Article

Verification of Control System Runtime Using an Executable Semantic Model †

Jan Sadolewski * and Bartosz Trybus *‡

Department of Computer and Control Engineering, Rzeszow University of Technology, 35-959 Rzeszów, Poland
* Correspondence: js@kia.prz.edu.pl (J.S.); btrybus@kia.prz.edu.pl (B.T.); Tel.: +48-17-865-12-25 (J.S. & B.T.)
† The journal paper is an extended version of conference paper.
‡ These authors contributed equally to this work.

Abstract: The paper outlines a methodology for validating the accuracy of a control system’s runtime implementation. The runtime takes the form of a virtual machine executing portable code compliant with IEC 61131-3 standards. A formal model, comprising denotational semantics equations, has been devised to specify machine instruction decoding and operations, including arithmetic functions across various data types, arrays, and subprogram calls. The model also encompasses exception-handling mechanisms for runtime errors, such as division by zero and invalid array index access. This denotational model is translated into executable form using the functional F♯ language. Verification involves comparing the actual implementation of the virtual machine against this executable model. Any disparities between the model and implementation indicate deviations from the specification. Implemented within the CPDev engineering environment, this approach ensures consistent and predictable control program execution across different target platforms.

Keywords: denotational semantics; model checking; IEC 61131-3; control software; F♯; runtime

1. Introduction

The predictable and reliable operation of control systems is an essential requirement in industrial practice. Programmable automation controllers (PACs) and programmable logic controllers (PLCs) provide flexibility in constructing control programs, which can lead to significant negative consequences if they are improperly prepared. To avoid this, various approaches exist. Some of them, which can be classified as software engineering, emphasize the proper execution of the software development process, reducing the likelihood of errors by the programmer [1]. In the case of control systems, this can also be achieved by using specially designed programming languages tailored for such applications. A typical example is the set of languages ST, FBD, LD, IL, and SFC, which are part of the industrial standard IEC 61131-3 [2]. It also defines the appropriate architecture for control software. Other concepts focus on preparing suitable testing procedures for such applications, aimed at detecting errors in control software before it is deployed in industrial conditions [3].

One solution considered in this work is the development of a specialized runtime environment designed to prevent the propagation of errors made by the programmer to the entire system. The occurrence of a critical runtime error in the program should be detected and prohibited. Such an isolated environment is often called a sandbox and can be implemented using hardware security mechanisms, such as memory protection [4]. However, these solutions require hardware expansion with additional circuits or the use of advanced processors with such capabilities [5]. In embedded systems, which may involve various hardware platforms, such extension is often unavailable and impractical. Hence, there is a need for software solutions that allow the isolation of the program execution environment from the rest of the system.

One option involves the supervised execution of the program by a software interpreter instead of directly by the hardware processor. The interpreter fetches individual instructions
from the program, translates them into processor code, and executes them. During these activities, the interpreter can check whether the execution of instructions will cause errors and react accordingly if it does. The prerequisite is to include a programming language interpreter library in the controller’s firmware, allowing the textual source code to be interpreted and executed [6]. This solution may be disadvantageous in control systems due to the additional memory and time required for analyzing the textual code by the appropriate parser and preparing native code for the target processor. Another option is to prepare a language and tools for the automatic and semi-automatic model analysis of software [7]. However, this approach can be time-consuming and error-prone due to the necessity of developing an additional compiler and an environment for theorem proving.

The solution considered in this article, although based on the concept of an interpreter, does not use textual code but binary code containing encoded instructions along with operands. Therefore, lexical analysis using a parser is not needed at the controller side. Instructions in binary code are defined for a virtual processor with specified characteristics. However, their implementation by the hardware processor is still necessary; hence, the binary code is called intermediate or portable code. In return, we achieve platform independence because the interpreter can translate the instructions of the virtual processor into specific instructions of a real processor, provided that support for the latter is included in the process. In distributed control systems where multiple control devices with varying hardware platforms are interconnected via a network, the software portability enabled by the intermediate code proves particularly advantageous. To execute the intermediate code, the interpreter uses specially allocated code and data memory areas as well as stacks defined programmatically. This creates a Harvard architecture, and the entire runtime environment will be referred to as a virtual machine [8,9] (meaning an application virtual machine, not related to hardware virtualization). Virtual machines have become prominent in IT due to platforms like the Java Runtime Environment [10] and the .NET Framework [11]. However, the architecture of the virtual machine (VM) presented here is specifically tailored for the hardware architecture of control devices and adapted for embedded systems with limited resources and performance.

The implementation must take into account the programming language available for the platform and hardware features such as the type of central processing unit, access to RAM and ROM (flash) memories, clock for time measurement, random number generation, etc. Thus, these implementations vary from platform to platform [12,13]. A high degree of universality is achieved by implementing a large part of the virtual machine in C/C++ languages available for numerous hardware platforms and processors [14]. Another solution, emphasizing performance, is to implement it in the languages for FPGA circuits [15–17].

The correctness of such an implementation must correspond to the specification of the virtual machine. So far, this has been verified using tests. This article presents a different solution based on a formal model of the virtual machine, which takes the form of denotational semantics equations encompassing the architecture, intermediate code processing, and individual instructions. Denotational equations are then written in the form of appropriate constructions of the F♯ functional language, creating a direct representation of the denotational model in programmatic form. Such a form of the model is executable, leading to the possibility of verifying the implementation of the machine for a specific hardware platform. Specifically, the implemented runtime and model with the same input data (intermediate code) are executed side by side. During this execution, the verification mechanism compares the state of the model and the virtual machine, particularly the contents of memory, registers, and stacks. In the event of discrepancies, an error in the machine implementation is reported. This allows for the preparation of the machine in a form that precisely matches the specification.

Since the machine supervises program execution, runtime errors are detected and signaled, and the runtime does not propagate their effects to other parts of the system. However, the programmability of control devices means that during execution, instructions with critical errors may occur, such as writing to a forbidden memory address or accessing
an array with an illegal index. In such cases, the control system becomes inoperative and should transition to a safe state to prevent negative consequences for the controlled object. This type of behavior can be defined by the programmer as a response to runtime errors. Programming environments (e.g., Java, C\textsuperscript{#} [11,18]) provide a mechanism for handling exceptions, which allows to implement specific behavior in response to an error during the program execution.

The previous conference paper [19] introduces a formal model of the exception mechanism tailored for an IEC 61131-3 oriented virtual machine, accompanied by a sample implementation. This paper extends the model into an executable form to conduct model-based checks on the virtual machine implementation. The expansion, coupled with the runtime exception handling, offers a chance to develop automation systems that are more robust and dependable.

2. Operation of the Runtime

Usually, programmers who work with IEC 61131-3 focus on software projects intended for deployment on various controllers. In this context, it is assumed that such a project is executed by a runtime component referred to as an execution unit. A project consists of one or more tasks composed of Program Organization Units (POUs), such as program or function blocks. Consequently, a smaller device typically provides a single execution unit capable of running one project, while a controller with more powerful hardware can accommodate multiple execution units, allowing many projects to run simultaneously as shown in Figure 1a. In this context, an execution unit maintains an instance of the virtual machine, which establishes an isolated environment for executing control programs stored as binary intermediate code, shown as eXecutable Control Program (XCP). In such a solution, a control project is compiled into the binary format and subsequently transferred to the execution unit, where it is executed through the virtual machine. The execution units also interface with platform resources for the virtual machine. An example of leveraging dual-core CPUs for creating a controller with two execution units is outlined in [20].

![Figure 1. Architecture of the runtime. (a) Project execution units. (b) Elements of the virtual machine.](image)

The virtual machine discussed here has been specifically designed to comply with the IEC 61131-3 standard [14]. It is a part of the Control Program Developer (CPDev) programming environment, which facilitates the development of control software for diverse platforms, such as ARM, FPGA, and others [21]. A programmer uses CPDev IDE to prepare POUs using textual languages such as ST or IL, and graphical ones such as FBD, LD and SFC. Then, the compiler generates the intermediate binary code for the virtual machine [21].

The architecture of the CPDev virtual machine is depicted in Figure 1b. The program code is stored in the code memory, from where individual instructions for the virtual machine and their operands are retrieved based on the indications of the CodeReg register (also known as the program counter). In the Instruction Processing module, the instruction is decoded and executed. In the case of reading and writing variables, the data memory is used. The DataReg register indicates the base address of the Program Organizational Unit.
(POU) for which the operation is performed. This enables working with multiple instances of POUs, such as function blocks or programs. Two stacks are used for entering and exiting subprograms: the data stack and the code stack. The data stack stores pointers to recently used database addresses (i.e., DataReg contents), while the code stack keeps track of return addresses (for CodeReg). Upon entering a subprogram, the current values of CodeReg and DataReg are pushed onto the code stack and data stack, respectively. Upon return, the contents of these registers are popped from the stacks. This stack mechanism facilitates nested function blocks.

The virtual machine does not have a special register for storing operation results like an accumulator. Therefore, executing instructions typically involves fetching operands from memory, i.e., data memory for variables and code memory for constants, and storing the result in data memory. The machine incorporates the Flags register, containing status flags signaling errors or unusual situations like an array index beyond the valid range, an unknown instruction code, or a cold start.

The instruction block for the virtual machine comprises three parts as illustrated in Figure 2. Each machine instruction is identified by the \textit{vmcode}, which consists of two components: the group identifier \textit{ig}, and the specific instruction type \textit{it}. This design facilitates the selection of type-specific instructions from a set of similar ones aimed at a specific functionality. Furthermore, for certain instructions that accept a variable number of operands, the \textit{it} component indicates the number of operands. The operand section of the instruction block contains either addresses of the operands in the data memory or immediate values. The size of this section varies depending on the \textit{ig} and \textit{it} components.

![Figure 2. Format of the instruction block.](image)

During the execution of code, a critical error may occur, leading the machine to report an exception. Table 1 displays a selection of system exceptions.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corrupted code</td>
<td>Invalid instruction block</td>
</tr>
<tr>
<td>Wrong memory access</td>
<td>Invalid address to code or data memory</td>
</tr>
<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 (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 programmer has the option to delineate a section of protected code using the constructs \_\_TRY, \_\_ENDTRY, \_\_CATCH, and \_\_FINALLY, which represent an extension beyond the standard [19]. In the event of an exception, the execution of the remaining instructions within the protected code is terminated. The occurrence of the failure is signaled through an exception information object. Following this, the processing of the first matching \_\_CATCH clause begins, corresponding to the type of the exception object. If no matching \_\_CATCH clauses are found, the execution proceeds to the upper-level exception handlers. In cases where there is no matching exception handler, the program’s execution is abruptly terminated with a failure. To facilitate the handling of nested exceptions, the ExceptionStack is employed as depicted in Figure 1b. Each entry in the ExceptionStack comprises the addresses for the code blocks \_\_CATCH, \_\_FINALLY, and \_\_ENDTRY, utilized during the processing of an exception.
3. Runtime Formal Model

3.1. Purpose

The presentation of a system operation and the basics of its functioning through the implementation of its individual components is a somewhat imprecise and informal approach because it depends on the programming constructs of the programming language in which the implementation is made. Since programming constructs in different languages can vary quite significantly, understanding the operation of the application and implementing it in a language other than the original one can be quite cumbersome.

For the control devices in question, the runtime implementation should be tailored for different hardware platforms with various CPUs and characteristics. Typically, this implementation is performed using C/C++ alongside platform-specific mechanisms, though other development scenarios are also feasible [14]. The formal model serves as a reference, providing a generic definition of the runtime components, such as memory types, stacks, instruction decoding, and instruction behavior. This allows the actual implementation to be developed correctly by referring to the model, which is not tied to a specific programming language, its constructs, or target hardware characteristics.

As shown later in Section 5, the formal model can also be translated into an executable form, allowing for side-by-side comparison of the operation of the reference definition and the actual implementation.

3.2. Model Semantics

Presenting the application’s operation in a formal manner can be achieved through semantics. Semantics is used to present the meaning of a program formally, independent of any specific programming language. The meaning of the program is defined by the expected result, or the outcome of executing the program.

Three types of semantics can be distinguished:

- Axiomatic semantics—defines the meaning of the program by specifying axioms and a set of rules, which is used to prove the program’s properties.
- Operational semantics—precisely defines the meaning of the program by specifying the successive computational steps during the program’s execution.
- Denotational semantics—defines the meaning of the program using abstract mathematical constructs [22,23], not describing the details of the constructs (i.e., bit representation of numbers, carry propagation during calculations, or memory access method).

Applying a semantic approach, which allows for a reliable presentation of the program’s operation formally, has many advantages. These include the following:

- Certainty of an accurate and understandable description of the program’s operation;
- Independence from a specific implementation;
- Ease of implementing the program in any programming language;
- Ability to reliably infer the correctness of the program’s operation.

In this work, the formal description of the CPDev virtual machine is constructed using denotational semantics in the form of denotational equations, which depict in an understandable manner the successive steps taken during the execution of individual instructions to achieve the desired result. Based on the formal model of the VM operation, it is possible to prepare the implementation that operates in the same way as defined in the specification in any programming language and for various target platforms.

3.3. Semantic Domains

The semantic domains represent abstract data types, modeling values used by the virtual machine. These domains arise from the architecture of the virtual machine and are expressed using formulas. The following domains related to memory are defined in (1):

- **Address**: depends on the implementation, but here, 4 bytes are assumed (32-bit).
- **Memory**: a function that, for a given address, returns a byte value.
- **CodeMemory**: represents the Memory (alias) with the program code.
- **DataMemory**: represents the Memory with variable values.

\[
\begin{align*}
\text{Address} &= \text{FourBytes} \\
\text{Memory} &= \text{Address} \rightarrow \text{OneByte} \\
\text{CodeMemory} &= \text{Memory} \\
\text{DataMemory} &= \text{Memory}
\end{align*}
\] (1)

The stacks of the virtual machine are defined in Formula (2). They are as follows:

- **Stack**: represents the stack using a Kleene closure (i.e., a sequence indicated by \( ^* \)). It is handled using helper functions Push and Pop, which will be discussed in Section 3.4.
- **CodeStack**: represents the Stack of return addresses (subroutines).
- **DataStack**: represents the Stack of database addresses.

\[
\begin{align*}
\text{Stack} &= \text{Address}^* \\
\text{CodeStack} &= \text{Stack} \\
\text{DataStack} &= \text{Stack}
\end{align*}
\] (2)

The following registers are defined in the model (3):

- **CodeReg**: the Address of the current instruction in the CodeMemory.
- **DataReg**: the base Address for operations in the DataMemory.
- **Flags**: each bit of this 16-bit value signals errors or unusual situations (array index exceeded, cold start, time overrun, etc.).

\[
\begin{align*}
\text{CodeReg} &= \text{Address} \\
\text{DataReg} &= \text{Address} \\
\text{Flags} &= \text{TwoBytes}
\end{align*}
\] (3)

The exception-handling mechanism involves nested protected code portions with one or more catch blocks. The stack \( \text{ProtStack} \) used for this purpose is represented as a Kleene closure of \( \text{ProtEntry} \) tuples, each consisting of four addresses. \( \text{ExcObj} \) is an Address indicating where the exception object has been stored. Consequently, the exception state \( \text{ExcState} \) is represented by the \( \text{ProtStack} \) and \( \text{ExcObj} \):

\[
\begin{align*}
\text{ProtEntry} &= \text{Address} \times \text{Address} \times \text{Address} \times \text{Address} \\
\text{ProtStack} &= \text{ProtEntry}^* \\
\text{ExcObj} &= \text{Address} \\
\text{ExcState} &= \text{ProtStack} \times \text{ExcObj}
\end{align*}
\] (4)

In a broad sense, the objective of executing a program is to transform the current state of the machine into a new one. The state is depicted as a combination of various aspects such as memory, stacks, registers, and flags. With the specific domains outlined in the model, the state of the virtual machine is encapsulated within the \( \text{State} \) domain as defined by Formula (5):

\[
\text{State} = \text{CodeMemory} \times \text{DataMemory} \times \text{CodeStack} \times \text{DataStack} \times \text{CodeReg} \times \text{DataReg} \times \text{ExcState} \times \text{Flags}
\] (5)

As seen, the \( \text{State} \) domain is formed through the amalgamation of code memory, data memory, code stack, data stack, code register, data register, and processor flags. These components collectively influence the outcome during the execution of each instruction, stemming from the underlying architecture of the machine and its instruction processing methodology. They can be succinctly represented by a set of tuples:

\[
(cm, dm, cs, ds, cr, dr, ex, flg)
\] (6)
where each tuple element corresponds to a value within its respective domain.

### 3.4. Memory and Stack Operations

One of the helper functions handling data and code memories is the memory read function. The task is to retrieve data from the given address and sometimes from subsequent addresses, forming a value of the specified size. The models of these functions are presented using Formula (7). The function `Get1BMem` retrieves a single byte from the specified address. The function `Get2BMem` retrieves two data bytes, one from the given address and the other from an address incremented by one, and then combines them into a `TwoBytes` value. The functions `Get4BMem` and `Get8BMem` retrieve 4 and 8 bytes, respectively, combining them into `FourBytes` and `EightBytes` values. The function `GetAddress` retrieves the value stored at the specified `Address` in `Memory`, where `Memory` itself is represented by another `Address`. Since the virtual machine lacks an accumulator and directly operates on addresses, the function `GetAddress` plays a crucial role in the model:

\[
\begin{align*}
\text{Get1BMem} &= (\text{Address} \times \text{Memory}) \rightarrow \text{Byte} \\
\text{Get2BMem} &= (\text{Address} \times \text{Memory}) \rightarrow \text{TwoBytes} \\
\text{Get4BMem} &= (\text{Address} \times \text{Memory}) \rightarrow \text{FourBytes} \\
\text{Get8BMem} &= (\text{Address} \times \text{Memory}) \rightarrow \text{EightBytes} \\
\text{GetAddress} &= (\text{Address} \times \text{Memory}) \rightarrow \text{Address}
\end{align*}
\]

The task of the memory update functions is to change the contents of memory at a given address with the provided value. As a result, a new memory is returned, which is a copy of the old one with the introduced change. These functions come in several variants, similar to the read functions. They respectively modify one, two, four, or eight bytes in memory. The variants of the update functions have been defined using Formula (8). Additionally, the function `MemMove` moves a portion of memory from the source `Address` to the target `Address`. The size of the portion is specified as `OneByte` (0 to 255):

\[
\begin{align*}
\text{Upd1BMem} &= (\text{Address} \times \text{Memory} \times \text{OneByte}) \rightarrow \text{Memory} \\
\text{Upd2BMem} &= (\text{Address} \times \text{Memory} \times \text{TwoBytes}) \rightarrow \text{Memory} \\
\text{Upd4BMem} &= (\text{Address} \times \text{Memory} \times \text{FourBytes}) \rightarrow \text{Memory} \\
\text{Upd8BMem} &= (\text{Address} \times \text{Memory} \times \text{EightBytes}) \rightarrow \text{Memory} \\
\text{MemMove} &= (\text{Address} \times \text{Memory} \times \text{Address} \times \text{Memory} \times \text{OneByte}) \rightarrow \text{Memory}
\end{align*}
\]

The helper functions `Push` and `Pop`, whose models are presented using Formula (9), perform operations on the code stack and data stack. The `Push` function pushes the given address onto the stack, returning a new stack, while the `Pop` function returns the last pushed address on the stack along with the new stack without the retrieved value. They are used during the invocation of user functions, functional blocks, and programs:

\[
\begin{align*}
\text{Push} &= (\text{Stack} \times \text{Address}) \rightarrow \text{Stack} \\
\text{Pop} &= \text{Stack} \rightarrow (\text{Address} \times \text{Stack})
\end{align*}
\]

To manage the `ProtStack` for exception handling, the following functions are introduced: `PushProt`, `PopProt`, and `PeekProt`. `PushProt` adds an item to the stack, `PopProt` removes an item from the stack, and `PeekProt` returns a copy of the topmost item without altering the stack:

\[
\begin{align*}
\text{PushProt} &= (\text{ProtStack} \times \text{ProtEntry}) \rightarrow \text{ProtStack} \\
\text{PopProt} &= \text{ProtStack} \rightarrow (\text{ProtEntry} \times \text{ProtStack}) \\
\text{PeekProt} &= \text{ProtStack} \rightarrow \text{ProtEntry}
\end{align*}
\]
The last group of helper functions comprises the processor flag control functions. The SetFlag function sets the status bits, while the ClearFlag clears the status bits. The flags are used for signaling cold start, invalid instruction, or array out-of-bounds conditions. The models of these functions are presented using Formula (11). The successive TwoBytes denote the actual value of Flags, bits to be set or reset, and new Flags:

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

(11)

3.5. Data Value Interpretation

Since the memory read functions return values in the form of memory bytes, it was necessary to introduce functions that assign numerical values to IEC 61131-3 data types, such as BOOL, BYTE, INT, LINT, REAL, and others. Additionally, functions were introduced to convert the numerical value of a given type to the byte format, as memory update functions accept a value represented in the form of bytes as a parameter. The models of functions performing the interpretation of values based on bytes and vice versa for some basic types are presented using Formula (12). Other types are interpreted analogously:

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

(12)

Arithmetic operations such as the addition, subtraction, and multiplication of operands of the same type may cause overflow of the range of that type. To address this issue, limited arithmetic operators are introduced: limited addition (⊕), limited subtraction (⊖), and limited multiplication (⊗). Consider, for instance, the signed addition ⊕ of integers. For signed integers, the operator ⊕ is defined by

\[
a \oplus b = \begin{cases} 
\text{if } (a + b) > \text{MaxRange}(a) \text{ or } (a + b) < \text{MinRange}(a) & \text{then } -(\text{MinRange}(a)) + (a + b) \mod (\text{MinRange}(a)) \\
\text{else } a + b & 
\end{cases}
\]

(13)

where the MinRange for INT, DINT, and LINT means \(-32768\), \(-2^{31}\), and \(-2^{63}\), respectively, and MaxRange for INT is 32767, for DINT, \(2^{31} - 1\), and for LINT, \(2^{63} - 1\) [24]. Similar definitions may be given for other operators and types.

Value conversions are utilized to transfer a value from one type to another (if feasible). Two instances of such conversions are depicted in Equation (14). For an unsigned value, ByteToWord inserts zero bits into the more significant byte of TwoBytes, while for signed
values, the byte is filled with the sign bit. Conversely, \(\text{WordToByte}\) decreases the value by eliminating the most significant bits.

\[
\text{ByteToWord} = \text{OneByte} \rightarrow \text{TwoBytes} \\
\text{WordToByte} = \text{TwoBytes} \rightarrow \text{OneByte}
\] (14)

### 3.6. Instruction Decoding

As detailed in Section 2, the identifier \(\text{vmcode}\) of a machine instruction comprises the identifier of the group \(\text{ig}\) and the specific type of the instruction \(\text{it}\). The processing of each instruction in the virtual machine can be divided into two parts: a generic function \(\mathcal{U}\), which involves fetching the instruction group and its variant, and a dependent function \(\mathcal{C}\), which is specific to each instruction executed by the machine. Processing both the generic and dependent parts results in returning a new state. So, the generic function decoding any instruction can be defined as follows:

\[
\mathcal{U}[\text{any_instruction}] = \text{State} \rightarrow \text{State}
\] (15)

After an instruction is decoded by \(\mathcal{U}\), the dependent function \(\mathcal{C}\) is called:

\[
\mathcal{C}[\text{instruction}] = \text{State} \rightarrow \text{State}
\] (16)

The semantic model of the function \(\mathcal{U}\) is expressed by Formula (17):

\[
\mathcal{U}[\text{any_instruction}] = \lambda s. \\
\quad (cm, dm, cs, ds, cr, dr, ex, f|g) := s \\
\quad \text{ig} := \text{Get1BMem}(cr, cm) \\
\quad cr_1 := cr \oplus 1 \\
\quad \text{it} := \text{Get1BMem}(cr_1, cm) \\
\quad cr_2 := cr_1 \oplus 1 \\
\quad s_1 := \text{match ig with} \\
\quad \hspace{1em} | 01 \rightarrow \text{match it with} \\
\quad \hspace{2em} | 22 \rightarrow \mathcal{C}[\text{ADD:INT:r:op1:op2}](cm, dm, cs, ds, cr_2, dr, ex, f|g) \\
\quad \hspace{2em} | 32 \rightarrow \mathcal{C}[\text{ADD:INT:r:op1:op2:op3}](cm, dm, cs, dr, ex, f|g) \\
\quad \hspace{1em} | \ldots \\
\quad \hspace{1em} | 04 \rightarrow \text{match it with} \\
\quad \hspace{2em} | 02 \rightarrow \mathcal{C}[\text{DIV:INT:r:op1:op2}](cm, dm, cs, ds, cr_2, dr, ex, f|g) \\
\quad \hspace{2em} | 09 \rightarrow \mathcal{C}[\text{DIV:REAL:r:op1:op2}](cm, dm, cs, cr_2, dr, ex, f|g) \\
\quad \hspace{1em} | \ldots \\
\quad \hspace{1em} | \ldots \\
\quad \hspace{1em} \text{end}
\] (17)

According to [25] or [26], \(\lambda\)-expression has the form of \(\lambda s.\text{body}\), where \(s\) denotes the current state, and \text{body} determines the value returned by the function. The computation of function \(\mathcal{U}\) begins with fetching the instruction group byte \(\text{ig}\), which is located in the code memory \(cm\) at the address indicated by the current value of the code register \(cr\). Retrieving this byte causes the instruction counter \(cr\) to be incremented to the value \(cr_1\). After retrieving the instruction type \(\text{it}\), the instruction counter is further incremented to the value \(cr_2\). Subsequently, matching to the appropriate group and variant of the function occurs. This involves processing the corresponding function \(\mathcal{C}\), invoked for the new state described by the tuple \((cm, dm, cs, ds, cr_2, dr, f|g)\).

Two examples of instruction groups are given in Formula (17). The \(\text{ig}=01\) denotes addition instructions, while \(\text{ig}=04\) denotes the division ones. In the first case, \text{it}=22 means the addition of two INT operands performed by \(\mathcal{C}[\text{ADD:INT:r:op1:op2}]\). For \text{it}=32, three operands are added by the function \(\mathcal{C}[\text{ADD:INT:r:op1:op2:op3}]\). Similarly, \(\text{ig}=04\) and...
it=02 call the DIV function for two INT operands, while it=09 means the division of two REALs. The result of the function C is a new state of the machine stored in the variable s1.

4. Denotations of Instructions

There are two kinds of virtual machine instructions: functions, and system procedures. Examples of some functions are shown in Table 2. The functions return one value each to be written into the variable that is the first operand (as said, an accumulator does not exist in this VM) and there may be up to 15 other operands. Note that such an order is different than in the Static Single Assignment of dataflow graphs used in typical compilers [27].

Table 2. Functions of the virtual machine

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>Negation</td>
<td>- (unary)</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
<td>&amp;</td>
</tr>
<tr>
<td>GT, GE</td>
<td>Greater, Greater or equal</td>
<td>&gt;, &gt;=</td>
</tr>
<tr>
<td>LT, LE</td>
<td>Less, Less or equal</td>
<td>&lt;, &lt;=</td>
</tr>
<tr>
<td>EQ, NE</td>
<td>Equal, Not equal</td>
<td>=, &lt;&gt;</td>
</tr>
<tr>
<td>MUL, DIV</td>
<td>Multiplication, Division</td>
<td>*, /</td>
</tr>
<tr>
<td>ADD, SUB</td>
<td>Addition, Subtraction</td>
<td>+, - (arithmetic)</td>
</tr>
<tr>
<td>EXPT</td>
<td>Power</td>
<td>**</td>
</tr>
</tbody>
</table>

Contrary to functions, system procedures do not return values directly. However, they may modify operand values in the data memory. Table 3 shows typical examples. The procedures control program flow, handle memory, call subprograms, etc.

Table 3. Procedures of the virtual machine.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMP, JNZ</td>
<td>Jump (unconditional, conditional)</td>
</tr>
<tr>
<td>JR, JRN</td>
<td>Relative jump (unconditional, conditional)</td>
</tr>
<tr>
<td>CALB, RETURN</td>
<td>Subroutine call, return</td>
</tr>
<tr>
<td>MCD</td>
<td>Initialize data memory</td>
</tr>
<tr>
<td>MEMCP</td>
<td>Copy data memory block</td>
</tr>
<tr>
<td>GARD, GAWR</td>
<td>Copy global to local memory or vice versa</td>
</tr>
<tr>
<td>CEAC</td>
<td>Check array index</td>
</tr>
</tbody>
</table>

Denotational equations modeling the execution of machine instructions have the common form $C[...] = \lambda s.\text{body}$, where the dots on the left side are replaced by a descriptor of a particular instruction, i.e., name and operands. The body is a sequence of operations that transform the actual state $s$ into a new one $s_1$ according to the instruction operation, values of the operands, and states of VM components. The following subsections present examples of semantic models for the selected functions and procedures.

4.1. Adding and Division Functions

The ADD instruction with two INT operands referenced in Section 3.6 can be represented with the denotational Equation (18):
C[ADD:INT:r:op1:op2] = \lambda s.
(cm, dm, cs, ds, cr, dr, ex, 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)) \oplus IntOf(Get2BMem(op2addr, dm))
s_1 := (cm, Upd2BMem(raddr, dm, FromInt(sv)), cs, ds, cr_3, dr, flg)

Splitting the current state \(s\) into components \((cm, dm, cs, ds, cr, dr, flg)\) through unification is the first operation in the equation.

A call to a particular function \(C\) by universal \(U\) is performed with the code register \(cr\) pointing to the first operand \((cr_2\) in (17)). The first operand of the ADD instruction refers to the addition result. Its address is acquired from the code memory \(cm\) and stored in \(r\) by:

\[r := GetAddress(cr, cm)\] (19)

However, since the operand may be a local variable of a subprogram, the value \(r\) means a relative address counting from the current value of database register \(dr\), set during the subprogram call. Therefore, the absolute address of the local variable in data memory is obtained by:

\[raddr := dr \oplus r\] (20)

In the case of a global variable, the value \(dr\) is zero, so \(raddr\) equals \(r\). Since the ADD instruction has other operands, the code register \(cr\) is later incremented to point out to the next memory location:

\[cr_1 := cr \oplus AddressSize\] (21)

The next two operands \(op1\) and \(op2\) are processed similarly.

After all the operands are available, addition may take place. The variable values to be added are read out from data memory \(dm\) and interpreted as INTs (Sections 3.4 and 3.5), using the expression:

\[\text{IntOf(Get2BMem(operandaddr, dm))}\] (22)

The result of the addition \(\oplus\) is stored in \(sv\). Then, the new state \(s_1\) is constructed as the tuple:

\[s_1 := (cm, Upd2BMem(raddr, dm, FromInt(sv)), cs, ds, cr_3, dr, flg)\] (23)
It contains a new value of data memory updated at the address \( raddr \) with two bytes of the addition result \( sv \). Finally, the new state \( s_1 \) is returned by the functions \( C \) and \( U \).

The semantic model of the \( DIV \) function for two \( INT \)-type operands is shown in (24).

The upper part of the equation is similar to the instruction \( ADD \) explained above. The divisor value is obtained from the operand \( op_2 \) with:

\[
divisor := \text{IntOf}(\text{Get2BMem}(op_2addr, dm))
\]  

Then, it is checked against zero to validate the operation. If so, a new expression information \( ex \) is constructed using the current value of the code register \( cr_3 \). A call to \( RAISE \) instruction will signal the division by zero and take an appropriate action. If \( divisor \) is correct (i.e., non-zero), the dividend value is retrieved and the division is made, with \( sv \) denoting the result. The updated data memory is the second element of \( s_1 \) as the result of invoking \( \text{Upd2BMem} \), and the two bytes stored at \( raddr \) are given by \( \text{FromInt}(sv) \).

4.2. Subroutines

Equation (26) illustrates the semantics of the \( CALB \) procedure, utilized for invoking a user subroutine, function, or functional block. Initially, the address of the instance \( inst \) for which the subroutine is called is decoded, followed by determining the subroutine’s address \( clbl \). Return addresses from the subroutine must be stored on the stacks \( cs \) and \( ds \); therefore, the new state returned by the procedure includes updated stacks after pushing the addresses \( cr_2 \) and \( dr \), with the code register set to the first instruction of the subroutine and the database register set to the instance address:
\[ C[\text{CALB:inst:clbl}] = \lambda s. (cm, dm, cs, ds, cr, dr, ex, flg) := s \]
\[ \text{inst} := \text{GetAddress}(cr, cm) \]
\[ \text{instaddr} := dr \oplus \text{inst} \]
\[ cr_1 := cr \oplus \text{AddressSize} \]
\[ clbl := \text{GetAddress}(cr_1, cm) \]
\[ cr_2 := cr_1 \oplus \text{AddressSize} \]
\[ s_1 := (cm, dm, \text{Push}(cs, cr_2), \text{Push}(ds, dr), clbl, \text{instaddr}, flg) \]
\[ s_1 \]

In Equation (27), the denotation of the RETURN procedure is provided. Its task is to return from a user subroutine by retrieving the values of the code and data registers stored on the cs and ds stacks, respectively, and setting them as the current values of the code and data registers. This procedure retrieves the value of the code register clbl, stored earlier on the code stack by the CALB procedure, and the updated code stack cstk after using the function Pop. Similarly, it retrieves the value of the data register dr and the new data stack dstk. As a result, the procedure returns a new state, which is updated by utilizing the retrieved values representing the new code and stacks, code register, and data register:

\[ C[\text{RETURN}] = \lambda s. (cm, dm, cs, ds, cr, dr, ex, flg) := s \]
\[ (clbl_1, cstk) := \text{Pop}(cs) \]
\[ (dr_1, dstk) := \text{Pop}(ds) \]
\[ s_1 := (cm, dm, cstk, dstk, clbl_1, dr_1, flg) \]
\[ s_1 \]

\[ C[\text{GAWR:dst:src:size:idx}] = \lambda s. (cm, dm, cs, ds, cr, dr, ex, flg) := s \]
\[ dst := \text{GetAddress}(cr, cm) \]
\[ cr_1 := cr \oplus \text{AddressSize} \]
\[ src := \text{GetAddress}(cr_1, cm) \]
\[ srcaddr := dr \oplus src \]
\[ cr_2 := cr_1 \oplus \text{AddressSize} \]
\[ size := \text{GetAddress}(cr_2, cm) \]
\[ sizeaddr := dr \oplus size \]
\[ sizeval := \text{WordOf} (\text{Get2BMem}(sizeaddr, dm)) \]
\[ cr_3 := cr_2 \oplus \text{AddressSize} \]
\[ idx := \text{GetAddress}(cr_3, cm) \]
\[ idxaddr := dr \oplus idx \]
\[ idxval := \text{WordOf} (\text{Get2BMem}(idxaddr, dm)) \]
\[ cr_4 := cr_3 \oplus \text{AddressSize} \]
\[ resultaddr := dst \oplus idxval \oplus sizeval \]
\[ um := \text{MemMove}(dm, resultaddr, dm, srcaddr, sizeval) \]
\[ s_1 := (cm, um, cs, ds, cr_4, dr, flg) \]
\[ s_1 \]
\[ C[CEAC:idx:lobase:hilevel] = \lambda s. \]
\[ (cm, dm, cs, ds, cr, dr, ex, flg) := s \]
\[ idx := \text{GetAddress}(cr, cm) \]
\[ idxaddr := dr \oplus idx \]
\[ idxval := \text{IntOf}(\text{Get2BMem}(idxaddr, dm)) \]
\[ cr_1 := cr \oplus \text{AddressSize} \]
\[ lobase := \text{GetAddress}(cr_1, cm) \]
\[ lobaseaddr := dr \oplus lobase \]
\[ lobaseval := \text{IntOf}(\text{Get2BMem}(lobaseaddr, dm)) \]
\[ cr_2 := cr \oplus \text{AddressSize} \]
\[ hilevel := \text{GetAddress}(cr_2, cm) \]
\[ hileveladdr := dr \oplus hi \]
\[ hilevelval := \text{IntOf}(\text{Get2BMem}(hileveladdr, dm)) \]
\[ lozero := idxval \ominus lobaseval \]
\[ um := \text{Upd2BMem}(idxaddr, dm, \text{FromInt}(lozero)) \]
\[ s_1 := \text{match lozero} \geq 0 \& lozero \leq hilevelval \text{ with} \]
\[ | \text{true} \rightarrow (cm, um, cs, ds, cr_3, dr, ex, flg) \]
\[ | \text{false} \rightarrow (ps, eo) := ex \]
\[ eo_1 := (cr_3, \text{OUT_OF_BOUNDS_EXC}) \]
\[ ex_1 := (ps, eo_1) \]
\[ nflag := \text{SetFlag}(flg, \text{OUT_OF_BOUNDS_EXC}) \]
\[ C[\text{RAISE:op}]((cm, dm, cs, ds, cr, dr, ex_1, nflag)) \]
\[ end \]
\[ s_1 \]

4.3. Array Operations

The procedure \texttt{GAWR} presented in Equation (28) copies elements of an array from local memory to an array in global memory. Both arrays may contain elements of any type, including nested arrays and structures. The procedure involves four operands: the source label (src) and the destination label (dst), the size of each element, and the array index (idx). The values of size and idx are addresses pointing to data of the \texttt{WORD} type. Since the operand dst refers to global memory, its value is an absolute address (zero dr). The resultaddr is calculated as the sum of the address dst and the product of idxval and sizeval.

It is crucial to acknowledge that the equations outlined for the \texttt{GAWR} procedure do not consider erroneous operands, such as an array index being out of bounds. To mitigate these potential failures and ensure predictable behavior, the instruction CEAC is invoked before \texttt{GAWR} to validate the array indexes as shown in the Equation (29). If an index is found to be out of bounds, an exception is raised accordingly.

5. Executable Form of the Model

Utilizing the semantic models outlined in Sections 3 and 4, an executable code mirroring the behavior specified for the virtual machine is developed using the F\# language. F\# is primarily a functional programming language that operates on the .NET platform.

5.1. Domains

Firstly, it is necessary to define data types according to the semantic domains from Section 3. The \texttt{Address} type in Listing 1 is defined as an alias for the uint32 data type of the .NET platform. Memory storage is implemented as an array of bytes. The code and data stacks are represented by a list collection storing data of the type \texttt{Address}. The machine state is defined by the \texttt{State} type, which is a tuple comprising code memory, data memory,
code stack, data stack, code register, data register, and flags. It also involves the exception state, consisting of the protected stack and the exception object. The code register and data register are variables of type Address, while the Flags type is an alias for uint16.

A set of IEC 61131-3 data type aliases has also been defined as the discriminated union ValueType.

Listing 1. Definitions of the model domains.

```plaintext
type Address = uint32

type Storage = byte array

type Memory = (Address * Storage) -> byte

type Stack = Address list

type Flags = uint16

type ProtEntry = (Address * Address * Address * Address)

type ProtStack = ProtEntry list

type ExcObj = Address

type ExcState = ProtStack * ExcObj

type State = (Storage * Storage * Stack * Stack * Address * Address * ExcState * Flags)

type ValueType =
    | BOOL of bool
    | BYTE of byte
    | WORD of uint16
    | DWORD of uint32
    | LWORD of uint64
    | SINT of sbyte
    | INT of int16
    | DINT of int32
    | LINT of int64
    | REAL of float
    | LREAL of float
    | TIME of uint32
```

5.2. Memory and Stack Operations

Listing 2 illustrates the implementation of `Get1BMem` and `Get8BMem` functions, which read 1 and 8 bytes of data from memory, respectively. Similar functions for reading 2 and 4 bytes are also implemented. They take an address and memory as parameters to specify the data retrieval. `Get1BMem` returns a byte value obtained from the specified memory address, while `Get8BMem` retrieves 8 bytes, converts them to a uint64 value, and returns it.

The lower part of Listing 2 showcases `Upd1BMem` and `Upd4BMem` functions, responsible for writing 1 and 4 bytes of data to memory, respectively. `Upd1BMem` writes the byte value to the memory at the specified address and returns the updated memory. `Upd4BMem` converts the uint32 value to an array of bytes, then inserts each byte value into the memory starting from the specified address.

Copying data from the source memory to the destination memory is achieved through the `MemMove` function as depicted in Listing 3. The first two parameters determine the starting index for copying data to the destination memory, while the subsequent two parameters denote the index and source memory, respectively. The final parameter specifies the number of bytes to be copied. Initially, a new memory is instantiated, serving as a duplicate of the destination memory. Subsequently, the function retrieves the specified number of bytes from the source memory. Then, it inserts the copied bytes from the source
memory into the newly created memory, commencing from the designated destination address, address1. Ultimately, the updated memory is returned.

**Listing 2.** Memory read and update functions.

```fsharp
let Get1BMem (address:Address, mem:Storage) : byte =    mem.[int <| address]

let Get8BMem (address:Address, mem:Storage) : uint64 =    let byteArray = [|
    let mutable iterator = address
    while (iterator < address + 8us) do
        yield mem.[int <| iterator]
        iterator <- iterator + 1us
    |]
    BitConverter.ToUInt64(byteArray,0)

let Upd1BMem (address:Address, mem:Storage, value:byte) : Storage =    let new_mem = Array.copy mem
    new_mem.[int <| address] <- value
    new_mem

let Upd8BMem (address:Address, mem:Storage, value:uint64) : Storage =    let new_mem = Array.copy mem
    let byteArray = BitConverter.GetBytes(value)
    let mutable iterator = address
    for b in byteArray do
        new_mem.[int <| iterator] <- b
        iterator <- iterator + 1us
    new_mem
```

**Listing 3.** Memory read and update functions.

```fsharp
let MemMove (address1:Address, mem1:Storage, address2:Address, mem2:Storage, count:uint16):Storage =    let new_mem = Array.copy mem1
    let copiedByteArray = [|
        let mutable iterator = address2
        while (iterator < address2 + count) do
            yield mem2.[int <| iterator]
            iterator <- iterator + 1us
    |]
    let mutable iterator2 = address1
    for b in copiedByteArray do
        new_mem.[int <| iterator2] <- b
        iterator2 <- iterator2 + 1us
    new_mem
```

On Listing 4, the implementations of stack-related functions are presented. The Push function takes as parameters the stack on which a value of type Address is to be pushed. If the stack is empty, the value is pushed onto the stack, and a new stack with the pushed value is returned. If the stack represented by the list already contains values, the list elements are reversed, the value to be pushed is added to the beginning of the list, and then the list elements are reversed again. The Pop function takes the stack as a parameter, and its task is to return the last pushed value on the stack along with the new stack after removing the value. If the stack is empty, the function returns None along with an empty stack and an appropriate message for the user. Otherwise, the last value from the list, which is the last
pushed address on the stack, is retrieved, and this value is returned along with the stack after removing this value.

**Listing 4.** Stack functions in F♯.

```fsharp
let Push (stack:Stack, address:Address) : Stack =
    match stack with
    | [] -> [address]
    | _ -> stack @ [address]

let Pop (stack:Stack) =
    match stack with
    | [] -> printfn "Empty stack"; (None, [])
    | _ -> let new_stack = List.take <| List.length stack - 1 <| stack
          (Some <| List.last stack, new_stack)
```

Due to the fact that memory read-and-write functions operate on unsigned integer values, while the core functions work on operands of various simple types such as BOOL, BYTE, WORD, INT, DWORD, DINT, LINT, REAL, and LREAL, it is necessary to implement functions that interpret unsigned integer values into appropriate simple types and vice versa.

The `IntOf` from the Listing 5 function takes a `uint16` value as a parameter, retrieves its representation as a byte array, and then interprets it into an `int16` value using the `BitConverter` class, which is returned as one of the discriminated union `ValueType` types. The `FromInt` function, takes a `ValueType` discriminated union type as a parameter (in this case, the `INT` type). Initially, it matches the appropriate union type to extract the numerical value of the parameter, which is then converted into an unsigned integer value (in this case, `uint16`), and returned as the result of the function. If an inappropriate type is passed to the function, an exception with information for the user is raised.

**Listing 5.** Value converters in F♯.

```fsharp
let IntOf (a:uint16) =
    let byteArray = BitConverter.GetBytes(a)
    let value = BitConverter.ToInt16(byteArray,0)
    INT value

let FromInt (a:ValueType) : uint16 =
    match a with
    | INT a -> let byteArray = BitConverter.GetBytes(a)
              let value = BitConverter.ToUInt16(byteArray,0)
              value
    | _ -> failwith "Wrong data type"
```

5.3. Machine Instructions

The semantic equations presented in Section 4 define the steps taken during the processing of individual instructions executed by the virtual machine but only for one particular data type. Most functions have multiple variants, largely due to the types of operands they accept. Some functions also have variants with different numbers of operands. The semantic models for different variants of functions differ only in the use of functions for memory read, memory write, and value interpretation. To avoid duplicating code and writing several similar functions performing the same operation but with slight modifications, generic functions are implemented to handle all variants of a given instruction. The discriminated union type `ValueType` is used (Listing 1) to represent the basic types.

The ADD instruction, which performs the addition operation, has multiple variants for different numbers of operands (2 to 15) and different data types. The generic imple-
mentation of \(\text{ADD}\) is presented in Listing 6. The function accepts a parameter of type \(\text{State}\), containing the current values of individual components describing the machine state. It also receives auxiliary functions as parameters, which handle data retrieval from memory (\(\text{getBytes}\)), their interpretation into the appropriate type (\(\text{typeOf}\)), type conversion into the appropriate byte representation (\(\text{fromType}\)), and data storage in memory (\(\text{updBytes}\)). Since \(\text{ADD}\) operates on different numbers of operands, an additional parameter \(\text{count}\) specifies how many operands are to be added. Initially, the address for storing the result is retrieved. Then, the operation retrieves a specified number of operands from the data memory based on the determined addresses and interprets them into the appropriate type using the interpretation function passed as a parameter. Finally, using the \(\text{reduce}\) function from the \(\text{List}\) module, each operand is matched to the appropriate \(\text{ValueType}\) union type to retrieve the value of each operand, after which the addition operation is performed on all operands. The resulting value is then stored in memory at the determined address \(\text{raddr}\) using the conversion function passed as a parameter. The function returns a new state containing the updated data memory after storing the result along with the current value of the code register \(\text{crx}\).

**Listing 6.** Generic \(\text{ADD}\) in \(\mathbb{F}_2\).

```fsharp
let ADD(state:State, getBytes:(Address*Storage)->'a , updBytes:(Address*Storage*')a->Storage, typeOf:'a -> ValueType, fromType:ValueType->'a, count:int) =
  let (cm,dm,cs,ds,cr,dr,ex,flg) = state
  let r = GetAddress(cr,cm)
  let raddr = dr + r
  let mutable crx = cr + AddressSize
  let mutable lst = []
  for i=1 to count do
    let opx = GetAddress(crx,cm)
    let opxaddr = dr + opx
    let opxvalue = typeOf(getBytes(opxaddr,dm))
    lst <- opxvalue :: lst
    crx <- crx + AddressSize
    printf $"Op%A{i}: %A{opxvalue} \n"
  let value = List.reduce (fun acc el ->
    match acc,el with
    | SINT a, SINT b -> SINT (a+b)
    | INT a, INT b -> INT (a+b)
    | DINT a, DINT b -> DINT (a+b)
    | LINT a, LINT b -> LINT (a+b)
    | BYTE a, BYTE b -> BYTE (a+b)
    | WORD a, WORD b -> WORD (a+b)
    | DWORD a, DWORD b -> DWORD (a+b)
    | LWORD a, LWORD b -> LWORD (a+b)
    | REAL a, REAL b -> REAL (a+b)
    | LREAL a, LREAL b -> LREAL (a+b)
    | _ -> failwith "Wrong data"
  ) lst
  printfn $"\nADD result: %A{value}"
  (cm, updBytes(raddr, dm, fromType(value)),cs,ds,crx,dr,ex,flg)
```
The DIV instruction is defined similarly as shown in Listing 7 with parameter decoding dropped for brevity. However, it includes an additional block to check for a bad divisor (i.e., when it equals zero). If such a condition is met, an exception object is created and the exception is raised.

**Listing 7.** Generic DIV in F².

```plaintext
let DIV(state:State, getBytes:(Address*Storage)->'a , updBytes:(Address*Storage*')a)->Storage, typeOf: 'a -> ValueType, fromType: ValueType -> 'a) =
  let (cm,dm,cs,ds,cr,dr,ex,flg) = state
  // getting values of raddr, dividend, divisor
match divisor with
| SINT 0y | INT 0s | DINT 0 | LINT 0L | BYTE 0uy | WORD 0us | DWORD 0u |
| LWORD 0UL | REAL 0.0f | LREAL 0.0 ->
  let (ps, eo) = ex
  let eo1 = (cr3, DIV_BY_ZERO_EXC)
  let ex1 = (ps, eo1)
  let dvflg = SetFlag(flg, DIV_BY_ZERO_FLG)
  RAISE(cm,dm,cs,ds,cr3,dr,ex1,dvflg)
| SINT _ | INT _ | DINT _ | LINT _ | BYTE _ | WORD _ | DWORD _ | LWORD _ |
| REAL _ |LREAL _ ->
  let value = match dividend, divisor with
  | SINT a, SINT b -> SINT (a/b)
  | INT a, INT b -> INT (a/b)
  | DINT a, DINT b -> DINT (a/b)
  | LINT a, LINT b -> LINT (a/b)
  | BYTE a, BYTE b -> BYTE (a/b)
  | WORD a, WORD b -> WORD (a/b)
  | DWORD a, DWORD b -> DWORD (a/b)
  | LWORD a, LWORD b -> LWORD (a/b)
  | REAL a, REAL b -> REAL (a/b)
  |LREAL a, LREAL b -> LREAL (a/b)
  |_ -> failwith "Wrong data"
  printfn "Operand1: %A Operand2: %A\nDIV result: %A"
  dividend divisor value
  let modBytes = updBytes(raddr, dm, fromType(value))
  (cm, modBytes, cs, ds, cr3, dr, ex, flg)
  |_ -> failwith "Wrong data"
```

The CEAC instruction from Listing 8, which verifies the correctness of an index value in an array, also utilizes exceptions. If the provided idxval falls between libaseval and hilevelval, the flag register remains unchanged. However, if the index is out of bounds, the exception object signals OUT_OF_BOUNDS_EXC using the RAISE function. Additionally, the OUT_OF_BOUNDS_FLG flag is set.
Listing 8. Array index checking with CEAC in F$^\#$.

```fsharp
let CEAC(state:State) : State =
    let (cm, dm, cs, ds, cr, dr, ex, flg) = state
    ...
    // getting values of idxval, lobaseval, hilevelval
    let tmp = match idxval, lobaseval with
        | INT a , INT b -> INT(a-b)
        | _ -> failwith "Wrong data"
    let um = Upd2BMem(idxaddr, dm, FromInt(tmp))
    let newflag = match tmp, hilevelval with
        | INT a , INT b ->
            if (a >= 0s && a <= b)
            then flg
            else
                let (ps, eo) = ex
                let eo1 = (cr3, OUT_OF_BOUNDS_EXC)
                let ex1 = (ps, eo1)
                let badIdxFlg = SetFlag(flg,OUT_OF_BOUNDS_FLG)
                RAISE(cm, dm, cs, ds, cr3, dr, ex1, badIdxFlg)
        | _ -> failwith "Wrong data"
    (cm, um, cs, ds, cr3, dr, ex, newflag)
```

6. Checking the Implementation against the Model

6.1. Procedure

Implementing the CPDev virtual machine for diverse target platforms involves various programming techniques, depending on platform specifics, available languages, and tools. The CPDev virtual machine has been ported to ARM, x86, RISC-V, and various other microprocessors and microcontrollers, with an FPGA-based machine also developed. Additionally, several applications for control devices have been created [14]. Typically, the runtime for a device is developed by industrial users based on the provided VM specification. Previously, each implementation required a manual series of tests to ensure compliance with the specification. These tests encompassed several dozen programs utilizing the full set of runtime features. Each change in the runtime necessitated the repetition of these tests. However, with the executable model in F$^\#$, automated verification becomes feasible, streamlining the process and reducing the manual effort required.

To verify the accuracy of the machine implementation on a particular platform, a series of control projects utilizing the VM functions and procedures are employed. These projects are compiled into intermediate binary code, which is utilized by both the executable model and the implemented runtime. The process involves the following steps:

- Creating a control project using IEC 61131-3 languages like ST, FBD, LD, or SFC within the CPDev environment.
- Compiling the program into the intermediate binary code for the virtual machine (XCP file, see Section 2).
- Running the program on the target platform and by the F$^\#$ model side by side.
- Comparing the state of the virtual machine on the platform with the state of the model.

The comparison occurs after the predefined cycle time of control devices, typical for PLCs. At this point, data memory, stack contents, and registers are retrieved from the device using a communication protocol. These retrieved values are then compared with the model pattern, and any disparities prompt a message to the user to rectify the implementation.

6.2. Tools

The runtime checking methodology utilizing the F$^\#$ executable model seamlessly integrates with the CPDev IDE, designed around the .NET framework. By augmenting the programming environment with the reference runtime, developers can execute a control
program on the target device (controller) using the implementation in question alongside the model.

Before executing each program cycle, the controller may receive data of the external signals via its inputs, such as sensors. These acquired values are then transferred to the CPDev IDE using online communication capabilities such as Modbus or TCP, and provided to the model, effectively stored in the data memory. This ensures that both the model and the runtime operate on the same input data, leading to consistent results.

Figure 3 illustrates a test session in the CPDev environment, verifying the implementation of the ADD and DIV functions for REAL variables. The simple test program is prepared in the FBD language, utilizing four global variables I, J, K, and L of type REAL. Additionally, local variables OUT1 and OUT2, and a constant 10.0 are introduced. In the lower part, a portion of the virtual machine code is displayed in binary format (on the left) and textual format (assembler notation). The program is executed cycle by cycle on the target device. The verification of the virtual machine state against the model is performed after each cycle with the results reported at the bottom. As illustrated in Figure 3, the verification passed successfully for all program cycles. In case of any disparities between the outcomes of the model and the runtime, an alert would be raised to the user, providing specific details for error correction.

![Figure 3. Verifying the ADD and DIV implementation in the CPDev environment.](image)

The debugging capabilities of the CPDev environment facilitate checking results not only after the completion of all operations in the program cycle but also within the cycle itself. In this scenario, the device operation is paused after each instruction performed by the virtual machine, allowing for an immediate comparison between the state of the model
and the implemented runtime. This approach helps detect erroneous implementations of the last executed instruction promptly. For example, the presented methodology identified issues in one of the runtime implementations, such as a missing exception raise in the DIV instruction and an extra byte being modified by the CEAC instruction, apart from the intended array element.

7. Final Remarks

The denotational equations forming the semantic model of the runtime appear to be a favorable choice, despite being primarily used for programming languages. As demonstrated, these equations can be naturally translated into executable $F^2$ code. This format enables the examination of whether the virtual machine implementation will execute the portable control program code consistently across different target platforms, regardless of the implementation techniques. The presented methodology holds promise for industrial applications, providing an automated method for verifying the correctness of the runtime, which may be particularly interesting in the field of distributed control systems consisting of heterogeneous hardware platforms. Consequently, it facilitates the development of predictable, consistent, and error-free control devices.

Author Contributions: Conceptualization, J.S.; methodology, J.S. and B.T.; software, J.S. and B.T.; validation, J.S.; formal analysis, J.S.; investigation, B.T.; resources, J.S. and B.T.; data curation, J.S. and B.T.; writing—original draft preparation, B.T.; writing—review and editing, B.T. and J.S.; visualization, B.T.; supervision, B.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.


Conflicts of Interest: The authors declare no conflicts of interest.

References


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.