Introduction to Software Analysis

https://www.youtube.com/playlist?list=PLF3-CvSRq2SaApl3Lnu6Tu_ecsBr94543

“We have as many testers as we have developers.
And testers spend all their time testing and developers spend half their time testing.
We're more of a testing a quality software organization than we're a software organization.” - Bill Gates

- Learn **methods to improve software quality**:
  - reliability, security, and performance, etc.
- Build **specialized tools for software diagnosis and testing.**
  - An example task is systematically testing an Android application in various end user scenarios.

War stories

- **Ariane rocket disaster of 1996**
  - Roughly 40 seconds after the launch, the rocket reaches an altitude of 2.5 miles, but then it abruptly changes course and triggers a self destruct mechanism, destroying its payload of expensive scientific satellites. So why did this happen and what was the aftermath of this disaster?
  - Caused **due to a numeric overflow error**: attempt to fit a 64-bit format data in 16-bit space.
  - The number was too big to fit and resulted in an overflow error. This error was misinterpreted by the rocket's onboard computer as a signal to change the course of the rocket. This failure translated into millions of dollars in lost assets and several years of setbacks for the Ariane program.
- **Security vulnerabilities**
  - Exploits of errors in programs
  - Widespread problem:
    - Moonlight maze (1996)
    - Code red (2001)
    - Titan rain (2003)
    - Stuxnet worm (2010)
  - Getting worse … (e.g., the advent of smartphones)

Program analysis

- Body of work to automatically discover useful facts about programs
- Broadly classified into 3 kinds:
  - Dynamic (run-time)
  - Static (compile-time)
  - Hybrid (combines dynamic + static)

Dynamic program analysis

- Infers facts of program by monitoring its runs
- Well-known dynamic analysis tools:
  - Array bound checking -> Purify
  - Memory leak detection -> Valgrind
  - Data race detection -> Eraser
  - Finding likely invariants -> Daikon

Static program analysis

- Infers facts of program by inspecting its code
- Well-known static analysis tools:
  - Suspicious error patterns -> Lint, FindBugs, Coverity
  - Checking API usage rules -> Microsoft SLAM
  - Memory leak detection -> Facebook Infer
  - Verifying invariants -> ESC/Java

Program invariants

- An invariant at the end of the following program is (z ==c) for some constant c.

  What is c?

```
int p(int x) { return x * x; }
void main() {
      int z;
      if (getc() == 'a')
            z = p(6) + 6;
      else
            z = p(-7) - 7;
}
```

- The value of c is 42.

```
int p(int x) { return x * x; }
void main() {
      int z;
      if (getc() == 'a')
            z = p(6) + 6;
      else
            z = p(-7) - 7;
      if (z != 42)
            disaster();
}
```

- The invariant we just discovered is a useful fact for proving that this program can never call disaster.

Discovering invariants using dynamic analysis

- The above program has only 2 parts. But in general, programs have loops or recursion, which can lead to arbitrarily many paths. Since dynamic analysis discovers information by running the program a finite number of times, it cannot in general, discover information that requires observing an unbounded number of paths.

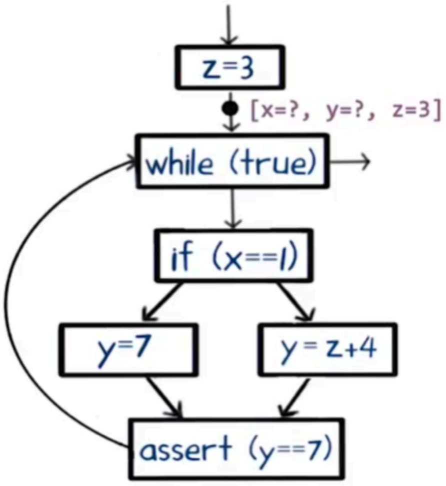
  As a result, a dynamic analysis tool, like Daikon, can at best detect **likely invariants.**
- From any run of the above program, Daikon can at best conclude that z = 42 is a likely invariant. It cannot prove that z will always be 42, and that the call of disaster can never happen.
  - (z == 42) might be an invariant
  - (z == 30) is definitely not an invariant
  - to conclusively determine that z = 42 is an invariant and therefore, showing that the program will never call disaster, we need static analysis.

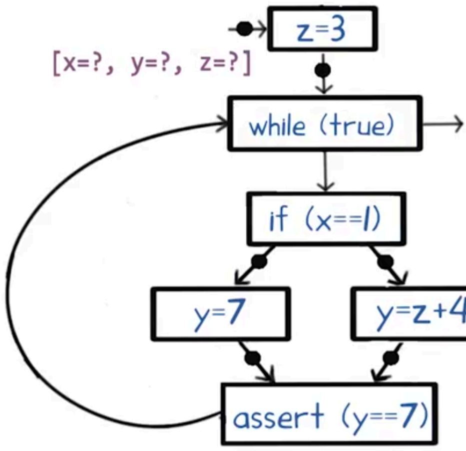
Discovering invariants using static analysis

  - (z == 42) ~~might be~~ is definitely an invariant
  - (z == 30) is definitely not an invariant
- By inspecting the source code to determine that the constant c has value 42, static analysis can therefore show at compile time that the program will never call disaster at run time.

Terminology

- Static analysis typically operates on a suitable intermediate representation of the program. One such representation is a **control-flow graph**. It is a graph that summarizes the flow of control in all possible runs of the program. Each node in the graph, corresponds to a unique statement in the program. And each edge outgoing from a node, denotes a possible successor of that node, in some execution.



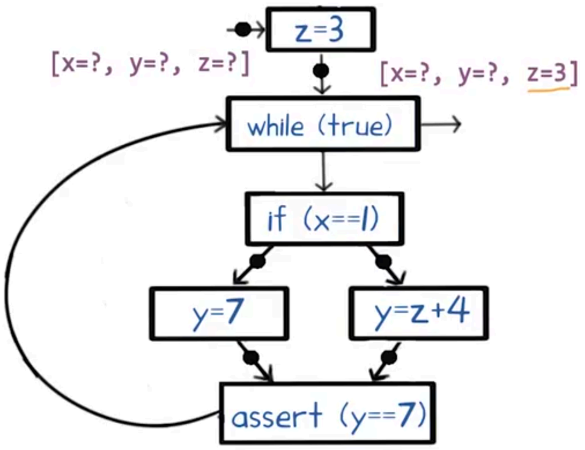


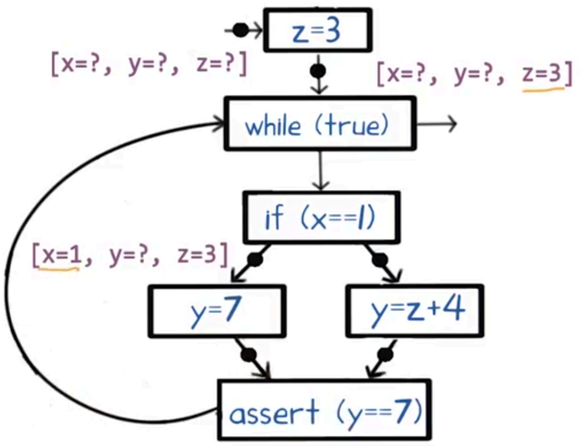
- **Abstract vs. concrete states**
  - Abstract state: tracks the constant values of the 2 variables in the above program.
  - Concrete state: tracks the actual values in a particular run.
  - Since static analysis does not run the program, it does not operate directly over concrete states.
    Instead, it operates over abstract states, each of which summarizes a set of concrete states.
    As a result of this summarization, the static analysis may fail to accurately represent the value of a variable in an abstract state.
    While this ensures the **termination** of the static analysis, even for programs with an unbounded number of paths, it can also lead the static analysis to miss variables that have a constant value.
    For this reason, we say that the **static analysis sacrifices completeness**.
    Conversely, whenever the analysis concludes that a variable has a constant value, this conclusion is indeed correct in all runs of the program. For this reason, we say that the **static analysis is sound**.

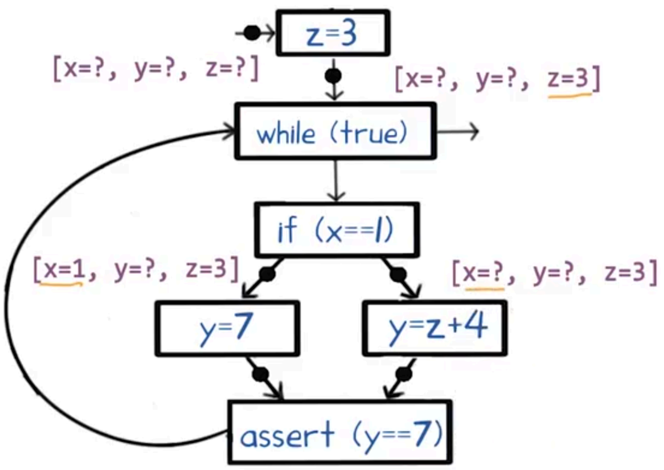
Example of static analysis problem

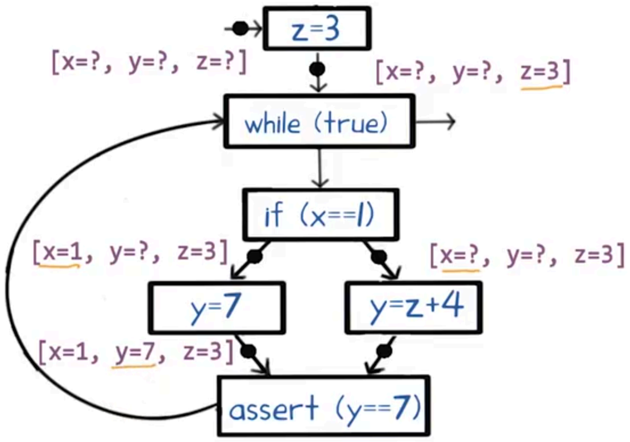
```
void main() {
      z = 3;
      while (true) {
            if (x == 1)
                  y = 7;
            else
                  y = z + 4;
            assert (y == 7);
      }
}
```

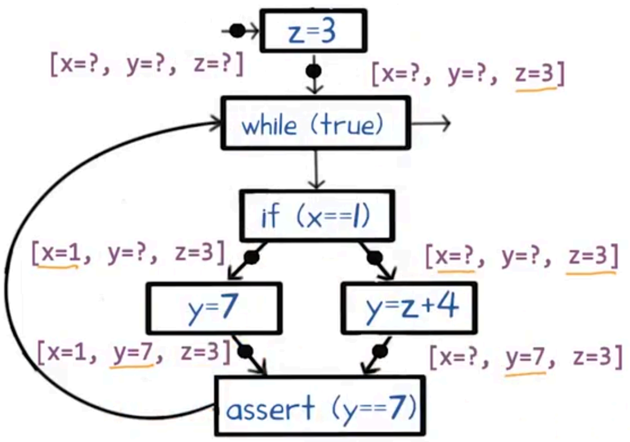
- Static analysis discovers that the variable y has the constant value 7 at the exit of this program. We will use a common static analysis method called **iterative approximation.**
- At each step, the analysis updates its knowledge about the values of the 3 variables at each program point. The analysis does this update based upon the information that it has inferred at the immediate predecessors of that program point.
- For instance, after the statement that assigns 3 to z, the analysis knows that the value of z is the constant 3.









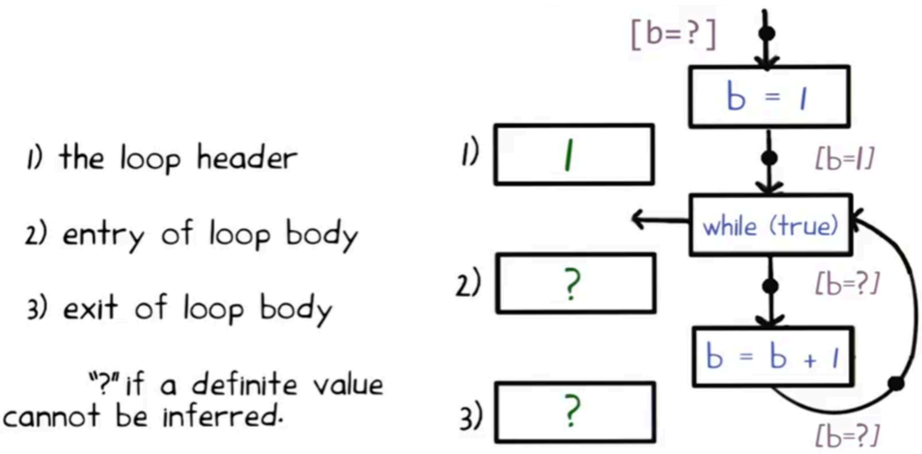


- At this point, the analysis has concluded that at each immediate predecessor of the assertion, the value of y is 7. It thereby concludes that the value of y in the assertion must be 7, and therefore that this assertion is valid.
- The term iterative approximation implies that in general, **the analysis might need to visit the same program point multiple times.**

  This is because of the presence of loops, which can require the analysis to update facts that were previously inferred by the analysis at the same program point.

2nd example of iterative approximation

- The final value of variable b that the analysis infers at the corresponding point in the following program are in the boxes bellow:



Dynamic vs static analysis

	Dynamic	**Static**
Cost	Proportional to program's execution time	Proportional to program's size
Effectiveness	Unsound (may miss errors)	Incomplete (may report spurious errors)

We say that a **dynamic analysis is unsound.** In other words, **it may produce false negatives.**

Static analysis, on the other hand does not miss errors but it may report spurious issues.

We say that **static analysis is incomplete**. In other words **it may produce false positives.**

Undecidability of program properties

- **Can program analysis be sound and complete?** (No false negatives, no false positives)
  - Not if we want it to terminate!
- Questions like "Is a program point reachable on some input?" are undecidable.
- Designing a program analysis is an art that involves striking a suitable **tradeoff** between domination **soundness and completeness.**
  - This tradeoff is typically dictated by the consumer of the program analysis.

Who needs program analysis?

- 3 primary consumers of program analysis:
  - Compilers
  - Software quality tools
  - Integrated development environments (IDEs)

Compilers

- Bridge between high-level languages and architectures
- Use program analyses to generate efficient code

```
int p(int x) { return x * x; }
void main(int arg) {
      int z;
      if (arg != 0)
            z = p(6) + 6;
      else
            z = p(-7) - 7;
       print(z);
}
```

We saw earlier how static analysis can discover the program invariant z == 42, at the end of the program above.

A compiler can use this invariant to simplify the program:

```
int p(int x) { return x * x; }
void main() {
       print(42);
}
```

It is easy to see that this simplified program, is more efficient than the original program, it runs faster, it is more energy efficient, and it is smaller in size.

Software quality tools

- Primary focus in the following courses
- Tools for testing, debugging and verification
- Use program analysis for:
  - Finding programming errors
  - Proving program invariants
  - Generating test cases
  - Localizing causes of errors

```
int p(int x) { return x * x; }
void main() {
      int z;
      if (getc() == 'a')
            z = p(6) + 6;
      else
            z = p(-7) - 7;
      if (z != 42)
            disaster();
}
```

The invariant z = 42 discovered for the above program by static analysis could be used by a program verification tool to prove that the program can never call disaster.

Integrated development environments (IDEs)

- Examples: Eclipse and Microsoft Visual Studio
- Use program analysis to help programmers:
  - Understand programs
  - Refactor programs
  - Restructuring a program without changing its external behavior.
- Useful in dealing with large, complex programs

What have we learned?

- What is a program analysis?
- Dynamic vs. static analysis: pros and cons
  - dynamic analysis works by running the program, whereas static analysis works by inspecting the program's code.
- Program invariants
- Iterative approximation method for static analysis
- Undecidability -> program analysis cannot ensure termination + soundness + completeness.
- Who needs program analysis?

Notă[1]