WEDNESDAY, Session 3
Introductory lecture

1 Bugs

Formal verification means using methods of mathematical argument to determine the correctness of systems (be they computer programs, computer systems, or chemical plant designs). If we always designed correct systems, however, we would have no use for formal verification (though we might still need formal specification). Most of the time, when we apply formal "verification", we are applying it to a system that is actually incorrect – it has "bugs". What we stand to gain by applying formal verification is to actually to find and correct these bugs. Thus, to understand the motivation behind formal verification, and to understand where we can fruitfully apply it, it is worthwhile to begin by studying a bit of "entomology" – that is, what species of bug are we after, where do they come from, and why are they so difficult to avoid in the first place?

1.1 Concurrency bugs

Here is a scenario: suppose you are building a complex harware/software system. For example:

- a 100K gate ASIC – perhaps 100 concurrent modules
- a flight control system – dozens of concurrent processes on multiple CPU's

Under test, the system fails approx. every three days, with unknown cause. You are now faced with a very interesting problem:

- The failure is not repeatable. It depends on the resolution of "race conditions".
- You cannot collect enough real-time data to find the bug
  - internal signals on chip are hard to observe
  - there is too much data to store anyway
  - the "Heisenbug" problem – tracing execution makes the bug go away

This is an example of a concurrency bug. Typically:

– Events x and y happening concurrently, say every 10^{10} cycles

– Programmer did not anticipate x and y could happen concurrently

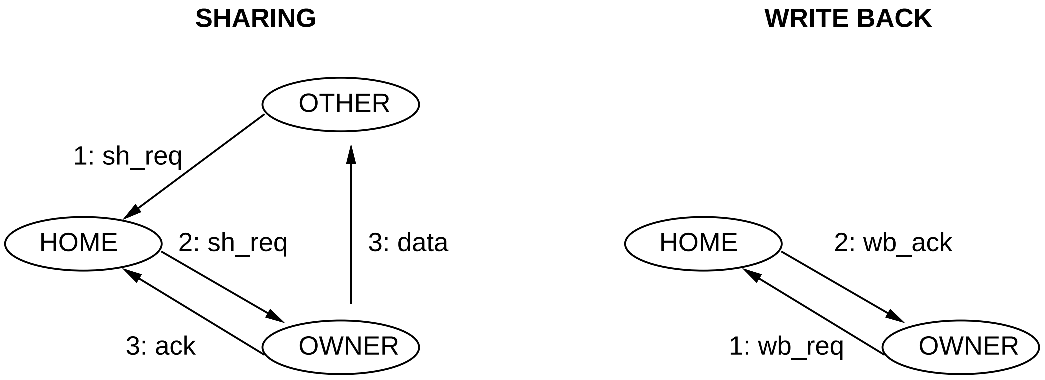
A trivial concurrency bug:

process A	process B	prestate: `x = 0`
⋮	⋮	
`x := x + 1;`	`x := x - 1;`	
⋮	⋮	
		poststate `x = 1` (!)

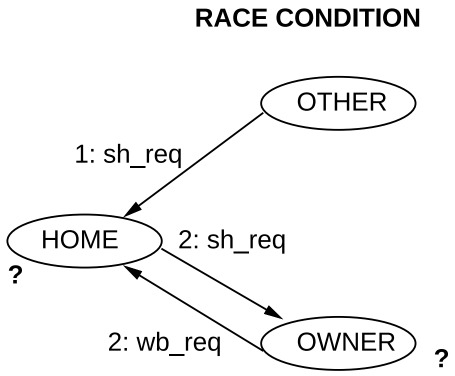
Such trivial errors can be prevented by good practice (i.e., using semaphores, monitors, *etc.*), but less trivial concurrency bugs are harder to prevent.

1.2 An example

In a distributed protocol, there are typically many possible "race conditions" where the outcome depends on the (unpredictable) speed of delivery of messages. Consider, for example the following two transactions occuring in a cache coherence protocol:



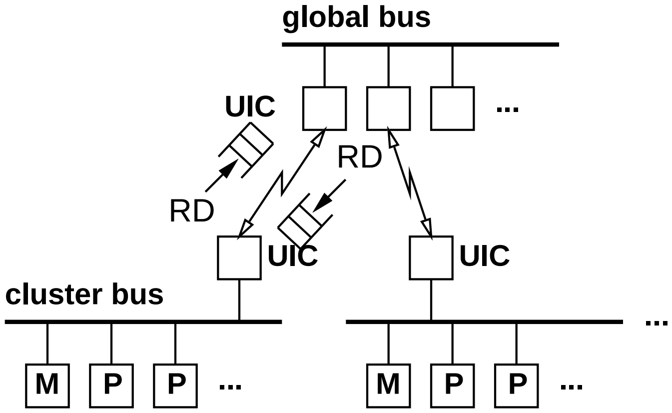
A race condition can occur, for example, if there a "write back" operation begins while a "sharing" transaction is in progress. In this case, unexpected messages are received:



There are typically 10's to 100's of such cases to consider in a real protocol. Designers usually think of most of them in design reviews...

1.3 A real-life concurrency bug

In this hierarchically structured cache coherence protocol, a bad sequence of 13 messages was required to reach a deadlocked state, where messages queued in opposite directions on a link lock each other out.



This is a bug that wasn't uncovered by simulations and design reviews.

2 The mathematical approach

Since simulation and testing are problematic, especially for concurrent systems, it makes sense to consider a more systematic, mathematical approach. This requires:

1. A *model* of the system as a mathematical object (*i.e.* a function or a relation). A

model typically represents formally:

- program state (program counter(s), variable valuations, *etc*)
- transition relation (state $\rightsquigarrow$ state)

2. A *specification method* for expressing desired properties in some logical language. For example,
$$\text{IC} \rightsquigarrow^* (\text{terminated} \Rightarrow \text{FC})$$

3. A *proof method* to show that the model satisfies those properties.

Better yet, we can exploit advances in

- automated deduction
- automated proof checking

to help construct and verify the proofs. Proving can be done at several levels:

- by-hand verification
- semi-automated verification: user suggests a proof which is checked by machine
- fully automated verification: proof done entirely by machine

As an added benefit, if the proving software can isolate bugs faster than testing and simulation, the development process will be accelerated.

2.1 Easy objections

1. Proofs are about models, not real systems.
2. Specifications are subject to errors, incompleteness.
3. Theorem proving software can be unreliable.
4. Computer proofs are unreadable.
5. Proofs are not validated by “community”.

The methodological answer

We can dispense with all these objections if the desired goal is not mathematical certainty (whatever that is), but rather a *methodology* that, when followed, produces more reliable programs or systems.

- Even a hand proof methodology, when followed rigorously, is likely to produce cleaner, more robust programs than an informal approach.
- Increased automation of proof is likely to produce better programs.

2.2 The real objection

The BIG problem is that computer systems and programs are orders of magnitude more complex than theorems proved by mathematicians. So what makes us think we can apply mathematical proof to such unwieldy objects?

- Automation: get computers to prove more substantial "lemmas" without human assistance.
- A compositional approach (divide-and-rule): break very large objects into smaller objects and prove properties of these smaller objects.

Almost all practical research in formal verification can be viewed as attacking one of these two problems: how to automate bigger proof problems, and how to divide proofs about large systems into proofs about smaller systems.

3 A selective history

Here we consider some of the historical developments in formal verification that are relevant to this course. This material is mostly derived from Gries [2] and Hailpern [3].

3.1 Early 1960's

Program proving suggested by a few "visionaries" (McCarthy, Dijkstra).

3.2 Late 1960's

"Software crisis" declared at NATO conference on Software Engineering.

3.2.1 Floyd's method for flow charts

- Label each control point with invariant assertions
- Proof: all assertions preserved by program transitions. This proves "partial correctness" (if program terminates, result is correct).
- Use well-founded orders to prove termination
- Floyd noted that the inference rules provide a semantics for the flow chart language (the link between programs and logic).

3.2.2 Hoare logic – a structured approach

Hoare provided a compositional approach, giving inference rules that correspond to program constructs.

- Hoare triple
  $$\{P\}\ S\ \{Q\}$$
  "if P holds and S terminates, than Q will hold"
- Axiom for assignment
  $$\{P^x_f\}\ x := f\ \{P\}$$
  example: $\{y \geq 0\}\ x := y\ \{x \geq 0\}$
- Compositional rules
  - for sequential composition $\dfrac{\{P_1\}\ S_1\ \{P_2\}\ S_2\ \{P_3\}}{\{P_1\}\ S_1; S_2\ \{P_3\}}$
  - for while loop $\dfrac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \text{while}\ B\ \text{do}\ S\ \{P \wedge \neg B\}}$

Note that the proof follows the program structure

1. Prove properties of program components

2. Combine properties using inference rules to prove program

This is the first example of a compositional approach. But note that the only properties we can specify are before/after conditions.

3.3 1970's

Large amount of work done on

- Axiomatizing various languages, constructs, *e.g.*
  - Procedure calls
  - Assignments to arrays, *etc.*
- Proving "relative completeness" (axiomatization proves everything that is provable from operational semantics).
- *Automating proofs*
  - Notably, the Boyer-Moore prover, which could automatically prove correctness of some recursively defined LISP functions, like APPEND, without the user having to suggest an invariant.

3.3.1 Proofs of concurrent programs – Owicki-Gries

In some sense, concurrent programs need formal verification more than sequential programs, since they are much more difficult to debug (the race condition problem). In principle, concurrent programs can be verified by invariants, just like sequential ones. However, for a concurrent program a "program location" is really a set of concurrent program locations. We certainly would not like to have to state an invariant for each such set. Instead Owicki and Gries set out a compositional system for concurrent program proving.

- Data shared only through "resources".

$$\text{resource r(x) : cobegin } S_1 \parallel \cdots \parallel S_n \text{ coend}$$

- Critical sections

$$\text{with r when } B \text{ do } S$$

"can enter S when B and resource r is free"

- Resource invariant $I(r)$
- Triple $\{P\}\ S\ \{Q\}$ means
  - if P true initially, then
    * finally (if S terminates) Q
    * $I(r)$ is preserved while S is in critical sections
- Coroutine rule:

$$\frac{\begin{array}{c}\{P_1\}\ S_1\ \{Q_1\}\\ \{P_2\}\ S_2\ \{Q_2\}\end{array}}{\{I(r) \wedge P_1 \wedge P_2\}\ \text{cobegin}\ S_1 \parallel S_2\ \text{coend}\ \{I(r) \wedge Q_1 \wedge Q_2\}}\ \text{(no interference)}$$

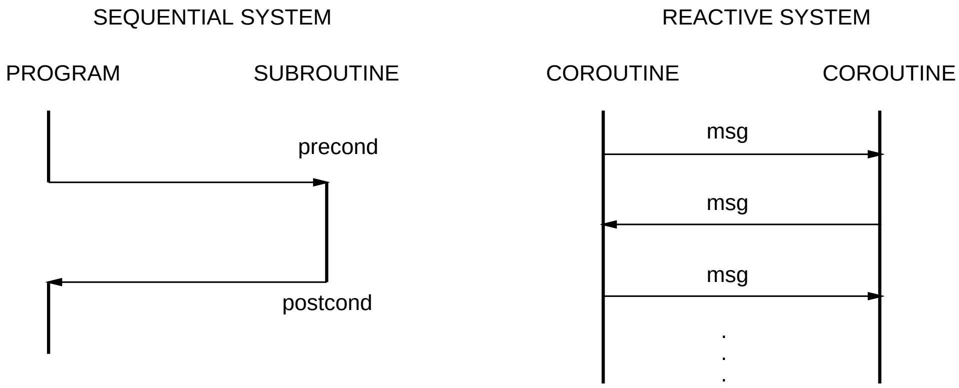
- Auxilliary variables added to program to aid in stating invariants

Note that this method is "compositional", but the properties you can prove about a component are very limited. Most proofs with such a method would really be global in nature (for example, each component introduces an auxilliary variable indicating its program counter, and the global invariant is stated in terms of these variables).

3.4 Late 1970's

3.4.1 Temporal logic and reactive systems

Pnueli recognizes that partial correctness and termination are not adequate abstractions of concurrent processes. Introduces the notion of a *reactive system* that interacts with its environment without terminating [4]:



Examples of reactive systems:

- control/operating systems
- communication protocols
- hardware

Because reactive processes interact over time, we need to be able to state and prove properties of execution *sequences*, not just pre- and post-conditions. Examples:

- If P is scheduled infinitely often, it eventually produces an output (liveness).
- If P sends a message, then it won't send another message *until* it receives an ack (safety).

3.4.2 Temporal operators

Extensions to the logic:

- $G\ p =$ "Henceforth p"
- $F\ p =$ "Eventually p"
- $X\ p =$ "At the next time p"

Example:

$$GF\ \text{scheduled}(p) \Rightarrow F\ \text{terminated}(p)$$

"if p is always eventually scheduled, it eventually terminates".

> **Compositionality**: By providing a rich enough language of properties, temporal logic allows us to reason compositionally about parallel programs, in the way that Hoare allows us to reason about sequential programs.

However, this does not mean that reasoning with TL is easy. For example, Hailpern provides a 15 page proof of the correctness of a nearly trivial protocol (the ABP).

3.4.3 Protocol analysis and reachability analysis

- The model:
  - Set of program states
  - Transition relation between states
- Example: communication protocols can often be modeled as finite state systems.
- Idea: instead of reasoning symbolically about the model, just construct it explicitly. This is called reachability analysis.
  - Colin West at IBM
  - Petri nets

Advantage:

- Totally automated

Disadvantages:

1. Can verify only limited properties (non-compositional)
   - Invariant assertions
   - Non-deadlock
2. State explosion problem
   - Can handle only fairly small models

For these reasons, this technique was more or less dismissed, since the ability to do modular proofs made it seem probable that axiomatic methods could handle large systems:

> "I believe the next ten years will see [program proving] extended to and practiced on large programs"
>
> David Gries, 1980

3.5 1980's

3.5.1 Model checking (Clarke/Emerson, Queille/Sifakis)

1. Specify properties in TL.
2. Build finite state model explicitly (as in reachability).
3. Check algorithmically that model satisfies specificaiton.
4. Produce counterexample sequence when false

This approach has a number of advantages:

- Remedies expressiveness problem of reachability analysis.
- Is compositional, since based on TL.
- Complexity is linear in formula size for temporal logic CTL.
- Very good for *disproving* properties.

For small but tricky asynchronous circuits:

- Showed previously published designs incorrect (embarrassing).
- Showed previously "proved correct" designs incorrect (highly embarrassing)

Kurshan develops closely related technique where finite automata are used to represent both the model and the properties to verify.

> *Lesson: if automation is the goal, one way to achieve it is to restrict the domain of application (for example, to finite state systems). The restricted domain may be richer than you think.*

3.6 Late 1980's

Succcess of model checking on small examples leads to various attacks on the "state explosion" problem.

3.6.1 Modularity – abstracting system components

- Using temporal formulas

  Same approach as Pnueli, but use model checking or decision procedures for lemmas.

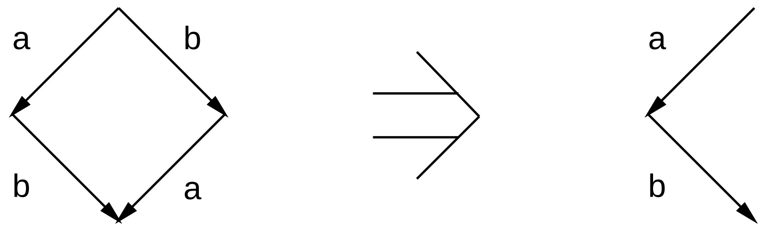
- Abstract finite state models by other finite state models.
- Automatically generated reductions

3.6.2 "Symbolic" model checking techniques

- Use compact boolean forms to represent state sets and transition relations (*e.g.* BDD's)
- Can handle state spaces many orders of magnitude larger, given certain conditions on the structure of the state space.

3.6.3 "Partial order" methods

- Often, state explosion is due to many (irrelavant) orderings of independent events.
- Treating these events as partially ordered allows us to group executions into equivalence classes and explore only one representative from each class.



3.6.4 Symmetry reductions

- Consider only one state in each class of symmetrically related states.
- Often process id's, address and data values are equivalent under symmetry transformations

Using combinations of the above techniques, very subtle bugs have been found in, for example

- Commercially design cache coherence protocols (including the Gigamax bug described above, for example)
- Communications ASIC's

In additional, model checking techniques have been applied "retroactively" to find the source of bugs in chips designs, where the bug occurred on the testbench, but the source could not be found due to "observability" problems (*e.g.*, Fujitsu, Silicon Graphics). This suggests that formal verification methods may be of more use in "disproving" than in "proving".

3.6.5 Research on more expressive formalisms

- Synthesis of automata theory with temporal logics.
- Finite state real time systems.
  - Real time temporal logics.
  - Timed automata

3.7 1990's

- Hybrid systems – systems with both discrete and continuously changing variables.
- ? (perhaps combining theorem provers and model checkers)

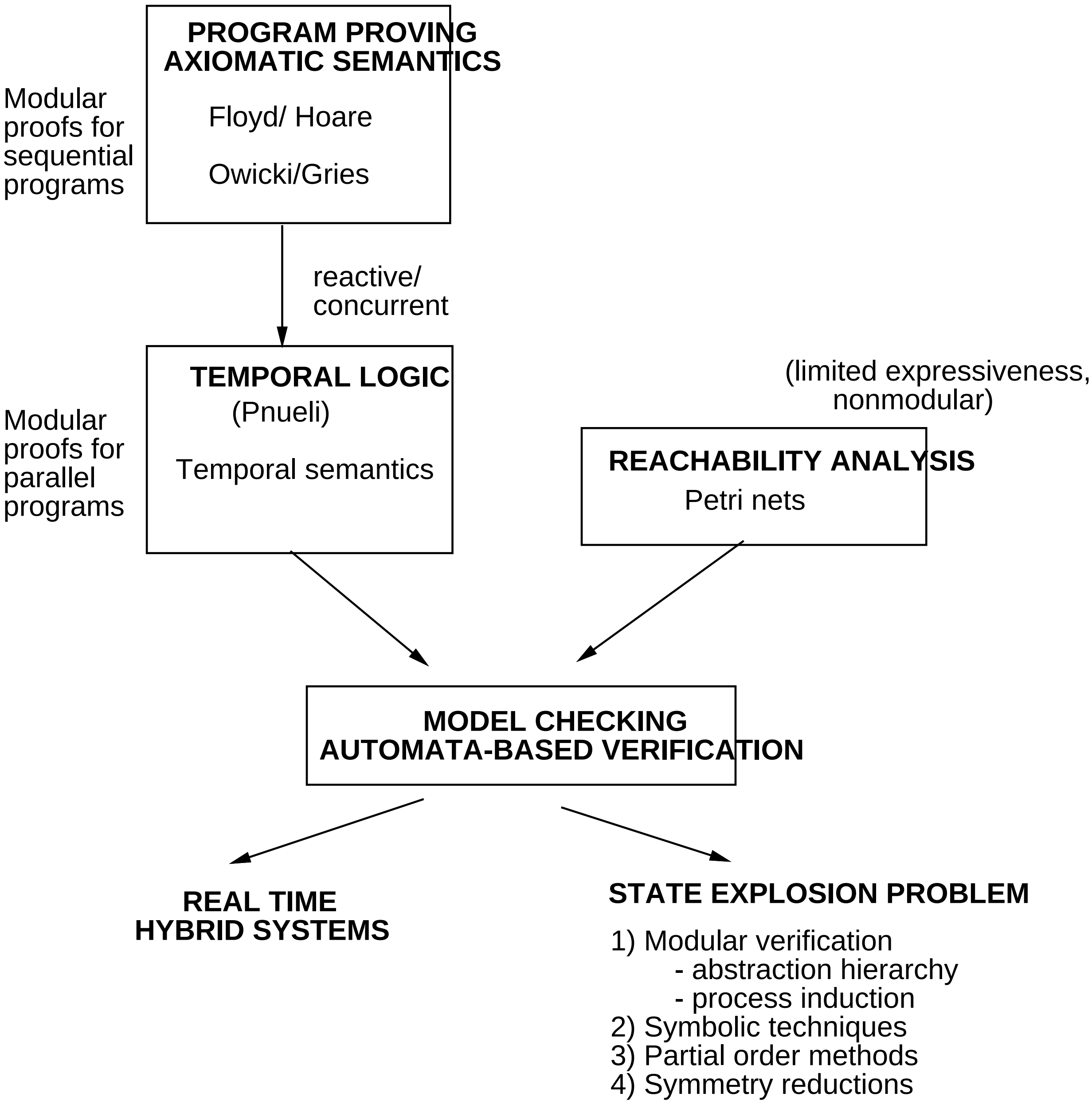
References

[1] P. Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Comp. Sci., vol. B: Formal Methods and Semantics*, chapter 15. Elsevier, 1990.

[2] D. Gries. *The science of programming.* Springer-Verlag, 1981.

[3] B.T. Hailpern. *Verifying concurrent processes using temporal logic.* Springer-Verlag, 1982.

[4] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.


PROGRAM PROVING
AXIOMATIC SEMANTICS
Floyd/ Hoare
Owicki/Gries
Modular
proofs for
sequential
programs
reactive/
concurrent
TEMPORAL LOGIC
(Pnueli)
Temporal semantics
Modular
proofs for
parallel
programs
(limited expressiveness,
nonmodular)
REACHABILITY ANALYSIS
Petri nets
MODEL CHECKING
AUTOMATA-BASED VERIFICATION
REAL TIME
HYBRID SYSTEMS
STATE EXPLOSION PROBLEM
1) Modular verification
- abstraction hierarchy
- process induction
2) Symbolic techniques
3) Partial order methods
4) Symmetry reductions