A static analyzer for Industrial robotic applications

Avijit Mandal, Meenakshi D'Souza
IIIT Bangalore, India
Email: {avijit.mandal, meenakshi}@iiitb.org

Raoul Jetley, Sreeja Nair†
ABB Corporate Research, Bangalore, India
Email: {raoul.jetley, sreeja.nair}@in.abb.com

***Abstract*—In this paper, we describe a static analysis approach to detect potential runtime errors for a programming language that is used to program industrial robots. The language we deal in this paper is RAPID, a high level programming language for programming ABB industrial robots. The presence of real-time interrupts, exception handlers and complex data-types makes it a difficult language for general purpose static analyzers. The properties of interest include some generic programming errors as well as some domain-specific properties that the robot system must comply with. Generic programming errors include properties like integer overflow, array access out of bounds and division by zero. An example of a domain-specific property is defining boundary limits for robotic arm movement. We have developed a tool to detect these errors successfully in the presence of real-time interrupts.**

I. Introduction

Industrial robots perform tasks involving high degree of repetition and accuracy in industrial settings. Such robots run on a proprietary real-time platform, and are programmed using domain-specific languages that specify the working envelope, motion control and tasks for individual robots.

There are several proprietary robotics programming languages available: for example, ABB has RAPID [2], Comau has PDL2 [10], Kawasaki has AS [16] and Universal Robots has URScript [11]. Robots are programmed to perform various tasks like pick and place, welding, path finding, moving objects, monitoring and control, locating and sharing data, etc. Typically, these programming languages have a list of custom instructions and a program flow. Instructions are specific, simple, real-time tasks for the robots. Program flow is defined using commands similar to a high-level, imperative language.

Robots that are programmed using these languages are subject to stringent safety regulations since most of them operate in safety critical systems or environments. Commonly used standard safety regulations include IEC 61508 [12] and ANSI/RIA R15.06-1999 [13]. Compliance to these standards is typically established through a combination of code review and extensive (manual) testing. However, a more exhaustive and efficient method for safety assurance is required as the programs grow larger and more complex.

For software driven controllers, the notion of safety is well developed. Stringent testing guidelines, program analysis to detect low-level errors and rigorous formal methods approaches help establish adherence to safety standards. However, most of the tools that aid in such verification and validation tasks are for general purpose languages like C/C++, Java, etc., and not applicable for robotics programming languages as these languages are proprietary and have their own instruction sets. We aim to bridge this gap by introducing a static analysis tool which can verify the adherence to safety standards and also enable users to write their project specific compliance rules, if any.

We consider the problem of detecting generic programming errors and compliance violations in the robotics programming language RAPID. Typical generic programming errors include division by zero, unused variables, dead code, array access out of bounds etc. Detecting such errors for a language like RAPID involves working with control flow structure that runs across several tasks spread out among different modules. There are also specific real-time commands that deal with handling of interrupts and exceptions.

Popular program analysis tools like Klocwork [7], CLANG static analyser [6], Coverity [8], LDRA testbed [9] etc., cannot be used for RAPID due to the presence of custom-made instructions and real-time interrupts. These instructions are not just procedures but, involve asynchronous interactions with a real-time platform. We have therefore developed a comprehensive program analysis framework for RAPID using the .NET development platform. Our framework includes extracting the AST and CFG for a RAPID project and performing detailed data flow analysis on these.

Even though our analysis framework is developed specific to RAPID, it can be used for any robotics programming language with similar features. Detecting generic errors is the first step towards building a robust verification and validation platform for such programming languages ensuring conformance to various safety considerations. For example, using our analysis framework, we are able to prove that a robot arm will never exceed the specified boundary for its movement, preventing clashes with other robots/components.

The rest of this paper is organized as follows. Section II introduces the RAPID programming language and its important features through detailed examples. We present our program analysis framework in Section III. An example that illustrates the use of our framework is presented in Section II. Section IV presents conclusion and future work.

II. Rapid

RAPID [2] is a structured programming language for programming ABB industrial robots. A program written in RAPID usually contains many high level *instructions* for robot specific actions which abstracts the complex mathematical

†Sreeja Nair is currently at Université Pierre et Marie Curie, Paris

calculations from the user. Apart from the instructions, the language uses routine *program flow* in an imperative style programming language.

Some common instructions found in a RAPID program include:

- `WaitTime 200;` Instructs the robot to wait for 200 seconds before doing any assigned work.
- `IDelete intr;` Disables the interrupt variable `intr`.
- `MoveL p1,v500,z10,tool1;` Moves the position of the robotic tool tool1 linearly to the position p1, with speed data v500 and zone data z10. This internally calculates the torque that needs to be applied to each axis (motor) to move linearly to the position `p1`.

RAPID supports several data types including constants, variables (of type num, string, bool etc.), arrays etc. The language also supports complex data types specific to robot instructions. Two such data types, `robtarget` and `pos` are described below.

- A declaration of the form
  `VAR robtarget p15:=[[600, 500, 225.3], [1, 0, 0, 0], [1, 1, 0, 0], [11, 12.3, 9E9, 9E9, 9E9, 9E9]];`
  defines the position of a robot. It is made of primitive datatypes. As the robot is able to achieve the same position in several different ways, the axis configuration is also specified. The first tuple provides the position in 3-D space. The last three tuples specify orientation of the tool, axis-configuration of the robot and the position of external axes respectively [2].
- Consider the declaration of `pos` below.
  `VAR pos := [500, 0, 940];`
  `pos` is used to define a position in the 3-D space where the X-coordinate, Y-coordinate, Z-coordinate are 500, 0, 940 respectively [2].

Program flow in RAPID is specified using standard imperative language constructs including relational and logical expressions, IF-THEN-ELSE statements and FOR and WHILE loops. RAPID also supports handling of exceptions and interrupts that alter the control flow of a program.

RAPID program is modularized by grouping code into procedures, there is a `main` procedure in RAPID from where execution begins. RAPID programs are highly hierarchical: A *project* contains one or more *tasks* which control robot actions. Tasks within a project can run asynchronously. RAPID code in tasks are logically grouped into *modules*, which, in turn, contain *routines*. Routines are the smallest unit of code, they can be a function, a procedure or a *trap* routine. A function provides a return value and a procedure contains a set of instructions. Trap routines are executed with interrupts occur and can alter the asynchronous execution of tasks.

The presence of real-time and domain specific instructions, asynchronous tasks, exception handling and handling of hardware interrupts make program analysis difficult for RAPID programs. Standard program analysis tools will not work in RAPID programs, even for detecting errors within a procedure.

```
PROJECT TestProject                                          Project
TASK T1                                                      Task
MODULE TestModule ! ------ The main module                   Module
VAR num h; VAR num l1; VAR intnum sig1;
VAR robtarget r1:=[[502,500,500],[1,0,0,0],[1,1,0,0],
                  [500,9E9,9E9,9E9,9E9,9E9]];
VAR robtarget r2:=[[402,400,400],[1,0,0,0],[1,1,0,0],
                  [500,9E9,9E9,9E9,9E9,9E9]];
VAR robtarget r3:=[[302,300,300],[1,0,0,0],[1,1,0,0],
                  [500,9E9,9E9,9E9,9E9,9E9]];
VAR robtarget r4;VAR signaldi di1;VAR tooldata tool1;
  PROC main()                                                Procedure
  SimpleBug;
  l1 := h - 10;
  SettoZero;
  CONNECT sig1 with SimpleError;
  ISignalDI di1, 1, sig1;
  r4 := [[h,h,h],[1, 0, 0, 0],[1, 1, 0, 0],
        [500, 9E9, 9E9, 9E9, 9E9, 9E9]];
  DrawShape;
  ENDPROC
  PROC SimpleBug()
  !---- Potential bug
  WHILE h < 3.4E+38 - 2 DO
      h := h+1;
  ENDWHILE
  ENDPROC
  PROC DrawShape()
  MoveL r1,v500,fine,tool1;
  MoveL r2,v500,fine,tool1;
  MoveL r3,v500,fine,tool1;
  MoveL r4,v500,fine,tool1;
  ENDPROC
  PROC SettoZero()
  l1 := 0;
  ENDPROC
  TRAP SimpleError
  h := h+2;
  ENDTRAP
ENDMODULE
ENDTASK
ENDPROJECT
```

Fig. 1: A sample RAPID module

We would like to observe that data structures like inter-procedural call graph need to be created for asynchronously running tasks, in the presence of interrupts.

Illustrative Example

Fig. 1 shows a sample RAPID module illustrating some of the features of the RAPID language. 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` [2] 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`.

III. Analysis Framework

Fig. 2 gives the overview of the tool. The tool has two stages— the *conversion stage* and *analysis engine*. The conversion stage deals with the extraction and pre-processing of the program and the analysis engine performs the analysis. The conversion stage is further divided into two – the extraction phase, which reads the project and converts it into the intermediate data structures, and the processing part, which parses the code and generates the control flow graph and call

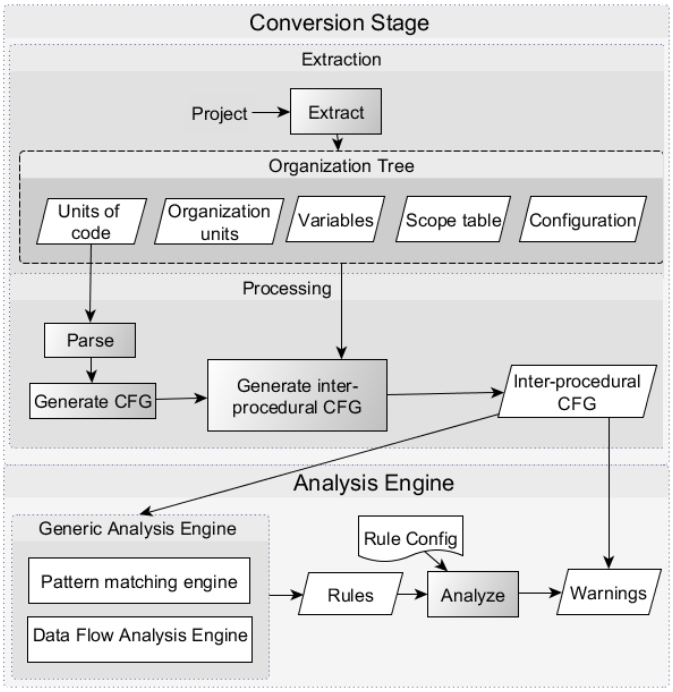

Fig. 2: Overview of the tool

graph. The two phases of the conversion stage populate the *Organization tree* which gives an intermediate representation of the project.

The organization tree contains a number of *organization units* and each in turn can contain other organization units. A *code unit* is a special organization unit which contains code. The code is processed into an Abstract Syntax Tree (AST), then into a Control Flow Graph (CFG) and stored inside the code unit. The program flow is encoded as a call graph and stored inside the organization unit. The AST, CFG and program flow information are then used by the analysis engine to perform the analysis.

The rules for the analysis are specified in a *RuleConfig* file which allows the user to select the subset of rules as per the project requirement. The analysis itself is performed in two stages. Computation intensive operations like data-flow analysis are performed in the first stage. Results of this analysis are added as annotations to the organization tree itself. The second stage performs a pattern check over the organization tree including the annotations to detect anomalies and generates a list of warnings for the user. The warnings are classified into different levels depending on their severity, 1 being the most critical and 5 the least severe.

Data Flow Analysis.

We use a combination of traditional Data Flow Analysis (DFA) [3] and abstract interpretation [1] to construct the annotated CFG. The DFA algorithm implemented:

- Uses *forward* or *backward* as direction during the CFG traversal
- Evaluates data flow facts at each program point as a semi-lattice with the meet operator
- Uses worklist algorithm [3] to generate data flow facts for each program point until the fix-point has been reached.
- Uses an abstract domain to represent the data flow facts at each program point. To start with, we have used interval abstraction [1] ($a \leq X \leq b$ where X is a variable a, b are to be determined by analysis) to represent the possible values of every variable at each basic block.
- Uses widening and narrowing [15] operations to stabilize the worklist algorithm at some program points.

Prototype implementation

We use the RobotStudio [14] platform API to extract the XML file from RAPID code. We analyze this XML file and store the results in a text file that records the warnings generated. This tool works in a standalone mode, but can be extended as plugin to the existing RobotStudio platform for RAPID.

The properties of RAPID programs that can be checked by our tool can be categorized in two types:

1) Syntactical rules checked using the AST generated by our tool.
2) Rules defined on the CFG augmented with data as the result of data flow analysis on the control flow graph.

Some of the properties that our tool can check are:

- Out of bounds array access – Perform data flow analysis on the inter-procedural CFG to evaluate valid access to array elements.
- Infinite loop – Check the value of a loop variable to assess whether it exceeds the maximum limit for a numeric data-type. We use widening and narrowing[15] techniques to detect such errors.
- Division by Zero – Check the interval of the denominator to detect whether it can evaluate to zero. Since we use widening operators during data flow analysis, there can be false positives in this detection.
- Unused variable detection – Checked against the def-use chain derived from the annotated inter-procedural CFG.
- Unused code detection – Checked in two ways. The first approach simply checks whether every basic block is reachable from the entry node of the CFG. The second approach uses the result of interval analysis to check the branch and loop conditions to determine whether any invariant condition leads to unreachable code.
- Boundary violation warning – Use results from interval analysis to detect if the robot arm violates a specified boundary. The boundary specification is stored in a configuration file by the user.
- Late binding violation – Check if the procedure generated during late binding is valid and is not already loaded in memory.

Illustrative Example for Error Detection

We now illustrate the use of our tool for the example presented in Section II.

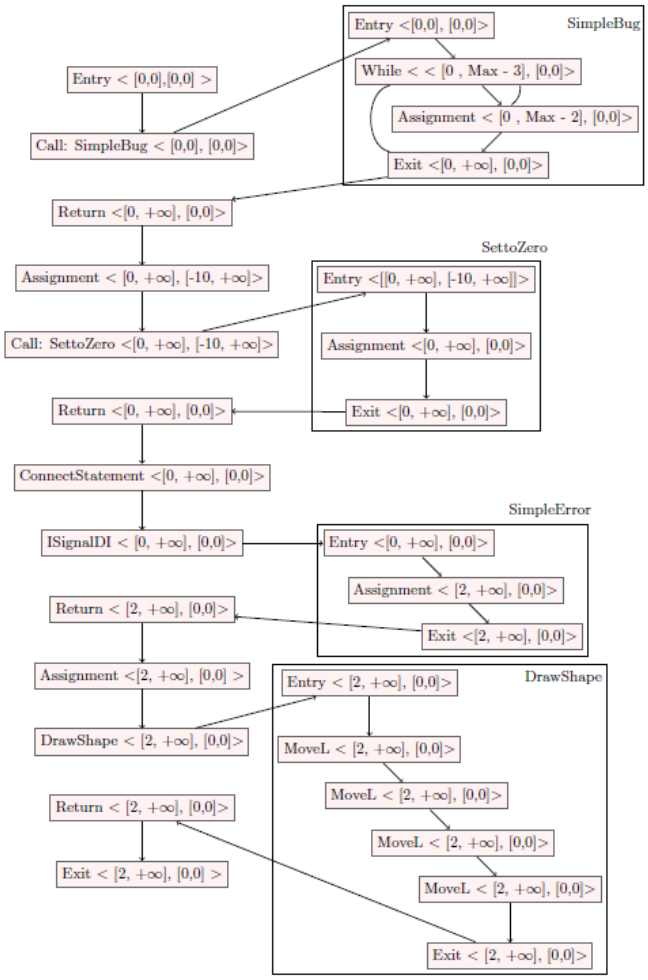

Fig. 3: Inter-procedural CFG annotated with values for the variables in the illustrative example

Fig. 3 shows the inter-procedural control flow graph for our code. The CFG is annotated using the results of data flow analysis performed using the interval domain.The first tuple in the annotated CFG denotes the value of the `num` variable `h` and the second tuple denotes the value of the `num` variable `l1`. We use *interval abstraction* [1] to bound the values of the variables within desired ranges.

A domain specific requirement is to ensure that the robotic arm will never cross a specified boundary. We have written rules in our analysis engine to check whether any point specified in the code is outside our permissible boundary at run-time. In the procedure `DrawSquare`, the fourth `MoveL` instruction the point `r4(robtarget)` will be outside the boundary. This can result in potential conflicts with other robots.

In order to detect this error, we specify the boundary for the robot arm in a configuration file. The specification has an interval for each x, y and z co-ordinates. The analysis algorithm iterates through all the basic blocks of the annotated inter-procedural CFG (Fig. 3) to check the values of the variable `pos`. The value at each basic block is compared with the value from the configuration file and a warning is raised when interval falls out of the specified boundary. As illustrated in Fig. 3, the `robtarget` variable `r4` might be out of the boundary (its first tuple is a `pos` variable whose all the co-ordinates are in the interval $[2, +\infty]$).

False positives may arise during this process, as we use interval abstraction. But we will be able to ensure absence of false negatives since all the values which the variable can take is considered.

The above approach can also be used to detect many programming errors like array access out of bounds, integer overflow, infinite loops etc.

IV. Conclusion and Future Work

We have presented a program analysis framework to detect potential runtime errors in the RAPID robotics programming language. Our framework can serve as a backbone for a full-fledged program analysis engine that can serve complete verification and validation requirements for robotics programs.

We plan to augment our framework with timing analysis, *WCET* analysis and symbolic execution based analysis. False positives in the result can be reduced by selecting a more refined numerical domain or a combination of different domains[17]. The aim is to eventually integrate the static analysis tool with the robotics development platform RobotStudio for RAPID.

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] Robotics, ABB “Technical reference manual RAPID Instructions, Functions and Data types.”, 2014.
[3] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. “Compilers, Principles, Techniques.”, Boston: Addison wesley, 1986.
[4] Reps, Thomas. “Program analysis via graph reachability.” Information and software technology 40.11: 701-726, 1998.
[5] Cortesi, Agostino. “Widening operators for abstract interpretation.” Software Engineering and Formal Methods, 2008. SEFM’08. Sixth IEEE International Conference on. IEEE, 2008.
[6] CLANG. A C family front-end for LLVM. URL: http://clang.llvm.org/
[7] Klocwork static code analysis. URL: https://www.klocwork.com/products-services/klocwork
[8] Coverity, URL: http://www.coverity.com/
[9] LDRA, URL: http://www.ldra.com/en/testbed-tbvision
[10] Dogliani, F., and Comau SpA. “New generation control architecture for the robot of the ’90s.” Science and Technology, 1990.
[11] Lapham, John. “RobotScript: the introduction of a universal robot programming language.” Industrial Robot: An International Journal 26.1: 17-25, 1999.
[12] Bell, Ron. “Introduction to IEC 61508.” Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55. Australian Computer Society, Inc., 2006.
[13] Thornton, J. “Robot safety: ANSI/RIA R15. 06-1999 and savvy safeguarding for robotic workcells.” Robotics Online, 2002.
[14] Robotics, ABB “Operating Manual RobotStudio.” Vasteras, Sweden, 2007.
[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.