Overview

- What is static analysis?
- Examples of some techniques
- Tradeoffs
- Commercial static analyzers
- Use in Assurance
What Does This Program Do?

- **Context**: we have developed a program and want some evidence about what it does and doesn’t do
  - I’ll call it a program, even though it’s probably an embedded system with multiple software components
    - That makes use of systems software and libraries
    - And interacts with hardware
  - We’ll come to these complexities later

- **Evidence** is a pretty strong notion: intended for assurance
  - **Never** does certain things
    - e.g., a runtime exception
  - **Always** does a certain thing
    - e.g., delivers a good value to the actuator

Weaker notions can be useful for bug finding
Evidence About Program Behavior

• One approach is **testing**

• We generate many tests and observe the program in execution
  o We are looking at the real thing—that’s good
  o But how can we get **evidence** for **always** and **never**?
  o Usually some notion of **coverage**, but it falls short of evidence

• Let’s look at an example
The Bug That Stopped The Zunes

Real time clock sets `days` to number of days since 1 Jan 1980

```c
year = ORIGINYEAR; /* = 1980 */

while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        } else... loops forever on last day of a leap year
    } else {
        days -= 365;
        year += 1;
    }
}
```

Coverage-based testing will find this

John Rushby Static Analysis for Assurance: 5
A Hasty Fix

```java
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 365) {
            days -= 366;
            year += 1;
        }
    } else {
        days -= 365;
        year += 1;
    }
}
```

- Fixes the loop but now `days` can end up as zero
- Coverage-based testing might not find this
- Boundary condition testing would
- But I think the point is clear…

John Rushby  
Static Analysis for Assurance: 6
The Problem With Testing

- Is that it only samples the set of possible behaviors
- And unlike physical systems (where many engineers gained their experience), software systems are discontinuous
- There is no sound basis for extrapolating from tested to untested cases
- So we need to consider all possible cases... how is this possible?
- It’s possible with symbolic methods
- Cf. $x^2 - y^2 = (x - y)(x + y)$ vs. $5*5-3*3 = (5-3)*(5+3)$
- Static Analysis is about totally automated ways to do this
The Zune Example Again

```c
[days > 0]
while (days > 365) {
    [days > 365]
    if (*) {
        [days > 365]
        if (days > 365) {
            [days > 365]
            days -= 366;  [days >= 0]
            year += 1;
        }
    }
    } else {
        [days > 365]
        days -= 365;     [days > 0]
        year += 1;
    }
}
[days >= 0 and days <= 365]
```

John Rushby

Static Analysis for Assurance: 8
Approximations

- We were lucky that we could do the previous example with full symbolic arithmetic.

- Usually, the formulas get bigger and bigger as we accumulate information from loop iterations (we’ll see an example later).

- So it’s common to approximate or abstract information to try and keep the formulas manageable.

- Here, instead of the natural numbers 0, 1, 2, ..., we could use:
  - zero, small, big
  - Where big abstracts everything bigger than 365, small is everything from 1 to 365, and zero is 0.
  - Arithmetic becomes nondeterministic.
    - e.g., small + small = small | big
The Zune Example Abstracted

[days = small | big]

while (days = big) {
    [days = big]
    if (*) {
        [days = big]
        days -= big;
        [days = big | small | zero]
        year += 1;
    }
}

} else {
    [days = big]
    days -= small;
    [days = big | small]
    year += 1;
}

}

[days = small | zero]
The Zune Example Abstracted Again

Suppose we abstracted to \{\text{negative, zero, positive}\}

\[\text{[days = positive]}\]
while (days = positive) { \[\text{[days = positive]}\]
  if (*) { \[\text{[days = positive]}\]
    if (days = positive) { \[\text{[days = positive]}\]
      days -= positive; \[\text{[days = negative | zero | positive]}\]
      year += 1;
    } \[\text{[days = negative | zero | positive]}\]
  } \[\text{[days = negative | zero | positive]}\]
} else { \[\text{[days = positive]}\]
  days -= positive; \[\text{[days = negative | zero | positive]}\]
  year += 1;
} \[\text{[days = negative | zero]}\]

We’ve lost too much information: have a \text{false alarm} that \text{days} can go negative (pointer analysis is sometimes this crude)
We Have To Approximate, But There’s A Price

- It’s no accident that we sometimes lose precision
- **Rice’s Theorem** says there are inherent limits on what can be accomplished by automated analysis of programs
  - Sound (miss no errors)
  - Complete (no false alarms)
  - Automatic
  - Allow arbitrary (unbounded) memory structures
  - Final results

Choose at most 4 of the 5
Sound approximations include all the behaviors and reachable states of the real system, but are easier to compute.
But Sound Approximations Come with a Price

May flag an error that is unreachable in the real system: a false positive, or false alarm.

John Rushby

Static Analysis for Assurance: 14
Unsound Approximations Come with a Price, Too

Can miss real errors: a false negative
Predicate Abstraction

- The Zune example used data abstraction
  - A kind of abstract interpretation
- Replaces variables of complex data types by simpler (often finite) ones
  - e.g., integers replaced by \{negative, zero, positive\}
- But sometimes this doesn’t work
  - Just replaces individual variables
  - Often its the relationship between variables that matters
- Predicate abstraction replaces some relationships (predicates) by Boolean variables
Another Example

start with r unlocked
do {
  lock(r)
  old = new
  if (*) {
    unlock(r)
    new++
  }
}
while old != new
want r to be locked at this point
unlock(r)
Abstracted Example

The significant relationship seems to be \texttt{old == new}

Replace this by \texttt{eq}, throw away \texttt{old} and \texttt{new}

\[
\begin{align*}
&\text{[!locked]} \\
&\text{do } \\
&\text{lock(r)} \quad \text{[locked]} \\
&\text{eq = true} \quad \text{[locked, eq]} \\
&\text{if (*)} \\
&\text{unlock(r)} \quad \text{[!locked, eq]} \\
&\text{eq = false} \quad \text{[!locked, !eq]} \\
&\text{}}} \\
&\text{[locked, eq] or [!locked, !eq]} \\
&\text{while not eq} \\
&\text{[locked, eq]} \\
&\text{unlock(r)}
\end{align*}
\]
Yet Another Example

```
z := n; x := 0; y := 0;
while (z > 0) {
    if (*) {
        x := x+1;
        z := z-1;
    } else {
        y := y+1;
        z := z-1;
    }
}
```

want $y \neq 0$, given $x \neq z$, $n > 0$

- The invariant needed is $x + y + z = n$
- But neither this nor its fragments appear in the program or the desired property
Let’s Just Go Ahead

First time into the loop
\[ n > 0 \]
\[ z := n; x := 0; y := 0; \]
while \((z > 0)\) \{  
  \[ x = 0, y = 0, z = n \]
  if (*) \{  
    x := x+1;
    z := z-1;  \[ x = 1, y = 0, z = n-1 \]
  } else \{  
    y := y+1;
    z := z-1;  \[ x = 0, y = 1, z = n-1 \]
  \} \[ x = 1, y = 0, z = n-1 \text{ or } x = 0, y = 1, z = n-1 \]
\}

Next time around the loop we’ll have 4 disjuncts, then 8, then 16, and so on

This won’t get us anywhere useful

John Rushby
Widening the Abstraction

- We could try **eliminate disjuncts**
- Look for a **conjunction** that is **implied** by each of the disjuncts
- One such is \([x+y = 1, z = n-1]\)
- Then we’d need to do the same thing with \([x+y = 1, z = n-1]\) or \([x = 0, y = 0, z = n]\)
- That gives \([x + y + z = n]\)

- There are techniques that can do this automatically
- This is where a lot of the research action is
Tradeoffs

- We’re trying to guarantee absence of errors in a certain class
- Equivalently, trying to verify properties of a certain class
- Terminology is in terms of finding errors
  - **TP** True Positive: found a real error
  - **FP** False Positive: false alarm
  - **TN** True Negative: no error, no alarm—OK
  - **FN** False Negative: missed error
- Then we have
  - **Sound**: no false negatives
  - **Recall**: $\frac{TP}{TP+FN}$ measures how (un)sound
    $TP+FN$ is number of real errors
  - **Complete**: no false alarms
  - **Precision**: $\frac{TP}{TP+FP}$ measures how (in)complete
    $TP+FP$ is number of alarms
Tradeoff Space

- Basic tradeoff is between soundness and completeness
- For assurance, we need soundness
  - When told there are no errors, there must be none
    So have to accept false alarms
- But the main market for static analysis is bug finding in general-purpose software, where they aim merely to reduce the number of bugs, not to eliminate them
- Their general customers will not tolerate many false alarms, so tool vendors give up soundness
- Will consider the implications later
- Other tradeoffs are possible
  - Give up full automation: e.g., require user annotation
Tradeoffs In Practice

**Testing** is complete but unsound

**Spark Ada with its Examiner** is sound but not fully automatic

**Abstract Interpretation** (e.g., PolySpace) is sound but incomplete, and may not terminate

- Astrée is pragmatically complete for its domain

**Pattern matchers** (e.g. Lint, Findbugs) are not based on semantics of program execution, neither sound nor complete

- But pragmatically effective for bug finding

**Commercial tools** (e.g., Coverity, Code Sonar, Fortify, KlocWork, LDRA) are neither sound nor complete

- Pragmatically effective
- Different tools use different methods, have different capabilities, make different tradeoffs
Properties Checked

- The properties checked are usually **implicit**
  - e.g., uninitialized variables, divide by zero (and other exceptions), null pointer dereference, buffer overrun

- Much of this is compensating for deficiencies of C and C++
  - Some tools support Ada, Java, not much for MBD
  - But Mathworks has Design Verifier for Simulink

- Some tools support user-specified checks, but...

- Some tools look at **resources**
  - e.g., memory leaks, locks (not freed, freed twice, use after free)

- Some (e.g., **AbsInt**) can do **quantitative** analysis
  - e.g., worst case execution time, maximum stack height
Real Software

• It’s not enough to check individual programs

• Need information from calls to procedures, subroutines
  ○ Analyze each in isolation, then produce a procedure summary for use by others

• Need summaries for libraries, operating system calls

• Analyzer must integrate with the build process

• Must present the information in a useful and attractive way

• Much of the engineering in commercial tools goes here
So How Good Are Static Analyzers?

- Some tool licences forbid benchmarking
- Hard to get representative examples
- **NIST SAMATE** study compared several
  - Found all had strengths and weaknesses
  - Needed a combination to get comprehensive bug detection
- This was bug finding, not assurance
- Anecdotal evidence is they are very useful for general QA
- Need to be tuned to individual environment
- e.g., Astrée tuned to Airbus A380 SCADE-generated digital filters is sound and pragmatically complete
- There are papers by Raoul Jetley and others of FDA applying tools to medical device software
Possible Futures: Combination With Testing

- Automated test generation is getting pretty good
- Use a constraint solver to find a witness to the path predicate leading to a given state
  - e.g., counterexamples from (infinite) bounded model checking using SMT solvers
- So try to see if you can generate an explicit test case to manifest a real bug for each positive turned up by static analysis
- Throw away those you cannot manifest
- Aha! Next generation of tools do this
Possible Futures: Integration With Testing

- Knowledge that possible error is unreachable is information that helps refine the abstraction
- So iterate abstraction, analysis, test generation
- Either finds error or proves its absence
- Microsoft India projects (Synergy, Dash, Yogi) explore this area
- Counterexample Guided Abstraction Refinement (CEGAR) is similar
Use in Assurance

• If you are satisfied with bug finding for standard properties

• Then one or more commercial static analyzers could do a good job for you

• If you want your own properties, talk with the vendors

• If you want soundness
  - PolySpace might work, or Simulink Design Verifier
  - Talk with the vendors (some have a “dial”)
  - Roll your own
Combined Methods

- Can think of static analysis as a search for invariants
- Other tools (e.g., model checkers) can use the invariants
- The more invariants and the stronger invariants you know, the more you can verify
- Different analyzers find different (classes of) invariants
- But the tools do not disclose the invariants they find
- Cooperation would be good: an invariant bus
- There are other ways to search for candidate invariants
  - Dynamic analysis: e.g., Daikon
- Could then use static analysis to confirm these
Rolling Your Own

- There’s plenty of promising research technology around
- But engineering it into an effective toolchain is a big investment
- Because of the fundamental limitations, don’t expect a single solution
  - Future tools should support plugins, toolbus integrations
- Maybe collaborate with a research group
Combined Arguments for Assurance

- Remember: static code analysis is just for code defects; says nothing about whether code meets requirements
- Standards vs. argument-based safety/assurance cases
- Multi-legged arguments
  - e.g., static analysis plus testing
  - Bayesian Belief Nets (BBNs)
- Backups and monitors
  - A formally verified backup or monitor can support a claim of possible perfection (e.g., 0.999 perfection)
  - This is conditionally independent of the reliability claim for the main system (e.g., 0.999 reliable)
  - Can multiply these together: system reliability 0.999999
The End