Program Analysis Tools in Automated Grading

Elizabeth Dinella Samuel Breese Evan Maicus
Barbara Cutler Buster Holzbauer Ana Milanova

Rensselaer Center for Open Source

Static Analysis

Static analysis is a critical tool to ensure software security and dependability. In industry and academia, static analysis can be used to find potential bugs or design problems. On Submitty, an open source homework server created at RPI, static analysis is used to ensure structural correctness of homework assignments. This functionality is used in RPI's CSCI 1100 Computer Science 1 to automatically grade small in-lecture exercises. For example, consider the following assignment:

```
for i in range(0,3):
    print i
```

(a) Assignment: rewrite the above code using a while loop

```
i = 0
while i < 3:
    i += 1
```

(b) Correct Student Rewrite

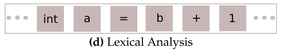
```
#while loop
for i in range(0,3):
    print i
```

(c) Incorrect Student Rewrite - creates a false positive when using tools like "grep"

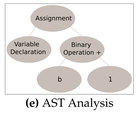
In this example, all three segments of code would produce the same textual output. Static analysis is required to accurately assess student's progress and avoid false positives that occur in simpler alternatives.

Lexical Analysis vs AST Analysis

Static Analysis can occur at many stages in the compilation process. In the lexical analysis stage, we analyze a sequence of tokens. At this level of analysis, we have no insight on the structure of the program. In the Abstract Syntax Tree (AST) stage, the compiler creates a tree that represents the structure of the program.



(d) Lexical Analysis



(e) AST Analysis

By conducting analysis at a level that includes structure, we can reason about many more interesting cases. Consider the following examples:

```
for i in range(0, 10):
    for j in range(0, 10):
        print("i: ", i, " j: ", j)
```

Lexical Analysis: 2 `for` loops

AST Analysis: a `for` loop nested in a `for` loop

```
a = "the dog runs"
b = "talks"
a.replace("runs", b.replace("t","w"))
```

Lexical Analysis: 2 calls to `replace`

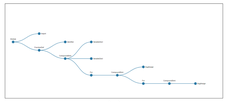
AST Analysis: a call to `replace` with a call to `replace` as an argument

Common AST Tool

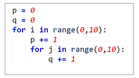
This tool provides analysis at the AST level. It is unique in that it allows instructors to easily add new static analysis tools for different programming languages. The tool builds a Common AST which captures the structural similarity of the different languages. The Common AST is a subset (or intersection) of the supported programming languages grammars.

The ASTs built by the Common AST Tool are shown below:

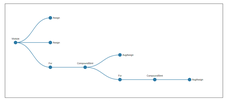
(f)



(g)



(h)

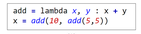


(i)

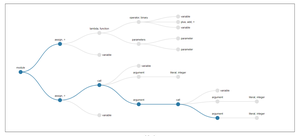
Union AST Tool

We also provide a second AST-level analysis tool to serve as a fallback when common AST matching fails. This tool internally represents syntax trees as generic tagged trees of nodes, which is easy to extend to new languages but makes general structural analysis impossible.

The dark blue colored nodes shown below indicate what is covered by the Common AST vs the light grey Union AST:



(j)



(k)

(l)

(m)

Use Cases for Static Analysis

Lexical Analysis use cases:

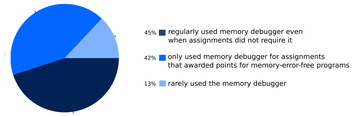
- Count `print`/assignment/multiplication/etc.
- Forbid use of `goto`/`auto`
- Verify use of `for` vs. `while`
- Verify use of Python `dict`

Use cases that can only be implemented with AST Analysis include:

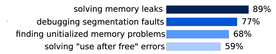
- Loop depth – naive complexity analysis
- Function calls itself – naive check for recursion
- Counting calls made on a specific object
- Forbid STL `vector::erase`
- Checking access of class members (`private`, `public`, `protected`, etc.)
- Reasoning about exception handling – confirm all exceptions are caught
- Reverse engineering UML class diagrams – design pattern check

Memory Debugging

To ensure students learn and demonstrate proper explicit memory management in their C++ programs, we use a memory debugger (Dr.Memory or Valgrind) in autograding our CSCI 1200 Data Structures course. Students receive points for correct program output/behavior and additional points if the memory debugger reports that their program has no memory errors (uninitialized memory, array bounds errors, etc.) or memory leaks. Students are encouraged to use the memory debugger on their own machine while working the homework. A survey of students found that:



Students reported that memory debugging tools were useful for:



At the end of the semester, 76% of students agreed that "The memory debugger improved my understanding of pointers, memory allocation, and memory deallocation in C++".

Code Coverage

Code coverage measures the percentage of source code lines that are executed during testing. Coverage is important as it measures the quality and completeness of the test suite as well as the conciseness of the code. Automated grading of code coverage in Submitty motivates students to think about and understand testing coverage and white-box testing in our CSCI 2600 Principles of Software course. 77% of students stated that automated grading of code coverage in Submitty has helped improve their understanding of code coverage and white-box testing. Automated grading of coverage eases the load on graders allowing them to focus on non automated grading tasks such as quality of specifications and design.

Verification with Dafny

Dafny is a programming language from Microsoft Research that supports formal specification and verification. In RPI's CSCI 2600 Principles of Software Course, we have introduced automated grading of Dafny. In previous semesters of this course, Dafny exercises were difficult to assign due to the burden of installing Dafny on local machines, the burden on graders, and the fact that Dafny's web interface is highly unreliable. In the Spring 2018 semester, a total of 225 students submitted a Dafny assignment to Submitty with a total of 645 submissions. 342 submissions of these submissions received full credit on Submitty. *Note: Students may resubmit before the assignment deadline to fix errors and improve their autograding score.*

JUnit

JUnit is the standard unit testing framework for Java developers. Support for JUnit is integrated into Submitty for students and instructors. In RPI's CSCI 2600 Principles of Software course, we use a combination of instructor-provided unit tests, student-written unit tests, and "hidden" instructor unit tests. All of these tests are run and autograded on Submitty. The immediate feedback from Submitty encourages students to think more seriously about edge cases and other testing.

Measuring Immutability

In software engineering an object is said to be "immutable" if once created, it always holds the same value. In RPI's Principles of Software Course, students are asked to write (im)mutability specifications. In preliminary work we have explored correlations between immutability and code correctness. In future work, we will integrate tools for inference of reference and method immutability for automated grading with Submitty.

Submitty `http://submitty.org`

Submitty is an open source programming assignment submission system from the Rensselaer Center for Open Source Software (RCOS), launched by the Department of Computer Science at Rensselaer Polytechnic Institute.

Related Publications

- *Analysis of Container Based vs. Jailed Sandbox Autograding Systems*
  Peveler, Maicus, Holzbauer, and Cutler, SIGCSE 2018 Poster
- *Correlation of a Flexible Late Day Policy with Student Stress and Programming Assignment Plagiarism*
  Breese, Maicus, Peveler, and Cutler, SIGCSE 2018 Poster
- *Supporting Team Submission and Peer Grading within Submitty*
  Peveler, Breese, Maicus, Aikens, Cyrus, Dinella, Anderson, Barthelmess, Lee, Montealegre, Wang, Holzbauer, Cutler, and Milanova
  SIGCSE 2018 Demo

Acknowledgments

- Rensselaer Center for Open Source (RCOS)
- Red Hat Software
- Google Summer of Code
- `https://github.com/Submitty/Submitty`