15-780: Grad AI
Lecture 14: Planning

Geoff Gordon (this lecture)
Tuomas Sandholm
TAs Erik Zawadzki, Abe Othman

Time

- Recall ***fluents***
- For KBs that evolve, add extra argument to each predicate saying when it was true
  - at(Robot, Wean5409)
  - at(Robot, Wean5409, t17)

Operators

- Given a representation like this, can define ***operators*** that change state
- E.g., given
    - *at(Robot, Wean5409, t17)*
    - and writing *t18* for *result(move(Robot, Wean5409, corridor), t17)*
- might be able to conclude
    - *at(Robot, corridor, t18)*
    - ¬ *at(Robot, Wean5409, t18)*

Goals

- Want our robot to, e.g., get sandwich
- Search for proof of *has(Geoff, Sandwich, t)*
- Try to analyze proof tree to find sequence of operators that make goal true

Russell & Norvig Ch 11–12

Complications

- This strategy yields lots of complications
  - frame or successor-state axioms (facts don’t change unless operator does it)
  - generalization of answer literal
  - unique names, reasoning about equality among situations…
- Result can be slow inference

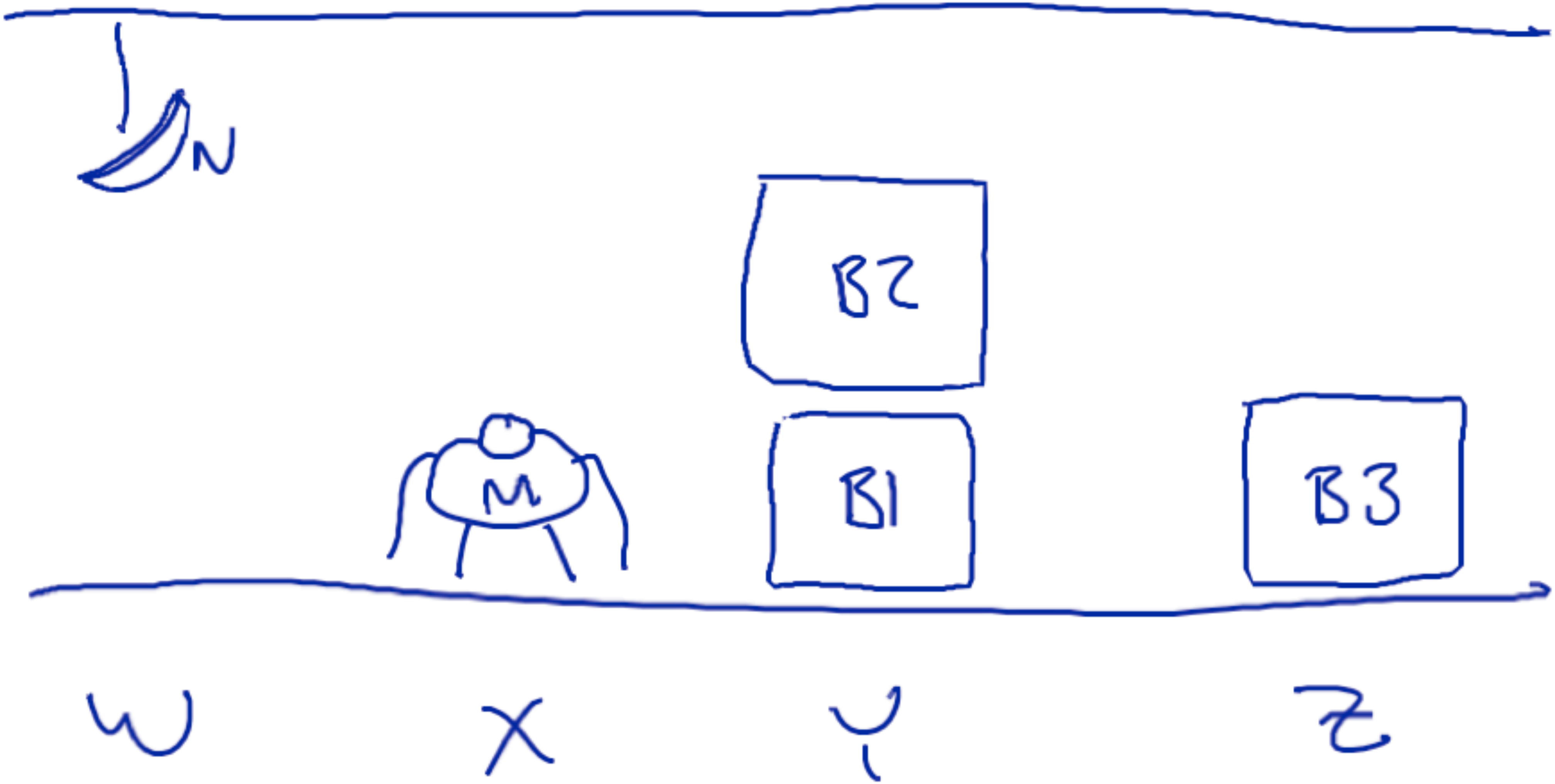
Planning

- Alternate solution: define a subset of FOL especially for planning
- E.g., STRIPS language *(STanford Research Institute Problem Solver)*

STRIPS

- State of world = { *true ground literals* }
  - no distinction between false, unknown
- goal = { *desired ground literals* }
  - done if goal $\subseteq$ state
- unique names, no functions, limited quantification, limited negation…
  - can get away w/o equality predicate

STRIPS example



Goal: full(M)

STRIPS example

- food(N)
- hungry(M)
- at(N, W)
- at(M, X)
- at(B1, Y)
- at(B2, Y)
- at(B3, Z)
- on(B2, B1)
- clear(B2)
- clear(B3)
- height(M, Low)
- height(N, High)

STRIPS operators

- Operator = { *preconditions* }, { *effects* }
- If preconditions are true at time t,
  - can apply operator at time *t*
  - effects will be true at time *t+1*
    - negated effect: delete from state
  - rest of state unaffected
- Basic STRIPS: one operator per step

Quantification in operators

- Preconditions of operator may contain variables (implicit ∀)
  - operator can apply if preconditions unify w/ state (using substitution X)
- Effects may use variables bound by precondition
  - state $t+1$ has e / X for each e in effects

Operator example

- Eat(target, p, l)
  - ***pre***: hungry(M), food(target), at(M, p), at(target, p), level(M, l), level(target, l)
  - ***eff***: ¬hungry(M), full(M), ¬at(target, p), ¬level(target, l)

Operator example

- Move(from, to)
  - ***pre***: at(M, from), level(M, Low)
  - ***eff***: at(M, to), ¬at(M, from)
- Push(object, from, to)
  - ***pre***: at(object, from), at(M, from), clear(object)
  - ***eff***: at(M, to), at(object, to), ¬at(object, from), ¬at(M, from)

Operator example

- Climb(object, p)
  - ***pre***: at(M, p), at(object, p), level(M, Low), clear(object)
  - ***eff***: level(M, High), ¬level(M, Low)
- ClimbDown()
  - ***pre***: level(M, High)
  - ***eff***: ¬level(M, High), level(M, Low)

Plan search

Plan search

- Given a planning problem (start state, operator descriptions, goal)
- Run standard search algorithms to find plan
- Decisions: search state representation, neighborhood def’n, search algorithm

Linear planner

- Simplest choice: ***linear planner***
  - Search state = sequence of operators
  - Neighbor: add op to end of sequence
- Bind variables as necessary
  - both op and binding are choice points
- Can search forward from start or backward from goal, or mix the two
- Example heuristic: number of open literals

Linear planner example

- Pick an operator, e.g.,
  - Move(from, to)
    - ***pre***: at(M, from), level(M, Low)
    - ***eff***: at(M, to), ¬at(M, from)
- Bind vars so preconditions match state
  - e.g., from: X, to:Y
    - ***pre***: at(M, X), level(M, Low)
    - ***eff***: at(M,Y), ¬at(M, X)

Apply operator

- food(N)
- hungry(M)
- at(N, W)
- at(M, X)
- at(B1, Y)
- at(B2, Y)
- at(B3, Z)
- on(B2, B1)
- clear(B2)
- clear(B3)
- level(M, Low)
- level(N, High)

Apply operator

- food(N)
- hungry(M)
- at(N, W)
- at(M, Y)
- at(B1, Y)
- at(B2, Y)
- at(B3, Z)
- on(B2, B1)
- clear(B2)
- clear(B3)
- level(M, Low)
- level(N, High)

Repeat…

- Plan is now [move(X,Y)]
- Pick another operator and binding
  - Climb(object, p), p:Y, object: B2
    - ***pre***: at(M,Y), at(B2,Y), level(M, Low), clear(B2)
    - ***eff***: level(M, High), ¬level(M, Low)

Apply operator

- food(N)
- hungry(M)
- at(N, W)
- at(M, Y)
- at(B1, Y)
- at(B2, Y)
- at(B3, Z)
- on(B2, B1)
- clear(B2)
- clear(B3)
- level(M, Low)
- level(N, High)

Apply operator

- food(N)
- hungry(M)
- at(N, W)
- at(M, Y)
- at(B1, Y)
- at(B2, Y)
- at(B3, Z)
- on(B2, B1)
- clear(B2)
- clear(B3)
- level(M, High)
- level(N, High)

And so forth

- A possible plan:
  - move(X, Y), move(Y, Z), push(B3, Z, Y), push(B3, Y, X), push(B3, X, W), climb(B3, W), eat(N, W, High)
- DFS will try moving XYX, climbing on boxes unnecessarily, etc.

Partial-order planner

- Linear planner can be wasteful: backtrack undoes most recent action, rather than one that might have caused failure
- ***Partial order planner*** tries to fix this
  - so does CBJ—can use together
- Avoids committing to details of plan until it has to (***principle of least commitment***)

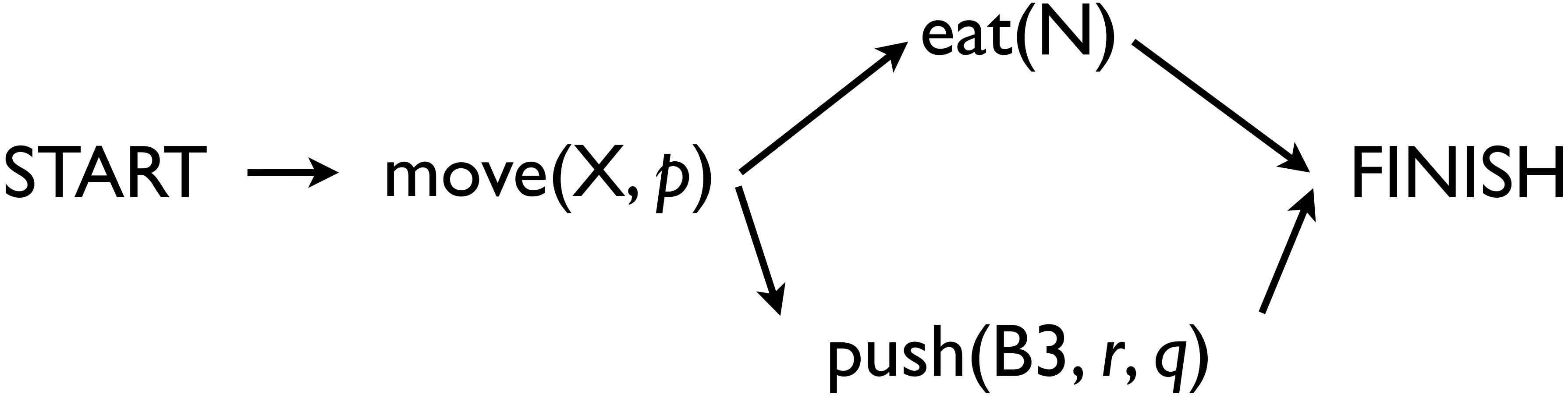
Partial-order planner

- Search state:
  - set of operators (partially bound)
  - ordering constraints
  - causal links (also called ***guards***)
  - open preconditions
- Neighborhood: plan refinement
  - resolve an open precondition by adding operator, constraint, and/or guard

State: set of operators

- Might include move(X, *p*) “I will move somewhere from X”, eat(*target*) “I will eat something”
- Also, extra operators START, FINISH
  - effects of START are initial state
  - preconditions of FINISH are goals

State: partial ordering



START

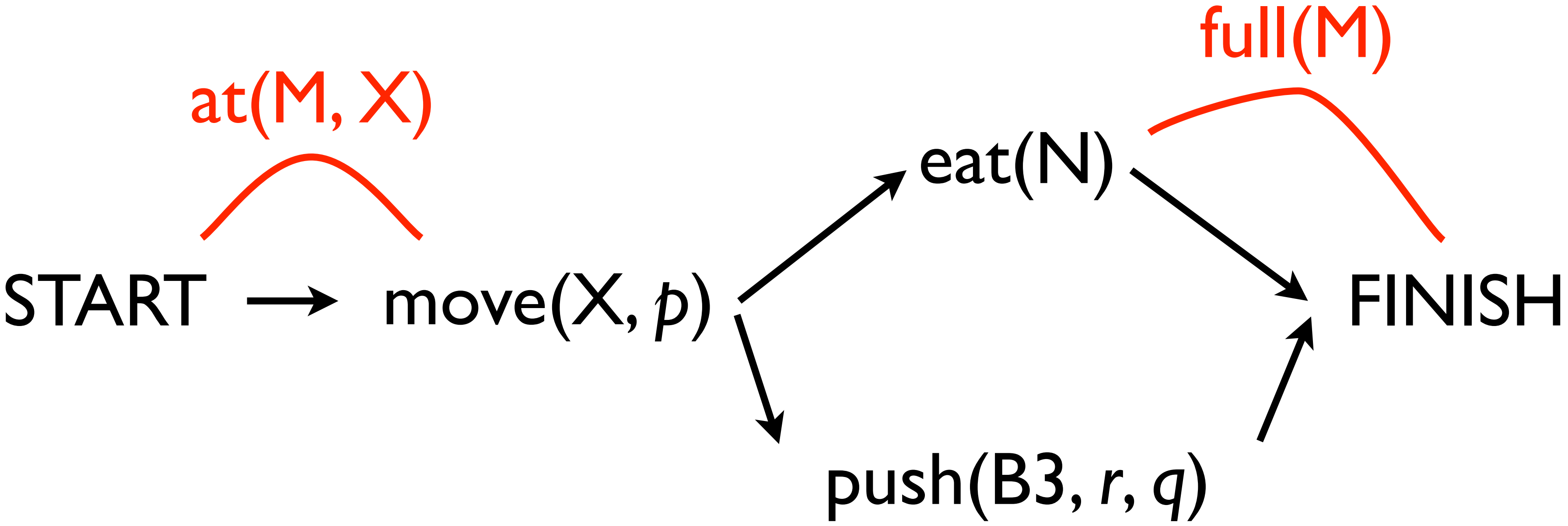
move(X, p)

eat(N)

push(B3, r, q)

FINISH

State: guards



at(M, X)

full(M)

START → move(X, *p*)

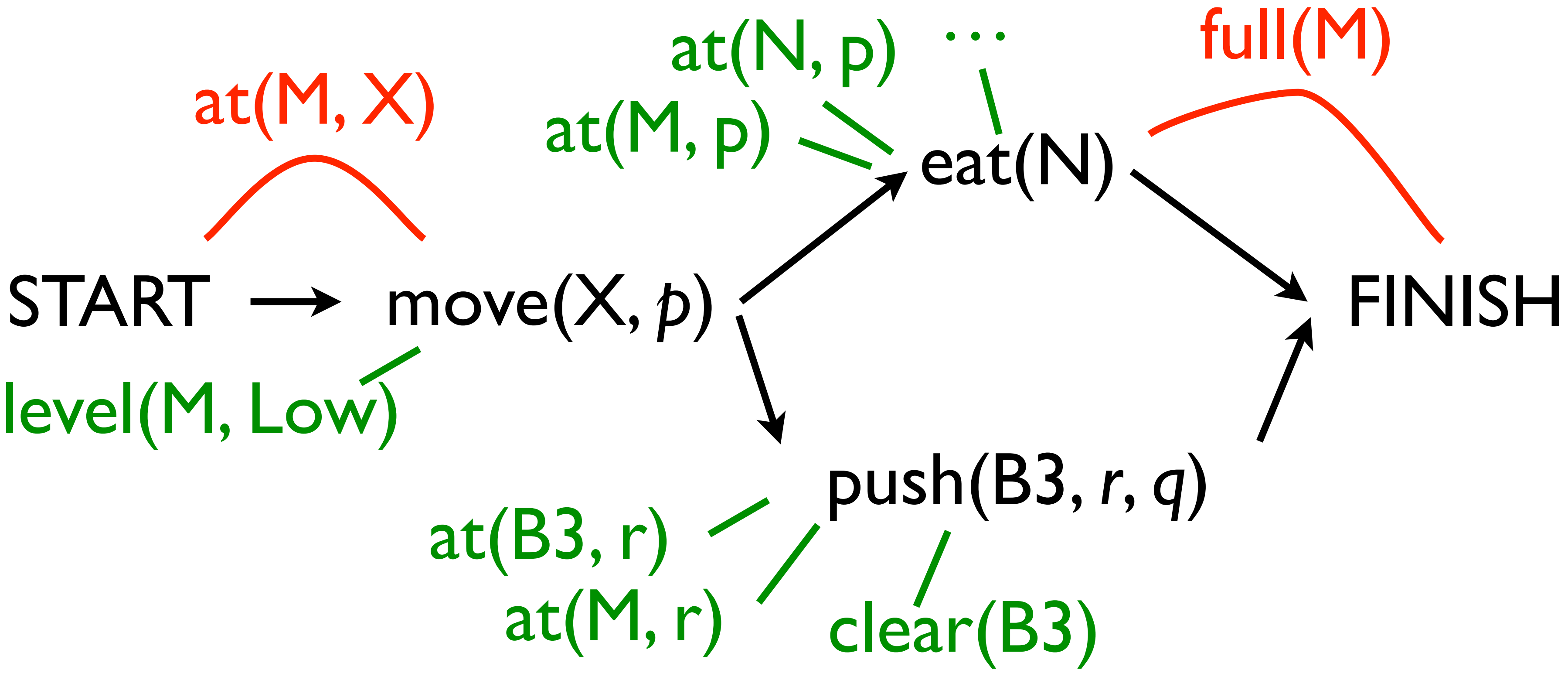
eat(N)

push(B3, *r*, *q*)

FINISH

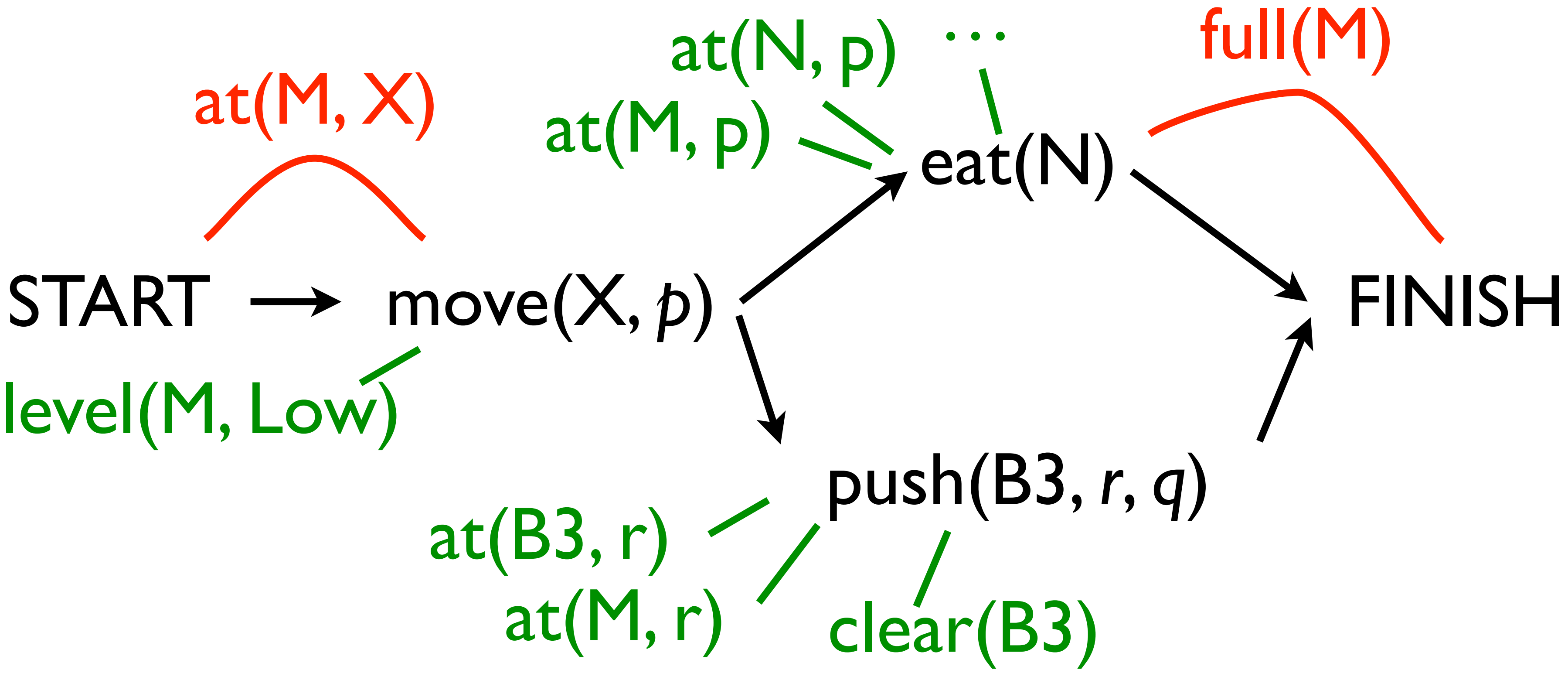
- Describe where preconditions are satisfied

State: open preconditions



- All unsatisfied preconditions of any action
- Unsatisfied = doesn't have a guard

Adding an ordering constraint



at(N, p)

. . .

full(M)

at(M, X)

at(M, p)

eat(N)

START

move(X, *p*)

FINISH

level(M, Low)

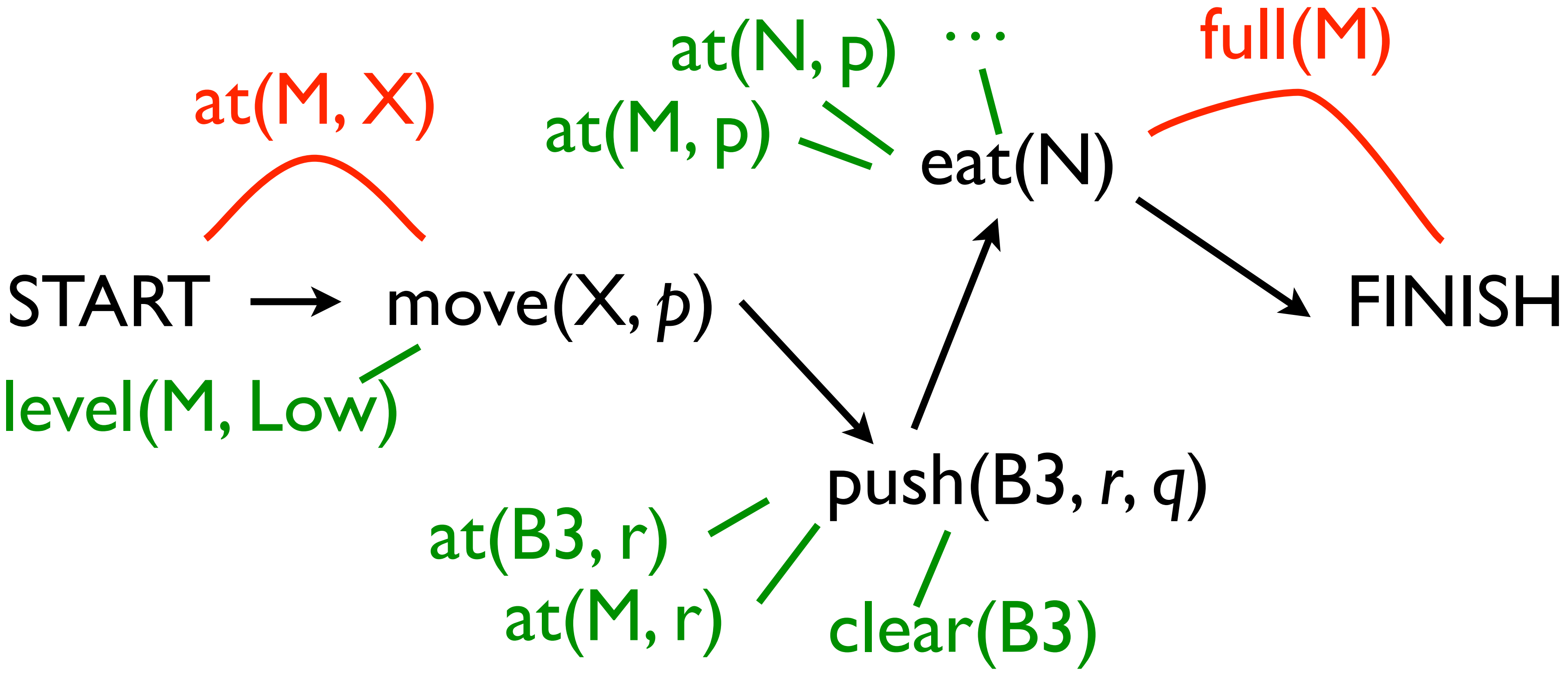
push(B3, *r*, *q*)

at(B3, r)

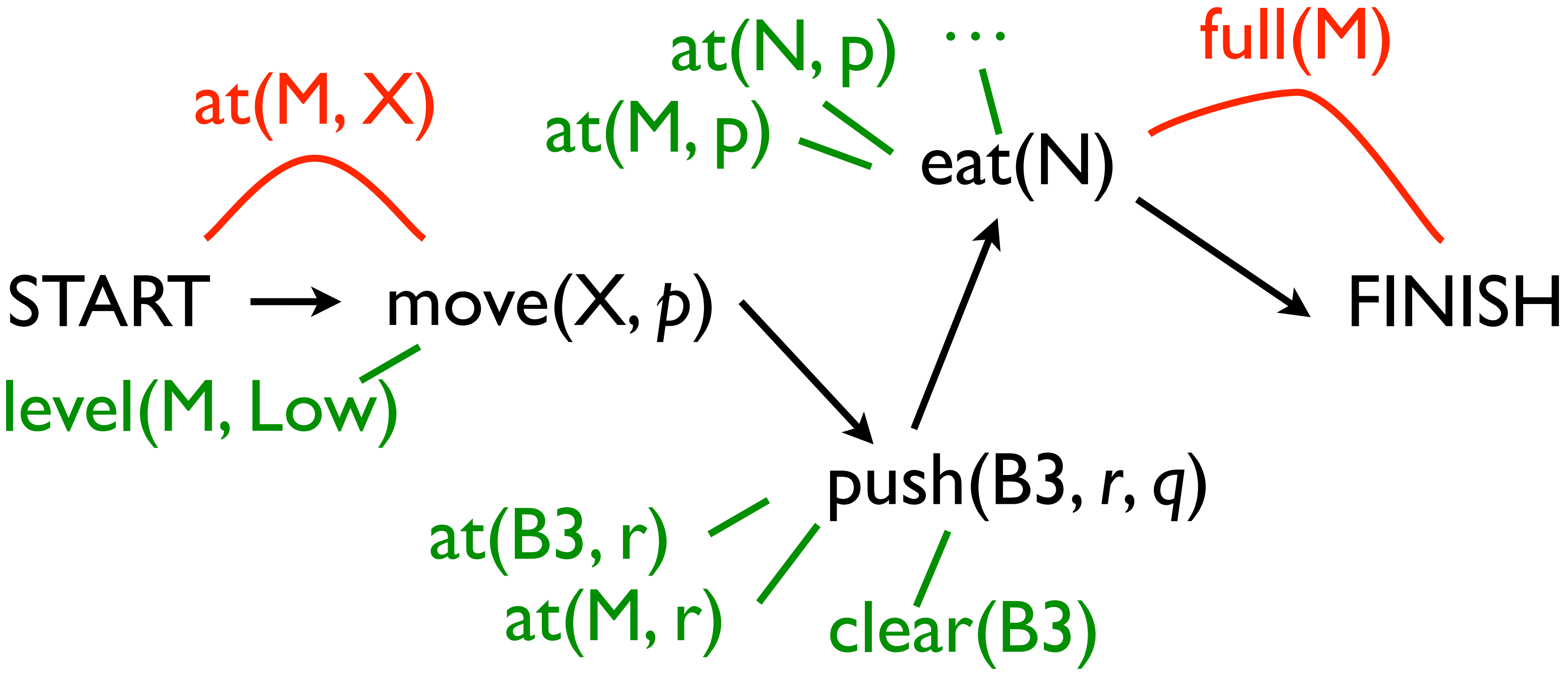
at(M, r)

clear(B3)

Adding an ordering constraint



Adding an ordering constraint



at(N, p) …

full(M)

at(M, X)

at(M, p)

eat(N)

START ⟶ move(X, *p*)

FINISH

level(M, Low)

push(B3, *r*, *q*)

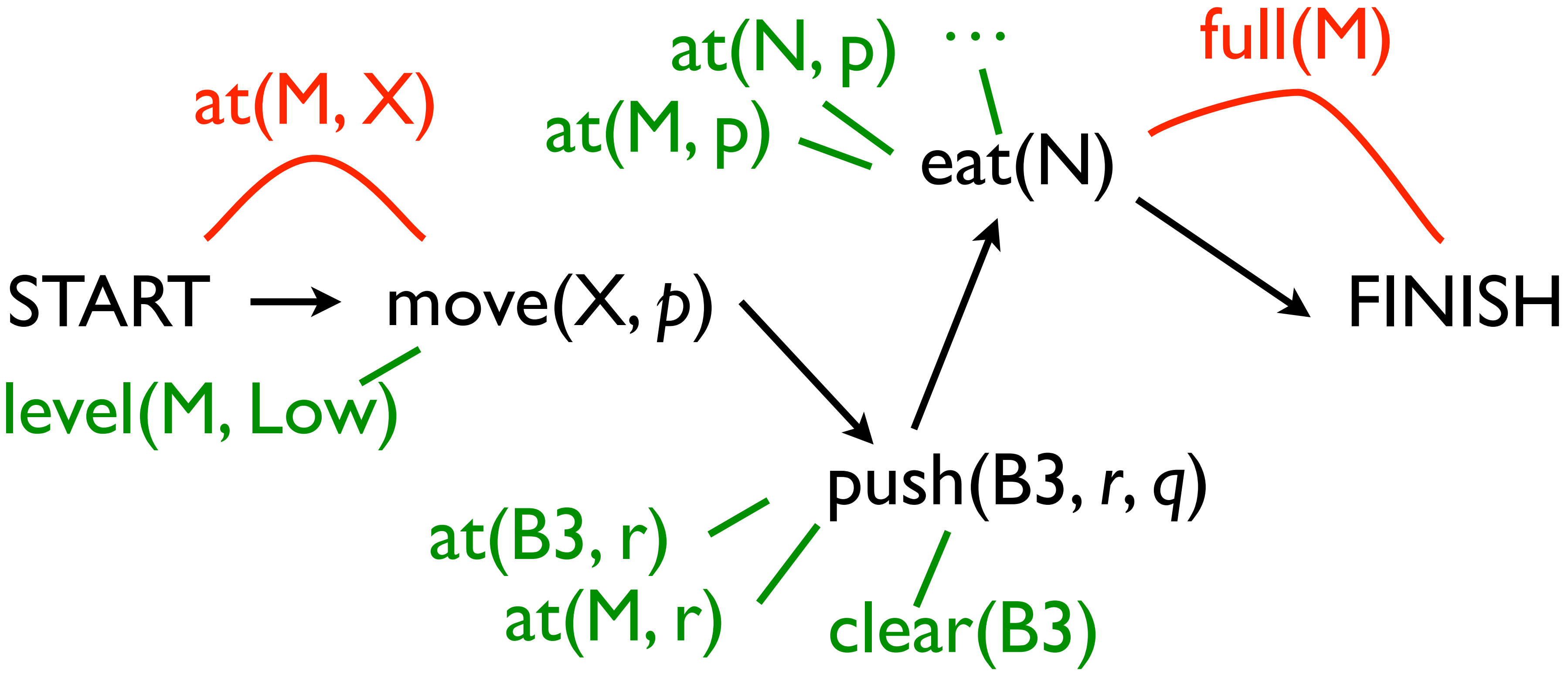
at(B3, r)

at(M, r)

clear(B3)

- Wouldn’t ever add ordering on its own—but may need to when adding operator or guard

Adding a guard



at(N, p) …

full(M)

at(M, X)

at(M, p)

eat(N)

START → move(X, *p*)

FINISH

level(M, Low)

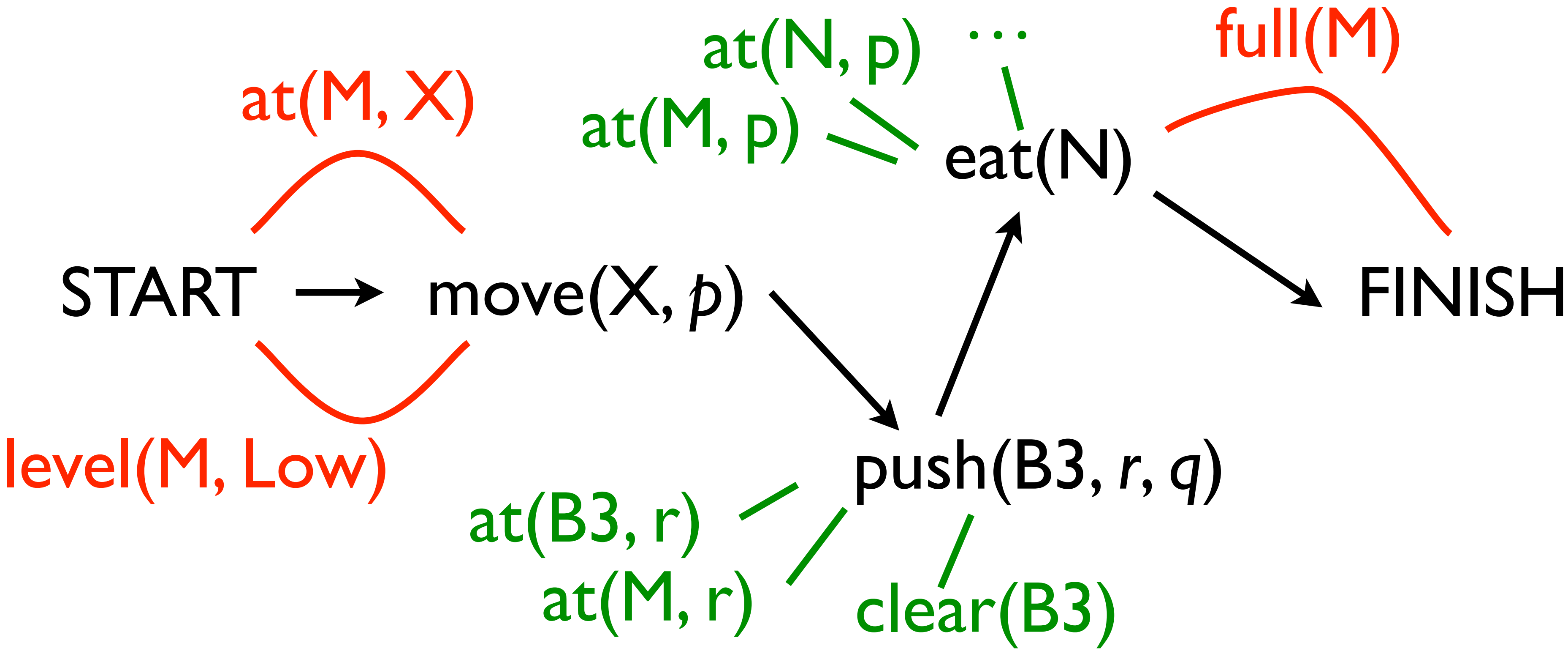
push(B3, *r*, *q*)

at(B3, r)

at(M, r)

clear(B3)

Adding a guard



at(N, p)

. . .

full(M)

at(M, X)

at(M, p)

eat(N)

START

move(X, *p*)

FINISH

level(M, Low)

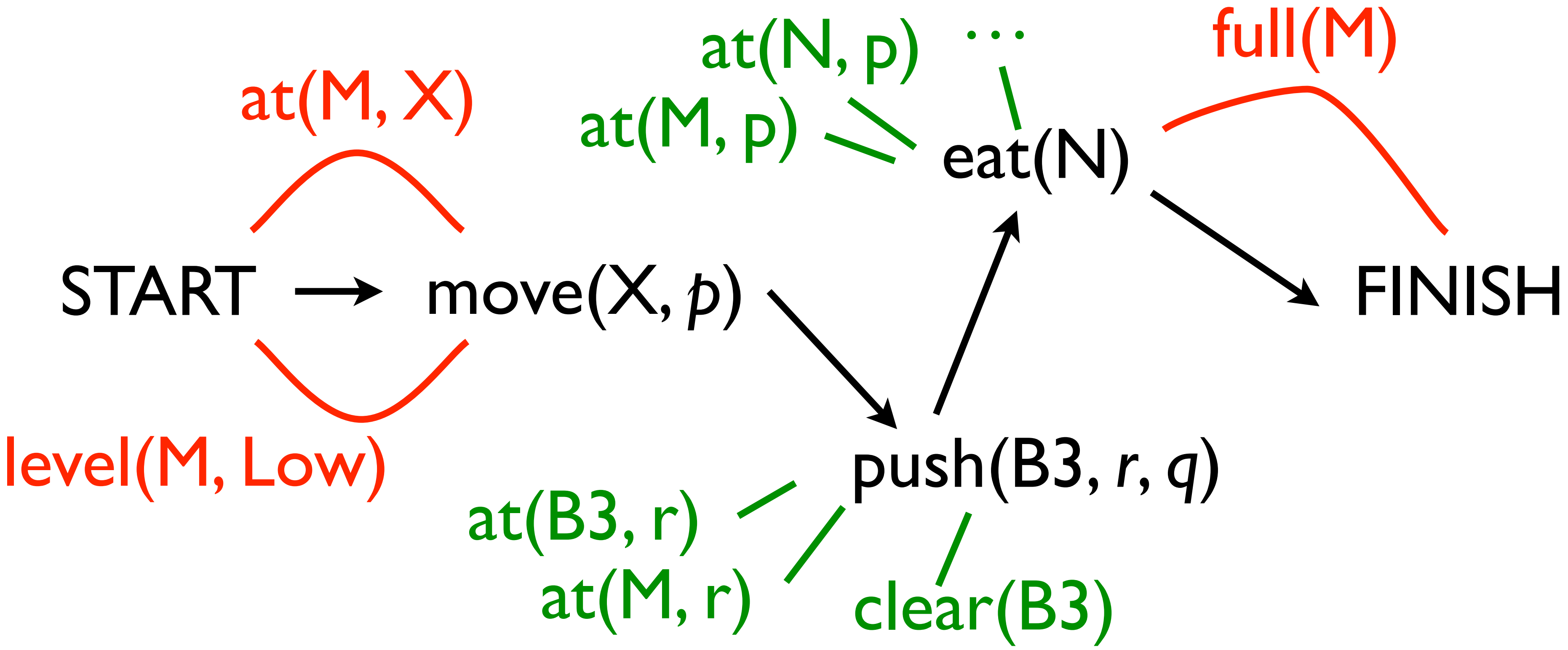
push(B3, *r*, *q*)

at(B3, r)

at(M, r)

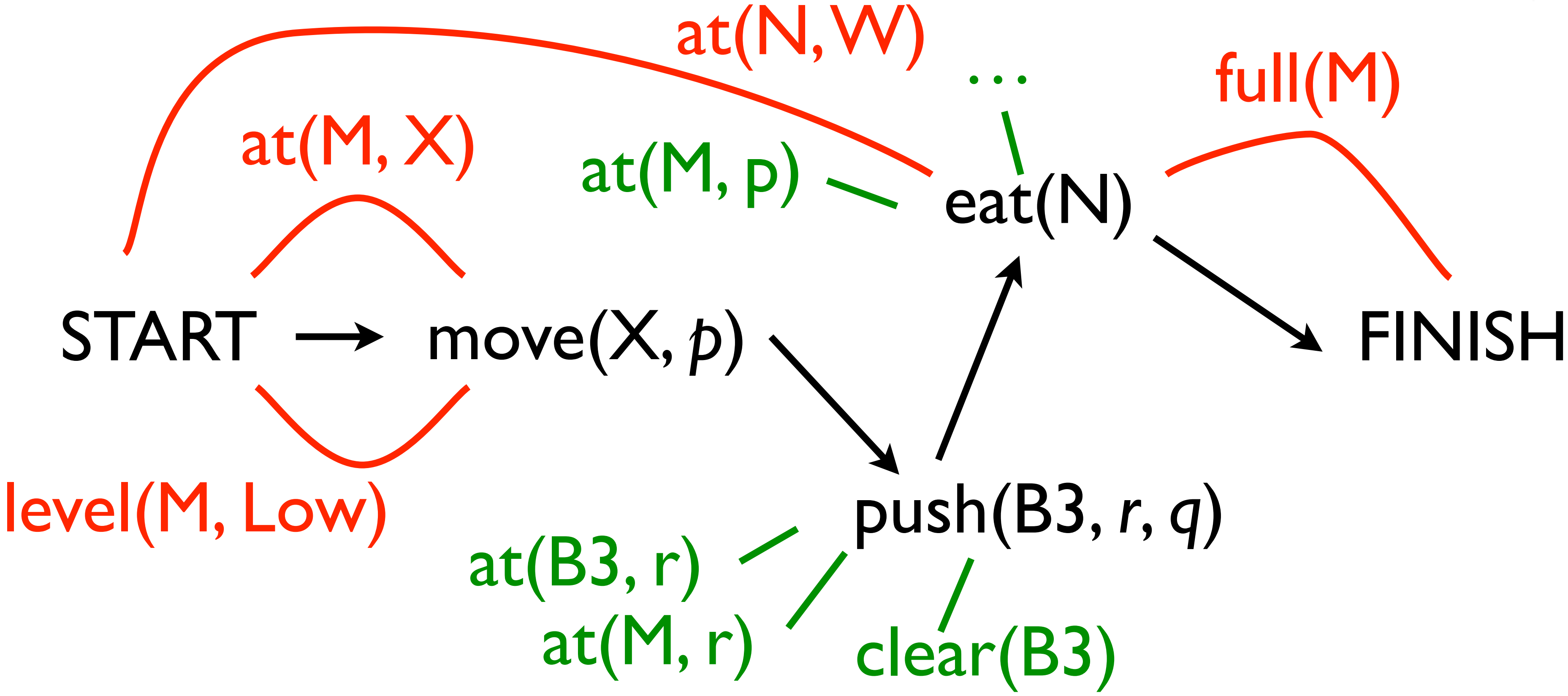
clear(B3)

Adding a guard



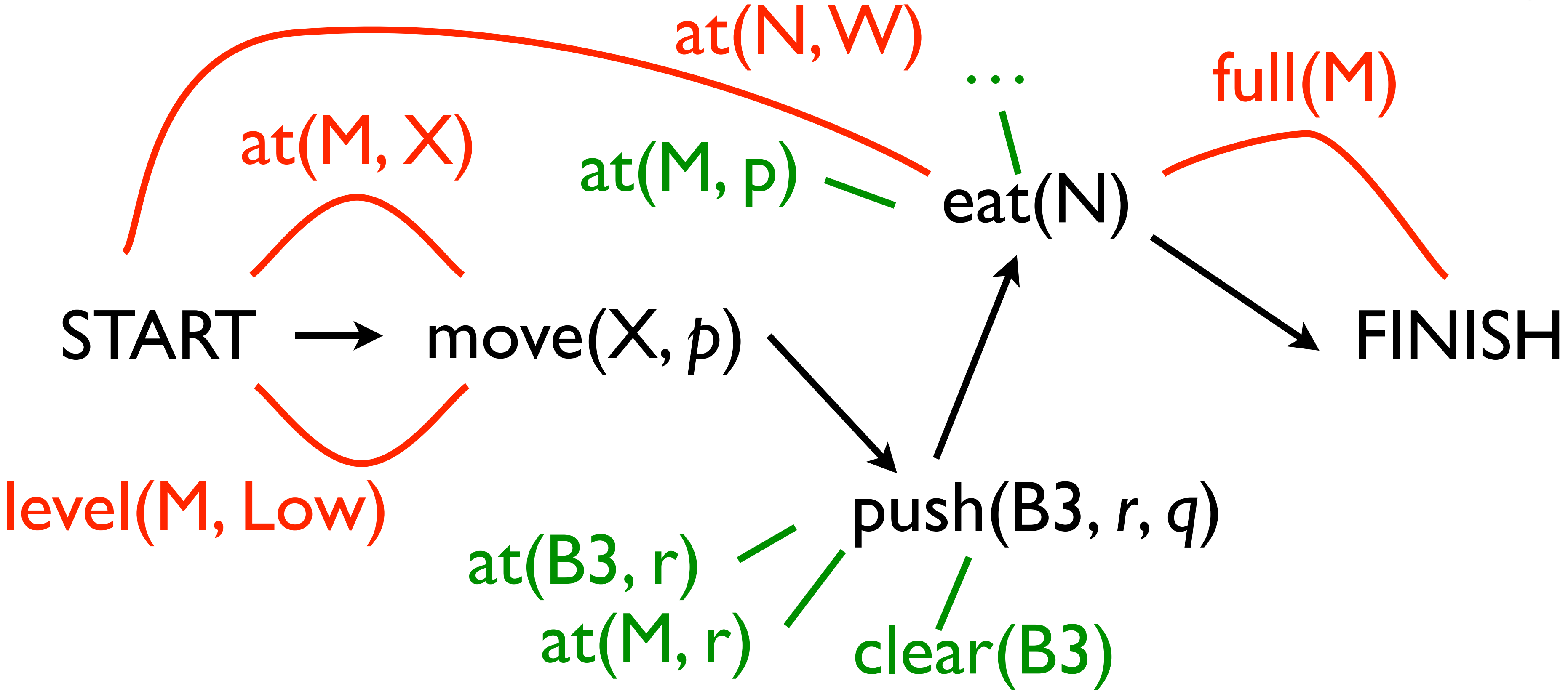
- Must go forward (may need to add ordering)
- Can't cross operator that affects condition

Adding a guard

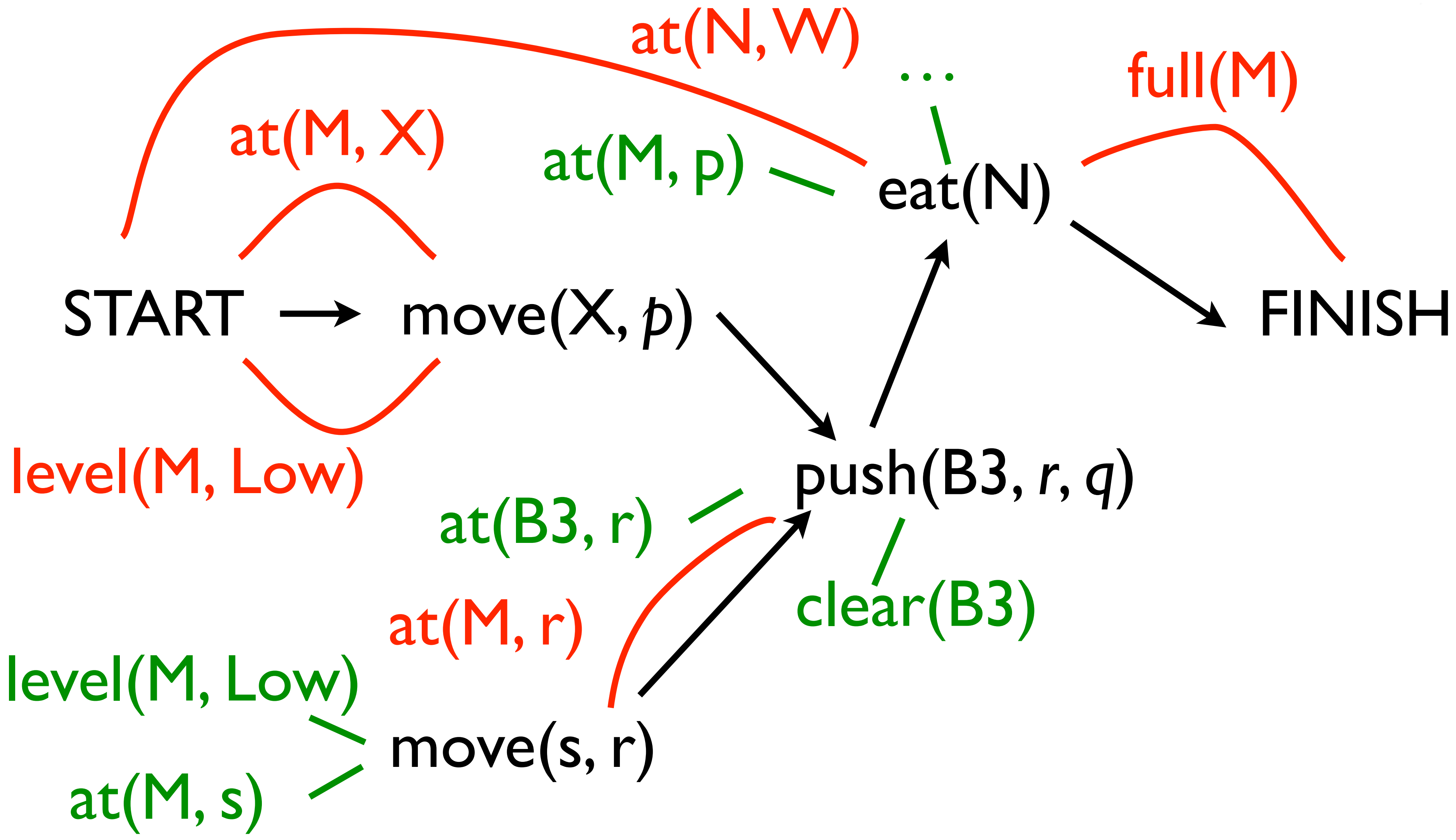


- Might involve binding a variable (may be more than one way to do so)

Adding an operator



Adding an operator



at(N, W)

. . .

full(M)

at(M, X)

at(M, p)

eat(N)

START

move(X, *p*)

FINISH

level(M, Low)

push(B3, *r*, *q*)

at(B3, r)

at(M, r)

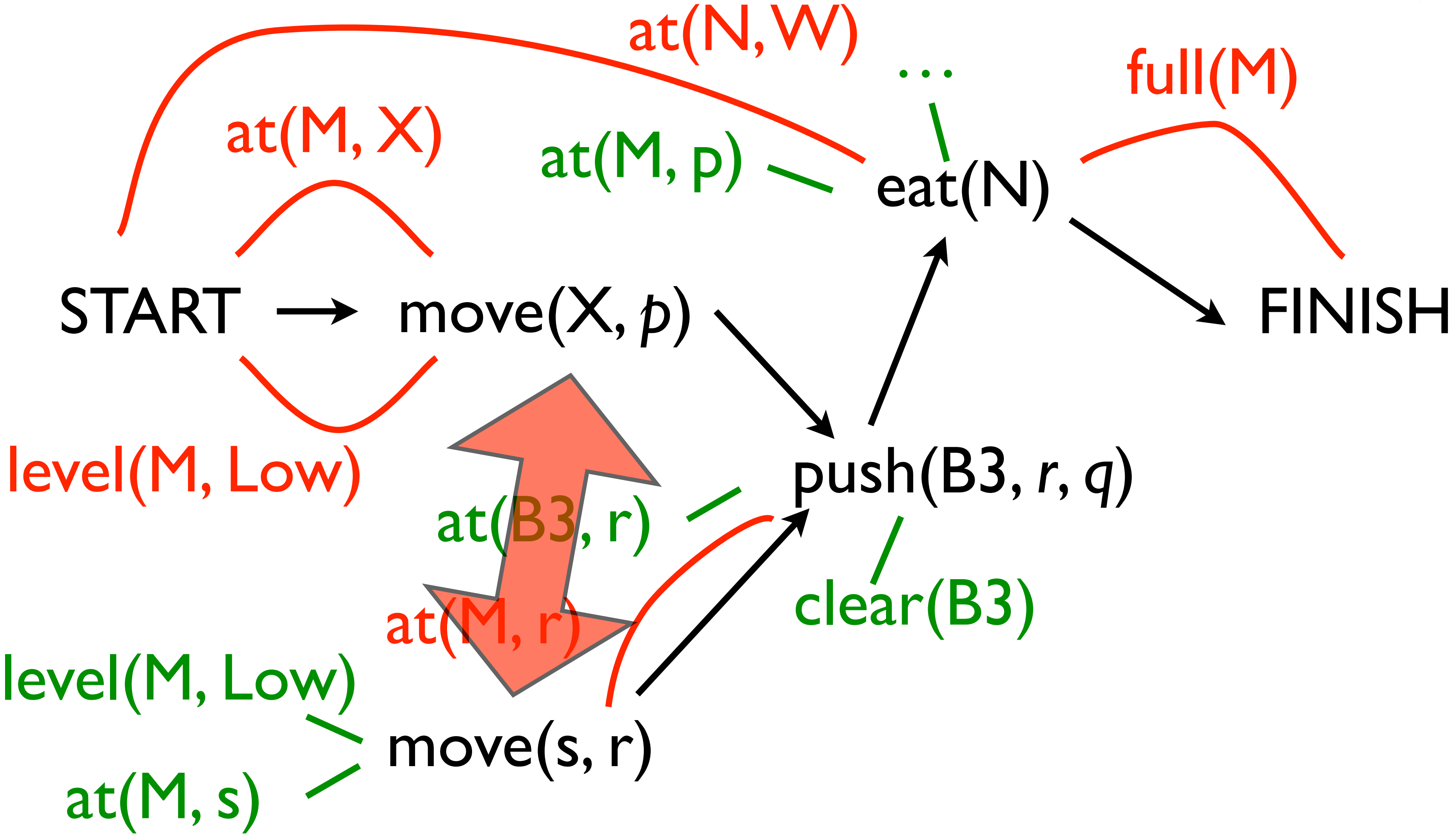
clear(B3)

level(M, Low)

move(s, r)

at(M, s)

Adding an operator



at(N, W)

. . .

full(M)

at(M, X)

at(M, p)

eat(N)

START

move(X, *p*)

FINISH

level(M, Low)

push(B3, *r*, *q*)

at(B3, r)

clear(B3)

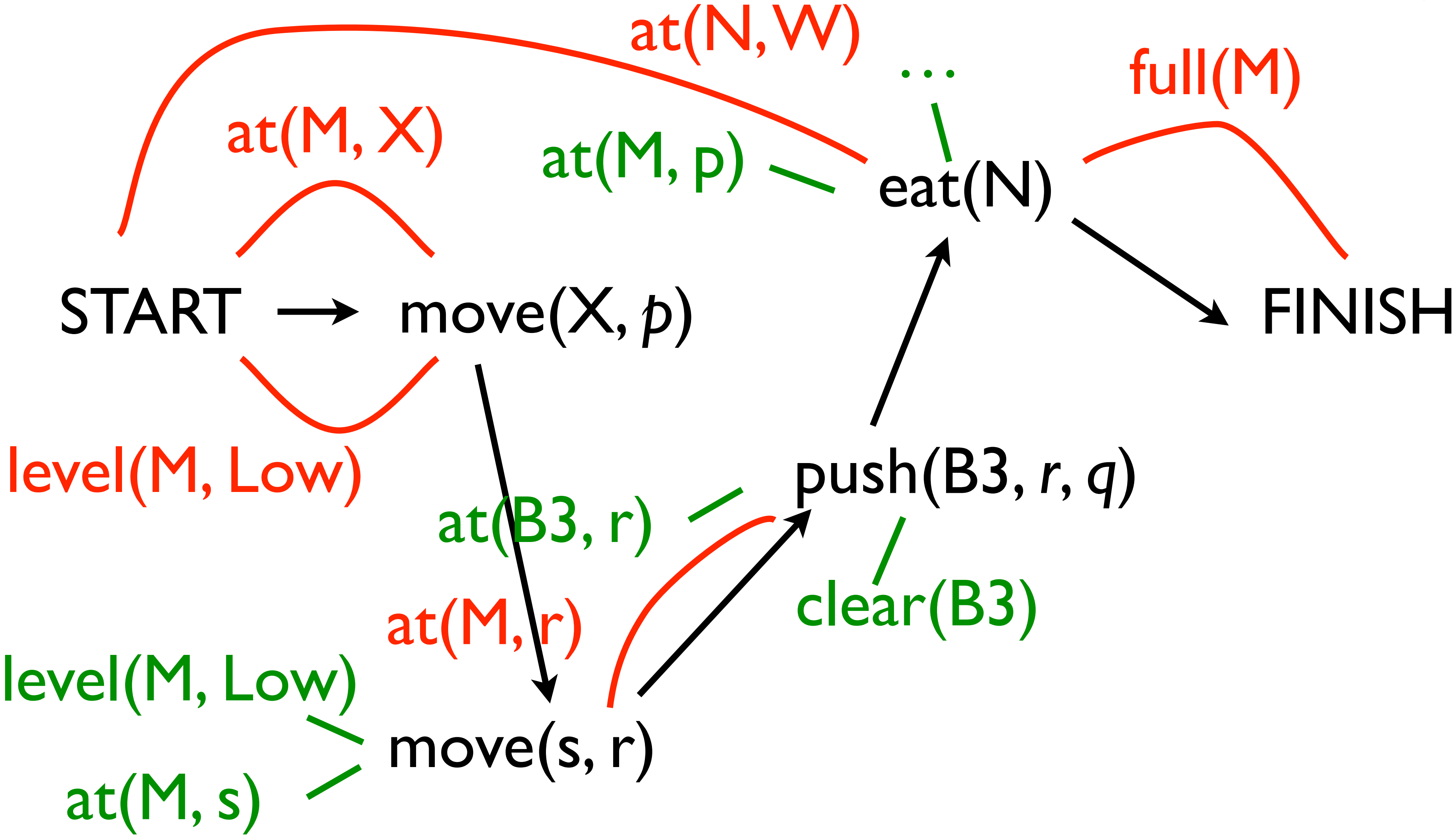
at(M, r)

level(M, Low)

move(s, r)

at(M, s)

Resolving conflict



at(N, W)

. . .

full(M)

at(M, X)

at(M, p)

eat(N)

START

move(X, *p*)

FINISH

level(M, Low)

push(B3, *r*, *q*)

at(B3, r)

clear(B3)

at(M, r)

level(M, Low)

move(s, r)

at(M, s)

Recap of neighborhood

- Pick an open precondition
- Pick an operator and binding that can satisfy it
  - may need to add a new op
  - or can use existing op
- Add guard
- Resolve conflicts by adding constraints, bindings

Consistency & completeness

- Plan ***consistent***: no cycles in ordering, preconditions guaranteed true throughout guard intervals
- Plan ***complete***: no open preconditions
- Search maintains consistency, terminates when complete

Execution

- A consistent, complete plan can be executed by **linearizing** it:
  - execute actions in any order that matches constraints
  - fill in unbound vars in any consistent way