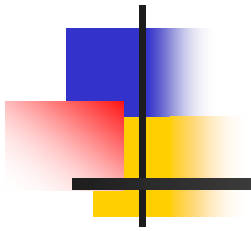


Real stuff!

Introduction to the Case-Study: A Model-Checker

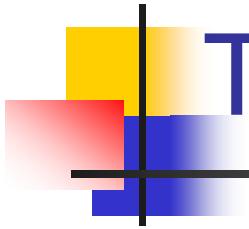
Alexandre David

B2-206

Classification of Problems

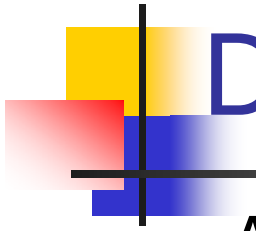
- Computation is known
  - can divide statically
  - load balancing “easy”
  - dependency problems
  - off-line setup.
- Ex:
  - warm-up: matrix-multiplication
  - extra: linear equation solver

- Computation is **not** known in advance
  - dynamic distribution
  - load balancing is an issue
  - dependencies make it more spicy
- Ex:
  - search, games
  - case-study: model-checking



The Problem

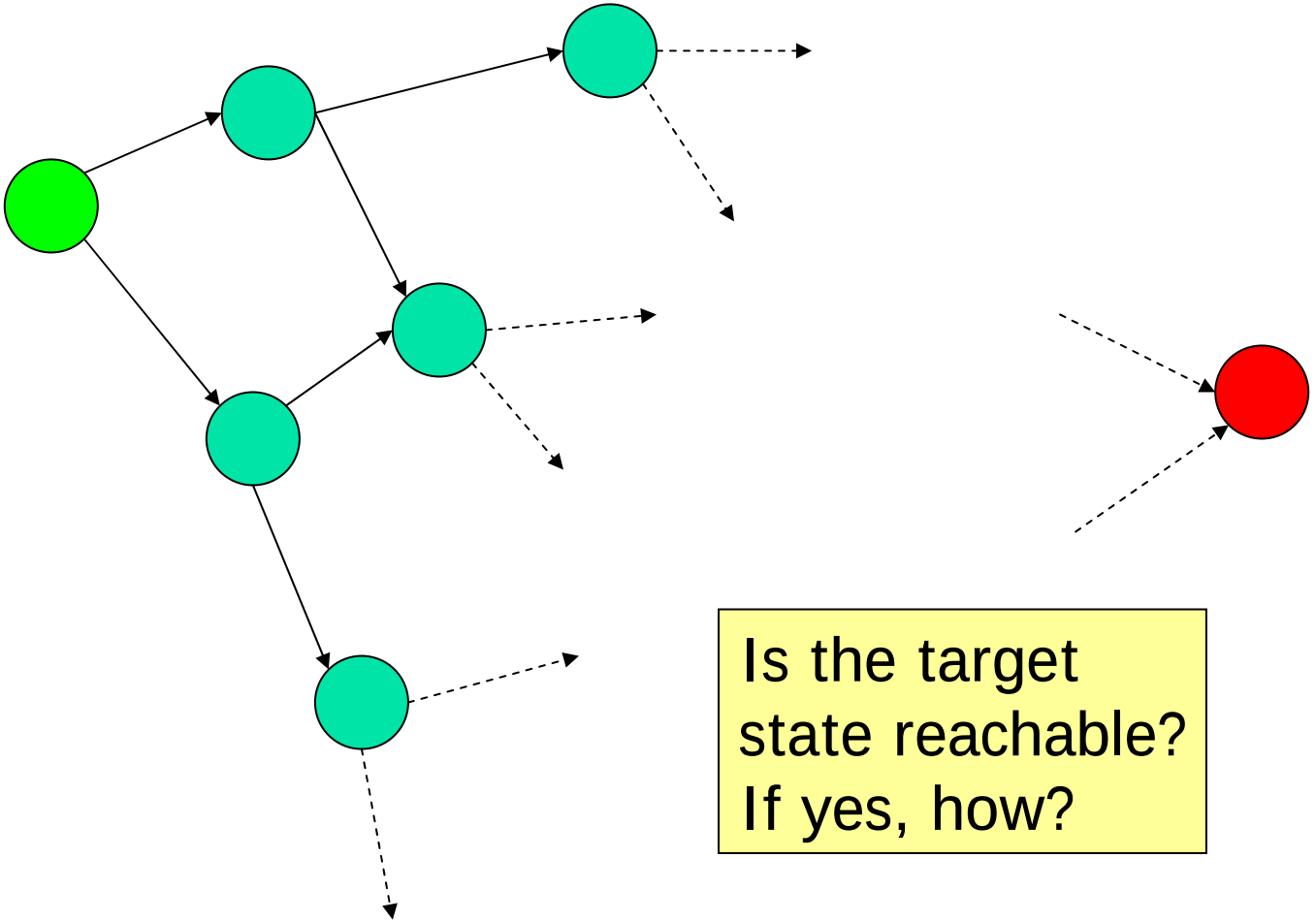
- Application domain: Searching, planning, AI, scheduling, formal verification...
- Idea:
  - You make a **model** of a system.
    Description language = automaton/state-machine.
  - Your system changes its **state** according to a **transition relation** = set of rules that tell how the system may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



Definitions

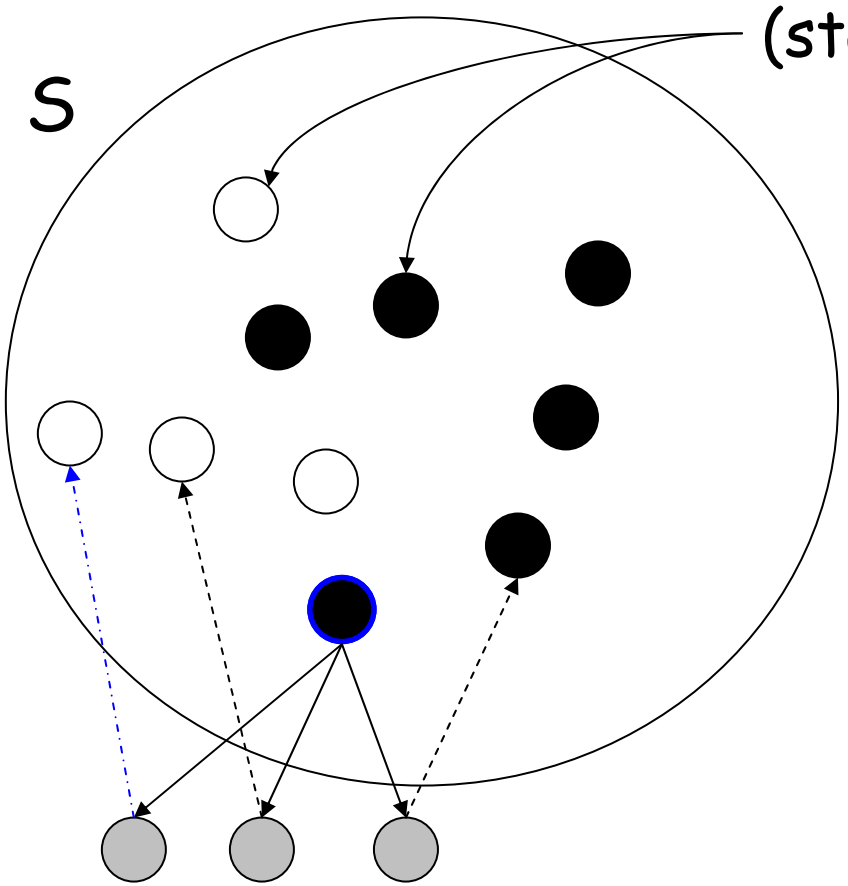
- A state is the snapshot configuration of a system, typically a tuple with the values of all the variables of the system.
- The system changes state by taking **transitions**. The rules are given by a transition relation.
- The set of all states is called the state-space.
- A state S is reachable if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

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



Is the target state reachable? If yes, how?

Exploration Algorithm



(state,color)

not explored

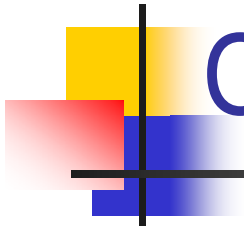
explored

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.

Exploration Algorithm

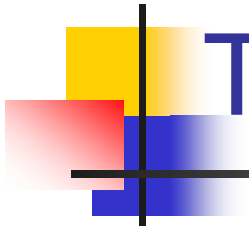
white = not explored yet.
black = explored.
White = $\{(a,c) \in S \mid c=\text{white}\}$.
$a \in S \Leftrightarrow \{(b,c) \in S \mid b=a\} \neq \emptyset$.
$\rightarrow$ = transition.

```
search(init,target):
S={(init,white)}
if init = target then return true
while White ≠ ∅ do
    pick (a,white) ∈ S
    S = S[(a,black)/(a,white)]
    forall a → a' do
        if a' ∉ S then
            S = S ∪ (a',white)
            if a' = target then return true
        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

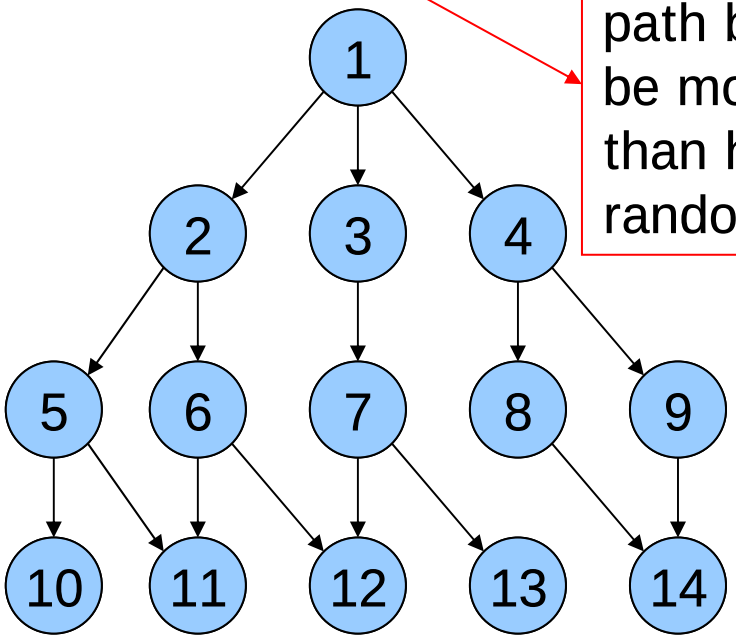
- How to represent S?
  - Hash table.
- How to pick-up the next state to be explored?
  - FIFO: Breadth-first-search.
  - LIFO: Depth-first search.
  - Priority queue: Guided search with heuristics.

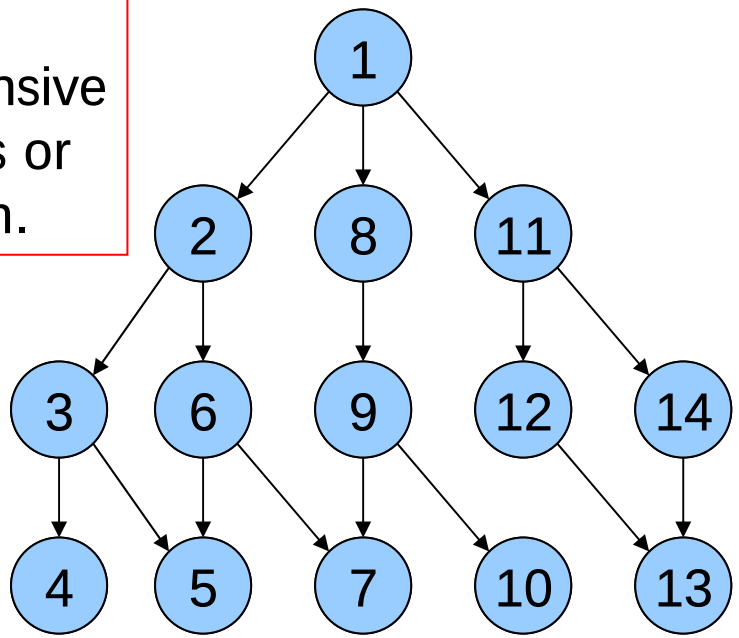
Search Orderings

Breadth-first-search (BFS)

Depth-first-search (DFS)

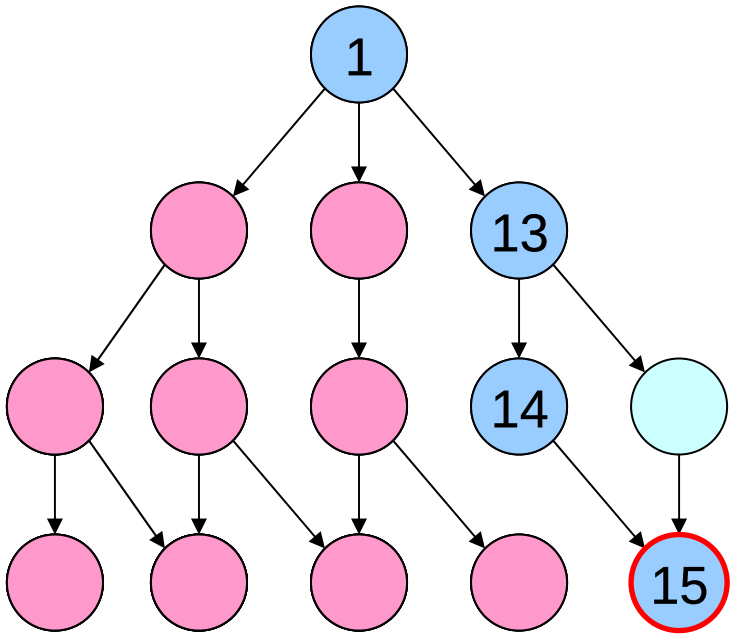
Gives shortest path but may be more expensive than heuristics or random search.





Clean-up deadlocks – DFS

Depth-first-search
(DFS)

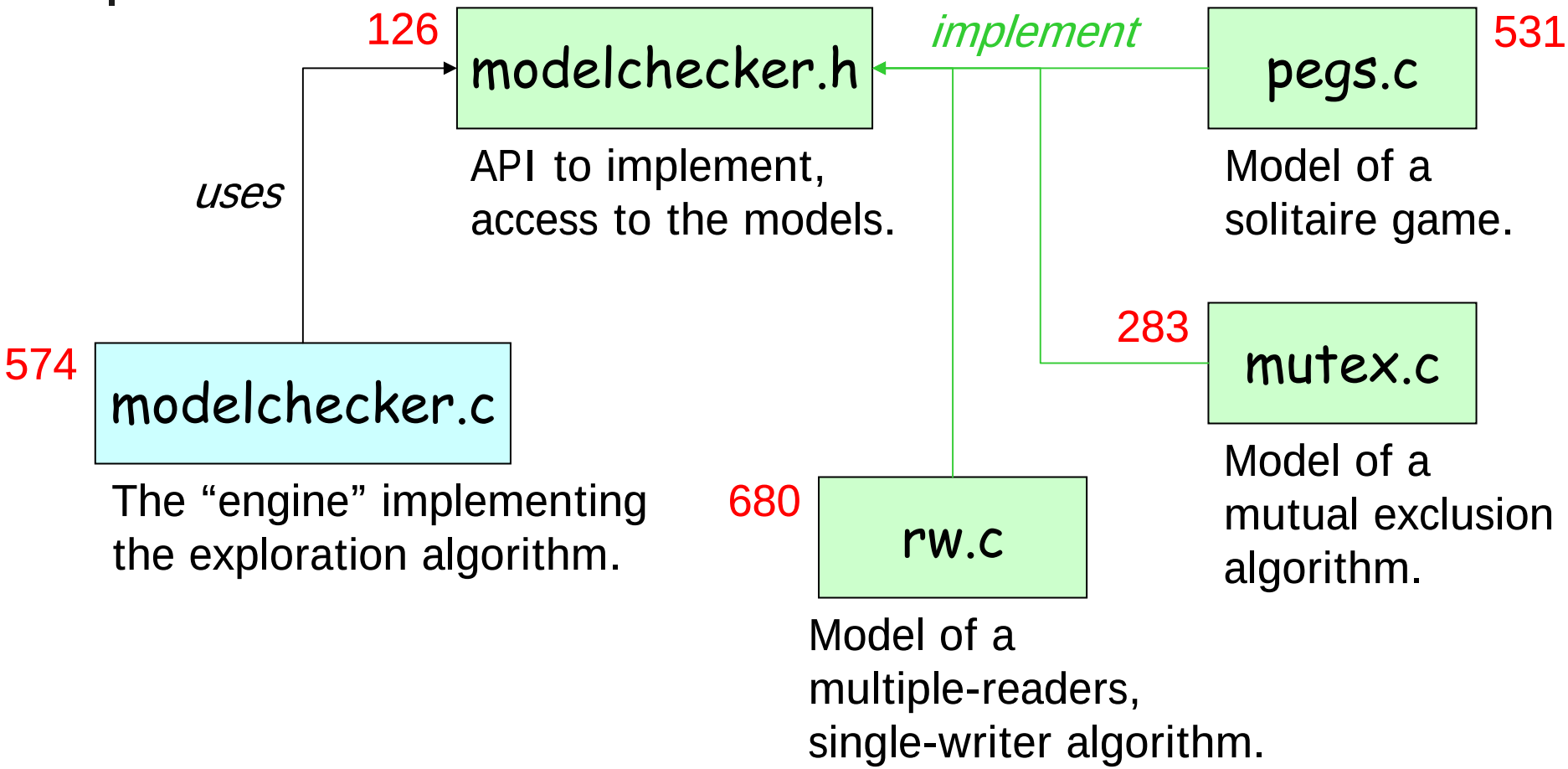


What can it do?

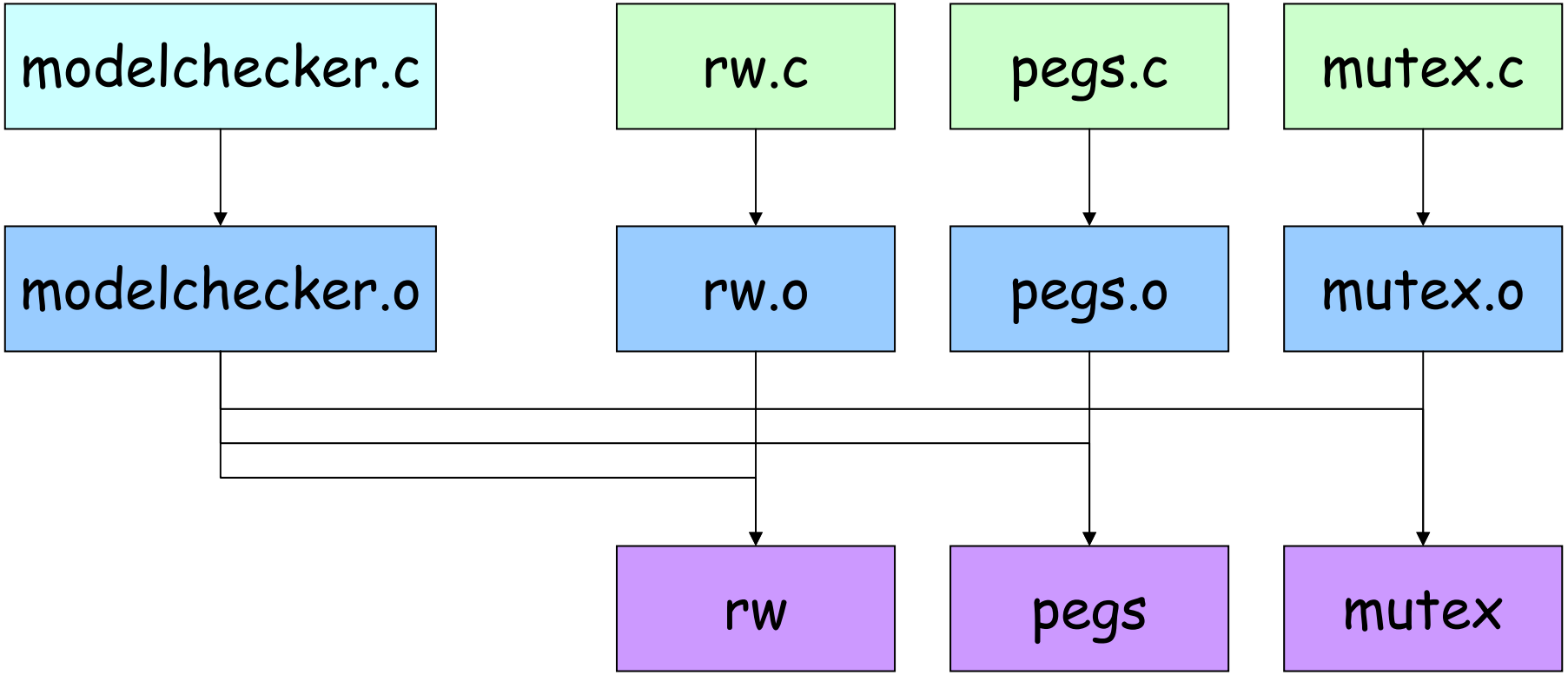
- BFS/DFS -f option.
- Clean-up deadlocks -g option (garbage).
- Check reachability properties (depends on models).
- Detect deadlocks -d option.
- Print system -s option.
- Print trace to found states.
- Can explore millions of states @ 300000+ states/sec.

Not a toy!

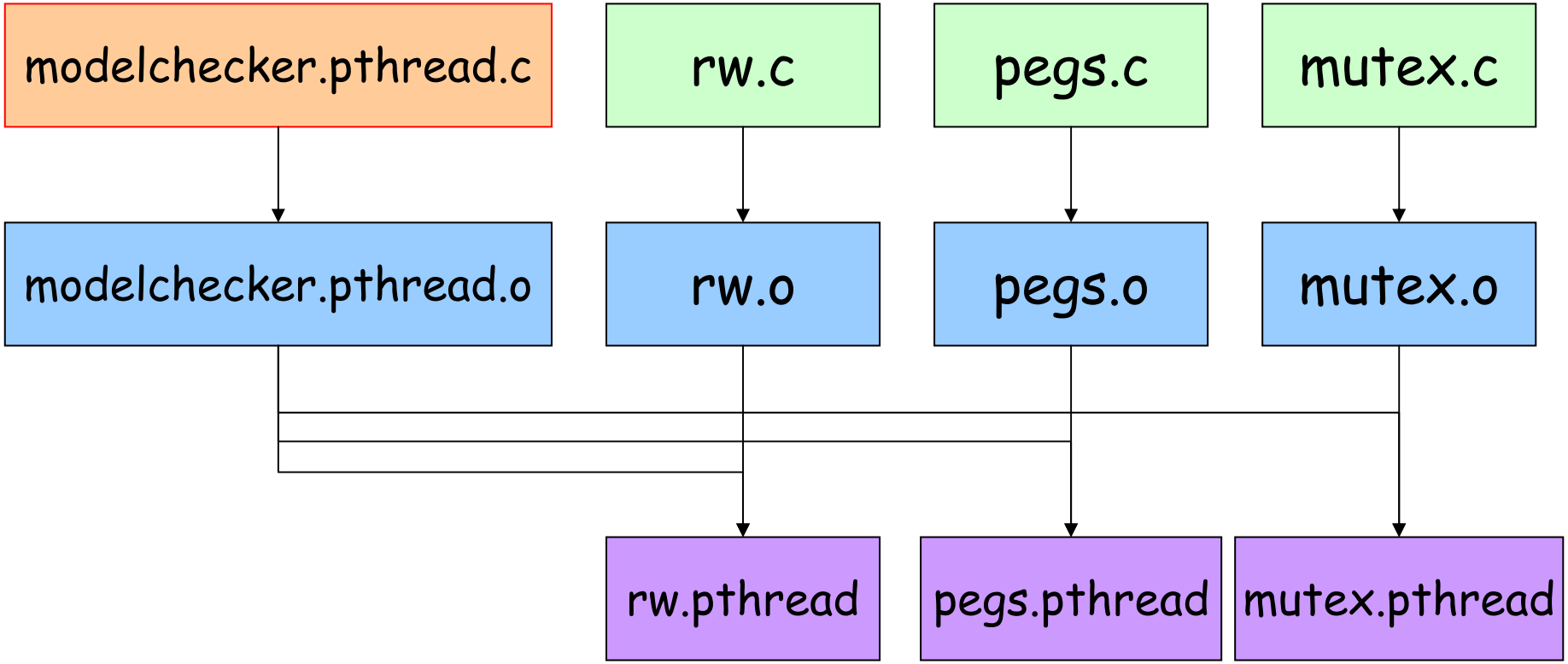
Design of the Model-Checker



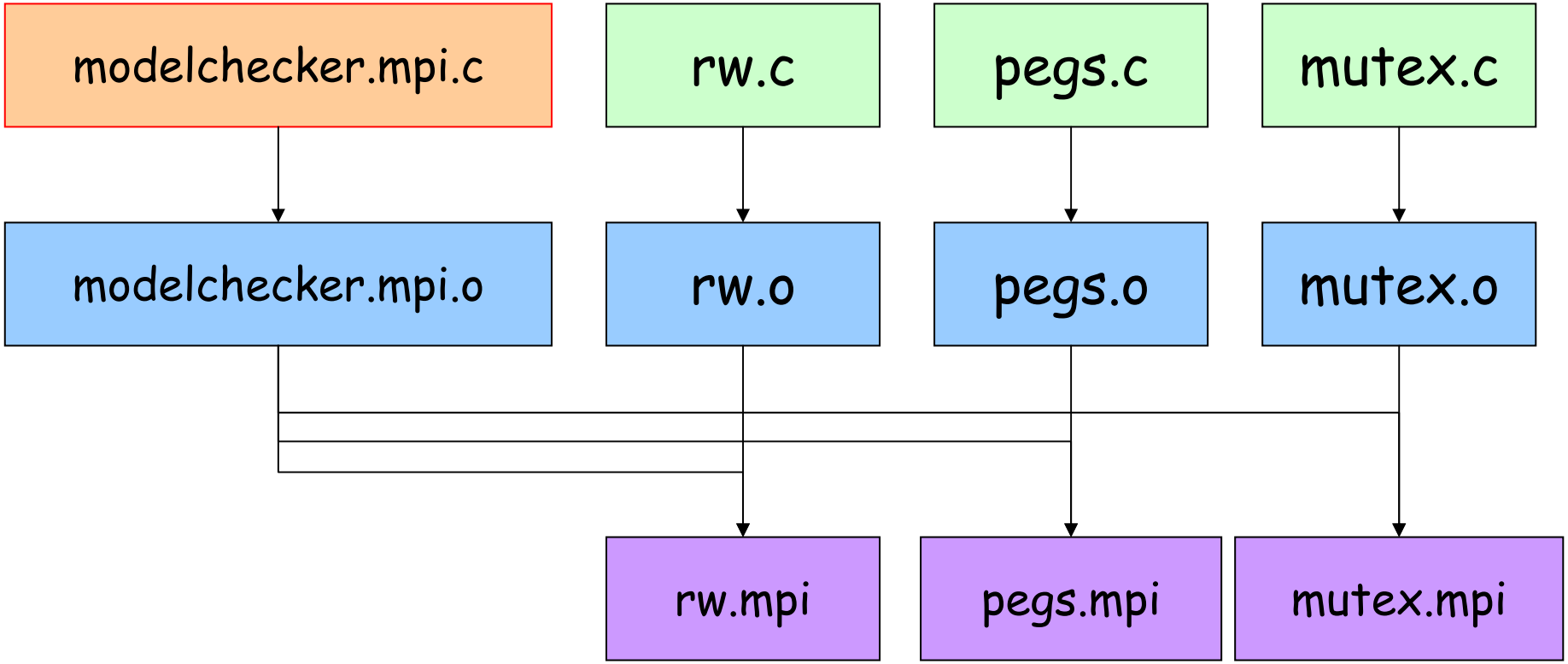
Compilation

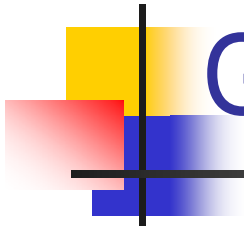


Compilation - pthread



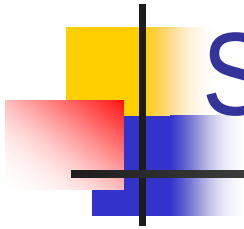
Compilation - mpi





Goal

- You are given a working model-checker with a Makefile.
  - Modify modelchecker.pthread.c to parallelize it using pthreads.
  - Modify modelchecker.mpi.c to parallelize it using mpi.
- But not now and not all at once.
- Linux: Install LAM (lam-runtime + lam4-dev).



Steps

- Now:
  - Discover the model-checker, make sure you can compile & run it.
  - Understand its structure, read the code.
- Later:
  - A simple version with pthread.
  - A better version with pthread.
  - A distributed version with MPI.