Program Analysis Tools in Automated Grading

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

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 1010/Computer Science 1 to automatically grade small in-lecture exercises.

For example, consider the following assignment:

1100 Computer Science 1 to automatically grade small in-lecture exercises.

Lexical Analysis vs AST Analysis

Lexical Analysis: 2 calls to replace
AST Analysis: a call to replace

Lexical Analysis: a call to replace
AST Analysis: a call to replace

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 (intersection) of the supported programming languages grammars. The AST built by the Common AST Tool are shown below:

AST Analysis use cases:

- Counting calls made on a specific object
- Finding STL vector::size
- Checking access to class member (private, public, protected, etc.)
- Reasoning about exception handling – confirm all exceptions are caught
- Reverse engineering UML class diagram – design pattern check

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 Abstraction of lexical analysis stage, we analyze a sequence of tokens. At this level of analysis, we have no insight on the structure of the program.

Lexical Analysis: 2 calls to replace
AST Analysis: a call to replace

For example, consider the following assignment:

Lexical Analysis: 2 calls to replace
AST Analysis: a call to replace

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

Lexical Analysis: 2 calls to replace
AST Analysis: a call to replace

Lexical Analysis: 2 calls to replace
AST Analysis: a call to replace

The tool builds a Common AST which captures the structural similarity of the different languages. The Common AST is a subset (intersection) of the supported programming languages grammars. The AST built by the Common AST Tool are shown below:

AST Analysis use cases:

- Counting calls made on a specific object
- Finding STL vector::size
- Checking access to class member (private, public, protected, etc.)
- Reasoning about exception handling – confirm all exceptions are caught
- Reverse engineering UML class diagram – design pattern check

Lexical Analysis use cases:

- Counting calls made on a specific object
- Finding STL vector::size
- Checking access to class member (private, public, protected, etc.)
- Reasoning about exception handling – confirm all exceptions are caught
- Reverse engineering UML class diagram – design pattern check

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 (intersection) of the supported programming languages grammars. The AST built by the Common AST Tool are shown below:

Use Cases for Static Analysis

- Lexical Analysis use cases:
  - Counting calls made on a specific object
  - Finding STL vector::size
  - Checking access to class member (private, public, protected, etc.)
  - Reasoning about exception handling – confirm all exceptions are caught
  - Reverse engineering UML class diagram – design pattern check

- AST Analysis use cases:
  - Counting calls made on a specific object
  - Finding STL vector::size
  - Checking access to class member (private, public, protected, etc.)
  - Reasoning about exception handling – confirm all exceptions are caught
  - Reverse engineering UML class diagram – design pattern check

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 correctness of the code. Automated grading of code coverage in Submitty motivates students to think about and understand testing coverage and when to refactor code.

In RPI’s 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 code 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 specifications and verification. In RPT’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 hurdles of installing Dafny on local machines, the hurdles on graders, and the fact that Dafny’s test interface is highly unreliable. In the Spring 2018 semester, a total of 225 students submitted a Dafny assignment to Submitty with a total of 64 submissions. 342 submissions of these submissions received full credit on Submitty. Students may student receive feedback and improve their assignment using autograding tools.

JUnit

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

Measuring Immutability

In automated grading, an object can be considered “immutable” if once created, it always holds the same value. In RPT’s Principles of Software Course, students are asked to write immutable specifications. In preliminary work, we have explored correlation between immutability and code correctness. In future work, we will integrate tools for reference of inferred and method immutability for automated grading with Submitty.