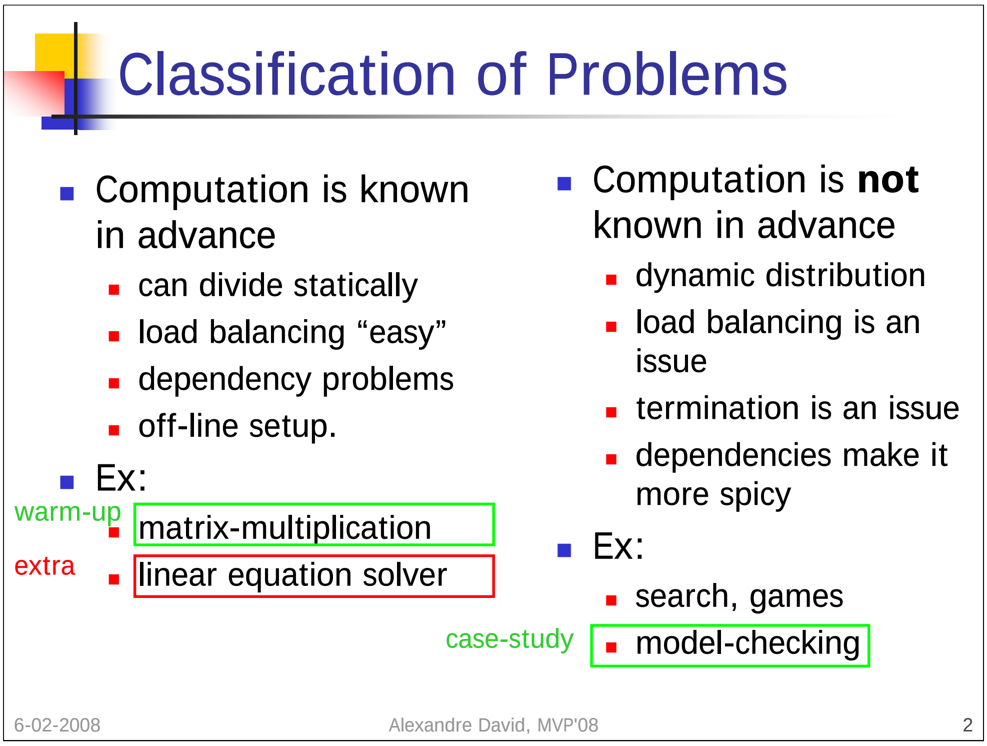
Real stuff!

Introduction to the Case-Study: A Model-Checker

Alexandre David

1.2.05

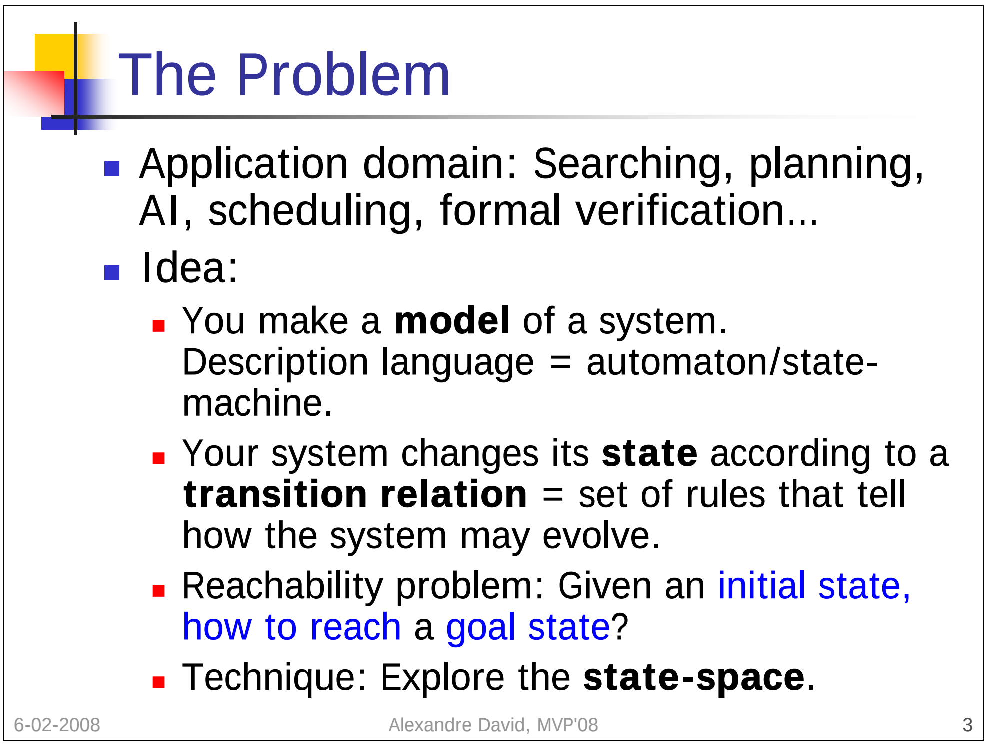
http://www.cs.aau.dk/~adavid/teaching/MVP-08/



Classification of Problems

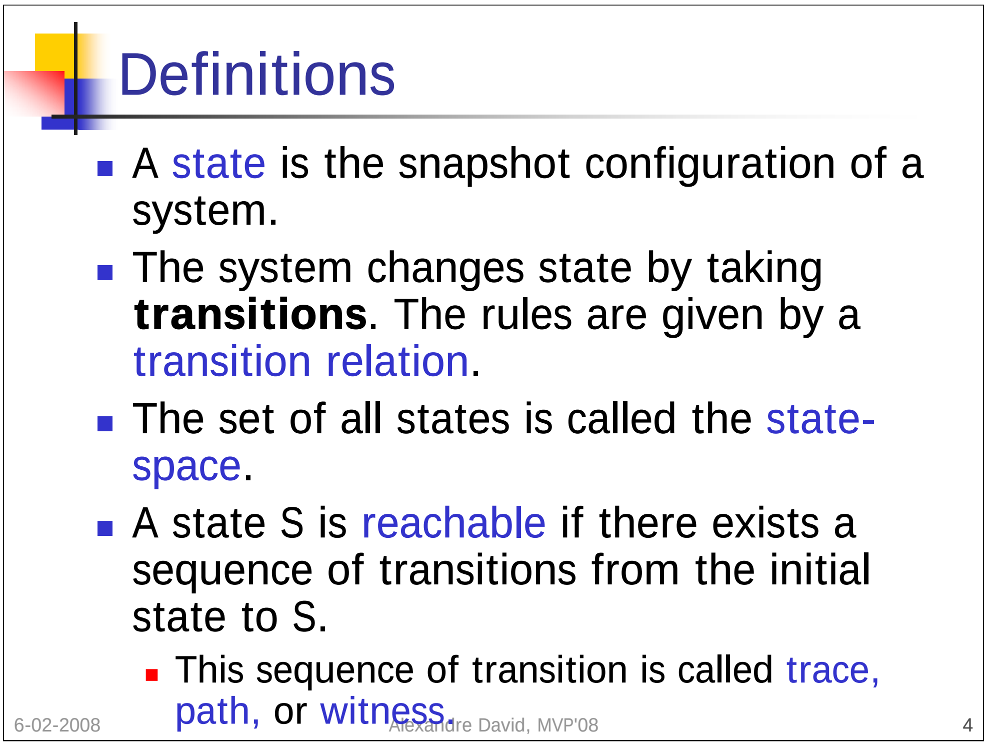
- Computation is known in advance
  - can divide statically
  - load balancing “easy”
  - dependency problems
  - off-line setup.
- Ex:
  - warm-up matrix-multiplication
  - extra linear equation solver
- Computation is **not** known in advance
  - dynamic distribution
  - load balancing is an issue
  - termination is an issue
  - dependencies make it more spicy
- Ex:
  - search, games
  - case-study model-checking

The book is a bit lacking on the dynamic side. The exercises complement this and address related issues.



The Problem

- Application domain: Searching, planning, AI, scheduling, formal verification…
- Idea:
  - You make a **model** of a system. Description language = automaton/state-machine.
  - Your system changes its **state** according to a **transition relation** = set of rules that tell how the system may evolve.
  - Reachability problem: Given an initial state, how to reach a goal state?
  - Technique: Explore the **state-space**.

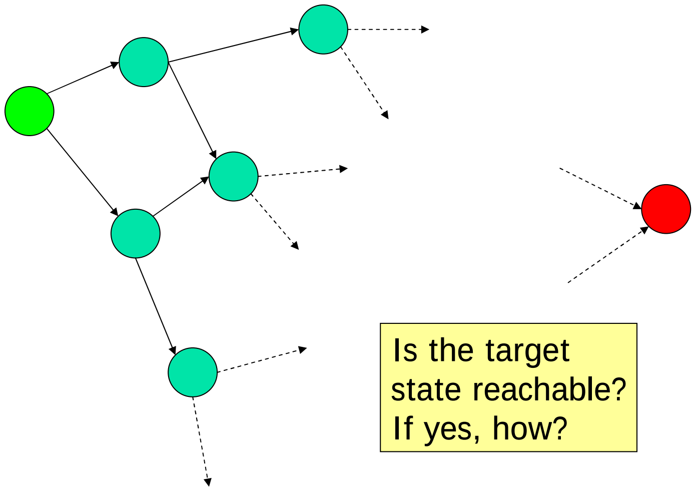


Definitions

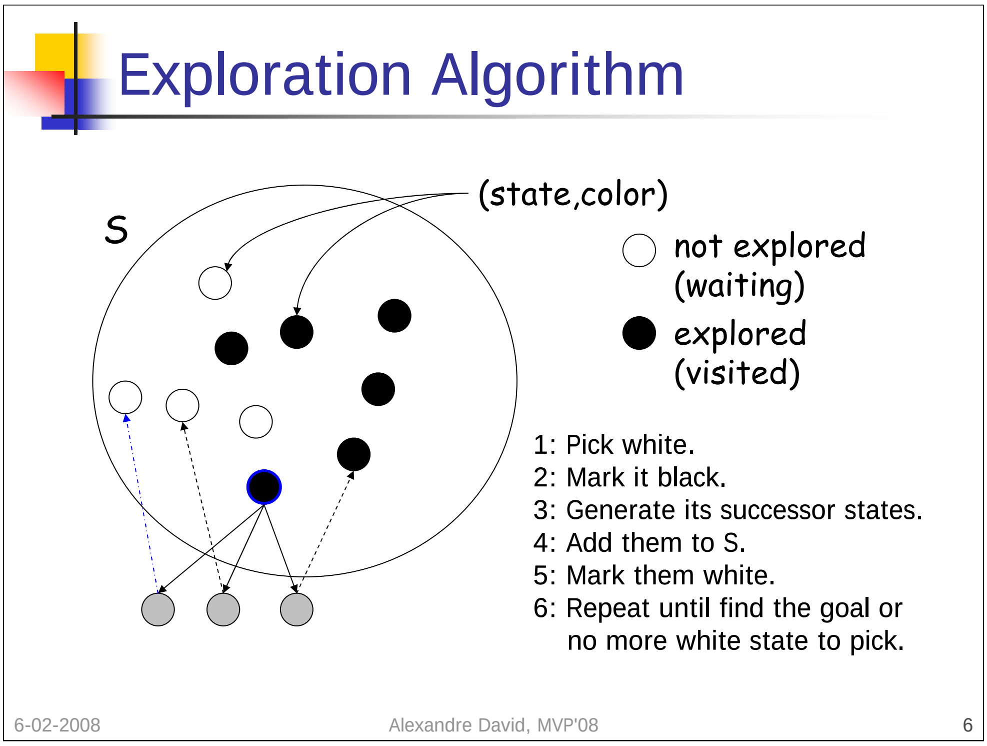
- A state is the snapshot configuration of a system.
- The system changes state by taking **transitions**. The rules are given by a transition relation.
- The set of all states is called the state-space.
- A state S is reachable if there exists a sequence of transitions from the initial state to S.
  - This sequence of transition is called trace, path, or witness.

A state is typically a tuple with the values of all the variables of the system. States also record “where the system” is terms of execution, like the instruction pointer, or from a model point-of-view the locations.

Searching, a.k.a. State-space Exploration



Is the target state reachable? If yes, how?

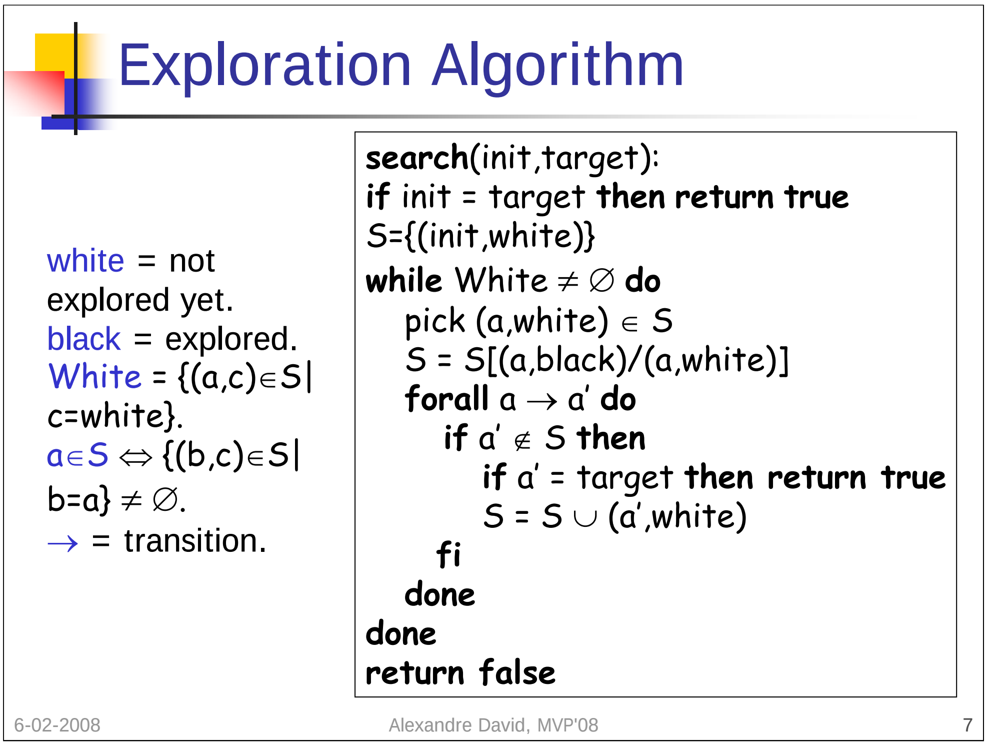


Exploration Algorithm

○ not explored (waiting)

● explored (visited)

1: Pick white.
2: Mark it black.
3: Generate its successor states.
4: Add them to S.
5: Mark them white.
6: Repeat until find the goal or
   no more white state to pick.



Exploration Algorithm

white = not explored yet.

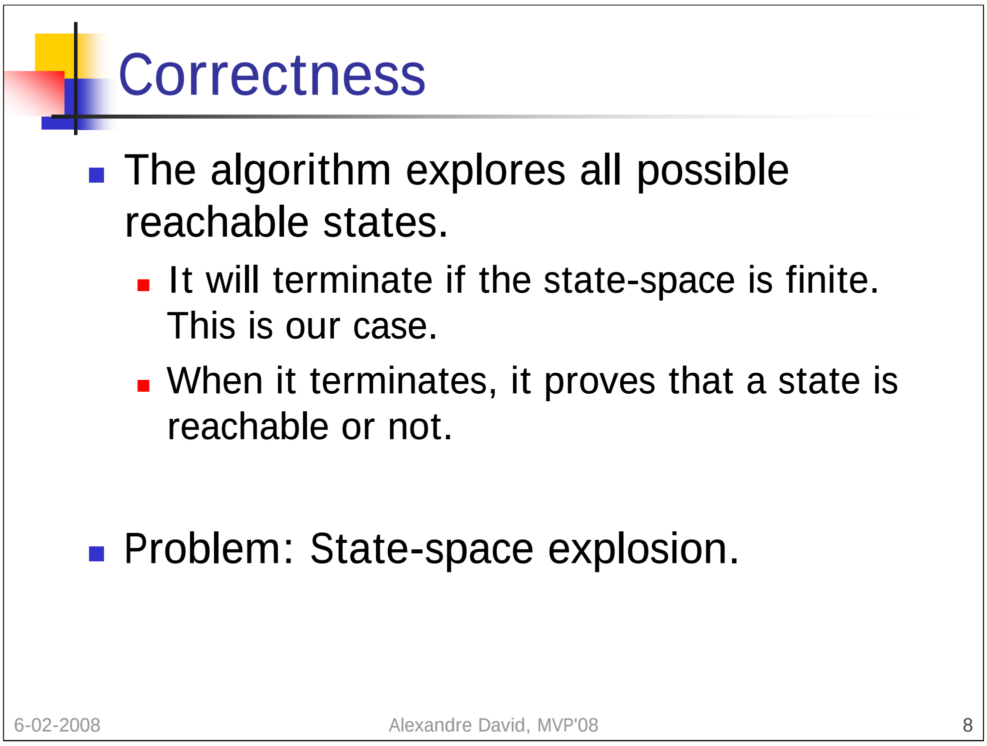
black = explored.

White = $\{(a,c) \in S \mid c = white\}$.

$a \in S \Leftrightarrow \{(b,c) \in S \mid b = a\} \neq \emptyset$.

→ = transition.

```
search(init,target):
if init = target then return true
S={(init,white)}
while White $\neq \emptyset$ do
    pick (a,white) $\in$ S
    S = S[(a,black)/(a,white)]
    forall a $\rightarrow$ a' do
        if a' $\notin$ S then
            if a' = target then return true
            S = S $\cup$ (a',white)
        fi
    done
done
return false
```

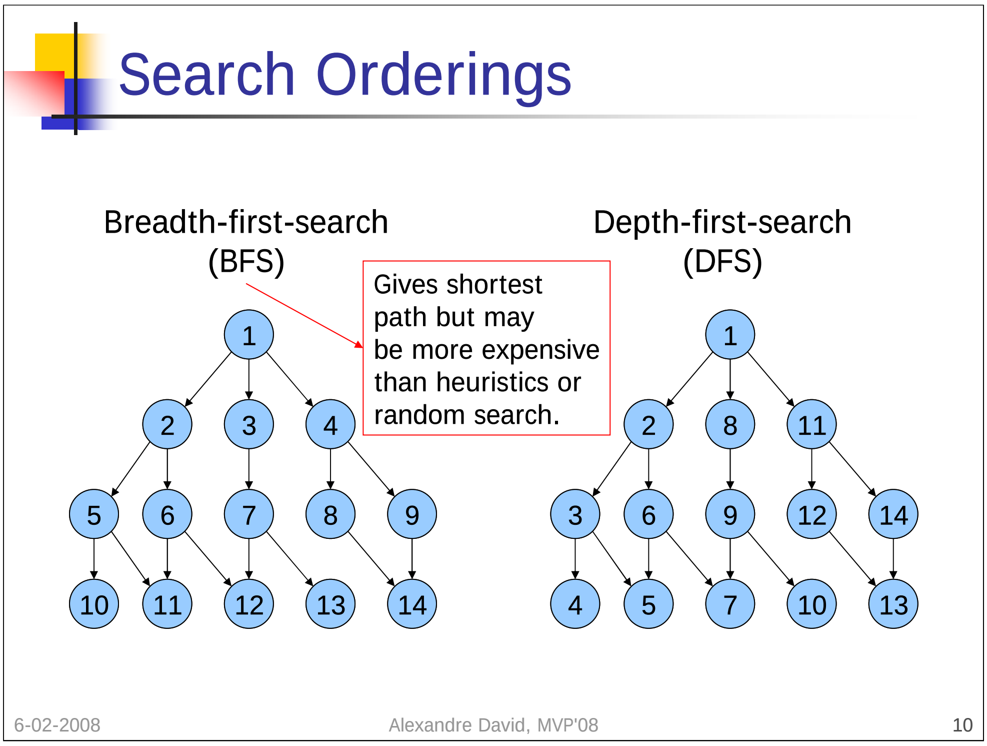


Correctness

- The algorithm explores all possible reachable states.
  - It will terminate if the state-space is finite. This is our case.
  - When it terminates, it proves that a state is reachable or not.
- Problem: State-space explosion.

Technicalities

- How to represent S for efficient look-up?
  - Hash table.
- How to pick-up the next state to be explored?
  - FIFO: Breadth-first-search.
  - LIFO: Depth-first search.
  - Priority queue: Guided search with heuristics.



Search Orderings

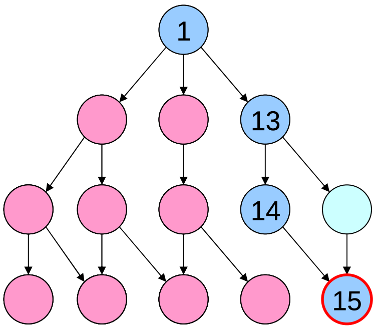
Breadth-first-search (BFS)

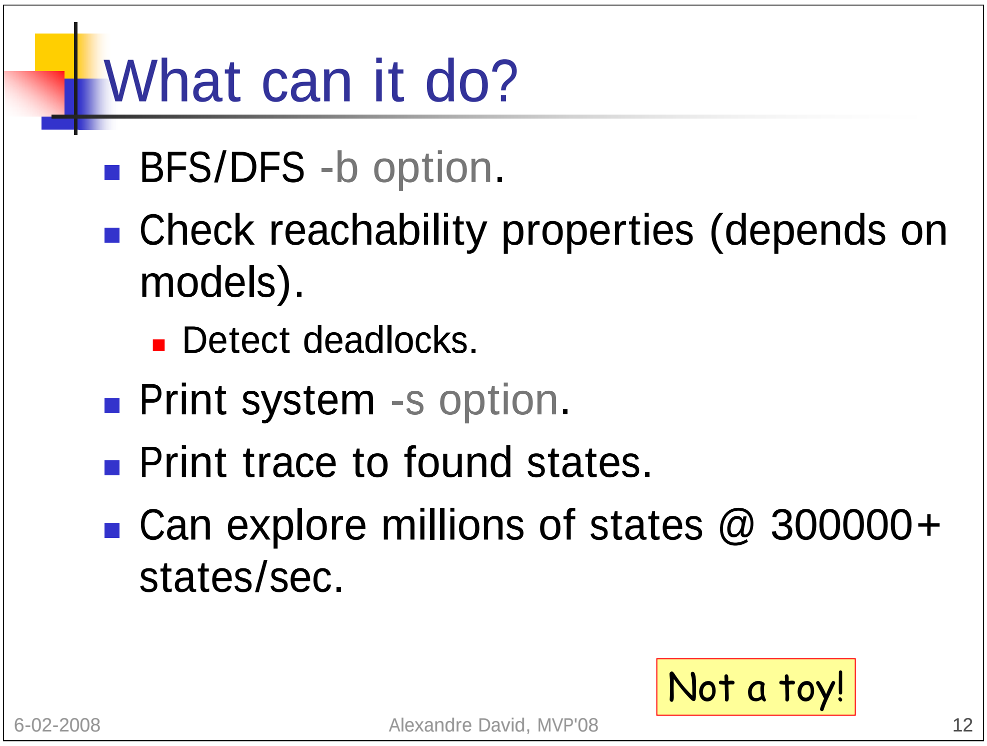
Gives shortest path but may be more expensive than heuristics or random search.

Depth-first-search (DFS)

Clean-up deadlocks – DFS

Depth-first-search
(DFS)

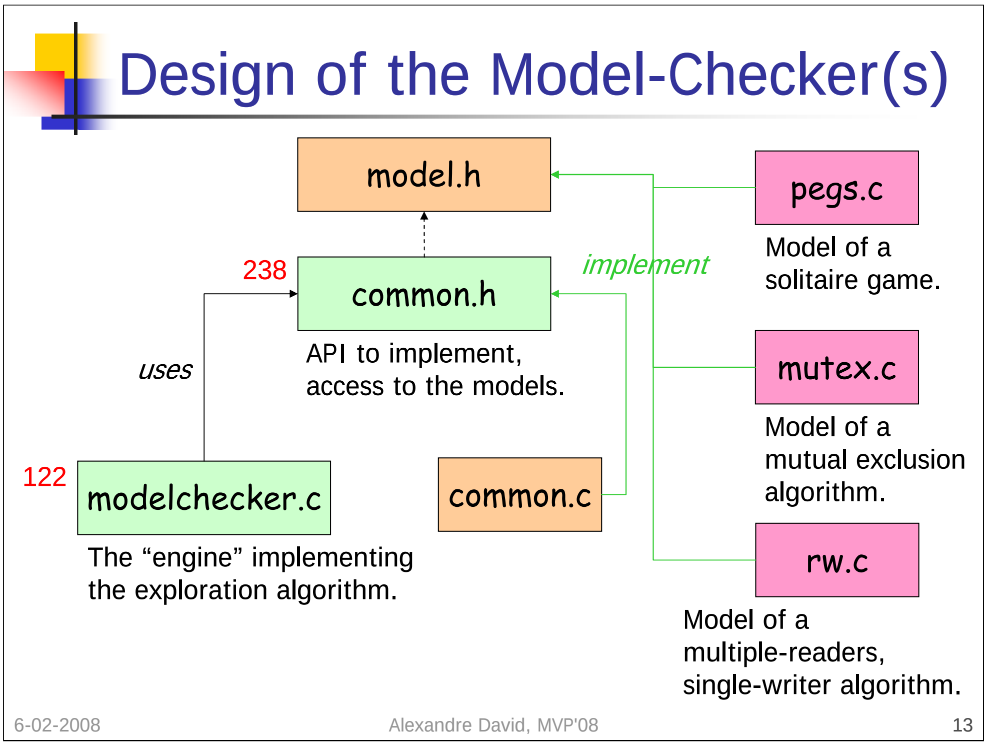




What can it do?

- BFS/DFS -b option.
- Check reachability properties (depends on models).
  - Detect deadlocks.
- Print system -s option.
- Print trace to found states.
- Can explore millions of states @ 300000+ states/sec.

Not a toy!



Design of the Model-Checker(s)

model.h

238 common.h

API to implement, access to the models.

uses

122 modelchecker.c

The “engine” implementing the exploration algorithm.

common.c

implement

pegs.c

Model of a solitaire game.

mutex.c

Model of a mutual exclusion algorithm.

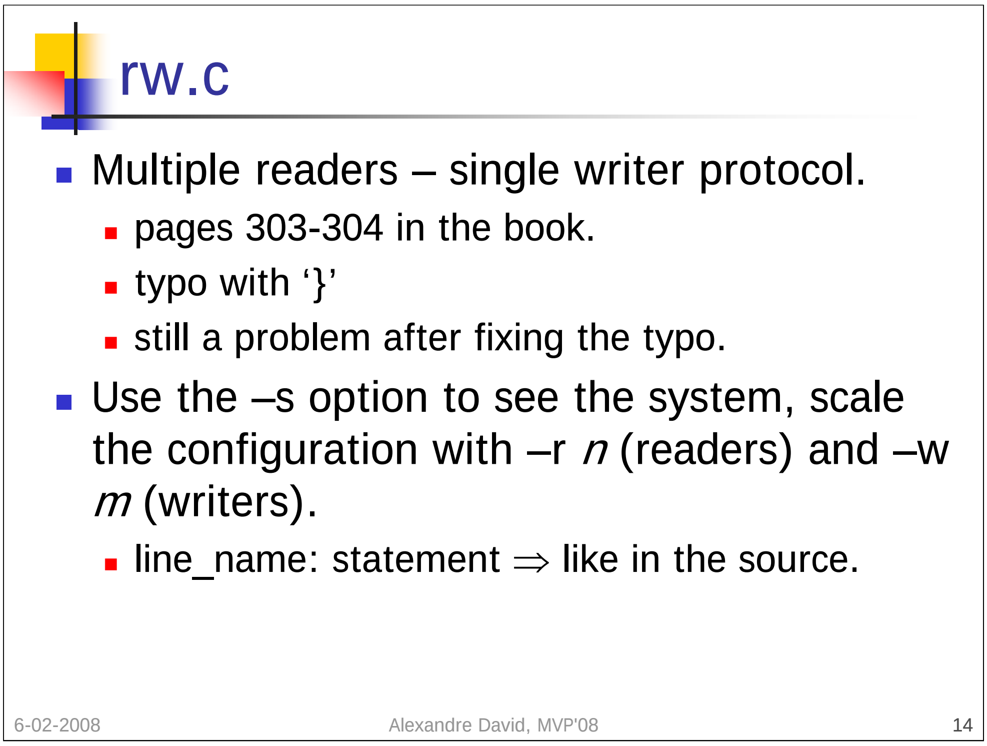
rw.c

Model of a multiple-readers, single-writer algorithm.

Red: You are not advised to read.

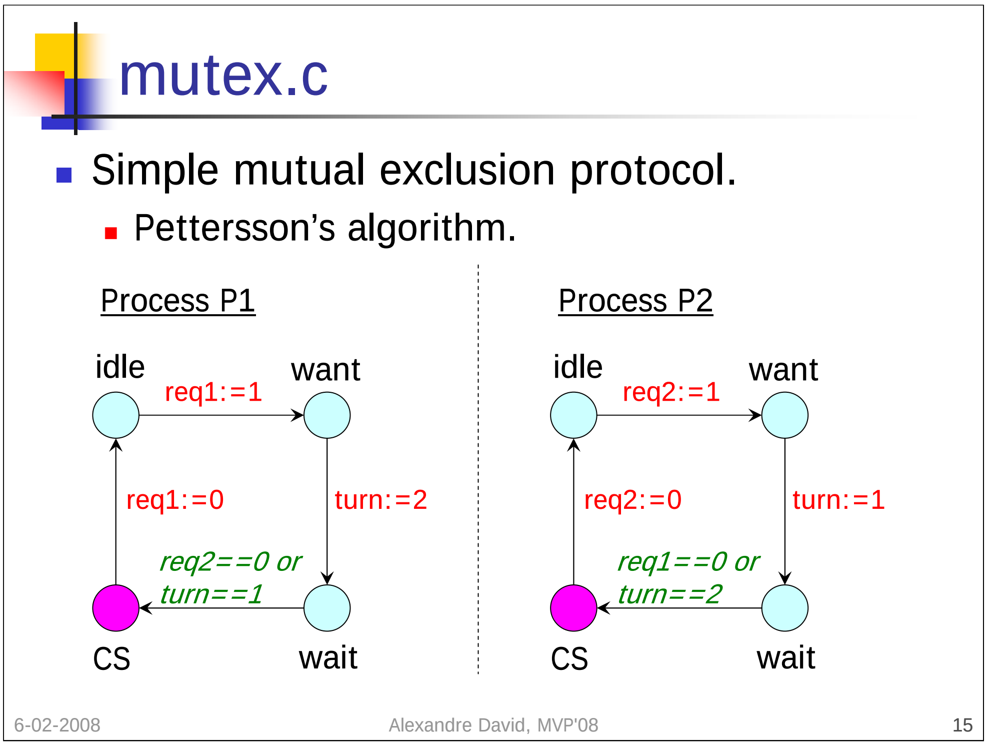
Orange: You do not need to read.

Green: Read and understand.



rw.c

- Multiple readers – single writer protocol.
  - pages 303-304 in the book.
  - typo with ‘}’
  - still a problem after fixing the typo.
- Use the –s option to see the system, scale the configuration with –r *n* (readers) and –w *m* (writers).
  - line_name: statement ⇒ like in the source.

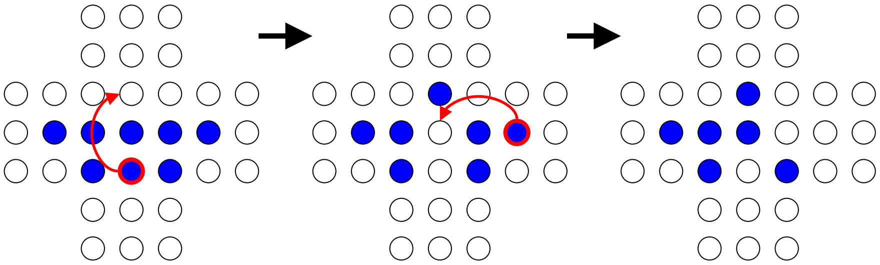


mutex.c

- Simple mutual exclusion protocol.
  - Pettersson’s algorithm.

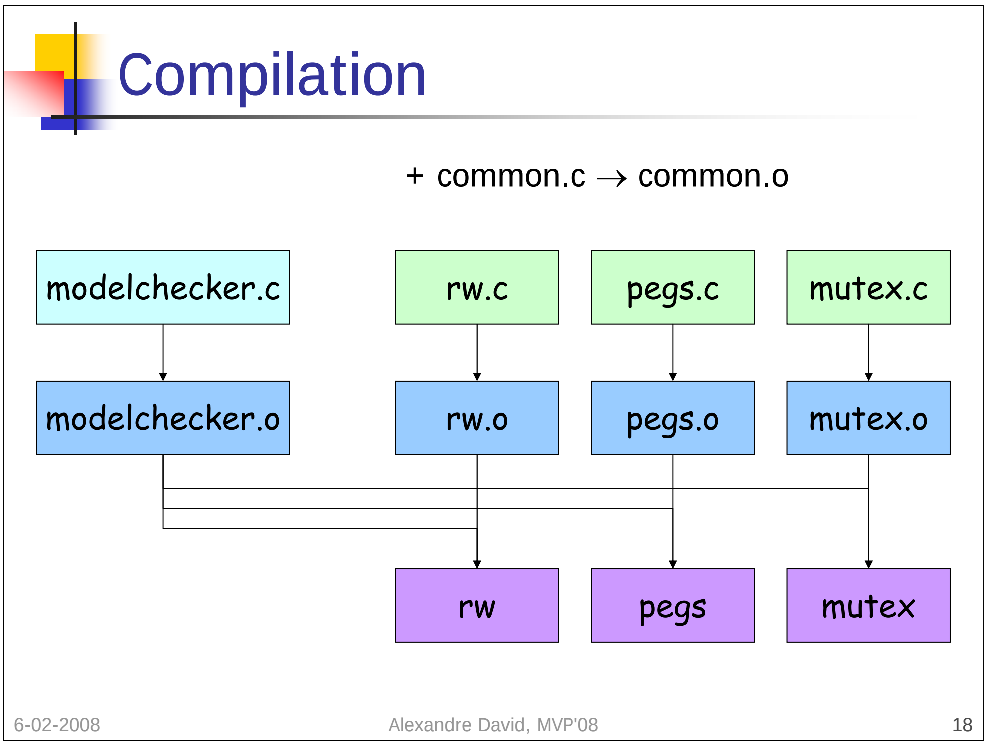
pegs.c

- Simple game:
  - initial configuration with *n* pegs → get 1 left.



Why?

- rw.c:
  - real protocol used in the book
  - easily scalable, loops etc...
- mutex.c:
  - simple and good for testing races.
- pegs.c:
  - state-graph = wide tree with known height,
  - fun.



Compilation

+ common.c → common.o

modelchecker.c

rw.c

pegs.c

mutex.c

modelchecker.o

rw.o

pegs.o

mutex.o

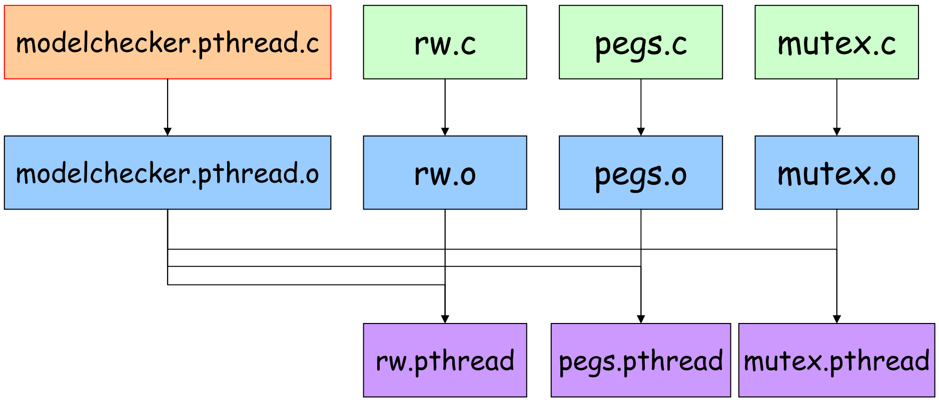
rw

pegs

mutex

Compilation - pthread

+ utils.pthread.c → utils.pthread.o



modelchecker.pthread.c

rw.c

pegs.c

mutex.c

modelchecker.pthread.o

rw.o

pegs.o

mutex.o

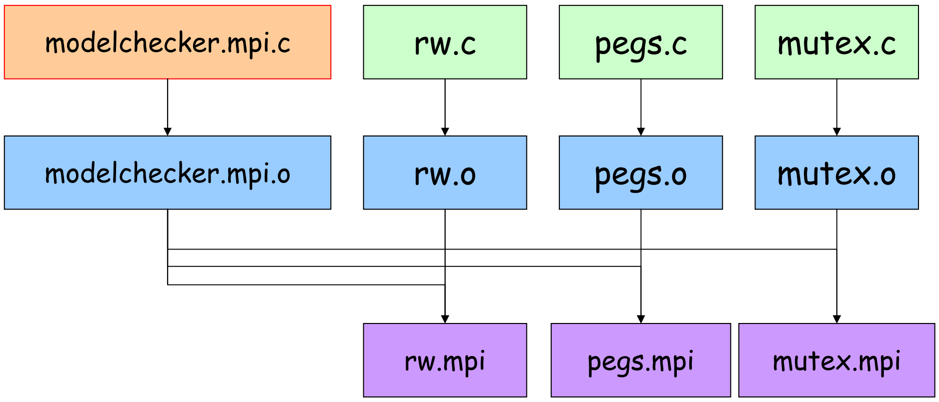
rw.pthread

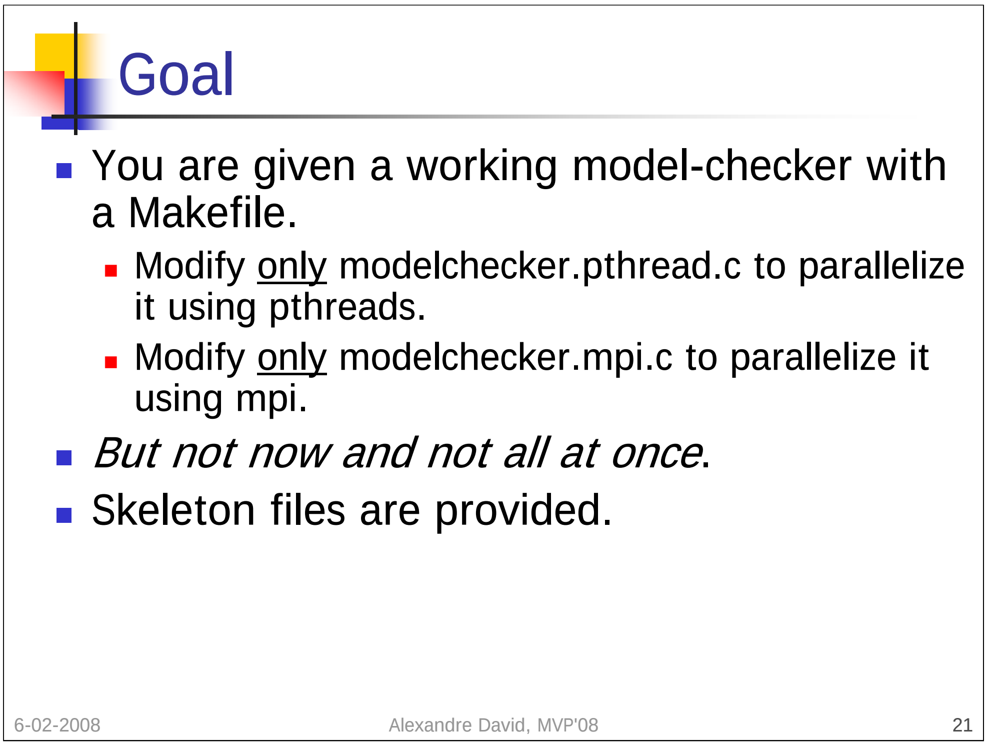
pegs.pthread

mutex.pthread

Compilation - mpi

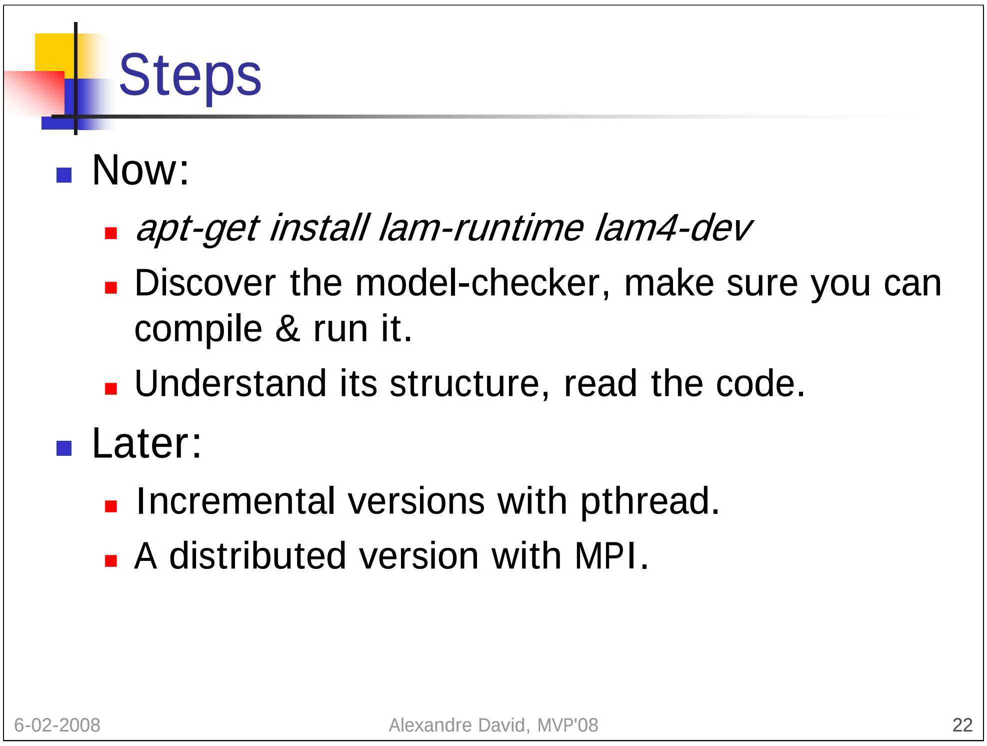
+ utils.mpi.c → utils.mpi.o





Goal

- You are given a working model-checker with a Makefile.
  - Modify only modelchecker.pthread.c to parallelize it using pthreads.
  - Modify only modelchecker.mpi.c to parallelize it using mpi.
- *But not now and not all at once*.
- Skeleton files are provided.



Steps

- Now:
  - *apt-get install lam-runtime lam4-dev*
  - Discover the model-checker, make sure you can compile & run it.
  - Understand its structure, read the code.
- Later:
  - Incremental versions with pthread.
  - A distributed version with MPI.