Automatically Deriving Symbolic Invariants for PLC Programs Written in IL

Sebastian Biallas[1], Jörg Brauer[1], Stefan Kowalewski[1] and Bastian Schlich[2]

[1] Embedded Software Laboratory, RWTH Aachen University
lastname@embedded.rwth-aachen.de
[2] Industrial Software Systems, ABB Corporate Research
firstname.lastname@de.abb.com

Abstract. In this paper, we propose a new approach to automatically derive invariants from Programmable Logic Controller programs by symbolically rewriting Instruction List code. These invariants describe the relations between all variables and capture the behavior of the program. Usually, invariants are created by users and verified using formal verification techniques such as model checking or static analysis. The process of manually deriving invariants, however, is error-prone and lengthy. Our approach generates these invariants automatically and removes the need to use formal verification techniques to verify them. Users only need to inspect the generated invariants and compare them to the expected program behavior. Using three example programs of different sizes, we show that the generated invariants are easy to understand and that the approach indeed scales for larger programs.

1 Introduction

Programmable Logic Controllers (PLCs) are frequently used in safety-critical systems, where the application of formal verification methods is recommended [1]. In the past, formal methods such as model checking [2, 3] or static analysis [4] have been applied to this task. Model checking, for instance, is used to verify whether the model of a systems satisfies the system's requirements. The generation of the model can in many cases be done automatically. However, expressing requirements, which are often given in natural language, in terms of temporal logic, is a very time-consuming and error-prone process.

Despite the advances in this regard [5], this drawback limits the applicability of formal verification methods to industrial applications. To alleviate this problem, we propose to use a different approach: Instead of verifying a previously stated specification, our method derives symbolic invariants.

Invariants as such have long served for reasoning about correctness of programs. They can describe relations between the valuations of variables that hold after the execution of a program fragment regardless of the input values [6]. Examples of invariants are, for instance, $x = -y$ or $x \leq 0$.

To derive invariants, our approach considers an input program written in Instruction List (IL) and iteratively rewrites this program into a symbolic repre-

sentation over quantifier-free linear arithmetic with Boolean connectives, following the semantics of the involved operations.

Our method is different from existing work in that it directly targets PLCs running in the *cyclic execution mode*, which consists of three steps, each of which is executed atomically: reading inputs, processing data, writing outputs. Our approach translates the semantics of the instruction set into linear constraints and then uses a set of rewriting rules to derive invariants. Since PLCs are typically running in safety-critical systems where timeliness is a strong concern, most of these programs do not consist of long running loops (not to say, infinite loops). This property ensures that the derived arithmetic expressions do not grow without bounds.

In previous work, Pavlovic et al. [3] have translated programs written in *Statement List* into the input language of the model checker NuSMV [7]. Their most important contribution was the formal verification of the program depicted in the left hand side of Fig. 1. As input for NuSMV, they used the following specification given in the temporal logic LTL [8]:

$$G(\mathtt{PC} = 2 \Rightarrow \mathtt{Byte} = (\mathtt{Bit0} + 2 * \mathtt{Bit1} + 4 * \mathtt{Bit2} + 8 * \mathtt{Bit3} + 16 * \mathtt{Bit4} + 32 * \mathtt{Bit5} + 64 * \mathtt{Bit6} + 128 * \mathtt{Bit7})) \tag{1}$$

In summary, this specification states that the program converts a bit-vector of length 8 into an unsigned byte. The approach described by Pavlovic et al., however, has two drawbacks. On the one hand, the runtime requirements for their approach is significant: Model checking took approximately 8h (even though this could be reduced to 113s with manual intervention). On the other hand, the specification needs to be formulated manually.

Our method can be seen as a response to these problems: By rewriting the instructions in the program, it derives the stated invariant automatically. Further, the runtime is essentially non-measurable, requiring less than 0.1s overall. In summary, we make the following contributions:

- We detail an automatic approach for deriving invariants from PLC programs written in IL.
- We present the effectiveness of this approach on three examples, where precise and expressive invariants were derived.

Our approach is as follows. The program is first rewritten into a so-called static single assignment (SSA) form, that explicitly shows all calculation steps in a symbolic representation (cp. Sect. 2). Then, this explicit form is used to derive symbolic invariants by analyzing the SSA expressions stored in output variables at the end of the PLC cycle (cp. Sect. 3). We additionally present a second example, where two invariants are derived depending on the actual input values and a third example analyzing an implementation of a PLCopen safety function block. The paper is concluded by presenting related work (cp. Sect. 4) and discussing results and future work (cp. Sect. 5).

2 Rewriting of IL Programs Into SSA Form

For the application of our method, we are analyzing IL programs, which is is one of the standardized languages for programming PLCs [9]. It is accumulator-based and similar to many machine languages. With its simple semantics it is ideal for deriving symbolic information.

We will motivate our approach with the `FromByte` program according to Pavlovic et al. [3], which was translated to IL [2]. An excerpt of this program is shown in Fig. 1 on the left side. It has eight Boolean inputs named `in0` to `in7` and converts these to the byte represented by them. This is accomplished by converting the inputs to the corresponding integer (*false* to 0 and *true* to 1), multiplying them by their significance and adding up the results in a temporary variable called `temp`.

For deriving symbolic invariants, we begin by rewriting IL programs into an SSA form [10]. In this form, each IL instruction is written as an assignment. Each of these assignments creates a new instance of the accumulator or a variable, indicated by a superscript number. If, e. g., the current accumulator $\mathsf{acc}^{(i)}$ is incremented by 1, the SSA expression

$$\mathsf{acc}^{(i+1)} := \mathsf{acc}^{(i)} + 1$$

would be generated. The superscript number is called *instance number*. On the left hand side (LHS) of such expressions, there is either a new instance of the accumulator or a new instance of some program variable. The right hand is side (RHS) is either

- a constant,
- a program variable that was not yet used on a LHS,
- an arithmetic, logic or relational operation of existing LHSs, or
- a data type cast. For example, a cast of the accumulator to the `BYTE` type is written u8(`acc`), indicating the 8 bit unsigned type.

The transformation of the IL program into SSA form is performed automatically. In order to achieve this, different execution paths are separated by partitioning the possible ranges of input variables as presented in [2]. Additionally, all loops are unrolled, removing conditional execution all together. Formally, the translation is shown in Fig. 2 for the instructions `LD`, `ST`, `ADD`, `SUB` and `AND`. We assume that there are already i instances of the accumulator, so a write to the accumulator creates instance $i + 1$. For the store to byte variable `var` a new instance $var^{(j+1)}$ is created assuming that there are already j instances. The symbol x represents either a constant or an existing LHS and is unchanged.

For now, this approach is limited to integer arithmetic and Boolean logic. Converting to other types results in separate invariants for all possible integer values.

For the `FromByte` program the results are presented on the right side of Fig. 1. In the first line, the instance $\mathsf{acc}^{(0)}$ of the accumulator is created, resembling the load instruction of the variable `in0`. After the 39 lines of the program, the

1	LD	in0	$\mathsf{acc}^{(0)} := {in_0}^{(0)}$
2	BOOL_TO_BYTE		$\mathsf{acc}^{(1)} := \mathrm{u8}(\mathsf{acc}^{(0)})$
3	ST	temp	$temp^{(0)} := \mathrm{u8}(\mathsf{acc}^{(1)})$
4	LD	in1	$\mathsf{acc}^{(2)} := {in_1}^{(0)}$
5	BOOL_TO_BYTE		$\mathsf{acc}^{(3)} := \mathrm{u8}(\mathsf{acc}^{(2)})$
6	MUL	2	$\mathsf{acc}^{(4)} := \mathsf{acc}^{(3)} * 2$
7	ADD	temp	$\mathsf{acc}^{(5)} := \mathsf{acc}^{(4)} + temp^{(0)}$
8	ST	temp	$temp^{(1)} := \mathrm{u8}(\mathsf{acc}^{(5)})$
	⋮		⋮
38	ADD	temp	$\mathsf{acc}^{(29)} := \mathsf{acc}^{(28)} + temp^{(6)}$
39	ST	from_byte	$from\_byte^{(0)} := \mathrm{u8}(\mathsf{acc}^{(29)})$

Fig. 1. Program FromByte and equivalent SSA form

expression $\mathrm{u8}(\mathsf{acc}^{(29)})$ (a BYTE cast of the 29$^{\text{th}}$ instance of the accumulator) is assigned to the output variable from_byte. Based on this final expression, we will derive the invariant of this program in the next section.

3 Generation of Invariants

In this section we will use the SSA form defined in the last section to generate symbolic invariants. The key idea is that for each variable introduced in SSA, we still have symbolic information how the value was calculated if we inspect the corresponding RHS. By inducing and thereby replacing all LHSs with the corresponding RHS expressions until we reach constants or variables with an instance number of 0, we can build up symbolic expressions for each LHS. If we do this for the output variable from_byte of the example program at the end of the PLC cycle, we obtain the following resolution steps:

$$
\begin{aligned}
\mathsf{from\_byte} &= from\_byte^{(0)} \\
&= \mathrm{u8}(\mathsf{acc}^{(29)}) \\
&= \mathrm{u8}(\mathsf{acc}^{(28)} + temp^{(6)}) \\
&= \mathrm{u8}((\mathsf{acc}^{(27)} * 128) + \mathsf{acc}^{(25)}) \\
&= \mathrm{u8}((\mathsf{in7} * 128) + \mathrm{u8}(\mathsf{acc}^{(24)} + temp^{(5)})) \\
&\dots \\
&= \mathrm{u8}((\mathsf{in7} * 128) + \mathrm{u8}((\mathsf{in6} * 64) + \mathrm{u8}((\mathsf{in5} * 32) + \mathrm{u8}((\mathsf{in4} * 16) \\
&+ \mathrm{u8}((\mathsf{in3} * 8) + \mathrm{u8}((\mathsf{in2} * 4) + \mathrm{u8}((\mathsf{in1} * 2) + \mathrm{u8}(\mathsf{in0}))))))).
\end{aligned}
$$

Here, each step consists of a replacement of a LHS by its corresponding RHS expression by a look-up in the list of SSA expressions. In the step from the second to the third line, e. g., $\mathsf{acc}^{(29)}$ was replaced by $\mathsf{acc}^{(28)} + temp^{(6)}$ (cf. Fig. 1 line 38). Afterwards, we can apply some simplifications on the expression found, to make them more readable and easier to understand.

LD x	$\mathsf{acc}^{(i+1)} := x$
ST var	$var^{(j+1)} := \mathrm{u8}(\mathsf{acc}^{(i)})$
ADD x	$\mathsf{acc}^{(i+1)} := \mathsf{acc}^{(i)} + x$
SUB x	$\mathsf{acc}^{(i+1)} := \mathsf{acc}^{(i)} - x$
AND x	$\mathsf{acc}^{(i+1)} := \mathsf{acc}^{(i)}$ and x

Fig. 2. IL instructions and how the are represented in SSA form

This includes

- the folding of constants (e. g. (*false* or *false*) is rewritten as *false*),
- the removal of unnecessary casts,
- the elimination unused subexpressions (e. g. (*true* or *expr*) is rewritten as *true*).

These steps are repeated until there are no further simplifications.

The crucial point here is that the final invariant for `form_byte` is exactly the invariant that was manually specified as the LTL formula (1) by Pavlovic et al. This means we can derive the invariant without a time-consuming model checking process here. Often, these invariants also give insight into the program behavior without manual writing specifications.

Now, we will formalize the derivation of the invariants. For each non-temporary program variable `var`, we have some instance $var^{(0)}$ which corresponds to the value of the variable at the beginning of the PLC cycle. This can be either an input value read from a sensor or the last value of the previous cycle. For each output variable $\mathtt{out}_i$, on the other hand, we have some final value ${out_i}^{(n_i)}$, where n_i is maximal. This corresponds to the final output value at the end of the PLC cycle. If we derive the invariants for all variables ${out_i}^{(n_i)}$ of the program using this technique, we get the dependence — for each cycle — of the new variable values on the old values (possibly input values) as symbolic invariants.

These invariants sometimes depend on the actual values of the inputs, which we will now show on an additional example. The program shown in Fig. 3 has two variables `X`, `Y` of type `BYTE`, where `X` is an input variable and `Y` is an internal variable. In each cycle, `Y` is decremented if `X` is greater than 100, otherwise incremented. Since this program has two different execution paths for $\mathtt{X} \in [0, 100]$ and $\mathtt{X} \in [101, 255]$, we get two different invariants for this program. Generating the invariants for the second program, results in the two invariants

$$\begin{aligned} \mathtt{X} \in [0, 100] &\implies Y^{(1)} = \mathrm{u8}(\, Y^{(0)} + 1) \\ \text{and } \mathtt{X} \in [101, 255] &\implies Y^{(1)} = \mathrm{u8}(\, Y^{(0)} - 1) \end{aligned}$$

for the distinct execution paths, where $Y^{(0)}$ and $Y^{(1)}$ are the values in `Y` before and after executing the cycle.

As a real-world example, we also generated invariants for an implementation of the PLCopen safety function block *emergency stop* [11], kindly provided by Soliman and Frey [12]. The function block has 5 Boolean inputs and the

```
1   VAR_INPUT   X: BYTE; END_VAR
2   VAR         Y: BYTE; END_VAR
3         LD    X
4         GT    100
5         JMPC  m
6         LD    Y
7         ADD   1
8         ST    Y
9         RET
10  m:    LD    Y
11        SUB   1
12        ST    Y
13        RET
```

Fig. 3. Example program with input-dependent invariants

implementation uses 11 internal variables (giving 2^{16} possible configurations). The implementation uses the internal variables to control the current state while monitoring the emergency stop signal, the reset function, etc. With our method, we derive 75 invariants for the function block. A typical invariant is:

$$\begin{array}{rrcl} & Activate^{(0)} = false & & Ready^{(1)} = false \\ \wedge & {S_1}^{(0)} = false & \Longrightarrow & \wedge\ S\_EStopOut^{(1)} = false \\ \wedge & {S_2}^{(0)} = true & & \wedge\ {S_1}^{(1)} = true \wedge {S_2}^{(1)} = false. \end{array}$$

The variables S_1 and S_2 indicate the states *idle* and *init*, while the other variables are the input and output variables according to the PLCopen specification. The variables of the program not shown are not relevant for the invariant. By this invariant, we can deduce that if we are in state *init* and the `Activate` signal is reset, the outputs `Ready` and `S_EStopOut` are reset and the state *idle* is signaled, independently of all other inputs or program states.

To gain an overview over program behavior, we provide a means for filtering the invariants for certain inputs or outputs. There are, for instance, only 10 invariants generated where the input `Activate` is *false*. Similar invariants can be inspected for the state variables `S_i` or output variables like `S_EStopOut`.

4 Related Work

Transferring formal verification methods from theory to practical applications is an active topic, and particularly important for safety-critical systems. This includes, but is not limited to model checking [2] and static program analysis [13, 4]. Additionally, techniques based on exact decision procedures have found application, for instance, the work by Sülflow and Drechsler [14] on SAT-based equivalence checking. More comprehensive overviews of existing approaches for formlization and verification of PLC programs are given elsewhere [15–17].

Invariant generation using program rewriting is a widely appreciated concept in the fundamental research on program verification [18], e.g., in the context of

verifying heap-manipulating programs [19]. However, to the best of our knowledge, the concept of applying rewriting logic was not applied to PLC programs before, where it allows to derive strong invariants due to the limitations of the underlying hardware platform. Often, even simple logics as the one described in this work suffice for deriving expressive invariants.

5 Discussion & Future Work

Invariants are a simple means to represent properties of programs. The logic considered in this paper — in contrast to other logics used for specifying properties such as LTL or CTL — is easy to formulate and understand. Usually, invariants are used in formal verification techniques like model checking to specify properties to be proven. In this paper, we proposed a new approach that automatically derives invariants for all variables of a program, i. e., an over-approximation of properties of the variables is generated automatically. Thus, users do not need to provide properties of a program and use formal verification techniques to prove these properties, which might be error-prone and lengthy. The invariants derived can then be analyzed by users to check whether the program behaves as expected.

Two specifics enabled the automatic generation of invariants for PLC programs and facilitated the scalability of the approach described: the underlying hardware of the PLCs and the usage of SSA form. The underlying hardware of PLCs is simple due to its use in safety-critical systems, well structured, and extensively documented. The use of a SSA form enables to come up with invariants, which are, for example, only valid after the last iteration of a loop during a PLC cycle as in the case of `from_byte` variable. This is especially helpful in PLC programs as the values of variables are only visible after the execution of a complete cycle and not during the cycle.

As shown by our examples, the approach delivers invariants that capture the behavior of programs in a way easily accessible by users. Additionally, the examples show that the approach also scales well for larger programs.

There are several directions for future improvement. First, not all properties can be expressed using invariants and it is therefore useful to investigate whether the same approach can be used to automatically generate properties using a more complex logic. Furthermore, users want to check whether their properties are satisfied by the program. This can be achieved by checking whether their invariants are entailed by the invariants generated by our approach. If the user-provided invariants are satisfied, it could be useful to also provide information about the additional properties which are satisfied by the program but not specified by the users. This process could be supported by a graphical representation of the invariants.

Acknowledgment

The work of Sebastian Biallas was supported by the DFG. The work of Jörg Brauer and Stefan Kowalewski was, in part, supported by the DFG Cluster

of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89.

References

1. International Electrotechnical Commission: IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems. International Electrotechnical Commission, Geneva, Switzerland (1998)
2. Schlich, B., Brauer, J., Wernerus, J., Kowalewski, S.: Direct model checking of PLC programs in IL. In: DCDS. (2009) To appear.
3. Pavlovic, O., Pinger, R., Kollmann, M.: Automated formal verification of PLC programms written in IL. In: VERIFY. Number 259 in Workshop Proce., CEUR-WS.org (2007) 152–163
4. Huuck, R.: Software Verification for Programmable Logic Controllers. Dissertation, University of Kiel, Germany (April 2003)
5. Mertke, T., Frey, G.: Formal verification of PLC-programs generated from signal interpreted petri nets. In: 2001 IEEE International Conference on Systems, Man, and Cybernetics. Volume 4., IEEE Computer Society Press (2001) 2700–2705
6. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. of the ACM **12**(10) 576–585
7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An opensource tool for symbolic model checking. In: CAV (2002). Volume 2404 of LNCS., Springer (2002) 241–268
8. Emerson, E.A.: Temporal and Modal Logics. In: Handbook of Theoretical Computer Science. Volume B. The MIT Press (1991) 995–1072
9. International Electrotechnical Commission: IEC 61131-3 Ed. 1.0: Programmable Controllers — Part 3: Programming languages. International Electrotechnical Commission, Geneva, Switzerland (1993)
10. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Effciently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. (1991) 451–590
11. PLCopen TC5: Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks. PLCopen, Germany (2006)
12. Soliman, D., Frey, G.: Verification and validation of safety applications based on PLCopen Safety Function Blocks using Timed Automata in UPPAAL. In: DCDS. (2009) To appear.
13. Bornot, S., Huuck, R., Lukoschus, B., Lakhnech, Y.: Utilizing static analysis for programmable logic controllers. In: ADPM. (2000) 183–187
14. Sülflow, A., Drechsler, R.: Verification of plc programs using formal proof techniques. In: FORMS/FORMAT, L'Harmattan (2008) 43–50
15. Baresi, L., Mauri, M., Monti, A., Pezze, M.: PLCTools: Design, formal validation, and code generation for programmable logic controllers. In: SMC. (2000) 2437–2442
16. Mertke, T., Menzel, T.: Methods and tools to the verification safety-related control software. In: SMC. (2000) 2455–2457
17. Bani Younis, M., Frey, G.: Formalization of existing PLC programs: A survey. In: CESA. (2003)
18. Nelson, G.: Verifying reachability invariants of linked structures. In: POPL, ACM (1983) 83–47
19. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: POPL, ACM (2008) 171–182