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 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>
<thead>
<tr>
<th>Table I</th>
<th>DATA TYPES AND NUMBER OF BYTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Bytes</td>
</tr>
<tr>
<td>BOOL, BYTE, SINT, USINT</td>
<td>1</td>
</tr>
<tr>
<td>INT, UINT, WORD</td>
<td>2</td>
</tr>
<tr>
<td>REAL, DINT, UDINT, DWORD</td>
<td>4</td>
</tr>
<tr>
<td>DATE, TIME, TIME_OF_DAY</td>
<td>4</td>
</tr>
<tr>
<td>LREAL, LINT, ULIINT, LWORLD</td>
<td>8</td>
</tr>
<tr>
<td>DAY_AND_TIME</td>
<td>8</td>
</tr>
<tr>
<td>STRING, WSTRING</td>
<td>var</td>
</tr>
</tbody>
</table>
subprograms, etc. From the programmers’ viewpoint there is no major difference in using functions or procedures.

### Table II

**Functions of the virtual machine**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPT</td>
<td>Power</td>
<td>+</td>
</tr>
<tr>
<td>NEG</td>
<td>Negation</td>
<td>- (unary)</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiplication</td>
<td>×</td>
</tr>
<tr>
<td>DIV</td>
<td>Division</td>
<td>/</td>
</tr>
<tr>
<td>ADD</td>
<td>Addition</td>
<td>+ (arithm.)</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtraction</td>
<td>- (arithm.)</td>
</tr>
<tr>
<td>CONCAT</td>
<td>String join</td>
<td>+ (text)</td>
</tr>
<tr>
<td>GT</td>
<td>Greater</td>
<td>&gt;</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or equal</td>
<td>≥</td>
</tr>
<tr>
<td>LE</td>
<td>Less or equal</td>
<td>≤</td>
</tr>
<tr>
<td>LT</td>
<td>Less</td>
<td>&lt;</td>
</tr>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>=</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
<td>≠</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
<td>∧</td>
</tr>
</tbody>
</table>

### Table III

**System procedures of the virtual machine**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP</td>
<td>Unconditional jump</td>
</tr>
<tr>
<td>JNZ</td>
<td>Conditional jump</td>
</tr>
<tr>
<td>JR</td>
<td>Unconditional relative jump</td>
</tr>
<tr>
<td>JRN</td>
<td>Conditional relative jump</td>
</tr>
<tr>
<td>RETURN</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>MEMCP</td>
<td>Copy memory block</td>
</tr>
<tr>
<td>MCD</td>
<td>Initialize data</td>
</tr>
<tr>
<td>FPAT</td>
<td>Fill memory block</td>
</tr>
<tr>
<td>GARD</td>
<td>Copy global memory to local area</td>
</tr>
<tr>
<td>GAWR</td>
<td>Copy local memory to global area</td>
</tr>
</tbody>
</table>

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

- **Get data from memory**
  
  \[\text{Get1BMem} = (\text{Address} \times \text{Memory}) \to \text{Byte}\]
  \[\text{Get2BMem} = (\text{Address} \times \text{Memory}) \to \text{TwoBytes}\]

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

- **Memory update**

  \[\text{Upd1BMem} = (\text{Address} \times \text{Memory} \times \text{OneByte}) \to \text{Memory}\]
  \[\text{Upd2BMem} = (\text{Address} \times \text{Memory} \times \text{TwoBytes}) \to \text{Memory}\]

Similarly for `GetABMem`, `Get8BMem`, etc.
• Memory move
  \[ \text{MemMove} = (\text{Address} \times \text{Memory} \times \text{Address} \times \text{Memory} \times \text{OneByte}) \rightarrow \text{Memory} \]

The source and target \textit{Addresses} of code or data \textit{Memory} should be provided. The number of bytes being moved ranges from 0 to 255 (\text{OneByte}).

• Stack functions
  \[ \text{Push} = (\text{Stack} \times \text{Address}) \rightarrow \text{Stack} \]
  \[ \text{Pop} = \text{Stack} \rightarrow (\text{Address} \times \text{Stack}) \]

The functions execute stack operations needed by subprograms. Note that \text{Pop} returns a pair, viz. \text{Address} and new \text{Stack}.

• Flag operations
  \[ \text{ClearFlag} = (\text{TwoBytes} \times \text{TwoBytes}) \rightarrow \text{TwoBytes} \]
  \[ \text{SetFlag} = (\text{TwoBytes} \times \text{TwoBytes}) \rightarrow \text{TwoBytes} \]

The \textit{Flags} domain is an alias to \textit{TwoBytes}. The successive \textit{TwoBytes} above denote actual flags, bits to be set or reset, and new flags.

• Value conversions
  \[ \text{ByteToWord} = \text{OneByte} \rightarrow \text{TwoBytes} \]
  \[ \text{WordToByte} = \text{TwoBytes} \rightarrow \text{OneByte} \]

For a value without a sign, \text{ByteToWord} places zero bits into the more significant byte of \text{TwoBytes}, otherwise the byte is filled with the sign bit. \text{WordToByte} reduces the value by removing the most significant bits.

The following sample functions provide numerical interpretations of \textit{OneByte}, \textit{TwoBytes} and two other memory chunks.

\[ \text{BoolOf} = \text{OneByte} \rightarrow \text{BOOL} \]
\[ \text{FromBool} = \text{BOOL} \rightarrow \text{OneByte} \]
\[ \text{IntOf} = \text{TwoBytes} \rightarrow \text{INT} \]
\[ \text{FromInt} = \text{INT} \rightarrow \text{TwoBytes} \]
\[ \text{DIntOf} = \text{FourBytes} \rightarrow \text{DINT} \]
\[ \text{FromDInt} = \text{DINT} \rightarrow \text{FourBytes} \]
\[ \text{LIntOf} = \text{EightBytes} \rightarrow \text{LINT} \]
\[ \text{FromLInt} = \text{LINT} \rightarrow \text{EightBytes} \]

Other types are interpreted analogously.

The numeric identifiers of VM instructions consist of the identifier of a group \textit{ig} and the identifier \textit{it} of a particular data type or procedure. In this way type-specific instructions or procedures may be selected. For some functions \textit{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 \(U\) has been defined. The algorithm of the function \(U\) is presented in Fig. 2. It is assumed that the code register \textit{cr} initially points to the group identifier \textit{ig} in code memory \textit{cm}. The algorithm starts by fetching \textit{ig} (one byte) and incrementing \textit{cr} to \textit{cr1}. Then it is acquired and the code register incremented to \textit{cr2}. At this moment the state of the VM is described by the tuple \((\text{cm}, \text{dm}, \text{cs}, \text{ds}, \text{cr}, \text{dr}, \text{flg})\) involving memories, stacks, registers and flags. For instance, for \textit{ig}=04 and \textit{it}=02 the DIV function is called with the two operands \textit{op1}, \textit{op2} of type \text{INT}, whereas \textit{it}=09 means operands of type \text{REAL}. The last group \textit{ig}=1C consists of system procedures, including JMP and CALB.

The semantics of the DIV function for \text{INT}-type operands is shown in Listing 1. The function divides two operands of type \text{INT}, \textit{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 \textit{r}. The updated data memory is the second element of \textit{s1} as the result of invoking \text{Upd2BMem}. The number stored at \textit{raddr} is given by \text{FromInt} (\textit{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 \textit{src} and destination \textit{dst} labels, \textit{size} of the elements, and array index \textit{idx}. The values \textit{size} and \textit{idx} are addresses to data of type \text{WORD}. Since the operand \textit{dst} refers to global memory, its
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 ... 
oper1()
oper2()
oper3()
oper4()
error("oper1")
error("oper2")
error("oper3")
error("oper4") success()

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

Listing 3 The try-catch approach

```c
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.

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Operand</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHPRS</td>
<td>excAddr</td>
<td>glabel</td>
</tr>
<tr>
<td></td>
<td>finAddr</td>
<td>glabel</td>
</tr>
<tr>
<td></td>
<td>contAddr</td>
<td>glabel</td>
</tr>
<tr>
<td>RAISE</td>
<td>excTyp</td>
<td>rdlabel</td>
</tr>
<tr>
<td>MEXT</td>
<td>nxml</td>
<td>glabel</td>
</tr>
<tr>
<td>POPRS</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CEXCF</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

State = CodeMemory × DataMemory × CodeStack × DataStack × CodeReg × DataReg × Flags × ProtStack × ExcObj

ProtStack = ProtEntry*

PushProt = (ProtStack × ProtEntry) → ProtStack
PopProt = ProtStack → (ProtEntry × ProtStack)

PeekProt = ProtStack → ProtEntry

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.
When the ST compiler encounters the __CATCH statement, it should generate the system procedure MEXIT. The purpose of this procedure is to determine 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:

\[
C[MEXIT:exc:nxtm] = \lambda s.
\]

\[
(cm, dm, cs, ds, cr, dr, flg, ps, eo) := s
\]

\[
dptrCtx := dr
\]

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

\[
cr_1 := cr \oplus \text{AddressSize}
\]

\[
finAddr := \text{GetAddress}(cr_1, cm)
\]

\[
cr_2 := cr_1 \oplus \text{AddressSize}
\]

\[
contAddr := \text{GetAddress}(cr_2, cm)
\]

\[
cr_3 := cr_2 \oplus \text{AddressSize}
\]

\[
s := (dptrCtx, excAddr, finAddr, contAddr)
\]

\[
ps_1 := \text{PushProt}(ps, s)
\]

\[
s_1 := (cm, dm, cs, ds, cr_3, dr, flg, ps_1, eo)
\]

\[
s_1
\]

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 POPRS system procedure with the exception object. The denotational model of RAISE can be represented as follows:

\[
C[RAISE:excObj] = \lambda s.
\]

\[
(cm, dm, cs, ds, cr, dr, flg, ps, eo) := s
\]

\[
tf := \text{SetFlag}(flg, F_{EXCPT})
\]

\[
dcr, dd, nstk) := \text{match } tf = flg \text{ with }
\]

\[
| \text{true} \rightarrow (ent, stk) := \text{PopProt}(ps)
\]

\[
| \text{false} \rightarrow (ent, stk) := \text{PopProt}(ps)
\]

\[
(cm, dr, stk)
\]

\[
s_1 := (cm, dm, cs, ds, dcr, dd, flg, nstk, eo)
\]

\[
s_1
\]

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 one of the runtime errors occurs. Selected system exceptions are listed in Table V.

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

The corresponding operations performed by the 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:

```c
int divisor = IntOf(Get2BMem(op2addr, dm))
```

```c
match divisor with
| 0 → excObj := (cr3, DIV_BY_ZERO_EXC) |
| _ → sv := IntOf(Get2BMem(op1addr, dm)) / divisor |
```

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 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

```c
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

```c
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 occurred 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.

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 Ministry 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


