6.005 Elements of Software Construction
Fall 2008

6.005 elements of software construction

state machine invariants

Daniel Jackson

plan for today

topics

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

The big ideas today are: the idea of formulating properties so you can check that a design does what you think it does; the idea of invariants, an incredibly powerful idea of widespread application in computer science; and the idea of interlocks, an architecture for enforcing invariants locally.

traffic lights

problem

road works

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

Figure by MIT OpenCourseWare.

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)

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
- concretely, every reachable state satisfies a property

  eg, not green in both directions

This is a classic distinction in computer science. The technical idea is that safety properties can be refuted by witnesses: you can show the judge the violating trace. But liveness properties have no finite witnesses: if I was foolish enough (to pick a random example :-) to buy a security that promised to pay me back eventually, but didn't specify when, the my complaint can always be rebutted by the argument that I just haven't waited long enough. Liveness properties such as not deadlocking are useful as necessary checks, because certainly if your interactive system can get into a state in which nothing subsequently happens, you're in trouble. So they're useful as general notions for algorithms. But for software engineering, liveness properties are never good enough, since eventually isn't what you want, and when you formulate the property in terms of a deadline -- the acknowledgement is sent within 1s -- it becomes a safety property (ie, that the clock does not advance one second before the ack happens).

We've said that the behavior of a state machine is its trace set, and a nice, abstract way to formulate properties is to stick to the language of traces. In that case, a property is a predicate on traces that tells you "this trace is good, this trace is bad". Or, equivalently, a property is a trace set, and the traces of the machine should be a subset. But in practice, we're usually reasoning about the machine concretely, so it makes sense to express a property in terms of states. In general, we need "temporal properties" that say what kinds of transitions can happen. Much of the time, though, a simpler and very powerful idea is good enough. We just classify the states into good states satisfying some property and bad states that don't, and we claim that every reachable state is in the good set.

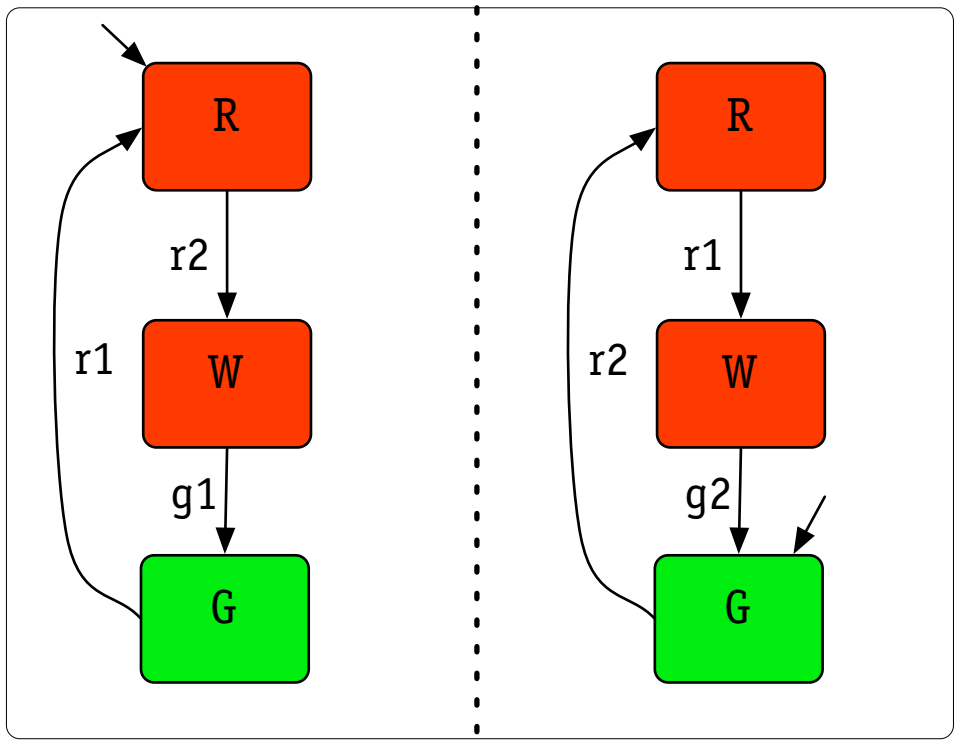
simpler traffic lights

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



Here's a model for a simpler traffic light scheme, without the message passing. We'll consider it first as it's easier to see what's going on in a simpler model. Note that r1 and r2 are shared, but g1 and g2 are not. So the machines have to synchronize on the r1 and r2 events, and make a transition together. We don't allow one to sit idle while the other one takes one of those event steps. But on g events, the machines can move independently. You may wonder where these rules come from. They're just one particular formalism that happens to work very nicely. Our syntax is actually based on David Harel's Statecharts, but the semantic rule we're using for how concurrent submachines execute is much simpler than the rule for Statecharts, and is based instead on Tony Hoare's Communicating Sequential Processes.

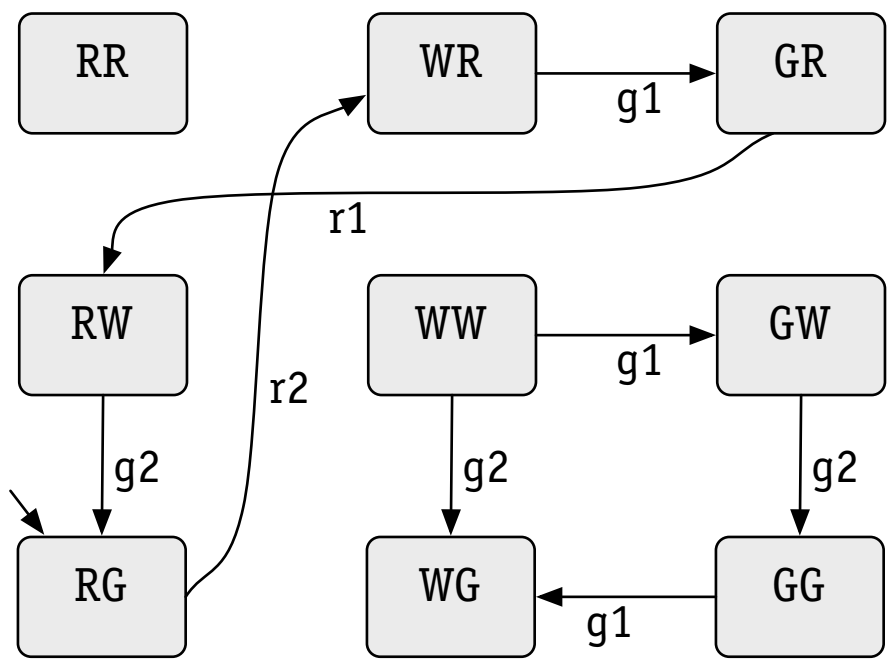
product machine

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!



In this case, we can just draw the product machine. But if there were 3 machines of 10 states each -- not a much larger diagram -- the product machine would have 1000 states!

traces revisited

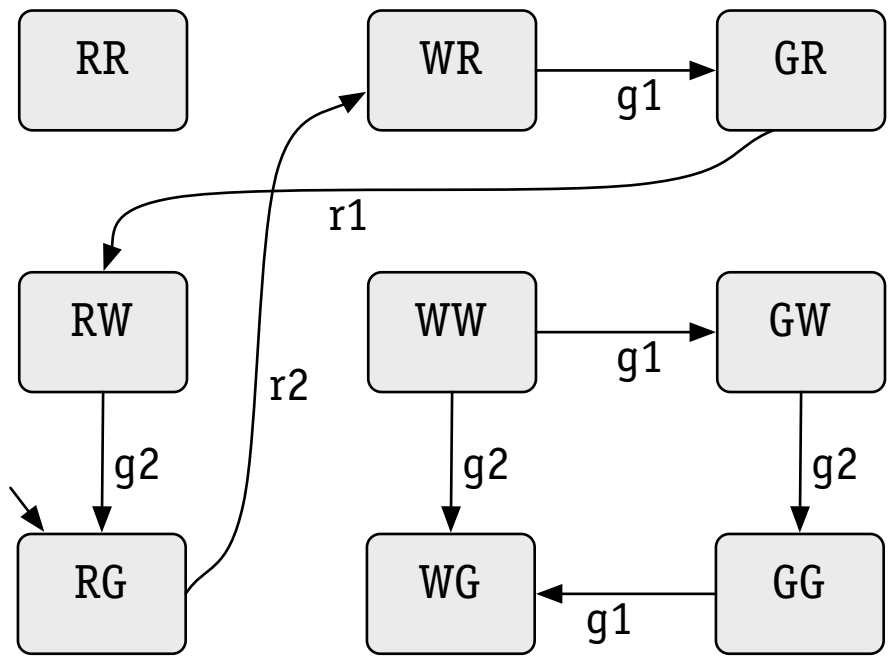
what's a trace?

- 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>, ...



You might want to check these traces against the diagram on slide 6 to check that you understand the parallel execution rule.

checking traffic light property

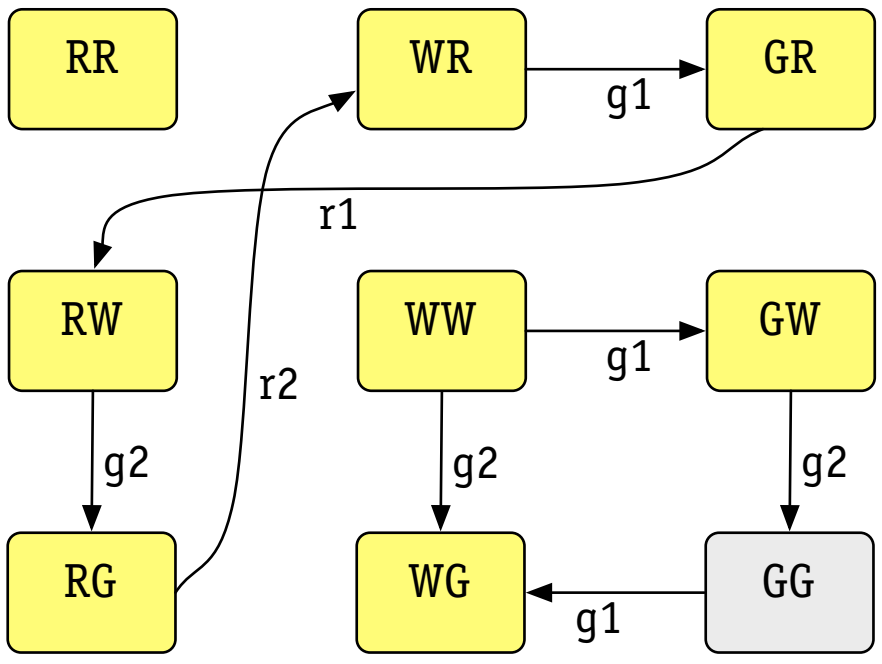
what's the traffic light property?

- 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

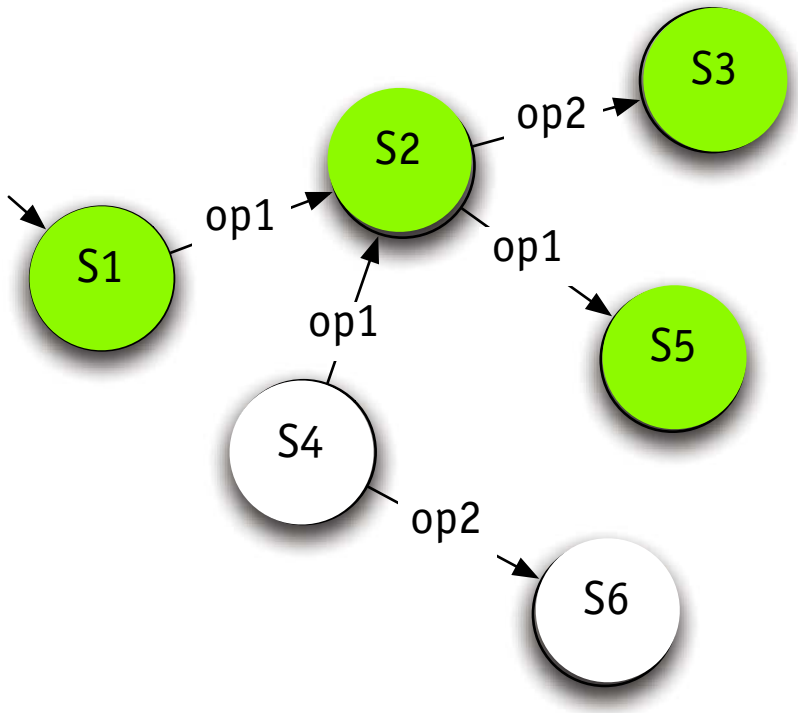
- how would you do this for 3 parallel machines of 10 states each?

state properties & invariants

property as state set

state property is equivalently

- predicate P(s) applied to state s
- subset {s: S | P(s)} of states



The green coloring shows the states satisfying our property P.

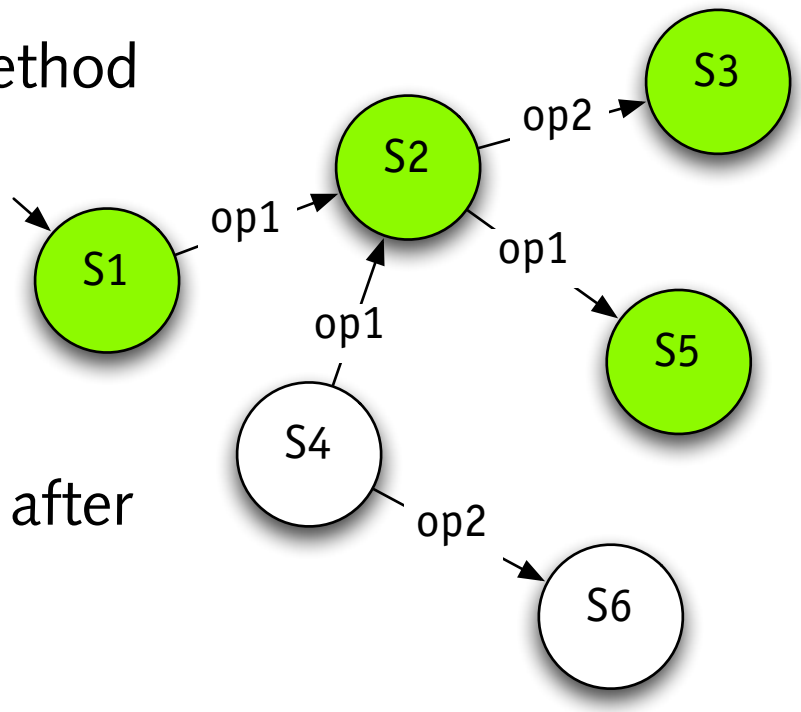
invariant reasoning

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 ⇒ property holds after
- if so, property is **"invariant"**



example

- initial state is green, and each op preserves greenness
- so greenness is an invariant

Note that every green state has a green successor. So greenness is "preserved".

why invariants work

strategy

- check property holds in initial state (1)

  $I(S_0)$

- check that transitions preserve property (2)

  $(s, e, s') \in R \wedge I(s) \Rightarrow I(s')$

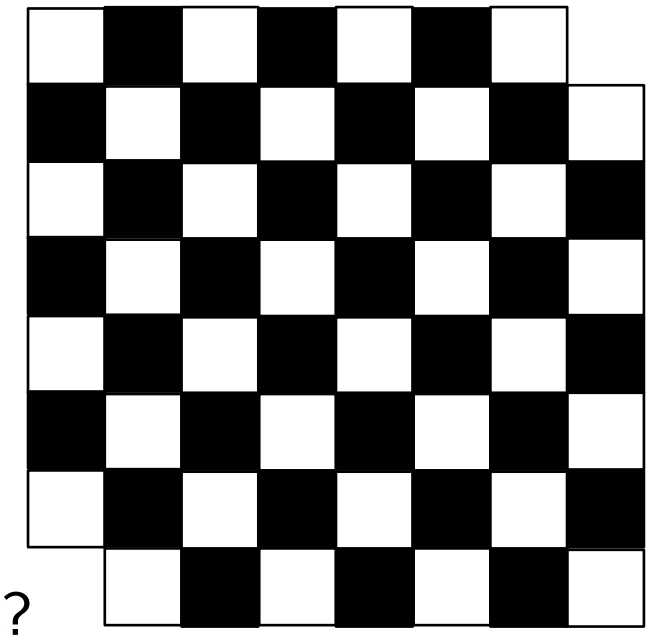
- then property is an invariant, and holds in all reachable states

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?)

Here's the standard mathematical definition of state invariants. I've expressed it in terms of events e, but in practice, this technique is usually applied to a textual version of the state machine, and you consider take each operation at a time, and consider whether that operation preserves the invariant -- with the operation standing for all the events in the same event class (that is, the transitions with the same label). We won't pursue this on state machine models, although you can see examples of it in the slides for Spring 2008. But we will do this for code when we come to representation invariants.

tiling the chessboard



a classic problem

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

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

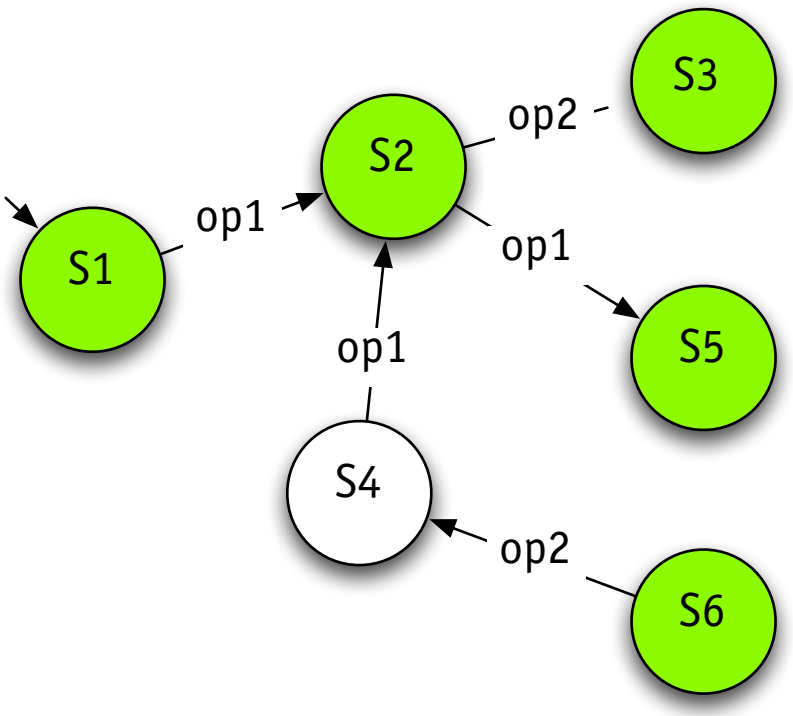
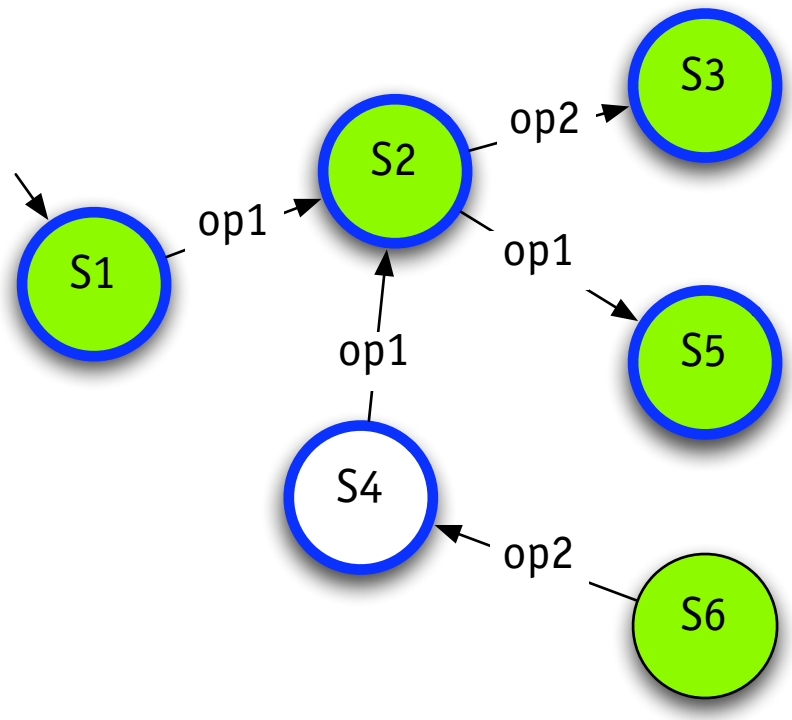


diagram (upper)

- **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



In this machine (showing in both diagrams), greenness is not an invariant. But the green states that lead to non-green states aren't reachable, so they shouldn't matter. The way we account for that is by "strengthening" the property to make it an invariant, essentially adding information about reachability. Here, conceptually, we've strengthened the property to "green and blue", and that's now an invariant. We'll see concrete examples of this soon.

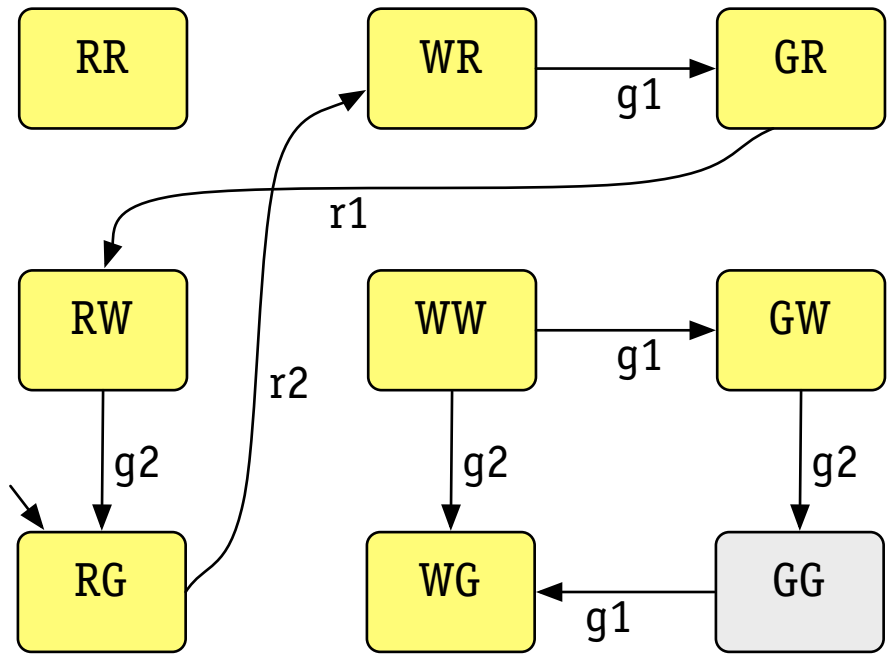
S3

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



Here's the idea applied to our simple traffic light system. The property we want to express is that both lights aren't green at once -- shown in yellow. But this isn't an invariant, because of the yellow to grey transition. It doesn't matter because the originating state is not reachable, but we have to show that.

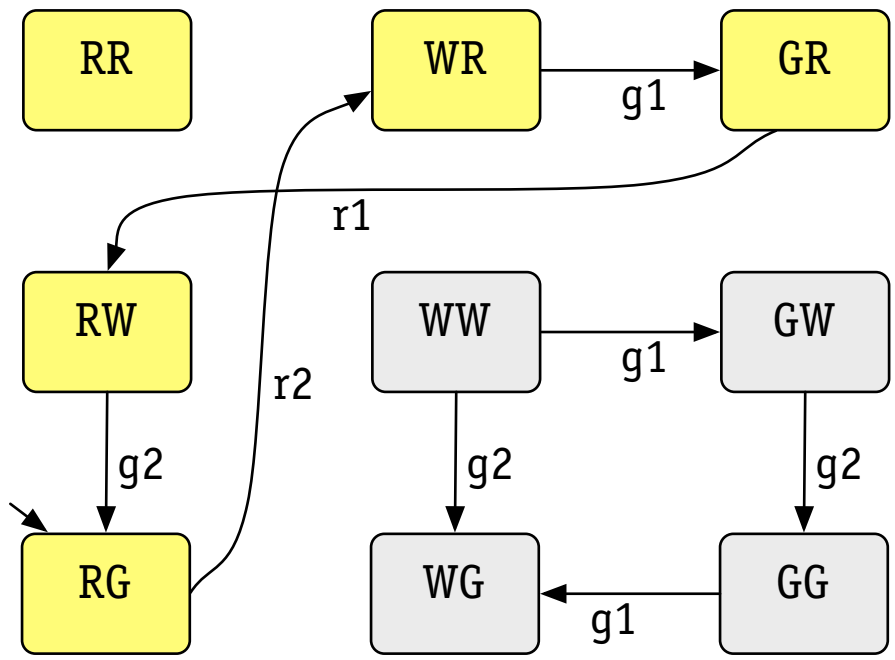
getting to the invariant

what's wrong?

- need to strengthen the property
- an invariant is

  R1 **or** R2
- which implies

  not GG



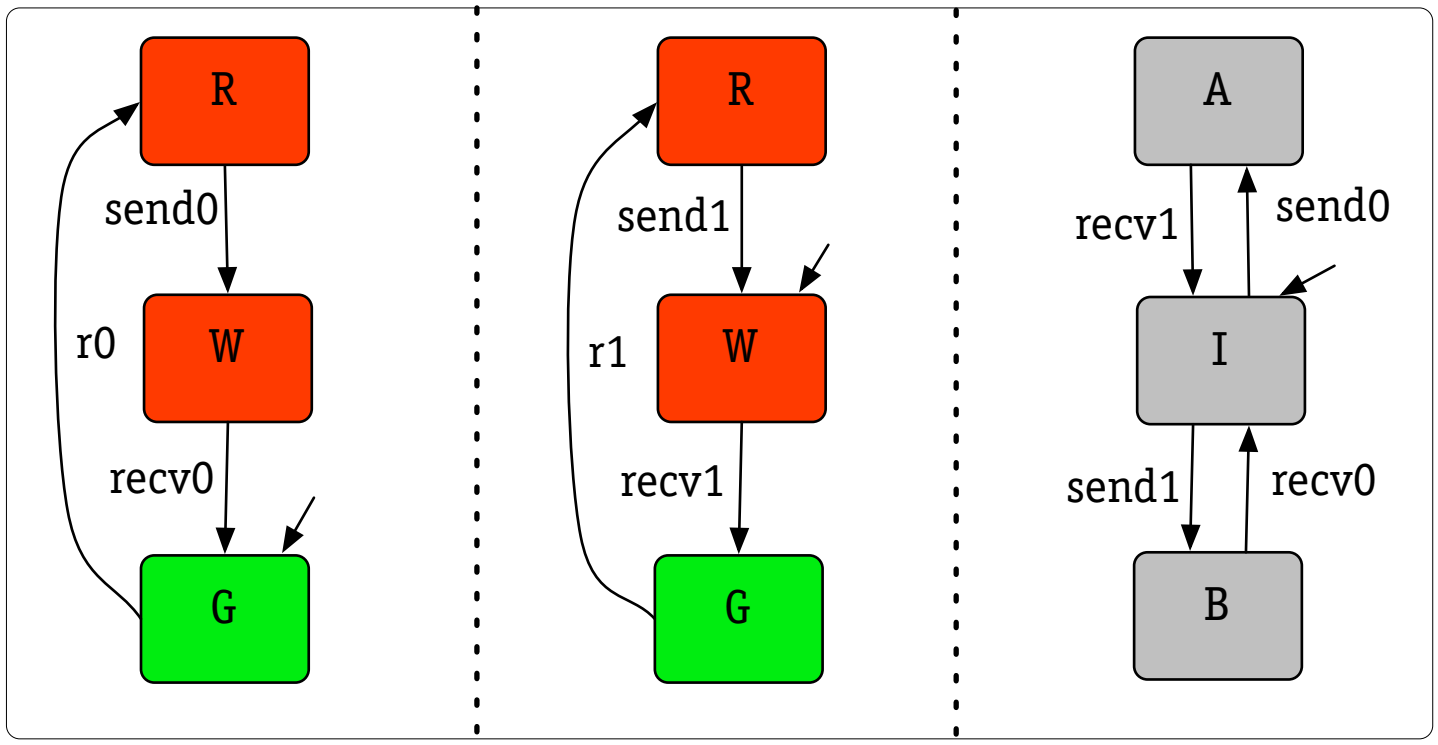
Here's how we rule out the unreachable states. We strengthen the property to say that at least one light shows red, and that is now an invariant.

The RR state, btw, represents a deadlock. There's no way out of it. Look back at slide 6 and check that you understand why.

Note that for the purpose of invariant reasoning, you can shift the initial state to any state that satisfies it. So any of the yellow states in the cycle will do as starting states for the protocol.

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

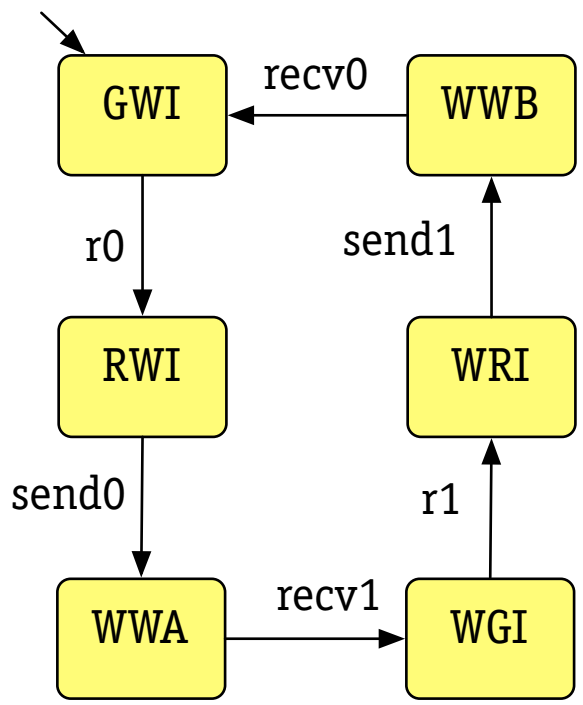
Here's the state machine model for the roadworks with message passing. The message passing mechanism is modelled explicitly by the machine on the far right. The convention in name events is that the numerical suffix says which machine does the event, so send0 means that 0 does the sending.

An alternative would be to extend the formalism with asynchronous message passing, but it's always better to stick to the simpler formalism you have if you can. In this case, modelling the message buffer is easy. if we wanted to model an unbounded buffer, we could use the textual notation, with a state component representing the buffer that's just like the variable we used in the midi-piano to record the sequence of events. An advantage of making the buffering explicit is that it then becomes easy to model various failure modes; if you wanted the protocol to be resistant to the dropping of messages, for example, you could just add a drop event with transitions from A to I and B to I, and check that the protocol still works.

product machine

showing reachable states only

- 27 possible state combinations, but only 6 reachable
- getting harder to check reachable states by eye...



I had to step through the machine to extract these reachable states. It was much easier than it usually is, because there's no non-determinism here: every state has only one successor.

is the property an invariant?

the desired property is

```
not GGx    // x 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

```
recv0 from WGB to GGI
```

- can't happen: when second process is green, no message waiting for first

After class, someone asked about a violating transition from GWB. That couldn't happen because recv1 is not enabled in state B of the message buffer.

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

Finding this invariant isn't easy; an invariant like this usually captures the essence of the design -- in this case that whenever there is a message in motion both sides are waiting: the receiving side because it must be waiting to get the message, and the sending side because it starts waiting for the return message.

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

For examples of model checkers, google: SMV, LTSA, SPIN, Pathfinder, FDR.

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")

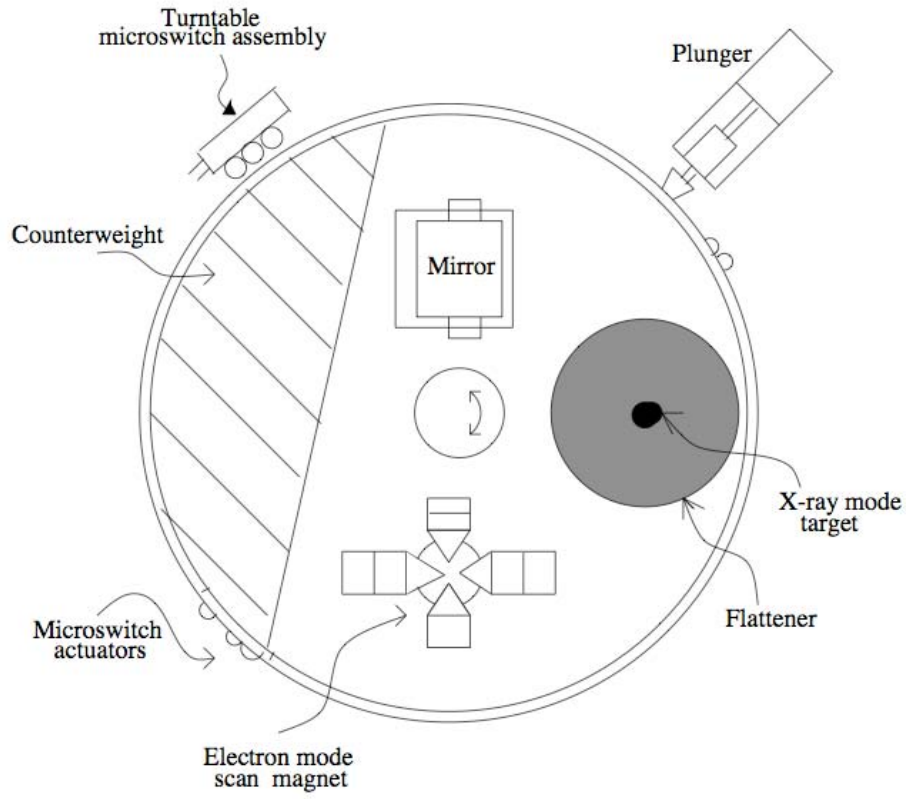
The terms "optimistic" and "pessimistic" are usually applied more narrowly in the area of transactions and concurrency control: how to deal with concurrent users of a database or shared resource.

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



Courtesy of Nancy Leveson. Used with permission.

what happened

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

diagram from Nancy Leveson, "The Therac-25 Accidents"; see http://sunnyday.mit.edu/papers/therac.pdf

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;
From: Security Absurdity: The Complete, Unquestionable, And Total Failure of Information Security
Noam Eppel;

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  // only if 0 ≤ i < 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

Interlocks are very useful, and you should use them as much as you can. At the lowest level, this just means making good use of runtime assertions in your code. But at a higher level, you should think when you design a system about what properties are most critical, and whether you could ensure them by a localized check. Another nice example of an interlock is doing a checksum in a file transfer protocol: when you've transferred the file, you can compute the checksum, and get an immediate confirmation that the transfer worked. If it didn't, you just run it again.

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

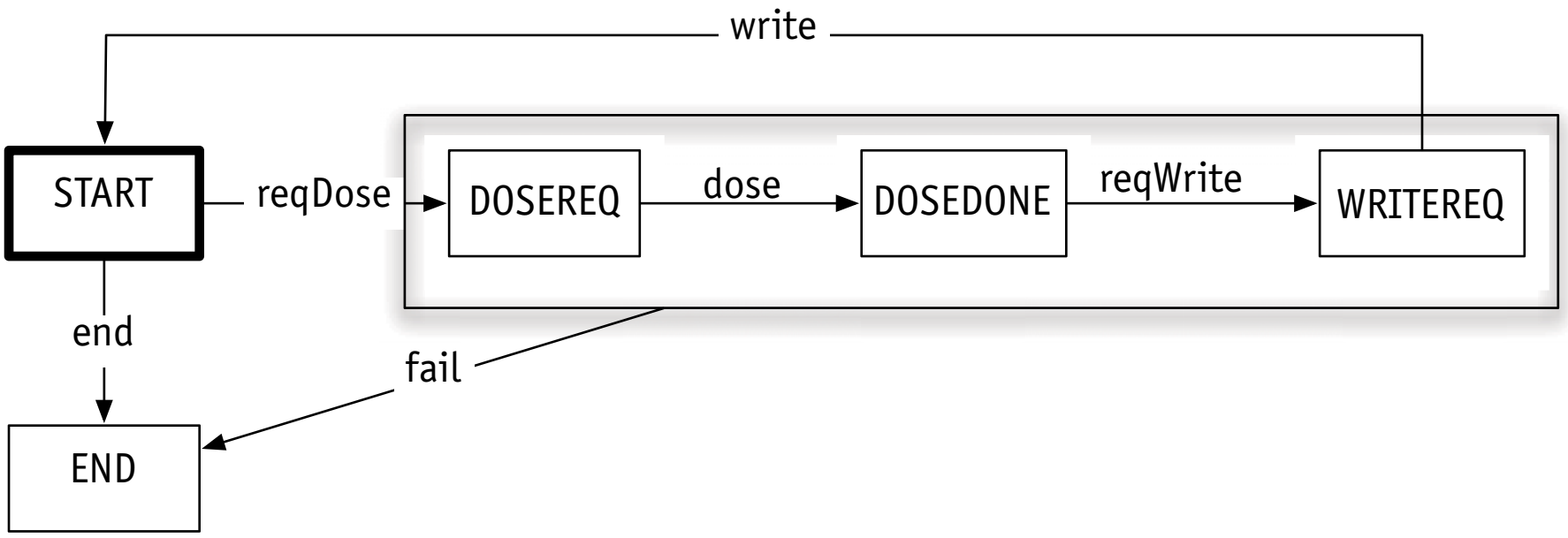
- give modular reasoning
- more on this later in the course

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 op dose: **post** d′ == d+1 **op** write: **post** r′ == r + 1
prove d - r ≤ 1

Here's a very simple example of invariant reasoning for a textual state machine. The problem -- a very real one -- is to record how much radiation has been given to a patient, when the action that delivers the dose, and the action that records it to disk, can both fail. The solution is to deliver the dose in small bits. If you always write after you dose, and the dose increment is 1, you can prove the invariant that the delivered dose d is always at most one greater than the recorded dose.