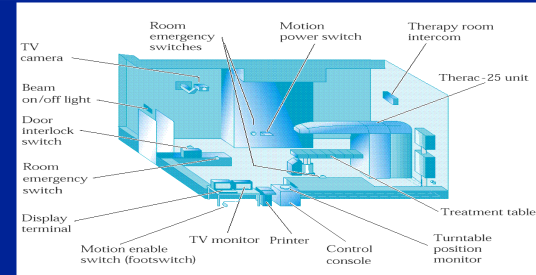
Software Engineering:
Where are we?
And where do we go from here?

V22.0474-001 Software Engineering

Lecture 23

Clark Barrett
New York University
4/15/2008

Therac-25



- Between 1985 and 1987, at least 6 accidental radiation overdoses were administered.
- All the victims were injured, and 3 of them later died.

Ariane 5 Rocket

- On June 4, 1996, an unmanned Ariane 5 rocket launched by the European Space Agency exploded just 40 seconds after its lift-off.
- Value of rocket and cargo: $500 million

Blackout

- In August, 2003, the largest blackout in our country's history occurred.
- Estimated cost to New York City alone: $1.1 billion.

What do these events have in common?

Caused by Software Bugs!

- Each of the overdoses from the Therac-25 was the result of a bug in the controlling software.
- The Ariane 5 explosion was the result of an unsafe floating point to integer conversion in the rocket's software system.
- A software bug caused an alarm system failure at FirstEnergy in Akron, Ohio. An early response to those alarms would likely have prevented the blackout.

More Evidence of Software Unreliability

Top Oxymoron from OxymoronList.com:
Microsoft Works

Thought Questions

- When we build a bridge or a building, we don't expect it to crumble and have to be rebuilt twice a week. Why is software so much less reliable than bridges or buildings?
- Do you have the knowledge and skills you need to create quality software?
- What have you learned in this class that can help?
- What tools and techniques do you think future software engineers will use to create more reliable systems?

Formal Verification

- "[Formal] software verification … has been the Holy Grail of computer science for many decades" – Bill Gates
- Formal verification techniques can be used to *prove* that a piece of software is correct.
- There are still many challenges to making FV practical, but there are also some success stories.

Outline

- What is Formal Verification?
- Model Checking
- Theorem Proving
- Systems and Tools

What is Formal Verification?

- Create a mathematical model of the system
  - An inaccurate model can introduce or mask bugs.
  - Fortunately, this can often be done automatically.
- Specify formally what the properties of the system should be
- Prove that the model has the desired properties
  - Much better than any testing method
  - Covers all possible cases
  - This is the hard part
- There are a variety of tools and techniques

Proof techniques

- Model Checking
  - Typically relies on low-level Boolean logic
  - Proof is fully automatic
  - Does not scale to large systems
- Theorem Proving
  - Typically uses more expressive logic (higher order logic)
  - Proof is manually directed
  - Unlimited scalability
- Advanced techniques combine elements of both

Outline

- What is Formal Verification?
- Model Checking
- Theorem Proving
- Systems and Tools

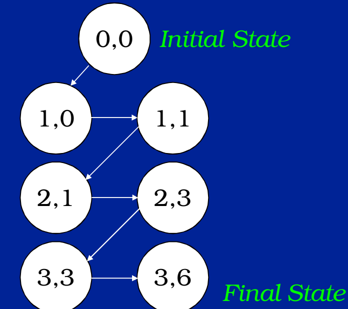
Formal Models

- Typically, a formal model is a graph in which each vertex represents a *state* of the program, and each edge represents a *transition* from one state to another.
- Consider this simple program:
- The states of this program are all possible pairs of the variables x and y.
- Fortunately, we can restrict our attention to the *reachable* states.

```
int x, y;
x = 0;
y = 0;
while (x < 3) {
    x++;
    y = y + x;
}
```

Reachable states

- Typically, a formal model is a graph in which each vertex represents a *state* of the program, and each edge represents a *transition* from one state to another.

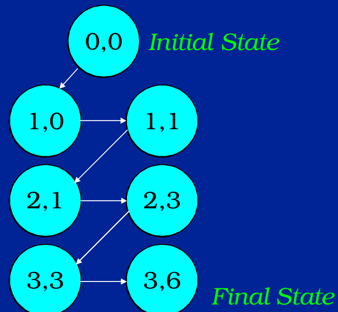



```
int x, y;
x = 0;
y = 0;
while (x < 3) {
    x++;
    y = y + x;
}
```

Checking Properties

- We can check a property by verifying that it is *true* in every reachable state. If the property is *false*, then there is a bug.

$x \geq 0$



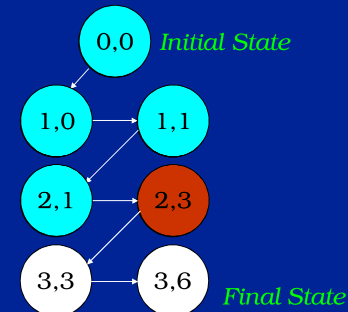

```
int x, y;
x = 0;
y = 0;
while (x < 3) {
    x++;
    y = y + x;
}
```

Checking Properties

- We can check a property by verifying that it is *true* in every reachable state. If the property is *false*, then there is a bug.

$x \geq 0$

$x \geq y$




```
int x, y;
x = 0;
y = 0;
while (x < 3) {
    x++;
    y = y + x;
}
```

Checking Properties

- We can check a property by verifying that it is *true* in every reachable state. If the property is *false*, then there is a bug.

$x \geq 0$

$x \geq y$

$y = \frac{x(x+1)}{2}$

0,0 *Initial State*

1,0 → 1,1

2,1 → 2,3

3,3 → 3,6 *Final State*

```
int x, y;
x = 0;
y = 0;
while (x < 3) {
    x++;
    y = y + x;
}
```

Checking Properties

- We can check a property by verifying that it is *true* in every reachable state. If the property is *false*, then there is a bug.

$x \geq 0$

$x \geq y$

$y = \frac{x(x+1)}{2}$

$PC = 2 \rightarrow y = \frac{x(x+1)}{2}$

2,0,0 *Initial State*

3,1,0 → 2,1,1

3,2,1 → 2,2,3

3,3,3 → 2,3,6 *Final State*

PC	
	int x, y;
0	x = 0;
1	y = 0;
	while (x < 3) {
2	x++;
3	y = y + x;
	}

State Explosion Problem

- In practice, models of real programs would have too many states to modelcheck.
- There are a number of techniques which can help:
  - Abstraction
  - Decomposition
  - Symbolic model checking
- Ultimately, model checking alone cannot prove properties of large programs.

Outline

- What is Formal Verification?
- Model Checking
- Theorem Proving
- Systems and Tools

Theorem Proving

- Theorem proving relies on human ingenuity and symbolic manipulation to prove that a program satisfies some property.
- Typically, proving a single property about a program will require proving many other properties as well.
- One approach is to annotate the program with theorems to be proved (also called *invariants* or *assertions*), and then prove that each theorem really does hold.

Theorem Proving

- Consider a slightly modified version of our simple program from before: this time there are many more reachable states.
- Suppose we wish to prove that at the end of the program, $y=x(x+1)/2$.
- We can annotate the end of the program with this property and work backwards from there.

```
int x, y;
x = 0;
y = 0;
while (x < 30) {
    x++;
    y = y + x;
}
```

Theorem Proving

- To show this property, we must look at the two possible previous locations in the program.
- For these two locations, it will be sufficient to prove that the program either doesn't end or that the property holds.
- With a bit of insight, we can see that these two formulas are more complicated than necessary. We can *strengthen* a formula by replacing it with a formula which implies it.

```
int x, y;
x = 0;
y = 0;
x<30 ∨ y=x(x+1)/2
while (x < 30) {
    x++;
    y = y + x;
    x<30 ∨ y=x(x+1)/2
}
y=x(x+1)/2
```

Theorem Proving

- To show this property, we must look at the two possible previous locations in the program.
- For these two locations, it will be sufficient to prove that the program either doesn't end or that the property holds.
- With a bit of insight, we can see that these two formulas are more complicated than necessary. We can *strengthen* a formula by replacing it with a formula which implies it.

```
int x, y;
x = 0;
y = 0;
x<30 ∨ y=x(x+1)/2
while (x < 30) {
    x++;
    y = y + x;
    y=x(x+1)/2
}
y=x(x+1)/2
```

Theorem Proving

- What is the condition that will guarantee the green assertion after executing y = y + x ?
- To find out, we imagine trying to prove the green condition using primed variables to represent the value after y = y + x and unprimed variables for the value before:

$$(?) \wedge y'=y+x \wedge x'=x \rightarrow y'=x'(x'+1)/2$$
$$(?) \rightarrow y+x=x(x+1)/2$$
$$(?) \rightarrow y=x(x-1)/2$$

```
int x, y;
x = 0;
y = 0;
x=0
while (x < 30) {

    x++;
    ?
    y = y + x;
    y=x(x+1)/2
}
y=x(x+1)/2
```

Theorem Proving

- Now we must find an assertion which is implied by the loop-end condition and the pre-loop condition, and which implies the green condition after executing x++ .
- To do this, we first must strengthen the pre-loop condition.

```
int x, y;
x = 0;
y = 0;
x=0 ∧ y=0
while (x < 30) {
    ?
    x++;
    y=x(x-1)/2
    y = y + x;
    y=x(x+1)/2
}
y=x(x+1)/2
```

Theorem Proving

- Now we must find an assertion which is implied by the loop-end condition and the pre-loop condition, and which implies the green condition after executing x++ .
- To do this, we first must strengthen the pre-loop condition.
- We finish with a set of assertions, each of which can be proven to follow from the annotations at all possible previous points in the program.

```
int x, y;
x = 0;
y = 0;
x=0 ∧ y=0
while (x < 30) {
    y=x(x+1)/2
    x++;
    y=x(x-1)/2
    y = y + x;
    y=x(x+1)/2
}
y=x(x+1)/2
```

Theorem Proving

- Notice that the final set of conditions *does not depend* on the number of loop iterations.
- In fact, this same proof can be used regardless of what the loop condition is.
- This is one advantage theorem proving has over model checking.

```
int x, y;
x = 0;
y = 0;
x=0 ∧ y=0
while (x < 30) {
    y=x(x+1)/2
    x++;
    y=x(x-1)/2
    y = y + x;
    y=x(x+1)/2
}
y=x(x+1)/2
```

Theorem Proving

- Each proof from the assertion before a statement to the assertion after the statement is called a *verification condition*.
- Verification conditions can be proved using an *automated theorem prover*.
- However, coming up with the assertions usually requires human guidance and can be quite challenging.

Outline

- What is Formal Verification?
- Model Checking
- Theorem Proving
- Systems and Tools

Model Checking

- SMV
  - Model checker for finite state systems
  - Based on extremely efficient data structures for representing Boolean logic
  - Very successful for hardware
- SPIN
  - Model checker for parallel systems
  - Limited to small state spaces

Theorem Proving

- Some Interactive Theorem Provers
  - PVS
  - HOL
  - ACL2
  - Isabelle
- Some automated domain-specific theorem provers
  - Simplify
  - ICS
  - CVC

Extended Static Checker (ESC)

- Systems Research Center at HP
  - (formerly Compaq, formerly DEC)
- Theorem Proving approach for simple properties in Java
- User annotates code with expected invariants
- Invariants are verified using automated theorem prover Simplify

Microsoft SLAM

- Clever combination of model checking and automated theorem proving
  - An abstract program is created in which all conditions are replaced with Boolean variables
  - Resulting Boolean program is model checked
  - If model checking fails, the potential error path is checked in the original program using an automated theorem prover
- Successfully used to find bugs in Windows drivers.
  - …reducing the frequency of "Blue Screens of Death"!

Conclusions

- Formal Software Verification is starting to become practical
- Still lots of work to be done
- How can it make you a better programmer?
  - Document your code with the properties and invariants that you think should be true
  - When you modify code, convince yourself that you are not breaking any invariants
  - Learn more about formal verification!
- Hopefully, someday software will be as safe and reliable as the other objects built by engineers!