ARCTIC

A Static Analyzer for COBOL-85 Programs

Roberto Giacobazzi
University of Verona
Italy
roberto.giacobazzi@univr.it

Alberto Lovato
University of Verona
Italy
alberto.lovato@univr.it

Isabella Mastroeni
University of Verona
Italy
isabella.mastroeni@univr.it

Abstract

We developed a static analyzer for COBOL-85 programs, called ARCTIC (AbstRaCT Interpretation of Cobol). It currently performs a lot of syntactic analyses, as well as semantic interval analysis. Semantic analysis is performed on an intermediate imperative language, simpler than COBOL. It has arithmetic operations, assignment, `If` and `While` statements. Each COBOL statement is translated into this language by a *translator*, an entity mapping different equivalent (sets of) statements into the same intermediate code. Every intermediate statement has an associated `execute` function that transforms state by modifying domain elements corresponding to each variable. Other than simpler statements, for `If` the execution is delegated to the two branches. For `While`, the `execute` function incorporates fixpoint computation for loops. The tool has command-line and remote interfaces, and can be used by means of a SonarQube plug-in.

Keywords Static Analysis, Abstract Interpretation, Software Engineering

1 Introduction

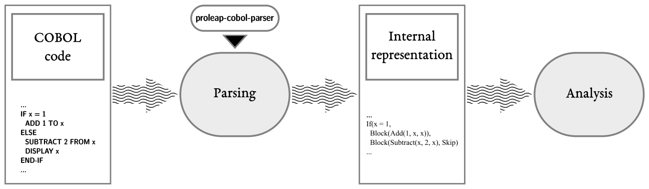
COBOL was designed in 1959 by CODASYL and still represents among the most widespread programming language on which legacy code is written and maintained [1]. In 1997, the Gartner Group reported that 80% of the world's business ran on COBOL with over 200 billion lines of code and 5 billion lines more being written annually [8]. Well known bugs such as the year 2000 problem (Y2K) was the focus of significant COBOL programming effort, sometimes by the same programmers who had designed the systems decades before. In 2006 and 2012, Computerworld surveys found that over 60% of organizations used COBOL (more than C++ and Visual Basic .NET) and that for half of those, COBOL was used for the majority of their internal software. 36% of managers said they planned to migrate from COBOL, and 25% said they would like to if it was cheaper. It is expected that maintaining legacy COBOL code will represent a major problem in next decades. Making changes to software and in particular unnecessary, uncontrolled, and careless changes, can have an effect on its appropriateness and validity, its correct operation, and can make it less efficient, or in extreme cases obsolete. From the research and academic point of view, COBOL is considered marginal and of modest interest, although still the largest part of the legacy code is written in this language. Expert COBOL programmers are nearly retiring and the higher education system does not provide new and well skilled programmers capable of maintaining and patching the huge amount of legacy code that is still running everyday in enterprises and financial institutions. Moreover, the lack of interest of the academic and research community in the study and development of COBOL has been standing for decades. As a consequence, the structure of existing tools and methods for debugging and verifying COBOL programs reflects this retard, often employing out of date technologies (mostly based on simple syntactic analysis or pattern matching with known bug repositories). Few tools exist that are based on formal methods for the verification and debugging of COBOL programs and none of them employ sound-by-construction program verifiers and debuggers. Formal methods [7] are mathematically based techniques for the specification, development, and verification of software and hardware systems. They establish the satisfaction of a required property (called the specification) by a formal model (called the semantics) of the behavior of a system (for example, a program and its physical environment). Formal (logic based) verification methods are very hard to put in practice for large code, because both the semantics and the specification of a complex system are extremely difficult to define. Even when this is possible, the proof cannot be automated without great computational costs and none of these methods are used for verifying COBOL code. Static code analysis is the fully automatic analysis of a computer system by direct inspection of the source or object code describing the system with respect to the semantics of the code (without executing programs, as in dynamic analysis). Abstract interpretation [6] is a standard and well recognized method for the design and implementation of scalable and sound static analysis tools. We propose the prototype of a sound-by-construction static program analyzer for COBOL programs for the verification of well defined properties. In particular, the analysis we propose allows numerical variable bound checking, but in the future we plan to develop other checks, such as taint analysis and type conversion analysis.

TAPAS '18, August 28, 2018, Freiburg im Breisgau, Germany
2018.

2 Tool Description

The analyzer was developed in Java, and considers programs written following the COBOL-85 standard. Analyzing Cobol programs involves basically three phases:

- parsing Cobol code
- building an internal representation of the parsed program
- running analyses on this internal representation



```
...
IF x = 1
  ADD 1 TO x
ELSE
  SUBTRACT 2 FROM x
  DISPLAY x
END-IF
...
```

```
...
If(x = 1,
  Block(Add(1, x, x)),
  Block(Subtract(x, 2, x), Skip)
...
```

Figure 1. Phases of the analysis of a Cobol program

Parsing COBOL-85 is the most widespread Cobol standard in use, and the one we decided to focus on. The syntax of Cobol, despite having a simple structure, is very wordy, since the designers wanted to create a programming language resembling English, to allow non-technical people, such as managers, to comprehend programs. This means a lot of keywords, and constructs. Indeed, the same code can be represented by many different (sets of) statements. As a simple example, the following three statements are equivalent:

```
ADD 1 TO X
ADD 1 TO X GIVING X
COMPUTE X = X + 1
```

There are also different *dialects*. Parsing in our tool is made by an external library, *proleap-cobol-parser* [2], a parser for COBOL-85, supporting ANSI 85, IBM OS/VS and MicroFocus dialects. It is actively developed, and used by several clients. We are contributing in reporting bugs encountered in parsing legacy Cobol code. The parser produces an abstract syntax tree (AST), representing Cobol elements such as divisions, statements and declarations in a hierarchical way. Elements of the AST can be accessed by using a *visitor*, a well-known pattern allowing client programmers to act on elements of a certain type by simply overriding a method in a class. Right after parsing, syntactic checks can be made. We developed more than 100 different syntactic analyses, checking simple—but undeniably significant—facts about code, like the use of obsolete or insecure statements, bad coding practices, or type errors. For example, the `ACCEPT` statement reads input from external sources, including standard input. In general, its use must be avoided, since it is a security threat to allow direct input from a user. As another example, too many nested control-flow statements, like `PERFORM`, `IF` or `EVALUATE`, make programs complex and difficult to maintain, so the nesting should be limited. These rules can be enforced by syntactic inspection of code.

Some analyses check the SQL code inserted into Cobol programs. The Cobol parser allows clients to access the SQL code enclosed in an `EXEC SQL` statement. This string of SQL code is then parsed with another external library, *JSqlParser* [3], by using the visitor pattern again.

Internal Representation For semantic analysis, parsed Cobol code is translated into an equivalent, much simpler language. It is an imperative language, with numeric variables, like those of Cobol. They can contain signed decimal numbers with an integer part and a fractional part. The exact type of the variable is declared in Cobol with a `PICTURE` clause. For example, the string `S99V9` indicates a sign, two digits for the integer part and one for the fractional. Regarding statements, it has arithmetic operations, assignment, `If` and `While`. *Translators* are classes with a `translate` method that transforms Cobol statements into intermediate statements. Cobol statements that concern arithmetic computation have corresponding intermediate statements, while those that are not (like `DISPLAY`) are translated into `Skip` statements, doing nothing. They are not simply deleted because they are used to keep track of original source and state at input of statement. Currently `COMPUTE` and `CALL` statements are approximated by the `Unknown` intermediate statement, that assigns the infinite interval (the most general information) to affected variables. In the future they surely could be represented by more precise intermediate statements.

Analysis The interaction between components happening when the interpreter executes the intermediate program is explained in Figure 2. For each statement in the program, the

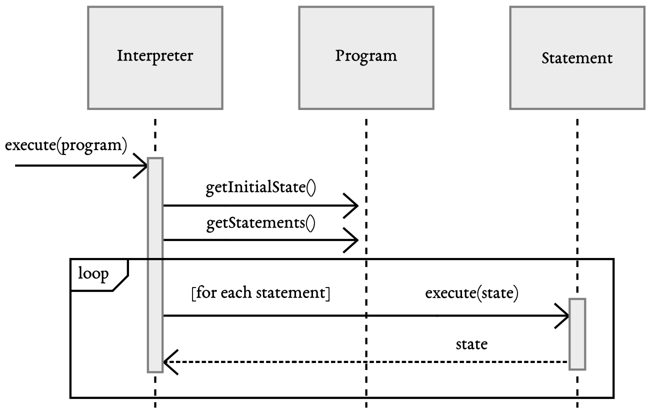


Figure 2. Execution of the program by the interpreter

`execute` method is called. The output state of the previous statement is used as input for the current statement. A statement can contain other statements—for example, branches of `If`—which transform state as well. Statements are implemented such that `execute` is called on the sub-statements whenever it is called on them. We initially focused on interval analysis, because it is simpler, but still can give valuable information on the content of each variable.

For interval representation and manipulation, the analyzer uses the *bd-intervals* library we developed. It implements arithmetic operations for intervals whose finite bounds are backed by `BigDecimal` objects. These are arbitrary precision decimal numbers, and so they can represent Cobol numeric values exactly. It features addition, subtraction, multiplication, division, along with union, intersection, widening and narrowing. It was developed entirely test-driven. The state of an `Interval` object is composed of the two bounds, of type `Bound`. The `Bound` interface has implementations `BigDecimalBound`, `EmptyBound`, `InfiniteBound`. The latter has concrete subclasses `NegativeInfiniteBound` and `PositiveInfiniteBound`. This hierarchy permits the creation of all intervals of interest, and makes the code simpler, by enabling dynamic dispatch to the class where responsibility lies for each particular operation. `Interval` objects are immutable, and hence they can be used in a concurrent context without additional precautions.

2.1 Development

The analyzer is currently in development at the University of Verona, using modern programming practices. Adopting test-driven development (TDD) allowed us to create modular and robust code. With TDD, production code is correct by design, with respect to the specifications formalized in test code. If tests pass, code is correct. Writing tests beforehand makes the system under test concise, modular and loosely coupled, since dependencies are not hardwired into production code. In fact, auxiliary objects used in production are generally substituted by *fake objects* when testing, and in order to *inject* references to these in place of real objects, dependency-breaking mechanisms are employed. With such a modular system, work can be distributed easily among developers.

3 SonarQube Plug-in

SonarQube [5] is a widely used platform for code quality management. Its most important module is a server program, to which one can connect with a browser and see statistics and issues related to a software project. A SonarQube *scanner* is a program that runs analyses and other computations on code, and then sends results to the server. The server saves the issues and provides a web interface that presents them nicely. For this reason, we are developing a plug-in for SonarQube, that communicates with the server version of Arctic and presents issues to the user. Figure 3 depicts interactions taking place when analyses are carried out by means of the plug-in. First of all, the list of *active* rules, i.e., those selected by the SonarQube user in the desired *quality profile*, is sent to the server. Then, issues related to analyzed files are sent back to the plug-in. Figure 4 shows issues generated by the interval analysis rule, as presented by the web interface of SonarQube.

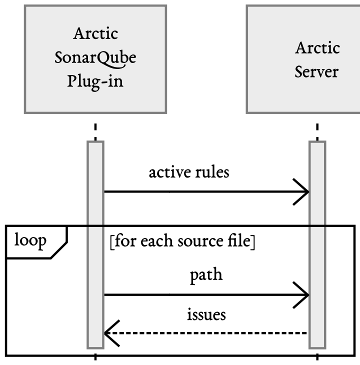
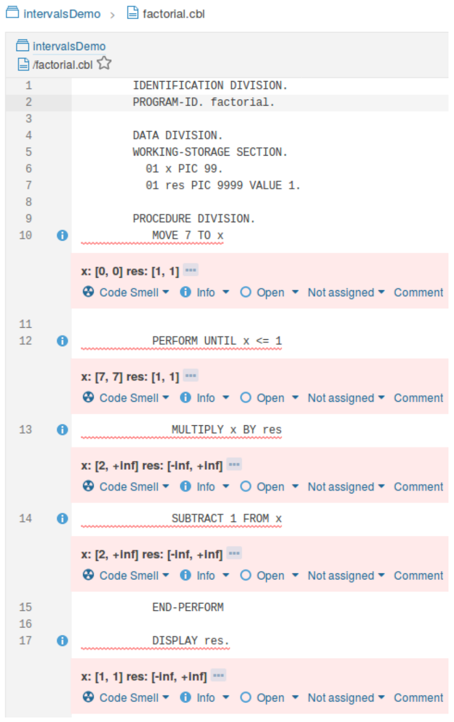


Figure 3. Interactions between the server and the SonarQube plug-in



```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. factorial.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
         01 x PIC 99.
         01 res PIC 9999 VALUE 1.

       PROCEDURE DIVISION.
          MOVE 7 TO x
x: [0, 0] res: [1, 1]

          PERFORM UNTIL x <= 1
x: [7, 7] res: [1, 1]
             MULTIPLY x BY res
x: [2, +Inf] res: [-Inf, +Inf]
             SUBTRACT 1 FROM x
x: [2, +Inf] res: [-Inf, +Inf]
          END-PERFORM

          DISPLAY res.
x: [1, 1] res: [-Inf, +Inf]
```

Figure 4. SonarQube view of interval analysis, where each issue represents intervals that are valid *before* the statement

4 Future Work

We intend to develop other analyses that will provide information useful in the discovering of important classes of vulnerabilities, such as injections of malicious code. Julia [4] is a mature static analyzer for Java code, performing many syntactic and semantic checks. In the future we might want to collaborate with the Julia team to integrate analyses and domains.

References

[1] 2009. IBM: Cobol z/OS language reference. (2009).

[2] 2018. COBOL-85 parser. https://github.com/uwol/proleap-cobol-parser. (2018). Accessed: 2018-08-04.

[3] 2018. Java SQL parser. https://github.com/JSQLParser/JSqlParser. (2018). Accessed: 2018-08-04.

[4] 2018. The Julia Static Analyzer. http://www.juliasoft.com. (2018). Accessed: 2018-08-04.

[5] 2018. SonarQube platform. https://www.sonarqube.org/. (2018). Accessed: 2018-08-04.

[6] P. Cousot and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*). ACM Press, 238–252.

[7] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria. 2008. Software engineering and formal methods. *Comm. of The ACM* 51, 9 (2008), 54–59.

[8] R. Kizior and D. Carrand P. Halpern. 2000. Does COBOL Have a Future?. In *Proceedings of the Information Systems Education Conference*.