Automated Planning

Plan-Space Planning / Partial Order Causal Link Planning

Jonas Kvarnström
Automated Planning Group
Department of Computer and Information Science
Linköping University

Partly adapted from slides by Dana Nau

jonas.kvarnstrom@liu.se – 2018

Motivations

Motivating Problem

jonkv@ida

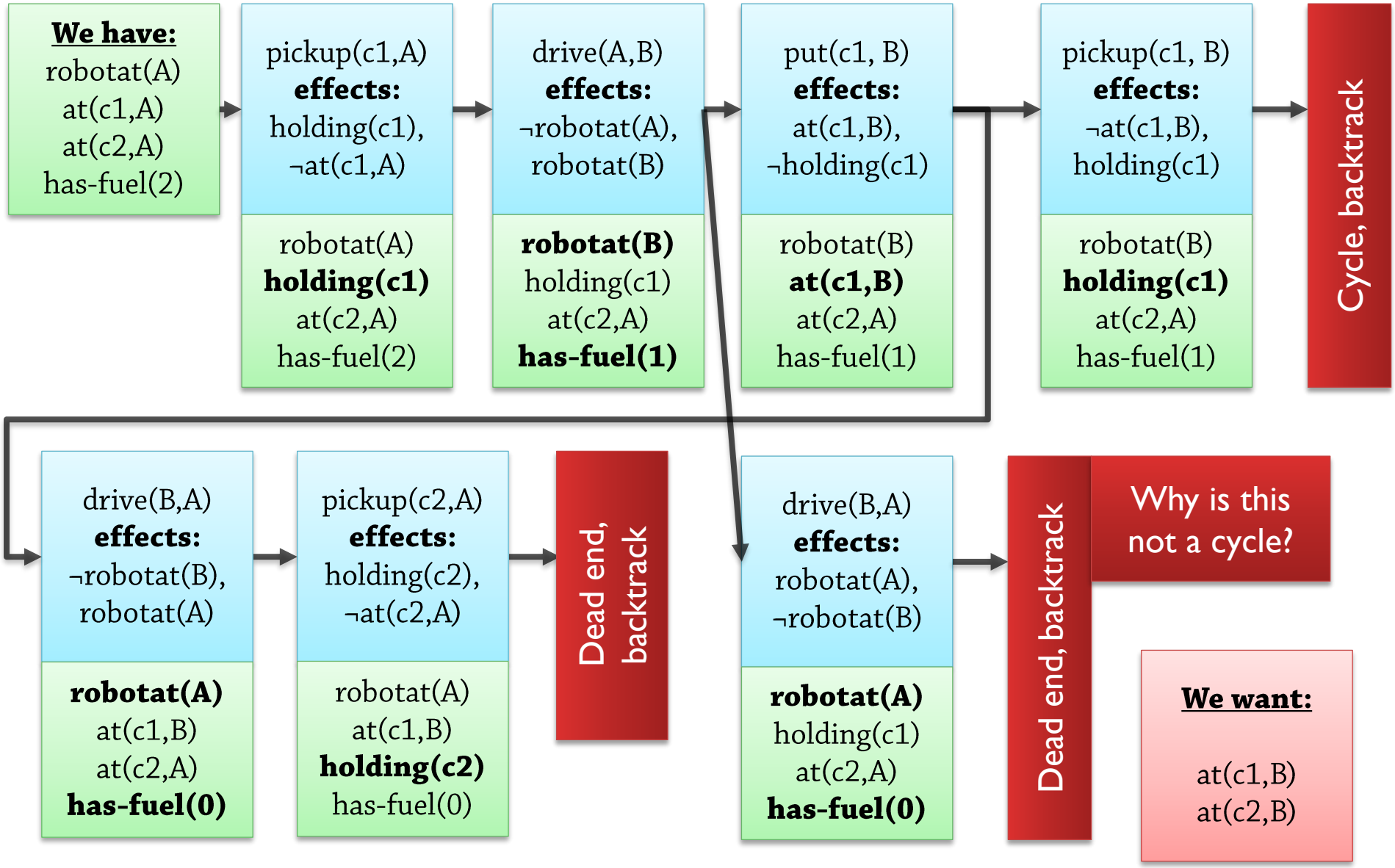
- Simple planning problem:
  - Two **crates**
    - At A
    - Should be at B

  - One **robot**
    - Can **carry** up to two crates
    - Can **move** between locations, which requires one unit of **fuel**
    - Has only two units of fuel

Let's see what a forward-chaining planner *might* do (depending on heuristics)…

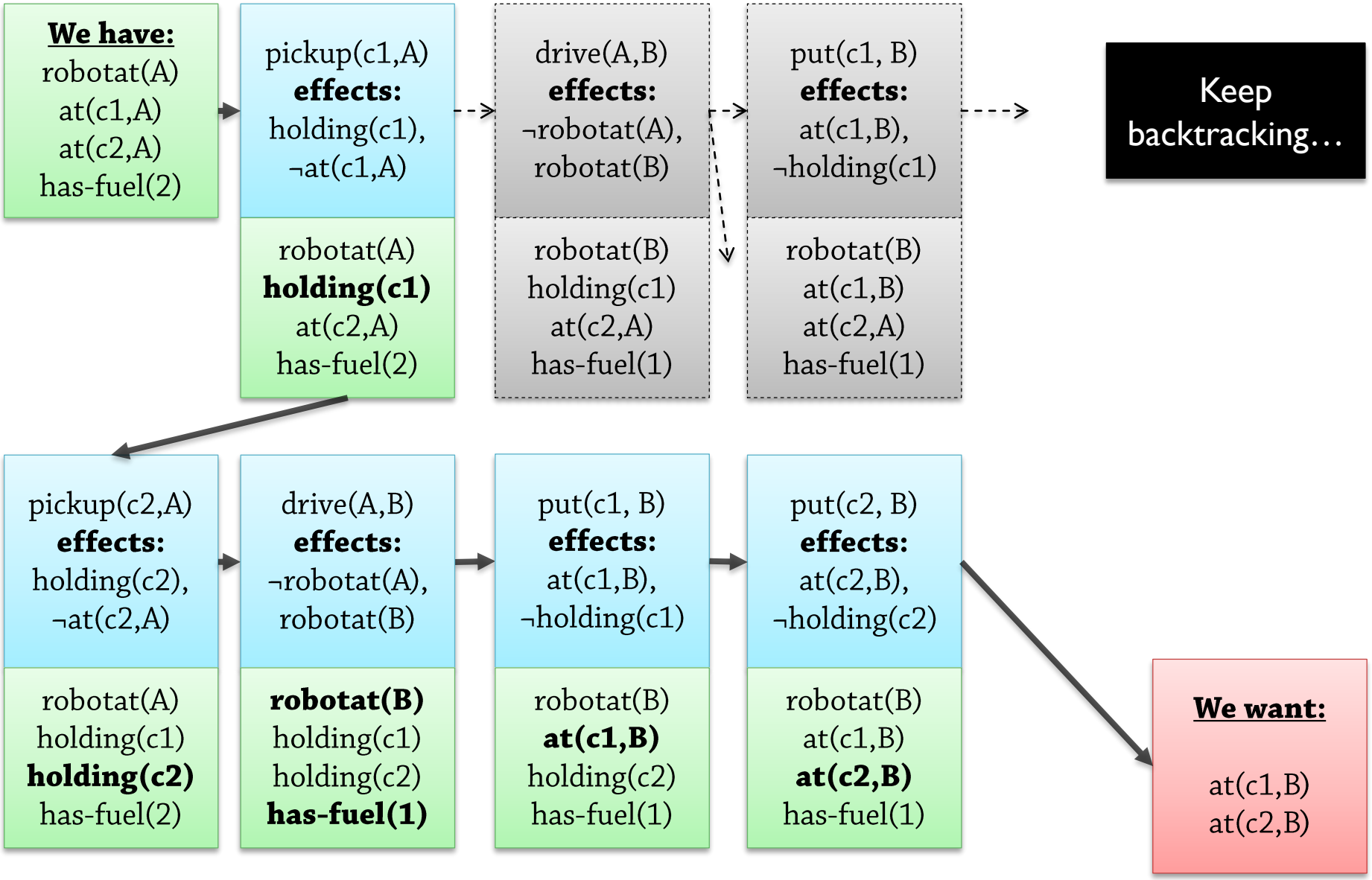
Motivating Problem 2: Forward Search

jonkv@ida



Motivating Problem 3

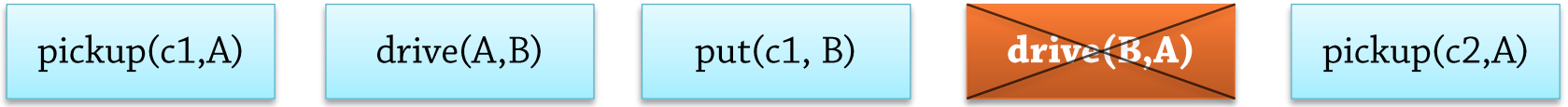
jonkv@ida



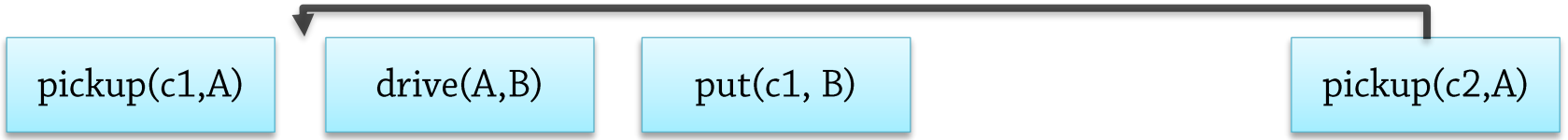
Motivating Problem 4

jonkv@ida

- Observations:
  - **Most** actions we added before backtracking were **useful** and **necessary**!



  - At first, we added them in the **wrong order**



  - **Forward** and **backward** planning **commits** immediately to action order
    - Puts each action in its **final place** in the plan
  - State space **heuristics** must tell us:
    - Which actions are **useful**
    - ***When*** *to add them to the plan*

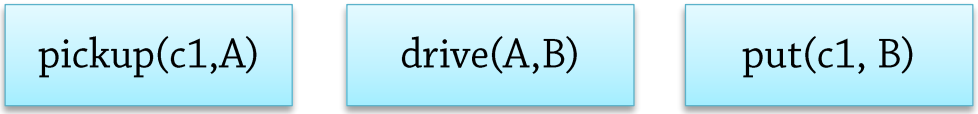
What if we could rearrange actions?

Planning Outside the State Space

First Step

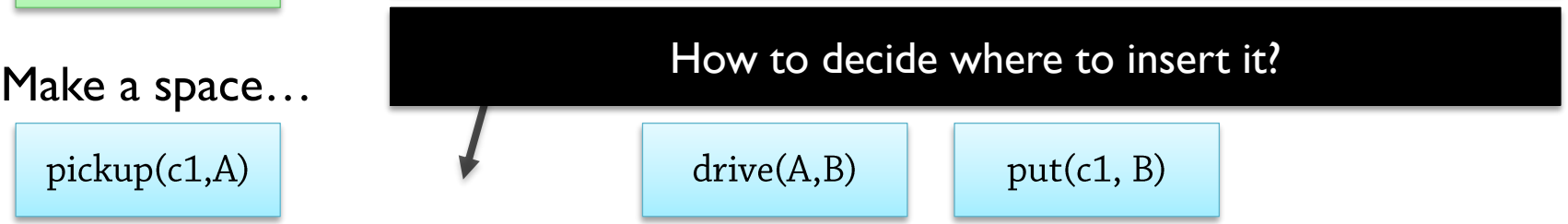
jonkv@ida

- **Sequences with arbitrary insertion**: Useful?
  - Add actions in sequence, as in state space planning…

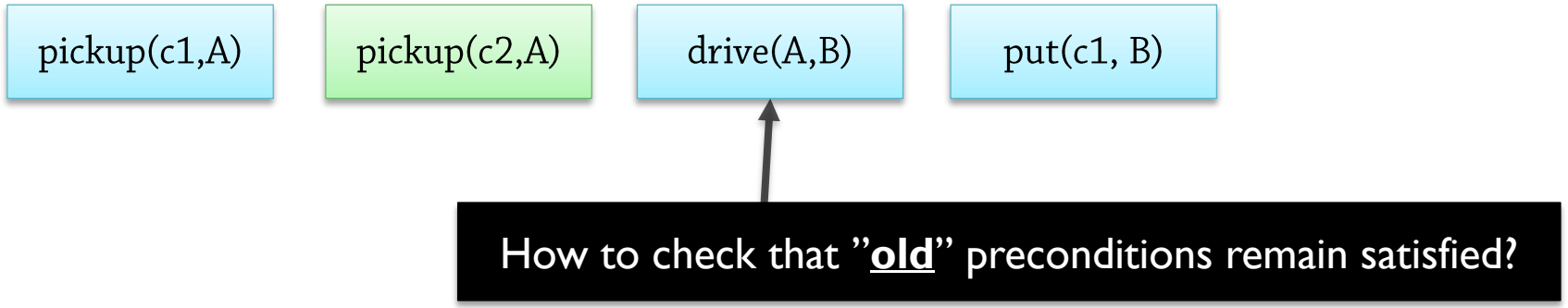


  - Realize you need another one…

  - Make a space…



  - …and place the action there

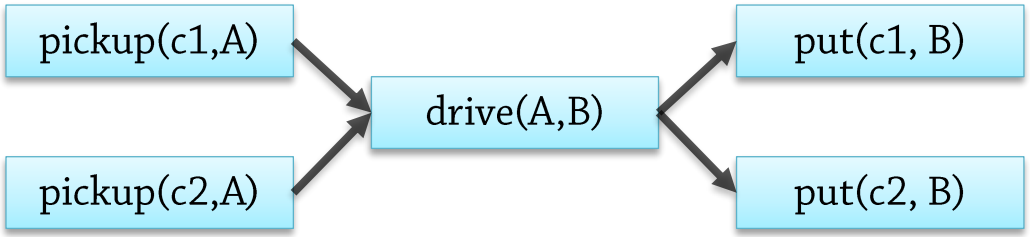


Second Step

jonkv@ida

- If we must deal with this complexity:
  - We can ”get more for the same price”

- Let’s skip sequences completely – a plan could be *partially ordered*:



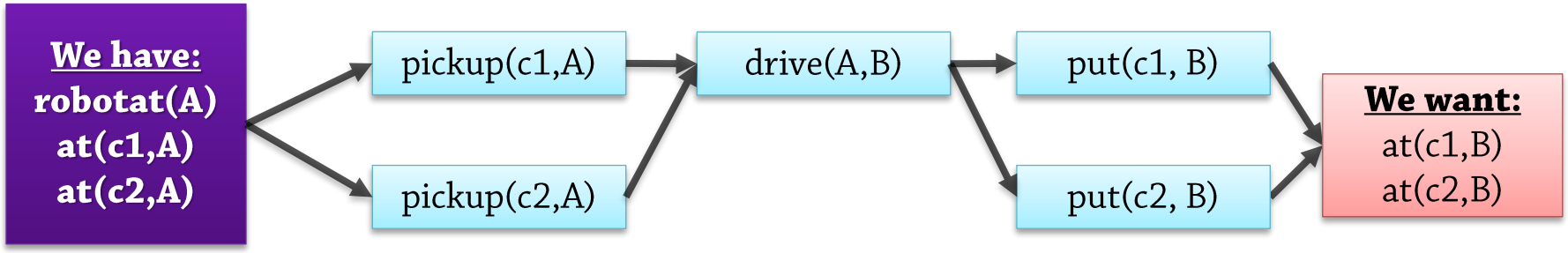
  - A set of **actions** $A = \{ a_1, a_2, a_3, \dots \}$
  - A set of **precedence constraints** $\{ a_1 < a_2, a_1 < a_3, \dots \}$
    - a_1 must finish before a_2 starts, …
    - Here: **solid arrows**

How do we *generate* these plans?

POCL 1: Introduction

jonkv@ida

- **Partial Order Causal Link** (POCL) planning
  - Use a partial order, as described
    - Not when executing the plan
    - Only to **delay commitment** to ordering
  - As in backward search:
    - Add **useful** actions to achieve necessary conditions
    - Keep track of what **remains** to be achieved
    - But: Insert actions "**at any point**" in a plan

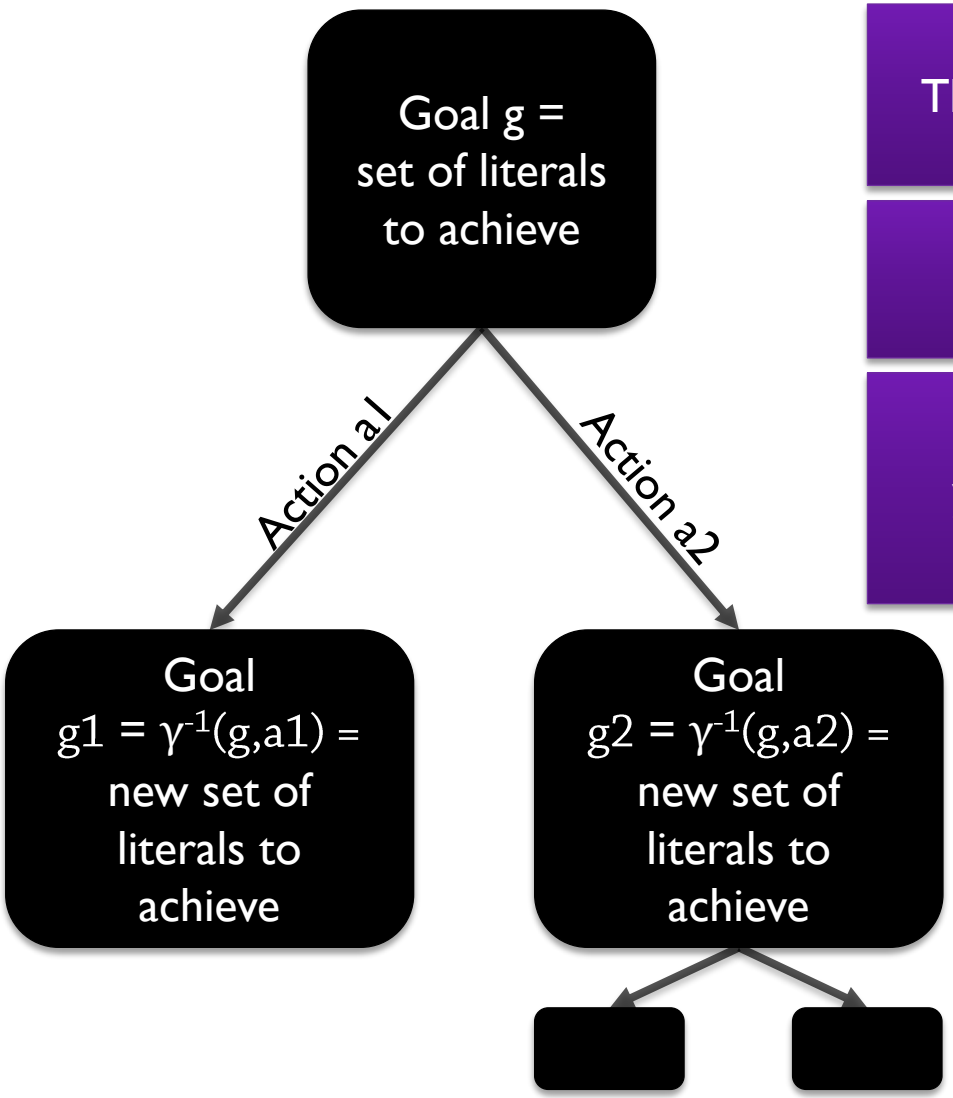


More sophisticated "bookkeeping" required!

POCL 2: Comparison to Backward Search

jonkv@ida

- Search tree for backward search, earlier:



The goal is a set of literals – simple!

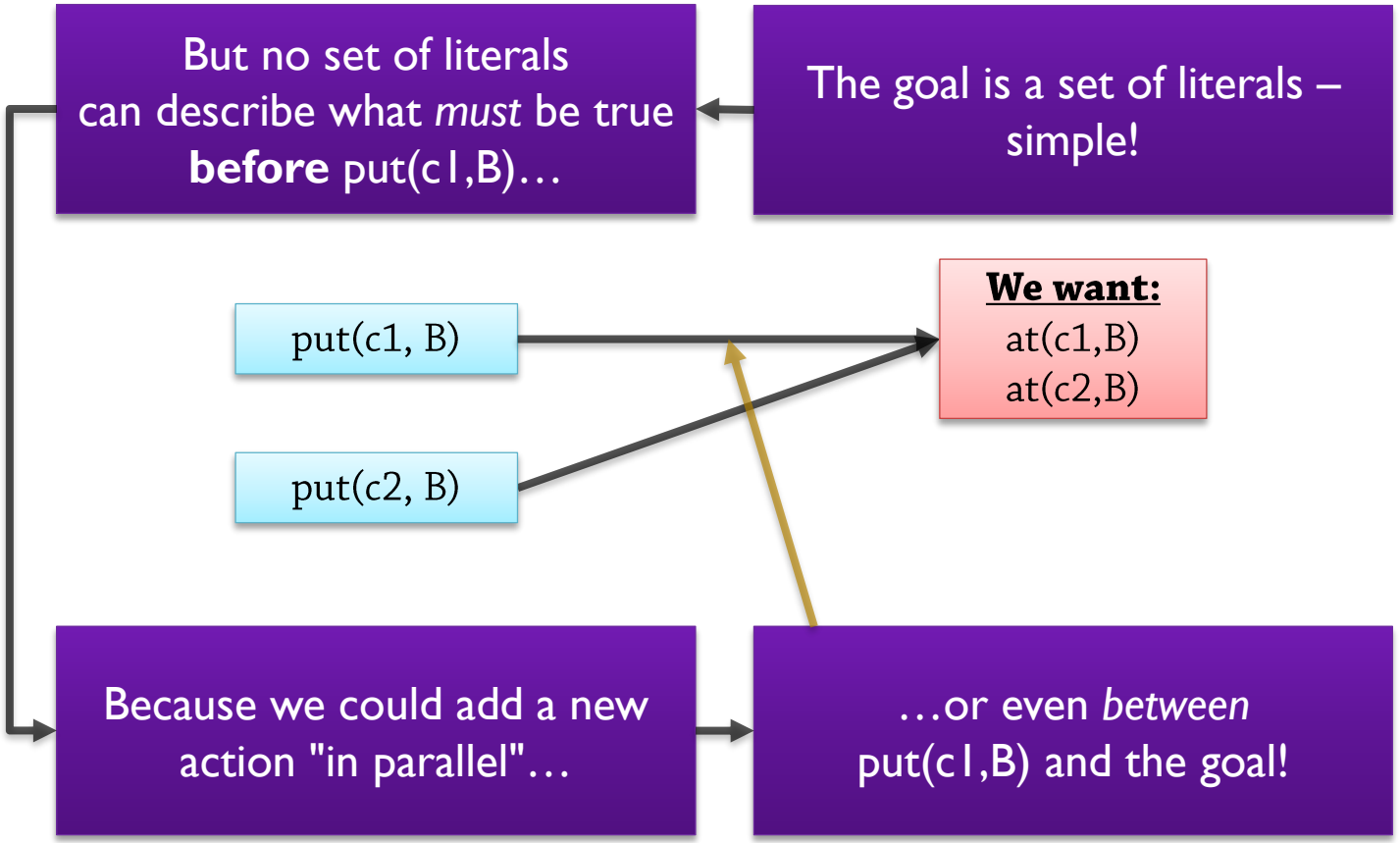
Every step takes you to
a new set of literals to achieve

From a search node,
you know how to reach the goal
using a *sequence* of actions

POCL 3: Comparison to Backward Search

jonkv@ida

- In POCL planning:
  - There is no sequence – and no clear ”before” relation!



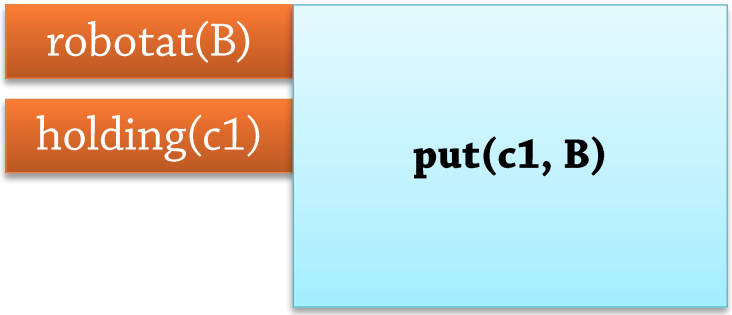
Has consequences for the POCL plan structure…

POCL 4: Conditions; Goal Action

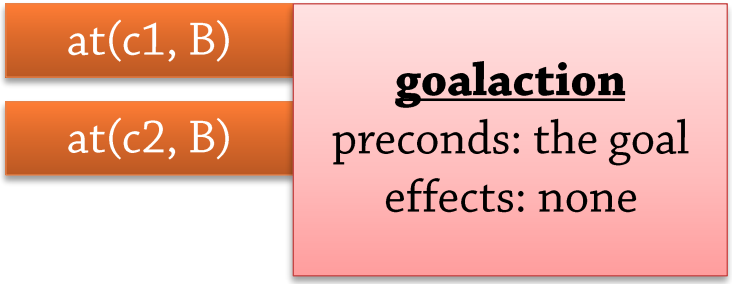
jonkv@ida

- Must keep track of **individual propositions to be achieved**
  - Throughout the plan – not a single state $g1 = \gamma^{-1}(g,a1)$
  - May come from **preconditions** of every action in the plan

Notation chosen for this presentation: Preconditions on the left/top side



  - May come from the **problem goal** as in backward search
    - Let's use a **uniform representation**
    - Add a "fake" **goal action** to every plan, with the goals as preconditions!

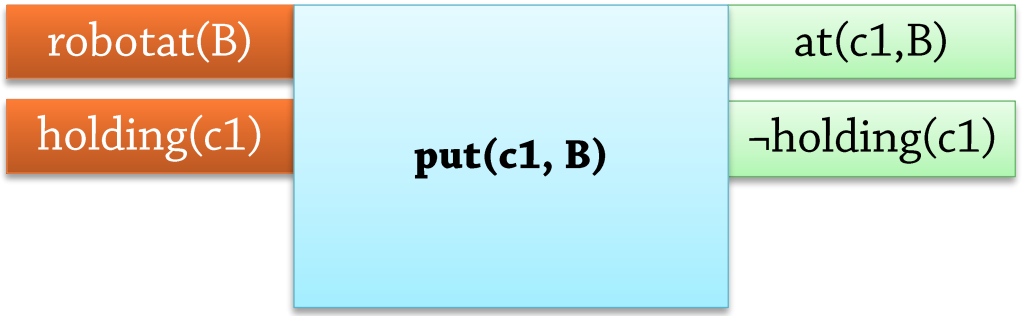


POCL 5: Effects; Initial Action

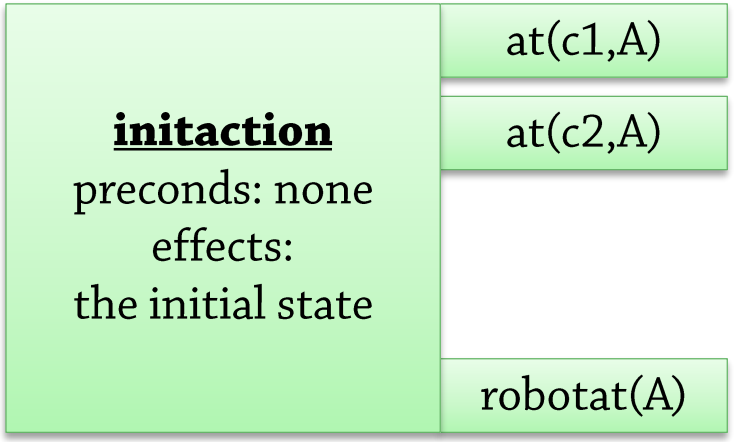
jonkv@ida

- Must keep track of **individual propositions that are achieved**
  - Throughout the plan – not from a single *relevant* action
  - May come from **effects** of every action in the plan

Notation chosen for this presentation: Effects on the right/bottom side



  - May come from the **initial state**
    - Add a "fake" **initial action** to every plan, with the initial state as effects!

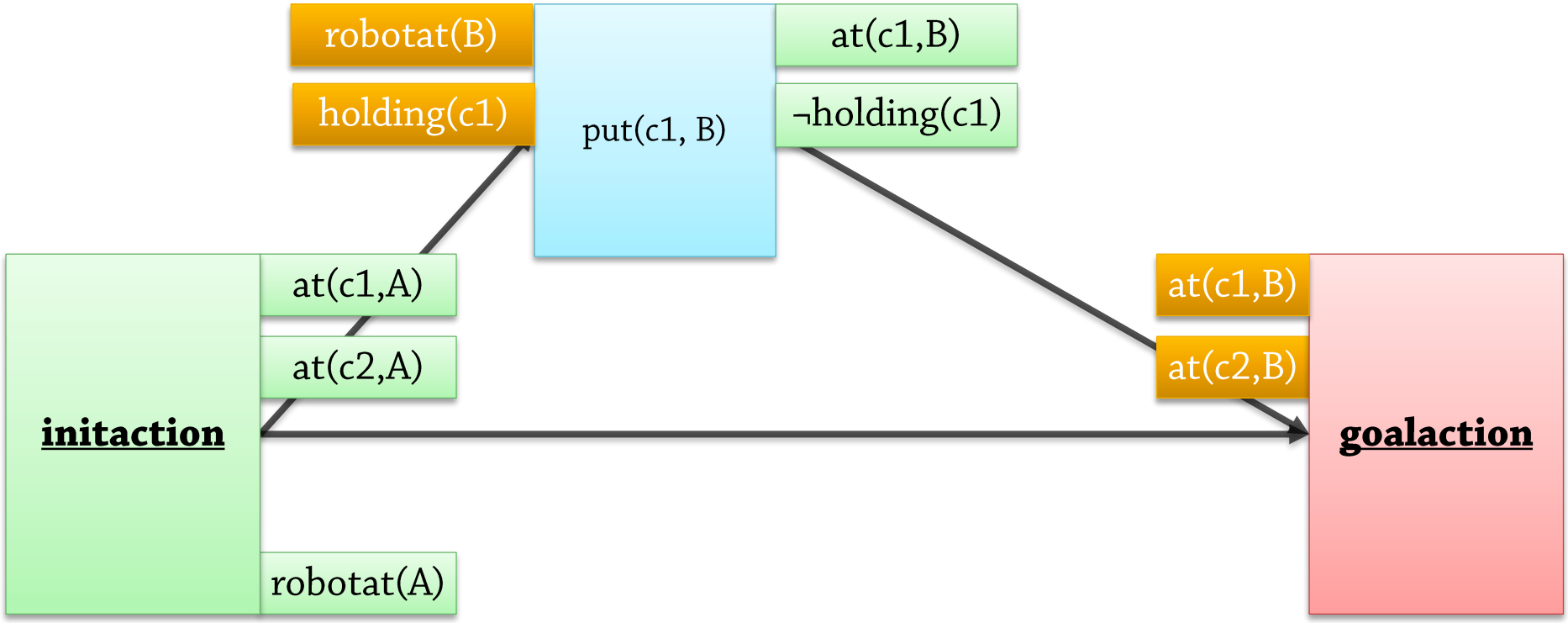


  - Effects are sometimes omitted from the slides, due to lack of space…

POCL 6: Precedence Constraints

jonkv@ida

- Plan structure so far:

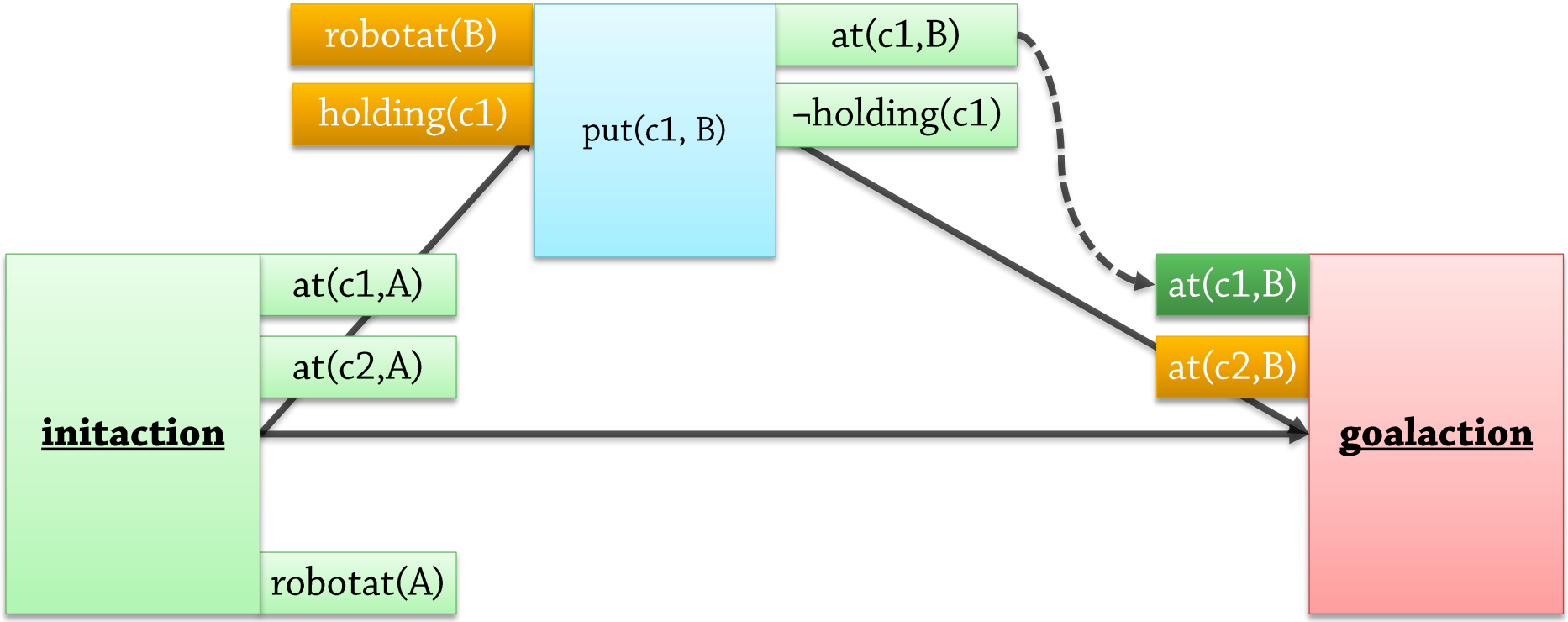


POCL 7: Causal Links

jonkv@ida

- Must keep track of **which** action achieves **which** precondition
  - **Causal links**

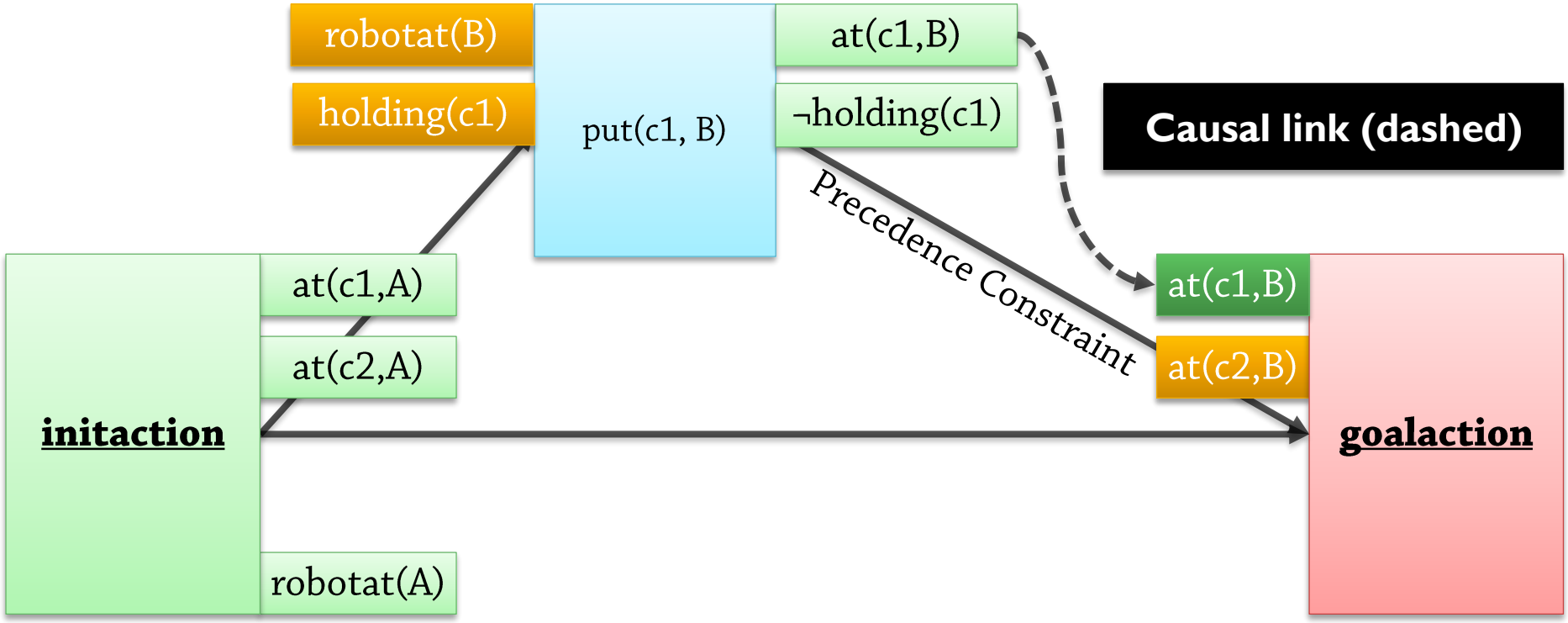
Causal link (dashed):
at(c1,B) must **remain true**
between the end of *put* and the beginning of *goalaction*.
No one must delete it!
Important for *threat management* (later)



POCL 8: Partial-Order Plans

jonkv@ida

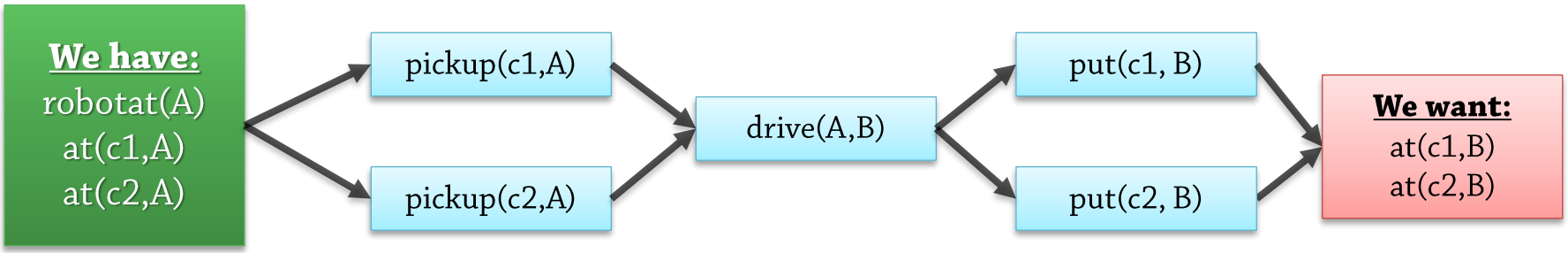
- To summarize, a ground **partial-order plan** consists of:
  - A set of **actions**
  - A set of **precedence constraints** $a \rightarrow b$
    - Action a must precede b
  - A set of **causal links** $a \xrightarrow{p} b$
    - Action a establishes the precond p needed by b



Partial-Order Solutions

jonkv@ida

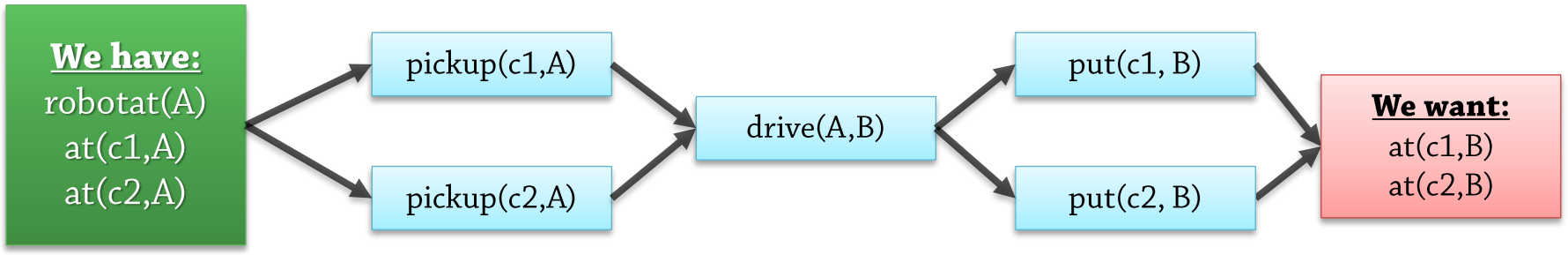
- Original motivation: **performance**
  - Therefore, a partial-order plan is a **solution** iff **all** **sequential** plans satisfying the ordering are solutions
    - Similarly, **executable** iff corresponding sequential plans are executable
    - <pickup(c1,A), pickup(c2,A), drive(A,B), put(c1,B), put(c2,B)>
    - <pickup(c2,A), pickup(c1,A), drive(A,B), put(c1,B), put(c2,B)>
    - <pickup(c1,A), pickup(c2,A), drive(A,B), put(c2,B), put(c1,B)>
    - <pickup(c2,A), pickup(c1,A), drive(A,B), put(c2,B), put(c1,B)>



Partial Orders and Concurrency

jonkv@ida

- Can be **extended** to allow **concurrent execution**
  - Requires a new formal model!
    - Our state transition model *does not define* what happens if c1 and c2 are picked up simultaneously!

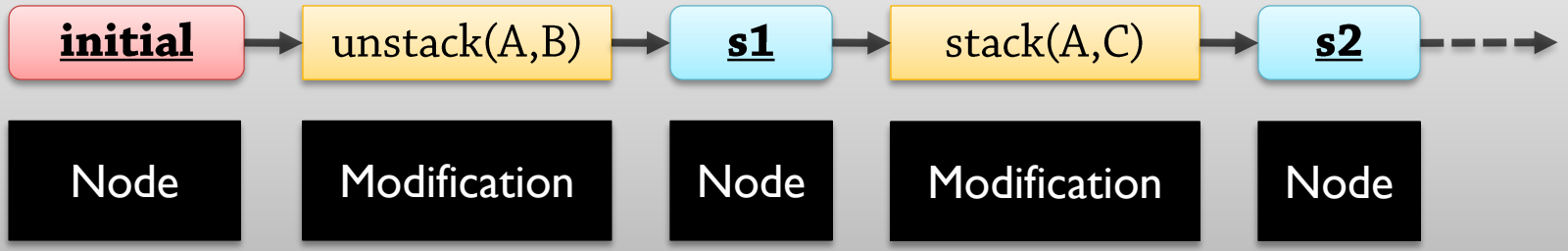


Generating Partial-Order Plans

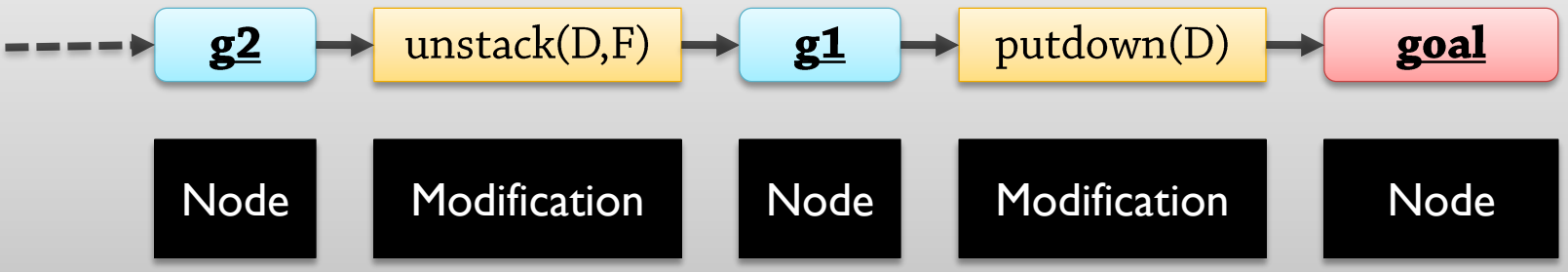
Context: Forward, Backward

jonkv@ida

Forward search: A search node is a "current state"



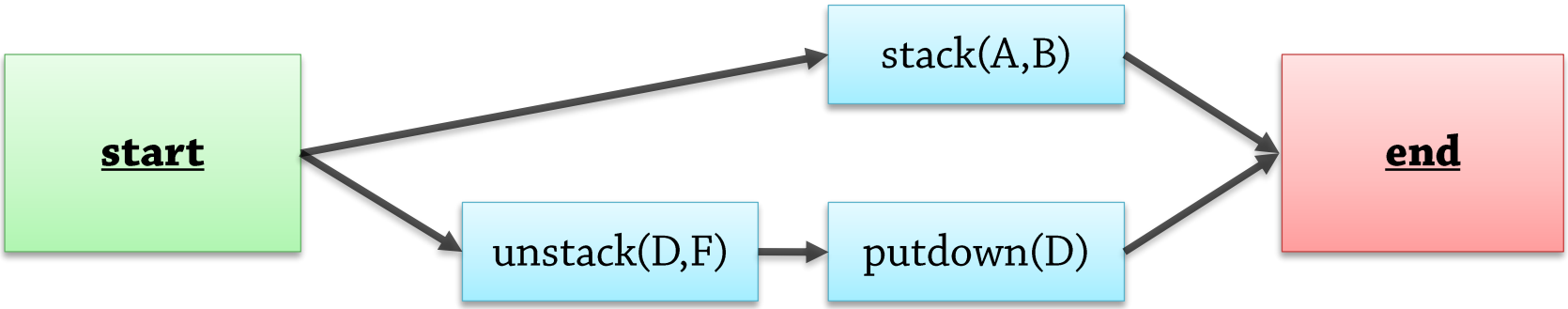
Backward search: A search node is a "current goal"



No Current State during Search!

jonkv@ida

- With **partial-order plans**: No "current" state or goal!
  - What is true after stack(A,B) below?
    - **Depends** on the order in which **other** actions are executed
    - **Changes** if we insert **new** actions **before** stack(A,B)!



A search node can't correspond to a state or goal!

Search Nodes are Partial Plans

jonkv@ida

- A node has to contain **more information**: **The entire plan**!
  - The **initial** search node contains the **initial plan**
    - The special **initial** and **goal actions**
    - A **precedence constraint**

Therefore, this is one form of "plan-space" planning!

Branching Rule

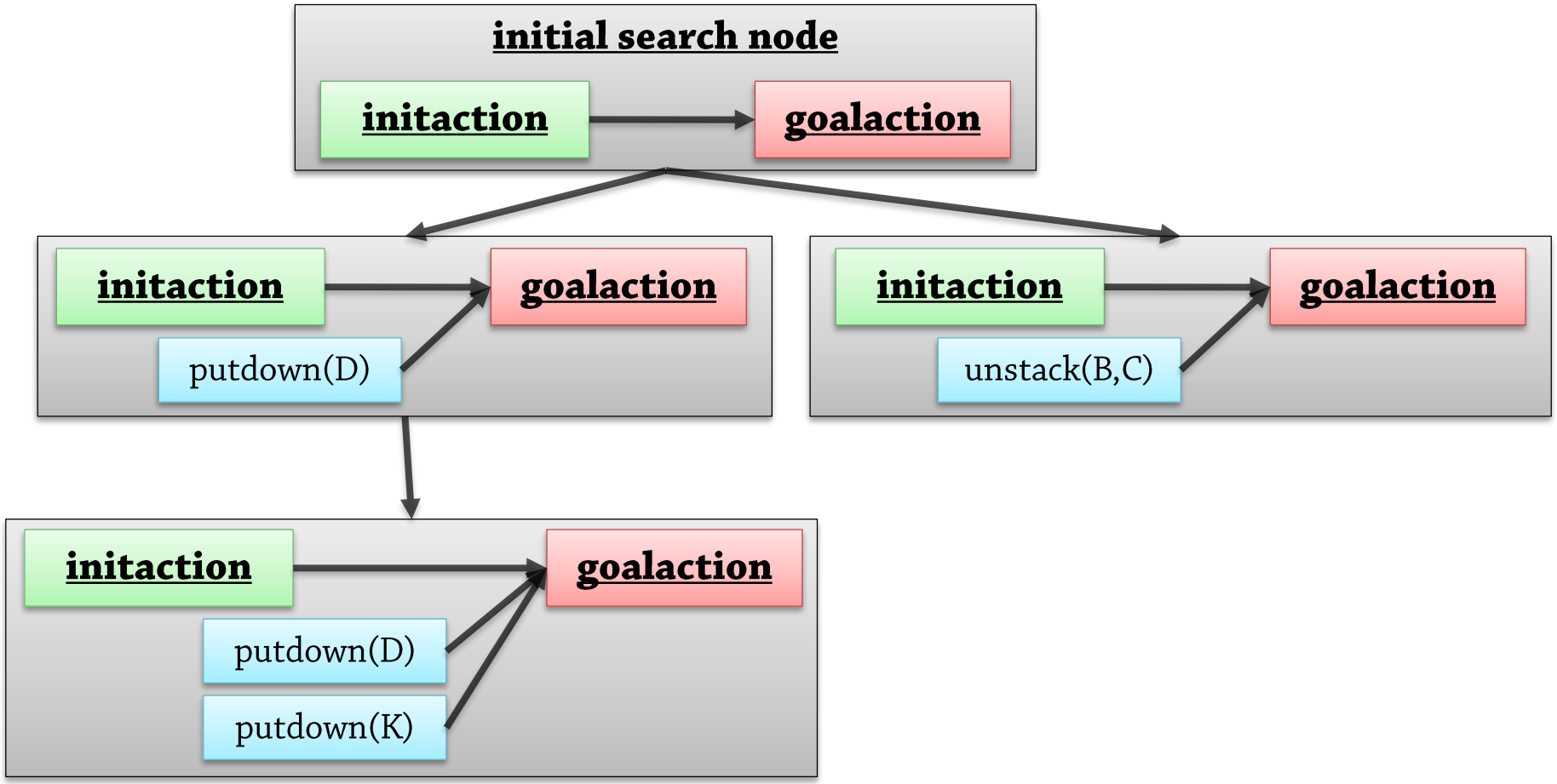
jonkv@ida

- We need a **branching rule** as well!
  - Forward planning: One successor per action **applicable** in *s*
  - Backward planning: One successor per action **relevant** to *g*
  - POCL planning: One successor for every way that a **flaw in the plan** (**open goal** or **threat**) can be **repaired**

Search Space

jonkv@ida

- Gives rise to a **search space**
  - Use search strategies, backtracking, heuristics, ... to search **this** space!



Successors fix Flaws:
Open Goals and Threats

Flaws

jonkv@ida

- Flaw, *noun*:
  1. a feature that mars the perfection of something; defect; fault: beauty without flaw; the flaws in our plan.
  2. a defect impairing legal soundness or validity.
  3. a crack, break, breach, or rent.

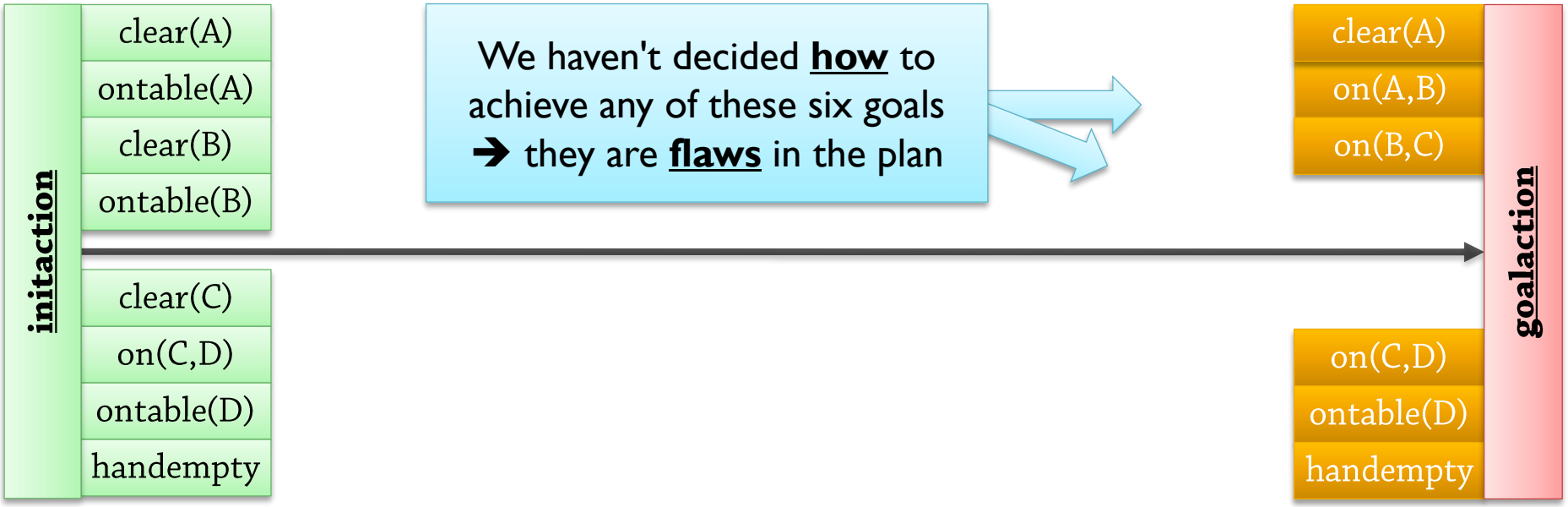
- Flaw, in POCL planning:
  - Something we **need to take care of** to **complete the plan**
  - Technical definition: An *open goal* or a *threat*

- Not:
  - Something that has ”gone wrong”
  - A problem during planning
  - A mistake in the final solution
  - …

Flaw Type 1: Open Goals

jonkv@ida

- Open goal:
  - An **action** *a* has a **precondition** *p* with **no incoming causal link**



clear(A) **is** already true in s0, but there is no causal link…

Adding one from s0 means clear(A) **must never be deleted**!
We need other alternatives too: Delete clear(A), then re-achieve it for goalaction…

Flaw Type 1: Open Goals

jonkv@ida

- To resolve an open goal :
  - Find an action *b* that causes *p*
    - Can be a *new* action
    - Can be an action *already* in the plan, if we can *make* it precede *a*
  - Add a **causal link**

Partial order! This was not possible in backward search…

Essential:
Even if there **is already** an action that causes p,
you can still add a **new** action that **also** causes p!

Resolving Open Goals 1

jonkv@ida

- Here: **Six** open goals
  - Could choose to **find support for clear(A)**:
    - From initaction
    - From a new unstack(B,A), unstack(C,A), or unstack(D,A)
    - From a new stack(A,B), stack(A,C), stack(A,D), or putdown(A)
  - Could choose to **find support for on(A,B)**:
    - Only from a new instance of stack(A,B)
  - …

8 distinct successors

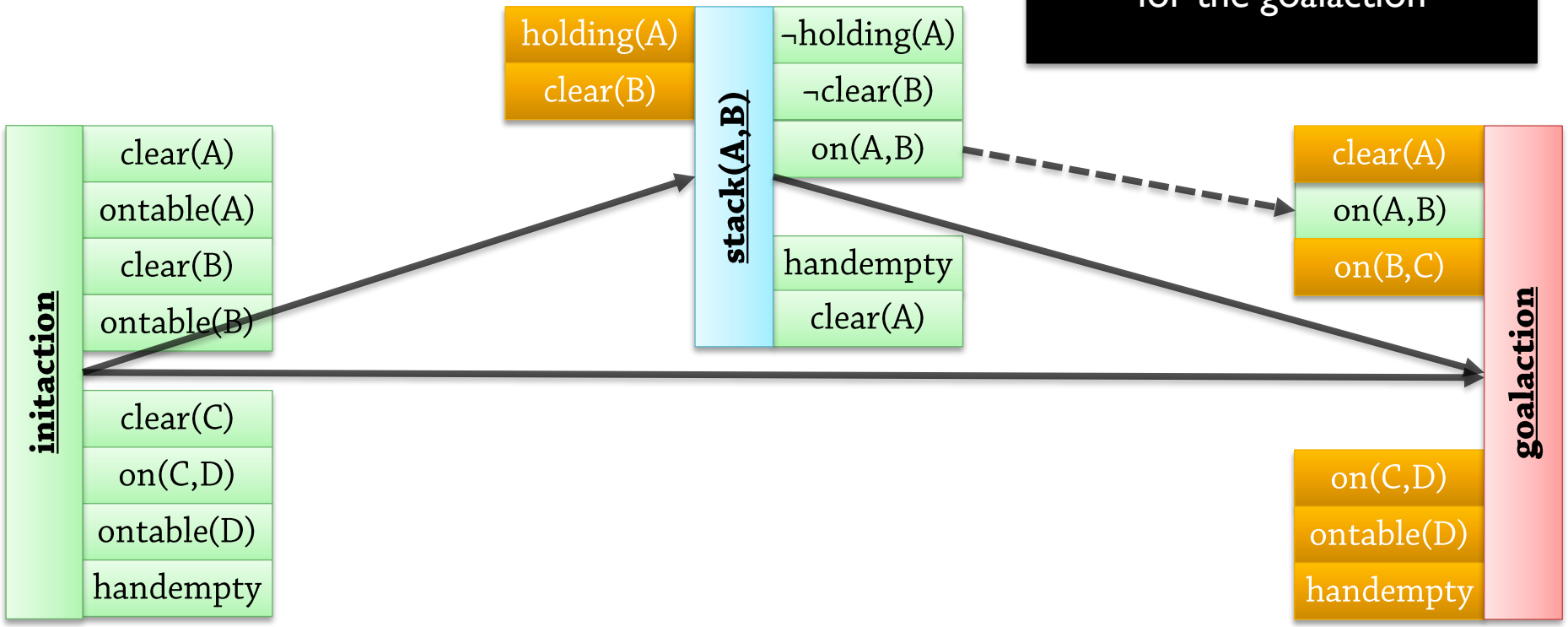
+1 successor

Resolving Open Goals 2

jonkv@ida

- Suppose we **add stack(A,B)** to **support (achieve) on(A,B)**
  - Must add a **causal link** for **on(A,B)**
    - Dashed line
  - Must **also** add precedence constraints
  - *Looks* totally ordered
    - Because it actually only has one "real" action…

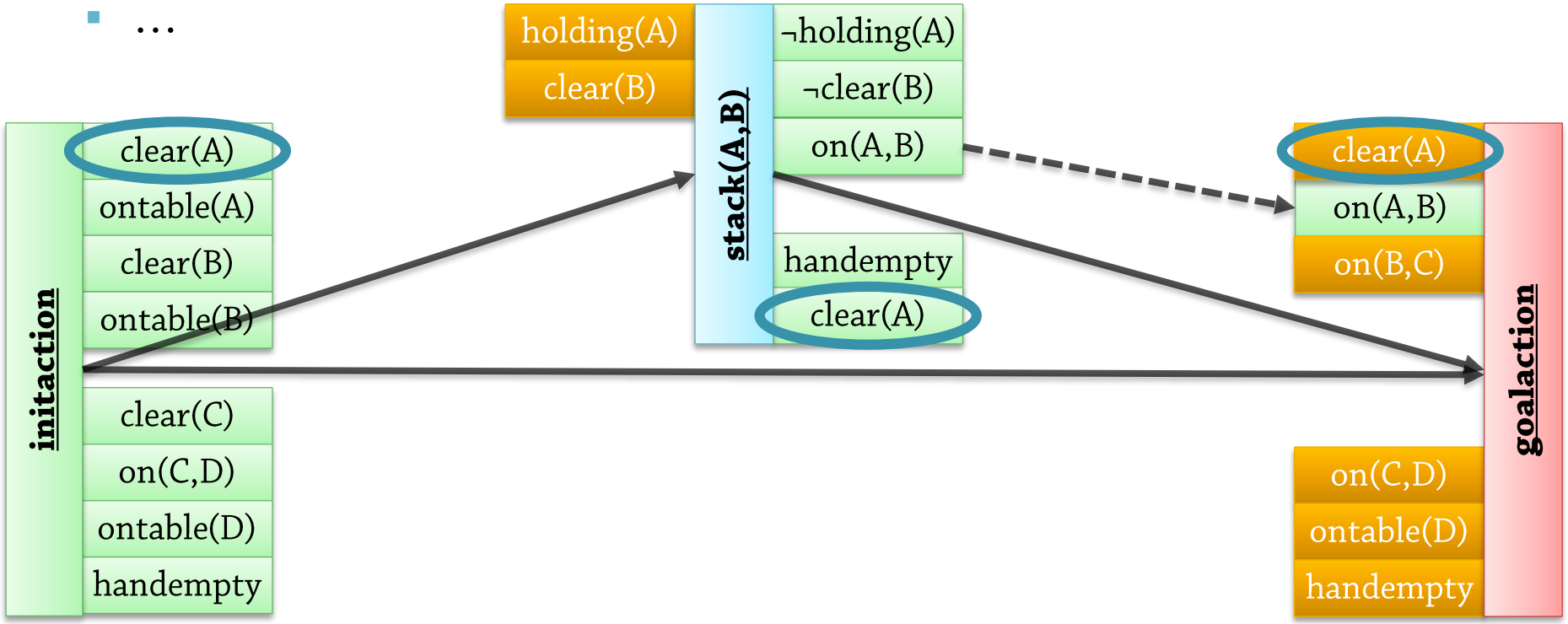
Causal link says:
This instance of stack(A,B)
is responsible for
achieving on(A,B)
for the goalaction



Resolving Open Goals 3

jonkv@ida

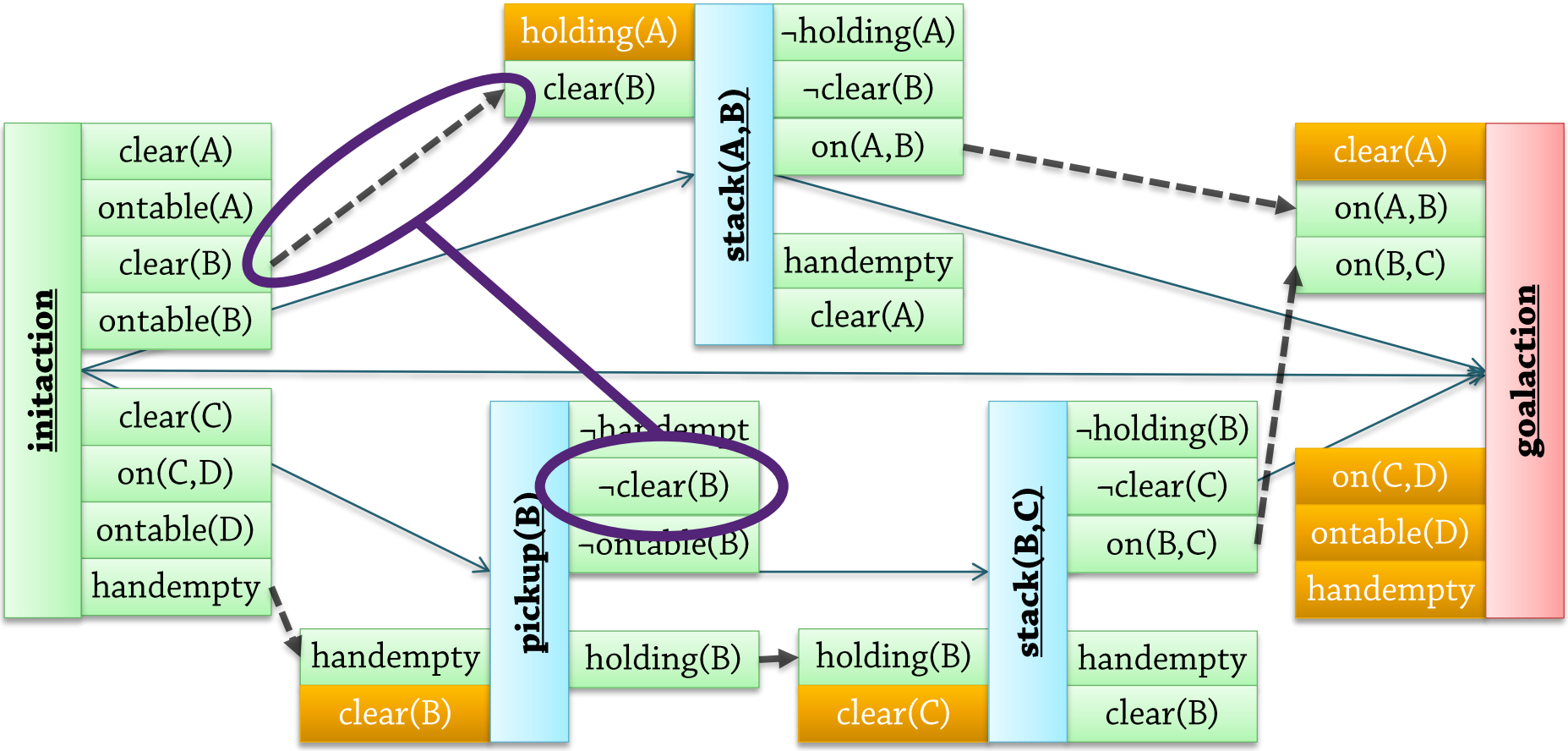
- Now: **7** open goals (one more!)
  - Can choose to find support for clear(A):
    - From the initaction
    - From the instance of **stack(A,B)** that we just added
    - From a **new** instance of **stack(A,B)**, stack(A,C), stack(A,D), or putdown(A)
    - From a **new** instance of unstack(B,A), unstack(C,A), or unstack(D,A)
  - …



Flaw Type 2: Threats

jonkv@ida

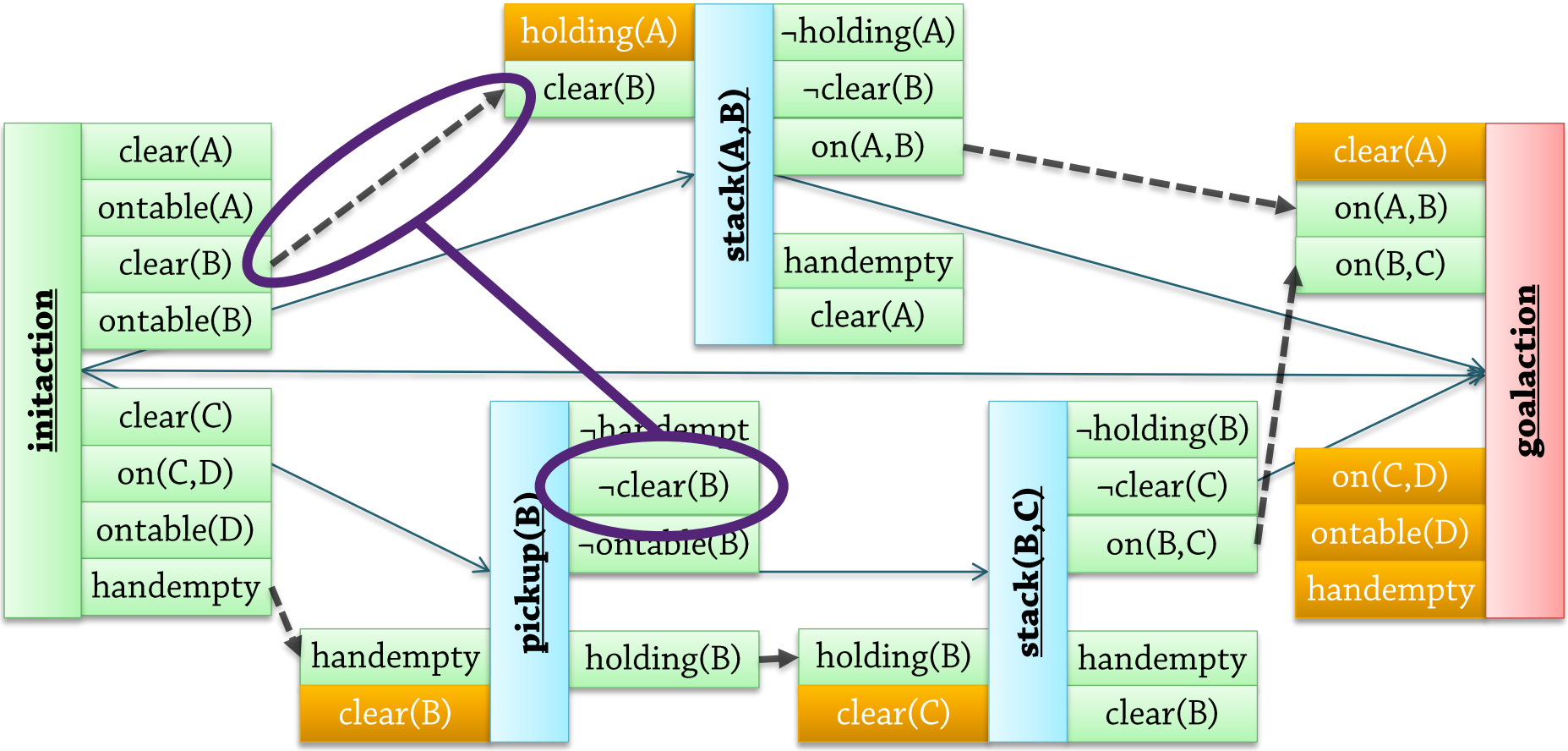
- Second flaw type: A **threat**
  - initaction **supports** clear(B) for stack(A,B) – there's a causal link
  - pickup(B) **deletes** clear(B), and may occur **between** initaction and stack(A,B)
  - So we can't be certain that clear(B) still holds when stack(A,B) starts!



Flaw Type 2: Threats (2)

jonkv@ida

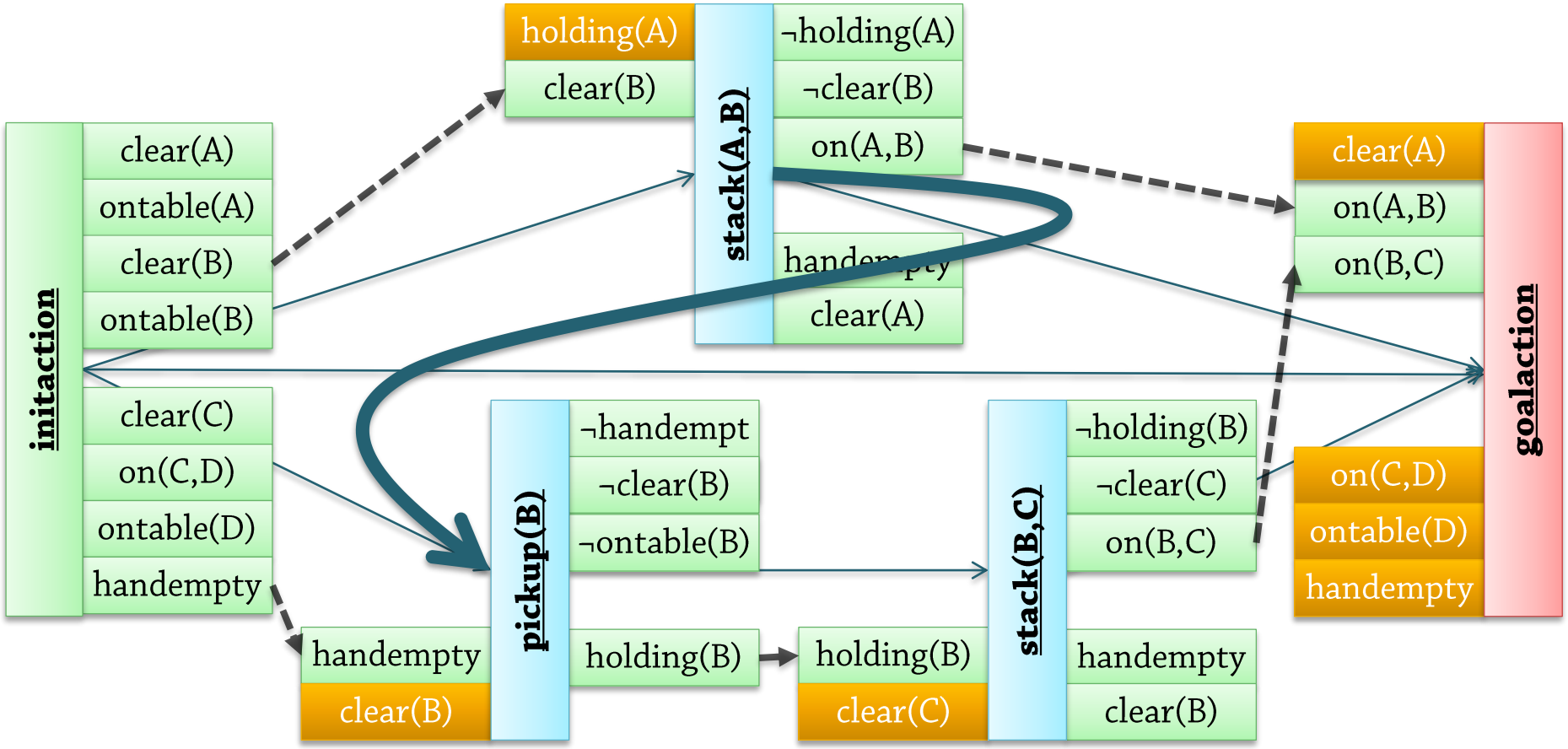
- Some possible **execution orders**:
  - <…, stack(A,B), pickup(B), …> -- preconditions of stack(A,B) OK
  - <…, pickup(B), stack(A,B), …> -- preconditions of stack(A,B) not satisfied



Resolving Threats 1

jonkv@ida

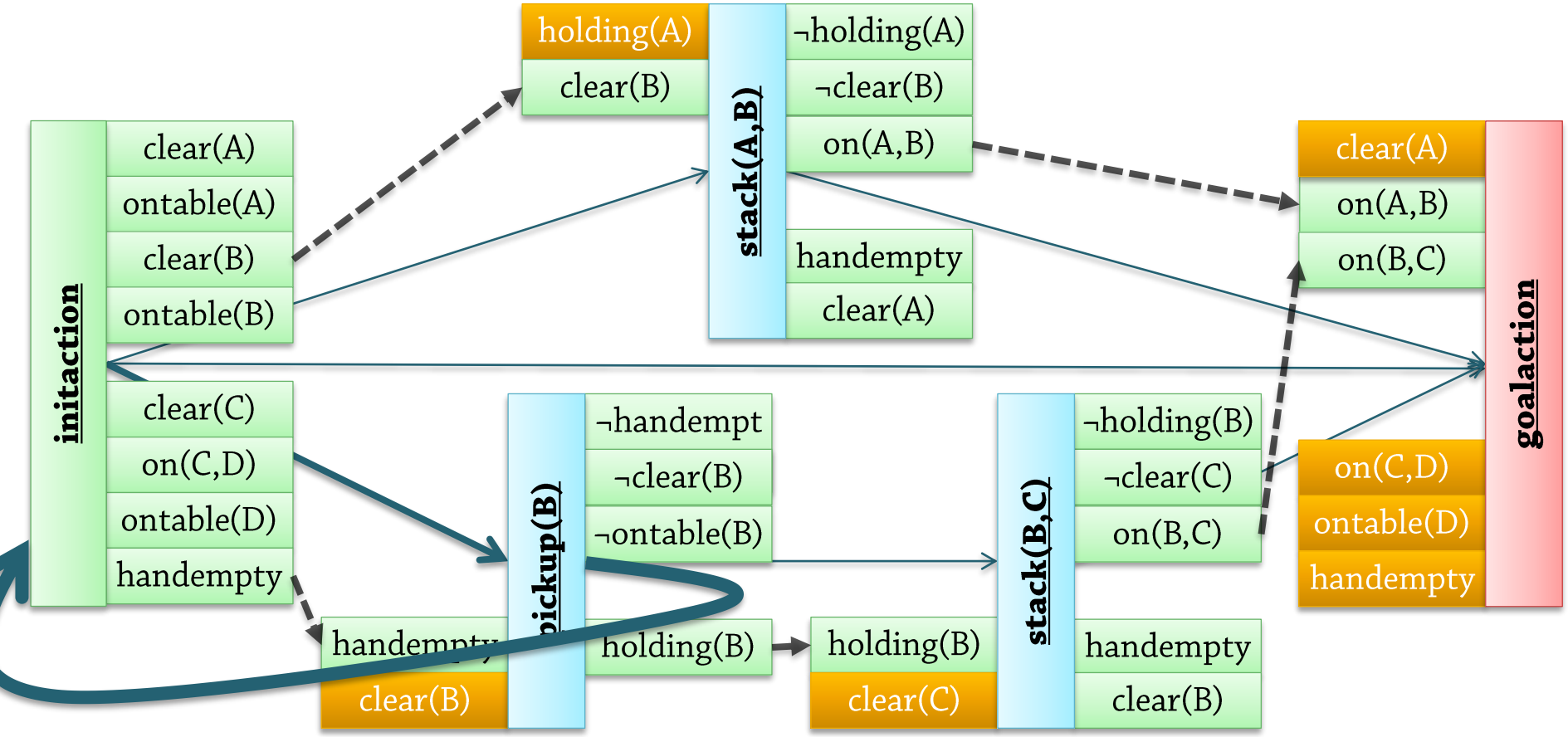
- How to **make sure** that clear(B) holds when stack(A,B) starts?
  - Alternative 1: The action that **disturbs** the precondition is placed **after** the action that **has** the precondition
    - Only possible if the resulting partial order is consistent (acyclic)!



Resolving Threats 2

jonkv@ida

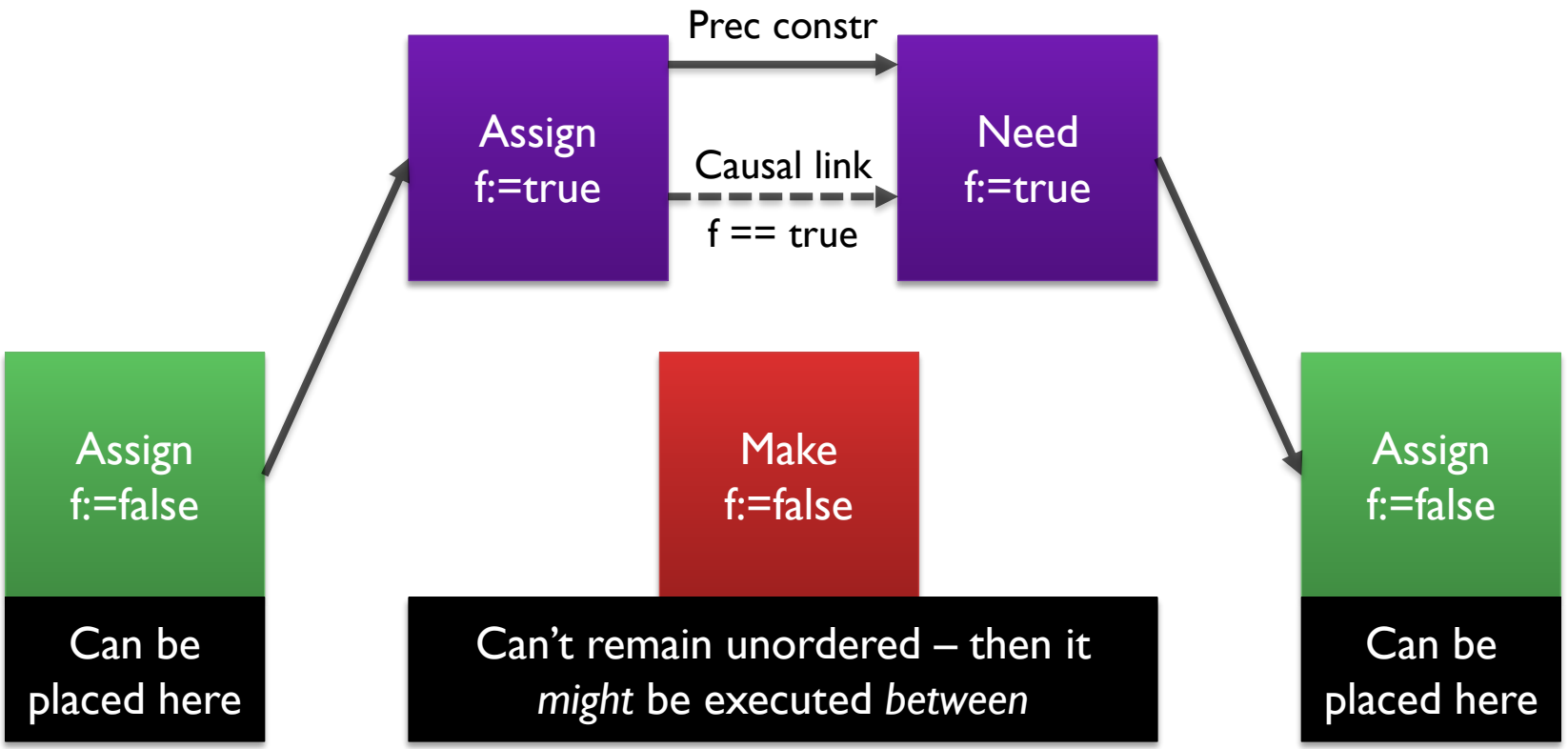
- Alternative 2:
  - The action that **disturbs** the precondition is placed **before** the action that **supports** the precondition
  - Only possible if the resulting partial order is consistent – not in this case!



Resolving Threats 3

jonkv@ida

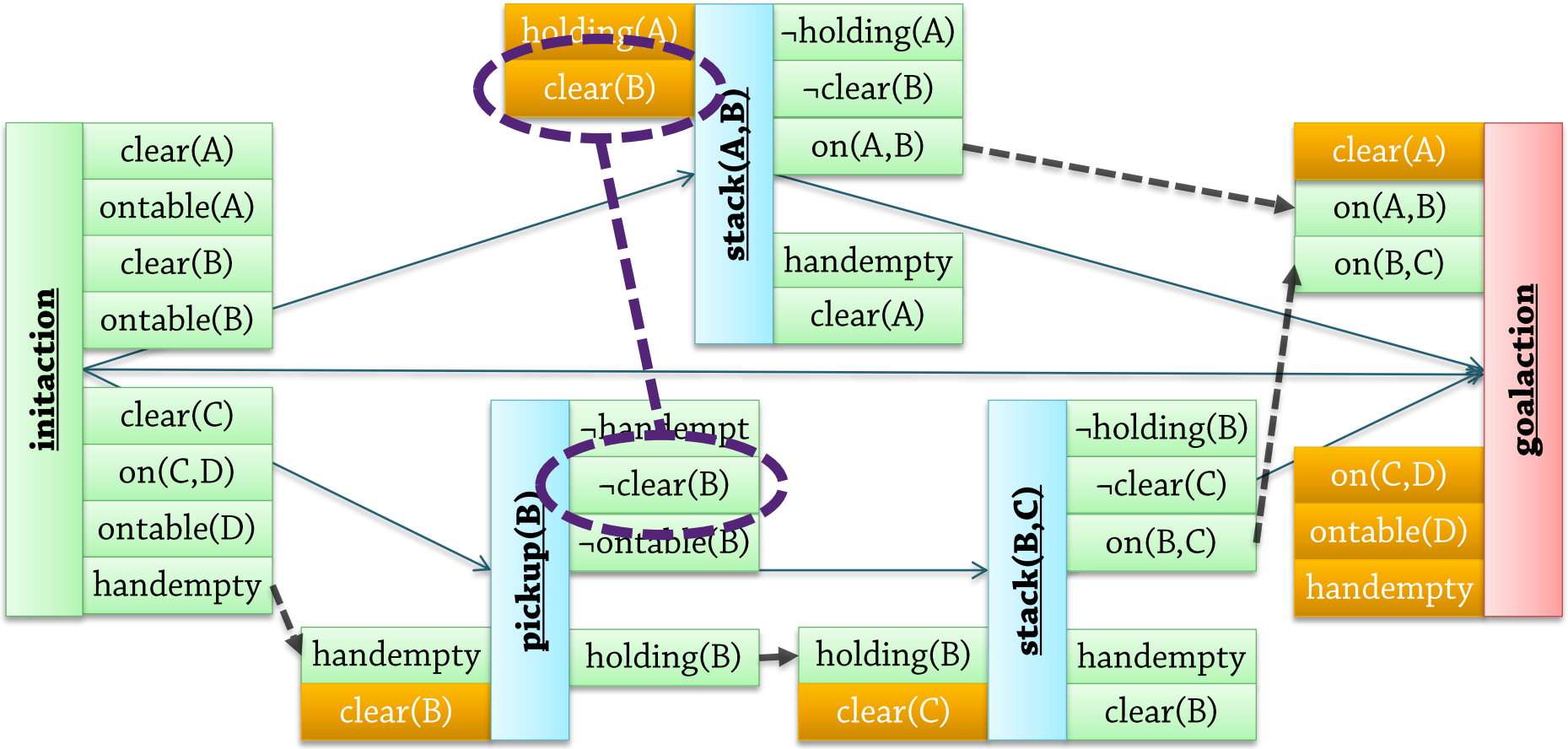
- Summary:



Resolving Threats 4

jonkv@ida

- Only **causal links** can be threatened!
  - Below, pickup(B) does **not** threaten the precond clear(B) of stack(A,B)
    - We haven't decided yet **how** to achieve clear(B): No incoming causal link
    - So we can't claim that its achievement is threatened!



POCL Algorithms

POCL planning

jonkv@ida

- **POCL planning – one possible formulation** (sound/complete):
  - planner(initial, goal):

```
open ← { Make-initial-plan(initial, goal) }
while open ≠ ∅:
    select a plan π in open
    open ← open – { π }
    flaws ← OpenGoals(π) ∪ Threats(π)
    if flaws = ∅ then
        return π
    else
        select any flaw φ ∈ flaws
        resolvers ← FindResolvers(φ, π)  // May be the empty set!
        foreach r in resolvers:
            π' ← Refine(r, π) // Actually apply the resolver
            open ← open ∪ { π' }
endwhile
return failure
```

Requires heuristics

Requires heuristics

Can prove: The plan is correct iff there are no remaining flaws (no open goals, no threats)

One flaw chosen – no backtracking!

But **all** resolvers must be tested…

Returns a **partially** ordered solution plan
Any **total** ordering of this plan will achieve the goals

The PSP Procedure

jonkv@ida

- **Plan-Space Planning (book)**: Same principle, slightly different
  - PSP(π)

    flaws ← OpenGoals(π) ∪ Threats(π)
    if *flaws* = ∅ **then return** π

    select any flaw φ ∈ *flaws*
    resolvers ← FindResolvers(φ, π)
    if *resolvers* = ∅ **then return** failure // *Backtrack*

    nondeterministically choose a resolver ρ ∈ *resolvers*
    π' ← Refine(ρ, π) // *Actually apply the resolver*
    return PSP(π')
    end

    Nondeterministic choice instead of "foreach resolver", list of open nodes

  - Call PSP(the initial plan)

Understanding Backtracking in POCL

jonkv@ida

```
planner(initial, goal):
   open ← { Make-initial-plan(initial, goal) }
   while open ≠ ∅:
      select a plan π in open
      open ← open – { π }
      flaws ← OpenGoals(π) ∪ Threats(π)
      if flaws = ∅ then
         return π
      else
         select any flaw φ ∈ flaws
         resolvers ← FindResolvers(φ, π)
         foreach r in resolvers:
            π' ← Refine(ρ, π) // Actually apply resolver
            open ← open ∪ { π' }
   endwhile
   return failure
```

If we choose a random flaw to resolve and then can't find a plan, we don't have to try another flaw. **Why?**

1) Every flaw *has* to be resolved
2) Choosing this flaw *later* cannot help us resolve it
3) Choosing this flaw *later* cannot help us resolve some other flaw

We must be able to try different resolvers.
Why?

Choosing one resolver *can* prevent other problem resolutions.

- Open goal: Use action A or B?
- Threat: Which order to choose?

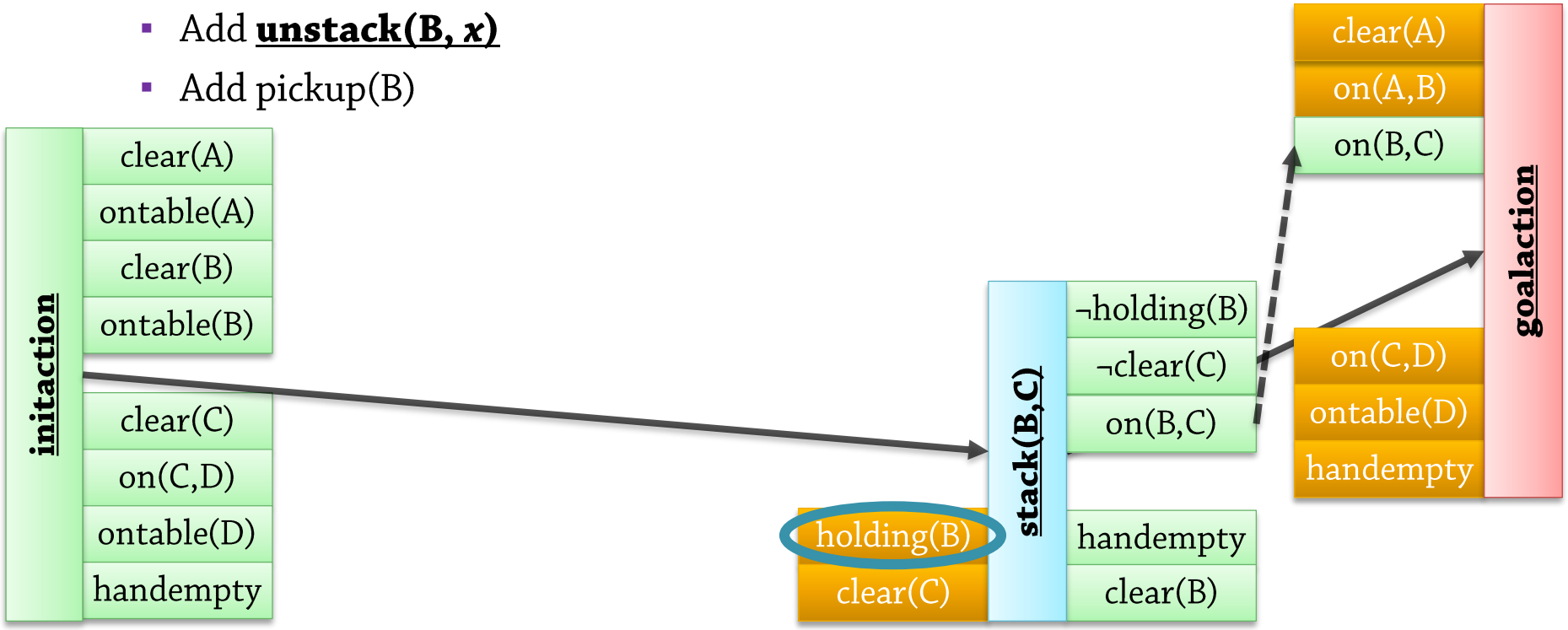
Lifted Planning:
Partial Action Instantiation

Partial Instantiation

jonkv@ida

- Suppose we want to achieve holding(B)
  - **Ground** search generates **many** alternatives
    - Add unstack(B,A), unstack(B,F), unstack(B,G), …
    - Add pickup(B)
  - Let's take the idea of **least commitment** one step further
  - **Lifted** search generates two **partially instantiated** alternatives
    - Add **unstack(B, *x*)**
    - Add pickup(B)

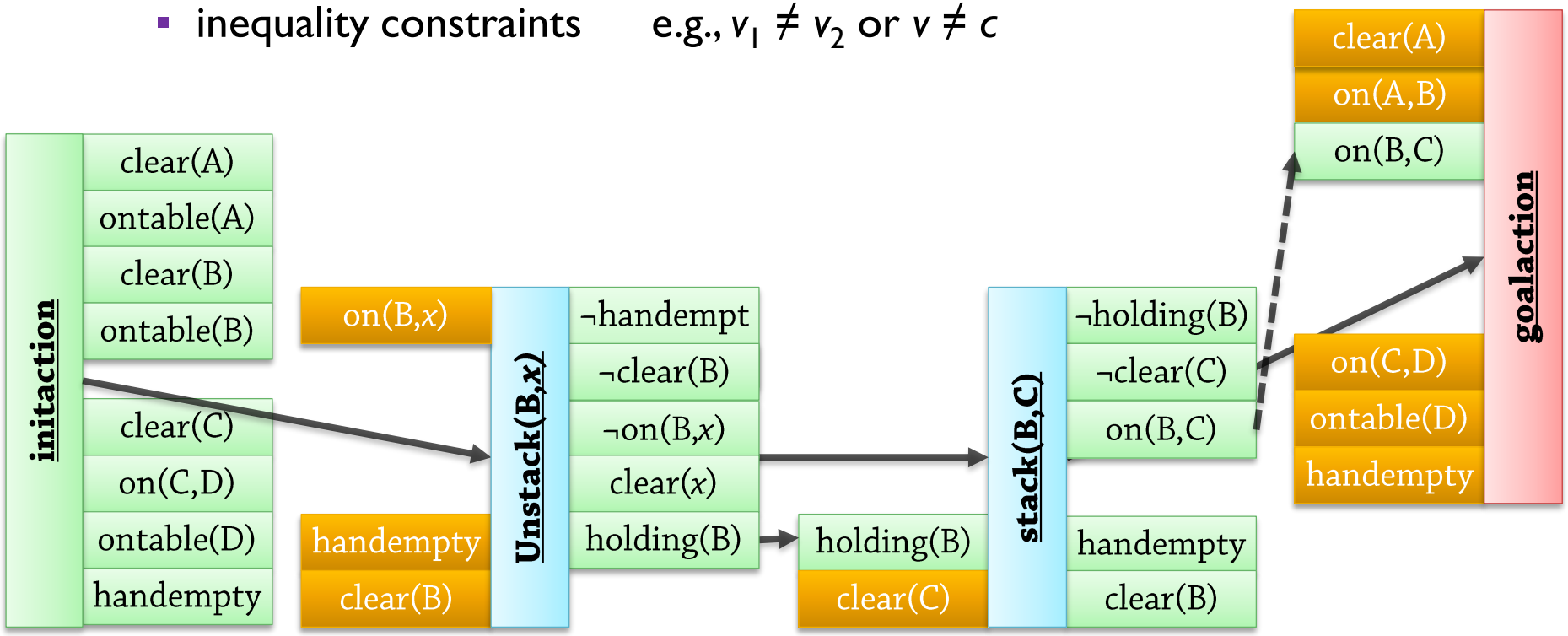
So far, we see no reason why we should unstack B from any **specific** block!



Partial-Order Plans

jonkv@ida

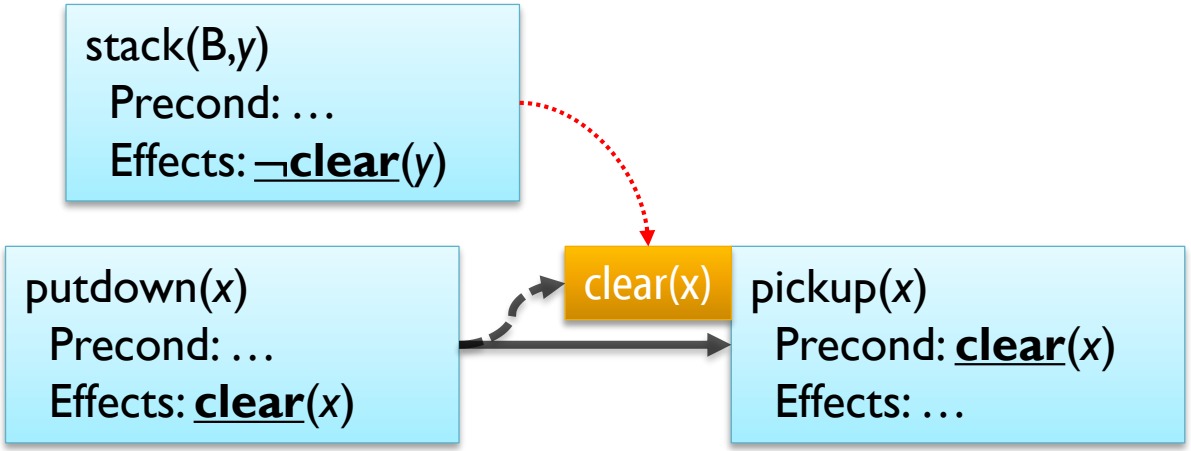
- A **lifted partial-order plan** consists of:
  - A set of **possibly unground actions**
  - A set of **precedence constraints**: *a* must precede *b*
  - A set of **causal links**: action *a* establishes the precond *p* needed by *b*
  - A set of **binding constraints**:
    - equality constraints e.g., $v_1 = v_2$ or $v = c$
    - inequality constraints e.g., $v_1 \neq v_2$ or $v \neq c$



Resolving Threats

jonkv@ida

- Another way of **resolving threats** for lifted plans:
  - For partly uninstantiated actions, we may find **potential** threats
    - stack(B,y) **may** threaten the causal link, but only if x=y
    - Can be resolved by adding a constraint: x != y



Complete Example

Example

jonkv@ida

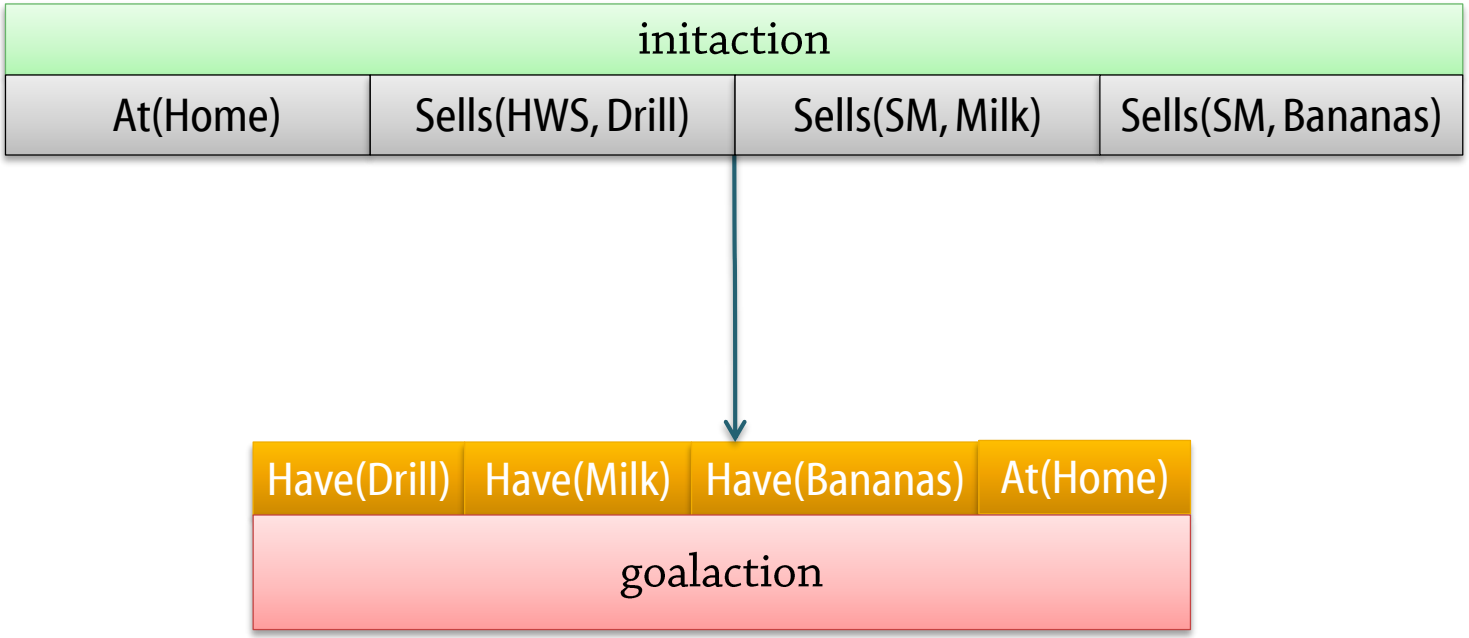
- Running Example: Similar to an example in AIMA
  - Russell and Norvig's *Artificial Intelligence: A Modern Approach* (1st ed.)

  - Operator **Go(from,to)**
    - Precond: At(from)
    - Effects: At(to), ¬At(from)
  - Operator **Buy(product, store)**
    - Precond: At(store), Sells(store, product)
    - Effects: Have(product)
  - **Initial state**
    - At(Home), Sells(HWS, Drill), Sells(SM, Milk), Sells(SM, Bananas)
  - **Goal**
    - At(Home), Have(Drill), Have(Milk), Have(Bananas)

Example (continued)

jonkv@ida

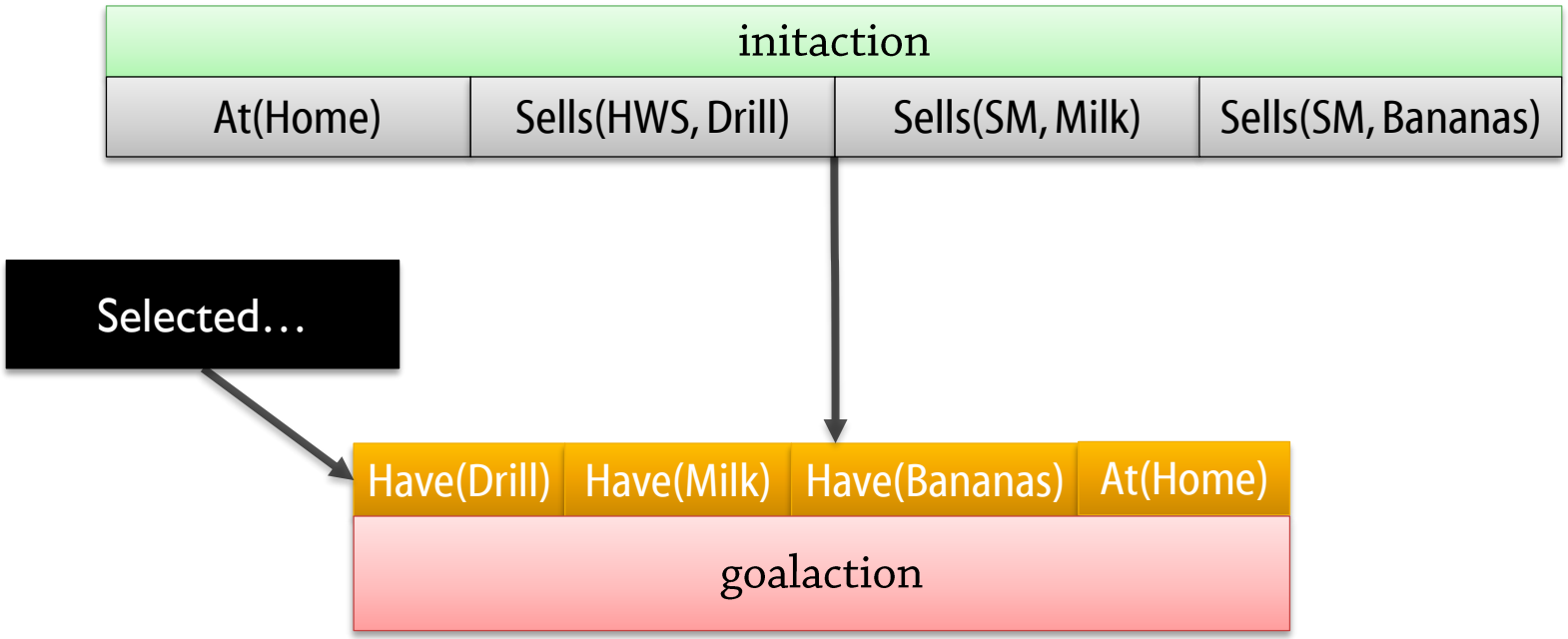
- PSP takes a plan π as its argument
  - Initial plan: **initaction**, **goalaction**, and an ordering constraint



Example (continued)

jonkv@ida

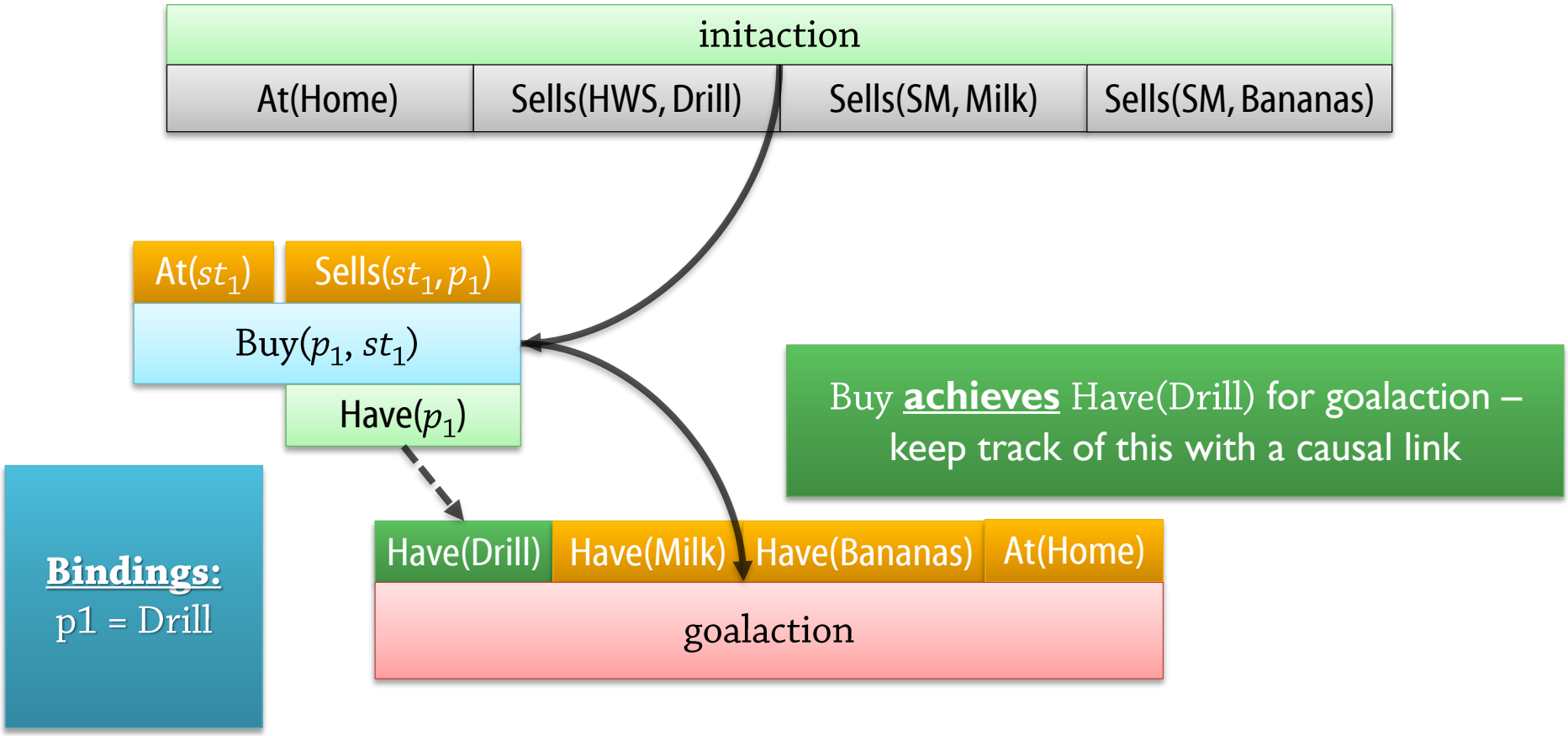
- Four **flaws** exist: Open goals
  - Suppose our heuristics tell us to resolve Have(Drill) first



Example (continued)

jonkv@ida

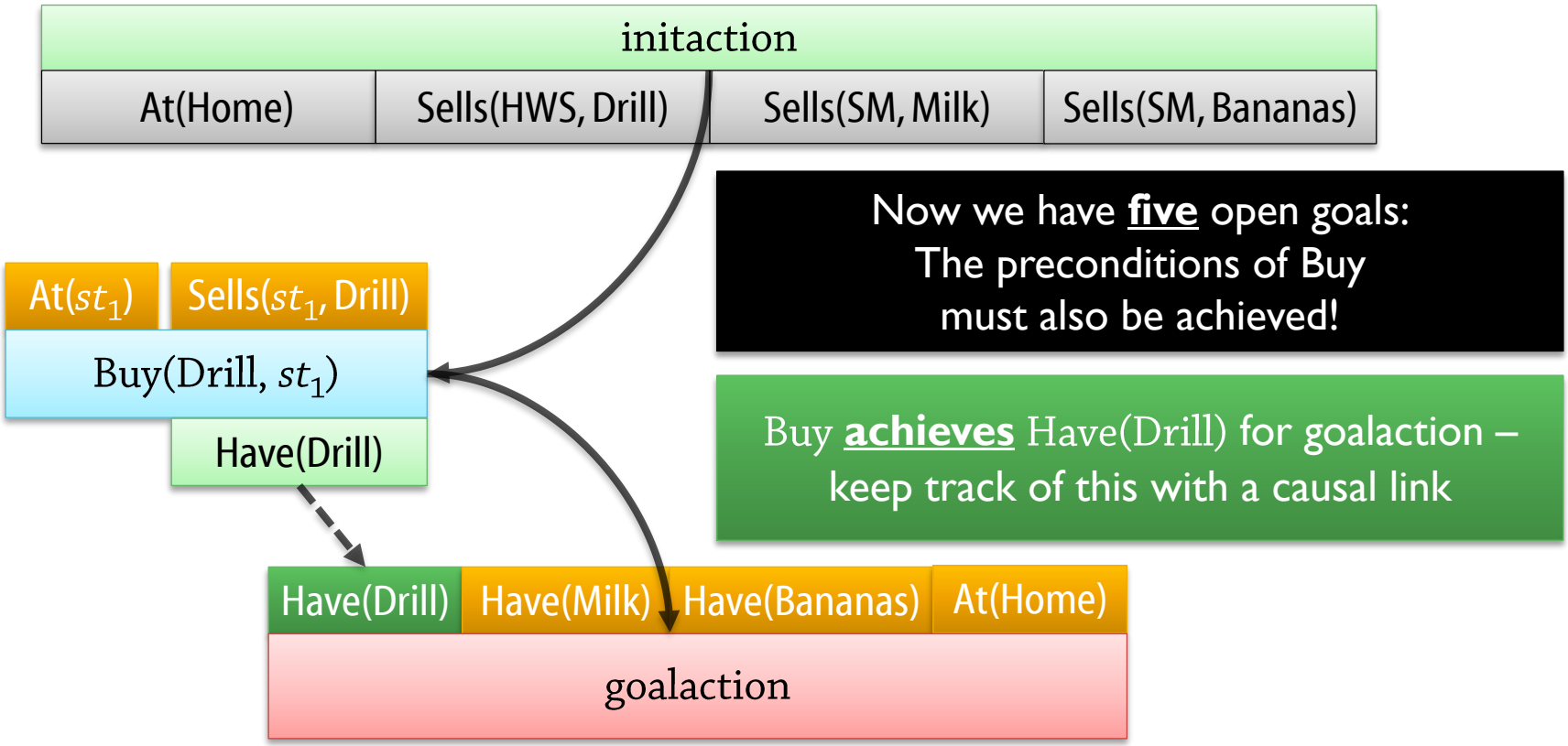
- Have(drill) is not achieved by any action in the current plan
- But **Buy(product, store)** achieves Have(product)
  - Partially instantiate:
    Buy(Drill, store) (right now we don't care **where** we buy it)



Example (continued)

jonkv@ida

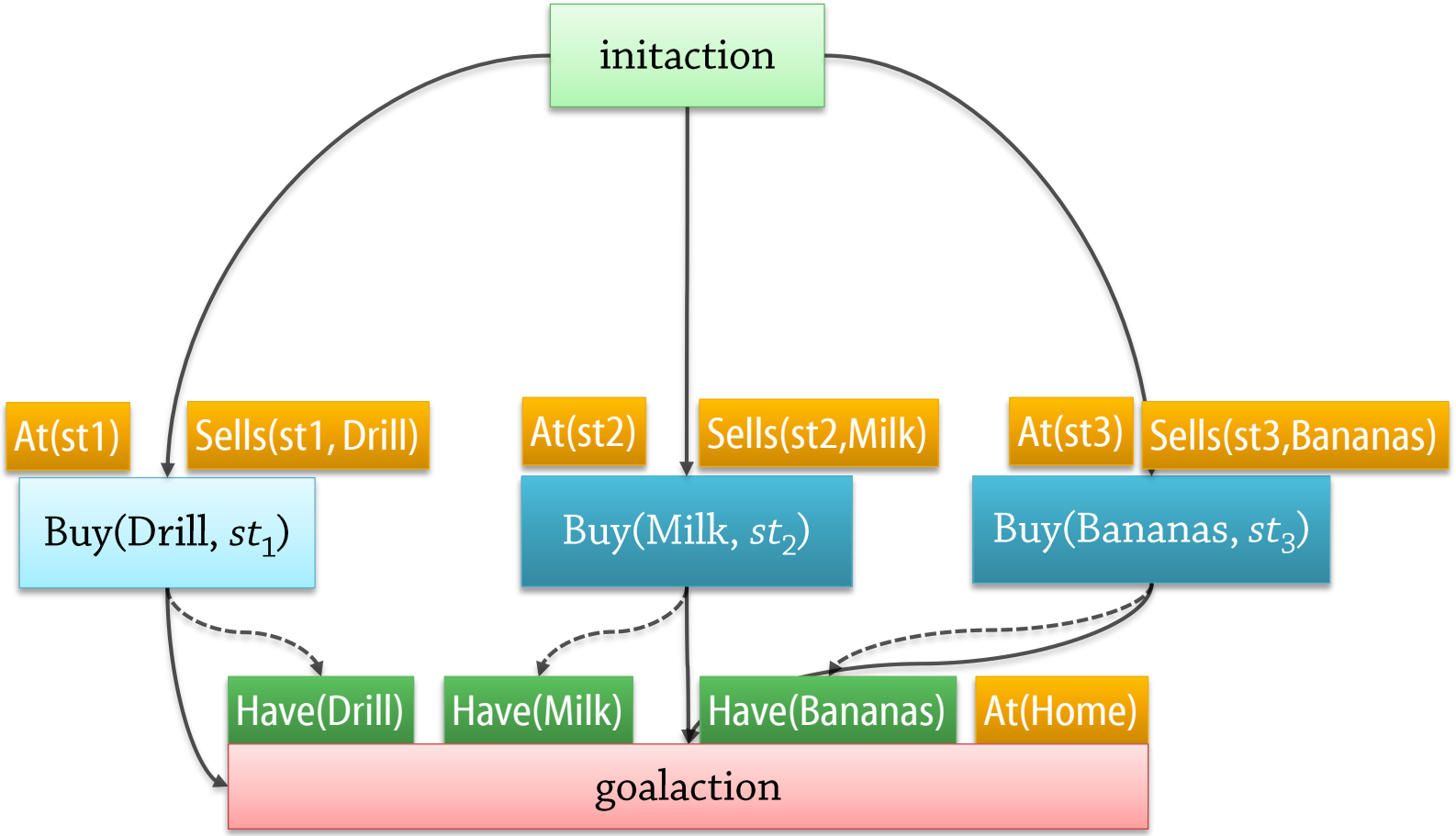
- **Alternative Notation** for simplicity
  - Variable bindings are implicit in the diagram



Example (continued)

jonkv@ida

- The first **three** refinement steps
  - These are the only possible ways to establish the Have preconditions
  - We don't care in which **order** we buy things!

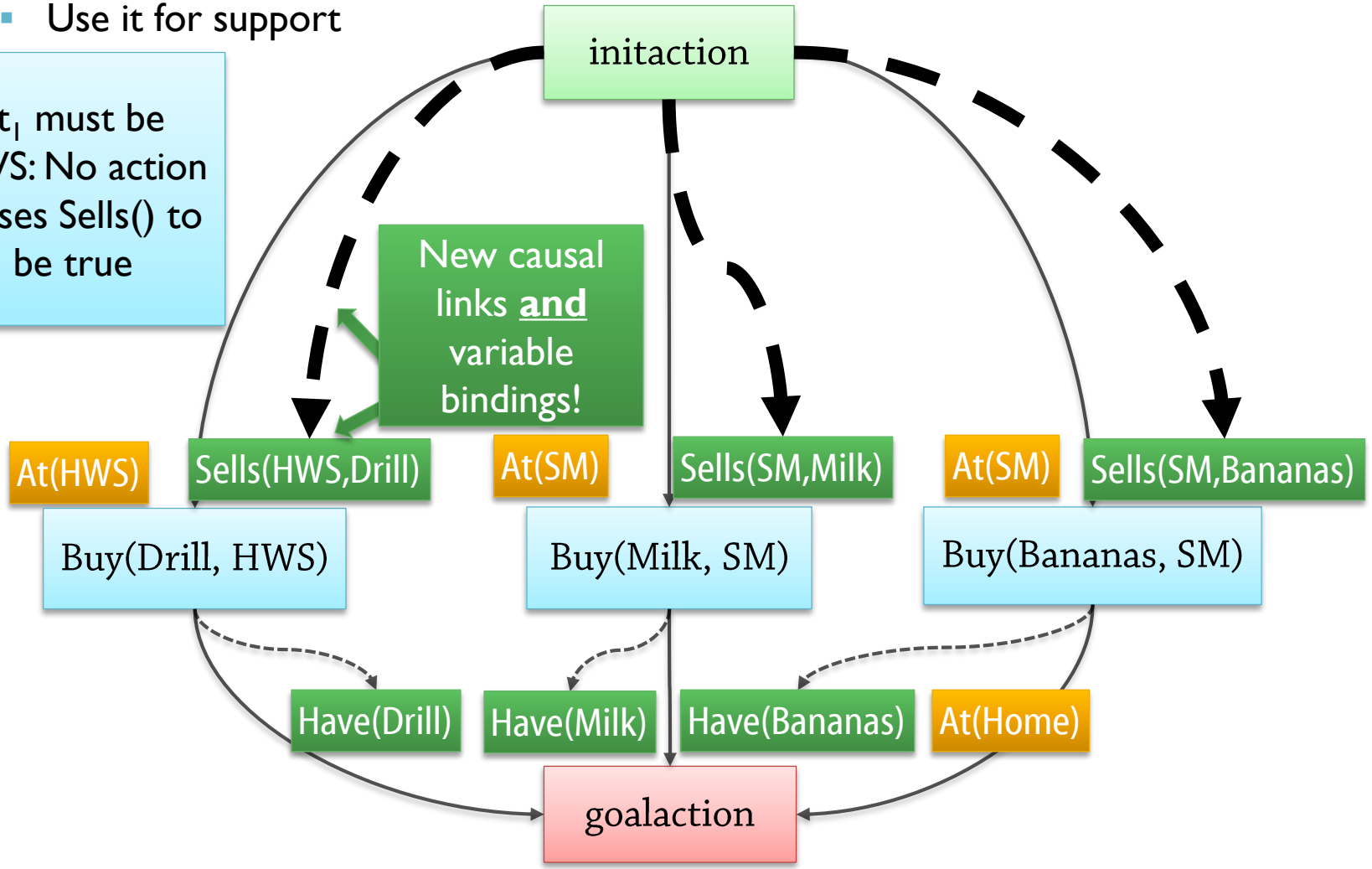


Example (continued)

jonkv@ida

- Three more refinement steps
  - No action causes Sells(…) to be true – except the "fake" initial action!
  - Use it for support

st_1 must be HWS: No action causes Sells() to be true

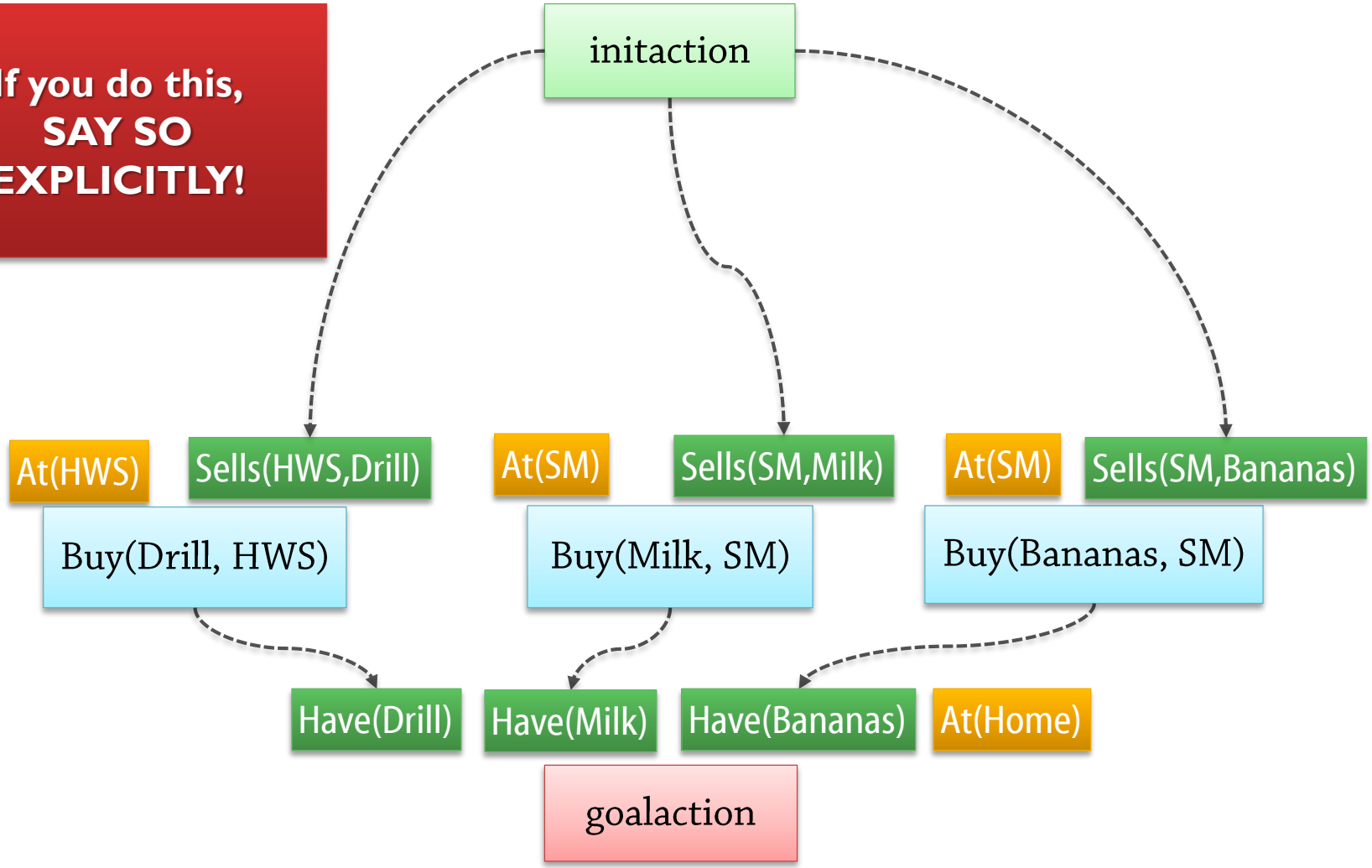


Example (continued)

jonkv@ida

- It's getting messy!
  - Let's omit the precedence constraints that are implicit in causal links…

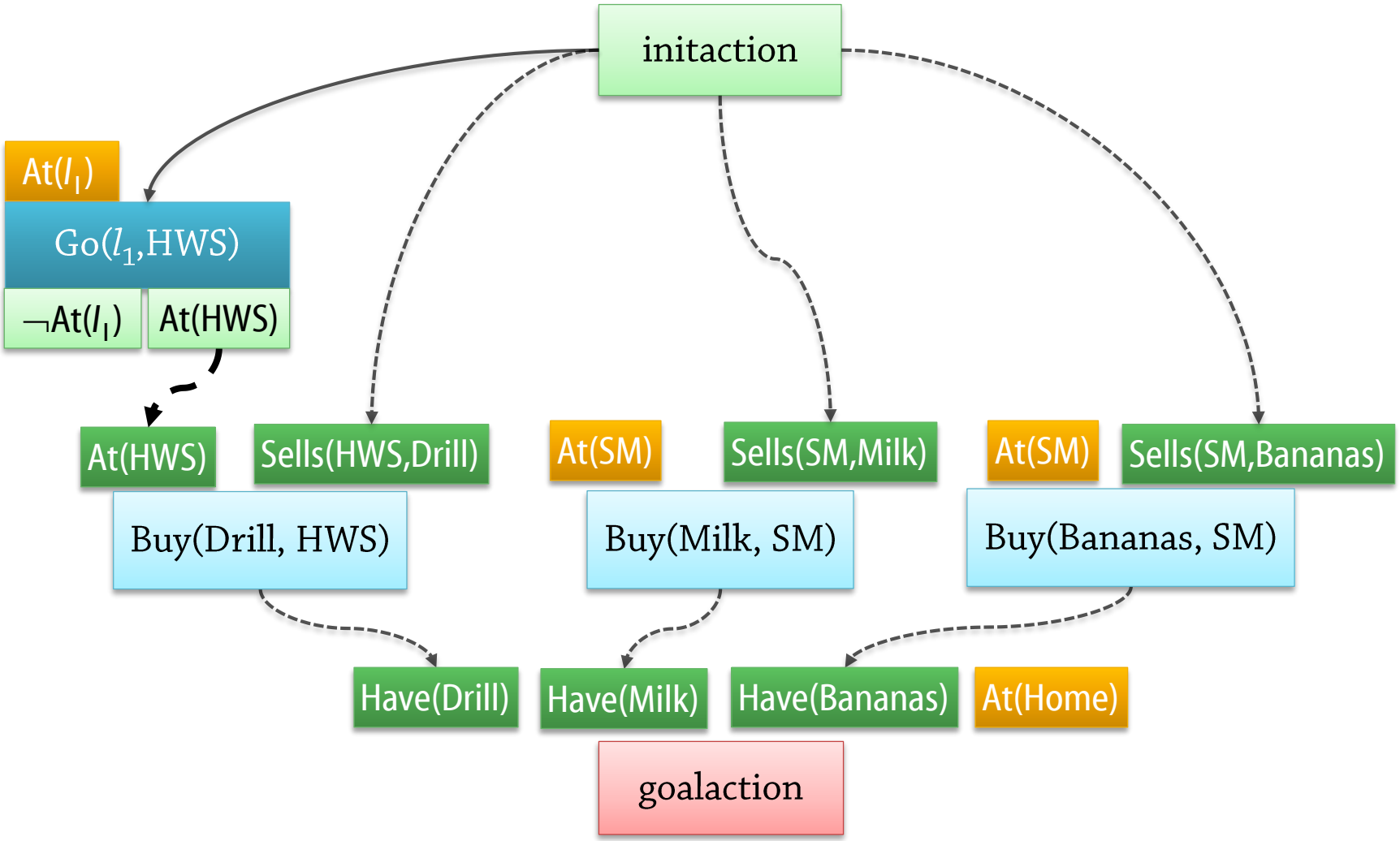
If you do this, SAY SO EXPLICITLY!



Example (continued)

jonkv@ida

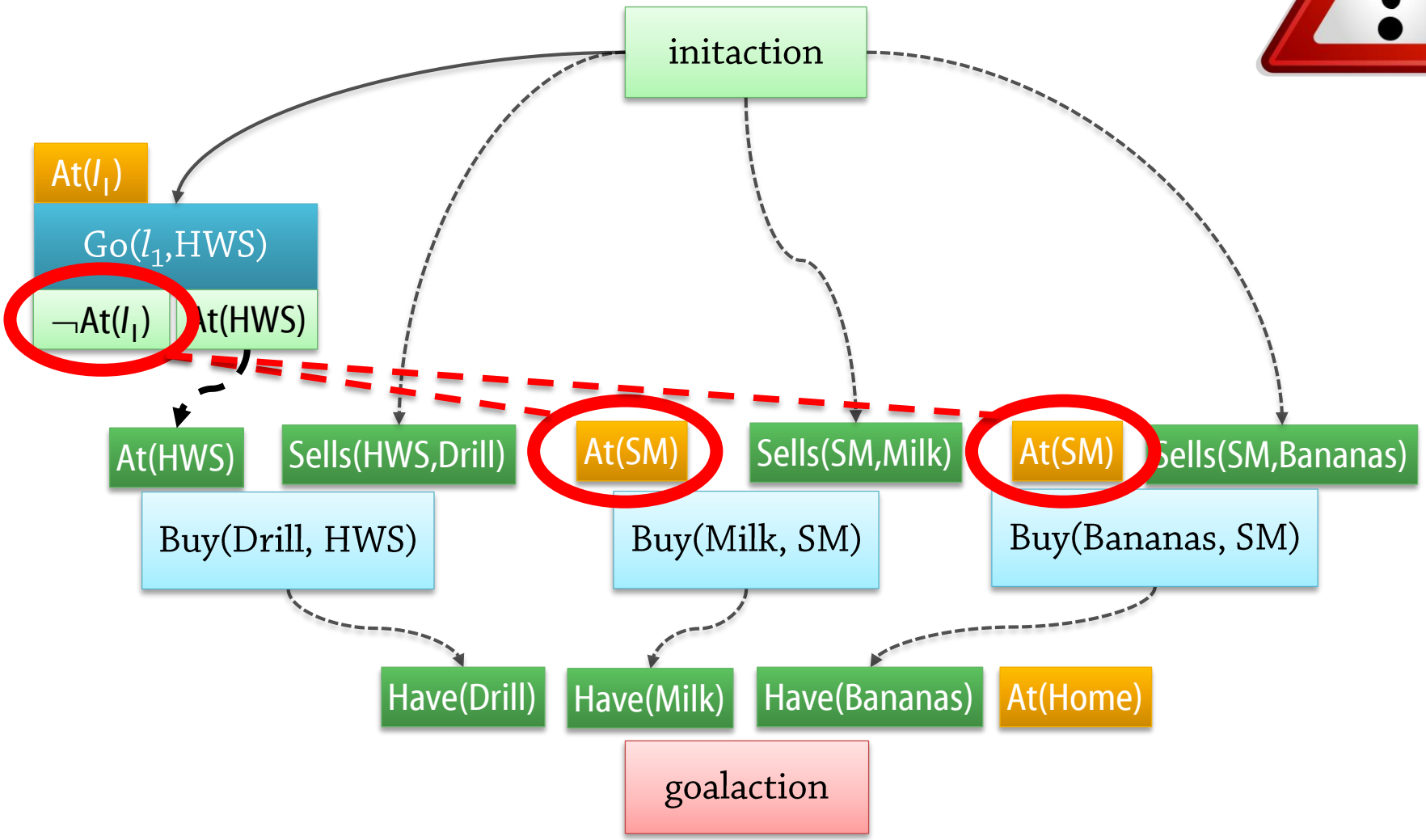
- To establish At(HWS): **Must** go there from **somewhere**



Example (continued)

jonkv@ida

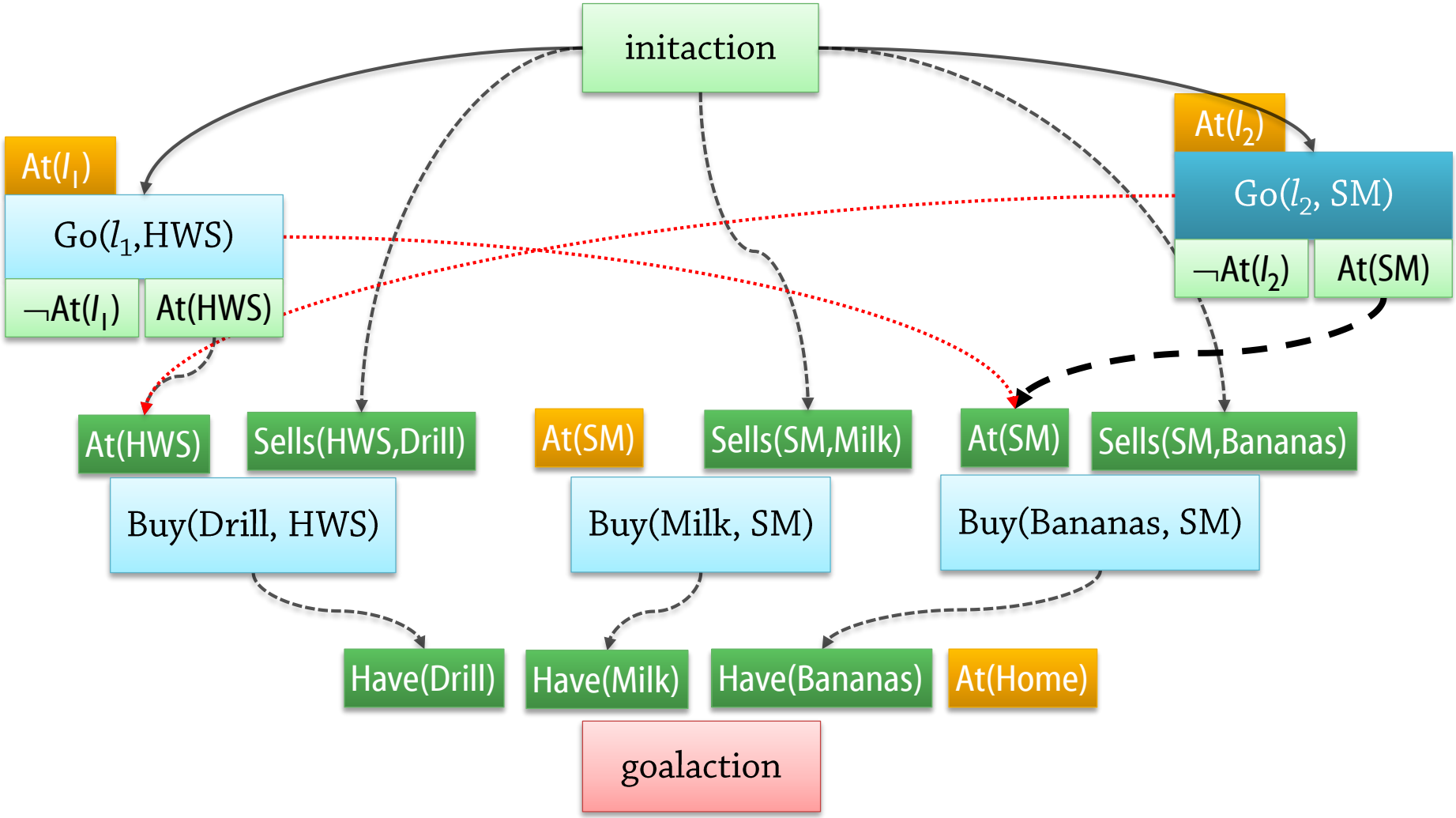
- Does $\neg at(l_1)$ threaten At(SM)?
  - No! Only a **causal link** to At(SM) can be threatened



Example (continued)

jonkv@ida

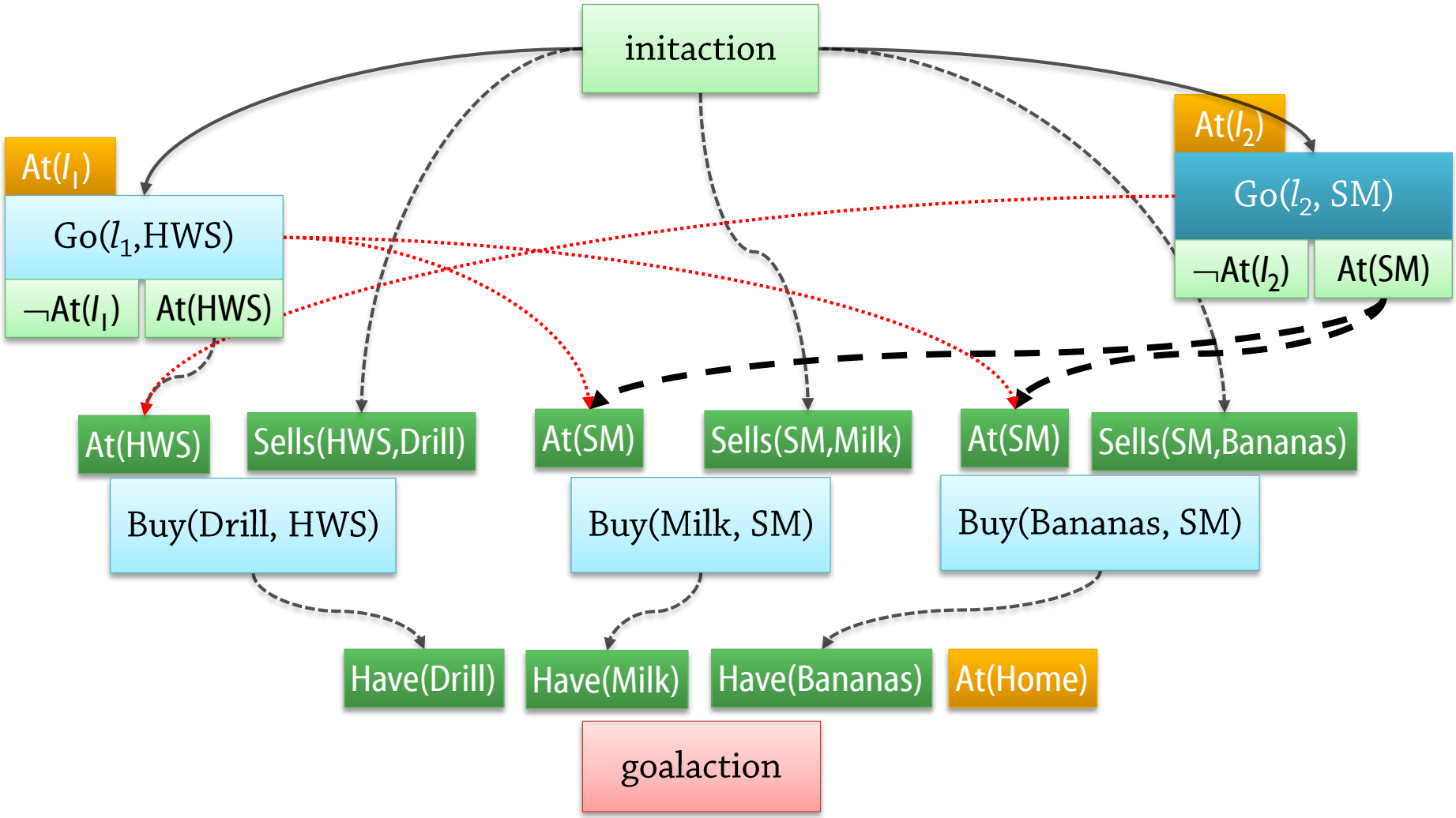
- To establish At(SM): Must go there from **somewhere**
  - Mutual **threats**…



Example (continued)

jonkv@ida

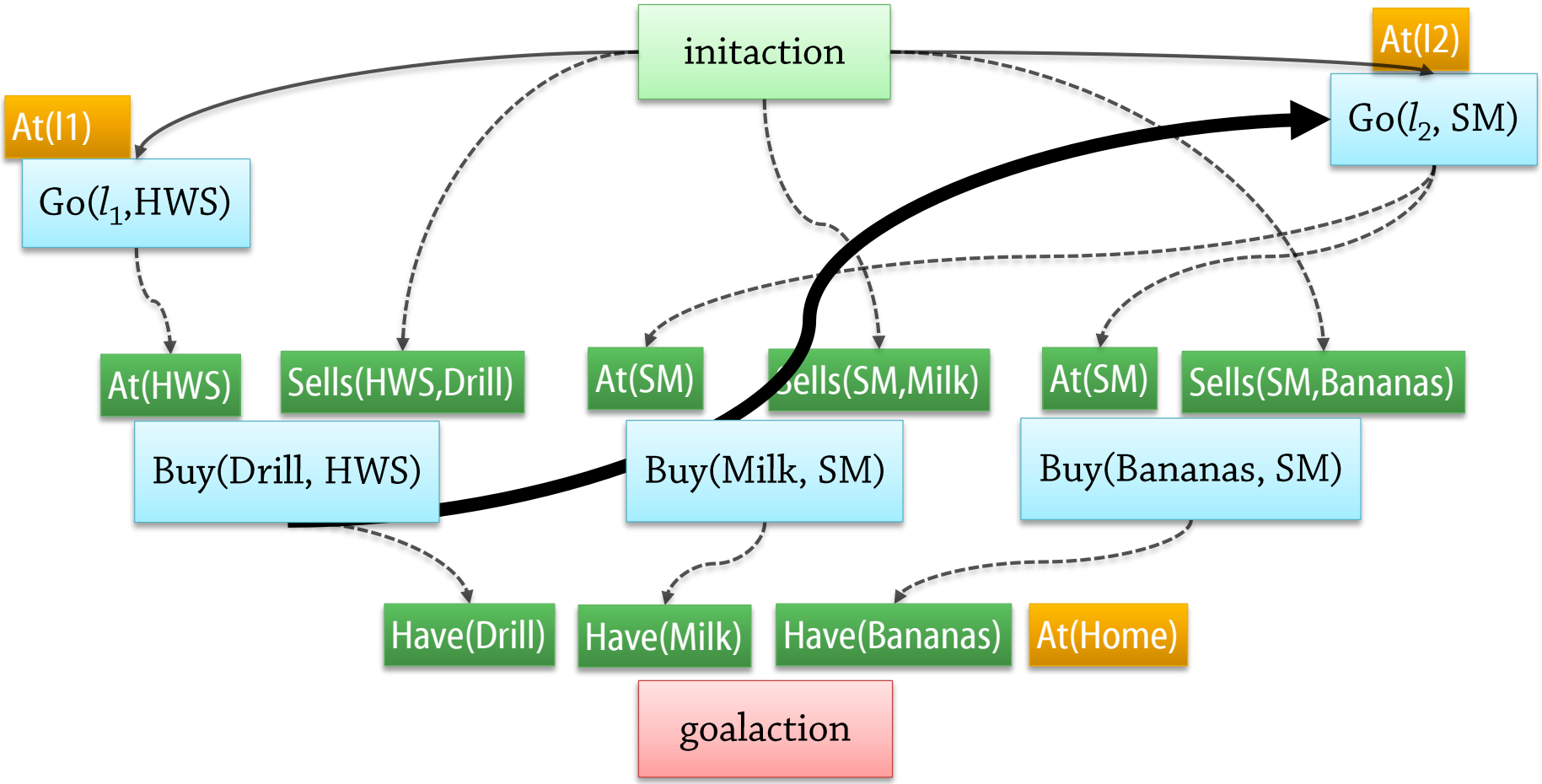
- Let's use the same action for **both** At(SM) preconditions…
  - More threats – could deal with them now or wait



Example (continued)

jonkv@ida

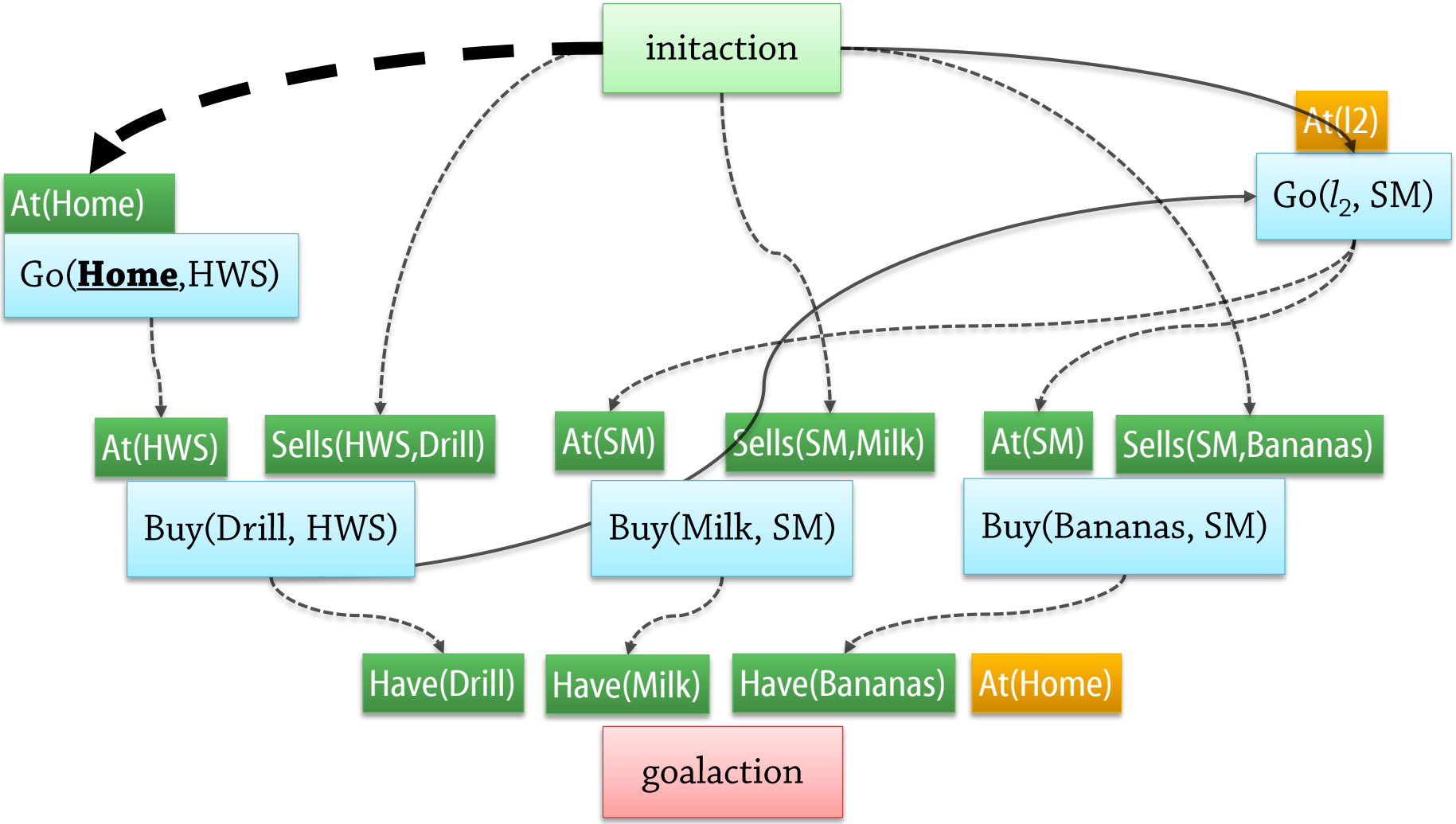
- **Nondet. choice**: how to resolve the threat to At(HWS)?
  - Our choice: make the "requirer" Buy(Drill) precede the "threatener" Go(l2, SM)
  - Also happens to resolve the other two threats
    - "Threatener" Go(l1, HWS) before "achiever" Go(l2, SM)



Example (continued)

jonkv@ida

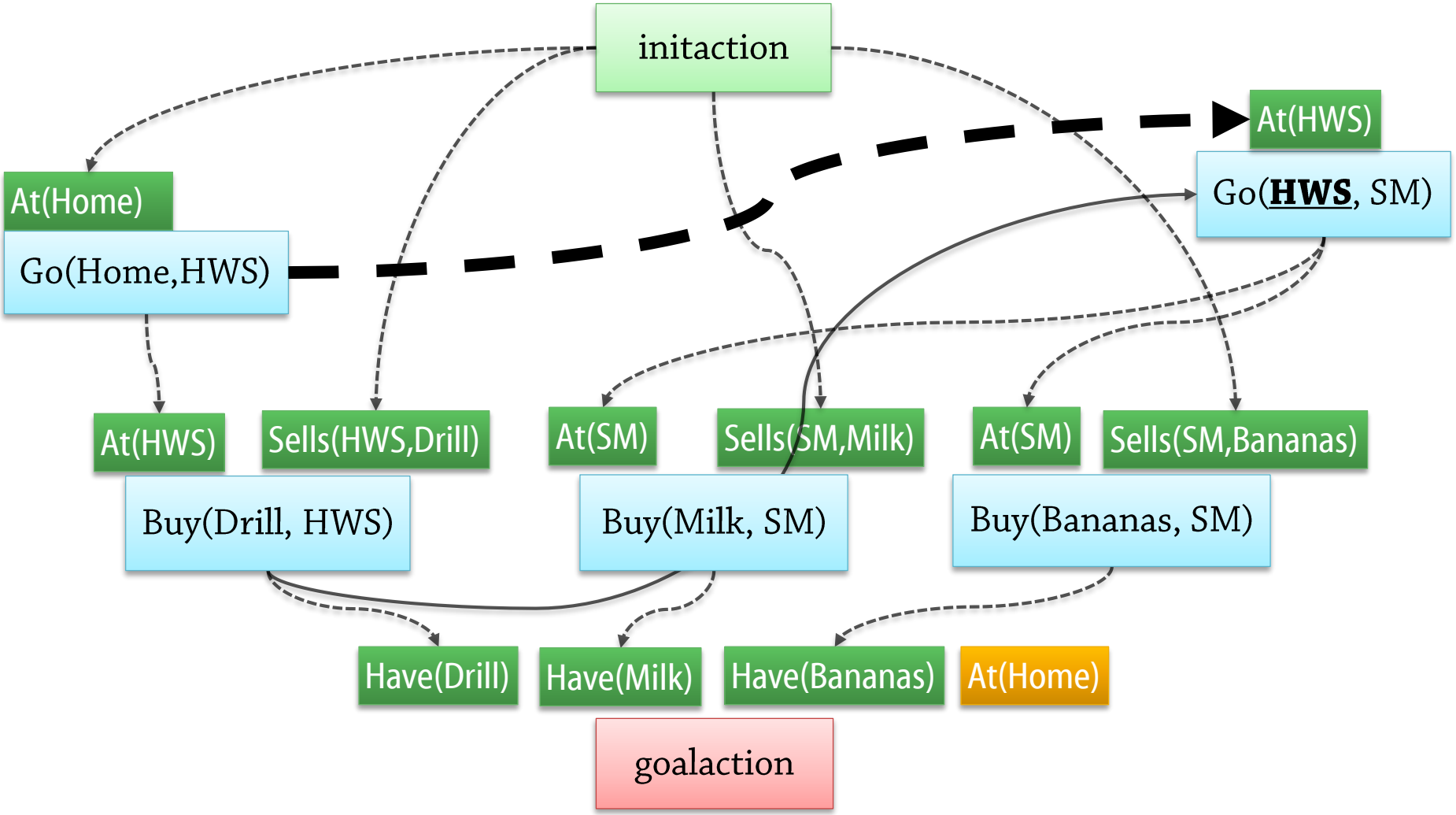
- **Nondet. choice**: how to establish At(l_1)?
  - We'll do it from initaction, with l_1=Home



Example (continued)

jonkv@ida

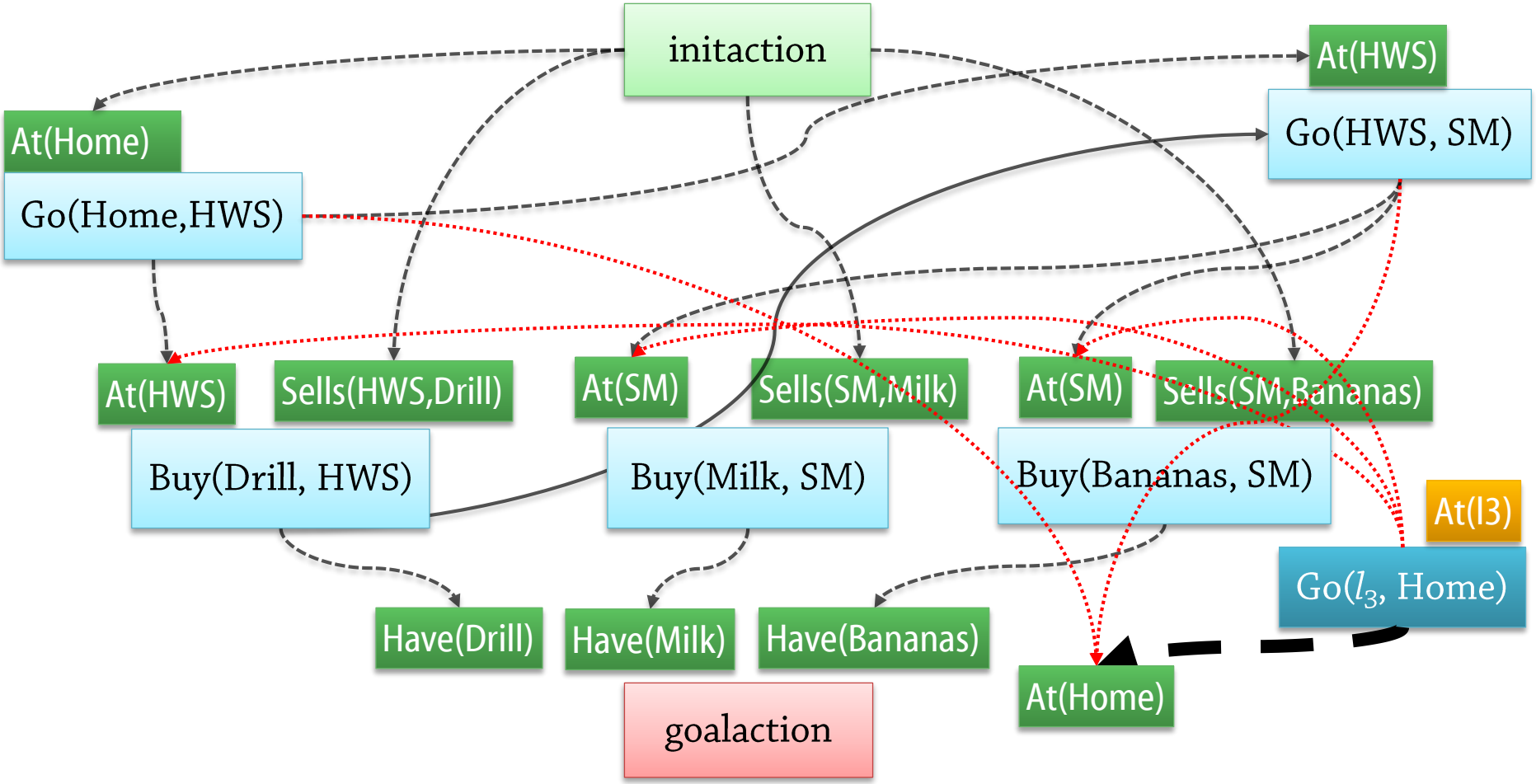
- **Nondeterministic choice**: how to establish At(l_2)?
  - We'll do it from Go(Home,HWS), with l_2= HWS



Example (continued)

jonkv@ida

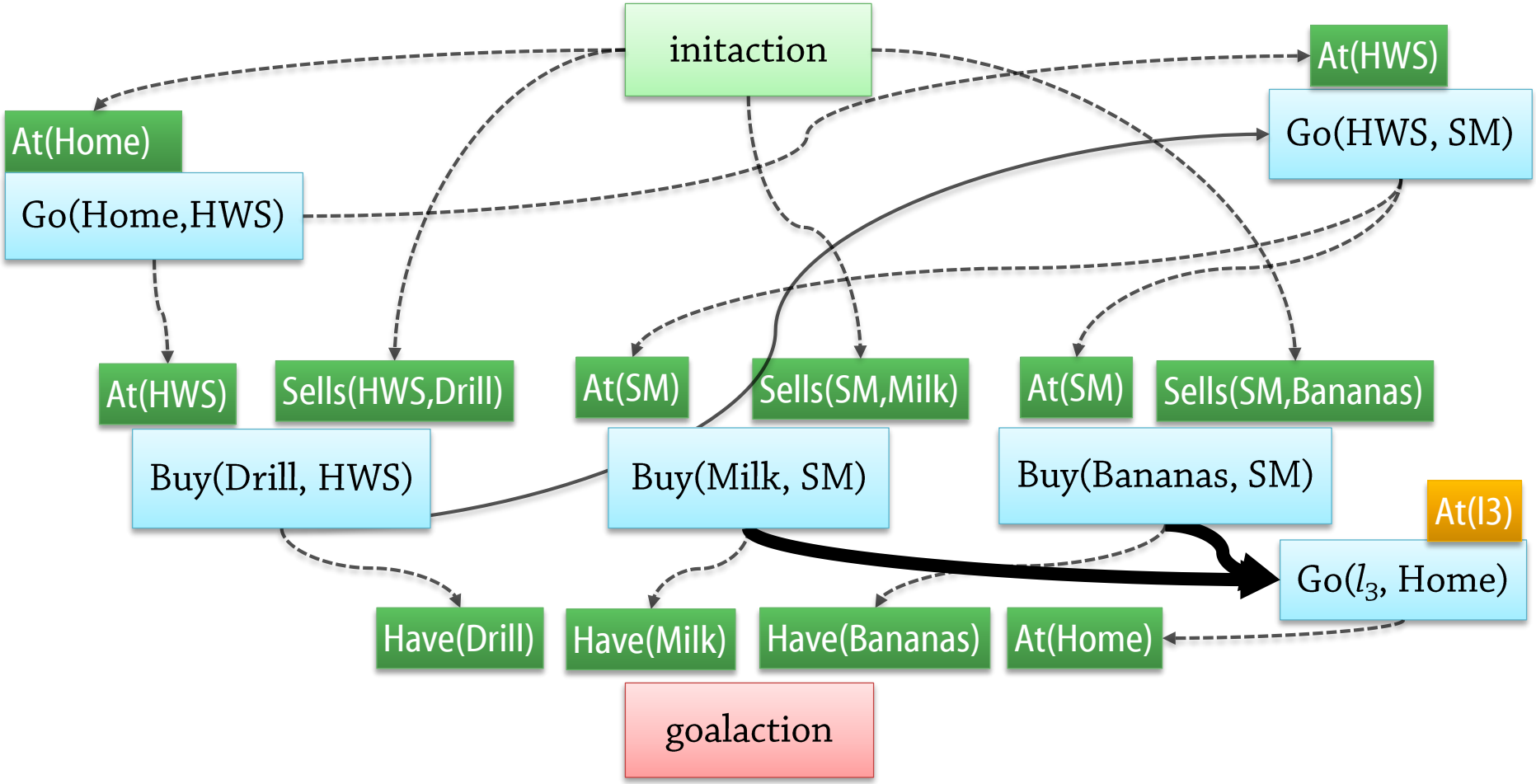
- The only possible way to establish At(Home) for goalaction
  - This creates several new threats



Example (continued)

jonkv@ida

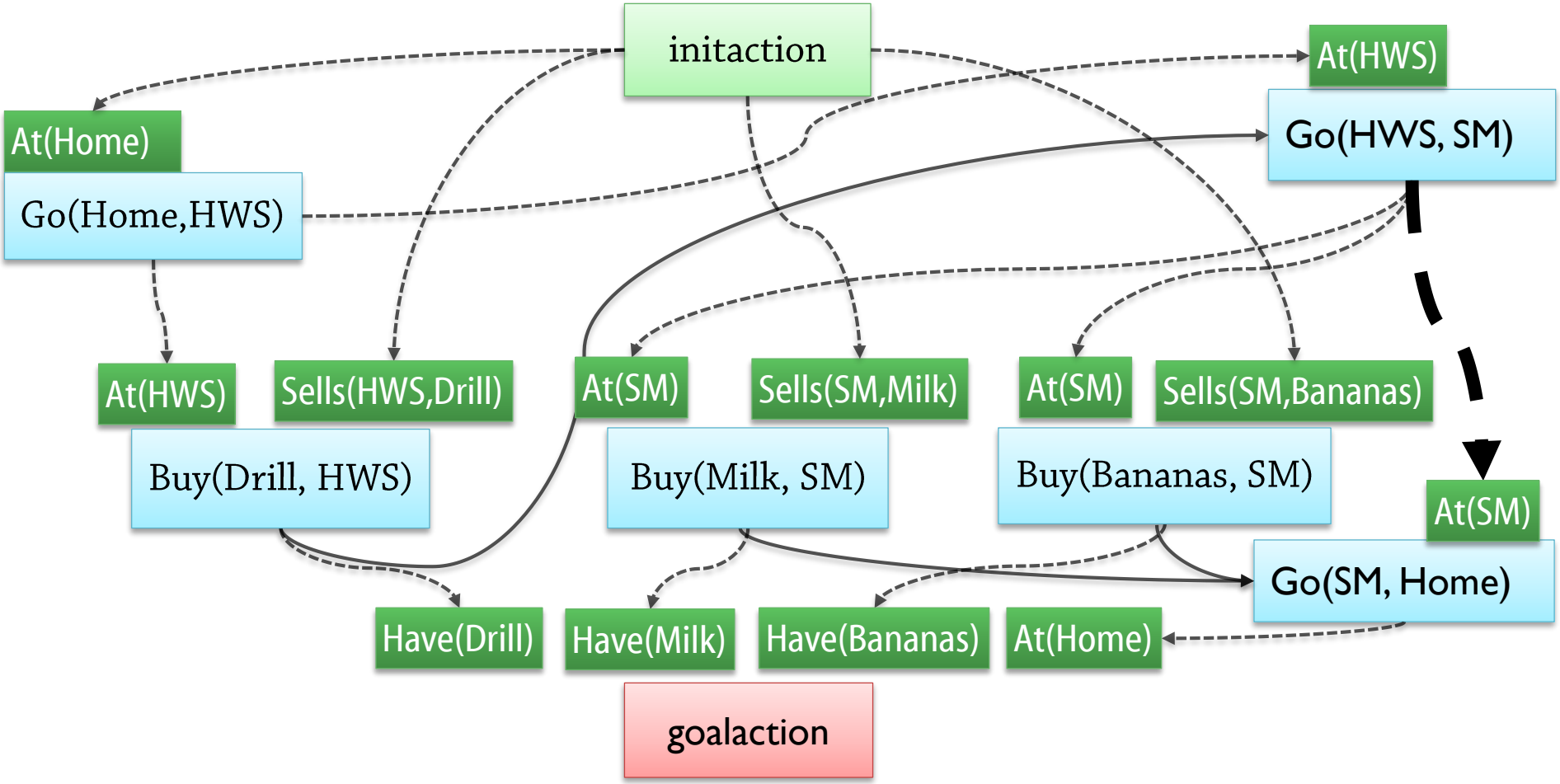
- To remove the threats to At(SM) and At(HWS):
  - Make go(HWS,SM) and go(Home,HWS) precede Go(l_3,Home)
  - This also removes the other threats

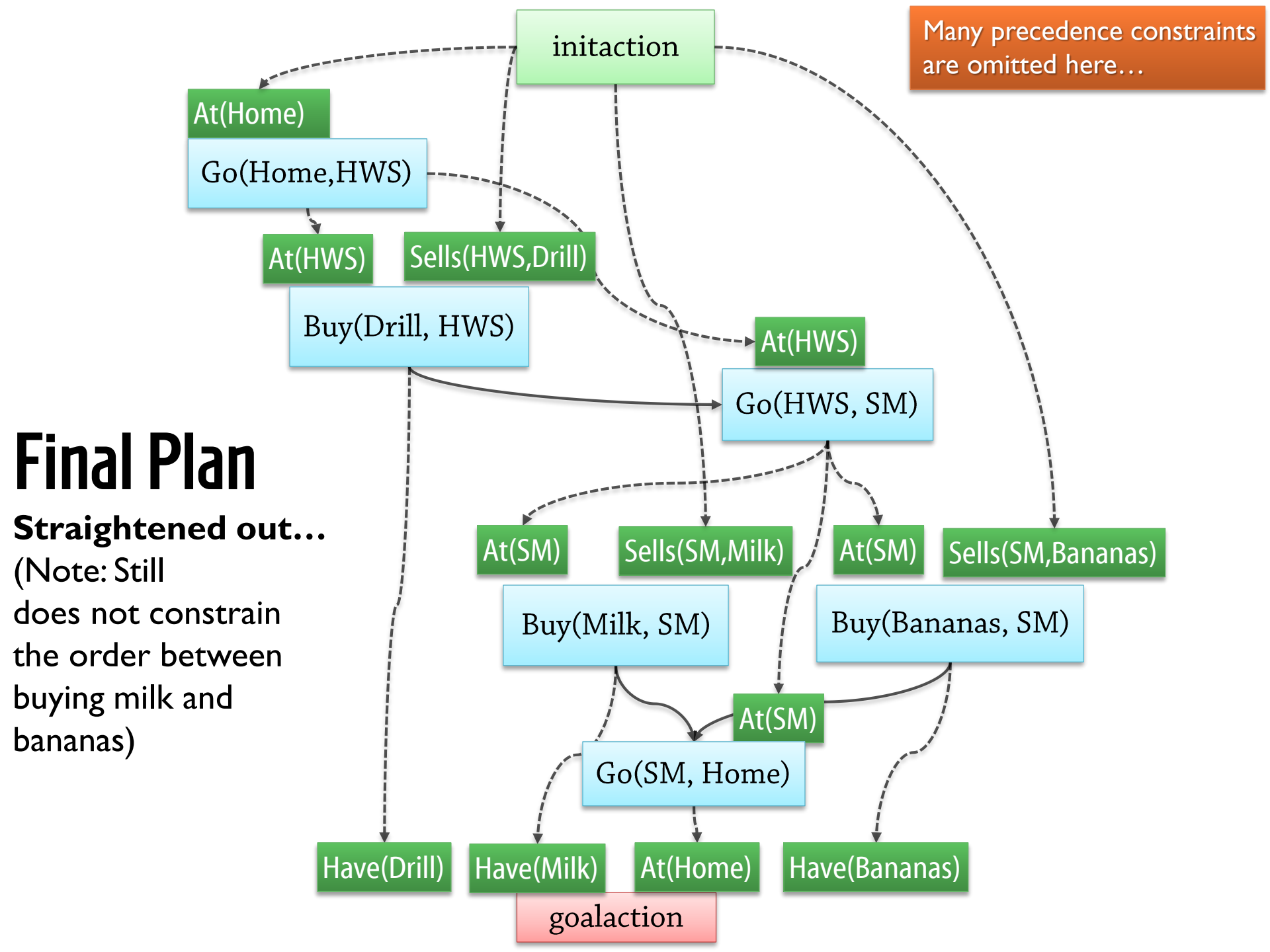


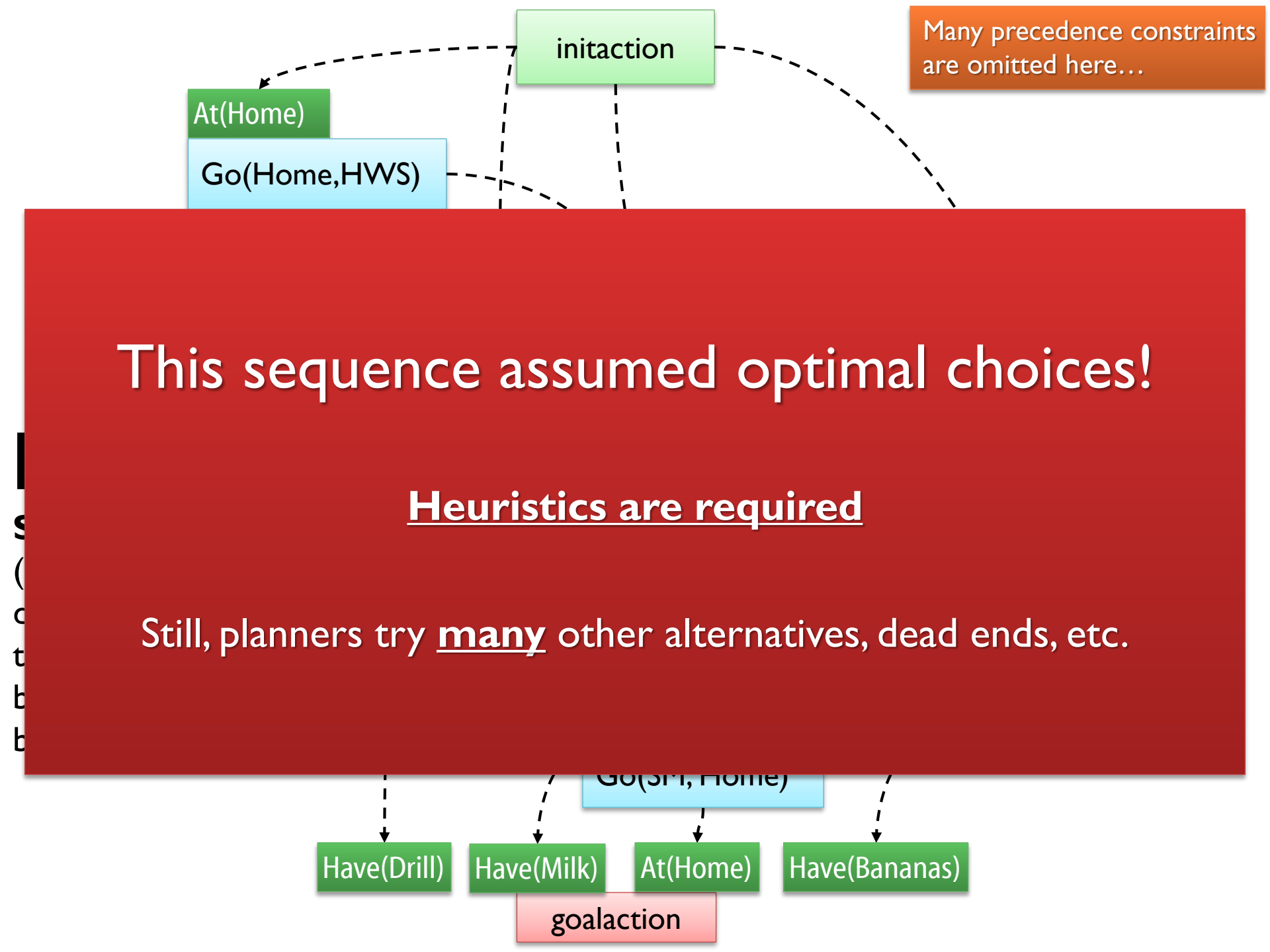
Final Plan

jonkv@ida

- Establish At(l_3) with l_3=SM




initaction
Many precedence constraints are omitted here…
At(Home)
Go(Home,HWS)
At(HWS)
Sells(HWS,Drill)
Buy(Drill, HWS)
At(HWS)
Go(HWS, SM)
Final Plan
Straightened out…
(Note: Still does not constrain the order between buying milk and bananas)
At(SM)
Sells(SM,Milk)
At(SM)
Sells(SM,Bananas)
Buy(Milk, SM)
Buy(Bananas, SM)
At(SM)
Go(SM, Home)
Have(Drill)
Have(Milk)
At(Home)
Have(Bananas)
goalaction


initaction
Many precedence constraints are omitted here…
At(Home)
Go(Home,HWS)
This sequence assumed optimal choices!
Heuristics are required
Still, planners try many other alternatives, dead ends, etc.
Have(Drill)
Have(Milk)
At(Home)
Have(Bananas)
goalaction

Heuristics:
A few brief examples

POCL planning

jonkv@ida

- **POCL planning – one possible formulation** (sound/complete):
  - planner(initial, goal):

```
open ← { Make-initial-plan(initial, goal) }
while open ≠ ∅:
    select a plan π in open
    open ← open – { π }
    flaws ← OpenGoals(π) ∪ Threats(π)
    if flaws = ∅ then
        return π
    else
        select any flaw φ ∈ flaws
        resolvers ← FindResolvers(φ, π)  // May be the empty set!
        foreach r in resolvers:
            π' ← Refine(ρ, π) // Actually apply the resolver
            open ← open ∪ { π' }
endwhile
return failure
```

Plan Selection

jonkv@ida

- Examples of (*generally non-admissible*) **plan selection** heuristics:
  - Count the **number of flaws** (see SNLP, UCPOP)
    - $h_f(\pi) = |\{\text{flaws in } \pi\}|$
  - Count the **number of open goals**
    - $h_{oc}(\pi) = |\{\text{open goals in } \pi\}|$
  - Adapt the **additive heuristic** (VHPOP)

$$h_{add}(\pi) = \sum_{\{open\ goals \xrightarrow{q} a_i\ in\ \pi\}} h_{add}(q)$$

  - Can be modified to address **reuse of actions** already in the plan (VHPOP):

$$h^r_{add}(\pi) = \sum_{open\ goals \xrightarrow{q} a_i\ in\ \pi} \begin{cases} 0 & \text{if there is an action } a_j \in \pi \text{ that achieves q and could be before } a_i \\ h_{add}(q) & \text{otherwise} \end{cases}$$

  - Adapt **planning graphs** (RePOP)

Flaw Selection

jonkv@ida

- Examples of **flaw selection** heuristics:
  - Choose **threats** before open goals (SNLP, UCPOP)
    - LIFO: Prefer the threat (or open goal) that was added last
  - Defer some threats until later
    - DSep: Delay threats that can be resolved through variable bindings ("separation")
    - DUnf: Delay threats that can be resolved in more than one way
  - Prefer the flaw with the fewest refinement options
  - Prefer open goals that **must** be resolved against the initial state
    - Cannot be achieved by any operator in the domain
  - As in plan selection, estimate costs of resolving flaws
    - $h_{add}()$, …
  - …

Summary

jonkv@ida

- Partial-order planning **delays commitment** to action ordering
  - Lower branching factor
  - More efficient in some situations

- Many POP planners still **assume sequential execution**
  - The intention was to find plans quickly,
    not to find partially constrained plans