper type
C⟦DIV:INT:r:op1:op2⟧ (cm, dm, cs, ds, cr₂, dr, flg)
it = 09
T
F
C⟦DIV:REAL:r:op1:op2⟧ (cm, dm, cs, ds, cr₂, dr, flg)
...
ig = 1C
T
F
it = 00
T
F
C⟦JMP:clbl⟧ (cm, dm, cs, ds, cr₂, dr, flg)
...
it = 16
T
F
C⟦CALB:inst:clbl⟧ (cm, dm, cs, ds, cr₂, dr, flg)
Faulty instruction
(cm, dm, cs, ds, cr₂, dr SetFlag(flg, FAULT))
...
STOP

Fig. 2. Algorithm of the universal function $\mathcal{U}$

The semantics of the DIV function for INT-type operands is shown in Listing 1. The function divides two operands of type INT, sv denotes the value of the division. The result of a function execution is stored in the location labeled by the first operand, here denoted by `r`. The updated data memory is the second element of s_1 as the result of invoking $Upd2BMem$. The number stored at $raddr$ is given by $FromInt(sv)$.

The procedure GAWR presented in Listing 2 is used in algorithms involving arrays. This procedure copies elements of an array in local memory to an array in global memory. The arrays may contain elements of any type, including arrays and structures. There are four operands, source `src` and destination `dst` labels, `size` of the elements, and array index `idx`. The values $size$ and idx are addresses to data of type WORD. Since the operand `dst` refers to global memory, its

Listing 1 The semantic equation of the DIV function

$$\begin{aligned}
\mathcal{C}[\![\texttt{DIV:INT:r:op1:op2}]\!] = \lambda s.\\
(cm, dm, cs, ds, cr, dr, flg) :=\ s\\
r :=\ GetAddress(cr, cm)\\
raddr :=\ dr \oplus r\\
cr_1 :=\ cr \oplus AddressSize\\
op1 :=\ GetAddress(cr_1, cm)\\
op1addr :=\ dr \oplus op1\\
cr_2 :=\ cr_1 \oplus AddressSize\\
op2 :=\ GetAddress(cr_2, cm)\\
op2addr :=\ dr \oplus op2\\
cr_3 :=\ cr_2 \oplus AddressSize\\
sv :=\ IntOf(Get2BMem(op1addr, dm)) \div\\
\div\ IntOf(Get2BMem(op2addr, dm))\\
s_1 :=\ (cm, Upd2BMem(raddr, dm,\\
FromInt(sv)), cs, ds, cr_3, dr, flg)\\
s_1
\end{aligned}$$

value dst is a direct address (zero dr). The $resultaddr$ is the sum of address dst and the product of $idxval$ and $sizeval$.

Listing 2 The semantic equation of the GAWR procedure

$$\begin{aligned}
\mathcal{C}[\![\texttt{GAWR:dst:src:size:idx}]\!] = \lambda s.\\
(cm, dm, cs, ds, cr, dr, flg) :=\ s\\
dst :=\ GetAddress(cr, cm)\\
cr_1 :=\ cr \oplus AddressSize\\
src :=\ GetAddress(cr_1, cm)\\
srcaddr :=\ dr \oplus src\\
cr_2 :=\ cr_1 \oplus AddressSize\\
size :=\ GetAddress(cr_2, cm)\\
sizeaddr :=\ dr \oplus size\\
sizeval :=\ WordOf(Get2BMem(\\
sizeaddr, dm))\\
cr_3 :=\ cr_2 \oplus AddressSize\\
idx :=\ GetAddress(cr_3, cm)\\
idxaddr :=\ dr \oplus idx\\
idxval :=\ WordOf(Get2BMem(\\
idxaddr, dm))\\
cr_4 :=\ cr_3 \oplus AddressSize\\
resultaddr :=\ dst \oplus idxval \otimes sizeval\\
um :=\ MemMove(dm, resultaddr, dm,\\
srcaddr, sizeval)\\
s_1 :=\ (cm, um, cs, ds, cr_4, dr, flg)\\
s_1
\end{aligned}$$

It is important to note that the equations provided for the DIV and GAWR procedures do not account for erroneous operands, such as a divisor equal to zero or an array index out of bounds. Consequently, to address these failures and prevent unpredictable behavior, the model had to be extended with an exception mechanism.

IV. Adding exception handling

Exceptions have been introduced to replace a sequence of nested `if-else` instructions when performing compound operations that may fail under certain circumstances. The complex branching of algorithm paths, depicted in Figure 3, can be challenging to analyze and distinguish between the normal execution path and the path taken to handle failures. To address this issue, some programming languages have introduced a `try-catch` construct, which separates the algorithm's main path (protected code) from the failure handling path. This approach allows programmers to focus on the operations that the algorithm needs to perform, while storing the failure handling logic in a separate section of the code.

When a failure occurs, it is reported through an exception, which can take various forms, ranging from a simple value like a number or string to a specifically designed object. The presence of an exception terminates the execution of the remaining instructions within the protected code. Subsequently, the processing of the first `catch` clause begins, but only if it matches the type of the exception object. If the exception does not match the type of the first `catch` clause, the subsequent `catch` clauses will be checked for a match. If no further `catch` clauses are found, the execution switches to the surrounding `try-catch` construct. However, if such a construct is not present, the execution is terminated with an unhandled exception state, preventing further execution.

Therefore, the revised code based on Figure 3 would resemble the examples provided in the code snippet shown in Listing 3. However, it is important to note that such code may overlook certain critical tasks that must be performed even if an exception occurs. To address this concern, the `try-catch` construct has been enhanced with an additional clause called `finally`. The `finally` clause contains the code that is always executed when the control exits the protected section of code, irrespective of whether a matched exception occurs, an unhandled exception is encountered, or no exception occurs at all.

The IEC 61131-3:2013 standard does not include constructs for writing code in an exception-style manner. However, certain manufacturers offer their own extensions to the ST language to support such functionality. For instance, the CODESYS development environment provides the following keywords to indicate protected code:

- `__TRY` – beginning of the protected code,
- `__CATCH` – point where failure path of code begins,
- `__FINALLY` – beginning of mandatory code executed always,
- `__ENDTRY` – end of the protected code.

```
if(oper1())
  if(oper2())
    if(oper3())
     if(oper4())
        return success();
     else return error("oper4");
    else return error("oper3");
  else return error("oper2");
else return error("oper1");
```

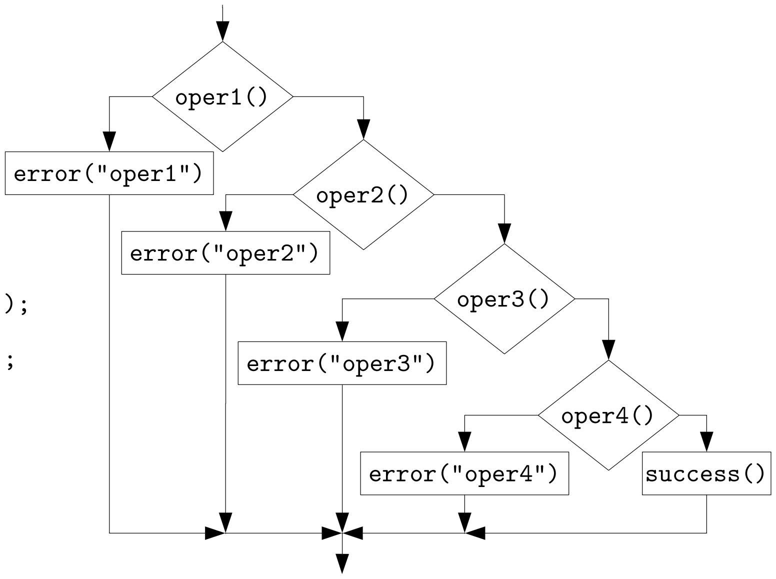


Fig. 3. An example of nested `if` commands and their block diagram

Listing 3 The `try-catch` approach

```
try {
    oper1();
    oper2();
    oper3();
    oper4();
    return success;
} catch(Exception1 e1) { ...
} catch(Exception2 e2) { ...
}
```

In our solution, it was decided to use keywords from CODESYS for exception handling for greater compatibility. In addition to the above keywords, `__THROW` keyword for throwing user-defined exceptions as in the general purpose programming languages (e.g. C++, Java, C#) is added.

To accommodate these language constructs, several changes need to be made to the denotational model of the virtual machine. Firstly, the $State$ tuple needs to be expanded to include two additional components: $ProtStack$ and $ExcObj$. $ProtStack$ represents a stack (Kleene closure) of $ProtEntry$ tuples, which consist of four addresses. To modify the $ProtStack$, the following functions are introduced: $PushProt$, $PopProt$, and $PeekProt$. The $PushProt$ function adds an item to the stack, $PopProt$ removes an item from the stack, and $PeekProt$ returns a copy of the topmost item without modifying the stack. $ExcObj$ is an $Address$ that indicates the location where the exception object has been stored. Since an $Address$ always refers to a memory location, a new flag, `EXCOBJ`, needs to be introduced in the $Flags$ mask to indicate the presence of the exception object.

TABLE IV
SYSTEM PROCEDURES FOR EXCEPTIONS

Name	Operand	Type
`PHPRS`	`excAddr`	`:gclabel`
	`finAddr`	`:gclabel`
	`contAddr`	`:gclabel`
`RAISE`	`excObj`	`:gdlabel`
`MEXCT`	`excTyp`	`:rdlabel`
	`nxtm`	`:gclabel`
`POPRS`	—	—
`CEXCF`	—	—

$$
\begin{aligned}
State = {} & CodeMemory \times DataMemory \times \\
& \times CodeStack \times DataStack \times CodeReg \times \\
& \times DataReg \times Flags \times ProtStack \times ExcObj
\end{aligned}
$$

$$
\begin{aligned}
ProtStack &= ProtEntry^* \\
ProtEntry &= Address \times Address \times Address \times Address \\
PushProt &= (ProtStack \times ProtEntry) \to ProtStack \\
PopProt &= ProtStack \to (ProtEntry \times ProtStack) \\
PeekProt &= ProtStack \to ProtEntry
\end{aligned}
$$

In order to handle the `__TRY`, `__CATCH`, `__FINALLY`, `__ENDTRY` keywords, as well as the additional `__THROW`, new system procedures are required. These procedures are listed in Table IV.

When encountering the `__TRY` instruction, the ST language compiler should generate the system procedure `PHPRS`. The purpose of this procedure is to store the current DPTR context, the address of the first `__CATCH` instruction, the address of the `__FINALLY` instruction, and the address of the `__ENDTRY` instruction on the $ProtStack$. The corresponding denotational model of `PHPRS` is presented below.

```
$\mathcal{C}[\![\texttt{PHPRS:ea:fa:ca}]\!] = \lambda s.$
  $(cm, dm, cs, ds, cr, dr, flg, ps, eo) := s$
  $dptrCtx := dr$
  $excAddr := GetAddress(cr, cm)$
  $cr_1 := cr \oplus AddressSize$
  $finAddr := GetAddress(cr_1, cm)$
  $cr_2 := cr_1 \oplus AddressSize$
  $contAddr := GetAddress(cr_2, cm)$
  $cr_3 := cr_2 \oplus AddressSize$
  $se := (dptrCtx, excAddr, finAddr, contAddr)$
  $ps_1 := PushProt(ps, se)$
  $s_1 := (cm, dm, cs, ds, cr_3, dr, flg, ps_1, eo)$
  $s_1$
```

When the ST compiler encounters the __CATCH statement, it should generate the system procedure MEXCT. The purpose of this procedure is to detect whether the exception type (excTyp operand) matches the current exception. If the exception type matches, the following instructions will be processed and the exception state will be cleared with a jump to the __FINALLY statement. If the exception type does not match, a jump to the next __CATCH statement is performed. If there are no further __CATCH statements, a jump to the __FINALLY keyword is executed. This behavior can be represented using the following denotational equation:

```
$\mathcal{C}[\![\texttt{MEXT:exct:nxtm}]\!] = \lambda s.$
  $(cm, dm, cs, ds, cr, dr, flg, ps, eo) := s$
  $exct := GetAddress(cr, cm)$
  $cr_1 := cr \oplus AddressSize$
  $nxtm := GetAddress(cr_1, cm)$
  $cr_2 := cr_1 \oplus AddressSize$
  $dext := Get4BMem(dm, dr \oplus exct \oplus typeOffset)$
  $sext := Get4BMem(dm, eo \oplus typeOffset)$
  $dcr :=$ match $sext = dext$ with
      | true $\to cr_2$
      | false $\to$ match $nxtm = -1$ with
          | true $\to stk := PeekProt(ps)$
              $(dct, ctch, fin, efn) := stk$
              $fin$
          | false $\to nxtm$
          end
      end
  $s_1 := (cm, dm, cs, ds, dcr, dr, flg, ps, eo)$
  $s_1$
```

After the last statement of __CATCH, the compiler should generate the system procedure CEXCF, which marks the end of exception handling. The denotational model of CEXCF can be defined as follows:

```
$\mathcal{C}[\![\texttt{CEXCF}]\!] = \lambda s.$
  $(cm, dm, cs, ds, cr, dr, flg, ps, eo) := s$
  $stk := PeekProt(ps)$
  $(dct, ctch, fin, efn) := stk$
  $flg_1 := ClearFlag(flg, F\_EXCPT)$
  $s_1 := (cm, dm, cs, ds, fin, dr, flg_1, ps, 0)$
  $s_1$
```

No special action is needed when the compiler encounters the __FINALLY statement. However, an action is required when the compiler encounters __ENDTRY in the ST input. In this case, the compiler should emit the POPRS system procedure, which can be represented by the following denotational model:

```
$\mathcal{C}[\![\texttt{POPRS}]\!] = \lambda s.$
  $(cm, dm, cs, ds, cr, dr, flg, ps, eo) := s$
  $tf := SetFlag(flg, F\_EXCPT)$
  $(dcr, ddr, nstk) :=$ match $tf = flg$ with
      | true $\to (ent, stk) := PopProt(ps)$
          $(dct, ctch, fin, efn) := ent$
          $(ctch, dct, stk)$
      | false $\to (ent, stk) := PopProt(ps)$
          $(cr, dr, stk)$
      end
  $s_1 := (cm, dm, cs, ds, dcr, ddr, flg, nstk, eo)$
  $s_1$
```

If a programmer wishes to throw their own exception in ST code, they can use the __THROW keyword. In this case, the ST compiler should emit the RAISE system procedure with the exception object. The denotational model of RAISE can be represented as follows:

```
$\mathcal{C}[\![\texttt{RAISE:excObj}]\!] = \lambda s.$
  $(cm, dm, cs, ds, cr, dr, flg, ps, eo) := s$
  $excObj := GetAddress(cr, cm)$
  $eo_1 := dr \oplus excObj$
  $flg_1 := SetFlag(flg, F\_EXCPT)$
  $stk := PeekProt(ps)$
  $(dr_1, ctch, fin, efn) := stk$
  $s_1 := (cm, dm, cs, ds, ctch, dr_1, flg_1, ps, eo_1)$
  $s_1$
```

V. Implementation in CPDev

An exception is reported in the CPDev virtual machine either automatically or manually. The call is invoked internally, when a system exception condition is met during the program execution. It may be also invoked by the programmer by calling the `RAISE` system procedure to manually trigger the failure path. The automatically generated system exceptions are reported when on of the runtime errors occurs. Selected system exceptions are listed in Table V.

TABLE V
SELECTED SYSTEM EXCEPTIONS

Type	Description
Division by zero	Invalid DIV instruction parameter
Modulo by zero	Invalid MOD instruction operand
Bad array index	Index array access out of bounds
Bad format	Invalid string format during parsing to numeric types (STRING_TO_INT, STRING_TO_WORD, STRING_TO_REAL, etc.)
Cycle overflow	Program execution exceeded the declared cycle time

The corresponding operations performed by th VM have been extended with exception reporting. For example, the semantic description of the DIV function from Listing 1 should be extended which checking for zero divisor as follows:

```
$divisor := IntOf(Get2BMem(op2addr, dm))$
match $divisor$ with
| 0 → $excObj := (cr_3, DIV\_BY\_ZERO\_EXC)$
    $\mathcal{C}[\![\texttt{RAISE:op}]\!](cm, dm, cs, ds,$
        $cr_3, dr, flg, ps, excObj)$
| _ → $sv := IntOf(Get2BMem(op1addr,$
    $dm)) \div divisor$
    $s_1 := (cm, Upd2BMem(raddr, dm,$
    $FromInt(sv)), cs, ds, cr_3, dr, flg, ps, eo)$
    $s_1$
end
```

Listing 4 shows the `DIV` instruction utilizing a C macro for division of several numeric types. The function `IG_DIV_04` implements the division for all relevant data types (group), thus avoiding repetitions of rather similar code. The function calls the parameterized macrodefinition `DIV_TYPE` which is common for all types. The value of an operand of a particular `TYPE` is determined in `DIV_TYPE` by the function `TYPE##Of` with given `sizeof(TYPE)`. The code calls an internal function `WM_RaiseException` in the case when the second operand (divisor) is zero. The division result `cmp` updates the INT value at `raddr` (`Upd2BMemData` also increments the code register). The function `IG_DIV_04` recognizes a particular type as the second nibble (half byte) of the type identifier `it` by masking `it & 0x0F`. Note that `case` and `break` are hidden for `switch` in the `DIV_TYPE` definition.

Listing 5 contains a simplified implementation of the GAWR procedure (Sec. III). The check is made if the parameter corresponding to the array index is less than zero. If so, the system exception is raised (EX_ARRAY_IDX).

In the case of an exception, the virtual machine examines the `ProtStack`. If the stack is empty, no `__TRY...__CATCH`

Listing 4 Implementation of the DIV instruction

```
#define DIV_TYPE(TYPE)  \
case IT_DIV_##TYPE & 0x000F: \
{ \
 TYPE sv = 0; \
 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))); \
 if (op2 == 0) \
           WM_RaiseException(EX_DIV0); \
 else { \
   sv = op1 / op2;\
   UpdMemData(raddr, From##TYPE(sv), \
     sizeof(sv)); } \
} \
break;


void IG_DIV_04(BYTE it)
{
        switch (it & 0x0F)
        {
                DIV_TYPE(SINT)
                DIV_TYPE(INT)
                DIV_TYPE(DINT)
                DIV_TYPE(LINT)
                DIV_TYPE(BYTE)
                DIV_TYPE(WORD)
                DIV_TYPE(DWORD)
                DIV_TYPE(LWORD)
                DIV_TYPE(REAL)
                DIV_TYPE(LREAL)
        default:  /* unknown code */
                flag |= FAULT;
        }
        return;
}
```

block has been defined by the programmer. In such a case, the system may perform one of the predefined actions:

- stop execution and go into a fail-safe state (set outputs to safe values)
- perform a cold start of the controller
- perform a warm start
- restart the program cycle.

Listing 6 contains a simple ST code using the exception-related keywords. The program includes a declaration of an array `BA` indexed from 0 to 10. The protected code between

Listing 5 Implementation of the GAWR instruction

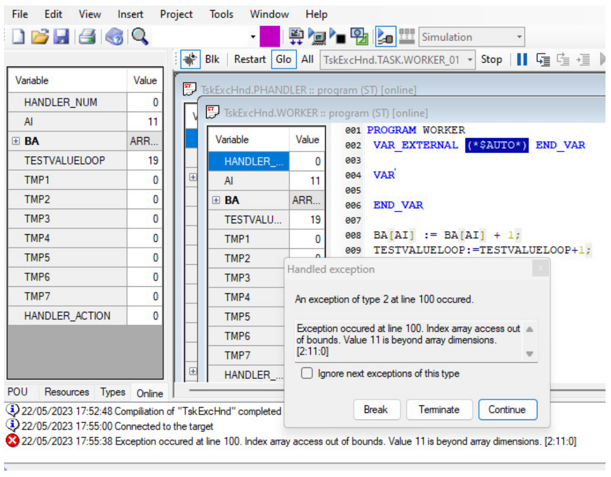
```
case VMF_GAWR & 0xFF:
{
  ADDRESS src = GetCodeAddress();
  ADDRESS dst = GetCodeAddress();
  INT idx = getINT(GetCodeAddress());
  BYTE size = getBYTE(GetCodeAddress());

  if (idx < 0)
    WM_RaiseException(EX_ARRAY_IDX);
  else
    memcpy(dst+idx*size, src, size);
}
break;
```

Listing 6 Exception handling in ST code

```
PROGRAM WORKER
  VAR
    BA : ARRAY[0..10] OF INT;
    AI : INT;
    RES, VALUE, SCALE : REAL;
    DIV_EX : DIV_BY_ZERO_EXCEPTION;
    OTHER_EX : ANY_EXCEPTION;
  END_VAR

  __TRY

    BA[AI] := BA[AI] + 1;
    RES := VALUE / SCALE;

  __CATCH(DIV_EX)

    SCALE := 1;
    DIV_EX.ACTION := RESTART_CYCLE;

  __CATCH(OTHER_EX)

    OTHER_EX.ACTION := TERMINATE;

  __ENDTRY

END_PROGRAM
```

`__TRY` and `__CATCH` increases the array element at the index pointed by the value of the variable `AI`. In case of a division by zero exception during the operation, the failure path from `__CATCH(DIV_EX)` is executed. If any other exception occurs, the next failure path from `__CATCH(OTHER_EX)` is taken. The exception block restores the `AI` value to the acceptable value of 0 and instructs the virtual machine to restart the program cycle.

An exception is usually an unexpected behavior, so CPDev IDE provides a set of tools to debug such situations. Figure 4 shows the CPDev Integrated Environment (IDE) running a program in a simulation mode. As one may observe, the exception has occured due to the fact, that the variable `AI` (array index) has been set to 11, so outside the array bounds. The environment allows to break the execution to examine the cause, to terminate the program completely or to continue with the default action for an exception.



Variable	Value
HANDLER_NUM	0
AI	11
BA	ARR...
TESTVALUELOOP	19
TMP1	0
TMP2	0
TMP3	0
TMP4	0
TMP5	0
TMP6	0
TMP7	0
HANDLER_ACTION	0

Variable	Value
HANDLER_...	0
AI	11
BA	ARR...
TESTVALU...	19
TMP1	0
TMP2	
TMP3	
TMP4	
TMP5	
TMP6	
TMP7	
HANDLER_...	

```
PROGRAM WORKER
 VAR_EXTERNAL (*$AUTO*) END_VAR

 VAR

 END_VAR

 BA[AI] := BA[AI] + 1;
 TESTVALUELOOP:=TESTVALUELOOP+1;
```

Handled exception

An exception of type 2 at line 100 occured.

Exception occured at line 100. Index array access out of bounds. Value 11 is beyond array dimensions. [2:11:0]

☐ Ignore next exceptions of this type

```
22/05/2023 17:52:48 Compiliation of "TskExcHnd" completed
22/05/2023 17:55:00 Connected to the target
22/05/2023 17:55:38 Exception occured at line 100. Index array access out of bounds. Value 11 is beyond array dimensions. [2:11:0]
```

Fig. 4. Debugging exceptions in CPDev IDE

VI. Final remarks

The introduction of exceptions into the control environment based on a virtual machine was driven by industry demands. The presented concepts from a programmer's perspective resemble solutions available in high-level object oriented programming languages like C# or Java. However, in this case, the solution needs to be applied to embedded controllers with limited resources and performance.

To address this, the proposed mechanism for exception handling was designed to minimize the extra operations performed by the CPU. The requirement for additional memory to accommodate the $ProtStack$ is relatively easy to fulfill, even for small devices. It is anticipated that incorporating exception infrastructure supported by a formal model will facilitate the development of more robust and reliable control solutions.

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 2019–2023. Project number 027/RID/2018/19, amount granted 11 999 900 PLN.

References

[1] *IEC 61131-3. Programmable Controllers. Part 3. Programming languages*. International Standard: IEC, 2013.
[2] *IEC 61499 — Function blocks*. International Standard: IEC, 2015.
[3] M. Simros, M. Wollschlaeger, and S. Theurich, "Programming embedded devices in IEC 61131-languages with industrial PLC tools using PLCopen XML," in *Proceedings of the CONTROLO'2012 Portuguese Conference on Automatic Control, Funchal, Portugal*, 2012. ISBN 9789729702532 pp. 51–56.

[4] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java® Virtual Machine Specification*. Oracle America, Inc., 2013. ISBN 9780133260441

[5] T. L. Thai and L. H., *.NET Framework Essentials*. O'Reilly Media, 2003. ISBN 9780596005054

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

[7] A. Blikle, "An experiment with denotational semantics," *SN Computer Science*, vol. 15, no. 1, pp. 1–31, 2020. doi: 10.1007/s42979-019-0013-0

[8] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Programming controllers in structured text language of IEC 61131-3 standard," *Journal of Applied Computer Science*, vol. 16, no. 1, pp. 49–67, 2008.

[9] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Open environment for programming small controllers according to IEC 61131-3 standard," *Scalable Computing: Practice and Experience*, vol. Volume 10, no. 3, pp. 325–336, 2009.

[10] B. Trybus, "Development and Implementation of IEC 61131-3 Virtual Machine," *Theoretical and Applied Informatics*, vol. 23, no. 1, pp. 21–35, 2011. doi: 10.2478/v10179-011-0002-z

[11] K. Cooper and L. Torczon, *Engineering a Compiler*. San Francisco: Morgan Kaufmann, 2022. ISBN 9780128154120

[12] K. Slonneger and B. L. Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory-Based Approach*. Addison-Wesley Publishing Company, Inc, 1995. ISBN 9780201656978

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

[14] J. Sadolewski and B. Trybus, "Denotational model and implementation of scalable virtual machine in CPDev," in *Proceedings of the 17th Conference on Computer Science and Intelligence Systems*, M. Ganzha, L. Maciaszek, M. Paprzycki, and D. Ślęzak, Eds., vol. 30, 2022. doi: 10.15439/2022F236 pp. 587–591.