Utilizing Static Analysis for Programmable Logic Controllers*

Sébastien Bornot · Ralf Huuck† · Ben Lukoschus
Lehrstuhl für Softwaretechnologie
Universität Kiel
Preußerstraße 1–9, D-24105 Kiel, Germany
{seb|rhu|bls}@informatik.uni-kiel.de

Yassine Lakhnech
Verimag
Centre Equation
2 av. de Vignate, F-38610 Gières, France
lakhnech@imag.fr

Abstract

Programmable logic controllers (PLCs) occupy a big share in automation control. However, hardly any validation tools for their software are available. In this work we present a lightweight verification technique for PLC programs. In particular, static analysis is applied to programs written in Instruction List, a standardized language commonly used for PLC programming. We illustrate how these programs are annotated automatically by an abstract interpretation algorithm which is guaranteed to terminate and is applicable to large-scale programs. The resulting annotations allow static checking for possible run-time errors and provide information about the program structure, like existence of dead code or infinite loops, which in combination contributes to more reliable PLC systems.

Keywords: programmable logic controllers, instruction list, static analysis, abstract interpretation

1 INTRODUCTION

Programmable logic controllers (PLCs) are increasingly used in many automation projects. While in the past mainly simple tasks had to be performed by PLCs, nowadays, more and more PLCs are taking over extensive controlling aspects. This holds in particular for industries as power generation, steel production, water, chemical and petrochemical [6]. By nature these systems are in general hybrid, i.e., a discrete control interacts with continuous signals which often leads to a complex controlling software. Therefore, it is much more likely that there are flaws or errors with respect to the specification in it. However, safety is a crucial requirement for these systems and, thus, the application of formal methods seems to be a promising way to increase the reliability of PLCs and systems utilizing them.

The IEC 1131-3 standard [5] defines and describes a number of programming languages for PLCs. In this work, we focus on one of these languages called *Instruction List* (IL) which is widely used in the industry. Our efforts are directed towards the development of a verification tool for IL programs which can handle industrial size programs and is also easy to use for non-computer scientists, i.e., people working with PLCs.

Hence, we focus on a "push button" tool where the required user-interaction is minimal. Therefore, the properties to be checked are of generic nature, like division by zero, infinite loops, dead code, whether array boundaries are exceeded etc. Our methodology is based on static analysis techniques, in particular abstract interpretation and and data flow analysis. Static analysis allows to detect possible run-time errors a priori without actually executing or simulating these programs. Of course, in general these properties can only be checked approximatively, but nonetheless provide valuable information to the correctness of a system.

In this work we focus on a crucial part, namely the abstract interpretation of Instruction List programs. This work is organized as follows: In Section 2 we give a brief introduction to the programming language IL as well as to static analysis and abstract interpretation. Subsequently, in Section 3 we present how the abstract interpretation is applied to IL programs and incorporated in our prototype tool. Finally, we conclude with a discussion on this technique and future improvements in Section 4.

2 FRAMEWORK

2.1 The Instruction List Language

One of the reasons in 1993 to install the IEC 1131-3 standard [5] was to unify different programming languages for programmable logic controllers. Basically, five languages were standardized which have similar expressiveness but differ in notation and concept. One of the languages that is used most throughout Europe by PLC vendors is Instruction List.

Instruction List is a low level language similar to assembler code. Basically, a program is a list of instructions where each instruction consists of an operator and one or more operands. These operations use or change the value stored in a single register called *current result* (CR). The current result can be read, modified or stored to a variable. The standard provides a number of operators. These range from addition or comparison to

*This work was partially supported by the European Community in the Esprit Long-Term Research Project 26270 – VHS (Verification of Hybrid Systems) and by the DFG (Deutsche Forschungsgemeinschaft) in the research program "Software Specification" under grant LA 1012/6-1.

†author to whom correspondence should be addressed

Program	CR	X	Y
`VAR`			
`  X,Y : INT;`			
`END_VAR`			
	CR unknown	$X \in [-\infty,+\infty]$	$Y \in [-\infty,+\infty]$
`LD 1`			
	$CR \in [1,1]$	$X \in [-\infty,+\infty]$	$Y \in [-\infty,+\infty]$
`ST X`			
	$CR \in [1,1]$	$X \in [1,1]$	$Y \in [-\infty,+\infty]$
`ST Y`			
	$CR \in [1,1]$	$X \in [1,1]$	$Y \in [1,1]$
`label`			
	CR unknown	$X \in [1,+\infty]$	$Y \in [1,+\infty]$
`LD X`			
	$CR \in [1,+\infty]$	$X \in [1,+\infty]$	$Y \in [1,+\infty]$
`ADD X`			
	$CR \in [2,+\infty]$	$X \in [1,+\infty]$	$Y \in [1,+\infty]$
`ST X`			
	$CR \in [2,+\infty]$	$X \in [2,+\infty]$	$Y \in [1,+\infty]$
`MUL Y`			
	$CR \in [2,+\infty]$	$X \in [2,+\infty]$	$Y \in [1,+\infty]$
`ST Y`			
	$CR \in [2,+\infty]$	$X \in [2,+\infty]$	$Y \in [2,+\infty]$
`LD X`			
	$CR \in [2,+\infty]$	$X \in [2,+\infty]$	$Y \in [2,+\infty]$
`LE 10`			
	$CR = \perp$	$X \in [2,+\infty]$	$Y \in [2,+\infty]$
`JMPC label`			
if jump taken:	$CR = true$	$X \in [2,+\infty]$	$Y \in [2,+\infty]$
if jump not taken:	$CR = false$	$X \in [2,+\infty]$	$Y \in [2,+\infty]$
`LD Y`			
	$CR \in [2,+\infty]$	$X \in [2,+\infty]$	$Y \in [2,+\infty]$
`DIV X`			
	$CR \in [0,+\infty]$	$X \in [2,+\infty]$	$Y \in [2,+\infty]$

Figure 1: An IL program annotated with the abstract interpretation results.

the calls of timer functions or floating point operations. Moreover, various kinds of jump operations to specified labels are available.

An example for an IL program is shown in the leftmost column of Figure 1. The command `LD` loads a value to the CR. `ST` stores the current result to a variable, `ADD` means addition, and `LE` compares the CR to a constant or variable and stores the result, a Boolean value, in the current result. The conditional jump `JMPC` depends on the Boolean value of the CR. If CR equals *true* control will jump to the specified label, otherwise, the next instruction is executed.

2.2 Static Analysis

Static analysis [3] is a technique to identify properties of programs that hold for all executions, called *invariants,* without actually executing these programs. This contrasts to *testing* which generates and checks *some* executions of the system, and *model checking* which builds a complete transition structure and thus checks *all* executions of the system. While testing samples just some behavior of the program, static analysis folds, i.e., over-approximates, the behavior. Hence, the analysis is pessimistic in the sense that it might lead to inconclusive results. However, static analysis is applicable to large-scale systems like industrial size IL programs and can provide valuable information for the system development process.

Static analysis can be used for array bound checking, infinite loop and dead code detection or to reveal divisions by zero, type inconsistencies, redundant code, etc.

2.3 Abstract Interpretation

One static analysis technique often used is *abstract interpretation* [4]. Basically, abstract interpretation assigns *abstract values* of an abstract domain to program variables. Abstract value means that in contrast to one *concrete* value of a variable (e.g., an integer obtained by one execution of the program), we only give an over-approximation for each variable (e.g., by an interval comprising all possible integer values) valid for all executions [3].

The advantage of such a pessimistic model is that an abstract interpretation algorithm can be designed such that it terminates for any given program, whereas simulation or execution of the program can never be guaranteed to terminate. This means, in any case we obtain some result.

The disadvantage of such a pessimistic model is that we usually obtain more possible behaviors than there actually exist in all concrete executions of the program. Depending on the choice of the abstract domain and how sophisticated methods for determining the abstract values are used, this leads to a higher or lower imprecision of the results. However, with a safe interpretation of the abstract values, program properties will never claimed to be satisfied if they actually are not. On the

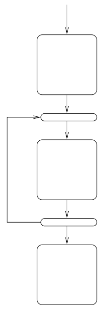

Figure 2: Flow graph of the IL program.

other hand it is possible that properties are interpreted as being unsatisfied although on the concrete level they are.

3 STATIC ANALYSIS OF IL PROGRAMS

In this section we present abstract interpretation on Instruction List programs. The general idea is to determine for each program variable in each line of code an approximation for their value range, i.e., their abstract value. Therefore, by following the control flow, we successively investigate each operation in each line and its effect to the program variables until some fixed point is reached. Since the numbers of repetitions in loops are generally undecidable, they have to be treated in a special way in order to guarantee termination of this algorithm. An acceleration technique called *widening* is used.

The algorithm we use here is adapted from [3]. For the sake of simplicity we do not give the full algorithm, but illustrate its effect by an example.

First of all, let us consider the IL program of Figure 1. In this example we have integer and Boolean variables. The integer values are abstracted by intervals including those with limits $+\infty$ and $-\infty$ and the Booleans are abstracted by elements of $\{true, false, \bot\}$, where $\bot$ denotes the fact that the concrete value is unknown. There are also special considerations due to the fact that `CR` is dynamically typed and can be of several types during an execution. This is the reason for the 'unknown' value at some places in the example.

The algorithm works as follows: initially, the abstract value for each occurrence of a variable is unknown. Following the control flow we successively determine their possible, i.e., abstract values. Consider the variable `X`. Before the first instruction (`LD 1`) we do not know anything about the possible values of `X`, hence, we assign the abstract value $[-\infty, +\infty]$ to it, which denotes its value can be anything. The first instruction does not provide further information on `X`, but the second one (`ST X`) assigns the value one to `X`. Hence, it can precisely be abstracted by the interval $[1,1]$.

The algorithm determines statically for each line of code the new values of the program variables depending on the previous line or set of previous lines if the current line is a junction node in the flow graph (cf. Figure 2) and the previous abstract values.

By following the control flow one might encounter loops like the one in the example between `label` and `JMPC label`. Generally, it is a priori undecidable how many times the loop will be taken, but we can observe that variables will increase. By merging the abstract values we know that at the junction (`label`), `X` and `Y` will be in an interval of the form $[1,a]$ where a is a value which will grow each time the loop is taken. To enforce termination of the algorithm, we use a technique called "widening" to determine in finite time a safe approximation of the abstract values corresponding to an infinite number of loop executions. In general widening is an over-approximation in order to terminate some iterating process. In the above case, we set the upper limit of the growing variables `X` and `Y` to $+\infty$ by choosing $[1,+\infty]$ as abstract value.

Finally, we can deduce from the abstract value of `X` that `X` will never be equal to 0 at the division (`DIV X`) operation.

This example shows that our abstraction is very rough: at the end we are not able to ascertain that `X` is greater than 10 which is determined by the conditional jump. This is due to the fact we cannot connect informations given by conditional jumps to variables, except the current result `CR`.

Thus, a specific part of our algorithm concerns `CR` (see Figure 3). We connect it to the other variables by representing it as an expression involving constants and variables. After each step, the expression of `CR` is updated. If it is an operation, we change the expression according to this operation. If it is a load (`LD`) or store (`ST`) instruction, we know that the value of `CR` is the value of the corresponding variable. Moreover, for each `ST` instruction, before updating `CR` we update the corresponding variable.

Its abstract value can easily be computed by replacing occurrences of variables by their abstract values and performing the operations on these. In the example (see Figure 3), after `MUL Y`, we know that $\mathtt{CR} = \mathtt{X} \cdot \mathtt{Y}$, $\mathtt{X} \in [2,20]$ and $\mathtt{Y} \in [1,+\infty]$, hence, $\mathtt{CR} \in [2,+\infty]$.

For conditional jumps, we can infer the value of `CR` by observing if the jump is taken or not. This leads to constraints on variables. In the example, if the jump is taken, then `CR` equals *true*, and thus, $\mathtt{X} \leq 10$, and since $\mathtt{X} \in [2,20]$, we know that $\mathtt{X} \in [2,10]$. When the expression involves too many operations or variables, this additional information might not be useful for constraining abstract values. But this is rarely the case since Instruction List programs often use few operations between `LD` and `ST`.

In the example, the conditional jump was taken only when `X` was less or equal to 10. Since it is this branch which caused the widening, we can safely apply this information after the widening and thus get the interval $[2,10]$ as abstract value for `X`.

```
VAR
  X,Y : INT;
END_VAR
```

Instruction	CR	X	Y
	CR unknown	$X \in [-\infty, +\infty]$	$Y \in [-\infty, +\infty]$
`LD 1`	$CR = 1$	$X \in [-\infty, +\infty]$	$Y \in [-\infty, +\infty]$
`ST X`	$CR = X$	$X \in [1, 1]$	$Y \in [-\infty, +\infty]$
`ST Y`	$CR = Y$	$X \in [1, 1]$	$Y \in [1, 1]$
`label`	CR unknown	$X \in [1, 10]$	$Y \in [1, +\infty]$
`LD X`	$CR = X$	$X \in [1, 10]$	$Y \in [1, +\infty]$
`ADD X`	$CR = X + X$	$X \in [1, 10]$	$Y \in [1, +\infty]$
`ST X`	$CR = X$	$X \in [2, 20]$	$Y \in [1, +\infty]$
`MUL Y`	$CR = X \cdot Y$	$X \in [2, 20]$	$Y \in [1, +\infty]$
`ST Y`	$CR = Y$	$X \in [2, 20]$	$Y \in [2, +\infty]$
`LD X`	$CR = X$	$X \in [2, 20]$	$Y \in [2, +\infty]$
`LE 10`	$CR = (X \leq 10)$	$X \in [2, 20]$	$Y \in [2, +\infty]$
`JMPC label` if jump taken:	$CR = \mathit{true}$	$X \in [2, 10]$	$Y \in [2, +\infty]$
if jump not taken:	$CR = \mathit{false}$	$X \in [11, 20]$	$Y \in [2, +\infty]$
`LD Y`	$CR = Y$	$X \in [11, 20]$	$Y \in [2, +\infty]$
`DIV X`	$CR = Y/X$	$X \in [11, 20]$	$Y \in [2, +\infty]$

Figure 3: Abstract interpretation results using expressions for CR.

Our prototype tool is currently able to perform this abstract interpretation on IL programs, with the restriction that we do not yet deduce anything from the fact that conditional jumps are taken or not. Despite of the fact that we have not yet implemented the full range of algorithms performing static analysis, we can gather enough information from abstract interpretation to identify certain properties. For example, we can find conditional jumps which either always or never jump, leading to dead code or infinite loops. Moreover, the tool performs types checking of the variables and the CR register.

4 Discussion

In this paper we presented an approach for abstract interpretation on Instruction List programs in order to perform static analysis. The abstract interpretation can be performed automatically by our prototype tool to check some properties (e.g., division by zero). The advantage of this approach in comparison to other formal techniques, e.g., model checking [9, 2], is that it allows easier handling of large-scale programs. However, this results partly from over-approximating and, hence, loss of information.

Therefore, future work consists of two parts. First, we will extend the power of the tool by completing the abstract interpretation algorithm and adding other properties to be checked. Moreover, the efficiency of different abstract domains as well different abstraction techniques have to be investigated.

Second, we plan to incorporate IL into the PLC structuring language called Sequential Function Charts (SFC). These are hierarchical automata whose transitions are labeled by guards (conditions defining enabledness of transitions) and whose states are labeled by actions to be performed. These actions can be other SFCs as well as programs, e.g., IL. We want to analyze SFCs where all actions which are not SFCs itself are IL programs. From the structure of the SFC and the analysis of previous programs, we will find more precise approximations of the variables and thus more precise diagnostics. Moreover, in [1] we present an automatic way to verify properties for SFCs using the model checking tool SMV [7, 8]. Our aim is thus to combine the different techniques and languages: we will use abstract interpretation to help model checking, since we need abstraction anyway to cope with systems of practical size.

References

[1] Sébastien Bornot, Ralf Huuck, Yassine Lakhnech, and Ben Lukoschus. Verification of sequential function charts using SMV. Accepted for the special session "Formal Verification and Formal Methodologies in the Industrial Validation Flow" of PDPTA 2000, Las Vegas, USA, June 2000.

[2] E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM symposium of Programming Languages*. ACM Press, 1983.

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming, Paris, France, April 13–15, 1976*, pages 106–130, Paris, France, 1976. Dunod.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California*, pages 238–252, New York, NY, 1977. ACM Press.

[5] International Electrotechnical Commission, Technical Committee No. 65. *Programmable Controllers – Programming Languages, IEC 61131-3*, second edition, November 1998. Committee draft.

[6] R.W. Lewis. *Programming industrial control systems using IEC 1131-3*, volume 50 of *Control Engineering Series*. The Institution of Electrical Engineers, revised edition, 1998.

[7] K.L. McMillan. *The SMV system*. Carnegie-Mellon University, February 1992. Draft manual describing SMV revision 2.2.

[8] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[9] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.