Formal Verification

Madhavan Mukund

Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

RMIT Symposium on Mathematics and Information Technology
Surat, 28 December 2010

Designing complex systems

Goals: Safety, security and other dependability properties

- How is it done for traditional (mechanical) systems?
  - e.g., an aeroplane wing
- How is it done for software systems?
  - e.g., a flight-control system

Product based approach vs process based approach (J Rushby, SRI)

Designing traditional systems

- Product based certification
  - Describes properties of (mathematical models of) product
- Primarily mathematical modelling and analysis
  - Build a model of the design, environment and requirements
  - Calculate that the design (in environment) meets requirements
  - To be useful, must be mechanized (e.g., finite element analysis)
- Modelling is validated by tests
  - Systems are continuous, limited testing is sound

Designing software systems

- Primarily by controlling the mechanism of software creation
  - Standards for coding, review, documentation
- Process based certification
  - No guarantee about resulting product!
- Testing is product based but
  - Complete testing is infeasible for reasonable sized systems
  - Extrapolation from incomplete tests is unjustified for discrete systems

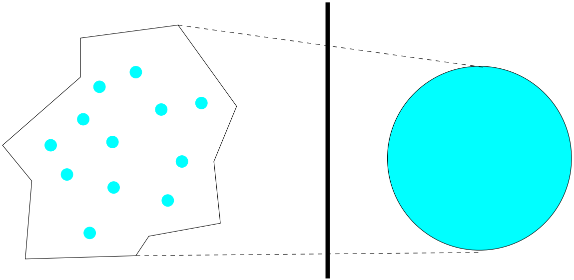
Designing software systems . . .

- Code review, testing etc are effective at finding coding bugs
- The difficulties lie in
  - missing requirement specifications,
  - incorrect interface descriptions,
  - lack of fault tolerance in design,
  - coordination problems in concurrent activities . . .
- Process based methods do very badly in such areas
- Case study (Lutz 1993)
  - 197 critical faults detected during integration and system testing of Voyager and Galileo spacecraft
  - Only 3 were coding errors

Software . . .

Testing/simulation

Formal analysis



Real system

- Partial coverage

Formal model

- Complete coverage

Formal verification!

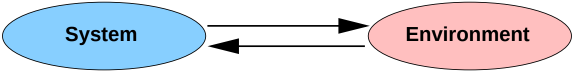
Formal verification: Product based certification for software

- Build a mathematical model of design, environment, requirements
  - Mathematics of verification is formal logic
  - Models are formal descriptions in a logical system
- Calculate that the design (in environment) meets requirements
  - Prove that assumptions+design+environment logically imply requirements
  - Use model checking or theorem proving
  - Formal calculations make assertions about all behaviours, even if infinite

Model checking

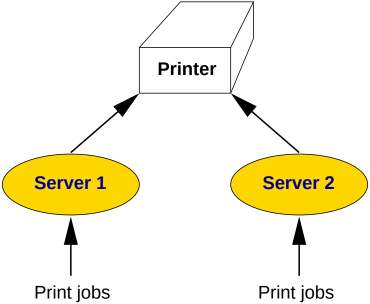
- Traditional systems

- Reactive systems



  - Schedulers, controllers, operating systems, ...
  - Desirable behaviour is nonterminating

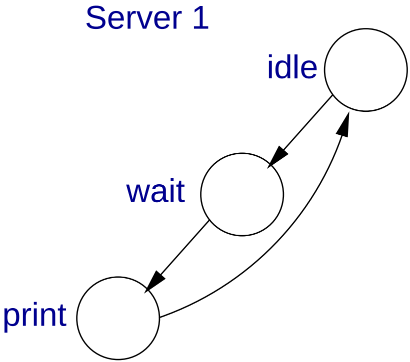
Naïve print server



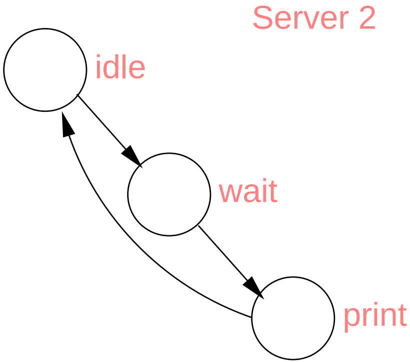
Server 1

status = idle;
loop
    receive job;
    status = wait;
    if (status(server 2) $\neq$ print)
        status = print;
    status = idle;
forever

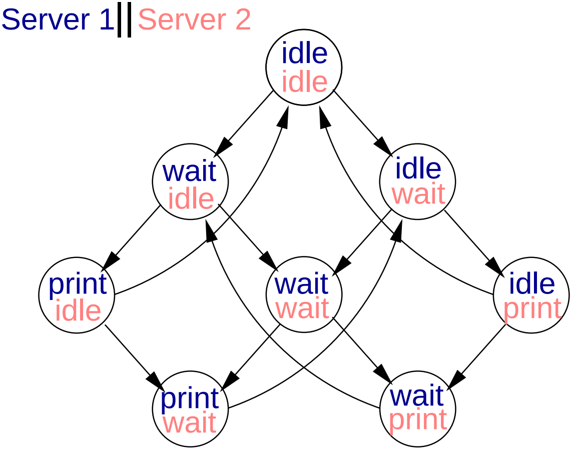
Naïve print server . . .



Naïve print server . . .

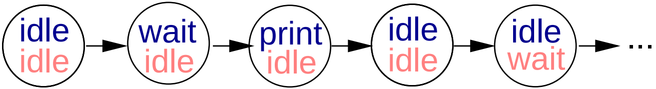


Naïve print server . . .



Naïve print server ...

- An execution is an infinite sequence of states



- Need a language to describe properties of such sequences
  - *Access to printer is mutually exclusive*
  - *Every print request is granted*
  - *Print requests are not lost while waiting*

Temporal logic

- Formulas are built from basic atomic facts
  e.g.,

  $i1$ = "status of server 1 is idle"

  $w2$ = "status of server 2 is waiting"

- Combine formulas using

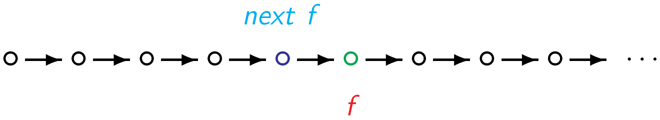
  Boolean connectives: *not*, *and*, *or*

  Temporal modalities:

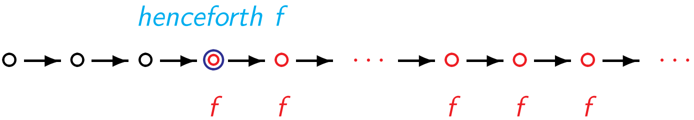
next f	*f* holds at the next position
henceforth f	*f* holds from now on
eventually f	*f* holds at some future position

Temporal modalities

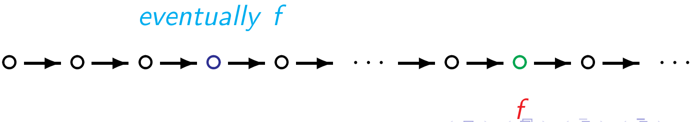
- Next



- Henceforth



- Eventually



Expressing desirable properties

- *henceforth* (*not*(*p*1 *and* *p*2))

  Printer access is mutually exclusive

- *henceforth* (*w*1 *implies eventually p*1 *and*
  *w*2 *implies eventually p*2)

  Every print request is granted

- *eventually*(*henceforth* f)

  f becomes a stable property

- *henceforth*(*eventually* f)

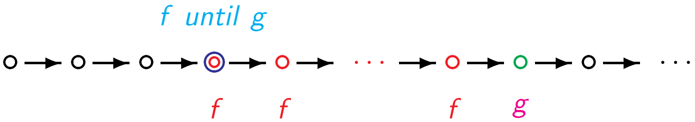
  f holds infinitely often

One more modality

- How do we express the following?

  Print requests are not lost while waiting

- A new (binary) modality: f *until* g



- The formula we want is $w1$ *implies* $(w1$ *until* $p1)$

Model checking

- An execution satisfies f if f holds at the initial position
- A system S satisfies f if every execution of S satisfies f

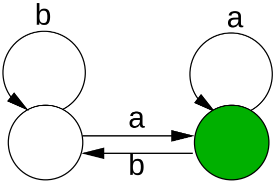
Model checking

Given S and f, does S satisfy f?

- Solve using Büchi automata
  - Formula $f \implies$ Büchi automaton A_f that captures *all* executions that satisfy f
  - Input system S $\implies$ automaton A_S
  - Is every execution of A_S also an execution of A_f?

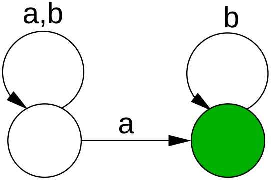
Büchi automata

- Automata on infinite inputs
- Accept an input if it visits a good state infinitely often



- Accept all sequences with an infinite number of a's

Büchi automata



- Accept all sequences with *finite* number of a's
    - Necessarily nondeterministic!

- $L(A)$, language of automaton A
- Model checking
    - Does S satisfy f $\Leftrightarrow$ Is $L(A_S) \subseteq L(A_f)$?
    - Can be checked algorithmically, relatively efficiently

Other temporal logics

- The temporal logic we have looked at is linear time
  - *Every* execution must satisfy *f*
- Alternative approach is branching time
  - Quantify over execution paths

    For some execution path, f

    For every execution path, f
- Branching time logics and linear time logics are incomparable in expressive power
  - Model checking is theoretically more efficient for simple branching time logic

Handling state explosion

- Systematic exploration of state space is a basic operation
- k components in parallel with m states each $\Rightarrow$ m^k global states
- Symbolic model checking
  - Use efficient representations of boolean functions

State spaces and boolean functions

- Each state s is “named” by an n bit vector $\lambda(s)$
- Transition relation $\rightarrow$ between states is a boolean function f on $2n$ variables

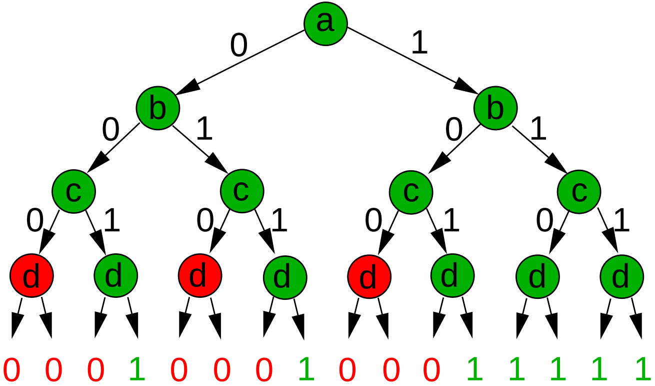
$$f(\lambda(s), \lambda(s')) = 1 \Leftrightarrow s \rightarrow s'$$

- A set S of states is a boolean function g on n variables

$$g(\lambda(s)) = 1 \Leftrightarrow s \in S$$

Boolean functions . . .

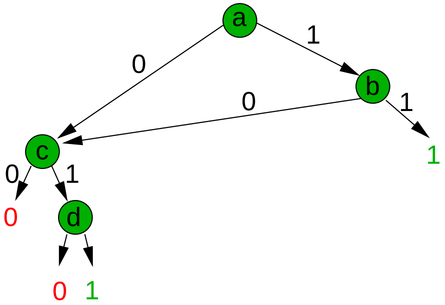
Ordered decision tree for $f(a, b, c, d) = ab + cd$



Binary decision diagrams

Compact representation of boolean functions (Bryant 1986)

- Reduced ordered binary decision diagram for $f(a, b, c, d) = ab + cd$
- Key idea
  Combine equivalent subcases



Binary decision diagrams . . .

- BDD for f is canonical (for a fixed variable order)
  - Check if $f = g$ by comparing their BDDs
  - e.g., can check if subsets of states S and T are the same
- Efficient algorithms for combining BDDs
  - Build BDD for f op g for boolean operator op from BDDs for f, g
  - e.g., given BDD for f and g, can build BDD for $f \wedge g$
- Use BDDs to represent and manipulate state spaces
  - Symbolic model checking (Clarke, McMillan et al)
  - Can significantly increase the sizes of state spaces that can be explored for model checking

Handling state explosion, cont'd

Other techniques

- Exploit symmetry in the system
  - Discard equivalent, symmetric configurations
- Exploit independence of actions
  - n independent actions can execute in $n!$ different ways
  - Sufficient to analyze any one of these sequences

Beyond finite-state systems

- What about non finite-state systems?
  - e.g., part of the state is an integer value
- Design property preserving abstractions
  - Want to establish property P for an infinite-state system G
  - Collapse G to a finite-state system G' and establish property P' for G' such that

$$G' \text{ satisfies } P' \text{ implies } G \text{ satisfies } P$$

or, in some fortunate situations,

$$G' \text{ satisfies } P' \text{ iff } G \text{ satisfies } P$$

Beyond finite-state systems . . .

Recursive programs

- Can have an unbounded stack of function invocations
- Natural model is pushdown automaton — most decision problems are undecidable
- Represent configurations of pushdown systems as strings
  - Set of reachable configurations is a regular language
  - Can compute successor and predecessor configurations
  - Effectively compute set of states reachable from s both forwards and backwards

Beyond finite-state systems ...

Theorem proving

- Use a stronger logical formalism to model system
  - e.g., first-order logic, with integers, reals etc
- Formulate verification as a theorem to be proved
- Use a mechanical theorem prover to verify properties
- Less automated than model checking
  - Theorem provers have idiosyncracies
  - Not all "obvious" proof strategies work!

Verification and testing

Verification has also had an impact on testing

- State space exploration techniques can be applied to get better coverage
- Automated generation of test plans (Jeron et al)
- Specification based testing of software
  - Use the design specifications to suggest test plans
  - More representative than post facto test plans based on implementation

State of the art

- Many software tools have been developed
  - Automata based model checkers such as SMV, Spin, . . .
  - Software model checkers such as SLAM extract finite-state models from program text
- Verification of large-scale systems is still far from automatic
  - Techniques are still too expensive for commercial industry
  - Notable exceptions are hardware companies, like Intel and AMD
  - Increasing use in safety-critical areas:
    Nuclear plants, avionics, satellite control etc

What lies ahead?

- Making the technology more useable
  - Better hardware increases sizes that can be handled
  - Still, *real* systems are often too large as a whole
  - Improve automation of techniques such as abstraction
  - Bridge the gap between model checking and theorem proving

    SAL initiative at SRI (Rushby et al)

- Build up "libraries" of verified designs
  - Build software like hardware, from known components