Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application

Herbert Prähofer, *Member, IEEE*, Florian Angerer, Rudolf Ramler, *Member, IEEE*, and Friedrich Grillenberger

***Abstract*—Static code analysis techniques examine programs without actually executing them. The main benefits lie in improving software quality by detecting problematic code constructs and potential defects in early development stages. Today, static code analysis is a widely used quality assurance technique and numerous tools are available for established programming languages like C/C++, Java, or C#. However, in the domain of programmable logic controller (PLC) programming, static code analysis tools are still rare, although many properties of PLC programming languages are beneficial for static analysis techniques. Therefore, an approach and tool for static code analysis of IEC 61131-3 programs has been developed which is capable of detecting a range of issues commonly occurring in PLC programming. The approach employs different analysis methods, like pattern-matching on program structures, control flow and data flow analyses, and, especially, call graph and pointer analysis techniques. Based on results from an initial analysis project, where common issues for static analysis of PLC programs have been investigated, this paper illustrates adoption and extensions of analysis techniques for PLC programs and presents results from large-scale industrial application.**

***Index Terms*—Programmable logic controllers, software quality, software maintenance, program analysis, static code analysis.**

I. Introduction

INDUSTRIAL automation systems are nowadays based on complex and large-scale software [1]. With this increased complexity, however, it becomes necessary to make use of advanced software engineering tools and techniques for productivity and quality improvement as well as cost reduction. For example, Vyatkin [2] reviews current software engineering research in the automation domain but also points out important challenges to be addressed in the future. In [3] and [4], a summary of essential software engineering challenges in the automation domain is given.

One promising software engineering technique is static code analysis which allows acquiring information about the structure but also possibly about runtime properties of a program without executing it [5]. Its main benefit lies in improving the quality of code in early development stages by providing means to reveal inadequate code constructs (bad code smells), violations of programming guidelines, and potential defects [6]. It complements program inspection, software testing, as well as program verification techniques. Compared to those techniques, it has several advantages: 1) in distinction to program inspection techniques, static code analysis techniques work fully automatically; 2) in distinction to software testing methods, it can be employed without elaborate test settings; 3) in distinction to program verification techniques, it does not require system specifications but works with rules identifying patterns indicating possible problems. On the other side, static analysis techniques only provide hints which usually must be examined by experts.

Numerous techniques and tools are available for general purpose programming languages like C/C++, Java, C# as well as for many others [7]. Moreover, several empirical studies confirm its valuable contribution to software quality [8], [9]. Compared to conventional software engineering practice, there is a huge gap in the application of static analysis techniques for programmable logic controllers (PLCs) based on the IEC 61131-3 standard. This is surprising since the properties of PLC programs—like the lack of dynamic memory allocation—are highly amenable to advanced program analysis techniques.

This paper addresses the existing gap and presents a comprehensive approach for static analysis of PLC programs based on the IEC 61131-3 standard. A set of program analysis methods have been developed for PLC programs based on the results from analysis of common issues in PLC programming, possible solution concepts, as well as characteristics of IEC 61131-3 languages. The methods have been implemented in a tool environment which currently supports structured text (ST) and sequential function chart (SFC) programs. The tool adopts a

Manuscript received November 20, 2015; revised February 15, 2016, March 31, 2016, and May 20, 2016; accepted August 4, 2016. Date of publication August 31, 2016; date of current version February 6, 2017. The research reported in this article has been partly supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH. Paper no. TII-15-1708.R3. *(Corresponding author: H. Prähofer.)*

H. Prähofer is with the Institute for System Software, Johannes Kepler University Linz, Linz 4040, Austria (e-mail: herbert.praehofer@jku.at).

F. Angerer is with Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems, Johannes Kepler University Linz, Linz 4040, Austria (e-mail: florian.angerer@jku.at).

R. Ramler is with the Software Competence Center Hagenberg, Hagenberg 4232, Austria (e-mail: rudolf.ramler@scch.at).

F. Grillenberger is with ENGEL Austria GmbH, Schwertberg 4311, Austria (e-mail: friedrich.grillenberger@engel.at).

Digital Object Identifier 10.1109/TII.2016.2604760

rule-based program analysis approach with extendable rule sets and is thus adaptable to different application domains and coding guidelines. Further, the applicability of the methods and the tool support have been evaluated in a large-scale industrial application.

This paper is structured as follows. After shortly reviewing program analysis techniques and its adoption for PLC programming, it 1) presents results from a problem analysis project to identify common programming issues in PLC programs together with possible solution methods; 2) describes the analysis methods and the tool implementation; and 3) presents experiences from large-scale industrial application in the partner company ENGEL Austria GmbH (www.engel.at).

The approach has first been published in [10] and [11]. While those former publications concentrated mainly on analysis methods, this paper gives a detailed description of the analysis structures, the rule framework and experiences from large-scale industrial application. The main contributions of this work are as follows.

1) Identification of common issues occurring in PLC programming together with possible solution methods.
2) A comprehensive tool environment which allows parsing ST and SFC code and computation of advanced analysis structures.
3) A rule framework which allows adapting and extending analyses to specific application domains and coding conventions.
4) Empirical results from large-scale industrial application.

II. Static Code Analysis Reviewed

A. Static Code Analysis Methods

Static code analysis works by examining the program elements, the program structure, or by approximating the behavior of the program. The techniques employed are manifold (see, for example, [5], [12], [13]). Many issues can be tackled by just examining the program elements at the level of an abstract syntax tree (AST) representation of a program (AST analysis). The analysis often adopts a pattern-matching approach where rules for AST patterns are defined and the analysis is accomplished by traversing the AST and checking the visited structures against defined rules.

Control flow and data flow analyses are techniques originally developed in context of compiler technology [12]. Control flow analysis works by determining the paths a procedure can take, which are represented as a control flow graph (CFG). Data flow analysis is about analyzing where data come from, where data go to, and how data are manipulated. Typical examples of data flow analysis methods are reaching definitions, live variables and available expressions.

Control flow and data flow analyses are intraprocedural techniques. In distinction, the call graph (CG) is a representation of all call dependences globally in a program. To compute a CG, one starts with the main routine and recursively follows all calls. This is straightforward for procedural languages; however, when it comes to object-oriented languages with object instances and dynamically bound method calls, the analysis becomes more difficult. First, one needs to know the object the method call is applied to, which results in an object-sensitive CG. Second, the method called is dependent on the dynamic type of the object. Thus, with dynamic object allocation and dynamic class loading like in Java, the exact computation of an object-sensitive CG is often infeasible.

A further important analysis technique is pointer analysis [14]. It computes all locations a reference or pointer variable can possibly point to and is a prerequisite for analyzing possible memory access operations [15] (the terms pointer variable and reference variable will be used interchangeably in the sequel). This information is represented as the so-called points-to sets (PSs) for each pointer variable. Several different types of pointer analysis methods are distinguished which differ in runtime complexity and precision of the PSs. For example, an object-insensitive method only computes one PS for a pointer variable declared in a type, whereas an object-sensitive method computes PSs for all individual object variables in each object instance. Finally, abstract interpretation [16] is a technique where the execution of a program is approximated.

Static code analysis tools and techniques may produce false positives (false alarms) since they are often based on methods using approximations. Several methods sacrifice precision for performance and accept overapproximation [17]. Consequently, applying static code analysis techniques means that an expert has to examine the results indicated in order to determine whether implied problems really prevail.

B. Static Code Analysis of PLC Programs

Numerous open source [9], [18] and commercial [19] methods and tools for static analysis are available for popular programming languages. Compared to this immense number applications, static analysis for PLC programming is still very rare and tool support started to emerge only recently [20]. Among the few examples, there is the professional edition of the CoDeSys development environment (www.codesys.com) providing analysis methods with about 50 configurable rules. Another one is PLC Checker by Itris (www.itris.fr). This tool helps to ensure quality by automatically validating coding rules and supports four different PLC dialects. The tool logi.LINT by Logicals (www.logicals.com) has a focus on graphical languages for PLC programming. Finally, OAC (www.oacg.co.uk) offers an IEC 61131-3 programming suite including the so-called integrity analyzer, which checks compliance to coding standards.

The available commercial static analysis tools for PLC applications are primarily implementing basic pattern-based AST analysis techniques. Only some tools such as PLC Checker also use control flow analysis techniques. To the best of the authors' knowledge, none of the tools supports global CG and pointer analysis, even though the use of references is common in PLC programs and global CG and pointer analysis seem very promising for PLC programs.

Besides the basic support provided by commercial tools, research has been exploring the use of advanced techniques for static code analysis of PLC programs. For example, the work of Nair [21] presents an approach for deriving software metrics from PLC programs. Arcade.PLC [22] is a research project that focuses on model checking and abstract interpretation for PLC programs. The available research aims at specific techniques and issues, whereas the work presented in this paper provides a comprehensive approach that is based on advanced analysis structures and is adaptable to the application area at hand by a configurable rule framework.

III. Problem Analysis

The development of the static analysis tool has been motivated by a problem analysis project where common issues and practical problems in PLC programming have been investigated together with possible solution methods. This section presents the results of this problem analysis project.

A. Common Issues

For identifying common issues in PLC programming, the programming conventions and guidelines and the defect database from the involved industry company have been analyzed. The issues that have been identified can be related to the following categories (cf. also [23]).

1) Code metrics.
2) Naming conventions.
3) Program complexity and possible performance problems.
4) Bad code smells.
5) Architectural issues.
6) Incompatible configuration settings.
7) Multitasking problems.
8) Dynamic statement dependences.

In the following, the different issue categories are discussed.

1) Code Metrics: Widely used metrics computed from source code, e.g., the total number of lines of code (LOC) of a project, LOC of individual program organization units (POUs), or nesting depth of statements, are of interest when analyzing programs [21].

2) Naming Conventions: The programming guidelines of the industry partner include a set of naming conventions for improving code readability. For example, different types of variables should start with corresponding prefixes to clearly indicate their type. Similar conventions exist for parameters, constants, types, and function block (FB) definitions.

3) Program Complexity and Possible Performance Problems: The performance of PLC programs, in particular, the problem of real-time execution constraints, is an important issue. Performance issues are usually difficult to find by static analysis techniques and intensive research effort has been invested in advanced techniques to make worst-case execution predictions for real-time systems, see, e.g., [24]. Nevertheless, classical static analysis techniques allow giving approximate hints about possible performance problems. For example, the guidelines by the industry partner define upper limits for the complexity of certain program elements.

4) Bad Code Smells: From more than ten years of developing PLC programs, the engineers at the involved industry partner have identified a remarkable set of statement constructs and expressions which are known to cause problems. They are often called bad code smells as they are symptoms that indicate an underlying problem. Typical examples of bad code smells are empty branches in conditional statements, loop variables that are modified within the body of the loop, or variables that are set multiple times in sequence without being read.

5) Architectural Issues: Similar to bad code smells, there are various constructs at the level of the program structure which are considered bad practice. For example, POU instances which are never called represent an unnecessary waste of memory, some variables declared in a POU might be better declared as TEMP variables, or a FB which only use TEMP variables would be better declared as a function. Other issues concern the execution of specific operations by tasks with defined priorities.

6) Incompatible Configuration Settings: Besides the actual program code, a PLC program consists of configuration settings which define, e.g., the connection of variables to hardware endpoints or mappings of variables to visualization elements. Such dependences are usually not checked by the compiler but can be checked by static analysis.

7) Multitasking Problems: The analysis of the defect database revealed many defects that were caused by multitasking problems. Some of them led to severe system failures that only occurred in field and which are extremely hard to find and reproduce in test settings. The root causes of such problems were typically subtle task interleaving and race conditions. For example, a problem which occurred multiple times was due to a tricky interference of settings of start and stop flags from different tasks as follows: a cyclically executed FB uses start and stop events to start and stop a process. When started, it first resets the stop flag. The flags are set based on events from a different task. The problem occurs when the start event is immediately followed by a stop event and before the cyclic task was able to execute. Then, in its next cycle, the process will start and reset the stop flag and the process is not stopped. Although the company knows of such problems and defines guidelines for their avoidance, they are difficult to check without tool support. Other multitasking issues observed were due to the fixed-priority preemptive task scheduling strategy of the runtime system used by the industry partner. To avoid data races, the company pursues the strategy that shared data are only written by tasks with same priorities, a constraint which is difficult to check in complex systems.

8) Dynamic Statement Dependences: The analyzed programming guidelines define constraints on the execution order of related function calls. A simple example is that a file open must be followed by a file close operation. However, there are many more dynamic dependences, which are specific to the analyzed system.

B. Solution Methods

The issues discussed earlier require different analysis methods and Table I relates the different issues to possible solutions methods as follows.

1) Code metrics can be obtained by traversing the AST representation of the program.
2) Naming conventions can be checked by traversing the AST, locating program elements for which naming conventions are defined, and then checking the name of the program element.

TABLE I
ANALYSIS SOLUTIONS FOR DETECTING ISSUES OF DIFFERENT ISSUE CATEGORIES

	(1) Code metrics	(2) Naming conventions	(3) Program complexity	(4) Bad code smells	(5) Architectural issues	(6) Incompatible configurations	(7) Multitasking problems	(8) Statement dependences
AST analysis	X	X	X	X	X	X	X	X
Control flow analysis			X	X				X
Data flow analysis				X	X			
CG analysis					X		X	X
Pointer analysis					X		X	

3) Program complexity is determined by examining specific program elements, e.g., the number of statements, or the CFG of POUs.
4) Bad code smells can be found by looking for specific patterns in the AST. Some issues, however, require examining the control flow or the data flow of a program.
5) Architectural issues usually require a system-wide CG and the analysis of pointers and references. For example, to check if a hardware endpoint is written by a high-priority task requires that all the targets of write operations are known, which requires PSs for pointer variables.
6) Static analysis of configuration settings requires that also the configuration files are parsed and represented in the AST. Then, such configuration dependences can be checked based on the extended AST representation of the program.
7) Static code analysis techniques can detect task interleaving which might be problematic. Most of these situations, however, require the employment of advanced CG and pointer analysis techniques.
8) The detection of violations of dynamic statement dependences can be accomplished by investigating the control flow and CG of a program.

C. Properties of IEC 61131-3 Languages

Several properties specific to the languages defined in IEC 61131-3 standard affect the applicability of static analysis techniques, either positively or negatively. These properties make the analysis also significantly different to other languages. In the following, the important properties and their implications on the analysis techniques are discussed.

1) No Dynamic Memory Allocation: The IEC 61131-3 standard does not define dynamic memory allocations but all the instances are allocated by variable declarations, which can be found statically in the program. Thus, all the instances of FBs can be determined statically by examining the source code. This also applies for the new 2013 version of the IEC 61131-3 standard, where object-oriented features have been introduced but object allocations are still by variable declarations only. Thus, PLC programs allow computing object-sensitive CGs and PSs, which is decisive for many static code analysis tasks.

2) Polymorphism and Dynamic Binding: Without the object-oriented features in the new standard, the IEC 61131-3 standard does not support polymorphism and dynamic binding. Therefore, for a call to a FB, it is statically known which code will actually be executed. With the object-oriented features of the 2013 standard, polymorphic POU reference variables and dynamic binding of methods are supported. However, due to the statically known instances and with a pointer analysis, it is still possible to statically determine the types of the referenced instances and thus the set of dynamically called methods.

3) References: The IEC 61131-3 standard allows references and defines an address-of and a dereference operator. Thus, a reference variable can point to different memory locations. This complicates analyzing access operations to shared data. As discussed above, an object-sensitive pointer analysis is facilitated by static memory allocations. However, the flexibility in using references and call-by-reference parameter passing complicates pointer analysis.

4) Procedure-Valued Variables: There also exists the concept of references to POU instances. When a call is done using a reference variable, the called instance cannot be determined statically by the call expression but is dependent on the reference contained in the variable. Moreover, in the 2013 standard, such reference variables can be polymorphic. Hence, a pointer analysis for procedure-valued variables is essential.

5) Asynchronous Events: Besides calling a POU by a simple call, POUs can also be activated by events. Such activation dependences have to be considered in the CG. Therefore, asynchronous events further complicate the computation of the CG of a program.

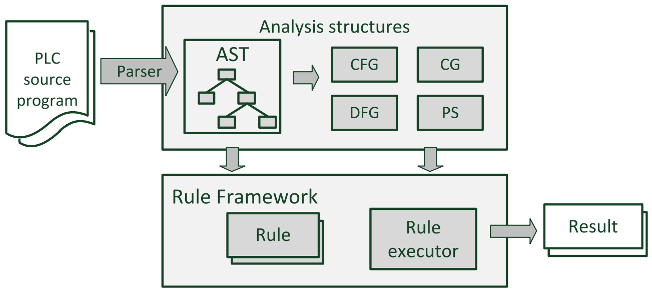

Fig. 1. Overview of the analysis tool.

IV. APPROACH

Based on the identified issues and the related analysis methods described earlier, a tool environment has been developed. Fig. 1 illustrates the overall structure of the approach and the tool support. Input is the source code of the PLC program, which

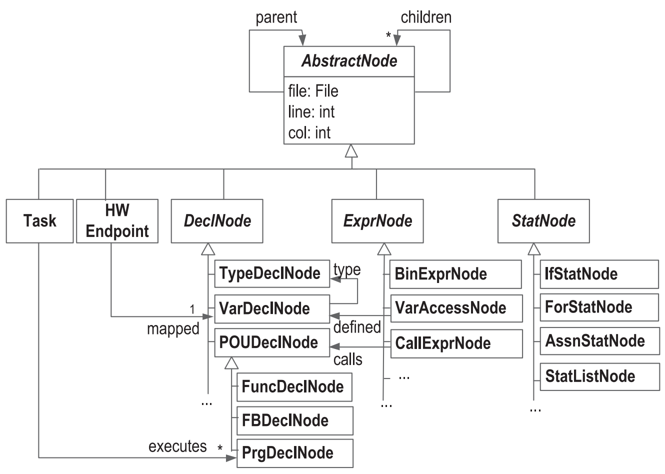

Fig. 2. AST class hierarchy (excerpt).

is translated to an extended AST representation. Based on the AST, advanced analysis structures are computed.

1) CFGs.
2) Data flow graphs (DFGs).
3) The CG.
4) The PSs for all the reference and pointer variables.

The AST and the computed analysis structures are input to a rule framework which is configurable with a set of rules where each rule is programmed to search for particular, possibly problematic program structures. The analysis results are the elements detected to cause rule violations. In the following, the approach and tool implementation are discussed.

A. Abstract Syntax Tree Representation

The central data structure for the analysis is an AST representation of the program, which is created by a parser from source code. The parser has been developed using a compiler–compiler tool and is based on the grammar of the specific IEC 61131-3 language dialect. The AST is a tree of nodes where each program element is represented by a node of a specific type. There are nodes for POU definitions, type declarations, variable declarations, statements and expressions, etc. The AST also includes other program elements like task definitions plus assignments of programs to tasks or hardware endpoints and their mappings to program variables. After creating the tree of nodes, links between related nodes are created in a resolving step, e.g., links are created from a variable access node to the variable declaration, from a call to the POU declaration, from a variable declaration to the type definition, etc.

The AST is implemented by a Java class hierarchy as depicted in Fig. 2. It has specialized node types for the different elements of a program. An abstract base class AbstractNode represents the root and defines properties which are common to all node types: file, line, and col store the source code position of the program element, parent and children are used for navigation. Derived from AbstractNode are special nodes for the different program elements: POU and type declarations, variable declarations, statements and expressions, and all the other nodes for

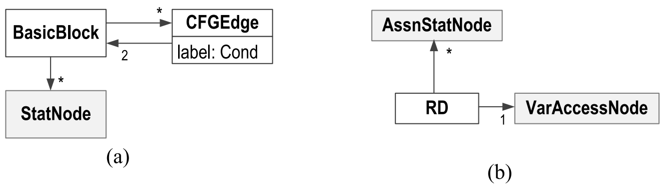

Fig. 3. Classes for CFG and DFG (nodes from AST in gray). (a) CFG, (b) DFG.

program elements. The nodes are interlinked as created in the resolving step. Together, approximately 100 classes are defined for representing ST and SFC code plus various configuration settings. The nodes, their types and properties serve as the primary input for the rule framework (cf. Section IV-D).

B. CFG and DFG

CFG and DFG are computed based on the AST using standard methods [5], [12]. The CFG represents the possible paths through a POU body. It is based on basic blocks where a basic block is a sequence of statements which are executed unconditionally. That means, the CFG of a procedure is a graph

$$CFG = \langle B, E \rangle$$

with the basic blocks $b \in B$ forming the nodes and $E \subseteq B \times B$ are the edges and there is an edge $(b_1, b_2) \in E$ from b_1 to b_2 if execution of b_2 can follow execution of b_1. In the tool, the CFGs are represented by the classes BasicBlock and CFGEdge as shown in Fig. 3(a). Note that BasicBlocks reference a list of statement nodes from the AST.

The DFG represents dependences between variable assignments to variable usages and therefore it represents reaching definitions information [5]. It is a mapping of variable access expressions to a set of assignment statements. Let $R(v)$ be all expressions reading a variable v and $A(v)$ be all assignments to a variable v. Then, reaching definition $RD(v)$ for a variable v is a mapping

$$RD(v) : R(v) \rightarrow 2^{A(v)}$$

which is the subset of assignments to v that possibly can define the values of the variable when reading it at an access expression $r \in R(v)$. Class RD implements the DFG in the tool [see Fig. 3(b)]. It encodes which assignments represented by AssnStateNodes can reach which variable usages represented by VarAccessNodes.

C. CG and PSs

As described in Section III, CG and pointer analyses are critical issues for PLC programs as they heavily rely on reference and pointer variables. As PLC programs do not use dynamic memory allocations, it is possible to perform object-sensitive CG and pointer analyses. The prerequisite for object-sensitive analysis, however, is a static computation and an explicit representation of all the instances in a program, which is called the instance model of a program.

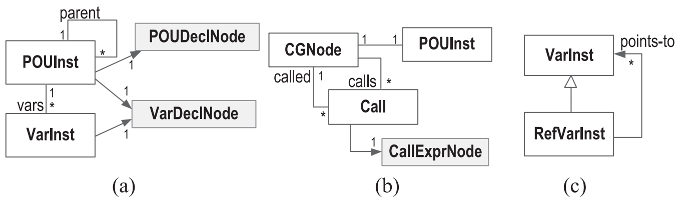

Fig. 4. Classes for instance model, CG and PSs. (a) Instance model, (b) Call graph, (c) Points-to sets.

1) Instance Model: In PLC programs based on the IEC 61131-3 standard, FBs are allocated by variable declarations and therefore all the FB instances can be determined from source code. We therefore call a variable declaration an *allocation site*. Thus, the POU instances are organized as a tree where the programs represent the roots, their children are the FB instances declared in variable declarations, which again have children according to their variable declarations. Additionally, an artificial instance *epsilon* is used for the root of the instance tree and which serves as the implicit parent of all instances not allocated explicitly, i.e., programs, functions, and global variables.

The set of instances I are formally defined as follows. Let S be the set of allocation sites, i.e., the set of all variable declarations (plus the implicitly allocated singletons like programs and globals). Then, an instance is identified recursively by an allocation site and a parent instance or by a vector $(s_1, s_2, \ldots, s_n)$, $1 \leq n < \infty$, with $s_1, s_2, \ldots, s_n$ being allocation sites. Let $S^i = \times_{j=1}^{i} S$ be allocation vectors of length i, then the instance space I is defined by

$$I \subseteq U_{i=1}^{\infty} S^i.$$

Fig. 4(a) shows the class diagram to implement the instance model in the analysis tool. All the POU instances are explicitly represented by objects of type POUInst and are linked by parent–children relations. Additionally, the variables declared in POU instances are represented by objects of type VarInst. POUInst as well as VarInst have references to their corresponding variable declaration nodes from the AST (VarDeclNode) and POUInst also to the node representing the POU type declaration (POUDeclNode).

2) CG: The object-sensitive CG is based on the instance model and defined as follows. Let $P \subseteq I$ be the POU instances. Then, the CG is defined as

$$C = (P, E)$$

with $E \subseteq P \times P$ where an element $(p_1, p_2) \in E$ means that there is a call of POU instance p_2 in POU instance p_1.

The implementation of the CG is shown in Fig. 4(b). CGNode objects directly correspond to the POU instances and contain the call relations represented by Call objects, which reference the AST node of the call statement. Computation of a CG from an AST representation works by starting from the programs directly executed by tasks, then following calls to POUs and add respective edges to the CG. However, the IEC 61131-3 standard also allows references to POUs and, when a call is performed using a reference variable, one has to consider all the POU instances the reference variable can point to, which is provided by a pointer analysis.

3) PSs: The pointer analysis determines the set of memory locations a reference or pointer variable can possibly point to. Let $R \subseteq I$ be allocation sites of reference and pointer variables. Then the points-to function

$$pts : R \rightarrow 2^I$$

gives for each reference or pointer variable r the set of instances $pts(r)$ this variable can possibly point to. Fig. 4(c) shows the implementation of PSs. There is a special class RefVarInst representing variable instances of reference or pointer variables. Such nodes can store a set of VarInst nodes representing the PS of that variable.

PSs are computed based on subset constraints between PSs of pointer variables which are extracted from pointer assignment statements. Based on results from [25], a fixpoint algorithm for computing CG and PSs has been presented in [11].

```
@ApplyFor( NodeType.class)
class RuleClass extends BaseClass<NodeType> { ...
    public void apply(NodeType node) { ... }
}
```

Listing 1. Basic pattern for rule class implementation.

D. Rule Framework

The rule framework uses the AST and analysis structures presented above to search for issues in PLC programs. It consists of a rule executor and a set of rules, where rules encode checks for issues. This section illustrates how rule execution works, how customized rules can be implemented, and provides some examples of rules.

Rule execution works by traversing the AST and selecting rules applicable to specific AST nodes. For example, when visiting a node representing an if-statement, the rule executor will apply all rules which are specified to be applicable to IfStatNode nodes. When applied, a rule retrieves relevant information from the node and further analysis structures, uses this information to check for issues, and generates a result message, which is then collected by the rule executor. Thus, rule execution is fulfilled fully transparent to the user. The responsibility of the rule developer is to implement rules by Java classes. In this way, the set of rules can be adapted and extended to meet specific application domains, programming models, and coding guidelines.

Rule implementation is backed by base classes, utility methods, and design patterns. A rule is only applicable to a specific node, and a rule class has to identify the corresponding node. Then, a rule checks the program element for issues. The approach for implementing rules is similar to writing JUnit test cases. Listing 1 shows the principle pattern how rules are implemented. Rule classes must be derived from a generic base class which is parametrized by the node type. There are different base classes, depending on the type of the rule (see below). Applicability of a rule is determined by the type of the node which

```
@ ApplyFor(BinaryExprNode.class)
class PgrmRuleFloatingPoint extends LocalRule<BinaryExprNode> {
   public void apply(BinaryExprNode n) {
      assertFalse( isReal(n.left()) && isReal(n.right()) && n.kind() == EQ,
         n, "Equality comparison of floating-point types done!");
   }
}
```

Listing 2. Local rule checking equality comparison of reals.

```
01 @ApplyFor( ForStatNode.class )
02 class LoopVarUsedAfterForRule extends NonLocalRule<ForStatNode> {
03    public void apply(ForStatNode forLoop) {
04      final VarDeclNode loopVar = forLoop.getLoopVar().getDecl();
05      assertNotExists(
06        getFollowers(forLoop, VarExprNode.class),
07        new Check<VarExprNode>() {
08          public boolean check(final VarExprNode varAccess) {
09             return varAccess.getDecl() == loopVar;
10          }
11        },
12        forLoop, "Access to loop variable after for-loop");
13    }
14 }
```

Listing 3. Nonlocal rule checking use of loop variable after for.

```
@ApplyFor( ProjectNode.class )
class ConcurrentWrites extends AccumulatingRule<ProjectNode> {
   Map<VarDeclNode, Set<Task>> globalWrites;
   public void apply(ProjectNode prjct) {
     globalWrites = new HashMap<VarDeclNode, Set<Task>>();
   }
   @Accumulate( AssnStatNode.class )
   public void handleAssignment(AssnStatNode assn) {
      Set<VarDeclNode> vars;
      if (isPointer(assn.getVar())) vars = getPointsTo(assn.getVar());
      else vars = Collections.singleton(assn.getVar());
      for (VarDeclNode v: vars) {
        if (isGlobal(v)) {
           if (globalWrites.get(v) == null)
              globalWrites.put(v, new HashSet<>());
           globalWrites.get(v).addAll(assn.getTasks());
        }
      }
   }
   public void evaluate(ProjectNode prjct) {
     for (VarDeclNode v: globalWrites.getKeys()) {
        int priority = globalWrites.get(v).iterator().next().getPriority();
        for (Task task: globalWrites.get(v)) {
          assertTrue(task.getPriority() == priority, task,
            String.format("Write %s with different priorities", v.getName());
        }
     }
   }
}
```

Listing 4. Accumulating rule for write operations to shared data.

is specified by the annotation ApplyFor defining the node type. Rule application means calling the apply method which takes the visited node as parameter. The apply method will, in its simplest form, check some condition on the node and, if not satisfied, will issue a message string which is transformed by the rule executor to the final result message.

Three types of rules are distinguished: local rules, nonlocal rules, and accumulating rules, which differ in how rule conditions are checked and how information about the program is gathered. The rule types correspond to base classes LocalRule, NonLocalRule, and AccumulatingRule.

Local rules only use local information of the visited AST node. Listing 2 shows an example which checks that real numbers are not compared for equality, as such comparisons are considered unreliable. The annotation @ApplyFor specifies that the rule is applicable to nodes of type BinExprNodes. The apply method checks if the left and right operands are real number expressions and an equality operator is used. This is checked in an assertFalse statement, one of the utility methods provided by the rule framework. It allows checking a condition and, when failed, will issue a violation message.

Nonlocal rules can also check additional analysis structures related to the visited node. For that purpose, a set of utility methods are provided for traversing and investigating analysis structures. For example, utility method getFollowers allows iterating over all statements following a given statement, an information obtained from the CFG. Moreover, nonlocal rules can use assertExists or assertNotExists methods, which will test that a certain condition is satisfied for at least one/none of the nodes in a given search space. With these features, nonlocal rules have equal expressiveness as logical formulae with quantifiers.

Listing 3 gives an example of a nonlocal rule. According to the coding conventions, a loop variable of a for-loop should only be used within the for-loop and accessing the variable afterwards is discouraged. The rule class in Listing 3 implements a respective check. In the apply method, the declaration of the loop variable is retrieved at line 4. Then, an assertNotExist method call is used to verify that there is no access of the loop variable after the for-statement. The assertNotExist method uses a search space which contains all nodes representing variable access expressions following the for-loop. Thus, getFollowers(forLoop, VarExprNode.class) provides an iterator over all VarExprNodes following node forLoop. The check is implemented by the inner class Check which checks if the variable accessed and the loop variable are the same.

Accumulating rules can be used for checks which have to collect information over a larger part of the program. That means accumulating rules will accumulate information about a program upon traversing a subtree of the AST. The subtree is determined by the AST node this rule applies to. After traversing the subtree and accumulating information, it will evaluate the information and compute results. Thus, accumulating rules work in three steps: initialize, accumulate, and evaluate.

The example in Listing 4 shows an accumulating rule that checks that all write operations to a global variable are performed from tasks with same priorities, which is known to possibly cause data races in the used fixed-priority preemptive scheduling system otherwise. The rule accumulates information from write operations globally in a program (applies to ProjectNode) and also considers points-to information. It uses a map globalWrites to store for each global variable the tasks where write operations happen. When the ProjectNode node is visited (which is at the very beginning of the AST traversal), the apply method is called to do the initialization. Then, the accumulation of the information about write operations is done in method handleAssignments. This method is marked by annotation Accumulate(AssnStatNode.class) which notifies the rule executor that it should call this method for all AssnStatNode nodes. In this method, the variable of the assignment is checked. First, it is tested if it is a pointer variable (line 10). If so, all the variables in the PS of this pointer variable are handled as possible targets of the assignment; otherwise, only the variable itself is considered. When the target variable is global, the tasks of the current statement are entered into the globalWrites table. As a final step and after all data have been collected, the evaluate method is called by the rule executor to evaluate the data collected and to report any issues.

E. Tool Implementation

The described methods have been implemented in a tool environment as outlined in Fig. 1. The analyzed source code files comprise IEC 61131-3 programs in the languages ST and SFCs plus proprietary extensions from the language implementer KEBA AG (www.keba.com). The tool itself is written in Java. Furthermore, the analysis tool has been integrated into SonarQube (www.sonarqube.org), which is an open platform for managing code quality in software development projects. SonarQube provides a Web-based front-end for visualization and analysis of software quality data from various sources. SonarQube is used for presenting the analysis results as part of a quality dashboard showing key metrics and trend measures. Furthermore, starting from the dashboard view, SonarQube allows drilling down to violations to support code inspection and defect removal. For example, considering issues detected by ConcurrentWrites rule from Listing 4, the user gets a list of global variables which are written by tasks with different priorities together with all the program positions where such writes happen. From those, he is able to directly jump to the respective code position in the editor.

As outlined in Section II, static analysis methods are approximate methods and can produce false positives. That means, an issue may be reported by the tool but an expert may decide that it is not relevant and can safely be ignored. Therefore, an important feature of the tool is a suppress mechanism similar to Java's @Suppress annotation. Placing a comment $$Suppress(rule1, rule2, ...) before the respective program element means that rules rule1, rule2, ... are ignored and violations will not appear in subsequent analysis runs.

V. Industrial Application and Evaluation

This section presents results from a performance evaluation and experiences from using the tool for analysis of a large-scale industrial system. The case study system is a software platform for automation of injection moulding machines. It is the basis for development of the control software for all the different machine series of the company. The platform contains a huge set of components for different functions for the core system but also for optional and alternative equipment. Altogether it comprises about 800k of LOC written in ST and SFC languages. The platform code represents the basis for deriving automation solutions for specific machines by first selecting and configuring components and then extending and adapting the solution to meet the specific requirements at hand.

TABLE II
Program Size and Tool Performance Data

Project	LOC	# POUs	# AST nodes	# CG nodes	# points-to	Parsing [ms]	Analysis [ms]
1	148 k	7141	434 k	19 k	5304	3338	1703
2	235 k	14 112	732 k	46 k	9644	8765	4431
3	295 k	23 946	882 k	76 k	16 776	12 204	7727
4	345 k	23 797	1049 k	79 k	14 847	14 527	8392
5	623 k	58 096	1704 k	179 k	48 199	46 235	39 321

Injection moulding machines are machines for producing plastic parts by injecting heated plastic material into a mold. It consists of a hydraulic system with one or more pumps for providing pressure for performing all the movements on the machine, a heating system for providing heated plastic material, a mold which can be opened and closed, an injection system for injecting the plastic into the mold, and an ejector. The core machining cycle therefore is closing the mold, injecting the heated material, opening the mold, and ejecting the part. Core components in the platform are a parameterizable and configurable machining cycle, a controller for the pump providing the required pressure in the hydraulic system, a low-level control system for controlling valves, an error and alarm system, functions for supervision and maintenance, and a huge number of software components for handling special equipment, e.g., using additional sensors or actuators or interfacing with material handling systems.

A. Performance Evaluation

Table II shows performance results from the application of the tool for five solutions of different sizes, which are derived from the software platform. The performance data have been obtained by running the tool on a PC equipped with a Core i7 820 CPU with 2.93 GHz and 8 GB DDR3-SDRAM and using Java 7 HotSpot 64-Bit Server VM. The values are the average over 4 runs, where 6 runs have been performed in total and the best and the worst results have been discarded.

Columns 2 and 3 of Table II show the size of the programs in terms of LOC and the number of POU instances. LOC is count of lines with executable code, excluding comments and empty lines. Then, the number of nodes of the resulting AST is given. The next columns show the size of the resulting CG and the size of the PSs in total number of elements for all sets. The last two columns show the time for parsing the programs and computing the analysis structures and performing the rule-based analysis, respectively. Program sizes range from 148 kLOC (7141 POU instances) up to 623 kLOC (58 096 POU instances). The parsing time for the largest program is about 46 s and the analysis time about 40 s. Thus, the runtime evaluation proves the practical applicability of analysis methods to industry-size programs.

B. Rule Set

We have implemented a set of 46 rules for checking the source code for issues. The rules are classified into categories analogous to the identfied categories of issues as outlined in Section III.

TABLE III
RULE CATEGORIES AND NUMBER OF RULES

#	Rule category	# Rules	# Local rules	# Nonlocal rules	# Acc rules
1	Metrics	1	0	0	1
2	Naming	5	5	0	0
3	Complexity	3	2	0	1
4	Bad code smells	17	8	5	4
5	Configurations	2	2	0	0
6	Multitasking	7	2	0	5
7	Architecture	8	0	1	7
8	Statement dep.	3	0	2	1
Total		46	19	8	19

Table III shows the number of rules in each category and also how many of them are local, nonlocal or accumulating rules. The majority of the rules checks for bad code smells and architectural issues. Only a few rules are available for multitasking issues; however, they are rather complex as they require CG and points-to information.

In the following, we discuss the implemented rules for the different categories. Note that the rules implement the conventions and guidelines of the partner company, which are derived from their experience, programming practice, and architectural considerations.

1) *Metrics.* Computation of metrics is not accomplished by a rule but computed by traversing the AST together with rule execution. The following metrics are computed: LOC, LOC for each POU, cyclomatic complexity, and comment lines.
2) *Naming.* There are different simple rules for checking naming conventions for POUs, different types of variables, and type declarations, as defined by conventions of the company.
3) *Complexity.* Complexity constraints are defined for the maximum number of LOC in a POU triggered by an event, the maximum nesting in a POU body, and the complexity of expressions.
4) *Bad code smells.* Rules checking for bad code smells are manifold. There are several simple rules which, e.g., check that branches of conditional statements are not empty, that certain operators are used properly (cf. Listing 2), that end-conditions of loops are simple, or that FB calls never miss input parameters (which is allowed in the IEC 61131-3 standard but considered bad practice in this company). More complex rules check for unreachable code, assignments which are never read, that variables are always initialized before being read, or the usage of loop variables outside for loops (cf. Listing 3).
5) *Configurations.* Configuration rules check if deployment configuration is consistent. For example, there is one rule which checks that variables which are input from the visualization systems are also declared RETAIN.
6) *Multitasking.* The multitasking rules investigate what is called by different tasks and in this way checks for possibly harmful situations. For example, there is a rule which checks that an FB instance is never called from different tasks because this might result in race conditions on the FB internal variables. The runtime system of the industry partner is based on a fixed-priority preemptive scheduling strategy which might cause data races when data elements are written by tasks with different priorities. Thus, a rule checks that global variables are never written by code executed by tasks with different priorities (cf. Listing 4). A further rule is provided to check for the problematic use of start/stop flags as outlined in Section III-A.
7) *Architecture.* The eight architectural rules check issues like that all POU instances are really used, that events cannot result in event loops, or that POU variables are of the proper kind, e.g., if VAR is used where a VAR_TEMP would be sufficient.
8) *Statement dependences.* Finally, rules in this category check implicit dependences between statements. For example, a call to open a file must be followed by a call to close the file.

C. Experiences From Industrial Application

The analysis tool has been integrated in the company's development process for quality assessment and improvement. Static code analysis is conducted on several different product configurations as part of the nightly build. Based on these products, more than 2/3 of the platform code (about 500 k LOC) is currently checked with static code analysis. The main reason why currently only a part of the code base is checked is that introducing static analysis for so far unchecked code represents a notable effort.

In the following, the experiences from using the analysis tool for the platform code are reported. The development team maintained detailed records about applying the tool for two iterations with the goal to get quantitative data about the effectiveness and precision of rules and about the time it takes to check and resolve reported issues. The first iteration took place in the incubation phase of the tool. At that time, no CG and PSs and only an incomplete rule set were available. In the second iteration, also CG and PSs and a refined and extended rule set were used.

A summary of the results is given in Table IV, which shows the number of violations reported by the tool, the number of violations resolved, the number of violations suppressed, the overall time it took to handle the violations, and the average time it took to resolve an issue. These numbers are shown for the individual rule categories and separately for the two iterations. In the first iteration, 150 components with a total of 120 kLOC have been processed, in the second iteration 39 components and 30 kLOC.

In the first iteration, the tool reported 549 violations from which 354 have been resolved and 195 have been suppressed. This took 48.4 h in total, which is about 5 min per violation. The most effective rules were the rules for bad code smells and for naming conventions. Only few issue regarding configurations have been detected, and none for complexity issues and statement dependences. Rules for checking task scheduling issues have not been available in the first period. A lot of violations were reported regarding the architecture, but most of them

TABLE IV
EXPERIENCES FROM CHECKING AND RESOLVING ISSUES IN TWO ITERATIONS OF APPLYING THE ANALYSIS TOOL

Rule category	# Violations	# Resolved	# Suppressed	Time [h]	Time/viol [min]	Iteration
Naming	148	141	7	13.5	5.5	1
	26	26	0	4.5	10.4	2
Complexity	0	0	0	0	–	1
	0	0	0	0	–	2
Bad code smell	207	191	16	19.5	5.7	1
	46	46	0	6.7	8.7	2
Configurations	7	7	0	0.7	6.0	1
	3	3	0	0.4	8.0	2
Task scheduling	–	–	–	–	–	1
	2	2	0	0.2	6.0	2
Architecture	187	15	172	14.7	4.7	1
	13	12	1	1.1	5.1	2
Statement dep.	0	0	0	0	–	1
	0	0	0	0	–	2
Total	**549**	**354**	**195**	**48.4**	**5.3**	1
	90	**89**	**1**	**12.9**	**8.6**	2

were identified as false positives, which triggered a revision and refinement of rules.

Analysis in iteration 2 has been conducted with a refined and extended set of rules, now also containing rules for checking task scheduling issues. With those rules, two critical multitasking issues have been detected. The refined rules for architectural issues produced 13 violations, of which all except one were resolved. The invested overall time was 12.9 h, which is about 8 min per issue.

D. User Feedback

The code analysis tool underwent a thorough evaluation throughout its industrial application and it is now used in daily practice since about 3 years. This section summarizes the feedback of the practitioners concerning the overall value of static code analysis and the usefulness of the various rules.

The primary goal of using static analysis is to improve the software quality before the software is given to the test team. The use of static analysis has been found to improve the quality of the software and the invested effort is reasonable compared to the total development effort. For example, a new component has been developed within a customer project recently and then checked for issues before integration into the platform code. Development time for the component was about one man-year but resolving the 150 issues detected by static analysis took only one week. Summing up, static analysis has increased the development team's confidence in the quality of the software because it eliminates certain issues and the whole development process became more agile and costs could be reduced.

Further, the head of the development team evaluated the rules regarding the severity of the violations detected, the accuracy of the rule, and the risk of introducing new problems when making changes to resolve the detected violations. The value contribution of individual rules shows a diverse picture, with some rules being considered more important than others. For example, rules concerning configuration dependences proved to be very useful because they are simple and accurate and problems caused by incorrect dependences are often not found in testing. Another example for rules regarded as highly valuable are those for detecting multitasking issues and race conditions. Usually, such problems lead to sporadic defects in the field and are extremely difficult to find. However, static analysis only gives hints and the resolution still requires a detailed analysis by an expert. Furthermore, fixes affecting multitasking code include a high risk of introducing other problems. Moreover, the development team has meanwhile extended the rule set by itself. Writing simple rules, for example, naming conventions or configurations was found very easy, but, e.g., writing a rule for checking for endless loops in SFCs represented a considerable effort.

A further important feedback from the industry partner concerned the suppress mechanism. Recall that static code analysis is an imprecise method and an additional manual analysis by an expert is required to exclude violations that are actually false alarms. In this context, the suppress mechanism has been found to be an absolute necessity which makes sure that indicated violations will not show up again.

VI. CONCLUSION AND OUTLOOK

This paper presented an approach and tool for static code analysis of PLC programs written in IEC 61131-3 languages. Motivated by the observation that static analysis methods and tool are scarce for PLC programming and that static analysis methods and tools have been very successful in other domains, common coding issues and possible solution methods for PLC programs have been investigated. Then, methods and tool support for static analysis of PLC programs have been developed. The tool comprises a parser for building an AST representation of a program, a resolving step for interlinking AST nodes, methods for creating advanced analysis structures like CFG, DFG, CG and PSs, and a rule framework. The rule framework works with extensible and adaptable rule sets, which are implemented by Java classes. Main challenges in developing the tool were coping with the magnitude of language extensions in the IEC 61131-3 dialect at hand, coping with the partially unclear language semantics, and considering all the cases of references passing when doing the pointer analysis. On the other hand, the static memory layout proved to be particularly helpful. A performance evaluation of the methods and tools showed their applicability for large industrial projects. The tool has been applied in large-scale industrial settings and is meanwhile integrated in the standard software development process of the involved industry partner. The application has shown the effectiveness of the tool support in terms of issues detected as well as the productivity gained.

The methods and tools described in this paper have initially been developed for a special dialect of the IEC 61131-3 standard and covered ST and SFC code. Current work includes porting the tool for working for the CoDeSys Version 2 (www.codesys.com) and supporting other languages, e.g., IL code. Further, there are more opportunities for static analysis of PLC programs. Our future work will thus pursue promising application areas, in particular, in software architecture analysis, evaluation and refactoring.

References

[1] V. Vyatkin, "Guest editorial: Special section on software engineering in industrial automation," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2337–2339, Nov. 2013.

[2] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Trans. Ind. Informat.*, vol. 9, no. 3, pp. 1234–1249, Aug. 2013

[3] B. Vogel-Heuser *et al.*, "Challenges for software engineering in automation," *J. Softw. Eng. Appl.*, vol. 7, no. 5, pp. 440–451, 2014.

[4] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *J. Syst. Softw.*, vol. 110, pp. 54–84, 2015.

[5] H. R. Nielson, F. Nielson, and C. Hankin, *Principles of Program Analysis*. New York, NY, USA: Springer-Verlag, 1999.

[6] P. Louridas, "Static code analysis," *IEEE Softw.*, vol. 23, no. 4, pp. 58–61, Jul./Aug. 2006.

[7] É. Payet and F. Spoto, "Static analysis of android programs," *Inf. Softw. Technol.*, vol. 54, no. 11, pp. 1192–1201, 2012.

[8] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 240–253, Apr. 2006.

[9] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Experiences using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, Sep./Oct. 2008.

[10] H. Prähofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, "Opportunities and challenges of static code analysis of IEC 61131-3 programs," in *Proc. 2012 IEEE 17th Conf. Emerg. Technol. Factory Autom.*, Krakow, Poland, 2012, pp. 1–8.

[11] F. Angerer, H. Prähofer, R. Ramler, and F. Grillenberger, "Points-to analysis of IEC 61131-3 programs: Implementation and application," in *Proc. 2013 IEEE 18th Conf. Emerg. Technol. Factory Autom.*, 2013, Cagliari, Italy, pp. 1–8 .

[12] S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA, USA: Morgan Kaufmann, 1997.

[13] B. Schlich, J. Brauer, and S. Kowalewski, "Application of static analyses for state space reduction to microcontroller binary code," *Sci. Comput. Program.*, vol. 76, no. 2, pp. 100–118, 2011.

[14] M. Y. Hung, P. S. Chen, Y. S. Hwang, R. D. C. Ju, and J. K. Lee, "Support of probabilistic pointer analysis in the SSA form," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2366–2379, Dec. 2012.

[15] H. Prähofer, R. Schatz, C. Wirth, and H. Mössenböck, "A comprehensive solution for deterministic replay debugging of SoftPLC applications," *IEEE Trans. Ind. Informat.*, vol. 7, no. 4, pp. 641–651, Nov. 2011.

[16] P. Cousot and R. Cousot, "Abstract interpretation and application to logic programs," *J. Logic Program.*, vol. 13, no. 2, pp. 103–179, 1992.

[17] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Inf. Softw. Technol.*, vol. 53, no. 4, pp. 363–387, 2011.

[18] B. Chelf and C. Ebert, "Ensuring the integrity of embedded software with static code analysis," *IEEE Softw.*, vol. 26, no. 3, pp. 96–99, May/Jun. 2009.

[19] A. Bessey *et al.*, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[20] A. Dubey, "Evaluating software engineering methods in the context of automation applications," in *Proc. 2011 IEEE Int. Conf. Ind. Informat.*, 2011, pp. 585–590 .

[21] A. Nair, "Product metrics for IEC 61131-3 languages," in *Proc. 2012 IEEE Int. Conf. Emerg. Technol. Factory Autom.*, Krakow, Poland, 2012, pp. 1–8.

[22] S. Stattelmann, S. Biallas, B. Schlich, and S. Kowalewski, "Applying static code analysis on industrial controller code," in *Proc. 2014 Int. Conf. Emerg. Technol. Factory Autom.*, Barcelona, Spain, 2014, pp. 1–4.

[23] K. Duschl, D. Gramß, M. Obermeier, and B. Vogel-Heuser, "Towards a taxonomy of errors in PLC programming," *Cogn. Technol. Work*, vol. 17, no. 3, pp. 417–430, 2015.

[24] R. Wilhelm *et al.*, "The worst-case execution-time problem—Overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 36–53, 2008.

[25] L. Li, C. Cifuentes, and N. Keynes, "Boosting the performance of flow-sensitive points-to analysis using value flow," in *Proc. Eur. Conf. Found. Softw. Eng.*, 2011, pp. 343–353.

Herbert Prähofer (M'11) received the M.Sc. and Doctoral degrees from Johannes Kepler University Linz, Linz, Austria, in 1986 and 1991, respectively, both in computer science.

He currently holds a position of an Associate Professor in the Institute for System Software, Johannes Kepler University Linz. His research interests include software development methods and tools, model-based software development, and their application in automation.

Florian Angerer received the M.Sc. degree in computer science from Johannes Kepler University Linz, Linz, Austria, in 2012, where he is currently working toward the Ph.D. degree in Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems.

His research interests are on program maintenance, configuration-aware program analysis and variability.

Rudolf Ramler (M'15) received the M.Sc. degree in business informatics from Johannes Kepler University Linz, Linz, Austria.

He is a Senior Researcher in the Software Competence Center Hagenberg, Hagenberg, Austria. He has more than 15 years of experience in software engineering research and technology transfer. His main research interests include software testing, software analytics, and software quality management.

Friedrich Grillenberger has received a university degree in Data Technology from the Johannes Kepler University Linz, Linz, Austria in 1997. He currently works for ENGEL Austria GmbH, Schwertberg, Austria, a global leader in injection molding machine manufacturing. As group leader, he is responsible for development of the control platform solution within the company. His main concern is the continuous improvement of the software quality and development processes.