|galois|

Software Test Techniques

with formal methods

The Plan

We want to explore testing methods that are enabled or augmented by formal methods.

I. Property-based testing: Similar to traditional forms of testing but using formal properties and generated test values

II. Concolic execution: Finding out what conditions need to be met in order to reach different paths of a program

III. Bounded Model-checking: Ensuring that all possible paths of a program satisfy a specific property (or set of properties)

¬(The Plan)

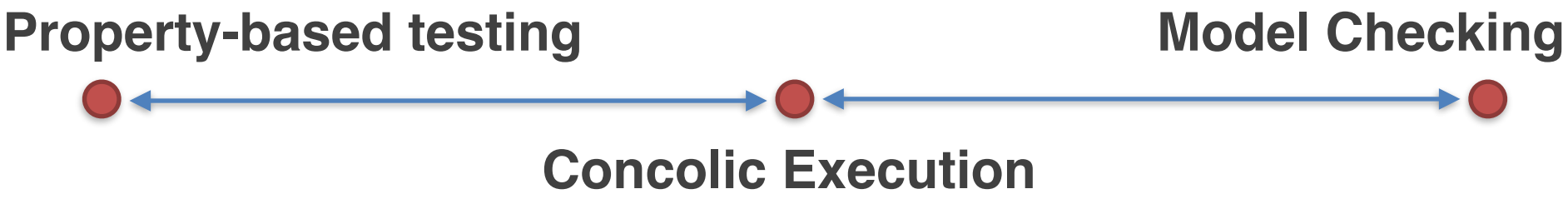
Time constraints limit what we can discuss in this talk, so we are explicitly *not* tying to

I. Cover all the theory of every technique we introduce

II. Provide a tutorial on any particular tool

The General Idea

The three techniques we're going to explore provide a range of assurance.



Property-Based Testing in 1 Slide

I. Define properties that your functions should satisfy (e.g. length before and after sorting a list should be the same)

II. System generates data (usually random) and tests the properties on that data, stopping when a counterexample is found

III. The counterexample goes through a ‘shrinking’ process in order to minimize the size/complexity of the counterexample

Benefits of PBT

Property-based testing is incredibly lightweight!

Easy to incorporate with other testing methods

Getting developers to think about properties of their code is a good stepping stone to more intricate formal methods

Unlike some of the other testing methods, counterexamples usually provide a clear path to finding a fix for the bug

QuickCheck

QuickCheck is the original property based testing tool. It is written in and for the Haskell language, but many ports to other languages exist.

Associativity is a common property of operations

```
λ -> let left a b c = (a + b) + c :: Double
λ -> let right a b c = a + (b + c) :: Double
λ -> quickCheck (\a b c -> left a b c == right a b c)
*** Failed! Falsifiable (after 2 tests and 1 shrink):
2.0
-0.3869204559163891
-0.7379227310831907
```

Success Stories

Property-based testing has been used to find many bugs.

Notable bugs:

(unnamed) car manufacturer's CAN bus

(unnamed) Database company's DBMS

See Quviq for many stories in detail

Concolic Execution in 1 Slide

I. Treat certain program variables as *symbolic variables* (i.e. they are just symbols with no 'value')

II. For each path in the program, track how the symbolic variables are used to learn what property leads to that execution path (this is called the *path condition*)

III. Path conditions can be negated to generate input values that cover new paths

IV. By repeating this process we can generate high-coverage test suites automatically

Benefits of Concolic Execution

Helps ensure that your test suite covers as much of your program as you require.

Easy to incorporate with other testing methods

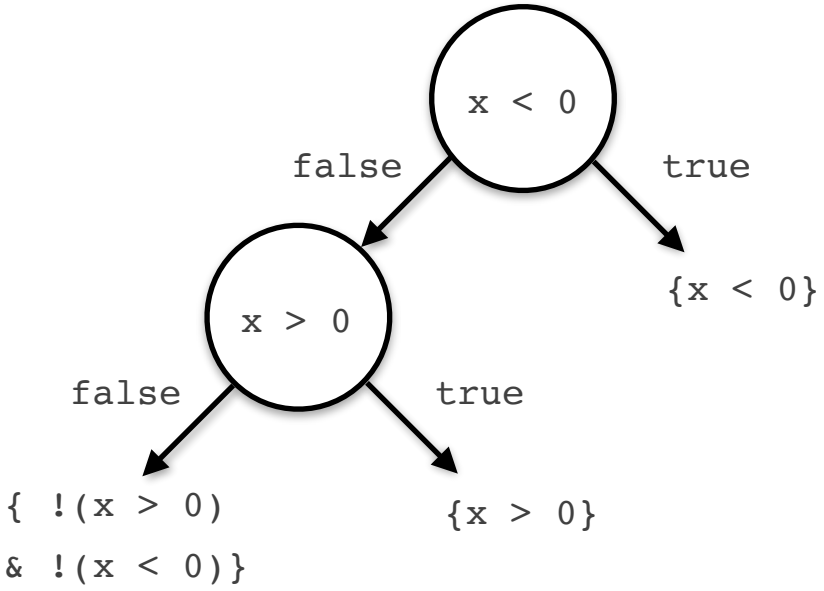
Path conditions themselves can provide interesting information about the program (e.g. bounds on lengths for data fields)

KLEE (Concolic Execution for C)

KLEE is a concolic execution framework for C built using the LLVM infrastructure.

```
int get_sign(int x) {
  if (x < 0)
    return -1;
  else if (x > 0)
    return 1;

  return 0;
}
```



What now?

From these path conditions we can test the function with input values for each path, checking to see if any of the three paths lead to an error

If you treat the input values to the program as symbolic you can generate input data for specific paths, ensuring that no errors occur

Success Stories

KLEE itself:

Used on many open-source projects, including GNU Coreutils

Big one:

Microsoft SAGE, used on all new security software at MS

Model Checking in 1 Slide

I. Simplify your program to a state machine (this is the *model*)

II. Create a specification of the required behavior using *temporal logic*

III. Check that all reachable states from Step I satisfy the specification from step II

IV. Usually applied to concurrent or distributed systems (i.e. not about individual components, but about the interaction of components)

Benefits of Model Checking

More than other techniques, Model Checking can target temporal properties such as ordering of events

Particularly well suited for decision making aspects of the program

Great for exploring the effect of program behaviors on the system as a whole

No manual proof!

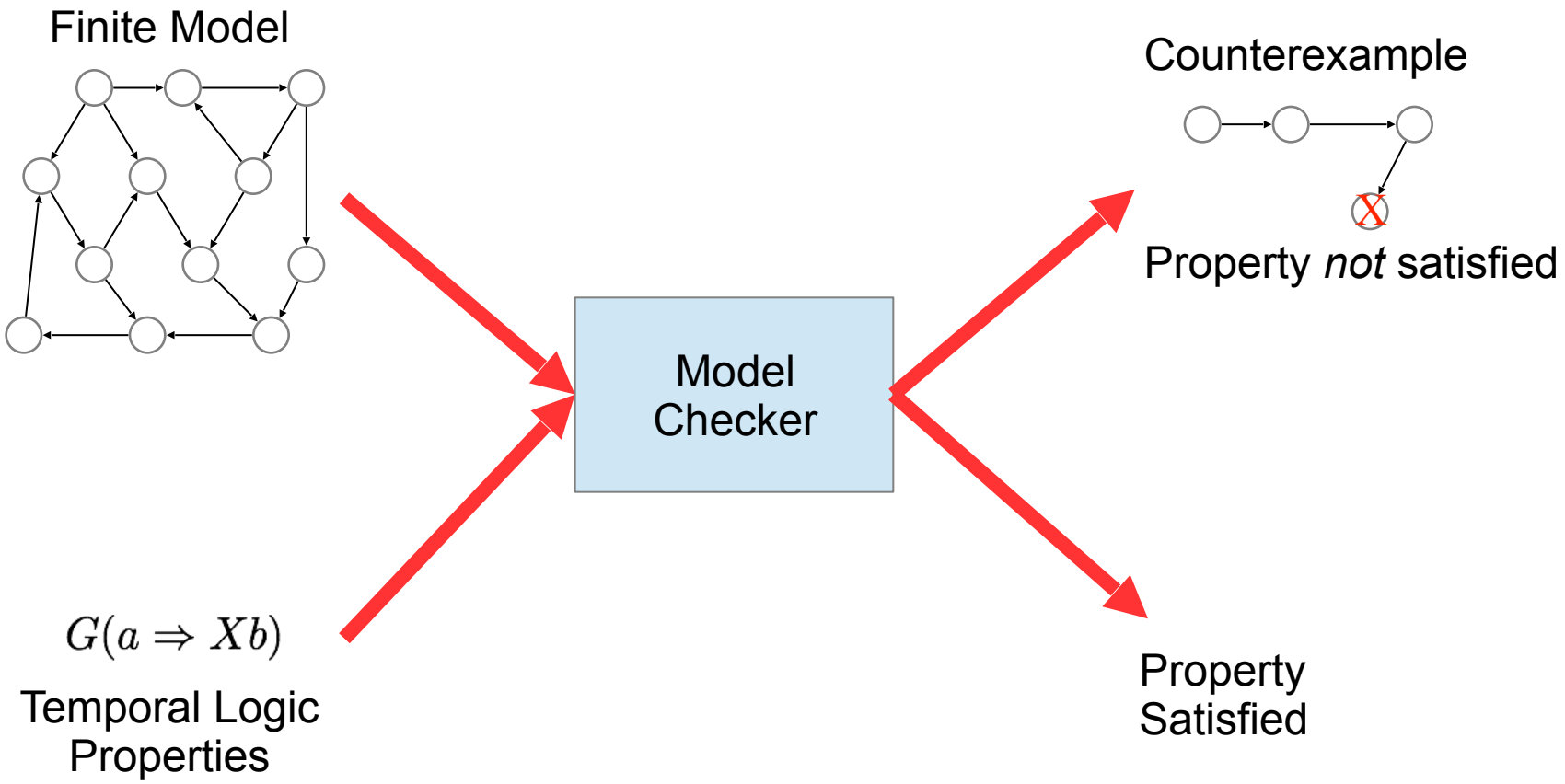
Example MC Properties

I. If the temperature exceeds 100 degrees, then the cooling subsystem is engaged within 1 minute

II. In this program a NULL pointer is *never* dereferenced

III. *After* action X, action Y *always precedes* action Z

Model Checking Workflow

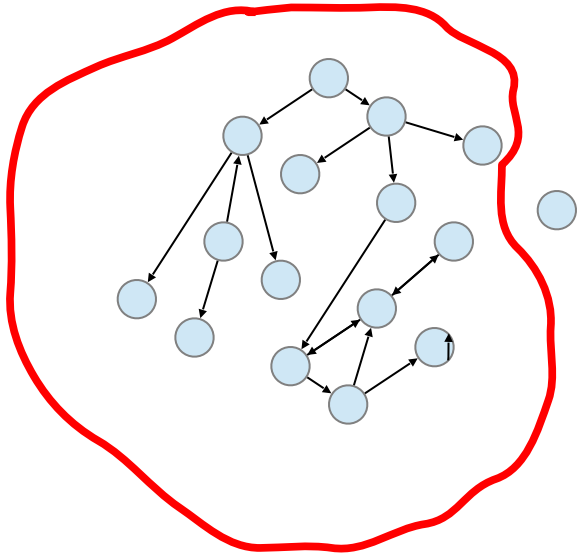


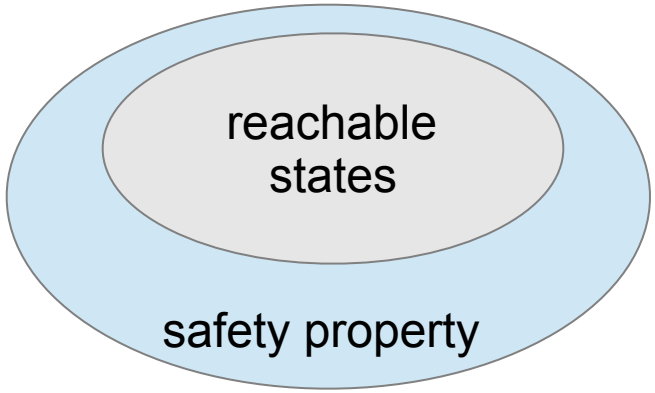
Model Checking Workflow

Another way to think about safety properties:

A safety property generalizes the reachable states

If something is true given your safety property, it has to be true in the states the system can reach





Success Stories

JPL used Model Checking for its software on NASA's Curiosity Rover

What Makes a Good Property?

Something that should hold for *all* programs (e.g. memory safety)

What Makes a Good Property?

Any good execution (even ones your program doesn't display) should satisfy the property

Any bad executions should fail the property

What Makes a Good Property?

Captures as much of the ‘good’ as possible

Contrast with unit tests which capture a single good execution

What Makes a Good Property?

Describes program's behavior on as much of the input space as possible

What Makes a *Bad* Property?

Anything that's too specific

Recap

I. PBT, CE, and MC each occupy different spaces in the tradeoff between ease of use and guarantees

II. PBT is a great technique to add in addition to already present test suites

III. CE can be used to ensure that a test suite achieves proper code coverage

IV. MC is for when you need strong guarantees over temporal properties

Tools for specific Languages

Property-based Testing		*Concolic Execution*		*Model Checking*	
Haskell	*- Quickcheck* *- SmallCheck*				
C	*- Theft* *- Quviq*	*C*	*- KLEE* *- CREST*	*C*	*- CBMC*
Java	*- JCheck* *- QC for*	*Java*	*- jCUTE* *- CATG*	*Java*	*- CBMC*
C++	*- RapidCheck*			*C++*	*- CBMC*