Formal Methods used in Software Verification

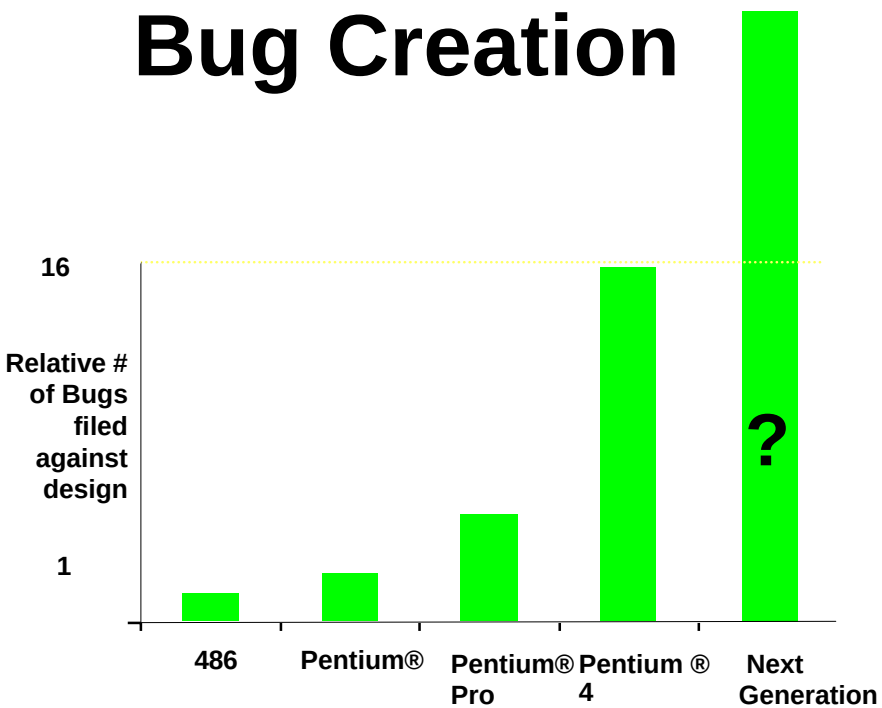
Robert T. Bauer
IBM/SWG/Rational/Beaverton
robert.bauer@acm.org

Bob Caldwell, Intel's Chief Architect for Williamette, while giving a talk about the formal verification, said that the real problems were managing the complexity - that the designs had to be scaled to what people could understand - that there were no technical barriers to building a 100 stage pipeline.

Brian Kernighan has been attributed to saying, "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are by definition, not smart enough to debug it."

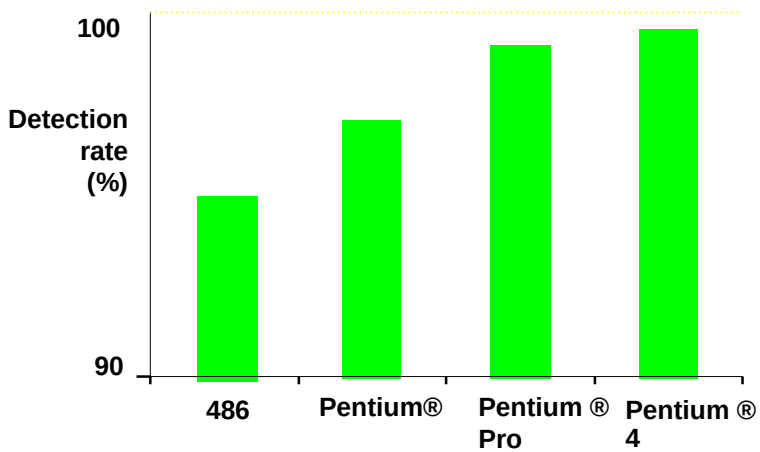
Bugs will be created faster than they can be detected

Bug Creation



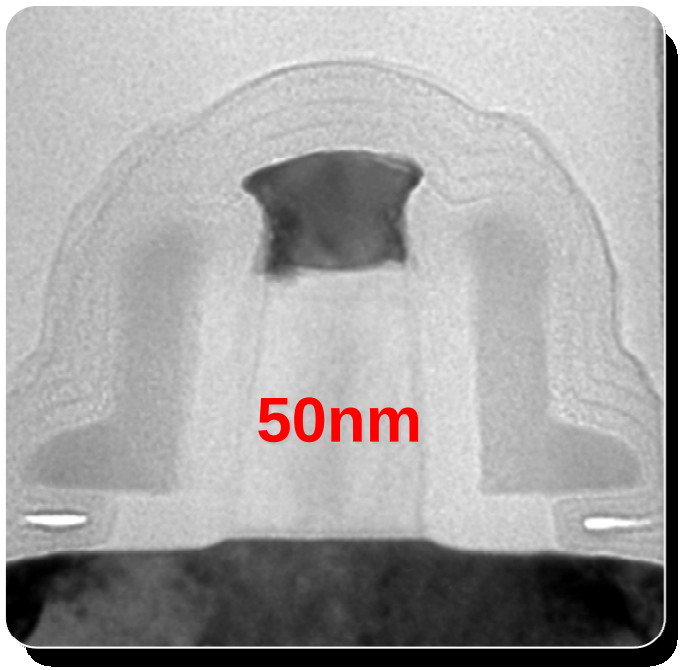
4x per lead design

Bug Detection



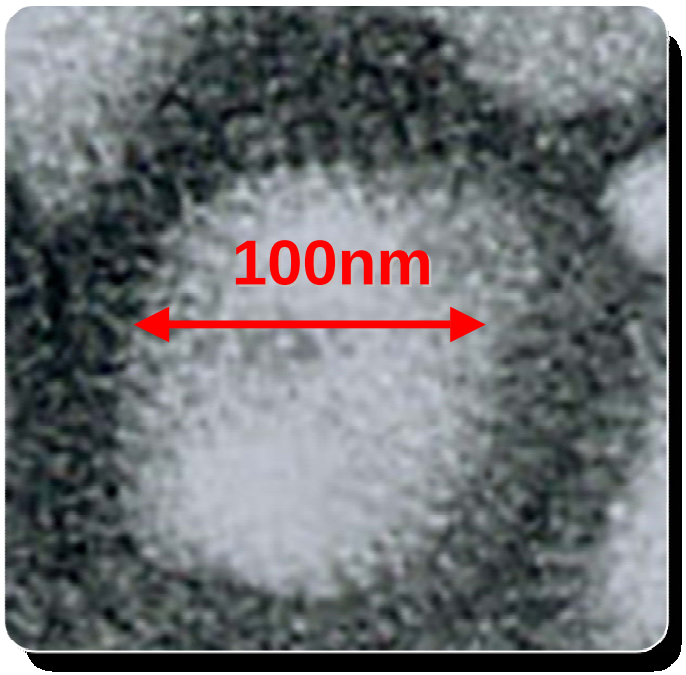
flattening

Silicon Devices Shrink to Virus Size



Transistor for 90nm Process

Source: Intel



Influenza virus

Source: CDC

Approaches

- **More Testing?**
  - **No. Development (per line of code) is getting cheaper, Testing is getting more expensive, eventually the product cost is dominated by testing.**
- **Better Engineers?**
  - **Yes, but education, training, and experience take a lot of time.**
- **Better Programming languages?**
  - **Yes, but the domain gets more complex.**
- **Better Environments?**
  - **Yes. On-Demand. Grid.**
- **Better Processes?**
  - **Yes, but that's not the focus here.**
- **Better Tools?**
  - **Yes!**

Testing Issues

- Code is bigger
- Code is being developed faster
- Middleware
- Unit testing isn’t getting better
- Testing is still given lip service
- Programs are more complex, but testing isn’t
- No career path for testers (most companies)

Testing Success’

- Organizations are buying into automation
- Use cases now common as part of the design
- Test before code paradigm (STeP, xP)
- Regression Tests
- Code coverage
- Process is being driven by testing (tell me how you test and I’ll tell you what you can deliver)

Kinds of Testing

- Demonstrative: Show that it works - demonstrate what it does
- Destructive: Show that it doesn’t work - demonstrate what it shouldn’t do
- Regression: Show that it still works (after changing something)

Complexity of Testing

- Testing is interesting because it is a provably unsolvable problem. In practice, it is “simply” intractable.
- Endless number of ways to attack the problem and all of them together don’t solve it completely.
- Testing is more complex than the design - for no one would start the design without having a sense that it would work.

Why Test?

- Experience has taught us that if we don’t test, customers will get products that don’t work.

Why Test?

- It’s a matter of culture that we can ship a product that is ridden with bugs. If the point of reference is seller-beware, the software will be different than if the perspective is buyer-beware.
- Classic Producer - Consumer risk model.

Why Formal Methods

- Only a few ways to solve big problems: Politics, Science, and Engineering.
- Political solution: Raise the cost/liability of producing/selling bad software. In the long-run, MS, Sun, etc., will support bad software liability laws - those laws will certainly act as entry-barriers.

Why Formal Methods?

- Science: Concerned with the whys - creating fundamental explanations
- Engineering: Applies science (and scientific principles) to build things

Why Formal Methods?

- If we had a “theory” about why there are defects in the code or a theory about how to create code, we could build engineering models, programming languages, etc., to reduce and/or eliminate errors.
- We don’t have good theories about software and/or software development

Why Formal Methods?

- All we have is testing - but testing is inadequate.
- For even a moderate program, we can’t test everything.
- Consider a program to add 32-bit numbers. How do you test it?

Why Formal Methods?

- If adding is a via a lookup table, you'd have to look at (2**32) ** 2 entries
- Irrespective of how the "adder" is constructed, a "good" test suite requires that many (2**64) test cases (and we haven't looked at bad input).

Why Formal Methods?

- Formal methods rely on “mathematical” proofs. And mostly, on exhaustive search of state space.
- They’re certain in a very specific sort of way.
- Very common in telecommunication (software systems).

Approaches used in FM

- Equivalence Checking (combinational logic - usually after adding test points, small FSM)
- Symbolic Execution including STE
- Model Checking (Symbolic and Explicit)
- Theorem Proving

Languages of FM

- Functional Programming
- Predicate Logic
- First Order Logic
- Higher Order Logic
- Temporal and other Modal Logics

Some Specifications

- EF (started /\ ~ready)-- possible to get to state where started holds, but ready does not.
- AG(msg_sent ==> AF msg_received) -- whenever a message is sent, it is received.
- AG(AF activated) -- a subsystem is activated infinitely often on every path
- AF(AG deadlock) -- always deadlocks

Some Specifications

- AG(floor = 3 /\ idle /\ door = closed ==> EG(floor = 3 /\ idle /\ door = closed) -- an elevator on the third floor, with ..., can remain on the third floor with ...
- AG(floor = 3 /\ idle /\ door = closed ==> EF(floor = 3 /\ ~idle /\ door = closed)-- but there is always a way to get the elevator to move

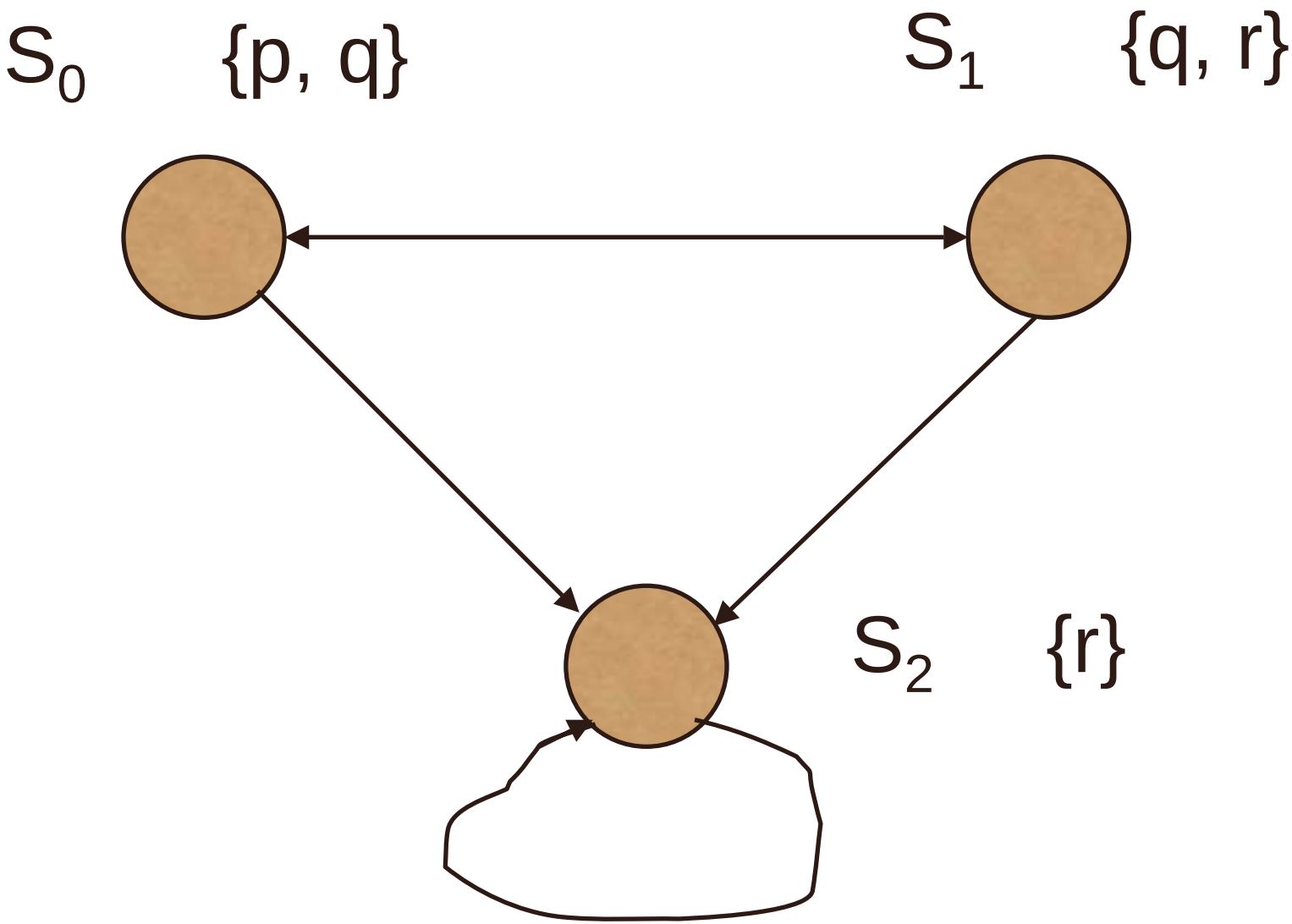
!!! BREAK !!!

Model Checking

- Language to describe the system - a framework within which to model systems
- Language to express the specification - the properties to be verified
- A methodology - we'll talk about a model-based approach that is (mostly) automatic

Model as directed graph

- d(s0, s1); d(s0, s2)
- d(s1, s0) ; d(s1, s2)
- d(s2, s2)
- L(s0): p,q
- L(s1): q, r
- L(s2): r


S_0 {p, q}
S_1 {q, r}
S_2 {r}

Concurrent software

Shared vars : G,Z

a1: x=G

a2: x=0

a3: y=1

a4: Z=2

a5:

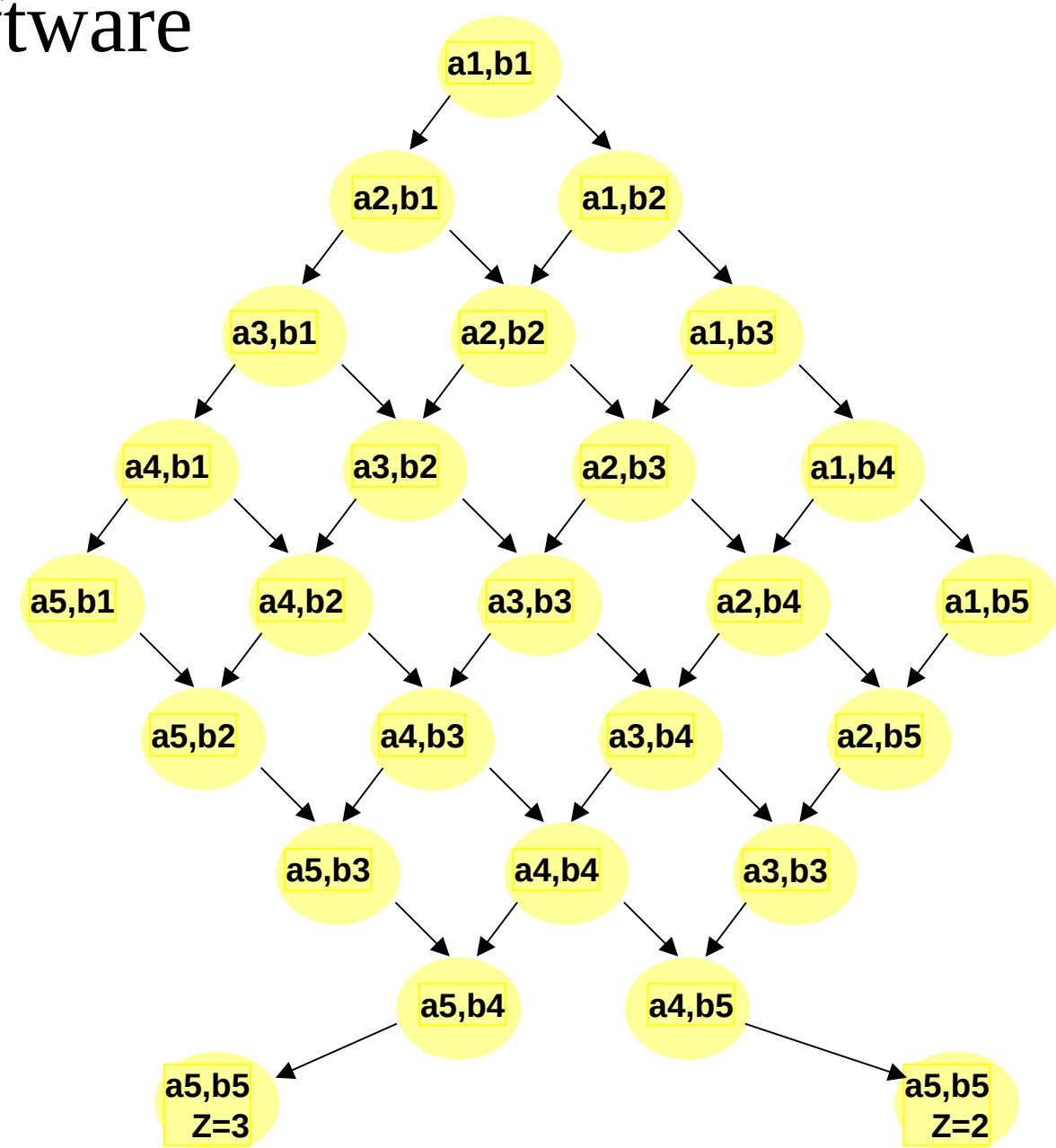
b1: p=G

b2: p=0

b3: q=1

b4: Z=3

b5:



Designer's Dilemma

The development process involves:

- Design
- Implement
- Test

The Dilemma:

Selection of a design requires an implementation to determine whether or not the design satisfies required properties.

RW Semaphore Model

```
#define READ 0
#define WRITE 1

bit xop; /* takes values of READ, WRITE */

byte nw;
byte nr;

byte s;
chan cs = [1] of {bit};

active [1] proctype read_process(){
 bit theKEY;

 cs?theKEY;
 if
  :: (xop == READ) ->
   if
    :: (nw > 0) -> atomic { (s == 0); nr++}
    :: (nw <= 0) -> nr++;
   fi;
  :: (xop == WRITE) -> atomic {nr++; (s == 0)}
 fi;
 xop = READ;
 s++;
 cs!theKEY;

 skip;

 cs?theKEY;
 nr--;
 s--;
 cs!theKey:
}
```

```
active [1] proctype write_process(){
 bit theKEY;

 cs?theKEY;
 if
  :: (xop == WRITE) ->
   if
    :: (nr > 0) -> atomic {(s == 0); nw++}
    :: (nr <= 0) -> nw++
   fi;
  :: (xop == READ) -> atomic {nw++; (s == 0)}
 fi;
 xop = WRITE;
 s++;
 cs!theKEY;

 skip;

 cs?theKEY;
 nw--;
 s--;
 cs!theKEY;
}

active [1] proctype startSEM(){
 cs!1;
}
```

let the writer process go first.
it gets the cs semaphore and proceeds.
since xop is unitialized, assume that it is 0 and that 0 means READ.
in that case nw is incremented, and we wait for xs to become 0.
Note since we have not initilized xs, we can assume it is also 0
(and indeed this is what spin does).
in this case, xop is set to WRITE and xs is incremented (via
the UP). now the cs semaphore is released.

at this point, a reader process goes forward.
xop is write, so we bump nr and then we wait for xs to become
zero. Problem: no process/task is running that can decrement (wait
on xs).

Spin demonstrates this in approximately 14 steps

Why use model checking?

- Concurrent programs
  - Hard to test, programmers have less intuition
  - Poor coverage
    - Buggy interleavings may occur only at the customer's site
    - Hard to reproduce the bug
  - SW companies are not willing to invest resources and skilled personnel

- Micro-code, smart-cards etc
  - Closer to hardware (in size and features)
  - Bugs are expensive to fix

- Critical software (intensive care systems, finance, security, anti-missiles systems etc.)

Why should testers care about this stuff?

- **Program analysis and model checking techniques will become increasingly important as we begin to deal with clock scaling issues:**
  - Hardware will become much more concurrent (SMP cores, hyperthreading)
  - Code will be come much more concurrent
  - Systems will become much more concurrent
- **Program analysis and model checking will become increasingly important as we address customer perceptions about product roll-outs**
  - Takes a long time
  - Not transparent to applications
  - Lots of regression testing
  - Lots of manual work
- **Program analysis and model checking will become increasingly important as the tools get better:**
  - MS runs prefix/prefast over 100% of code – SLAM/Zing is smaller effort, but 100% of device driver code supplied by MS is verified; SLAM/Zing growing in importance within the OS itself

Model Checkers use Temporal Logic

- Allows statements about
  - both states and computational paths
  - states in the “future”
- Operators like
  - “Always Eventually P” – from current state, system always eventually reaches some state where P is true
  - “Possibly Eventually P” – from current state, it is possible to reach a state where P is true
  - “Always Forever P” – from current state, no matter what computation is done, P will always remain true
  - “Possibly Forever P” – from current state, it is possible for a computation to always keep P true

Writing Specifications is not easy

- Two threads should never both be in critical region
  - Always Forever not (Critical(p) and Critical(q))
  - not Possibly Eventually (Critical(p) and Critical(q))
- User process should never get root permission
  - Always Forever not RootPermission
- Thread should always terminate
  - Always Eventually Terminated(p)
- Thread should never deadlock
  - Always Forever (not
    Always Forever not Running(p))
- If a thread reaches critical section, it should always leave
  - Always Forever
    ((not Critical(p)) or
    (Always Eventually (not Critical(p)))

A real specification

```
#define fb (bufferFULL==1)
#define nfb (bufferFULL==0)

never { /* !([](fb -> <>nfb)) */
T0_init:
        if
        :: (1) -> goto T0_init
        :: (!nfb && fb) -> goto accept_S2
        fi;
accept_S2:
        if
        :: (!nfb) -> goto accept_S2
        fi;
}
```

So how do we use this?

- Need an ontology - every system, every component of a system has properties and behaviours
- Ontology: a conceptualization of realization - common language, common semantics
- Shouldn’t the design work before I actually assemble the components?

Conclusion

- Modelling a system is hard. But even the simplest test has an implicit model
- Writing temporal specifications helps define the model (ontology)
- Define system behaviour in terms of state machines!
- Don’t be afraid to use Formal Verification
- Formal Verification is in its infancy

Q & A