state machine invariants

Martin Rinard
February 23, 2009
plan for today

topics

‣ traffic lights: what safety means
‣ state properties and invariants
‣ reasoning about traffic lights
‣ interlocks: runtime enforcement
traffic lights
problem

road works
- road narrows to one lane
- workers have flags but can’t see each other

protocol
- initially: one shows green, one shows red
- worker gives last car a message and shows red
- worker at other end gets message and shows green

does it work?

variants
- passing the baton (used on railways)

Illustration taken from story describing this protocol at http://www.csmonitor.com/2008/0912/p19s05-hfes.html
state machine properties

what can we ask about a state machine?

› safety: does it do anything bad?
  do cars crash in the middle?

› liveness: does it do anything good?
  do cars ever get to go?

in practice, liveness rarely useful

› “eventually” isn’t good enough
  “happens before midnight” is a safety property (“no chime before op”)

how to formulate safety?

› abstractly, every trace satisfies a property
  eg, not green in both directions

› concretely, every reachable state satisfies a property
simpler traffic lights

consider simpler traffic lights first

- one can’t go into waiting for green until other goes red

parallel machine semantics: reminder

- in each step, one event occurs
- if event belongs to both machines, both must do the transition
- if event belongs to just one, only that machine moves
product machine

can form a single product machine

› states are tuples
› one state from each machine

‘state explosion”

› k machines of N states
› product machine has $N^k$ states
› this is why concurrency is hard!
traces revisited

what’s a trace?

- a trace is an event history
- machine has set of traces
- includes empty trace

example

- traces of traffic light system include:
  
  $$<>,$$
  
  $$<r2>,$$
  
  $$<r2, g1>,$$
  
  $$<r2, g1, r1>,$$
  
  $$<r2, g1, r1, g2>, ...$$
checking traffic light property

what’s the traffic light property?
  ‣ crucial property: never green both ways
    not GG

how to check?
  ‣ just look at product machine
  ‣ color satisfying states yellow
  ‣ check all reachable states are yellow

but doesn’t scale
  ‣ how would you do this for 3 parallel machines of 10 states each?
state properties & invariants
A property can be represented as a state set:

- A predicate $P(s)$ applied to state $s$
- A subset $\{s: S \mid P(s)\}$ of states
**Invariant reasoning**

**How to check safety property?**

- If diagram is small, just check every reachable state.
- But if state machine is large, need better method.

**Invariant reasoning**

- Check property holds in initial state.
- Check each operation “preserves” property.
  - Property holds before $\Rightarrow$ property holds after.
- If so, property is “invariant.”

**Example**

- Initial state is green, and each op preserves greenness.
- So greenness is an invariant.
why invariants work

strategy

› check property holds in initial state (1)
  \[ I(S_0) \]

› check that transitions preserve property (2)
  \[(s, e, s') \in R \land I(s) \Rightarrow I(s')\]

› then property is an invariant, and holds in all reachable states

why?

› consider any trace

› holds at start by (1)

› can repeatedly add events using (2), and holds after each
  (in general, this unfolding gives a tree: can you see why?)
tiling the chessboard

a classic problem

- 8 x 8 chessboard can be filled with 32 dominoes
- now remove top-left and bottom-right squares
- can you tile remaining 62 squares with 31 dominoes?

invariant reasoning

- consider number of black and white squares covered
- invariant: \#black = \#white
- initially, \#black = \#white = 0
- only operation is placeDomino (loc)
  always adds 1 to \#black and to \#white, so it preserves the invariant
- board with corners removed has 32 black, 30 white
  this state does not satisfy the invariant, so it’s not reachable
strengthening

when property is not an invariant
› even though it holds for all reachable states
› need to strengthen the property
› typical feature of inductive reasoning

diagram (upper)
› op2 takes green S6 to non-green S4
› but S6 is not reachable!

diagram (lower)
› consider green–blue invariant
› now preserved, and green–blue => green
traffic light invariant

consider our property not GG

- unfortunately, it’s not an invariant
- consider the transition (GW, g2, GG)
- property holds before but not after
getting to the invariant

what’s wrong?

› need to strengthen the property

› an invariant is
  R1 or R2

› which implies
  not GG
back to roadworks
state machine model

designations

r0: worker 0 raises red flag
send0: worker 0 sends message to worker 1
recv0: worker 0 receives message from worker 1 and raises green flag
product machine

showing reachable states only

- 27 possible state combinations, but only 6 reachable
- getting harder to check reachable states by eye...
is the property an invariant?

the desired property is

\[ \text{not } GGx \quad / / \text{ } x \text{ is informal way of saying that other process can have any state} \]

exercise

› is this an invariant?
› if not, find a violating transition
› can this transition occur?

solution

› a violating transition is
  \[ \text{recv0 from WGB to GGI} \]
› can’t happen: when second process is green, no message waiting for first
finding an invariant

exercise
  › strengthen the property to make it an invariant

solution
  › add to property: if there’s a message in transit, both workers are waiting
    not GGx and (xxI or WWx)

proof that this is invariant
  › only two ways to get to GGx: from WGx or from GWx
  › in either case, need recv event to occur
  › but recv event can only occur in xxA or xxB
modelling faults

possible faults
› dropped message (driver forgets, or veers off road)
› duplicated message (forged by mischievous driver)

can we model these?
› yes, make message submachine non-deterministic
› add drop transitions from A and B to I
› add dup transitions from I to A and B

in practice, analysis is hard
› use a tool such as a model checker to do it automatically
› many such tools for state machines
interlocks
invariants are your friend

often they give you
  › the simplest way to express important properties

can often be checked in code
  › with runtime assertions

can be reasoned about
  › inductive reasoning especially powerful
interlocks

interlock of ‘gatekeeper’
- simplest way to maintain an invariant
- check at runtime, and don’t let invariant be broken

two approaches
- pessimistic control: check before transition, and maybe disallow
- optimistic control: check after transition, and undo it if bad

key advantage
- small module can enforce invariant in large system (“small trusted base”)
Therac-25 accident

Therac-25 radiotherapy machine
- two modes: electron beam and xray
- rotating turntable with 3 positions
- two power levels, lo and hi
- invariant: in X-ray mode, use flattener
  - power = HI ⇒ turntable = FLATTENER

what happened
- earlier version had hardware interlocks
- software had concurrency bug
- invariant violated and 6 patients overdosed, of whom at least 3 died

is your PC secure?

typical patch size
  ‣ 100MB

typical time to download
  ‣ 10 minutes

average time to infection*
  ‣ 4 minutes

* [Windows XP, default firewall settings] Unprotected PCs Fall To Hacker Bots In Just Four Minutes
  Gregg Keizer; Nov 30, 2004; http://www.techweb.com/wire/security/54201306
  From: Security Absurdity: The Complete, Unquestionable, And Total Failure of Information Security
  Noam Eppel; http://securityabsurdity.com
buffer overflows

problem of buffer overflows

- a major source of security vulnerabilities
- huge cost to industry and individuals

how they work

- program reads messages into buffer of fixed capacity
- buffer is stack allocated; below it is return address for call
- rogue agent passes big message
- buggy code writes message over return address
- return address is replaced by address of code inside message
how to avoid overflows

interlocks

› invariant: buffer size < buffer capacity
› check before writing message into buffer
› eg, for each array update, check bounds
   \[ a[i] = e \quad // \text{only if } 0 \leq i < \text{MAX} \]

so why don’t people do this?

› most programs in “unsafe” languages like C that don’t check bounds
› programmers say too costly to check (but cost of not checking?)

lesson

› add an interlock (with a safe language, or a data abstraction)
› or prove invariant preservation
why not interlocks?

interlocks

 › are great when they’re possible
 › but they don’t always fit the context

problem areas

 › rejecting events makes things complicated (and users unhappy)
 › doing the check might damage performance
   eg: database index is properly ordered
 › may not be able to see the state
   state is distributed, so nobody has global view
   eg. node can’t see state of whole network
   state cannot be read at all
   eg. radiotherapy machine can’t read dose received by patient
summary
principles

design = model + properties

› whenever you design a behavior
› ask what properties you expect it to have
› the power of redundancy

invariants are your friend

› give **modular** reasoning
› more on this later in the course

interlocks reduce trusted base

› enforce a property locally
› then less code to worry about
backup slides
example: radiotherapy

- problem & approach
  - want to deliver dose and record on disk, but both may fail
  - so break dose into small segments, and alternate deliver/record

given specs \textbf{op dose: post }d' == d+1 \quad \textbf{op write: post }r' == r + 1
prove \( d - r \leq 1 \)