Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili

[DΣIM]

AI Planning

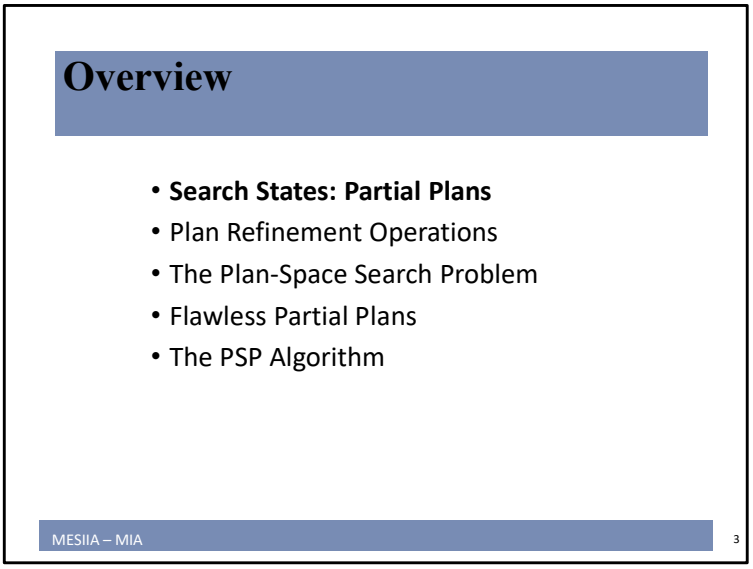
Hatem A. Rashwan

Plan-Space Search and Hierarchical Planning

Plan-Space Search

Plan-Space Planner (PSP)
Partial Plans

Partial-order planning is an approach to automated planning that maintains a partial ordering between actions and only commits ordering between actions when forced to that is, ordering of actions is partial. Also, this planning doesn't specify which action will come out first when two actions are processed. By contrast, **total-order planning** maintains a total ordering between all actions at every stage of planning. Given a problem in which some sequence of actions is required in order to achieve a goal, a **partial-order plan** specifies all actions that need to be taken but specifies an ordering between actions only where necessary.

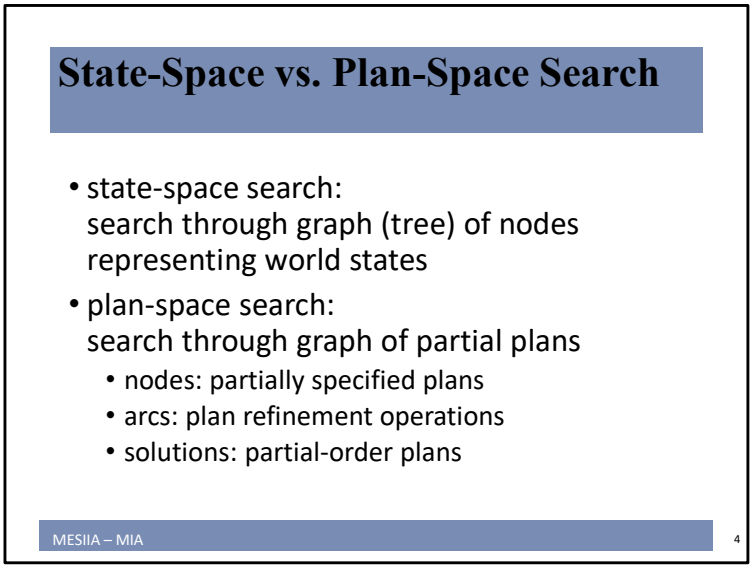


Overview

➢**Search States: Partial Plans**

- now: introducing a completely different search space with partial plans as search states

•Plan Refinement Operations
•The Plan-Space Search Problem
•Flawless Partial Plans
•The PSP Algorithm



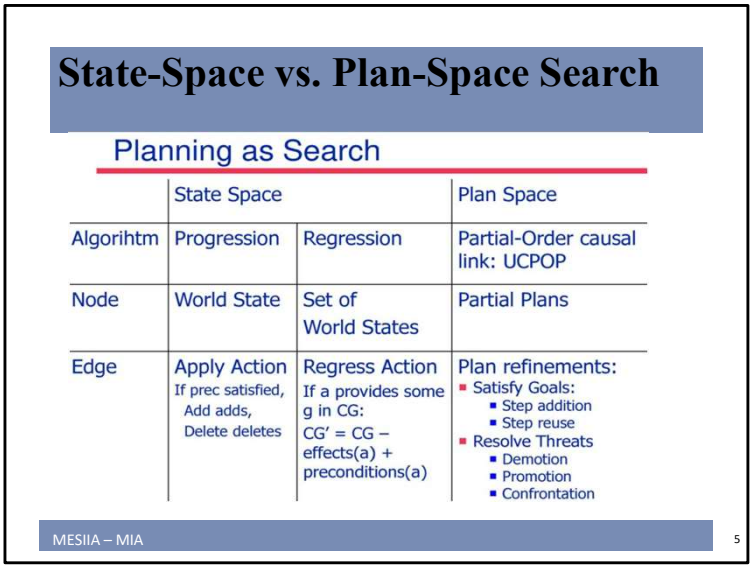
State-Space vs. Plan-Space Search

•state-space search: search through graph (tree) of nodes representing world states

- •search space directly corresponds to graph representation of state-transition system by generating the graph (tree) during the search.

•plan-space search: search through graph of partial plans

- **•nodes: partially specified plans**
- **•arcs: plan refinement operations**
  - •least commitment principle: do not add constraints to the plan that are not strictly needed
- **•solutions: partial-order plans**
  - •partial-order plan: set of actions + set of orderings; not necessarily total order
  - •state-space algorithms also maintain partial plan – but always in total order

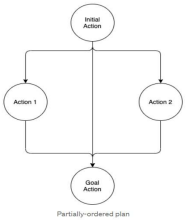


UCPOP: : soUnd, Complete, Partial Order Planner

Plane refinements is to add actions, ordering, causal links and bindings variables.

Partial Plans

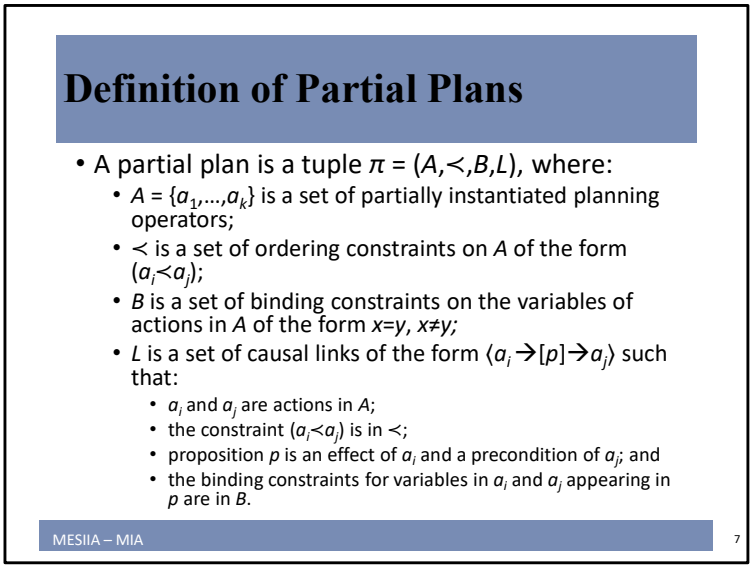
- plan: set of actions organized into some structure
- partial plan:
  - subset of the actions
  - subset of the organizational structure
    - temporal ordering of actions
    - rationale: what the action achieves in the plan
  - subset of variable bindings



Here is the informal definition of partial plans

Partial Plans

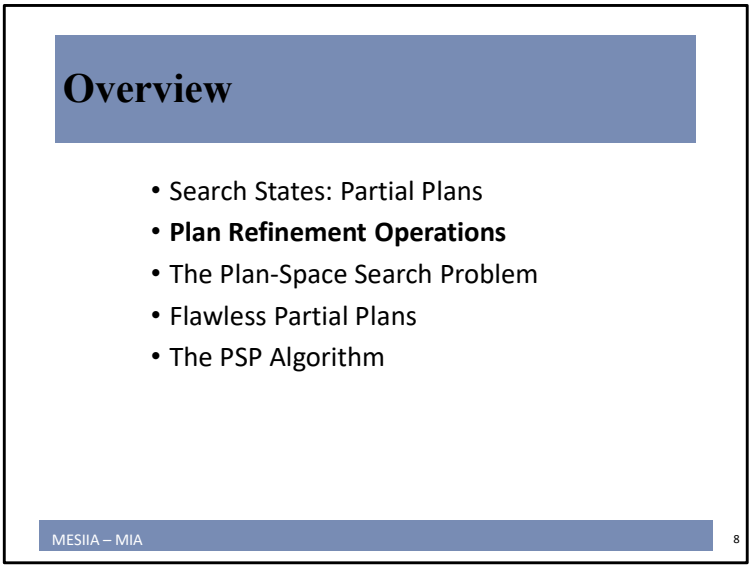
- **plan: set of actions organized into some structure**
  - organization e.g., sequence
- **partial plan:**
  - **subset of the actions**
  - **subset of the organizational structure**
    - **temporal ordering of actions**
    - **rationale: what the action achieves in the plan**
      - refers only to subset of actions
    - **subset of variable bindings** (variable binding (plural variable bindings) (programming) the association between a variable name (identifier) and its value.)
- plan refinement operators accordingly: add actions, add ordering constraints, add causal links, add variable bindings



Here is the formal definition of partial plans

Definition of Partial Plans

- **A partial plan is a tuple $\pi = (A,\prec,B,L)$, where:**
  - **$A = \{a_1,\ldots,a_k\}$ is a set of partially instantiated planning operators;**
  - **$\prec$ is a set of ordering constraints on A of the form $(a_i \prec a_j)$;**
  - **B is a set of binding constraints on the variables of actions in A of the form $x=y$, $x\neq y$, or $x \in D_x$;**
  - **L is a set of causal links of the form $\langle a_i -[p]\rightarrow a_j\rangle$ such that:**
    - **a_i and a_j are actions in A;**
    - **the constraint $(a_i \prec a_j)$ is in $\prec$;**
    - **proposition p is an effect of a_i and a precondition of a_j; and**
    - **the binding constraints for variables in a_i and a_j appearing in p are in B.**
- sub-gaols in a partial plan: preconditions without causal links
- different view: partial plan as set of (sequential) plans
  - those that meet the specified constraints and can be refined to a total order plan by adding constraints
- note: partial plans with two types of additional flexibility:
  - actions only partially ordered and
  - not all variables need to be instantiated



Overview

•Search States: Partial Plans

- just done: introducing a completely different search space with partial plans as search states

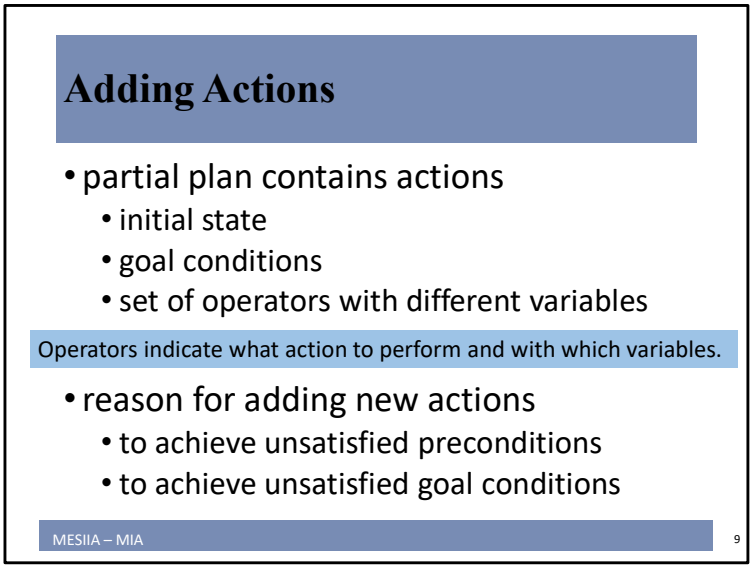
➢Plan Refinement Operations

- now: state transitions in the new search space – refining partial plans

•The Plan-Space Search Problem

•Flawless Partial Plans

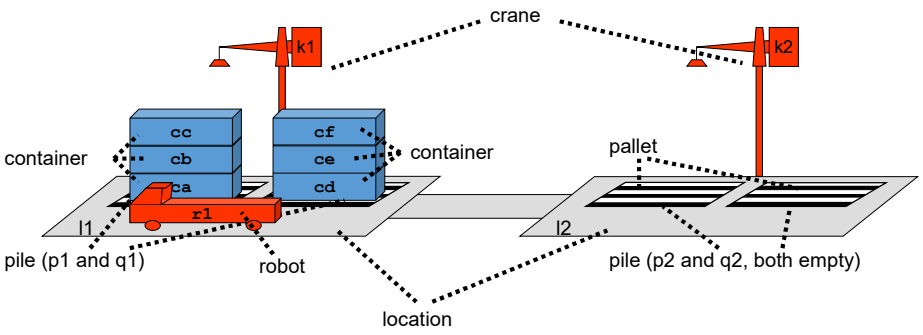
•The PSP Algorithm



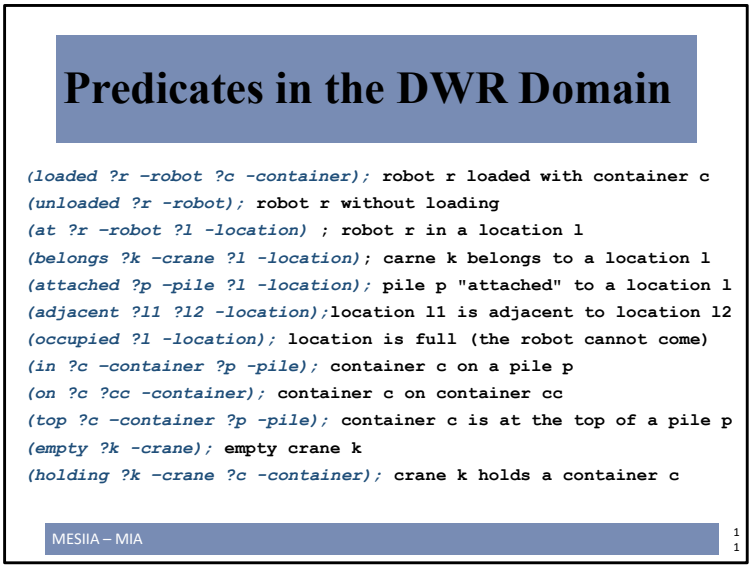
Adding Actions

- **partial plan contains actions**
  - **initial state**
  - **goal conditions**
  - can be represented as two (dummy) actions with only effects or preconditions
  - **set of operators with different variables**
- least commitment principle: introduce actions only for a reason
- **reason for adding new actions**
  - **to achieve unsatisfied preconditions**
  - **to achieve unsatisfied goal conditions**
- note: new actions can be added anywhere in the current partial plan

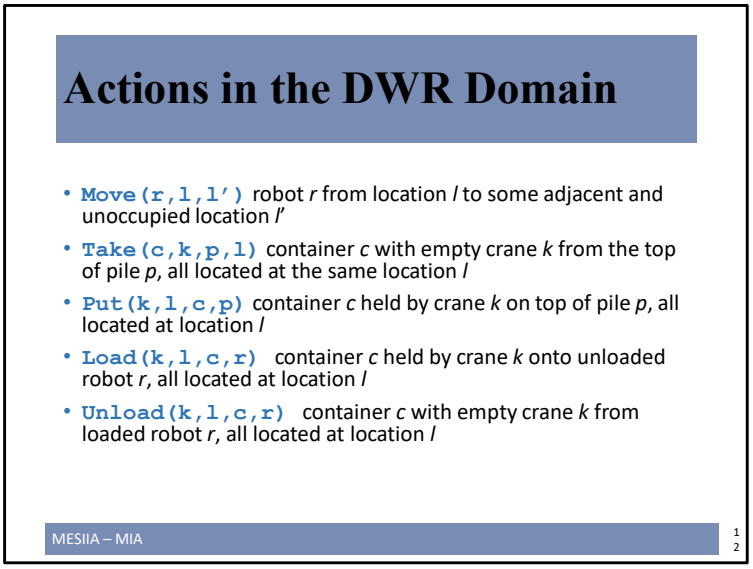
Dock-Worker Robots (DWR) Example State



Let's remember the Dock-Worker robots problem that we explained in the first lecture.



Predicates in the DWR Domain including 12 predicates



Actions in the DWR Domain

•move robot *r* from location *l* to some adjacent and unoccupied location *l'*

•take container *c* with empty crane *k* from the top of pile *p*, all located at the same location *l*

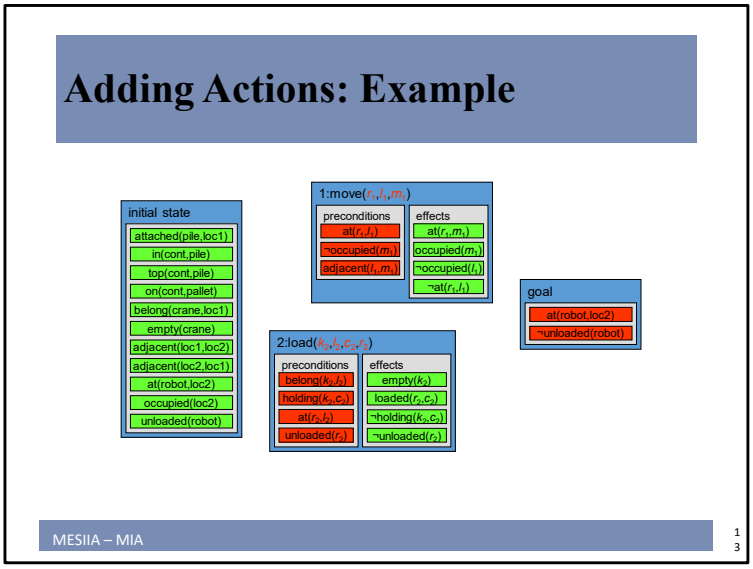
•put down container *c* held by crane *k* on top of pile *p*, all located at location *l*

•load container *c* held by crane *k* onto unloaded robot *r*, all located at location *l*

•unload container *c* with empty crane *k* from loaded robot *r*, all located at location *l*

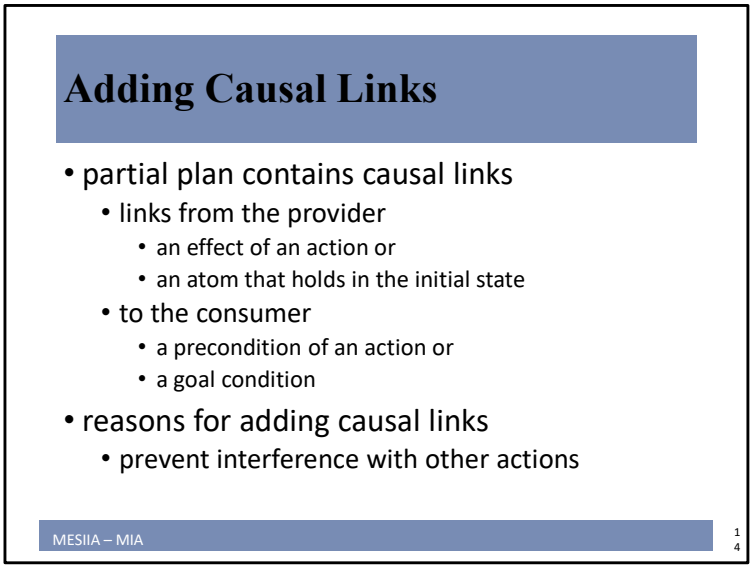
•formal specifications will follow when we have introduced a formal action description language

•problem: how to represent actions formally? first-order logic?



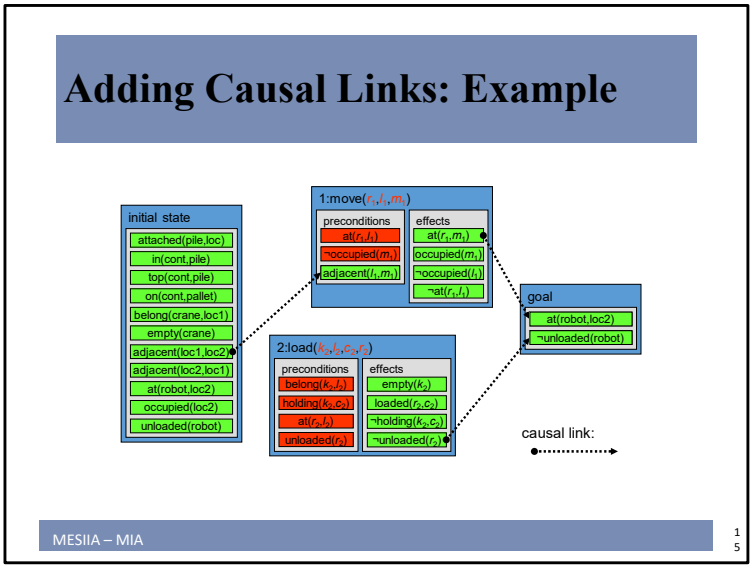
Adding Actions: Example

- empty plan:
  - initial state: all initially satisfied conditions (green)
  - goal: conditions that need to be satisfied (red)
- add operator: 1:move(r_1,l_1,m_1)
  - number (1) to provide unique reference to this operator instance
  - also used as variable index for unique variables
  - least commitment principle: choose values for variables only when necessary
- add operator:2:load(k_2,l_2,c_2,r_2)



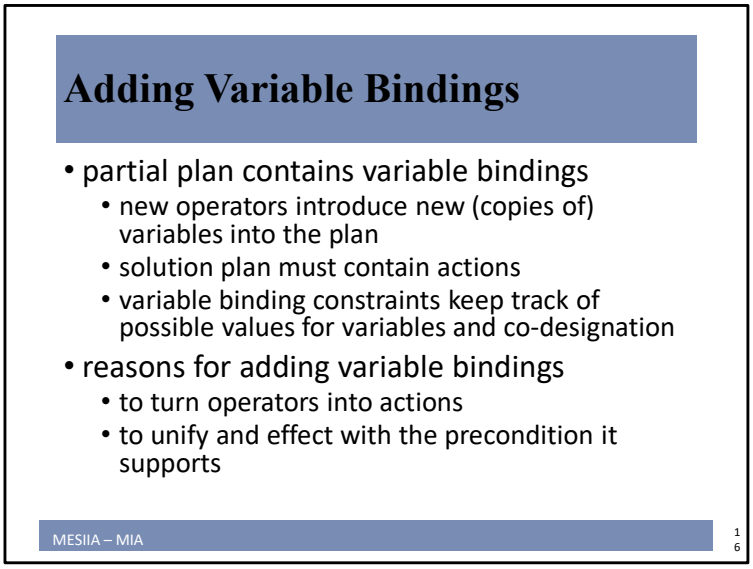
Adding Causal Links

- **partial plan contains causal links**
  - **links from the provider**
    - **an effect of an action or**
    - **an atom that holds in the initial state**
  - **to the consumer**
    - **a precondition of an action or**
    - **a goal condition**
  - causal link implies ordering constraint
    - but: provider need not come directly before consumer
- **reasons for adding causal links**
  - **prevent interference with other actions**
  - keeping track of rationale: any action inserted between provider and consumer must not overcome conditions in causal link
  - preconditions without a causal link pointing to them are open sub-gaols



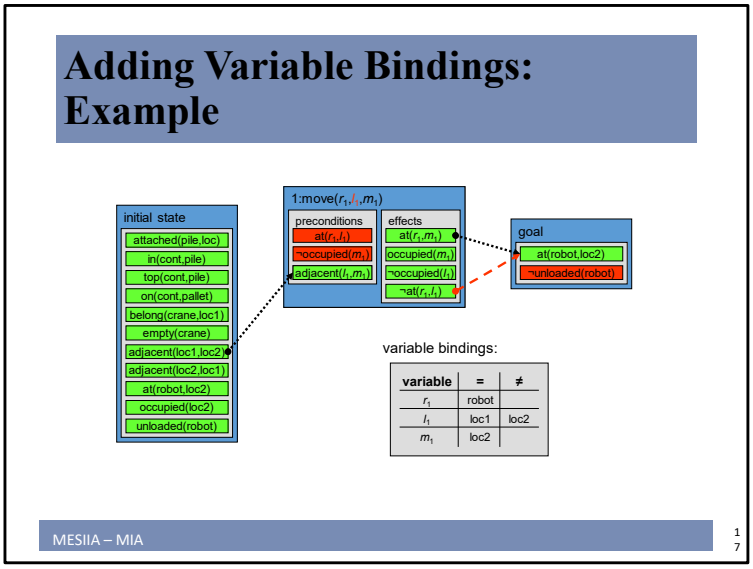
Adding Causal Links: Example

- add link from 1:move to goal
  - changes colour of goal to green – now satisfied
- add link from 2:load to goal
- add link from initial state to 1:move



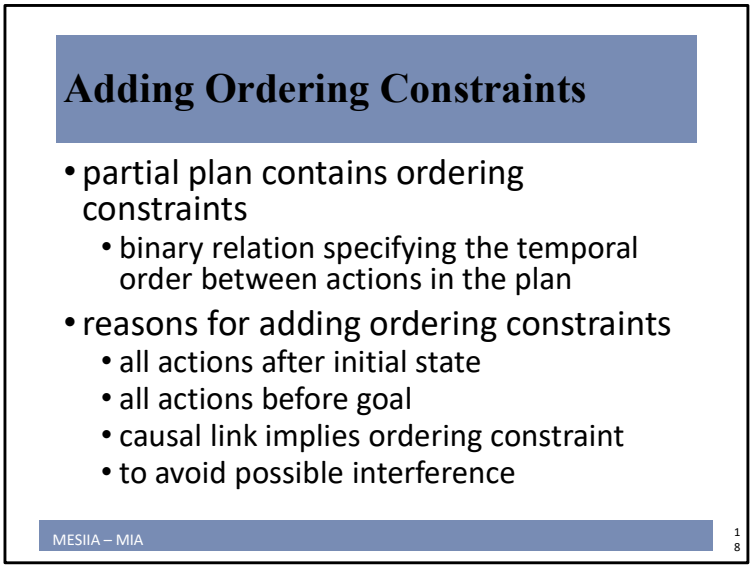
Adding Variable Bindings

- **partial plan contains variable bindings**
  - **new operators introduce new (copies of) variables into the plan**
    - each copy of an operator has its own set of variables that are different from variables in other operators instances
  - **solution plan must contain actions**
  - **variable binding constraints keep track of possible values for variables and co-designation**
  - convention (here): give number to operator instances to distinguish them; let variables have index of operator they belong to least commitment principle:
    - least commitment principle: add only necessary variable binding constraints
- **reasons for adding variable bindings**
  - **to turn operators into actions**
  - **to unify and effect with the precondition it supports**



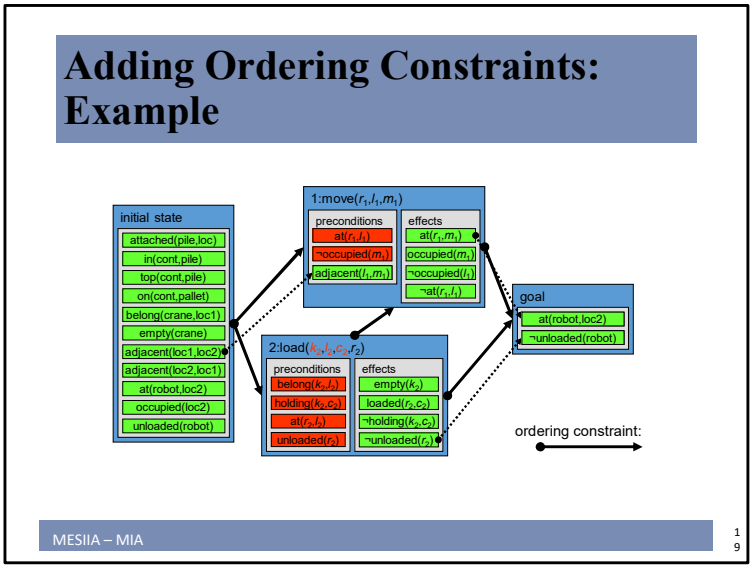
Adding Variable Bindings: Example

- bind variables due to causal link:
  - bind r_1 to robot
  - bind m_1 to loc2
  - note: variables in operator no longer red to indicate they are bound
- clobbering: move may also destroy goal condition
- introduce variable inequality: $l_1 \neq$ loc2
- clobbering now impossible
- introduce causal link from initial state
- bind l_1 to loc1
  - note consistency with inequality



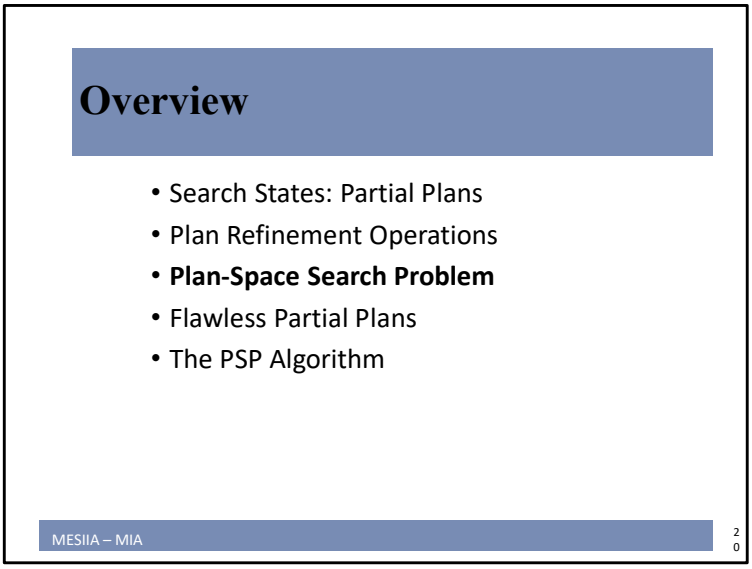
Adding Ordering Constraints

- **partial plan contains ordering constraints**
  - **binary relation specifying the temporal order between actions in the plan**
  - temporal relation: qualitative, not quantitative (at this stage)
- **reasons for adding ordering constraints**
  - **all actions after initial state**
  - **all actions before goal**
  - **causal link implies ordering constraint**
  - **to avoid possible interference**
    - interference can be avoided by ordering the potentially interfering action before the provider or after the consumer of a causal link
    - least commitment principle: introduce ordering constraints only if necessary
- result: solution plan not necessarily totally ordered



Adding Ordering Constraints: Example

- ordering constraints
  - due to causal links
  - also: all actions before goal
- ordering: all actions after initial state
- orderings may occur between actions



Overview

•Search States: Partial Plans

•Plan Refinement Operations

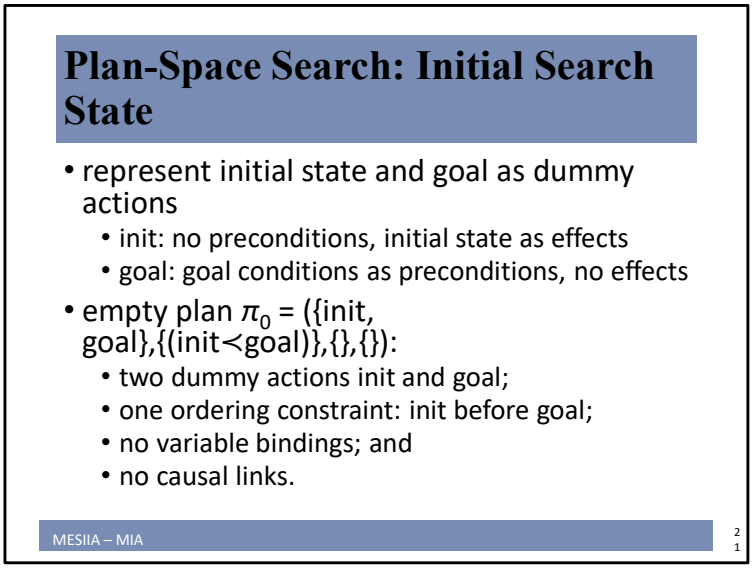
- just done: state transitions in the new search space – refining partial plans

➢**The Plan-Space Search Problem**

- now: definition of the plan-space search problem and solutions

•Flawless Partial Plans

•The PSP Algorithm



Plan-Space Search: Initial Search State

•problem: plan space representation does not maintain states, but need to give initial state and goal description

•represent initial state and goal as dummy actions

•init: no preconditions, initial state as effects

•goal: goal conditions as preconditions, no effects

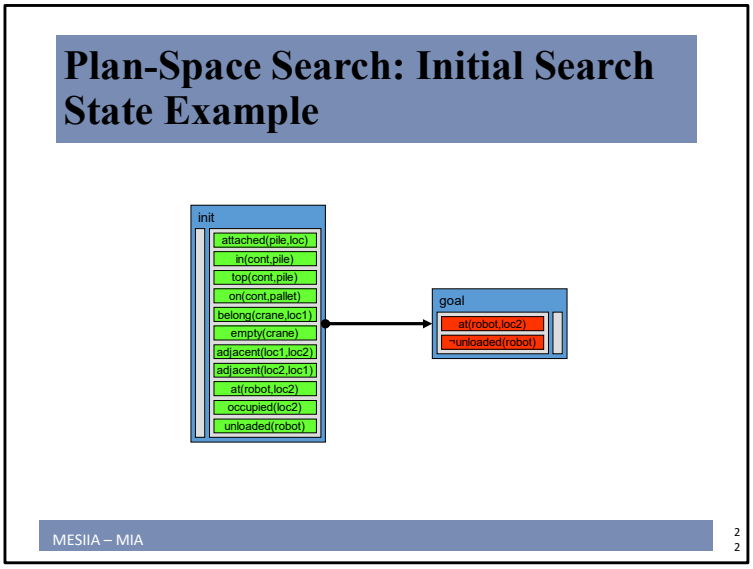
•empty plan π_0 = ({init, goal},{(init≺goal)},{},{}):

•two dummy actions init and goal;

•one ordering constraint: init before goal;

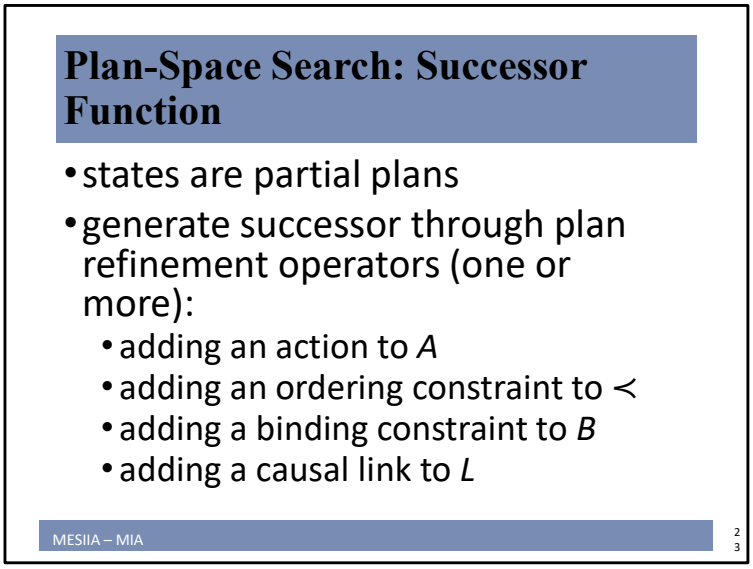
•no variable bindings; and

•no causal links.



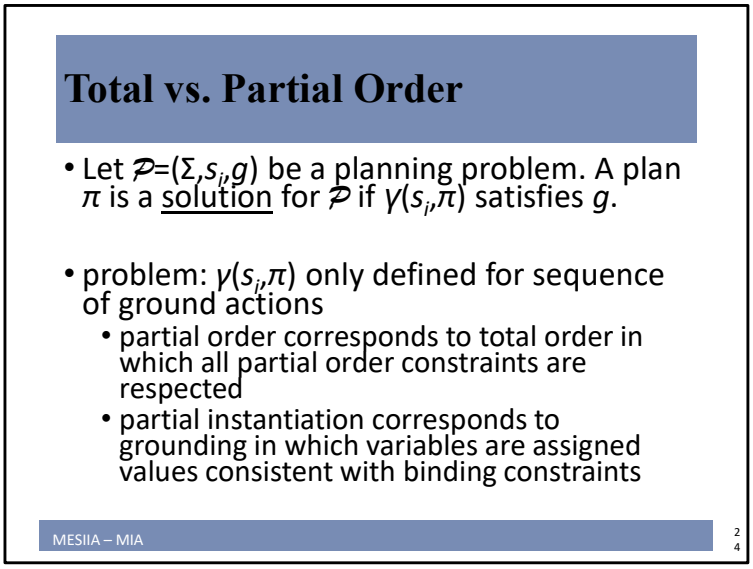
Plan-Space Search: Initial Search State Example

•note empty box for preconditions in init and empty box for effects in goal



Plan-Space Search: Successor Function

- **states are partial plans**
- **generate successor through plan refinement operators (one or more):**
  - more required to keep partial plans consistent, e.g., adding a causal link implies adding an ordering constraint
  - **adding an action to *A***
  - **adding an ordering constraint to ≺**
  - **adding a binding constraint to *B***
  - **adding a causal link to *L***
- successors must be consistent: constraints in a partial plan must be satisfiable
- plan-space planning decouple two sub-problems:
  - which actions need to be performed
  - how to organize these actions
- partial plan as set of plans: refinement operation reduces the set to smaller subset
- next: to define planning as plan-space search problem: need to define goal state



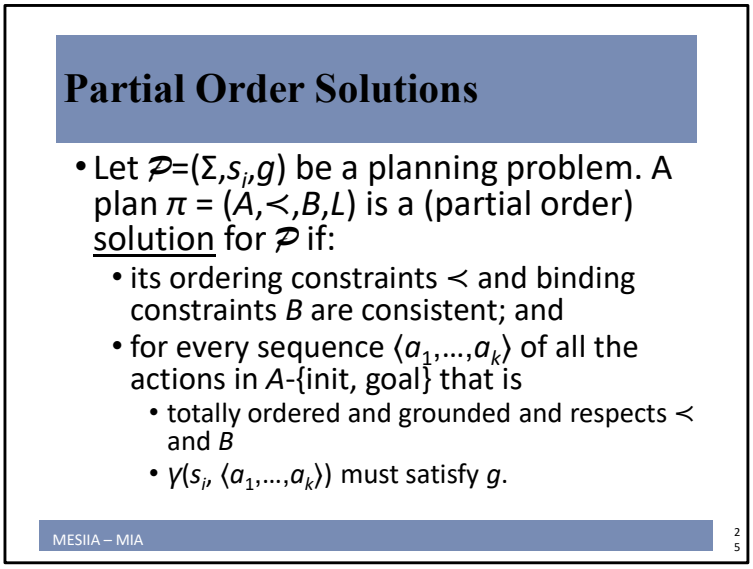
Total vs. Partial Order

•Let $\mathcal{P}=(\Sigma,s_i,g)$ be a planning problem. A plan π is a solution for $\mathcal{P}$ if $\gamma(s_i,\pi)$ satisfies g.

- •solution defined for state transition system

•problem: $\gamma(s_i,\pi)$ only defined for sequence of ground actions

- **•partial order corresponds to total order in which all partial order constraints are respected**
  - •partial ordering is consistent iff it is free of loops
  - •note: there may be an exponential number of total ordering consistent with a given partial ordering
- **•partial instantiation corresponds to grounding in which variables are assigned values consistent with binding constraints**
  - •note: exponential combinatorics of assigning values to variables



Partial Order Solutions

•Let $\mathcal{P}$=(Σ,s_i,g) be a planning problem. A plan π = (A,≺,B,L) is a (partial order) solution for $\mathcal{P}$ if:

•its ordering constraints ≺ and binding constraints B are consistent; and

•for every sequence ⟨a_1,…,a_k⟩ of all the actions in A-{init, goal} that is

•totally ordered and grounded and respects ≺ and B

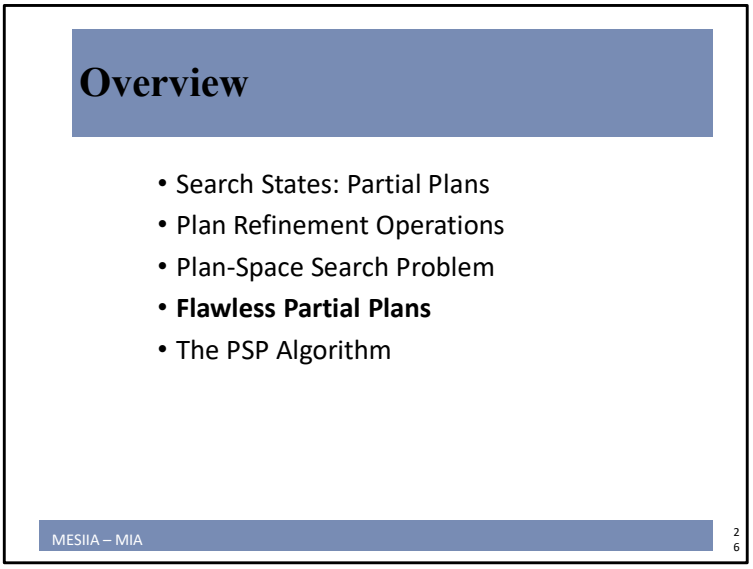
•$\gamma(s_i, \langle a_1,\ldots,a_k\rangle)$ must satisfy g.

•note: causal links do not play a role in the definition of a solution

•with exponential number of sequences to check, definition is not very useful (as computational procedure for goal test)

•idea: use causal links to verify that every precondition of every action is supported by some other action

•problem: condition not strong enough



Overview

•Search States: Partial Plans

•Plan Refinement Operations

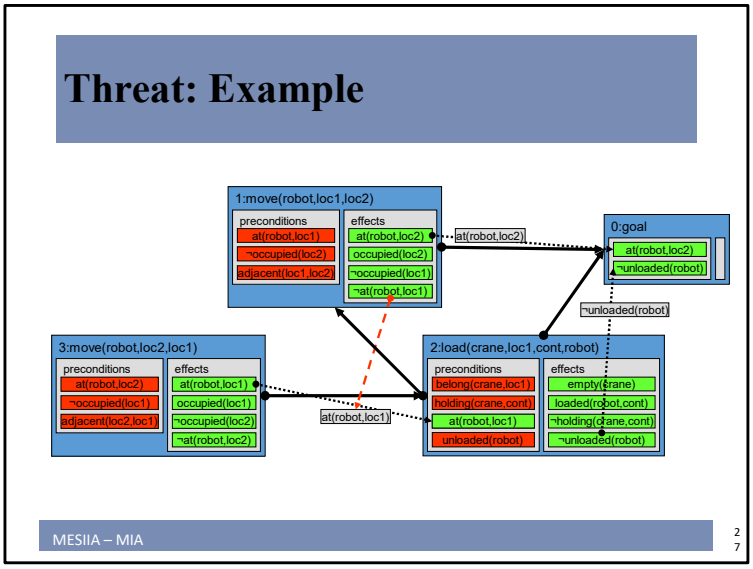
•The Plan-Space Search Problem

- just done: definition of the plan-space search problem and solutions (without goal test)

➢Flawless Partial Plans

- now: the goal test that completes the search problem

•The PSP Algorithm



Threat: Example

•start with partial plan from previous example (grounded; initial state not shown due to limited space on slide)

•introduce new 3:move action to achieve at(robot,loc1) precondition of 2:load action

- •note: still many unachieved preconditions – not a solution yet

•add causal link to maintain rationale

•add ordering to be consistent with causal link

•new: label causal link with condition it protects

•threat: effect of 1:move is negation of condition protected by causal link

- •if 1:move is executed between 3:move and 2:load the plan is no longer valid

•possible solution: additional ordering constraint

Threats

- An action a_k in a partial plan π = $(A,\prec,B,L)$ is a threat to a causal link $\langle a_i -[p]\rightarrow a_j\rangle$ iff:
  - a_k has an effect $\neg q$ that is possibly inconsistent with p, i.e. q and p are unifiable;
  - the ordering constraints $(a_i \prec a_k)$ and $(a_k \prec a_j)$ are consistent with $\prec$; and
  - the binding constraints for the unification of q and p are consistent with B.

Threats

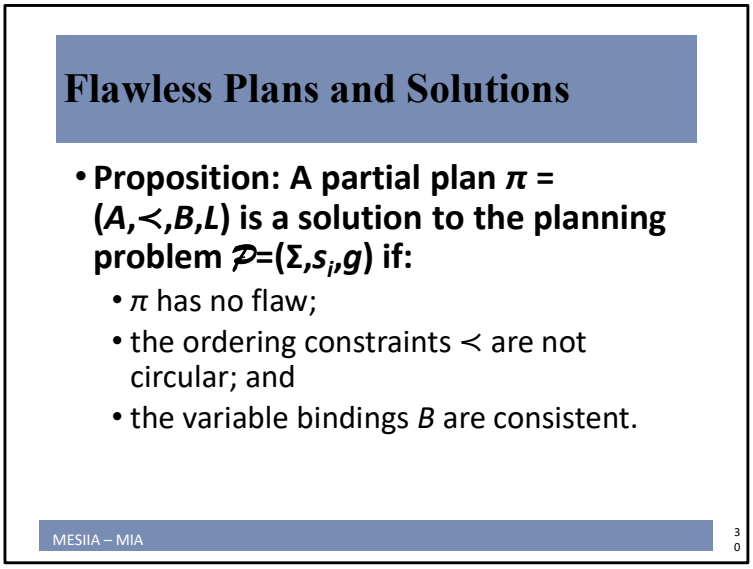
•An action a_k in a partial plan $\pi = (A,\prec,B,L)$ is a threat to a causal link $\langle a_i -[p]\rightarrow a_j\rangle$ iff:

- **a_k has an effect $\neg q$ that is possibly inconsistent with p, i.e. q and p are unifiable;**
- **the ordering constraints $(a_i \prec a_k)$ and $(a_k \prec a_j)$ are consistent with $\prec$; and**
- **the binding constraints for the unification of q and p are consistent with B.**

Flaws

- A flaw in a plan $\pi = (A, \prec, B, L)$ is either:
  - an unsatisfied sub-goal, i.e. a precondition of an action in A without a causal link that supports it; or
  - a threat, i.e. an action that may interfere with a causal link.

Flaws
•**A flaw in a plan $\pi = (A, \prec, B, L)$ is either:**
•**an unsatisfied sub-goal, i.e. a precondition of an action in *A* without a causal link that supports it; or**
•**a threat, i.e. an action that may interfere with a causal link.**



Flawless Plans and Solutions

•Proposition: A partial plan $\pi = (A,\prec,B,L)$ is a solution to the planning problem $\mathcal{P}=(\Sigma,s_i,g)$ if:

- **•π has no flaw;**
- **•the ordering constraints $\prec$ are not circular; and**
- **•the variable bindings *B* are consistent.**

•computation:

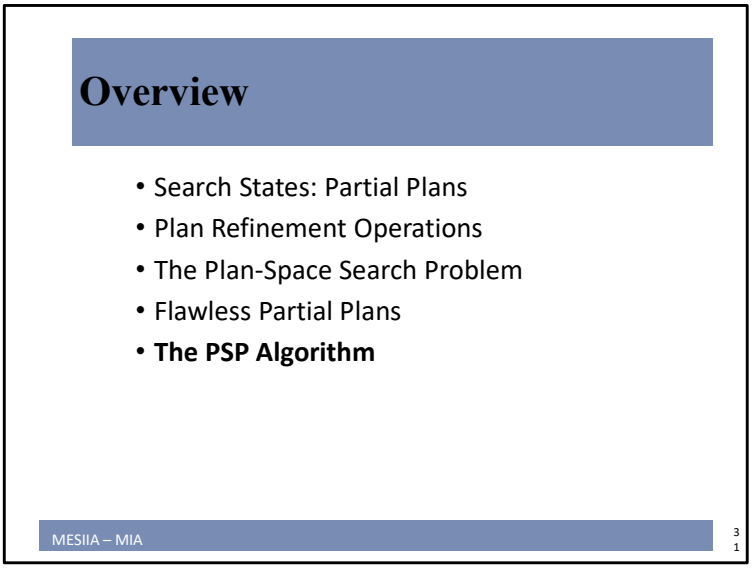
- •let partial plans in the search space only violate the first condition (have flaws)
- •partial plans that violate either of the last two conditions cannot be refined into a solution and need not be generated

•Proof: by induction on number of actions in *A*

- **•base case: empty plan**
  - •no flaws – every goal condition is supported by causal link from initial state
- **•induction step: totally ordered plan minus first step is solution implies plan including first step is a solution:**

$$\gamma(s_i, \langle a_1,\ldots,a_k\rangle) = \gamma(\gamma(s_i, a_1), \langle a_2,\ldots,a_k\rangle)$$

  - •truncated plan is solution to different problem



Overview

•Search States: Partial Plans

•Plan Refinement Operations

•The Plan-Space Search Problem

•Flawless Partial Plans

- just done: the goal test that completes the search problem

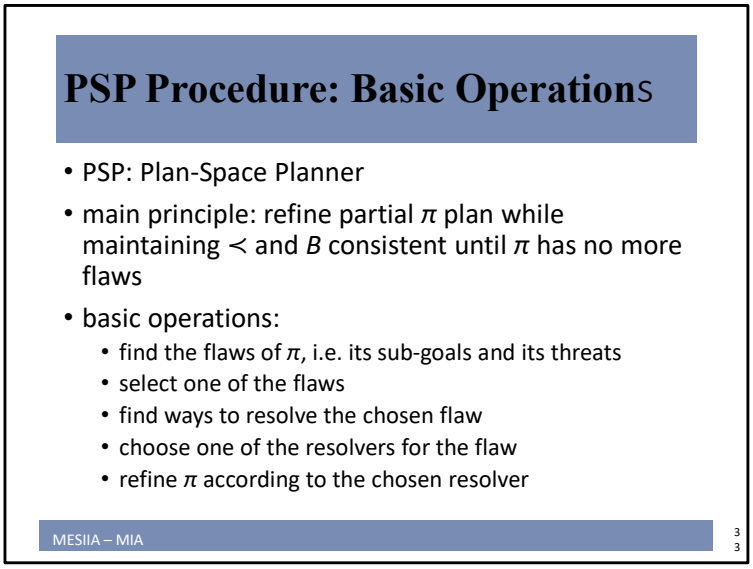
➢**The PSP Algorithm**

- now: a generic plan-space search planning algorithm

Plan-Space Planning as a Search Problem

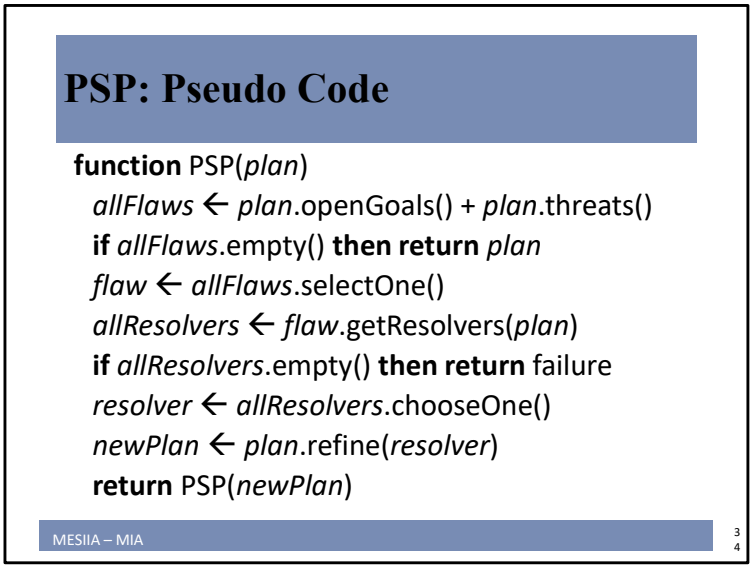
- given: statement of a planning problem $P=(O,s_i,g)$
- define the search problem as follows:
  - initial state: π_0 = ({init, goal},{(init≺goal)},{},{})
  - goal test for plan state p: p has no flaws
  - path cost function for plan π: $|\pi|$
  - successor function for plan state p: refinements of p that maintain $\prec$ and B

Plan-Space Planning as a Search Problem
•**given: statement of a planning problem $P=(O,s_i,g)$**
•**define the search problem as follows:**
•**initial state: π_0 = ({init, goal},{(init≺goal)},{},{})**
•**goal test for plan state p: p has no flaws**
•**path cost function for plan π: $|\pi|$**
•**successor function for plan state p: refinements of p that maintain $\prec$ and B**
•note: plan space may be infinite even when state space is finite

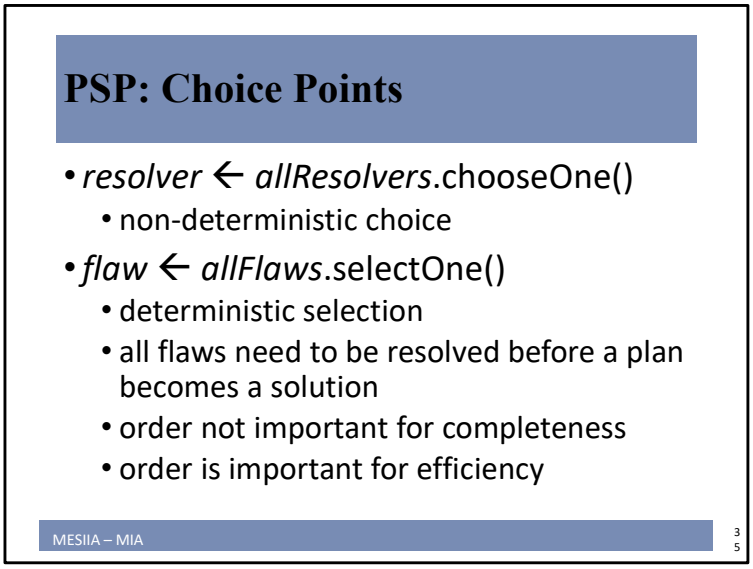


PSP Procedure: Basic Operations

- **PSP: Plan-Space Planner**
- **main principle: refine partial π plan while maintaining $\prec$ and B consistent until π has no more flaws**
- **basic operations:**
  - **find the flaws of π, i.e. its sub-goals and its threats**
    - simple for empty plan – all goal conditions are unachieved sub-goals and no threats
  - **select one of the flaws**
  - **find ways to resolve the chosen flaw**
  - **choose one of the resolvers for the flaw**
  - **refine π according to the chosen resolver**
    - modify the plan in such a way that $\prec$ and B are in a consistent state for the generated successor
      - aim: no need to verify consistency of $\prec$ and B for goal test



- **PSP: Pseudo Code**
- **function PSP(*plan*)**
  - refines the given partial plan into a solution plan; start with initial plan π_0
- ***allFlaws* ← *plan*.openGoals() + *plan*.threats()**
- **if *allFlaws*.empty() then return *plan***
  - see proposition in previous section: no flaws implies solution
- ***flaw* ← *allFlaws*.selectOne()**
- ***allResolvers* ← *flaw*.getResolvers(*plan*)**
  - represents all possible ways of removing the selected flaw from the partial plan
- **if *allResolvers*.empty() then return failure**
  - no resolvers means plan cannot be made flawless
- ***resolver* ← *allResolvers*.chooseOne()**
- ***newPlan* ← *plan*.refine(*resolver*)**
  - must maintain consistency of $\prec$ and B; new plan may contain new flaws
- **return PSP(*newPlan*)**

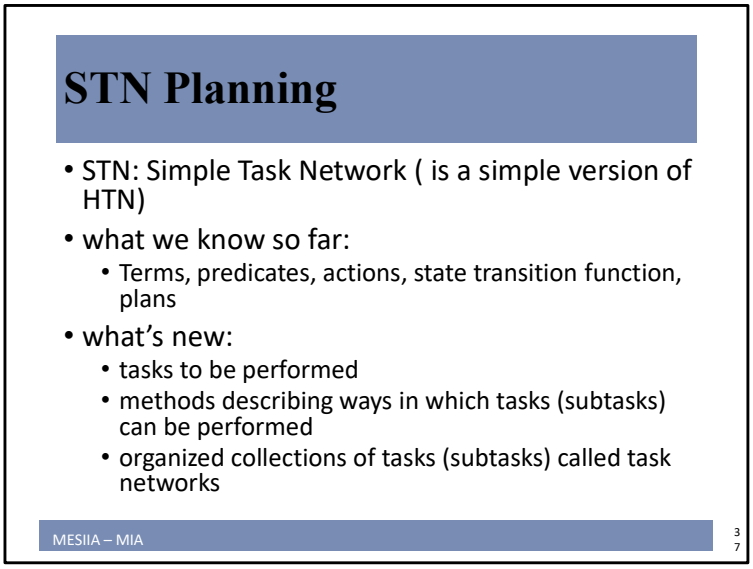


PSP: Choice Points

- ***resolver* ← *allResolvers*.chooseOne()**
  - **non-deterministic choice**
- ***flaw* ← *allFlaws*.selectOne()**
  - **deterministic selection**
  - **all flaws need to be resolved before a plan becomes a solution**
  - **order not important for completeness**
  - **order is important for efficiency**
    - for finding first plan, not so for finding all plans
    - deterministic implementation: using IDA*, for example

Hierarchical Planning

Hierarchical Task Network Planning (HTN)



STN Planning

•STN: Simple Task Network

•STN: simplified version of the more general HTN case to be discussed later

•what remains:

•terms, literals, operators, actions, state transition function, plans

•what's new:

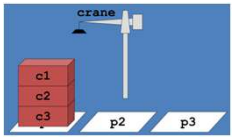
•tasks to be performed

•methods describing ways in which tasks (subtasks) **can be performed**

•organized collections of tasks (subtasks) **called task networks**

DWR Stack Moving Example

- task: move stack of containers from pallet p1 to pallet p3 in a way that preserves the order



- (informal) methods:
  - move topmost: take followed by put action
  - move stack: repeatedly move the topmost container until the stack is empty
  - move via intermediate: move stack to intermediate pile (reversing order) and then to final destination (reversing order again)

DWR Stack Moving Example

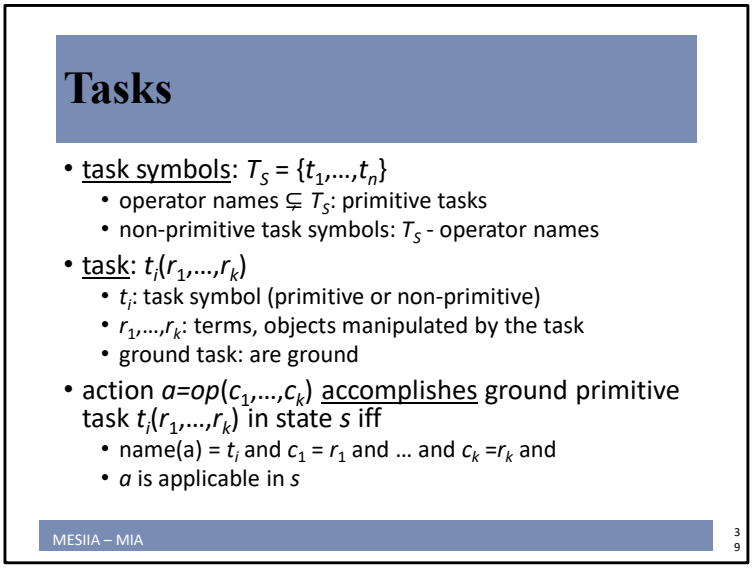
•task: move stack of containers from pallet p1 to pallet p3 in a way the preserves the order

- •preserve order: each container should be on same container it is on originally

•(informal) methods:

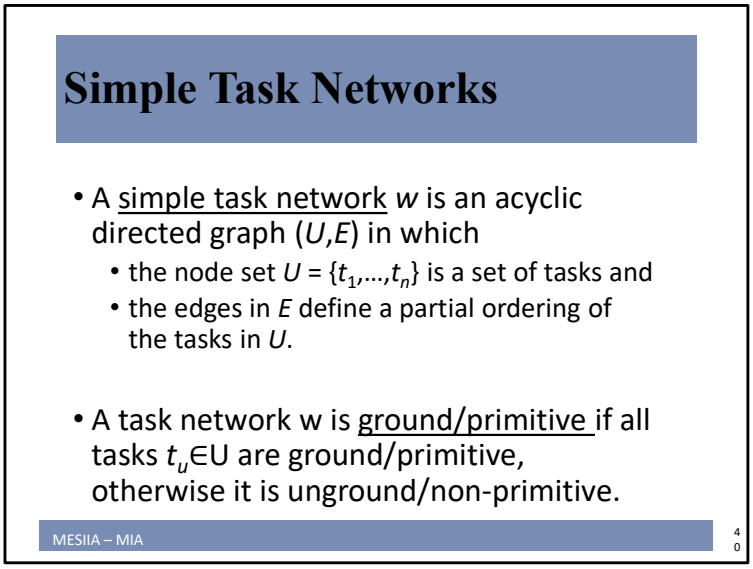
- •methods: possible subtasks and how they can be accomplished
- **•move via intermediate: move stack to intermediate pile (reversing order) and then to final destination (reversing order again)**
- **•move stack: repeatedly move the topmost container until the stack is empty**
- **•move topmost: take followed by put action**
  - •action: no further decomposition required

•note: abstract concept: stack



Tasks

- **task symbols: $T_S = \{t_1,\ldots,t_n\}$**
  - used for giving unique names to tasks
  - **operator names $\subsetneq T_S$: primitive tasks**
  - **non-primitive task symbols: T_S - operator names**
- **task: $t_i(r_1,\ldots,r_k)$**
  - **t_i: task symbol (primitive or non-primitive)**
    - tasks: primitive iff task symbol is primitive
  - **$r_1,\ldots,r_k$: terms, objects manipulated by the task**
  - **ground task: are ground** (all variables are related to objects of the world)
- **action *a* accomplishes ground primitive task $t_i(r_1,\ldots,r_k)$ in state *s* iff**
  - action *a* = (name(*a*), precond(*a*), effects(*a*))
  - **name(a) = t_i and**
  - ***a* is applicable in *s***
    - applicability: *s* satisfies precond(*a*)
- note: unique operator names, hence primitive tasks can only be performed in one way – no search!



Simple Task Networks

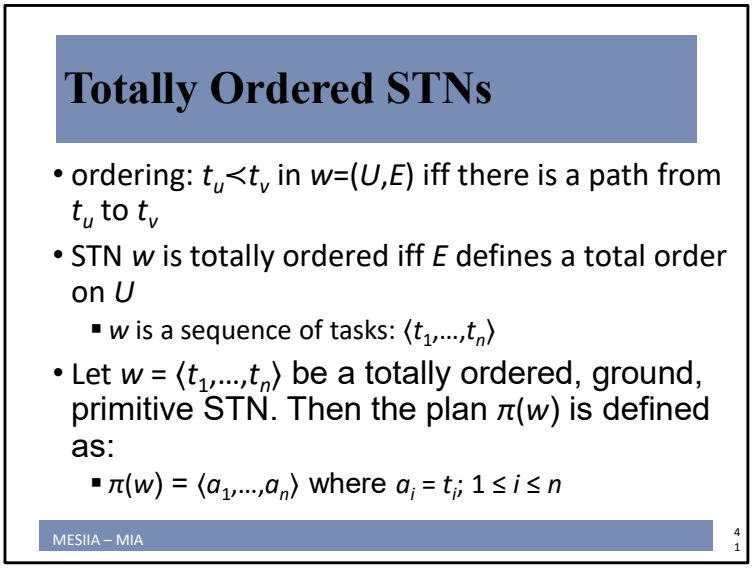
•A simple task network *w* is an acyclic directed graph (*U*,*E*) in which

•the node set $U = \{t_1,\ldots,t_n\}$ is a set of tasks and

•the edges in *E* define a partial ordering of the tasks in *U*.

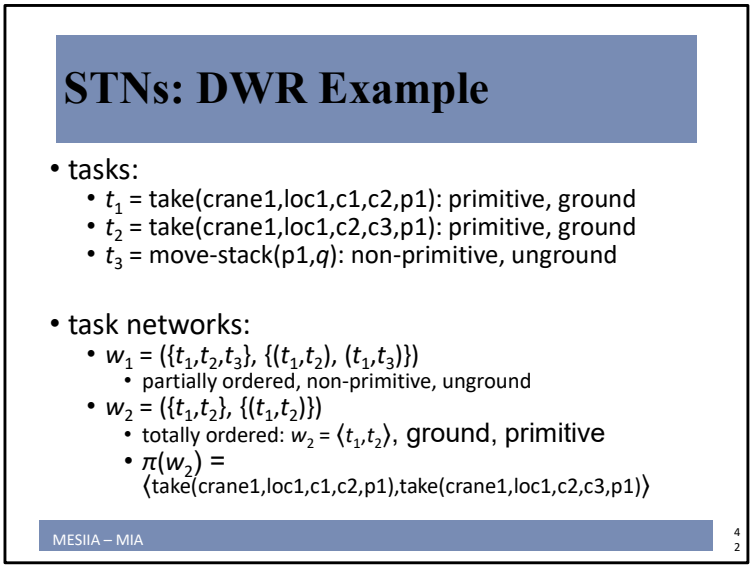
•A task network w is ground/primitive if all tasks $t_u \in U$ are ground/primitive, otherwise it is unground/non-primitive.

•simple task network: shortcut “task network”



Totally Ordered STNs

- **ordering: $t_u \prec t_v$ in $w=(U,E)$ iff there is a path from t_u to t_v**
- **STN w is totally ordered iff E defines a total order on U**
  - **w is a sequence of tasks: $\langle t_1,\ldots,t_n\rangle$**
    - sequence is special case of acyclic directed graph
    - t_1: first task in U; t_2 :second task in U; …; t_n: last task in U
- **Let $w = \langle t_1,\ldots,t_n\rangle$ be a totally ordered, ground, primitive STN. Then the plan $\pi(w)$ is defined as:**
  - **$\pi(w) = \langle a_1,\ldots,a_n\rangle$ where $a_i = t_i$; $1 \le i \le n$**



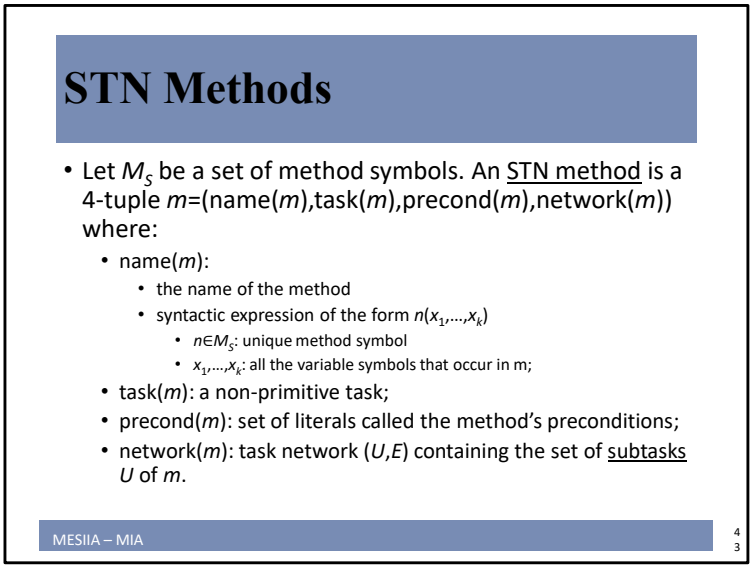
STNs: DWR Example

•tasks:

- **•t_1 = take(crane,loc,c1,c2,p1): primitive, ground**
  - •carne "crane" at location "loc" takes container "c1" of container "c2" in pile "p1"
- **•t_2 = take(crane,loc,c2,c3,p1): primitive, ground**
- **•t_3 = move-stack(p1,q): non-primitive, unground**
  - •move the stack of containers on pallet "p2" to pallet "q" (variable)

•task networks:

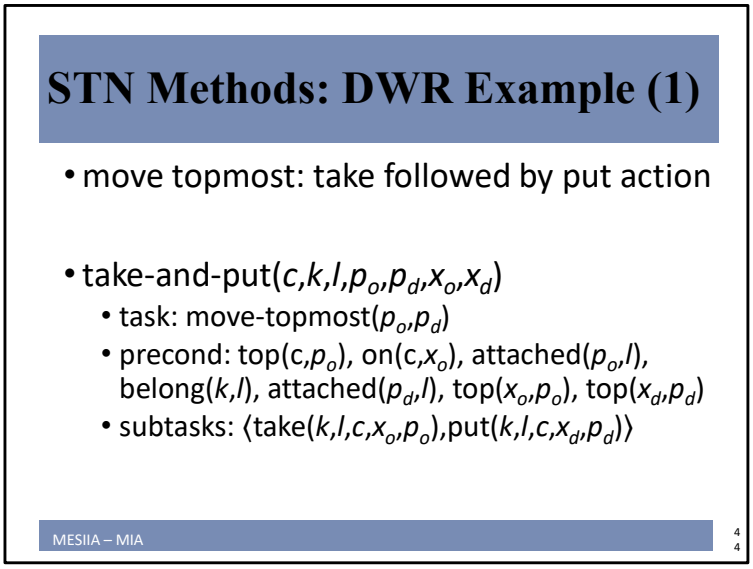
- **•w_1 = ({t_1,t_2,t_3}, {(t_1,t_2), (t_1,t_3)})**
  - **•partially ordered, non-primitive, unground**
- **•w_2 = ({t_1,t_2}, {(t_1,t_2)})**
  - **•totally ordered: w_2 = ⟨t_1,t_2⟩, ground, primitive**
  - **•$\pi(w_2)$ = ⟨take(crane,loc,c1,c2,p1),take(crane,loc,c2,c3,p1)⟩**



STN Methods

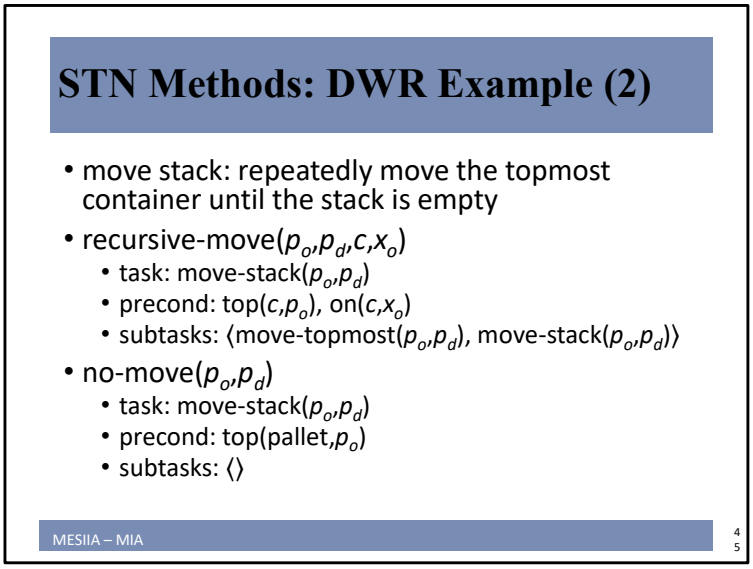
•Let M_S be a set of method symbols. An STN method is a 4-tuple m=(name(m),task(m),precond(m),network(m)) where:

- •method symbols: disjoint from other types of symbols
- •STN method: also, just called method
- **•name(m):**
  - **•the name of the method**
    - •unique name: no two methods can have the same name; gives an easy way to unambiguously refer to a method instances
  - **•syntactic expression of the form $n(x_1,\ldots,x_k)$**
    - **•$n \in M_S$: unique method symbol**
    - **•$x_1,\ldots,x_k$: all the variable symbols that occur in m;**
      - •no "local" variables in method definition (may be relaxed in other formalisms)
- **•task(m): a non-primitive task;**
  - •what task can be performed with this method
    - •non-primitive: contains subtasks
- **•precond(m): set of literals called the method's preconditions;**
  - •like operator preconditions: what must be true in state s for m to be applicable
    - •no effects: not needed if problem is to refine/perform a task as opposed to achieving some effect
- **•network(m): task network (U,E) containing the set of subtasks U of m.**
  - •describes one way of performing the task task(m); other methods may describe different way of performing same task: search!
    - •method is totally ordered iff network is totally ordered



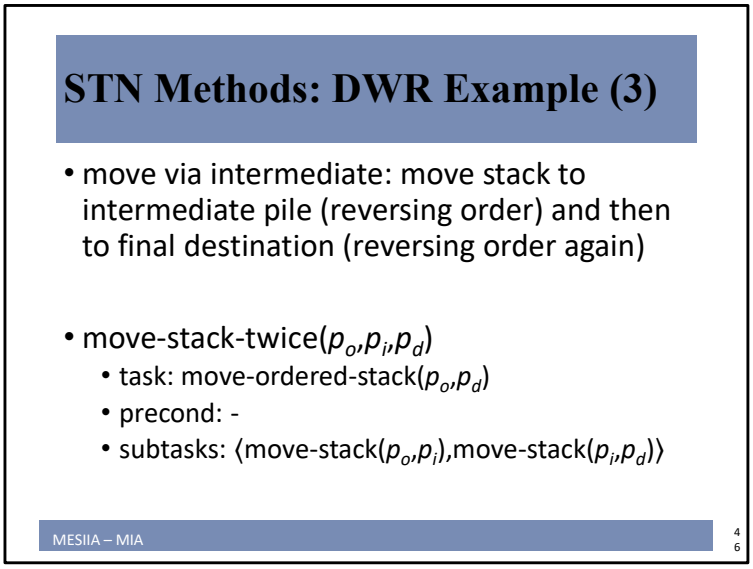
STN Methods: DWR Example (1)

- **move topmost: take followed by put action**
  - simplest method from previous example
- **take-and-put(*c*,*k*,*l*,p_o,p_d,x_o,x_d)**
  - using crane *k* at location *l*, take container *c* from object x_o (container or pallet) in pile p_o and put it onto object x_d in pile p_d (*o* for origin, *d* for destination)
  - **task: move-topmost(p_o,p_d)**
    - move topmost container from pile p_o to pile p_d
  - **precond:**
    - **top(c,p_o), on(c,x_o)**: pile must be empty with container *c* on top
    - **attached(p_o,*l*), belong(*k*,*l*), attached(p_d,*l*)**: piles and crane must be at same location
    - **top(x_d,p_d)**: destination object must be top of its pile
  - **subtasks: ⟨take(*k*,*l*,*c*,x_o,p_o),put(take(*k*,*l*,*c*,x_d,p_d))⟩**
    - simple macro operator combining two (primitive) operators (sequentially)



STN Methods: DWR Example (2)

- **move stack: repeatedly move the topmost container until the stack is empty**
- **recursive-move(p_o,p_d,c,x_o)**
  - move container c which must be on object x_o in pile p_o to the top of pile p_d
  - **task: move-stack(p_o,p_d)**
    - move the remainder of the satck from p_o to p_d: more abstract than method
  - **precond: top(c,p_o), on(c,x_o)**
    - p_o must be empty; c is the top container
    - method is not applicable to empty piles!
  - **subtasks: ⟨move-topmost(p_o,p_d), move-stack(p_o,p_d)⟩**
    - recursive decomposition: move top container and then recursive invocation of method through task
- **no-move(p_o,p_d)**
  - performs the task by doing nothing
  - **task: move-stack(p_o,p_d)**
    - as above
  - **precond: top(pallet,p_o)**
    - the pile must be empty (recursion ends here)
  - **subtasks: ⟨⟩**
    - do nothing does nothing

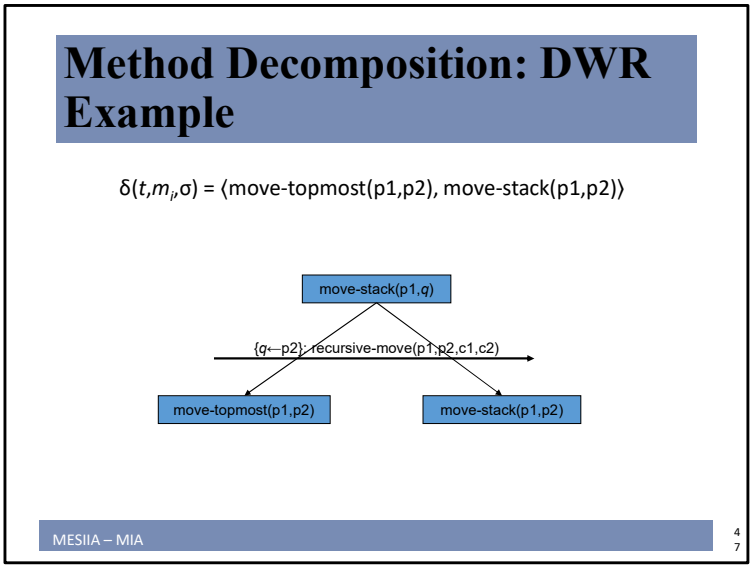


STN Methods: DWR Example (3)

•move via intermediate: move stack to intermediate pallet (reversing order) and then to final destination (reversing order again)

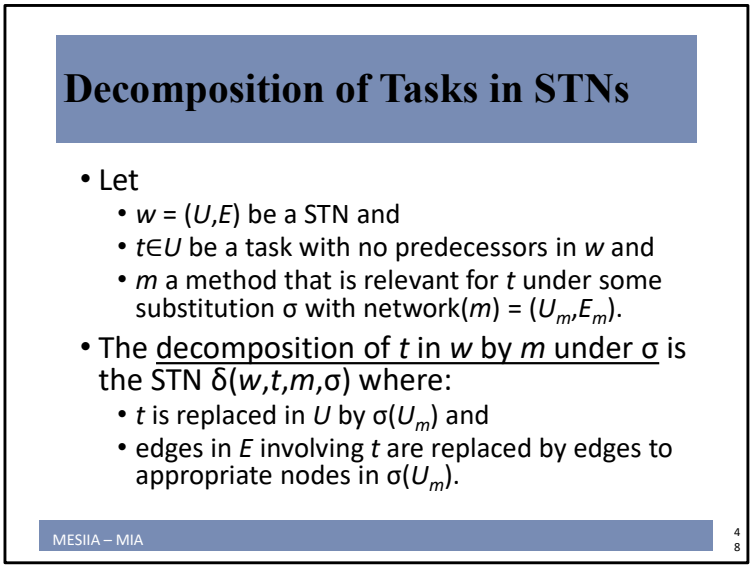
•move-stack-twice(p_o,p_i,p_d)

- •move the stack of containers in pile p_o first to intermediate pile p_i then to p_d, thus preserving the order
- **•task: move-ordered-stack(p_o,p_d)**
  - •move the stack from p_o to p_d in an order-preserving way
- **•precond: -**
  - •none; should mention that piles must be at same location and different
- **•subtasks: ⟨move-stack(p_o,p_i),move-stack(p_i,p_d)⟩**
  - •the two stack moves



Method Decomposition: DWR Example

- **$\delta(t,m_i,\sigma) = \langle$move-topmost(p1,p2), move-stack(p1,p2)$\rangle$**
- **[figure]**
- graphical representation (called a decomposition tree):
  - view as AND/OR-graph: AND link – both subtasks need to be performed to perform super-task
  - link is labelled with substitution and method instance used
  - arrow under label indicates order in which subtasks need to be performed
  - often leave out substitution (derivable) and sometimes method parameters (to save space)



Decomposition of Tasks in STNs

•idea: applying a method to a task in a network results in another network

•Let

•w = (U,E) be a STN and

•$t \in U$ be a task with no predecessors in w and

•m a method that is relevant for t under some substitution σ with network(m) = (U_m,E_m).

•The decomposition of t in w by m under σ is the STN δ(w,u,m,σ) where:

•t is replaced in U by σ(U_m) and

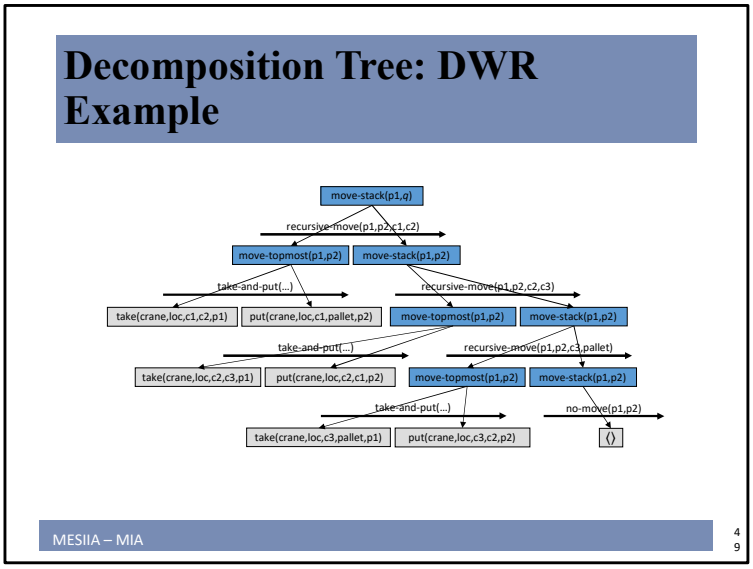
•replacement with copy (method maybe used more than once)

•edges in E involving t are replaced by edges to appropriate nodes in σ(U_m).

•every node in σ(U_m) should come before nodes that came after t in E

•σ(E_m) needs to be added to E to preserve internal method ordering

•ordering constraints must ensure that precond(m) remains true even after subsequent decompositions

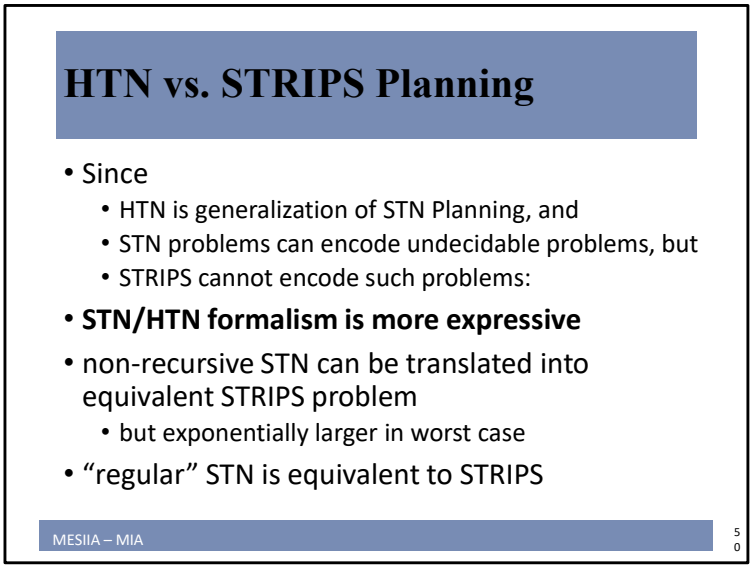


Decomposition Tree: DWR Example

•choose method: recursive-move(p1,p2,c1,c2) – binds variable *q*
•decompose into two sub-tasks
•choose method for first subtask: take-and-put: c1 from c2 onto pallet
•decompose into subtasks – primitive subtasks (grey) cannot be decomposed/correspond to actions
•choose method for second sub-task: recursive-move (recursive part)
•decompose (recursive)
•choose method and decompose (into primitive tasks): take-and-put: c2 from c3 onto c1
•choose method and decompose (recursive)
•choose method and decompose: take-and-put: c3 from pallet onto c2
•choose method (no-move) and decompose (empty plan)

•note:

- •(grey) leaf nodes of decomposition tree (primitive tasks) are actions of solution plan
- •(blue) inner nodes represent non-primitive task; decomposition results in sub-tree rooted at task according to decomposition function δ
- •no search required in this example



HTN vs. STRIPS Planning

- **Since**
  - **HTN is generalization of STN Planning, and**
  - **STN problems can encode undecidable problems, but**
  - **STRIPS cannot encode such problems:**
- **STN/HTN formalism is more expressive**
- **non-recursive STN can be translated into equivalent STRIPS problem**
  - **but exponentially larger in worst case**
- **"regular" STN is equivalent to STRIPS**
  - non-recursive
  - at most one non-primitive subtask per method
  - non-primitive sub-task must be last in sequence

End