Static Analysis of Industrial Controller Code using ARCADE.PLC

Sebastian Biallas Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
{lastname}@embedded.rwth-aachen.de

Stefan Stattelmann Bastian Schlich
ABB Corporate Research
Research Area Software
Ladenburg, Germany
{firstname}.{lastname}@de.abb.com

Abstract

In this paper, we present the static analysis capabilities of ARCADE.PLC: A tool to check software for Programmable Logic Controllers, which are frequently used in industry to control or monitor technical processes. These devices are usually programmed using domain specific programming languages and unorthodox programming practices such as a huge number of variables and large function size. This imposes a challenge to standard static analysis techniques.

1. Introduction

Programmable logic controllers (PLCs) are control devices used in industry to control, monitor, and operate technical processes [4]. They typically comprise a set of inputs (connected to sensors), outputs (connected to actuators) and a program operating on internal memory. In the most common mode of operation, the program is called periodically at a high frequency, computing new outputs based on the current inputs and internal memory. ARCADE.PLC[1] is a framework for the verification and analysis of PLC code [1], combining model-checking and static analysis.

The static analysis of ARCADE.PLC implements typical checks for runtime errors and code smells, such as variables with constant values, missing case labels in switch statements, unreachable code, conditions with constant result or constant subexpressions, division by zero, and illegal access into arrays or structured data types. These checks can be applied to a whole controller program (regarding all inputs as non-deterministic) or to a single function of the program.

This paper presents the challenges and results of adapting the static analysis capabilities of ARCADE.PLC to industrial PLC code from the ABB Compact Control Builder development environment.

2. Technical and Pragmatic Challenges

2.1 A Zoo of Languages

The IEC 61131-3 standard defines five languages [4, Part 3], which includes textual languages such as Instruction List (IL) and Structured Text (ST), graphical languages such as Function Block Diagram (FBD) and Ladder Diagram (LD), and the state based language Sequential Function Chart (SFC). Typically, PLCs are programmed in a mixture of these languages, e.g., by writing function blocks in ST, which are then combined graphically in FBD. Although standardized in principle, each PLC vendor usually implements a slightly different dialect of these languages, sometimes even for different products of the same vendor. For instance, ABB Compact Control Builder offers a graphical language which adds meaning to the order in which function blocks are arranged in the user interface. To cope with the different languages and dialects, ARCADE.PLC translates all input programs into an intermediate code, on which the abstract interpretation is performed. Possible warnings are then mapped back to the original program location.

[1] http://arcade.embedded.rwth-aachen.de/

2.2 Challenges in Accessing Real-World Code

The ABB Compact Control Builder tool uses encryption to protect some parts of the code. While all function block libraries are distributed as source code and compiled to native code internally, the libraries are protected by encryption to avoid modification of the source code by the users of a library. This protection was introduced since modification by control engineers let to problems when different versions of a library were used for control applications that were relying on "unofficial" patches.

The missing information in encrypted libraries gives rise to a technical challenge: since libraries can be completely encrypted, even the signatures of the function blocks in some libraries are not available for an external tool. Thus, the static analysis engine must derive an appropriate signature for types from encrypted libraries based on the way they are used in the unencrypted parts of the source code. This creates a significant amount of overhead when testing static analyses during analysis development. As it was often unclear whether certain analysis warnings were triggered by flaws in the analysis or simply by missing information, a lot of manual inspection was necessary.

2.3 Size of the Programs

In principle, a controller program is one big function. It is not uncommon for such a program to have thousands of lines of code, accessing thousand global variables per cycle, and hundreds of function blocks which themselves have variables for the internal state. The sheer size thwarts a naïve flow-sensitive analysis because of memory constraints.

Additionally, the way function block calls are handled introduces even more variables: There are basically two ways to call a function block in most PLC languages: In the first version, input and output parameters are passed directly (akin to most regular programming languages). Another, semantically equivalent way to call is to access the input and output parameters outside the call:

```
functionblock.input1 := 1;
functionblock.input2 := a;
functionblock();
result := functionblock.output;
```

When implementing a context-sensitive analysis of the function block calls, this syntax entails that all input and output parameters of the function block have to be tracked at the caller's side, introducing even more variables. A context-sensitive analysis is advised for

most simple function blocks from the standard library, such as timers, edge detection, flip-flops, etc. We use a liveness-based pre-analysis [2] to infer which variables values are relevant in which parts of the program. This techniques nicely handles the huge number of variables and allows a context-sensitive analysis of certain function blocks.

2.4 Abstract Domains

PLC programs usually operate on simple data types and data structures only. Typically, the following operations should be precisely reflected by a static analysis to produce good results:

- Simple integer and float arithmetic
- Bitwise logical operations and tests
- Conversions between bitwise and integer types
- Small sets of discrete values (i. e., enumerations)

To handle these operations in our static analysis, we use a partially reduced product [3] of the following domains: We use the integer and float interval domain to handle arithmetic. A bit-wise interval domain is used to reflect bit-wise logic operations. Finally, we use k-set domain (sets with at most k distinct values otherwise $\top$). In our experience, this setup is sufficient to capture most typical PLC program operations.

2.5 Identifying Useful Analyses

We had to tune our checks in practice to account for different coding styles. The following code fragment, e. g., shows a pattern which we frequently encountered during our case study:

```
42 If CONDITION1 Then
43    OUTPUT := 65535;
44 ElsIf CONDITION2 Then
45    OUTPUT := INPUT1 And (INPUT2 Or (INPUT3 Xor 65535));
46 ElsIf Not CONDITION2 Then
47    OUTPUT := INPUT1 And (INPUT2 Or (INPUT3 Xor 0));
48 End_If;
```

Checking the condition in line 46 is obviously superfluous as the condition in line 46 is the negation of the condition checked in line 44. Arcade.PLC thus correctly reported that the condition in line 46 yields a constant result. However, since essentially every else-statement in the projects we analyzed were written in this way, this resulted in a larger number of reported warnings, which were not real problems in the code. Hence, we had to deactivate this warning to make the number of warnings manageable.

2.6 Specific Checks

Compact Control Builder programs can make use of the firmware functions `GetStructComponent` and `PutStructComponent`. They allow for accessing the n-th component of a structured data type. If n is less than 1 or greater than the number of elements in the struct, a runtime error is signaled. It is additionally detected if the accessed element is of the wrong type. To allow for offline checking of correct usage of these functions, we implemented two new checks into Arcade.PLC.

The first check infers the bounds for the index expression used in the respective calls. If there are no structure elements for some of the possible values, we issue a warning that the structure might be accessed outside of its bounds. Additionally, we check that all structure elements in this interval have the same type. The first check is only an adaption of an array index out of bounds check to these firmware functions. The second check, however, is a domain-specific analysis which is able to detect runtime errors statically, but it only makes sense in the context of Compact Control Builder programs.

3. Current Results & Future Work

During our case study using a real-world project from an industrial plant containing about 50,000 lines of code, all checks could be executed in about 10 minutes. After some fine tuning, the number of warnings was reasonably low. The warnings could thus be easily inspected manually. Some of the remaining false positives are caused by the absence of relational information in our domain. We plan to add (weakly) relational domains in the future.

What we learned is that not every analysis which looks useful in theory can fulfill this promise in practice. On the other hand, looking at real-world code can inspire new analyses and triggers the need to optimize existing analysis techniques. We therefore believe that applying analysis tools on large real-world projects helps tremendously in improving static analysis tools. Whenever possible, information about the application domain should be considered. This includes considering the end user of an analysis tool. An ideal static analysis should also be useful for someone who does not understand the underlying theories.

Acknowledgments

This work was supported, in part, by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* and by the DFG *Cluster of Excellence on Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89. Further, the work of Sebastian Biallas was supported by the DFG.

References

[1] S. Biallas, J. Brauer, and S. Kowalewski. Arcade.PLC: A verification platform for programmable logic controllers. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 338–341. ACM, 2012. ISBN 978-1-4503-1204-2.

[2] S. Biallas, S. Kowalewski, S. Stattelmann, and B. Schlich. Efficient handling of states in abstract interpretation of industrial programmable logic controller code. In *Proceedings of the 12th International Workshop on Discrete Event Systems*, Cachan, France, 2014. IFAC. To appear.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[4] International Electrotechnical Commission (IEC). IEC 61131-3 – Programmable Controllers—Part 3: Programming languages. 2003.