Proc. 2010 Third International Conference on Software Testing, Verification and Validation. IEEE, 2010.

Model Checking PLC Software Written in Function Block Diagram

Olivera Pavlović [1]
Braunschweig, Germany
Email: olivera.r.pavlovic@googlemail.com

Hans-Dieter Ehrich
Technische Universität Braunschweig
Braunschweig, Germany
Email: hd.ehrich@tu-bs.de

***Abstract*—The development of Programmable Logic Controllers (PLCs) in the last years has made it possible to apply them in ever more complex tasks. Many systems based on these controllers are safety-critical, the certification of which entails a great effort. Therefore, there is a big demand for tools for analyzing and verifying PLC applications. Among the PLC-specific languages proposed in the standard IEC 61131-3, FBD (Function Block Diagram) is a graphical one widely used in rail automation. In this paper, a process of verifying FBDs by the NuSMV model checker is described. It consists of three transformation steps: FBD→TextFBD→tFBD→NuSMV. the novel step introduced here is the second one: it reduces the state space dramatically so that realistic application components can be verified. The process has been developed and tested in the area of rail automation, in particular interlocking systems. As a part of the interlocking software, a typical point logic has been used as a test case.**

***Keywords*-formal verification; model checking; PLC; FBD; IEC 61131-3;**

I. Introduction

Programmable Logic Controllers (PLCs) are a special type of computer used in automation systems [1]. Generally speaking, they are based on sensors and actuators which have the ability to control, monitor and interact with a particular process or a collection of processes. These processes are diverse and can be found, for example, in household appliances, emergency shutdown systems for nuclear power stations, chemical process control and rail automation systems.

IEC is an organization that provides international standards for electrical, electronic and related technologies. The standard IEC 61131-3 [2] describes inter alia PLC programming languages. There are five PLC languages proposed in the standard. Two of them are textual languages: (a) IL - Instruction List, and (b) ST - Structured Text. The other three programming languages are graphical languages: (c) FBD - Function Block Diagram, (d) LD - Ladder Diagram and (e) SFC - Sequential Function Chart.

In this paper, the application and verification of PLCs in the rail automation domain is considered[1]. One area of applying PLCs in this domain is the area of electronic interlocking systems based on PLCs. Generally, electronic interlockings are used to control signals, points, line crossovers and level crossings, thereby ensuring safe operation. The most of the interlocking software has been written in the graphical language FBD. The goal of our work is to investigate the verification of FBDs.

In the past years, there has been an increasing interest in analyzing PLC applications with formal methods. The low-level language IL has been the most investigated language in terms of PLC verification. Hence, first attempts to verify FBDs are made by verifying the IL representation of an FBD program.

Let us briefly describe some of the approaches for IL verification. In [3], timed automata are used to model IL programs. For verification, the model checker UPPAAL is used. Function and function block calls are not implemented. [4] proposes Petri nets and SMV for model checking IL programs. As data structures, anything can be used that can be coded with 8-bits. Another method that proposes verification with SMV is sketched in [5]. Time and timers are not part of the model in this work. Comparing the existing IL verification techniques and analyzing the properties of the software to be verified, we took the latter method as a starting point. The theory behind our improvement of the IL technique and its tayloring towards the PLC family SIMATIC S7 of the Siemens AG was described in [6]. The tool that automates this process was published in [7]. This way, we managed to make model checking of IL format interlocking software fully automatic.

Unfortunately, the models became so complex that just small parts of the software could be verified. Therefore, we took a different approach in the second phase of the project. Our goal was to verify existing industrial software and not just parts of it.

So we suggested to verify FBD programs instead of IL programs. We sketched the idea in [8]. In this paper, we elaborate on the method, develop the theory behind it, and develop the techniques towards full automation. There is some work on FBD verification which has been published in the last years ([9] and [10]). But these papers do not offer enough detail to enable comparison with our work.

The paper is organized as follows. Section 2 briefly reviews the PLC structure and PLC programming languages. The theoretical background of the method for FBD verification is described in Section 3. There we introduce the textual

[1]This work has been developed while the first author was in the "Railway Automation Graduate School (RA:GS!)" of Siemens AG, Industry Sector of Braunschweig

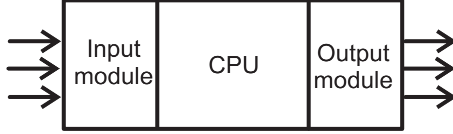


Figure 1. PLC organization

representation of FBD. Section 4 contains a case study which illustrates the application area for the work presented here. The automation of the verification method is described in Section 5. Finally, the last section draws conclusions and indicates plans for future work.

II. Programmable Logic Controllers

As already mentioned, PLCs are a special type of computer based on sensors and actuators able to control, monitor and influence a particular process. In this section, the PLC structure and programming languages are described.

A. PLC Structure

A typical PLC organization is represented in Fig. 1. Input and output modules are used to transmit data between PLC and connected peripherals. The CPU is a part of a programmable controller responsible for reading inputs, executing the control program, and updating outputs. The focus of a PLC is to repeat periodically the execution of a control program. There are three main phases of this cyclic behavior of a PLC: read data from inputs (sensors), execute the control program, and write data to outputs (actuators).

B. PLC Programming languages

The program organization units proposed in IEC 61131-3 can be delivered by the manufacturer or programmed by the user according to the rules defined in this standard. In this work, the software Step7 is used. This is the current software version for programming the PLC family SIMATIC S7 of the manufacturer Siemens AG [11].

The FBD programming language [12] is a restricted graphical representation of the machine-orientated language IL. This means that not all IL programs can be represented in FBD, but on the other hand each FBD program can be mapped to IL. FBD programs are similar to circuit diagrams in electrical engineering and consist of simple elements. For example, in Fig. 2 the following elements can be found. $CMP{=}{=}I$ (comparison of two integers), & (conjunction of two Booleans), $>=1$ (disjunction of two Booleans), and $=$ (assignment of a value to a variable).

III. Theoretical Background

With processors getting more and more powerful, and memories growing bigger and bigger, verification becomes feasible for more and more complex programs. The verification methods at hand, in particular model checking, turn out to work quite well for our application area. As a tool, we use NuSMV (a New Symbolic Model Verifier). NuSMV was developed by IRST (Instituto per la Ricerca Scientifica e Tecnologica) and CMU (Carnegie Mellon University) [13]. It is a reimplementation and extension of SMV, the first model checker based on BDD.

There is no standardized process yet to verify PLC. In this section, we present a verification process for PLC software written in FBD. There are essentially three steps:

A. in order to make FBD programs processable by NuSMV, graphical FBD programs are translated into textual textFBD programs;
B. connections between two graphical FBD elements are represented in the textFBD file by a special type of variables - *circuit variables*. In order to avoid circuit variables in the NuSMV state space, textFBD programs are translated into tFBD programs;
C. a tFBD program can then be easily represented by a NuSMV program.

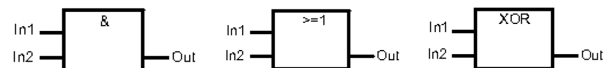
In Fig. 2, the process is shown by means of an example which we will also use later on.

A. From FBD to textFBD

We present the FBD components and their corresponding textFBD statements along with their informal semantics. Then we indicate their formal operational semantics and mention how isomorphism of FBD and textFBD semantics can be proved, referring to [14] for the details.

1) FBD and textFBD syntax: In the textFBD format of an FBD program, each graphical FBD operator is given a textual representation. We give an overview of the FBD elements and their representations in textFBD.

• **Bit operations** Logical *AND, OR* and *Exclusive-OR* operations



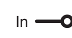
are represented in textFBD by

$$Out = (In1 \circ In2) \text{ where } \circ = \& \text{ or } | \text{ or } XOR$$

The *AND* and *OR* operations may have more than two inputs, giving rise to corresponding textFBD constructs like

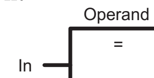
$$Out = ((In1 \ \& \ In2) \ \& \ \ldots \ \& \ In\, n)$$

The instruction *negate binary input* negates the input of an FBD operator



This is represented in textFBD by *!In*.

The FBD assignment



is simply represented by $Operand = In$.

Among the bit operations, there are also *reset output*(*R*) and *set output*(*S*).

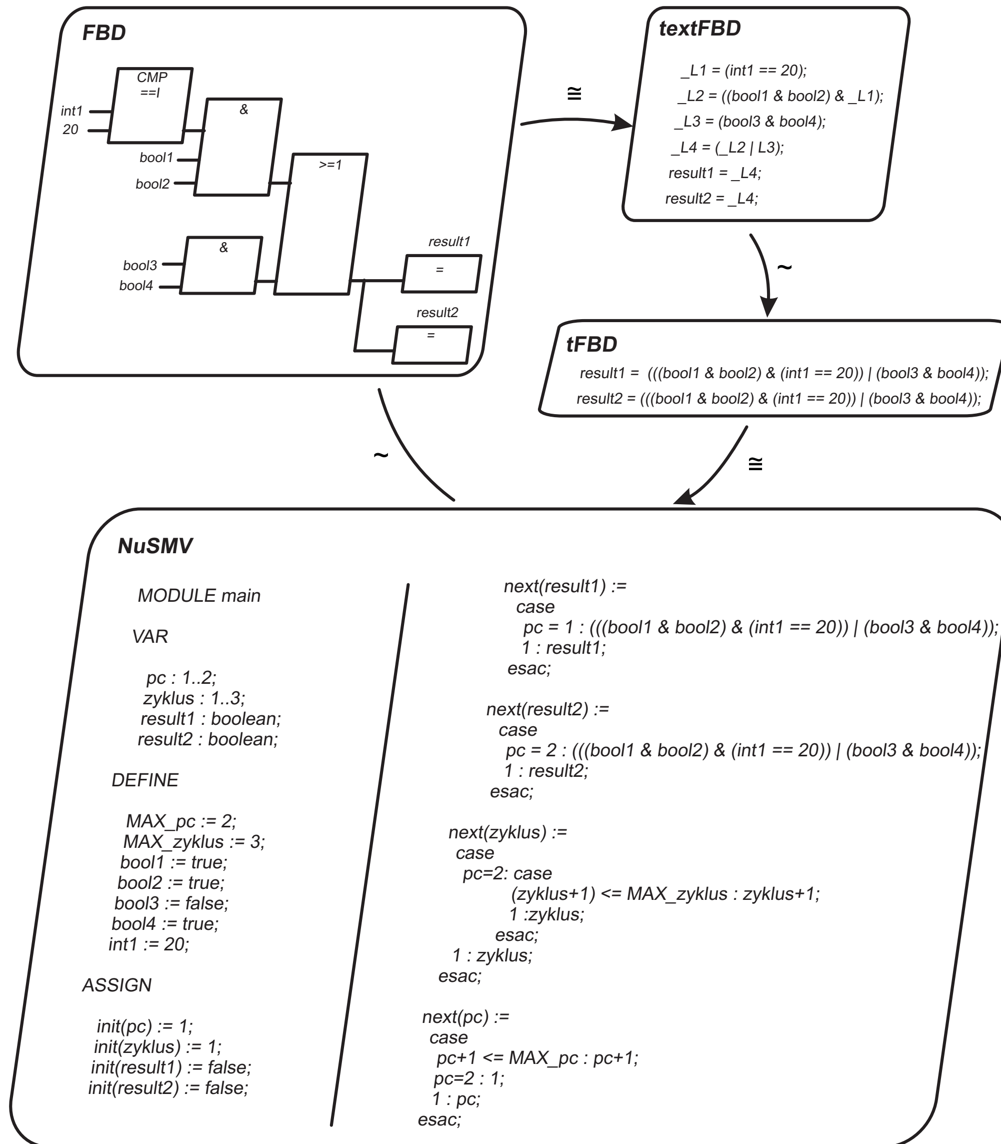
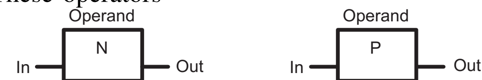


Figure 2. From FBD to NuSMV

If the input value of the *R* operator is *true*, then the operand is set to *false*. If the input value is false, then the operand is unchanged. As for the operator *S*, the operand is set only if its input value is *true*. A more precise semantics of the operators is given in [14]. Here we focus on their syntax. The operators are represented in textFBD by

$$R(Operand, In),\ S(Operand, In)$$

By means of the *N* operator *negative edge detection 1→0*, the signal state at the input is compared with that in the operand (*the edge memory bit*). If the input is *false* and the operand has stored *true* in the previous cycle, then a *negative edge* is recognized. In this case, the output is set to *true*, and to *false* otherwise. The other way around, the *P* operator *positive edge detection 0→1* recognizes a *raising edge*. These operators



are represented in textFBD by

$$Out = N(Operand, In),\ Out = P(Operand, In)$$

- **Comparators** For comparing two input values, the following comparison operators may be used: *equal (==), unequal(<>), greater (>), greater or equal (>=), less (<), less or equal (<=)*. For instance, the operator

which tests whether two inputs are equal, is represented in textFBD by $Out = (In1 == In2)$.

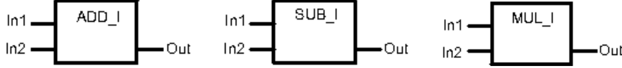
• **Jumps** Jump operations can be separated into conditional jumps and absolute jumps. Depending on the input value *true* or *false*, a conditional jump can be expressed by *JMP* or *JMPN*, respectively. The effect is to set the program counter to the position marked by *Label* if the *In* condition is *true* (*JMP*) or the *In* condition is *false* (*JMPN*), respectively.

In textFBD, this is represented by

$$JMP(In, Label),\ JMPN(In, Label)$$

An absolute jump corresponds with a *goto* statement and is simply represented by $JMP(true, Label)$.

• **Integer math instructions**



Addition, subtraction or multiplication of two integers is represented in textFBD by

$\omega(Out, In1, In2)$ where $\omega = ADD_I$ or SUB_I or MUL_I .

• **Move** The *MOVE* operator copies the value at the input to the output: $Out = In$.

For generating the textFBD file, the concept of a *circuit variable* is very important. These variables are generated when connections between two operands are to be represented. The circuit variables are marked as $_Li$ variables (cf. fig. 2).

Fig. 2 gives an impression of the translation from FBD to textFBD.

2) FBD and textFBD semantics: Let $\bar{h}$: FBD → textFBD be a map mapping each FBD element e to its corresponding textFBD representation $a = \bar{h}(e)$. The order of executing FBD operators in a network is determined by a mapping *next*: 2^{FBD} → FBD determining which element is executed next, depending on the set of elements already executed. In textFBD, this role is taken by the program counter which can be defined as a mapping p: textFBD → $\mathbb{N}$, mapping each statement a to its line number *pc* in the program.

An FBD network N may be given a transition system $\mathcal{T} = (\mathcal{C}, c_0, \rightarrow)$ as operational semantics, where $\mathcal{C}$ is the set of FBD configurations of N, c_0 is the start configuration, and $\rightarrow$ is the next-configuration relationship. An FBD configuration is a triple $c = (\sigma, e, E)$ where σ is a state of the program variables, e is the element in the network N to be executed next, and E is the set of component elements in N not yet executed.

Correspondingly, a textFBD program P may be given a transition system $\mathcal{S} = (\mathcal{D}, d_0, \hookrightarrow)$ as operational semantics, where $\mathcal{S}$ is the set of textFBD configurations, d_0 is the start configuration, and $\hookrightarrow$ is the next-configuration relationship. A textFBD configuration is a triple $d = (\sigma, a, pc)$ where σ is as above, a is the textFBD statement to be executed next, and pc is the program line in which a is.

For the details of how these transition systems are defined, we refer to [14]. There it is also shown that there is a bijective mapping $h\colon \mathcal{C} \rightarrow \mathcal{D}$ with the property that $h(c_0) = d_0$ and $c \rightarrow c' \Leftrightarrow h(c) \hookrightarrow h(c')$.

Thus, an FBD network and its corresponding textFBD program have isomorphic operational semantics, so they are equivalent in a strong sense.

B. From textFBD to tFBD

The new tFBD format has the advantage that many circuit variables are avoided, thus reducing the state space for model checking dramatically. A textFBD line in which a new circuit variable is created may be omitted in tFBD under certain circumstances. Then, in each other line of the textFBD program where this circuit variable is used, it may be substituted by the corresponding expression. The process is illustrated in Fig. 2.

To be more precise, textFBD programs are transformed to tFBD programs in the following way: each textFBD assignment $_L_i = f_i(x)$ of an expression $f_i(x)$, where x is a sequence of arguments, to a circuit variable $_L_i$ is omitted. Instead, each occurrence of $_L_i$ in righthand sides of other textFBD statements is substituted by $f_i(x)$.

Similar to a textFBD program, a tFBD program can be given an operational semantics in the form of a transition system $\mathcal{S}' = (\mathcal{D}', d_0', \hookrightarrow')$ where $\mathcal{D}'$ is the set of configurations, d_0' is the start state, and $\hookrightarrow'$ is the next-state transition function. We refer to [14] for details.

The behaviour of textFBD and tFBD transition systems with respect to circuit variables is illustrated in Fig. 3.

Clearly, since variables are eliminated, there can be no bijection between the state spaces and thus no isomorphism. Reducing the state space was precisely the motivation for introducing the transformation of textFBD to tFBD.

But still, the operational semantics of textFBD and tFBD can be equivalent, albeit in the weaker sense of observational equivalence: they can be strongly bisimilar. Let $\mathcal{S} = (\mathcal{D}, d_0, \hookrightarrow)$ be a textFBD transition system and $\mathcal{S}' = (\mathcal{D}', d_0', \hookrightarrow')$ the corresponding tFBD transition system. $\mathcal{S}$ and $\mathcal{S}'$ are strongly bisimilar, ($\mathcal{S} \sim \mathcal{S}'$), iff there is a relationship $B \subseteq \mathcal{D} \times \mathcal{D}'$ which is a strong bismulation for (d_0, d_0'). That means: $(d_0, d_0') \in B$ and for all $(d, d') \in B$ we have

$$d \hookrightarrow g \in \mathcal{D} \Rightarrow \exists g' \in \mathcal{D}' \text{ with } d' {\hookrightarrow'} g' \text{ and } (g, g') \in B$$
$$d' {\hookrightarrow'} g' \in \mathcal{D}' \Rightarrow \exists g \in \mathcal{D} \text{ with } d \hookrightarrow g \text{ and } (g, g') \in B$$

Fig. 3 shows system states before and after using circuit variables, (σ_1 or σ_1'), respectively, and (σ_j, or σ_2', or σ_k or. σ_3'), respectively.

The following example shows that there is a problem.

Example 1. *(Comparing FBD, textFBD and tFBD)*
Let x and y be two Boolean variables which are combined

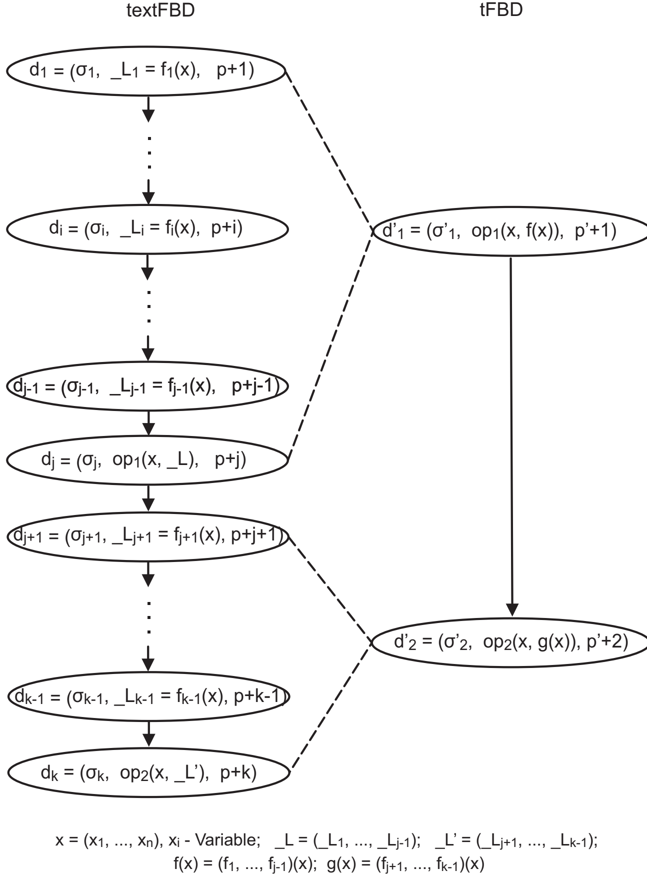


Figure 3. Substitution of circuit variables

by conjunction. If the result is *true*, then x is set to *false* by the R operator. The same happens with y. If x and y are *true* in the beginning, tFBD does not yield the expected result (cf. Fig. 4).

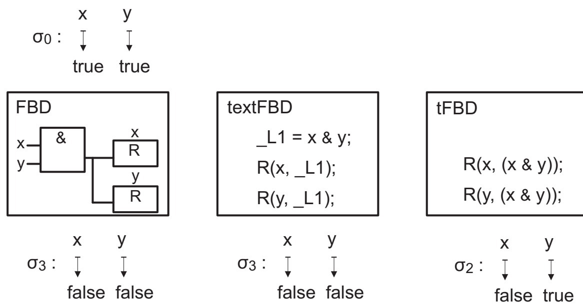


Figure 4. Example: FBD, textFBD and tFBD synopsis

The problem is solved by restricting the use of variables appropriately, forbidding situations where a circuit variable may not be substituted: 1) if an operator with local variables is to be assigned to the circuit variable; 2) if the circuit variable is used as an operand in an operator with local variables. With this restriction, strong bisimilarity between the textFBD and tFBD operational semantics can be shown [14].

Example 2. *(Strong bisimulation)*
Strong bisimulation for the example in Fig. 3 works as follows.

$$B = \{(d_1, d'_1), \ldots, (d_i, d'_1), \ldots, (d_{j-1}, d'_1), (d_j, d'_1), (d_{j+1}, d'_2), \ldots, (d_{k-1}, d'_2), (d_k, d'_2)\}$$

C. From tFBD to NuSMV

The main program is shown in *MODULE main* (cf. Fig. 2). The module may have several sections. For our FBD modeling, the *VAR*, *DEFINE*, *ASSIGN* and *SPEC* sections are used: the *VAR* section for variable declarations; the *DEFINE* section for defining symbols for frequently used expressions; the *ASSIGN* section for describing assignments, and the *SPEC* section for specifying *CTL* specifications to be checked in the model. For an example, cf. Fig. 2. For more detail, cf. [14].

A possible program property to be checked may look like this: *"at the end of the first cycle, the variables $x_1, \ldots, x_n$ should have values $a_1, \ldots, a_n$".*

Specifications for a NuSMV model can be written in CTL (Computation Tree Logic), LTL (Linear Time Logic) or PSL (Property Specification Language). For example, for specifying the above property in CTL, we have to write CTLSPEC in front of the formula. The property given above then looks as follows.

CTLSPEC AG((*cycle* = *1* & *pc* = *MAX_PC*) ⇒ ((*x1* = *a1*) & ... & (*xn* = *an*)))

Where the program cycle is determined by the integer variable $cycle$. A more extended case study is given in the next section.

The user of our software has the choice between a *modular* or a *compressed flat* representation of NuSMV functions. In modular representation, each function is given a separate module. The advantage is that modeling of FBD programs is rather straightforward. The disadvantage is that with each instantiation of a module, the state space grows very rapidly. In our compressed flat representation, all functions are specified in one module so that the state space remains constant when running the model checker. Indeed, this can be quite efficient. The disadvantage is that the functions have to be (re)modeled by hand, an effort that may only by feasible for small systems. Please note that our compressed flat representation is different from classical flattening operations like that one in NuSMV.

Summing up, the process given above gives isomorphic transformations in the first two steps, and a strongly bisimilar transformation in the third step. This way, model checking PLC software written in FBD becomes feasible in practice. This is demonstrated in the next section.

Description	Precondition	Expected reaction
From a right position, the left direction relays is activated by means of a reposition command, and the reposition process is activated	*ModuleIF* = *Right*, *ComIF* = *ReposCom*	*ModuleIF* =*Left*+*S1*
After at least 20 *ms*, the position relays is activated	=>, *iTime* = 20	*ModuleIF* = *S2*
After at least 30 *ms*, protection is activated	=>, *iTime* = 30	*ModuleIF* = *S3*
After at least 40 *ms*, the frog is started to move	=>, *iTime* = 40	*ModuleIF* = *S4*

Table I
EXAMPLE OF A TEST CASE: MOVING THE FROG OF A POINT LEFT

ModuleIF - Module Interface

	Left	*Right*	*S1*	*S2*	*S3*	*S4*
bPointPosition	0	1				
bDirectionRight	0	1				
bDirectionLeft	1	0				
bRepetition			1	0		
bRepositionActive			1			
bPointPositionRelays				1	0	
bProtection					1	0
iTime						0

ComIF - Command Interface

	ReposCom
bComActive	1
bFunction	1
iElement	100

Table II
DEFINITION OF INTERFACES - VARIABLE DOMAINS

IV. CASE STUDY

Here we describe how the method is applied in the area of railway automation. We concentrate on FBD software as it is used by one family of PLC-based interlocking systems. Interlocking systems are railway facilities which are used for the central control of points and signals (cf. [15]). They have outdoor and indoor parts. The indoor parts consist of hardware and software.

The interlocking software is composed of several components which are responsible for controlling the various interlocking functions. One such component is a point. Like the other components, it consists of several code modules. These code modules depend on the equipment, but each one is designed for only one point. One function block diagram in such a code module of a point component is responsible for controlling the point. This function block diagram is used here as a use case for demonstrating the verification method proposed in this paper.

From this module, the corresponding NuSMV model is created using our method. The model is presented below. We start with describing the test cases, taken from practice, which are used for checking correctness of the software. The test cases form the basis for constructing the verification scenarios.

A. Test case description

In table I we give a simplified description of a test case of the code module for point control. The activation of the point actuator component is checked in four steps before moving the point blade. Each step is represented by a description, a definition of preconditions (start configuration of the test case), and a definition of the expected reaction.

For better understanding table I, we explain the concept of a *variable domain* in more detail. In the description of preconditions and expected reactions of a test case, we use the integer program variable *iTime* which changes with the PLC *cycle* and may be controlled by the software.

We also use variable domains like *ModuleIF* or *ComIF*. A variable domain contains a description of several program variables.

This can be better explained using the variable domain *ModuleIF* as an example. It is defined in table II. The following variables belong to this domain: *bPointPosition, bDirectionRight, bDirectionLeft, bRepetition, bRepositionActive, bPointPositionRelays, bProtection* and *iTime*. If *ModuleIF* has the value *Right*, then *bPointPosition*=1, *bDirectionRight*=1 and *bDirectionLeft*=0.

Variable domains are used in the description of interfaces. As shown in table II, the module interface is described by the variable domain *ModuleIF* and the variables *bPointPosition*, *bDirectionRight*, *bDirectionLeft*, etc. A variable domain contains different variable assignments in different test cases. In order to enable a clear test case description, all variable assignments are listed for each variable domain. For example, the variable assignments for *ModuleIF* is defined by *ModuleIF* $\in$ {*Left, Right, S1, S2, S3, S4,* ...}.

Thus, the precondition in the first step of the test case is represented by *ModuleIF*=*Right* expressing that the direction relays should be in the right position. This means that, before executing the test case, we must have *bPointPosition*=1, *bDirectionRight*=1 and *bDirectionLeft*=0 (cf. table II). Similarly, at the beginning of the test case, *ComIF* is defined as *ReposCom*. If an interface is not specified in the precondition of a test case, it assumes its initial value which should be defined at the beginning of the test table.

When describing the preconditions of the 2nd, 3rd and 4th step of the test case, we have the symbol combination “=>”. This means that the test case is based on the previous step. For instance, in the 2nd step, all variables except for *iTime* have their values from the expected outcome of the 1st step.

The latter is described by *ModuleIF=Left+S1*. That means that *bPointPosition*=0, *bDirectionRight*=0, *bDirectionLeft*=1, *bRepetition*=1 and *bPositionActive*=1.

When adding two or more variables of a variable domain symbolically, it is possible that a variable occurs in the description of both (or all) elements of the sum. In this case, the variable is to be assigned the value of the last occurrence. For instance, *bPointPosition* is given the value 1 in *Left+Right*.

When describing the preconditions of a test case, we may also assign values to single variables instead of value domains. For instance, the variable *iTime* is assigned different values in the 2nd, 3rd and 4th steps of the test case. The same holds when describing the expected reaction.

B. NuSMV Component Model

An extract of the NuSMV model of the point component is shown in table III. We show only lines from the tFBD model in which the following program variables change: *bPointPosition, bDirectionRight, bDirectionLeft, bRepositionActive, bRepetition, bComActive, bFunction, iElement*. These are the variables which are used in the specification. The variables having no role in our context are denoted by var1, var2, etc. in table III.

In the model description as well as in the specification, the data are represented in a changed format in order to hide the reference to the original software. The rules for transforming FBD programs into the NuSMV input language are described above in section III. In this section, we show how to represent specifications as logical formulae.

C. Description of verification scenarios

In the case study, verification scenarios are described in CTL. Two parameters are important: 1) the program counter *pc* which assumes the value *MAX_pc* at the end of a program cycle, and is then set to 1 again; 2) the cycle counter *cycle*. As said before, the test case descriptions of the component serve as basis for formulating the verification scenarios. Since a test case is defined by a precondition and an expected reaction, we may represent this by the following scenario.

$$\begin{aligned} & ModuleIF = Right \quad \& \quad ComIF = ReposCom \Rightarrow \\ & \quad \mathsf{AG}\big((cycle = 1 \quad \& \quad pc = MAX_pc) \Rightarrow \\ & \qquad ModuleIF = Left + S1 \big) \end{aligned} \tag{1}$$

Bearing in mind the interface description in table II, the first step of the scenario may be represented by the following formula.

$$\begin{aligned} & ((bPointPosition = 1 \quad \& \quad bDirectionRight = 1 \quad \& \\ & \ bDirectionLeft = 0) \quad \& \quad (bComActive = 1 \quad \& \\ & \ bFunction = 1 \quad \& \quad iElement = 100)) \Rightarrow \\ & \quad \mathsf{AG}\big((cycle = 1 \quad \& \quad pc = MAX_pc) \Rightarrow \\ & \qquad (bPointPosition = 0 \quad \& \quad bDirectionRight = 0 \quad \& \\ & \qquad bDirectionLeft = 1 \quad \& \quad bRepetition = 1 \quad \& \\ & \qquad bRepositionActive = 1)\big) \end{aligned} \tag{2}$$

```
tFBD
...
48   R(bRepositionActive, (var10 | var11 | var12));
...
50   R(bRepositionActive, ((((var13 & !var14) & (var15 &
     !var16)) | var17) | var18));
...
52   R(bRepositionActive, (var13 & !var14) & (var19 > var20));
...
88   bPointPosition = var1;
...
96   bRepositionActive = ((var21 & !var22) | ((!var21 & !var23)
     | var24));
...
104  bDirectionRight = bPointPosition;
105  bDirectionLeft = !var2;
...
120  S(bPointPosition, var2);
...
122  S(bRepetition, var3);
123  R(bRepetition, ((var4 & !var5) | ((var6 & !var7) & var8) |
     ((var7 & !var6) & var9)));
...
131  R(bPointPosition, var2);
...
```

```
DEFINE
bComActive := 1;
bFunction := 1;
iElement := 100;

ASSIGN
init(bPointPosition) := 1;
next(bPointPosition) := case
    pc = 88 : var1;
    pc = 120 & var2 : 1;
    pc = 131 & var2 : 0;
    1 : bPointPosition;
  esac;

init(bDirectionRight) := 1;
next(bDirectionRight) := case
    pc = 104 : bPointPosition;
    1 : bDirectionRight;
  esac;

init(bDirectionLeft) := 0;
next(bDirectionLeft) := case
    pc = 105 : !var2;
    1 : bDirectionLeft;
  esac;

init(bRepetition) := 0;
next(bRepetition) := case
    pc = 122 & var3 : 1;
    pc = 123 & ((var4 | !var5) | (((var6 & !var7) & var8) | ((var7 &
!var6) & var9))) : 0;
    1 : bRepetition;
  esac;

next(bRepositionActive) := case
    pc = 48 & (var10 | var11 | var12) : 0;
    pc = 50 & ((((var13 & !var14) & (var15 & !var16)) | var17) |
var18) : 0;
    pc = 52 & ((var13 & !var14) & (var19 > var20)) : 0;
    pc = 96 : ((var21 & !var22) | ((!var21 & !var23) | var24));
    1 : bRepositionActive;
  esac;
```

Table III
NUSMV MODEL OF THE POINT COMPONENT

The remaining three formulae are created in a similar way. As mentioned before, the symbol combination "=>" says that the precondition of the current step has to be extended with the expected reaction of the previous step. For the 2nd step, we thus have

$$(ModuleIF = \ Left + S1 \ \ \& \ \ iTime = 20) \Rightarrow \mathsf{AG}(\ (cycle = 1 \ \ \& \ \ pc = MAX_pc) \Rightarrow ModuleIF = S2) \quad (3)$$

Alternative representation of the specification: If only one formula is to be checked, we may take the variable assignments from the precondition and define their values as initial values of the model variables. Then a scenario may be described by the following formula

$$\mathsf{AG}(\ (cycle = 1 \ \ \& \ \ pc = MAX_pc) \ \Rightarrow \ \text{"expected reaction"}) \quad (4)$$

With this approach, we build a slightly different NuSMV model where the set of initial states in the model is reduced. As an example, table III shows part of the model which refers to some of the variables of our accompanying test case. There the initial values of the variables *bPointPosition, bDirectionRight* and *bDirectionLeft* are defined using the *init* clause. In contrast, the variables *bComActive, bFunction* and *iElement* are defined in the DEFINE section because their values are unchanged in the model.

D. Verification results

The textFBD format of the software component under consideration has 165 lines of code. It uses about 100 variables (90 Boolean and 10 integer). The model verification was performed on an Intel(R) Xeon(R) CPU 5150 computer with 2,66 GHz and 3,25 GB RAM.

A detailed description of the NuSMV model checker may be found in [16] and [17]. Its most important properties are summarized in [13]. The basic steps of NuSMV verification work as follows.

1) in the first step, the model is read; an internal hierarchic representation is set up and stored
2) in the second step, the hierarchic representation is transformed into a flattened representation; it contains only one module in which all modules and processes are instantiated
3) then the BDD variables are generated
4) the flattened model is represented using BDDs
5) after generating the BDD representation, the CTL specifications can be checked

The execution of the first three steps took about 1 second. The execution times for the other two steps was different depending on which of the variants explained above was used.

One model for all scenarios: Formulae (1), (2) and (3) describe how the specification for the first variant of the NuSMV model is constructed. The initial values of the variables in question are not defined in the model. From

<N-Statement>	::= . . .	
	| <Bitlogic> <BitlogicEnd>	(1)
<Bitlogic>	::= . . .	
	| A <Operand> <End>	(2)
	| A(<End> <Compare>) <End>	(3)
	| <Bitlogic> A <Operand> <End>	(4)
	| <Bitlogic> O <End> <Bitlogic>	(5)
<Compare>	::= L <Operand> <End> L <Operand> <End> COMPARETOK <End>	(6)
<BitlogicEnd>	::= . . .	
	| <AssignEndN>	(7)
<AssignEndN>	::= = <Var> <End> <AssignEnd>	(8)
<AssignEnd>	::= | <AssignEnd> = <Var> <End>	(9)

Table IV
EXCERPT OF THE GRAMMAR

about 10^{65} states, about 10^{14} states were reachable. Setting up the BDD-based model took slightly more than half an hour. Checking the specifications took 30 to 80 seconds per formula.

One model for one scenario: In the second case, a model is generated for each scenario. The preconditions for the scenario are initialized in the model. The specification is given in the form of formula (4). In the NuSMV model, about 6000 of the 10^{60} states were reachable. Setting up the BDD-based model took about 40 minutes. Checking the specification then took less than 1 second.

V. AUTOMATION OF THE VERIFICATION METHOD

As mentioned before, the CTL specification is set up using the descriptions of the test case and the module interfaces (cf. Fig. 5). CTL formulae are constructed automatically by combining information from the tables. Creating the NuSMV model, however, is a little more complicated, but it can be automated.

With the method described here, an arbitrary FBD program is modeled in a way that makes it possible to verify it with the NuSMV model checker. As mentioned above in section III, we propose to first construct the textFBD model, then the tFBD model, and then the NuSMV model. In what follows, we give a more detailed description of this process.

A. Constructing the textFBD format

This is the first step in verifying an FBD program. The textFBD representation is generated from the IL representation of the program. IL is a machine-oriented PLC programming language, and each FBD program can be represented in IL. The range of textFBD statements is equivalent to that of FBD statements (cf. [12]).

The IL format of an FBD program can be transformed to textFBD using an unambiguous context-free grammar. An excerpt of it is shown in table IV. In the table, transformation rules are shown as they are used for transforming the network in figure 2. The IL format of the network is shown in figure 6.

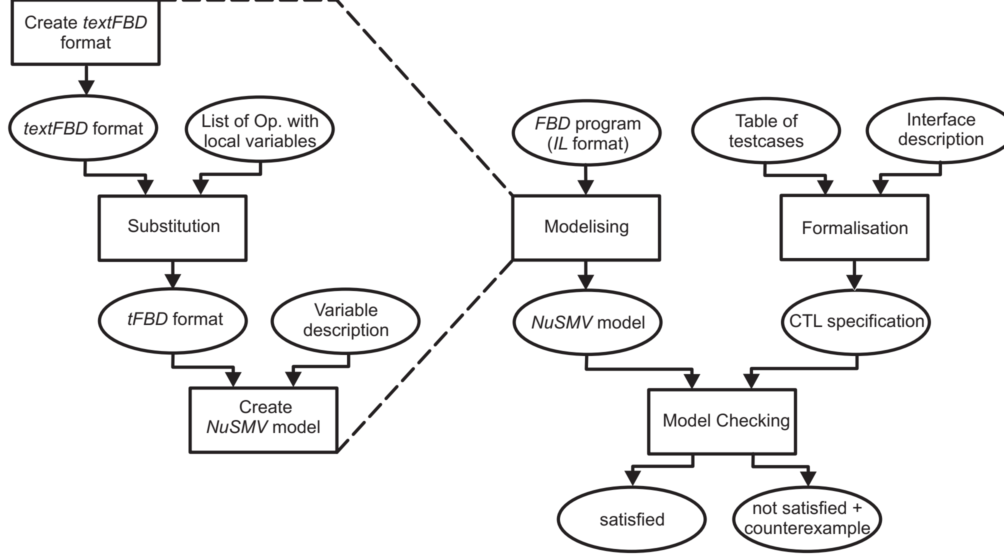


Figure 5. Model checking FBD programs

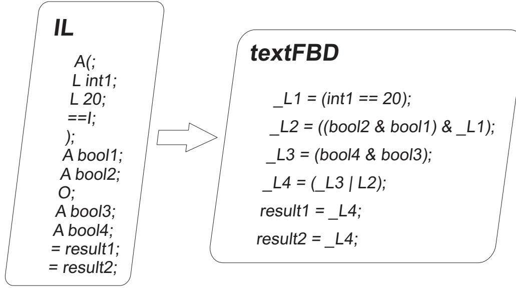


Figure 6. Example: IL and textFBD formats of an FBD program

The example network may be considered as a complex statement consisting of a logical bit operation followed by two assignments (cf. rule (1) in table IV). A logical bit operation always has an output. Since no dangling lines may exist in a network, this output must be consumed in some way. In our case, the logical bit operation ends with assignments (cf. rule (7)). To enable using two such assignments, rules (8) and (9) are needed.

Among the logical bit operations, we have a comparison of two integers (rule (6)). If the result of this operation is to be combined with the *AND* of two further Boolean variables, first rule (3) and then rule (4) must be applied. An *AND* operation of two Boolean variables can be recognized using rules (2) and (4). Rule (5) describes how two logical bit operations can be combined with an *OR* operation.

The circuit variables in the textFBD file ($_Li$ variables) are generated when connections between two operands are to be represented (cf. fig. 6). Such a variable is generated when the recognition of an FBD operand is terminated. In our example, this means the following.

- As soon as the comparison is recognized, $_L1$ is generated.
- Then the recognition of the *AND* operation follows (until the *OR* operation is read), with the subsequent generation of $_L2$.
- In order to execute the *OR* operation, the subsequent *AND* operation ($_L3$) is needed.
- Only then $_L4$ is generated as a disjunction of $_L2$ and $_L3$.
- Finally, the result of the logical bit operation in variable $_L4$ is assigned to the variables *result1* and *result2*.

On the basis of the grammar and the circuit variable concept, textFBD files are generated.

B. Constructing the tFBD format

Although the textFBD format of the FBD program can be transformed to NuSMV directly, we first minimize the model in order to minimize the NuSMV state space. As shown in section III, we may do away with many circuit variables and thus reduce the model size. This substitution is expressed in the tFBD format of the FBD program.

For constructing the tFBD format, only the list of FBD operations is needed which use local circuit variables. When transforming a textFBD file, each statement is checked whether it uses an operator from the list. If yes, it is copied into the tFBD file without change. Otherwise, we first substitute the circuit variable (cf. fig. 5).

C. Constructing the NuSMV model

Transforming a tFBD file to the NuSMV input language has been treated above in subsection III-C. We have shown how to represent tFBD statements by NuSMV transformation rules. In order to complete a NuSMV model, the

variable declarations have to be added (cf. Fig. 5). Here, for instance, integer ranges are declared . Overflow is avoided by the way the programs are constructed.

VI. Conclusion

In this paper, we present a method for the automated formal verification of PLC software. In particular, we look at FBD software. In order to verify the software, we propose to represent the graphical SPS programming language textually in two observationally equivalent ways: textFBD and tFBD. From the latter format, we derive a NuSMV model. Its state space is dramatically smaller than that of a NuSMV model derived directly from textFBD, so that applications of practical size can be model checked.

The method was put to the test in the area of railway automation. In a case study, a component of an interlocking software, the logic controlling a point, was verified. The design of the model as well as the construction of the NuSMV model were automated. With this successful project, we are confident to pave the way for applying the method in practice.

NuSMV was chosen by convention, not by rational decision. So it would be interesting to look at other model checkers, for instance SPIN, and see whether improvements in the process can be achieved.

There are two important aspects in applying formal verification in practice: 1) the method should be applicable to relatively big and realistic models; 2) the execution times should be acceptable. The 1st point is satisfied by our method because the case study is directly taken from practice, without any omissions or abstractions which would make it more "academic". As for the 2nd point, the execution times for setting up and transforming the model and verifying the specifications are acceptable. It is true that they are greater than what simulations would take. But there is a fundamental difference between simulation and verification where the entire state space is being explored.

Our realistic case study is an important step to convince not only railway engineers that automated formal methods are practical. The next step in applying our methods should be a state-based specification. This way, we expect that the advantage of our method would become even more evident.

Acknowledgements

We are very grateful to the three anonymous referees who did a wonderful job in caring about the details of our paper and made many valuable suggestions. For all remaining deficiencies, of course, the authors are responsible.

References

[1] W. Giessler, *SIMATIC S7 SPS-Einsatzprojektierung und -programmierung*. VDE Verlag GMBH, 2005.

[2] International Electrotechnical Commission, *International Standard 61131-3, Programmable controllers - Part 3: Programming languages*, 2003.

[3] A. Mader and H. Wupper, "Timed automaton models for simple programmable logic controllers," in *Proc. of 11th Euromicro Conference on Real Time Systems*, 1999, pp. 114–122.

[4] M. Heiner and T. Menzel, "A Petri net semantics for the PLC language Instruction List," in *Proc. of the International Workshop on Discrete Event Systems (WoDES)*, 1998, pp. 161–166.

[5] G. Canet, S. Couffin, J. j. Lesage, and A. Petit, "Towards the automatic verification of PLC programs written in Instruction List," in *IEEE International Conference on Systems, Man and Cybernetics*, 2000, pp. 2449–2454.

[6] O. Pavlovic, R. Pinger, M. Kollmann, and H. Ehrich, "Principles of formal verification of interlocking software," in *Proc. of the 6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*, E. Schnieder and G. Tarnai, Eds. GZVB, 2007.

[7] O. Pavlovic, R. Pinger, and M. Kollmann, "Automation of formal verification of PLC programs written in IL," in *Proc. of 4th International Verification Workshop in connection with CADE-21*, B. Beckert, Ed. CEUR-WS.org, 2007.

[8] ——, "FBD-based PLC verification demonstrated on interlocking software," in *International Conference : ERTS EMBEDDED REAL TIME SOFTWARE 2008*, S. 01/02/2008, Ed.

[9] M. J. Song, S. R. Koo, and P.-H. Seong, "Verification method for the FBD-style design specification using SDT and SMV," in *IASTED Conf. on Software Engineering*, 2004, pp. 206–211.

[10] K. Y. Koh, E. K. Jee, S. J. Jeon, P. H. Seong, and S. D. Cha, "A formal verification method of Function Block Diagrams with tool supporting: Practical experiences," in *Annals of DAAAM for 2008 & Proceedings of the 19th International DAAAM Symposium*, 2008.

[11] *Working with STEP 7 V5.3*, SIEMENS, 2004.

[12] *Function Block Diagram (FBD) for S7-300 and S7-400 Programming*, SIEMENS, 2004.

[13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model verifier," *International Journal on Software Tools for Technology Transfer*, vol. 2, 2000.

[14] O. Pavlovic, *Formale Verifikation von Software für speicherprogrammierbare Steuerungen mittels Model Checking*. TU Braunschweig, 2009, Dissertation.

[15] J. Pachl, *Systemtechnik des Schienenverkehrs. Bahnbetrieb planen, steuern und sichern*. Vieweg+Teubner, 2008.

[16] R. Cavada, A. Cimatti, C. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev, *NuSMV 2.4 User Manual*, CMU and ITC-irst, http://www.nusmv.irst.itc.it.

[17] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, and M. Roveri, *NuSMV 2.2 Tutorial*, CMU and ITC-irst, http://www.nusmv.irst.itc.it.