A generic static analysis framework for domain specific languages

Avijit Mandal
ABB Corporate Research
Bangalore, India
avijit.mandal@in.abb.com

Devina Mohan
ABB Corporate Research
Bangalore, India
devina.mohan@in.abb.com

Raoul Jetley
ABB Corporate Research
Bangalore, India
raoul.jetley@in.abb.com

Sreeja Nair[†]
ABB Corporate Research
Bangalore, India
sreeja.in@gmail.com

***Abstract*—Software used to monitor and control operations within an automation system is defined using domain-specific languages. Latent errors in the control code, if left undetected, can lead to unexpected system failures compromising the safety and the security of automation system. Traditional analysis techniques are insufficient to detect such errors for control system software as they do not cater specifically to the underlying domain-specific language. However, given the diversity of different automation domains, there is no standard platform for analysis of these languages.**

This paper proposes a generic static analysis framework for domain-specific languages, based on abstract interpretation. The analysis approach exhaustively detects runtime errors in control code and ensures compliance to good programming practices. Runtime errors and coding violations are checked against abstract syntax trees and control flow graphs derived from the code. Data Flow Analysis (DFA) and pattern based matching techniques are used to identify domain specific errors and coding violations for control languages.

A proof-of-concept prototype is implemented for three different domain-specific languages IEC 61131-3, used for PLC programming, EDDL a programming language used for configuration of digital devices and RAPID a programing language used for industrial robots.

***Index Terms*—Static analysis, Data Flow Analysis (DFA), Worklist algorithm, Generic programming errors, Safety requirements, Interrupt, Exception, Inter-procedural CFG.**

I. Introduction

Automation engineering systems rely on software to monitor and control various operations like batch processing, arc welding etc. This software, however, is prone to errors that may creep in during development. Such errors, if left undetected, can manifest themselves as failures and or exploited by malicious intruders. Moreover, fixing such errors at later stages of development or after deployment entails high maintenance cost and requires extra effort.

Automation systems use domain specific languages for monitoring and control. Due to the use of complex data structures, task-based parallel execution and unique semantics for execution order, analysis techniques for general-purpose programming languages are not always adequate for these systems. Thus, we need specialized tools and techniques that address the domain-specific languages.

One solution to detect errors in automation systems is to use static code analysis to identify potential sources of errors during compile time. The technique involves examining all possible execution paths in the source code. Static code analysis for general-purpose programming languages is widespread across the software industry. Many static analyzers are widely adopted, e.g., Coverity for C, PolySpace for C/C++/Ada etc. However, such tools are rarely used for automation systems. Typically, verifications tools for control software focus on pattern based matching to detect code violations. Such tools are platform dependent and do not port well across various developing environments.

The work presented in this paper proposes a generic static analysis framework for industrial software. The framework can be used to efficiently perform DFA and pattern-matching based checks on industrial software. The framework can be extended to various programming languages addressing different industrial domains. The main objective is to build a verification platform to ensure correctness of safety-critical software. The paper focuses on the adaptation of the proposed framework to real-world control applications. To this effect, the framework has been extended for analysis of the following languages:

- IEC 61131-3 languages for PLC programming [4]
- Electronic Device Description Language (EDDL), used for configuration of field devices [6], and
- RAPID, a domain-specific language used for programming industrial robots [5]

Key contributions in this paper are,

- Generic datatype to represent the parsed information for the three languages
- Flexible DFA engine to encode more data flow rules as needed by varying the domain
- Flexible rule engine to process data for further analysis

The rest of the paper is structured as follows: Section II explores existing related work. Section III introduces preliminaries. Section IV explains the framework of the prototype. Section V demonstrates about implementation and extensions

[†]Sreeja Nair is currently at Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France

for domain-specific languages. Section VI presents analyses of real-world applications and results thereof. Finally, Section VII summarizes the paper and lists future directions.

II. Related work

There are number of static analysis tools available for general-purpose languages, both as commercial products and in the open source domain. E.g, Polyspace Bug Finder [18] detects run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software using abstract interpretation. Tools like Klocwork [12], CodeSonar [35], CLANG static analyzer [11], Coverity [13], Astree [21] find potential runtime errors for safety critical systems. Visual Expert [19] is a PL/SQL code analysis tool. Pylint [20] is a source code, bug and quality checker for the python programming language. LDRA testbed [14] is analysis and testing tool suite for object oriented programming language such as Java. Pixy [24] addresses static analysis for PHP scripts.

However in automation engineering, the use of static analysis tools are not as consistent. For industrial automation systems, Stattelmann, et al [22] described a static analysis approach for control software of industrial process automation systems. Prahofer et al. [23] give another static analysis approach for IEC 61131-3 programs based on abstract syntax tree structures and methods that use control flow graphs, data flow graphs, call graphs and value dependency graphs. Another approach for static analysis of programmable logic controllers is described by Bornot et. al. [25]. State-of-the-art verification tools for control systems [29], [28] are based on pattern matching techniques to detect violations of coding standards. The work done in verification of control code [31] uses an intermediate representation for model-checking. Pavlov et al. [36] discuss an approach converting PLC programs written in IL to models of the NuSMV model checker. An approach in Bauer et. al. [32] uses model checking to prove properties expressed in temporal logics on PLC programs. In robotics, Sunada WH et al. [26] present an approach of analysis for complex behaviour of robots. Je Huang et al. [27] describes an approach for run-time verification of safety properties for robots running on robot operating system (ROS). All these tools/methods are generally platform-specific.

Cross-platform (multi-language) static analyzers like Facebook Infer [34] mostly address general-purpose languages such as Java, C, C++, and Objective-C, checking for null pointer problems, memory leaks and concurrency issues. Other tools that support multiple language are LDRA testbed [14], Klocwork [12] and Coverity [13].

The available state-of-the-art cross-platform tools are for generic programming language and no commercial-grade cross-platform framework for static analysis of automation engineering/control languages. With this in mind, we developed a cross-platform framework for automation engineering. This improves the state-of-the art for static analysis tools. The framework checks for code violation and conformance to coding standards.

III. Preliminaries

IEC 61131-3 [2] is a family of languages used for process automation of industrial equipments. It supports five different programming languages, namely, Function Block Diagram, Structured Text, Instruction List, Ladder Diagram, and Sequential Function Chart. Controllers are programmed using a high-level language. Such languages can be used to automate a circuit of an oil and gas application as well as mining application in an industrial setting. The execution order for programs is determined by data connections and positions of individual blocks that can be run in parallel, synchronously or asynchronously. In addition, these languages have unique semantics, such as support for multiple input and output ports, that require special handling. Applications are composed of program units. Program units contain code. Variables can be primitive as well as structured data-types.

Electronic device description language (EDDL) [6] is a state of the art method for configuration of digital devices. EDDL is a text-based device description technology that illustrates device parameters, user interactions and all parameters contained in device like calibration settings, flow speed etc. With the help of EDDL, device manufactures explains their devices to the system. It conforms to IEC 61804-3 standards. Since EDDL is a text file, it provides cross-platform compatibility, easy integration, avoid device version conflicts and simpler maintenance in the long-term. It is derived from ANSI C and useful for any fieldbus protocol like FOUNDATION, HART etc. A typical EDDL program includes interactive methods corresponding to specific property of the device. It allows nesting among domain specific data structures that can be extended or redefined to add additional semantics to the properties.

RAPID [3] is a high-level language for ABB industrial robots. It is a domain-specific language that specifies motion control and tasks for each robot. It supports concurrency, synchronization, message-passing and communication. It is a hierarchical language where a robot application is modularized into tasks. The tasks are split into modules and modules are composed of procedures and different kinds of routine. These routines contain some code. It is a mixture of general purpose programming languages that includes loops, IF-ELSE blocks and built-in methods. It also contains complex data-types, e.g, a position data is a tuple of components specifying coordinate of a robot arm, orientation of robot axes and orientation of the tool attached to the robot arm.

An **abstract syntax tree (AST)** [7] is a tree representation of the abstract syntactic structure of source code. A **basic block** [7] is a structure containing a sequence of maximal program instructions. It can be exited only from the last of the instructions of the basic block. Let I_i be the i-th instruction in the code. It can be formally defined as,

$$I_0 \in B_0$$

$$\begin{cases} I_{i+1} \in B_{i+1} & \text{if } I_{i+1} \text{ is a target of a branching instruction} \\ I_{i+1} \in B_i & \text{otherwise} \end{cases}$$

A **control flow graph(CFG)** [37] is a graphical representation of all the paths which can be taken during an execution of the source code. It can be formally defined as a rooted directed graph $G = (N, E)$ where N is a set of basic blocks with two special basic blocks $entry$, $exit \in N$. $E : N \rightarrow N$, is a set of edges joining two basic blocks.

IV. Framework

The errors and warnings from various domain experts and engineers were identified. Moreover, the common properties across all languages and common errors encountered were noted. This helped us in designing a generic extraction and processing architecture. The common errors helped us in designing the rule engine. Figure 1 gives an overview of the tool. The overall working can be divided into two main stages:

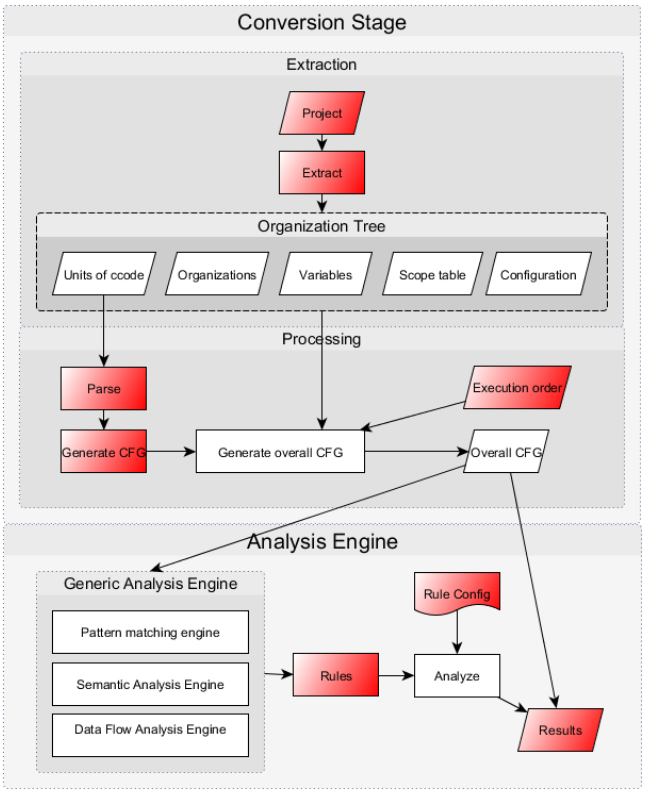

Fig. 1. Overview of the prototype

- **Conversion Stage:** This stage is divided into two stages, **Extraction:** Control code is extracted from project and loaded to the tool. The entire code is transformed into organization tree which further broken down into organization units that represents the code blocks. A code unit contains actual control code. It populates all necessary information like, variables, configuration and code hierarchy.
  Processing Stage: Scoping of variables was an definite problem. During the processing stage, the local and global variables are kept in a scope table. This can be used to extract the information during DFA. In compliance with the syntax and semantic of the language, control code is parsed in order to generate the Abstract Syntax Tree (AST). AST represents the syntactical structure of the control code. The control code is transfigured into intermediate data structures. AST in turn generate the CFG for the entire application. The nodes of the CFG are basic blocks of the source code. AST and CFG structures help to define the notion of program points [1] for our static analysis engine.
- **Analysis Engine:** Analysis engine consists of generic analysis engine and rule engine. Generic analysis engine mainly performs DFA and pattern based matching to detect potential sources of defects. The rule engine is configurable. This helps the user to write custom rules. The pattern matching engine checks rules that requires information about the syntactic structure of the program. The DFA engine computes the data flow facts by using Worklist algorithm for each of the basis blocks of the program. Worklist algorithm works at the core of DFA engine. It is of two types,
  - Forward
  - Backward

  For some of the rules like, live variable analysis backward worklist algorithm is used. Proposed framework provides flexibility to use backward worklist instead of forward worklist for specific rules.
  Algorithm 1 computes the DFA results and stores the result in the dictionary called Dfa_{IntCFG}. The algorithm takes the inter-procedural CFG [9] as an input and populates the data flow results in the dictionary. It maintains a list of basic blocks to be processed. The algorithm converges when it reaches a fixed point or the state of the block remain constant and the list is empty. The $transfer$ function mentioned in the algorithm converts the input to a basic block to the output by executing each operation on the ranges of the variables. This operation are defined later in this section. Stabilizing techniques like widening and narrowing [15] is used for faster convergence. Abstraction [1] for the program variables are used to avoid problem of state space explosion. Intervals are used as abstraction. It is a simple domain to represent the values for each variable at each basic block. Interval for a variable v can be defined as below,

$$I_v = (a_1, a_2), a_1, a_2 \in \mathbb{R}$$

  A set of operators defined on the intervals are,
  - **Arithmetic:** Add, Subtract, Multiply, Division, XOR, AND, OR etc.

$$I_0 = (a_0, a_1), I_1 = (a_2, a_3)$$

$$I_0 + I_1 = (a_0 + a_2, a_1 + a_3)$$

$$I_0 - I_1 = (a_0 - a_2, a_1 - a_3)$$

Algorithm 1 Worklist algorithm

```
1: procedure WORKLIST(IntCFG, Type)                                   ▷
2:     list ← {b | b is a basicblock in IntCFG }                      ▷
3:     dfa_IntCFG ← INITIALIZE(list)
4:     while list not empty do                                        ▷
5:         b1 ← list[0]
6:         OldValue ← dfa_IntCFG[b1]
7:         if Type ≠ Forward then
8:             inputs ← b1.successor
9:         else
10:            inputs ← b1.predecessor
11:        end if
12:        for each element ∈ inputs do
13:            value ← value ∪ dfa_IntCFG[element]
14:        end for
15:        value ← value.transfer
16:        if OldValue ≠ value then
17:            dfa_IntCFG[b1] ← value
18:            if Type ≠ Forward then
19:                outputs ← b1.predecessor
20:            else
21:                outputs ← b1.successor
22:            end if
23:            for each element ∈ outputs do
24:                list ← list ∪ element
25:            end for
26:        end if
27:        list.delete(0)
28:    end while
29:    return dfa_IntCFG                                              ▷
30: end procedure
```

Algorithm 2

```
1: procedure INITIALIZE (list)                                        ▷
2:     Dictionary d
3:     for each b ∈ list do
4:         if b.Type ≠ entryblock then
5:             d[b] ← [ ]
6:         else
7:             intervals ← {(minval(v), maxval(v)) |
   ∀v ∈ var(b)}
8:             d[b] ← intervals
9:         end if
10:    end for
11:    return d                                                       ▷
12: end procedure
```

$$I_0 \times I_1 = (\min(a_0 \times a_2, a_1 \times a_3, a_0 \times a_3, a_1 \times a_2), \max(a_0 \times a_2, a_1 \times a_3, a_0 \times a_3, a_1 \times a_2))$$

$$I_0 / I_1 = (\min(a_0/a_2, a_1/a_3, a_0/a_3, a_1/a_2), \max(a_0/a_2, a_1/a_3, a_0/a_3, a_1/a_2))$$

$$I_0 \text{ OR } I_1 = (\min(a_0 \text{ OR } a_2, a_1 \text{ OR } a_3, a_0 \text{ OR } a_3, a_1 \text{ OR } a_2), \max(a_0 \text{ OR } a_2, a_1 \text{ OR } a_3, a_0 \text{ OR } a_3, a_1 \text{ OR } a_2))$$

$$I_0 \text{ AND } I_1 = (\min(a_0 \text{ AND } a_2, a_1 \text{ AND } a_3, a_0 \text{ AND } a_3, a_1 \text{ AND } a_2), \max(a_0 \text{ AND } a_2, a_1 \text{ AND } a_3, a_0 \text{ AND } a_3, a_1 \text{ AND } a_2))$$

$$\text{NOT } I_0 = (a_1 + \epsilon, a_0 - \epsilon), \epsilon \text{ tends to } 0, \epsilon > 0$$

– **Relational:** $\leq, \geq, <, >, =$ etc.

$$I_0 = I_1 = \begin{cases} (1,1), a_0 = a_2 \text{ and } a_1 = a_3 \\ (0,0), a_1 \leq a_2 \text{ or } a_3 \leq a_0 \\ (0,1), \text{otherwise} \end{cases}$$

$$I_0 < I_1 = \begin{cases} (1,1), a_1 < a_2 \\ (0,0), a_0 \geq a_3 \\ (0,1), \text{otherwise} \end{cases}$$

$$I_0 \leq I_1 = \begin{cases} (1,1), a_1 \leq a_2 \\ (0,0), a_0 > a_3 \\ (0,1), \text{otherwise} \end{cases}$$

$$I_0 > I_1 = \begin{cases} (1,1), a_0 > a_3 \\ (0,0), a_1 \leq a_2 \\ (0,1), \text{otherwise} \end{cases}$$

$$I_0 \geq I_1 = \begin{cases} (1,1), a_0 \geq a_3 \\ (0,0), a_1 < a_2 \\ (0,1), \text{otherwise} \end{cases}$$

Built-in functions of each language are implemented separately and not included in the generic framework as they are domain specific. The rules configuration file gives flexibility to the user to select the subset of the rules according to the project requirement. In the rules configuration file, the properties for the rules can be specified to be checked against the control code. E.g, in level of nesting rule, the parameter to set the limit for nesting in loops can be specified. If the level of nesting in code exceeds the limit, the analysis engine can detect and notify the user.

Analysis engine detect code violations in accordance with the rules specified in the generic analysis engine. Syntactic errors can be checked against the AST where as generic programming errors can be verified through DFA. The output file contains the meta-data for the error like type, description, line number and warning level. The warning level depicts the severity of the error.

V. Language specific implementation

Figure 2 explains analysis engine in accordance to different rules based on their scope. Language and project specific rules can use the analysis framework of the tool for their analysis. Language specific rules are defined on particular language. These rules might not be applicable for other languages. An example of such a rule of RAPID is the boundary violation check for the robot arm. Another example for EDDL is variable redefinition rule which is critical for EDDL language and hence should be checked at compile time. An example of project specific rule is nesting levels in the code. For the ease of readability and maintainability, nesting levels of the code for a particular application needs to be low. Conformance to NORSOK standards for oil and gas application is another example of project(application) specific rules. The project specific and language specific rule engine is decoupled from the analysis framework for the ease of the user. The rules can be categorized into two types based on the type of analysis required to check the rules. They are a. Pattern based rules and b. DFA rules.

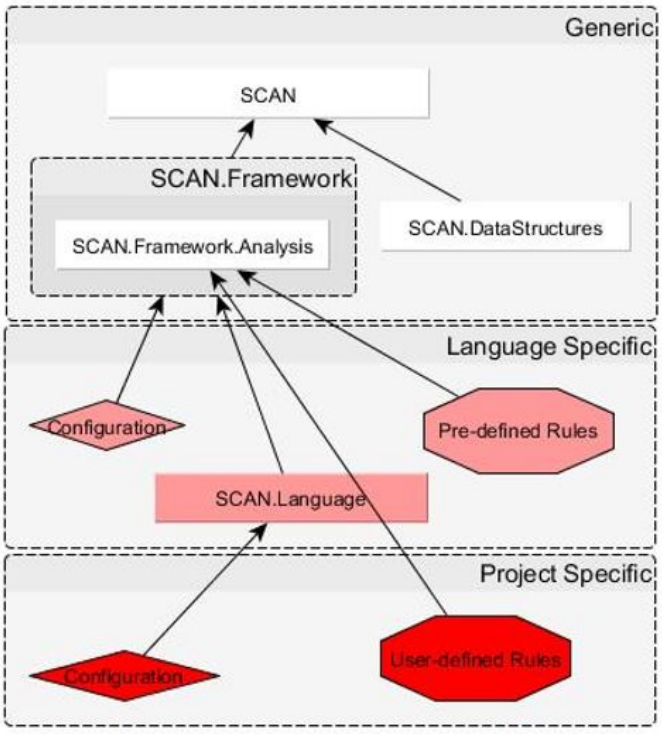

Fig. 2. Overview of the SCAN tool

Pattern based rules are checked with the help of AST. Such rules can be project or language specific. Sometimes certain information about the variables are also needed for checking the pattern based rules. Such rules are comparatively easier to check. Initially, variables and values were generated for each code unit at the run time, inefficiently. Thus holders for each variable was generated and was then updated at the run-time.

```
CreateArray( Array => Array_var,
             FirstIndex := 1,
             LastIndex := 2,
             ArrayElement := array_element,
                             Status => status );
index_val := status;
index_val := index_val +1;
putArray( Array := Array_var,
          Index := index_val,
          ArrayElement := array_element,
          Status => status );
index_val := index_val + 3;
putArray( Array := Array_var,
          Index := index_val,
          ArrayElement := array_element,
          Status => status );
while (run < 3) do
if (run < 3 ) then
   Mountain_Height := 5;
else
   Mountain_Height := 7;
end_if;
exit;
run := run + 1;
end_while;
```

Fig. 3. Sample code for IEC 61131-3

For this reason, CFG and inter-procedural CFG is constructed and annotated with the data flow results. This serves as a holder for each of the variable.

A. IEC 61131-3

Figure 3 is a buggy code written in IEC 61131-3 language. The variable `status` is initialized to 1 during declaration. The `CreateArray` function creates an array with `FirstIndex` 1 and `LastIndex` 2. The variable `index_value` takes the value of `status` (which is 1) and then it is incremented by 1. The call to `putArray` function works fine with the value of `index_val` 2. Then `index_val` is incremented by 3. The second call to `putArray` function fails due to array access out of bounds as `index_val` is now 5. Our tool report it as **Array Out-of-Bound** error. This is generic property across all the languages.

B. EDDL

Fig 4 represents the sample EDDL code. The code is divided into two sections, `VARIABLE` and `METHOD`. EDDL allows to declare the `VARIABLES` separately with all the attributes. The code includes the declaration of a `METHOD` named `Sample_Method`. The method contains if and else block along with some computations.

The tool detects **String Concatenation** error at line 17 because the size of `b` declared is less than the cumulative size of `a` and `c`. It is a DFA based rule because length of the string is known at the run time only.

A little further in the code, `x` is assigned the value 6. The condition mentioned in the if-block becomes true. The first statement of the if-block assigns value 6 to the variable `len1`. The expression defined in line 22 generates the error **Divide-by-zero** because the denominator becomes zero. This rule comes in the category of DFA based rules because range of the variables are known at the run time. Annotated CFG is generated using our framework for figure 4 for DFA analysis.

The variable `len1` is assigned the value 20 inside if condition of the outer else block. An error is generated for line 28 because the variable `len1` is used before assignment. This rule can be checked by pattern based matching technique with the help of an AST. The list containing all the variables defined in the declarative block is maintained. If any variable is used in the code which is not the part of the list, **Uninitialized Variable** error is reported.

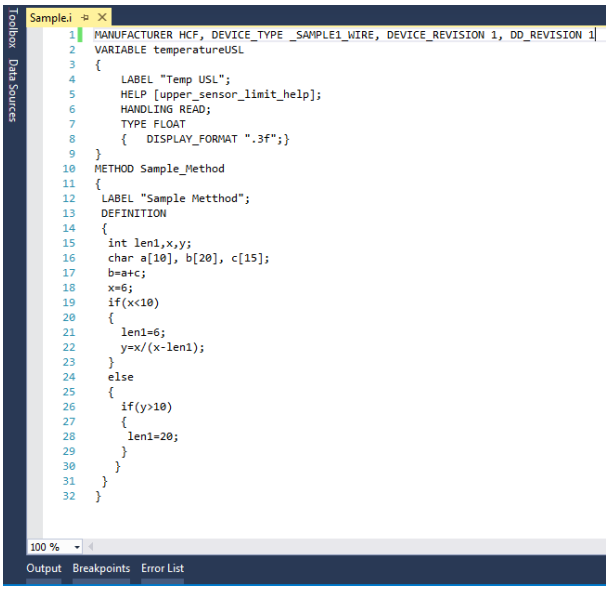

Fig. 4. Sample code for EDDL

C. RAPID

Figure 5 is a code snippet for RAPID and also depicts the hierarchal structure for a RAPID application. The application **TestProject** contains a task **T1**. **T1** contains several procedures and routines including the **main** procedure where the execution of the whole application starts. The module in the figure consists of a `main` procedure and few other procedures `SettoZero`, `DrawShape`, `SimpleBug`, and the trap routine `SimpleError` The variable (`di1`) causes the program to alter its flow and execute the trap routine `SimpleError`. Here `di1` represents a digital input signal and the in-built procedure call `ISignalDI` enables the interrupt `sig1` when `di1` is set to 1. `sig1` is connected to the trap routine `SimpleError` using `CONNECT`. This causes the trap routine `SimpleError` to service the interrupt `sig1`.

Each module consists of procedure/routines. To ensure that each code tab gets the latest values of the variables, the interleaving of procedure/routines between different modules must

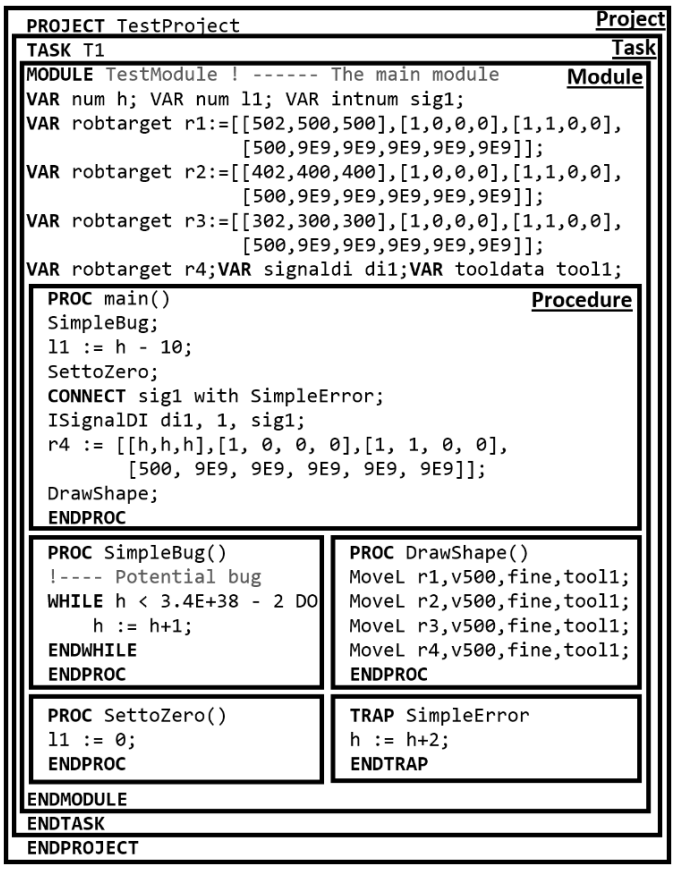

Fig. 5. Sample code for RAPID [5]

be considered. This motivates us to use inter-procedural CFG [9]. Inter-procedural CFG is used to generate an annotated inter-procedural CFG as demonstrated in figure 6. It represents the annotated inter-procedural DFA results for our example RAPID code(fig 5). The range of the variables are depicted in each node. There is a edge from **main** to the entry node of the procedure **SimpleBug**. This demonstrates the program path alteration during the execution of the code. In this example, the error is caused during such an alteration of program path.

RAPID has complicated data structures like, `robtarget`,

```
VAR robtarget p15:=[[600, 500, 225.3],
[1, 0, 0, 0], [1, 1, 0, 0],
[11, 12.3, 9E9, 9E9, 9E9, 9E9]]
```

which is a position data for the robot arm. We represent it abstractly for verifying safety properties like boundary violation for robot arm. Updated values of each of tuple in the **robtarget** variable is maintained in the annotated CFG. The first tuple in the annotated inter-procedural CFG denotes the value of the numeric variable h and the second tuple denotes the value of the numeric variable $l1$. Interval abstraction [1] is used to bound the values of the variables within desired ranges. Some domain-specific requirement can be defined in the rules file. An example of such a rule can be the robot arm never crosses a specified boundary. In the procedure `DrawSquare`,

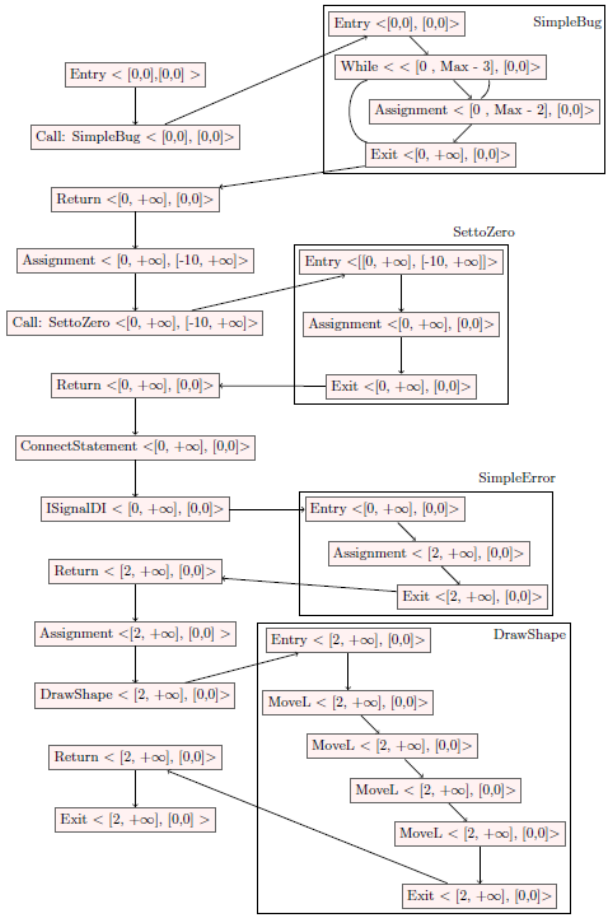

Fig. 6. Inter-procedural CFG annotated with values for the variables in the illustrative example [5]

the fourth `MoveL` instruction causes the robot arm to move outside the boundary. This can result in potential conflicts with other robots or static obstacles. The use of interval abstraction generates false positives during this process. The absence of false negatives is ensured since all the values of the variable is considered. This approach can detect many programming errors like array access out of bounds, integer overflow, infinite loops etc.

VI. Results

Based on level of generality and coupling with the language, the analyzed rules can be categorized into three types,

- **Generic Programming rules:** Only depends on the analysis framework. Generic across all languages.
- **Language specific rules:** Based on the language. This might not exist for other languages.
- **User specified rules:** Rules based on the application.

Based on program syntactic structures used to analyze the rules can be categorized into three types,

- **Pattern-based rules:** AST is used for performing the analysis.
- **Semantic rules:** AST and some data-type information is used for performing the analysis.
- **DFA:** CFG is used for the analysis.

Rules	Language	Generic errors	Language based	User specific
Pattern-based	IEC 61131-3	11	5	2
	EDDL	4	12	1
	RAPID	11	7	0
	Total	34	24	3
Semantic	IEC 61131-3	0	0	0
	EDDL	0	1	0
	RAPID	5	0	0
	Total	5	1	0
DFA	IEC 61131-3	5	1	1
	EDDL	4	2	0
	RAPID	1	4	1
	Total	10	7	2

TABLE I
Distribution of rules

The table I depicts the distribution of the rules based on the categories identified. E.g, Divide-by-zero is a generic error and it is analyzed by DFA technique. These rules were collected from domain experts and engineers.

Language	Rule name	Category	Error	Warning
IEC 61131-3	Incorrect Attribute	language specific	10	0
	Uninitialized variable	generic	0	76
	Datatype mismatch	language specific	0	9
	Divide-by-zero	generic	0	7
	Duplicate identifier	generic	0	1
EDDL	Divide-by-zero	generic	0	5
	Missing Mandatory Menus	language specific	0	23
	Unused variables	generic	0	24
	Assignment for comparison	generic	3	0
RAPID	Illegal wait statement	language specific	4	0
	Function side effects	language specific	0	4
	routines not used	generic	12	52
	Unused variable	project specific	41	15
	Arithmetic overflow	generic	2	2
	Constant	project specific	0	30
	Unreachable code	generic	3	7

TABLE II
Number of rules analyzed for each language

Table II discusses various rules analyzed for all three languages. These rules can be different types like, language-specific, project/application specific. Large number of errors were observed in some of the syntactic properties. Some errors were project/application specific rules like depth of nesting. In some applications, the depth of nesting must be low. In

case of RAPID the tool analyzes an industrial example which has a total 11812 statements. The cyclomatic complexity of the project is high (Maximum cyclomatic complexity 520, maximum depth of nesting is 7, maximum number of operators is 7). The industrial application also contains some exception handlers and interrupt service routines.

The main drawback of the proposed prototype is the generation of false positives. Interval domain records the range for variables in the program. Error detection solely depends on the value of the variable that may not be true during runtime. E.g, the range for the variable a is [-5, 5], the tool spotted it as error under Division-by-zero for the expression $c = b/a$. This is because 0 is one of the possible value for the variable a. Since the current value of the variable is unknown at the compile time, tool sometimes generates false positives. Another drawback is that the tool can not handle recursion efficiently. To overcome this, the worst case scenario is assumed when the function call stack exceeds a limit.

VII. Conclusion and future Work

This paper presents a generic static analyzer for three domain specific control languages that includes IEC 61131-3, EDDL and RAPID. DFA and pattern based matching are the main techniques used for the static analysis of the control code. The tool detects the potential sources of error and also ensures compliance with coding standards. The results of the tools were verified by the engineers and proved to be useful while deploying control code to the plant.

Currently, we have minimal support for handling recursion in our proposed prototype. Static analysis by worklist algorithm takes moderate amount of time for larger code. The tool can be improved in terms of time taken by the analyzer. There is scope for further work to create a generic static analyzer for other control languages. Another direction in future work can be increased use of the tool during control code development across different domains. It is a major challenge as every language has unique features. A model-checking approach in which the requirements are specified as formal language to generate rule definition is another approach for the future.

References

[1] Cousot, Patrick, and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.” Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. ACM, 1977.

[2] PLCopen: The third edition of IEC 61131-3

[3] Robotics, ABB “Technical reference manual RAPID Instructions, Functions and Data types.”, 2014.

[4] Nair, Sreeja, et al. ”A static code analysis tool for control system software.” Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on. IEEE, 2015.

[5] Mandal, Avijit, et al. ”A static analyzer for Industrial robotic applications.” Software Reliability Engineering Workshops (ISSREW), 2017 IEEE International Symposium on. IEEE, 2017.

[6] Mohan, Devina et al. ”Static Code Analysis for Device Description Language.” 2nd Symposium on Application of Formal Methods for Safety & Security of Critical Systems (AFMSS), 2018.

[7] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. “Compilers, Principles, Techniques.”, Boston: Addison wesley, 1986.

[8] Nielson, Flemming, Hanne R. Nielson, and Chris Hankin. Principles of program analysis. Springer, 2015.

[9] Reps, Thomas. “Program analysis via graph reachability.” Information and software technology 40.11: 701-726, 1998.

[10] Cortesi, Agostino. “Widening operators for abstract interpretation.” Software Engineering and Formal Methods, 2008. SEFM’08. Sixth IEEE International Conference on. IEEE, 2008.

[11] CLANG. A C family front-end for LLVM. URL: http://clang.llvm.org/

[12] Klocwork static code analysis. URL: https://www.klocwork.com/products-services/klocwork

[13] Coverity, URL: http://www.coverity.com/

[14] LDRA, URL: http://www.ldra.com/en/testbed-tbvision

[15] Cousot, Patrick, and Radhia Cousot. “Comparing the Galois connection and widening/narrowing approaches to abstract interpretation.” International Symposium on Programming Language Implementation and Logic Programming. Springer Berlin Heidelberg, 1992.

[16] AS Language Reference Manual, Kawasaki Industries Ltd., 2002.

[17] Miné, Antoine, “Weakly Relational Numerical Abstract Domains”, PhD thesis, École Normale Supérieure, 2004.

[18] PolySpace. http://mw.polyspace. corn.

[19] http://www.visual-expert.com/

[20] https://www.pylint.org/

[21] http://www.astree.ens.fr/

[22] Stattelmann, Stefan, et al. ”Applying static code analysis on industrial controller code.” Emerging Technology and Factory Automation (ETFA), 2014 IEEE. IEEE, 2014.

[23] Angerer, Florian, et al. ”Points-to analysis of IEC 61131-3 programs: Implementation and application.” Emerging Technologies and Factory Automation (ETFA), 2013 IEEE 18th Conference on. IEEE, 2013.

[24] Jovanovic, Nenad, Christopher Kruegel, and Engin Kirda. ”Pixy: A static analysis tool for detecting web application vulnerabilities.” Security and Privacy, 2006 IEEE Symposium on. IEEE, 2006.

[25] Bornot, Sbastien, et al. ”Utilizing static analysis for programmable logic controllers.” ADPM 2000: 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, September 18-19, 2000, Dortmund, Germany. 2000.

[26] Sunada, W. H., and S. Dubowsky. ”On the Dynamic Analysis and Behavior of Industrial Robotic Manipulators With Elastic Members.” Journal of Mechanisms, Transmissions, and Automation in Design 105.1 (1983): 42-51.

[27] Huang, Jeff, et al. ”ROSRV: Runtime verification for robots.” International Conference on Runtime Verification. Springer, Cham, 2014.

[28] CoDeSys Static Analysis 3.5.2.0, CoDeSys Professional Developer Edition, 2013, http://www.codesys.com/products/codesys-engineering/professional-developer-edition.html.

[29] Logi.CAD, Logi.lint, 2013, http://www.logicals.com/en/.

[30] Pavlovic, R. Pinger, and M. Kollmann, Automated Formal Verification of PLC Programs Written in IL, in 4th International verification workshop VERIFY, collocated with Conference on Automated Deduction CADE, 2007.

[31] . Bauer, S. Engell, R. Huuck, B. Lukoschus, and O. Stursberg, Verification of PLC programs given as sequential function charts, Integration of Software Specification Techniques for Applications in Eng, pp. 517540, 2004

[32] Adiego, Borja Fernndez, et al. ”Applying model checking to industrial-sized PLC programs.” IEEE Transactions on Industrial Informatics 11.6 (2015): 1400-1410.

[33] Cousot, Patrick, and Radhia Cousot. ”Comparing the Galois connection and widening/narrowing approaches to abstract interpretation.” International Symposium on Programming Language Implementation and Logic Programming. Springer, Berlin, Heidelberg, 1992.

[34] Facebook Infer, http://fbinfer.com/

[35] codesonar, https://www.grammatech.com/products/codesonar

[36] Pavlovic, Olivera, Ralf Pinger, and Maik Kollmann. ”Automated formal verification of PLC programs written in IL.” Conference on Automated Deduction (CADE). 2007.

[37] Frances E. Allen (1970): Control flow analysis. In: ACM SIGPLAN Notices - Proceedings of a symposium on Compiler optimization, 5, Association for Computing Machinery, Association for Computing Machinery, pp. 119, doi:10.1145/390013.808479.