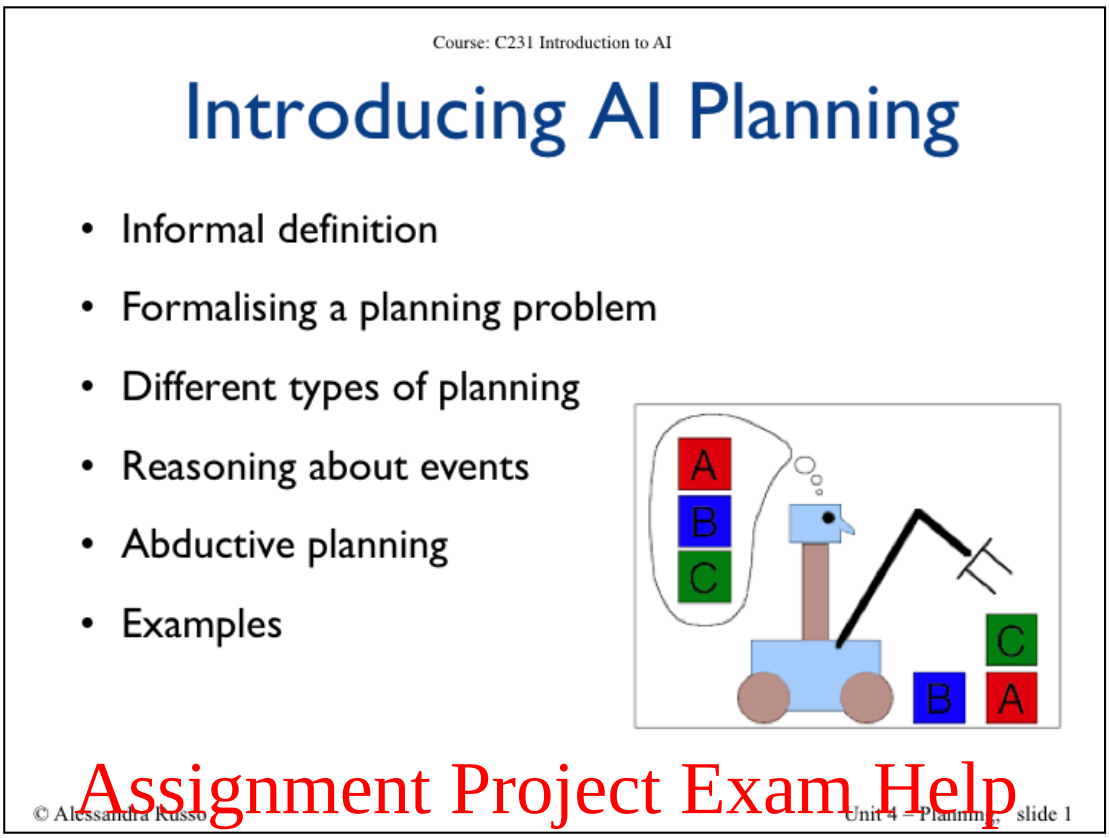


We have concluded the previous lecture with a brief introduction of what a planning problem is, and we have given a simple example of such a task. In this lecture, we will concentrate more on planning. There are two main classes of planning, called respectively "planning with certainty" and "planning under uncertainty". The first class of planning works under very strong assumptions for which the agent has access to the full state of the world and has complete description of effects of actions. Planning under uncertainty, on the other hand, assumes that the agent has partial knowledge of the state of the world and therefore does not have full knowledge and certainty on effects of actions when performed. The agent should plan to react to its environment, and decide what to do depending on its preferences.

In this lecture, we concentrate on planning with certainty. We will, in particular, define the main components of a planning problem, highlight some of the existing languages used for specifying a planning problem and introduce different approaches to solve planning problems.

Planning is strictly related to temporal reasoning. So, we will then introduce a specific formalism called Event Calculus for representing and reasoning about events and effects of events over time, and we will look at a specific type of planner, called Abductive Event Calculus Planner and show how the same abductive inference that we have covered in the previous lecture can be used to solve a planning problem.

We will then conclude with some examples.

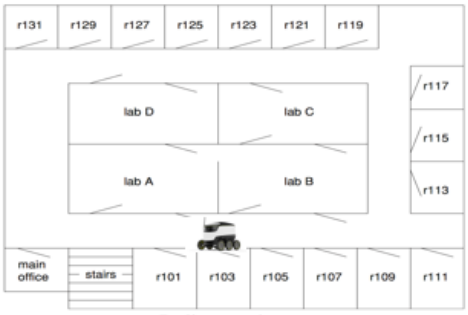
The content of this lecture is based on the textbooks: "Artificial Intelligence: Foundations of Computational Agents", by Poole & Mackworth, and "Artificial Intelligence: A Modern Approach" by Russell & Norvik.

Course: C231 Introduction to AI

What is planning?

Planning is about how an agent achieves its goals.

Find a sequence of actions to solve a goal.



Delivery robot

To reason about what to do, an agent must have:

- goals,
- model of the world,
- model of consequences of actions.

© Alessandra Russo Unit 4 – Planning, slide 2

Planning is about how an agent achieves its goals. An agent does not usually achieve its goals in one step. What the agent does depends on the state it is in, which, in turn, depends on what it has done in the past. So, to solve a planning problem, an agent needs a representation of its actions and their effects. It can then use these models to find a sequence of actions that allow it to achieve its goals.

Given a description of the possible initial states of the world, a description of the desired goals, and a description of a set of possible actions, with their effects, a planning problem consists of synthesizing a plan that is guaranteed (when applied to any of the initial states) to generate a state that contains the desired goals. Such a state is called **goal state**.

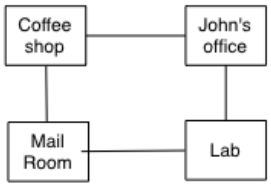
Let's consider for instance a delivery robot example.

Course: C231 Introduction to AI

Example: delivery robot

State features

rhc, rhm, mw, swc, cs, off, mr, lab



Actions

move clockwise (**mc**), deliver mail (**dm**),
pick up a coffee (**puc**), deliver the coffee (**dc**),
pick up mail (**pum**), move unticlockwise (**mcc**),

Action specification

Actions	*Precond*	*Postcond*
puc	not rhc $\wedge$ cs	rhc'
dc	rhc $\wedge$ off	not swc'
pum	wm $\wedge$ mr	rhm'
dm	rhm $\wedge$ off	

Explicit state space representation is a set of triples <s, a, s'>

But too many states and representation not very flexible

We consider here a simplified delivery robot example with four different locations: coffee shop, lab, mail room and Simon's office. The robot task is to deliver coffee to Simon's office, collect mails from the delivery room and deliver them to Simon's office.

Formal representation of a planning problem can be either propositional or first-order, depending on the type of planning approach that is used. Propositional representations are computationally faster, but they suffer of the problem of exponential explosion of the state space that the planner has to consider in order to compute valid plans. In this simplified example, we use a propositional representation.

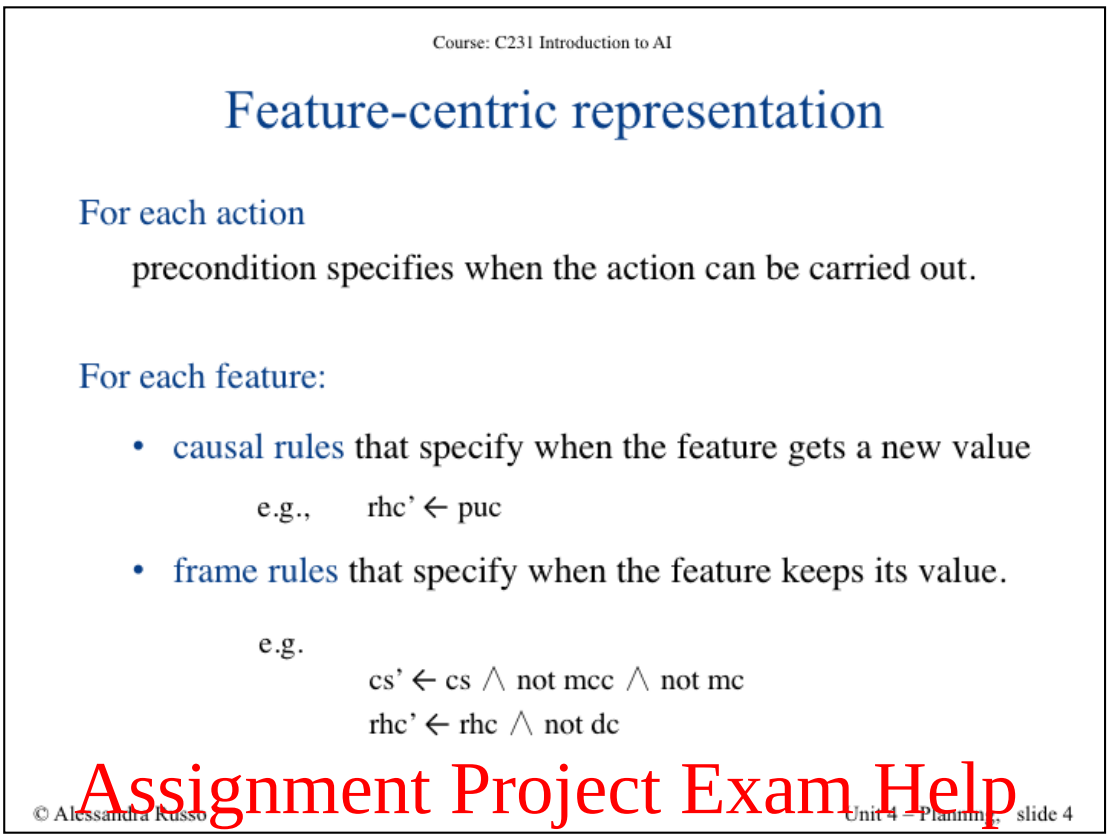
Each state is expressed as a Boolean combination of four main features: robot has coffee (**rhc**), robot has mail (**rhm**), mail is waiting (**mw**) and Simon wants coffee (**swc**), together with the four possible locations in which the robot might be: robot is at the coffee shop (**cs**), at Simon's office (**off**), at the mail room (**mr**) and in the lab (**lab**).

The robot can perform six actions: move clockwise (**mc**), move unticlockwise (**mcc**), pick up a coffee (**puc**), deliver the coffee (**dc**), pick up mail (**pum**) and deliver mail (**dm**).

Not all actions can be carried out in every states. Actions have preconditions that specify when they can be carried out. They have also post-conditions or effects, which specify properties that the resulting states should satisfy.

For example, the robot can pick up coffee if it is at the coffee shop and it does not have the coffee already. Similarly, it can deliver coffee if it is holding a coffee and it is in Simon's office, it can pick up mails if there are mails waiting and the robot is in the mail room, and finally, it can deliver mails if it holds mails and it is in john's office.

Given action descriptions in terms of preconditions and effects, the planning state space could be expressed by triples defining the current state, actions transitions and new states. But such an explicit state space representation has limitations. In particular, the number of states grows exponentially with the number of state features, and the addition of new features would require the reformulation of the entire state space.



Two other approaches can be used to specify a planning problem: (i) a feature-centric approach and (ii) an action-centric approach. We briefly introduce these two methods in this and the next slide, respectively.

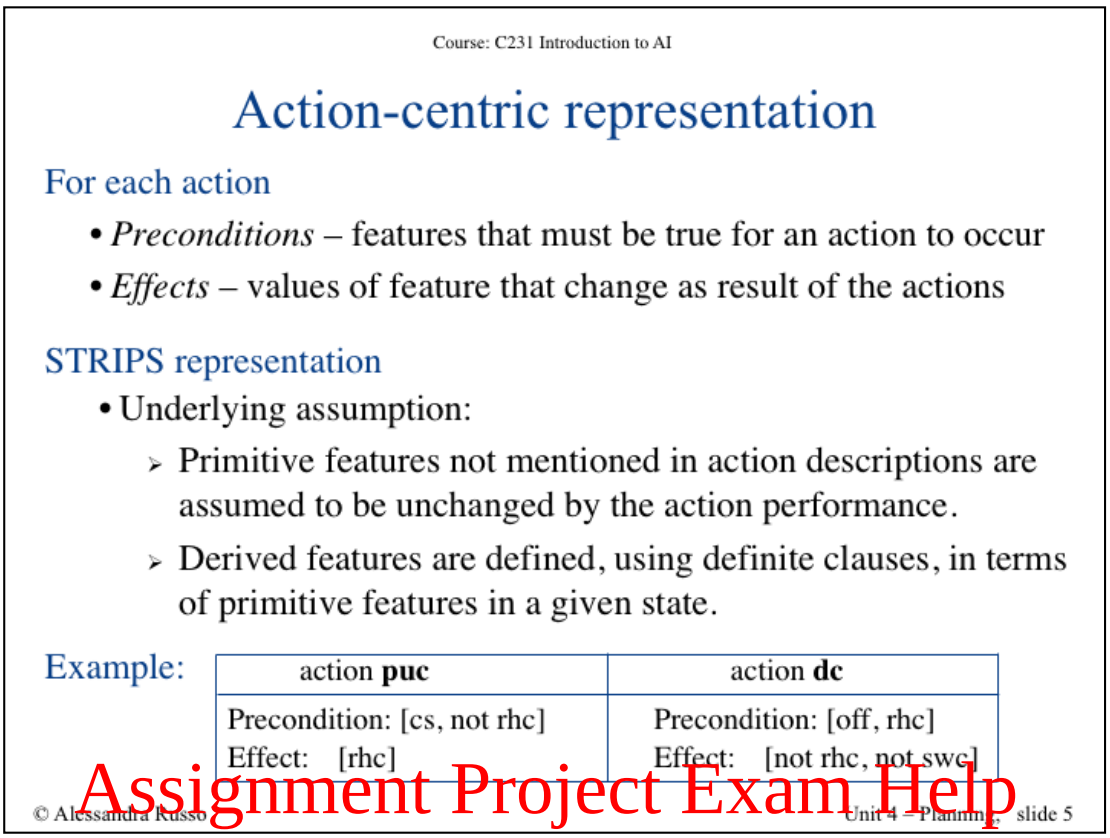
The feature-centric approach consists of defining, for each feature, two types of rules, **causal rules** and **frame rules**.

A **causal rule** specifies under what conditions and action performance a feature gets a new value. For example, in this slide the state "robot holds a coffee" becomes true if the robot performs the action of picking up a coffee.

A **frame rule** specifies under what conditions a feature maintains its value, i.e. does not change. For example, in this slide, the feature of "the robot being at the coffee shop" **persists** if the root is currently at the coffee shop and does not move clockwise or unticlockwise.

In a feature-centric approach, specifying a planning problem means specifying, for each feature, what makes the feature change its value, and what makes it keep its value. So, basically specifying what changes as well as what does not change.

But note that both causal rules and frame rules do not define when an action can be performed or not. This is still captured by the notion of precondition of an action.



In the action-centric representation of a planning problem, the emphasis is on actions instead of state-features. The description is more compact and consists of describing, for each action, precondition and effects. An example of action-centric specification of planning is STRIPS (Stanford Research Institute Problem Solver), which you have already covered in the first part of this course. As you have already seen, in this approach, the key assumptions are:

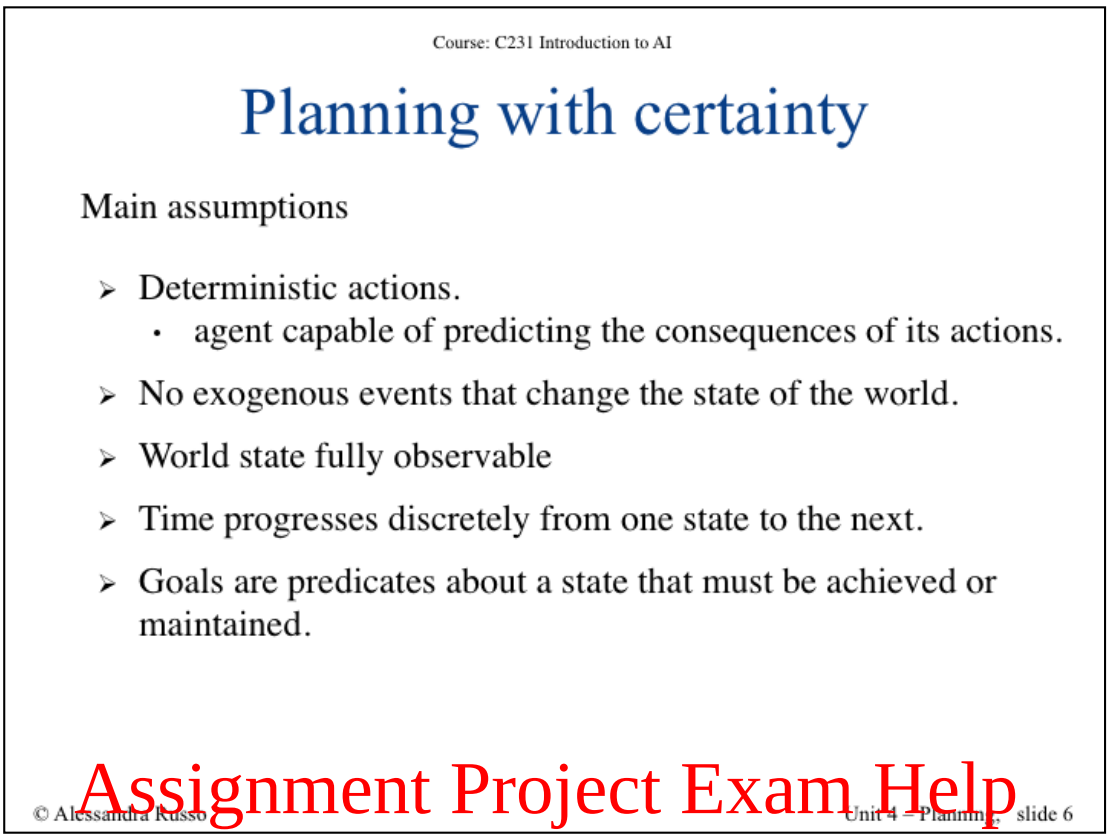
1) features, divided into **primitive** and **derived** features. The specification of changes over derived features are expressed using definite clauses over values of primitive features.

2) primitive features not mentioned in the effects of actions are assumed to remain unchanged. So only the features that are changed need to be specified in the effects of an action. This is because the underlying assumption is that most things are left unchanged by a single action. Therefore, by default, unchanged features are not included in the definition of effects of actions.

So a primitive feature V has value v after an action "act", if V = v is part of the effect list of the action act, or if V is not mentioned in the effect list of act, and had already value v immediately before the action act is performed. Derived features can be derived from the values of the primitive features at each time.

We have given, in this slide, two examples of action descriptions in STRIPS.

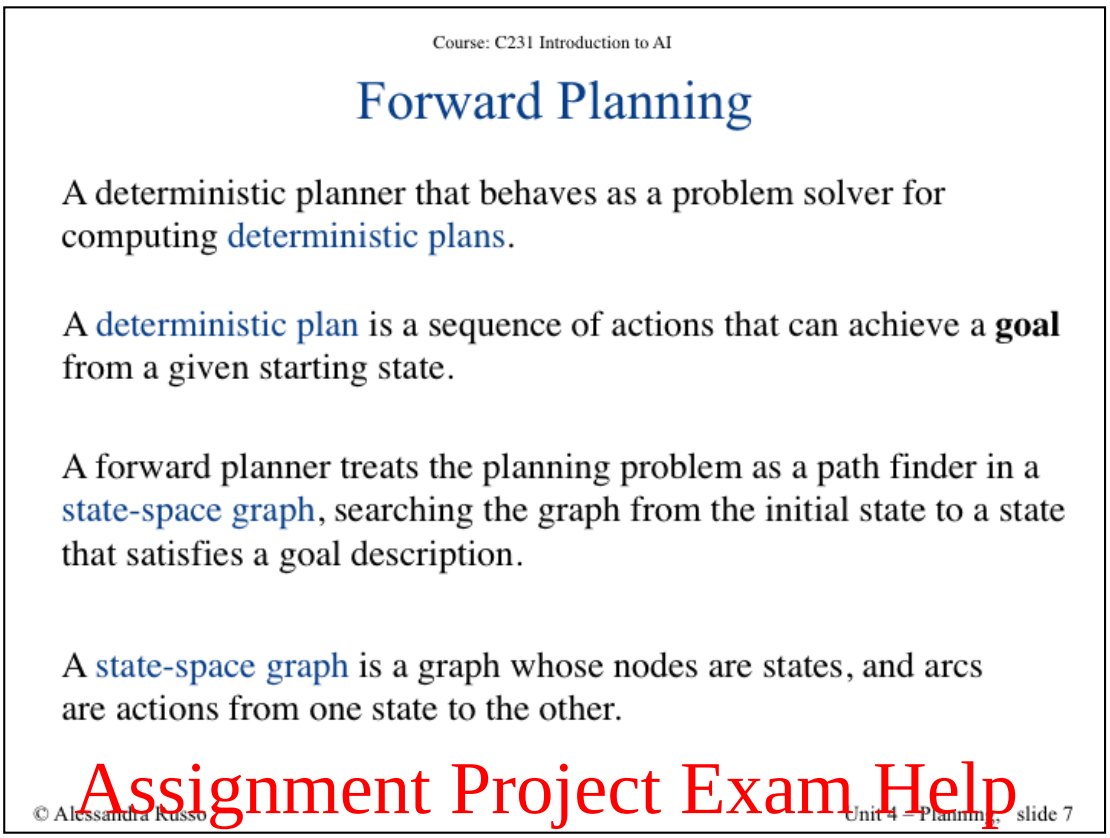
It is easy to see that STRIPS representations can be mapped into a feature-centric representations. For every feature that appears in the effect list of an action, we can define, in fact, a causal rule in a corresponding feature-centric representation. Similarly, for every feature not mentioned in the effect list of an action, we would need to write a frame rule.



We have informally introduced the various components of a planning specification. This specification is used to reason about the state space for possible plans. The search for plan solutions within this space depends on the type of the particular planning algorithm used and on specific properties of the planning approach.

In particular, we consider for the rest of this lecture, planning approaches that assume:

- **The agent's actions are deterministic**. In this way, the agent knows the effects of actions and can infer precisely what will be true and/or false after the execution of its actions. Hence, he knows the consequences of its actions.
- **There are no exogenous events** that can occur beyond the control of the agent. In this way, if the agent knows what the current state is and known the consequences of its actions, it will know what is true and false in the next state.
- **The world is fully observable.** This means that the agent can observe the current state of the world and knows at each time point what is true and false in the current state of the world.
- Regarding time points, we also assume that **time progresses discretely**. Given a current time (and therefore a current state), the next time point is the next state in which the agent will find itself after the execution of its action.
- Finally, **goals** are predicates of states that must be **achieved** or **maintained**.



A deterministic plan is a sequence of actions that an agent can perform to achieve a goal from a given starting state. A deterministic planner is therefore a **problem solver** that produces a plan. The solver takes in input an initial world description, a specification of the actions available to the agent, and a goal description.

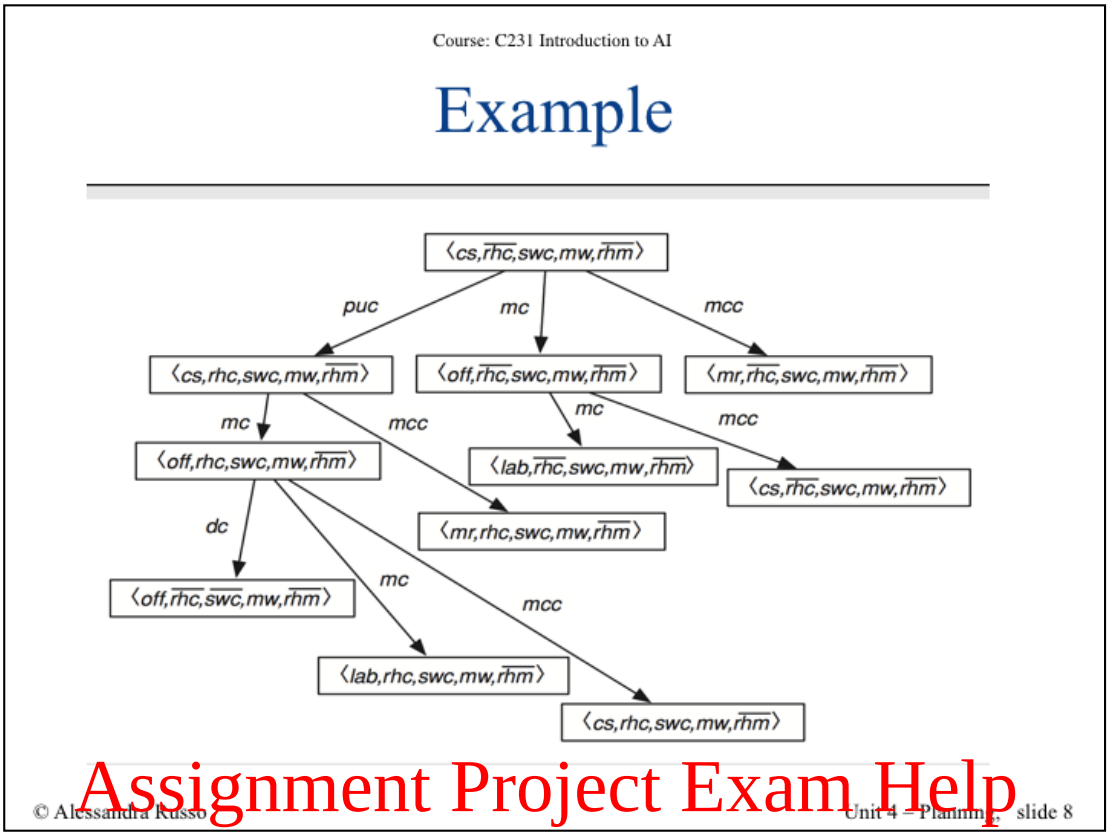
One of the simplest ways for computing a plan is to treat the planning problem as a path finding in the state-space graph, where nodes are states, and arcs correspond to actions from one state to another. For every state in this graph, the coming out arcs are all legal actions that can be carried out in that state. That is, whose precondition holds in that state and where the resulting state satisfies every given maintenance goal. A plan is a path from the initial state to a state that satisfies the achievement goal.

A forward planner searches the state-space graph from the initial state looking for a state that satisfies the achievement goal.

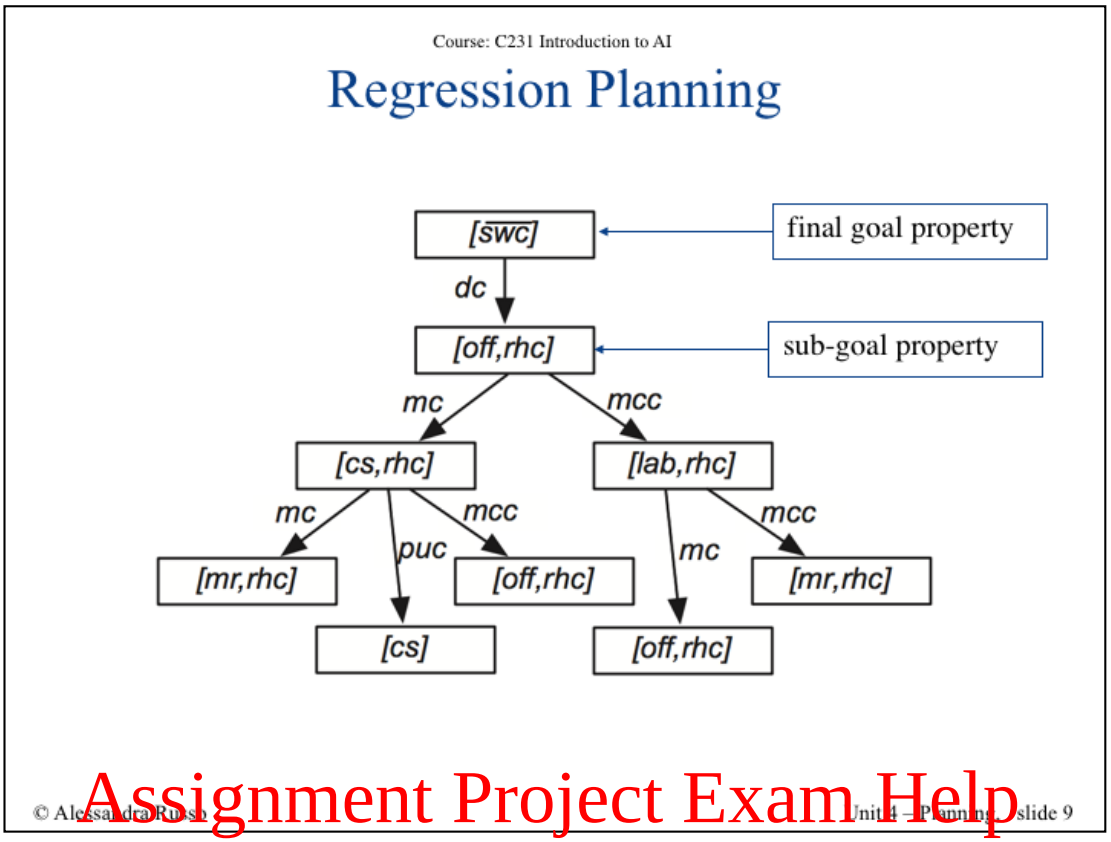
Search algorithms, such as A* with multiple-path pruning or iterative deepening, can be used during the computation of a plan. The complexity of the search space is defined by the **forward branching factor** of the graph. **The branching factor is the set of all possible actions at any state, which may be quite large**.

When the domain becomes big, the branching factor increases and so the search space explodes. This complexity can be reduced by finding good heuristics that overcome the combinatorial explosion.

To perform the computation using a forward planner, a node of the state-space can be either a full complete state description (as partly shown in next slide), or the partial plan (or path) from the initial state to that node. In the latter case what holds at the current state can be derived using the path and the action descriptions. Each of these two different representations have pro and cons. The former may be expensive in the memory space required, whereas the latter maybe computationally expensive in determining, for instance, loops during the search.



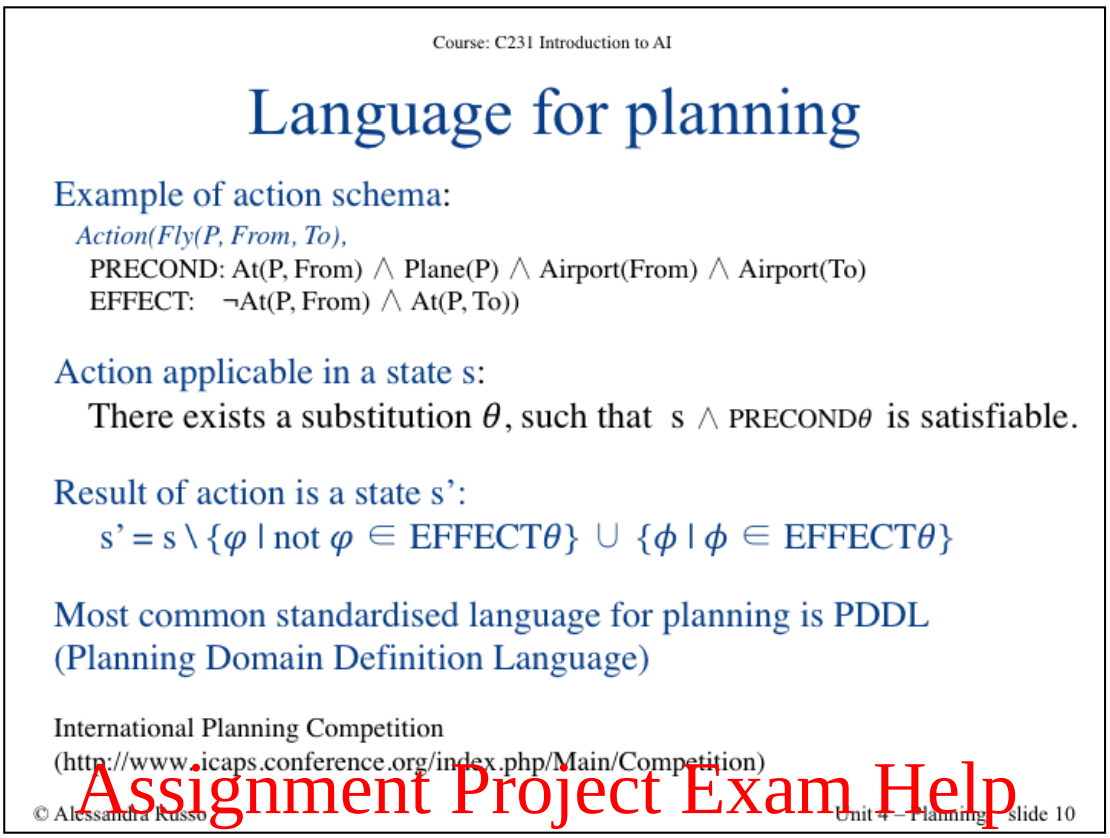
This slide shows part of a state-space graph for a forward planner. The graph starts from the given initial state and shows for each state the actions that can be performed given their respective preconditions (see slide 3).



In the forward planning we have described the state-space searching as a forward search method. But it is also possible to perform search **backward** starting from the set of states that satisfy the goal. Whereas the initial state is usually fully specified and so the search starts off from a single state, the goal does not usually fully specify a state and so there would be many goal states that satisfy the goal. So searching from multiple states is not very efficient.

A regression planning is a form of backward search that uses however a different search space. The nodes in the search space are not states, but sub-goals to be achieved, where each sub-goal is still a set of assignments to (some of) the features. Arcs are still actions. In particular, an arc from a node g to g′, labeled with an action **act**, means **act** is the last action that is carried out before goal g is achieved, and the node g' is the goal that must be true immediately before **act** so that g is true immediately after **act**. The start node is the final goal to be achieved, which is also a conjunction of assignments of values to state-based features. Basically in a regression planning, the graph start from the goal property that needs to be achieved and the graph expands considering as next states the sub-goals together with the action that leads to the goal state. An example is given in this slide.

Suppose the goal is to achieve "not swc". The start node is [not swc], indicted in the figure with a overlined swc). The planner than chooses an action that achieves "not swc". In this case, there is only one possible action: **dc**. The preconditions of dc are off $\wedge$ rhc. Thus, there is one arc in the state space $\langle$[not swc], [off , rhc]$\rangle$ labeled with dc. Then the planner considers the node [off , rhc]. There are two actions that can achieve off, namely mc from cs and mcc from lab. There is one action that can achieve rhc, namely puc. However, puc has as a precondition cs $\wedge$ rhc, and if it were applied to get the sub-goal [off , rhc], would have required a next node [cs $\wedge$ not rhc $\wedge$ off], which is inconsistent because the robot cannot be in two different locations at the same time. So the only possible actions are "mc" and "mcc", as shown in the slide. And so on, until a node is reached that satisfies a given initial state.

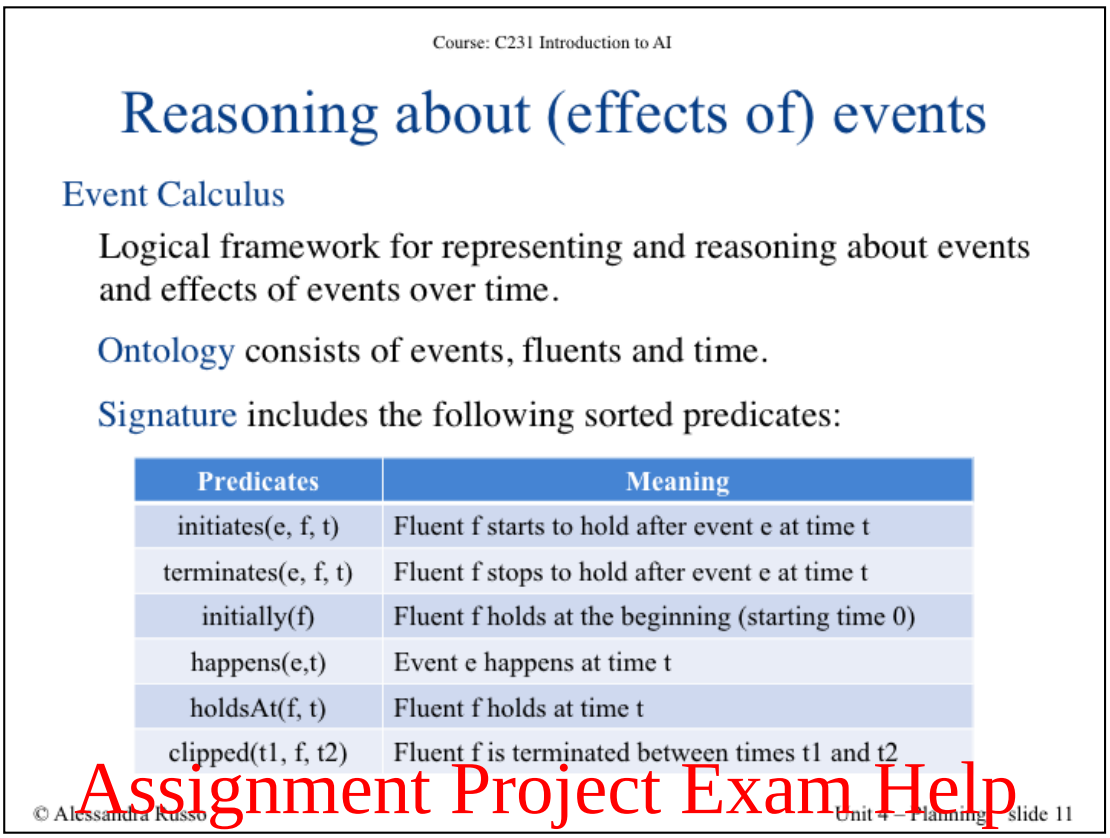


One of the key aspect of planning is finding a language that is expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient algorithms to operate over it. In the previous slides we have used examples written in STRIPS language. These were mainly expressed in propositional logic. But more general representations of actions would instead use (typed) predicate logic.

This slide gives an example of actions specified using an unground representation where the action itself has variables *p, from, to*, and pre-conditions and effects are also expressed in terms of these variables. This representation is called **action schema**, and all possible instantiations of this action schema constitute the actual actions. Evaluation of preconditions and effects of actions are then given by instantiations of the preconditions and effects of the action schemas.

Different planning formalisms have been proposed in the AI community, and to facilitate benchmarking between different planning algorithms, these formalisms have now been systematized within a standard syntax called the the Planning Domain Definition Language, or PDDL. PDDL includes sublanguages for STRIPS, ADL. Since the proposal of PDDL standard language, an international competition for planning algorithms has been created and run every year together with the top conference on automated planning and scheduling (ICAP). See for example ICAP 2018 competition on

http : // www. icaps.conference.org/index.php/Main/Competition

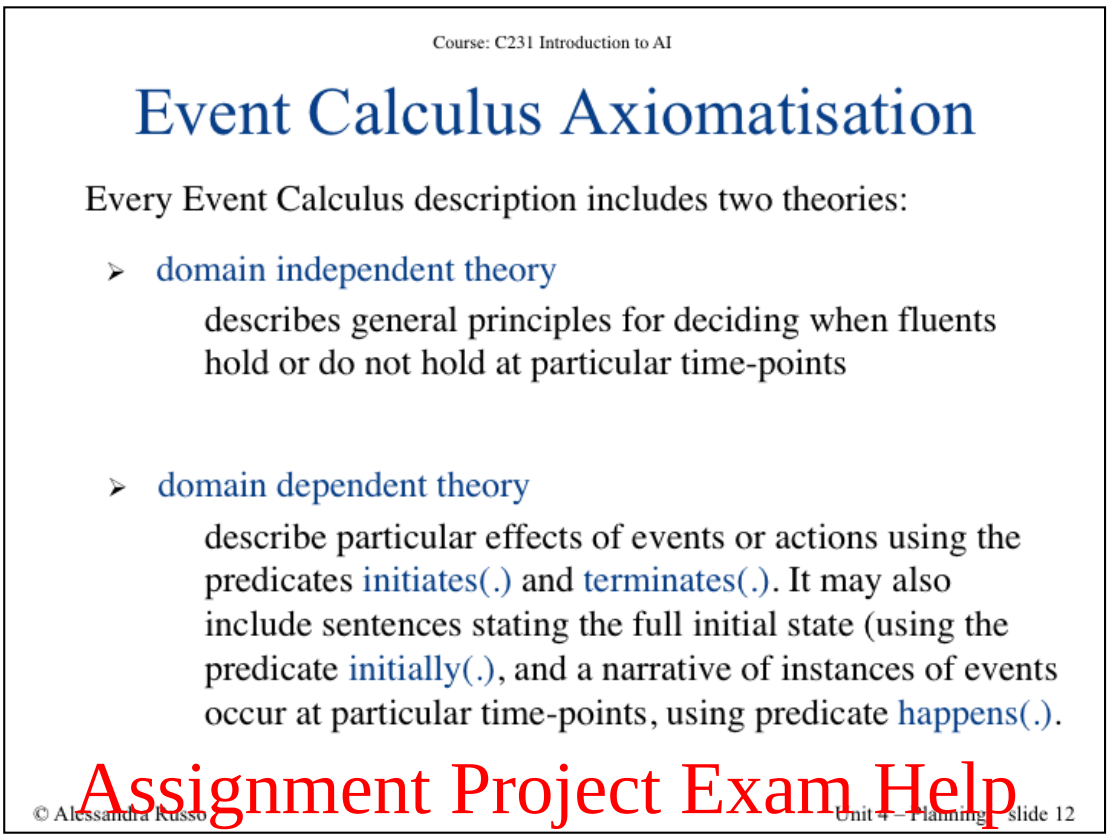


In 1986, Kowalski and Sergot proposed Event Calculus, a logical framework for representing and reasoning about events and effects of events over time. It was initially used as a mechanism for updating databases. But since then Event Calculus has been used and formalised in many different ways. We consider here a simple Event Calculus (EC) formalisation.

The EC language is based on an ontology consisting of (i) a set of time-point, which is isomorphic to the non-negative integers, (ii) a set of time-varying properties, called fluents, and a set of events. The EC language includes the sorted predicates given in this slide, where e is an event variable, f is a fluent variable and t is a time variable.

The predicate *happens(e, t)* indicates that event actually occurs at time-point t; the predicate *initiates(e,f,t)* (resp. *terminates(e,f,t)*) means that if event e were to occur at t it would cause fluent f to be true (resp. false) immediately afterwards. The predicate *holdsAt(f,t)* indicates that fluent f is true at t.

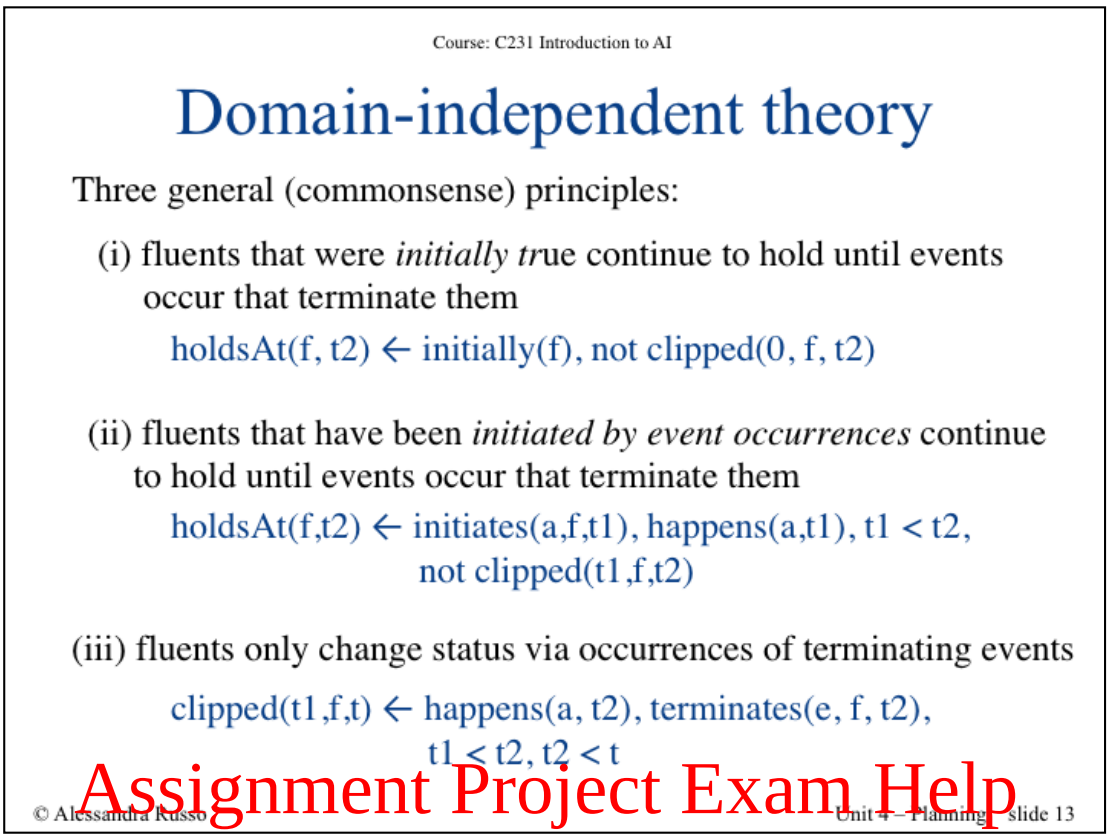
Fluents could be seen as state-based features that become true or false over time depending of the occurrence of events and effects that events have on them (i.e. whether they initiate or terminate them). Events instead are assumed to be atomic, in the sense that their occurrence is instantaneous. Event do not have a duration.



Every Event Calculus description includes two main theories.

(1) a core collection of domain-independent axioms (or rules) that describe general principles for deciding when fluents hold or do not hold at particular time-points.

(2) a collection of domain-dependent axioms, describing the particular effects of events, using the predicates *initiates(.)* and *terminates(.)*. The domain dependent theory includes also axioms stating the occurrence of specific events at specific time points, using the predicate *happens(.)*, often referred to as a *narrative*, and axioms about what is true and false at the initial state, using the predicate *initially(.)*.



We give in this slide the domain-independent axiomatisation of Event Calculus. This is a simplified version. More complex axiomatisations have been proposed, but the general principles formalised here are at the core of many axiomatisation of Event Calculus.

The axioms apply to any planning problem domain as they describe general principles of effects of events and persistency of state-based properties.

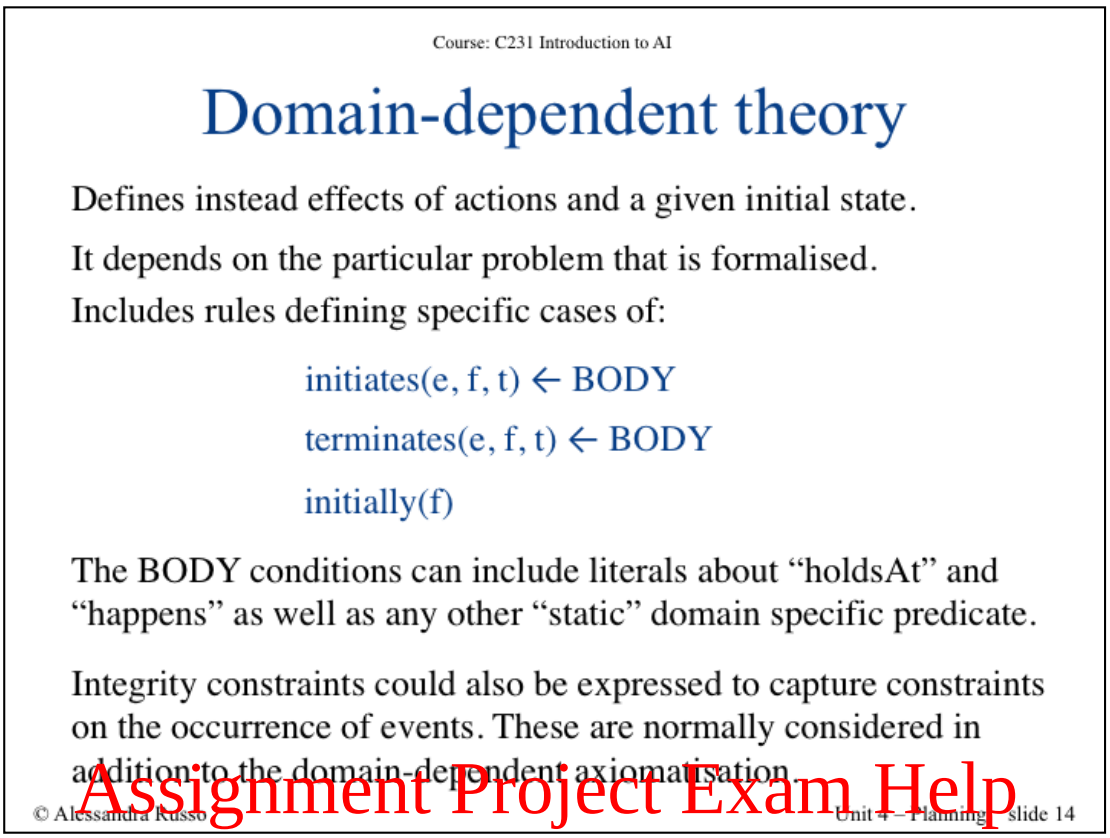
The predicate clipped(.) defined in the third axiom is an auxiliary predicate that expresses the occurrence of an event between two time points that terminates a fluent.

The main characteristic of this axiomatization is that it does not explicitly define when fluents are false. The "holdsAt" predicate refers only to fluents being true at a particular time point. This is because the above axiomatization assumes a close world assumption. Any fluent that cannot be proved to be true at a given time point is assumed to be false. This close world assumption plays an important role in the definition of this simplified EC axiomatisation. In general, a more elaborated axiomatisation would be needed that expresses when a fluent is defined to be true and also when it is defined to be false. The falsity of a fluent can be defined with an axiom that is symmetric to axiom (ii), and that uses instead a second auxiliary predicate called "declipped".

Below is an example of the additional axioms that would be needed to define the falsity of fluents:

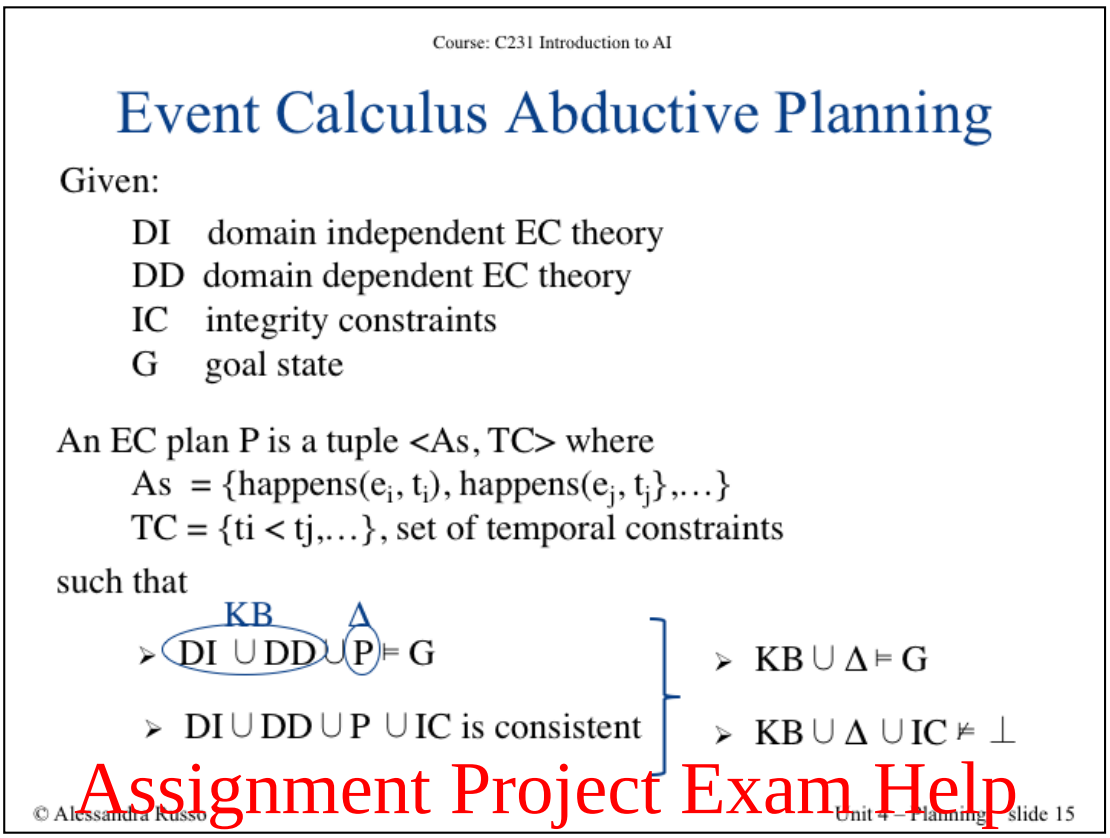
$$\text{holdsAt}(\text{neg}(f),t2) \leftarrow \text{happens}(a,t1), \text{terminates}(a,f,t1), t1 < t2, \text{not declipped}(t1,f,t2)$$

$$\text{declipped}(t1,f,t) \leftarrow \text{happens}(a, t2), \text{initiates}(e, f, t2), t1 < t2, t2 < t$$



The second component of an EC formalisation of a planning problem description is the domain-dependent axiomatization. This normally includes:

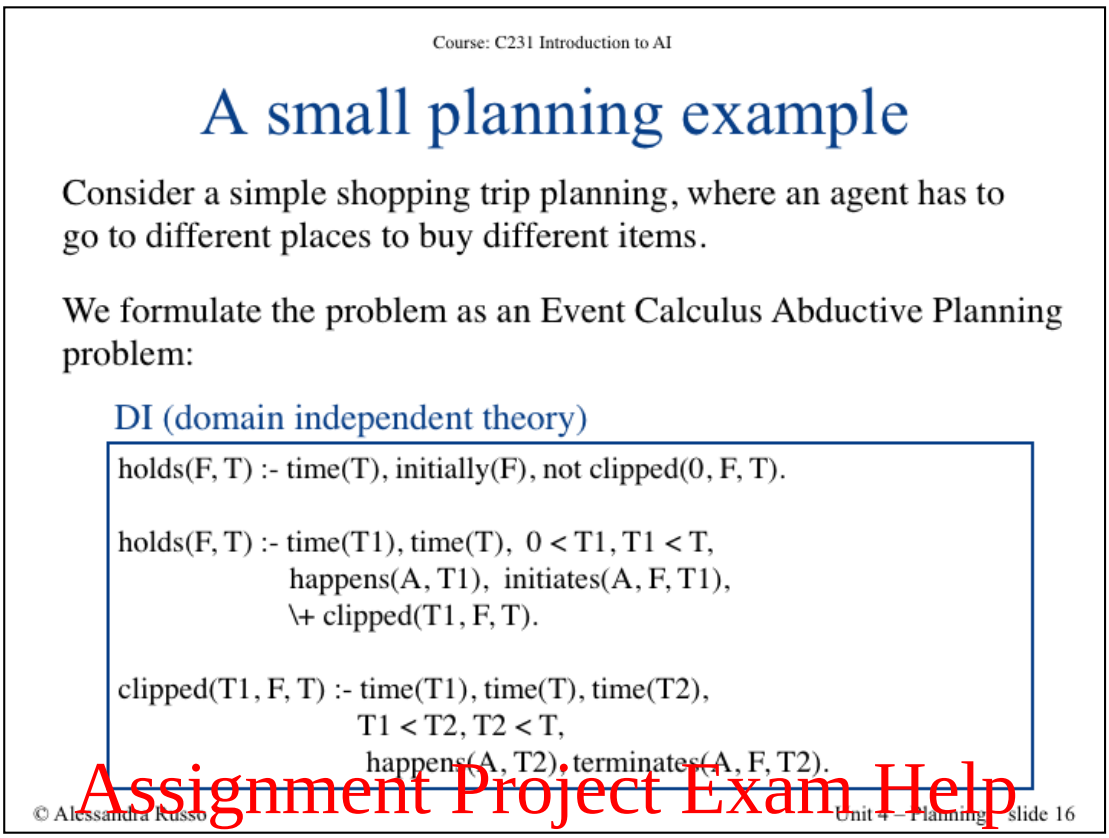
1) set of rules that specifies the effect of individual actions on state-based fluents. Effects of actions may depends on the current state as well as the occurrence of other actions. One possible assumption that we can make is that only one action can occur at a given time. In this case, effects of actions may simply depend on what is true or false at the current time point when the action is applied. The effect of actions is captured by the definition of *initiates* and *terminates* predicates. The body of these rules can therefore include literals with predicates "holdsAt".
2) definition of the initial state. This is a set of facts fully defining the fluents that are initially true. Fluents that are false at the beginning do not need to be specified, since, by the close world assumption, what is not true is implicitly assumed to be false.
3) integrity constraints on the occurrence of actions may also be included in the domain dependent axiomatisation.



Given an Event Calculus formalisation of a planning problem, and a goal, it is possible to define the notion of a plan in terms of ground literals of events that lead the goal, and temporal constraints that these events have to satisfy.

The EC formalisation may also include integrity constraints on the occurrence of events. In this case, the set of events that constitute a plan, together with the EC theories, will have to be consistent with the given integrity constraints.

It is easy to see that the conditions that a plan has to satisfy matches the conditions of an abductive solution, where the knowledge base is given by the two theories of domain independent and domain dependent EC axiomatisation. An EC planning problem can therefore be reduced to an abductive task where the set of abducibles is given by the set A = {happens(.), < (.)}. The "<" predicates are over time variables.



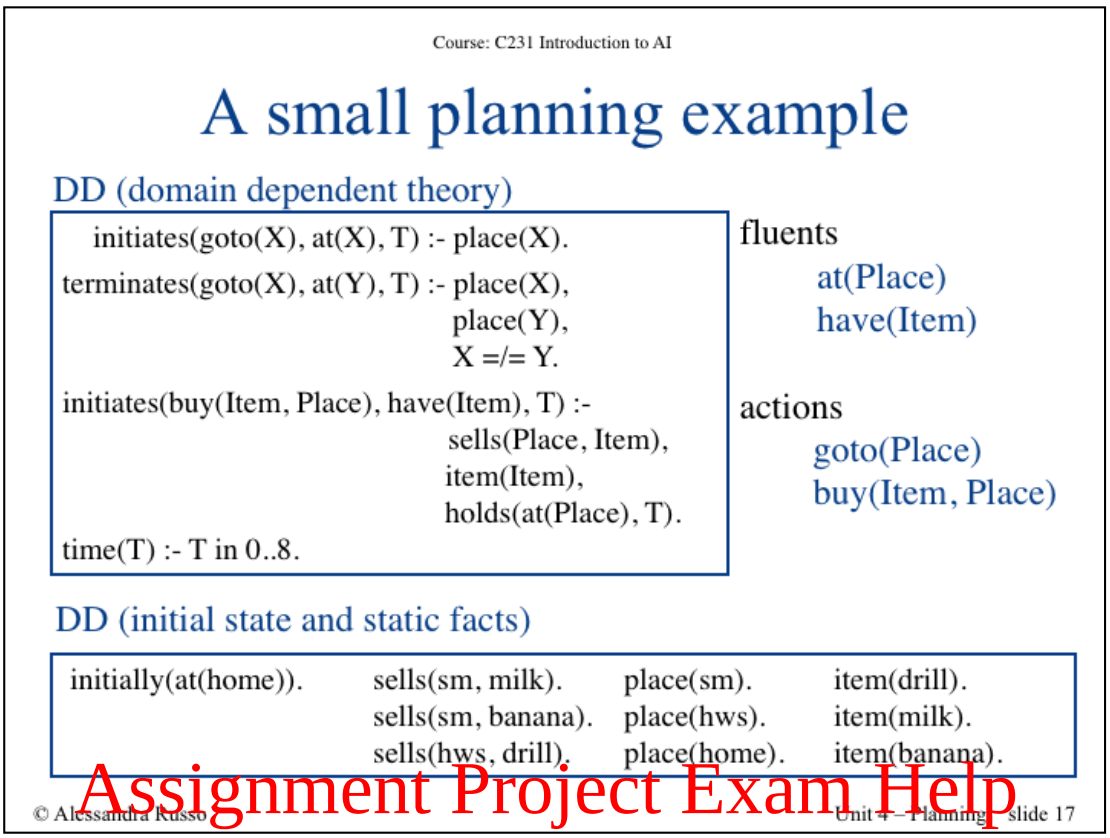
We consider here a simple shopping trip problem where an agent has to plan a trip to go to different places in order to buy different items. This example is extracted from the Russell&Norvik textbook, where it is instead expressed using the STRIPS language.

We show here how we can instead formulate it as an Event Calculus Abductive Planning problem and solve it using a general purpose abductive proof procedure.

Three main components need to be defined:

1) **Domain Independent Event Calculus theory**. The axioms given in slide 13 are represented here. We will consider this simplified version of DI axiomatisation of Event Calculus in our course.

Note that the axiomatization given in this slide includes a predicate “time”, to define the sort of type time, and the predicate “<”. This is a “relational operator” for expressing constraints over finite domains. The variables are in this case of type integer and constraints over integer domains can be expressed using the following relational operators: =; =/=; <; =<; >=; >.



The second component includes axioms about effects of actions over fluents. We assume that the problem includes two fluents "at(place)" and "have(item)", and two actions "goto(place)", "buy(item, place)".

Domain Dependent Event Calculus theory given in this slide defines the effects of these two actions on the two fluents. Specifically, going to a place initiates the agent being at that place, and terminates the agents begin at another place (first two axioms). Also, buying an item at a place initiates the agent having that item as long as the place sells the item and the agent is at that place.

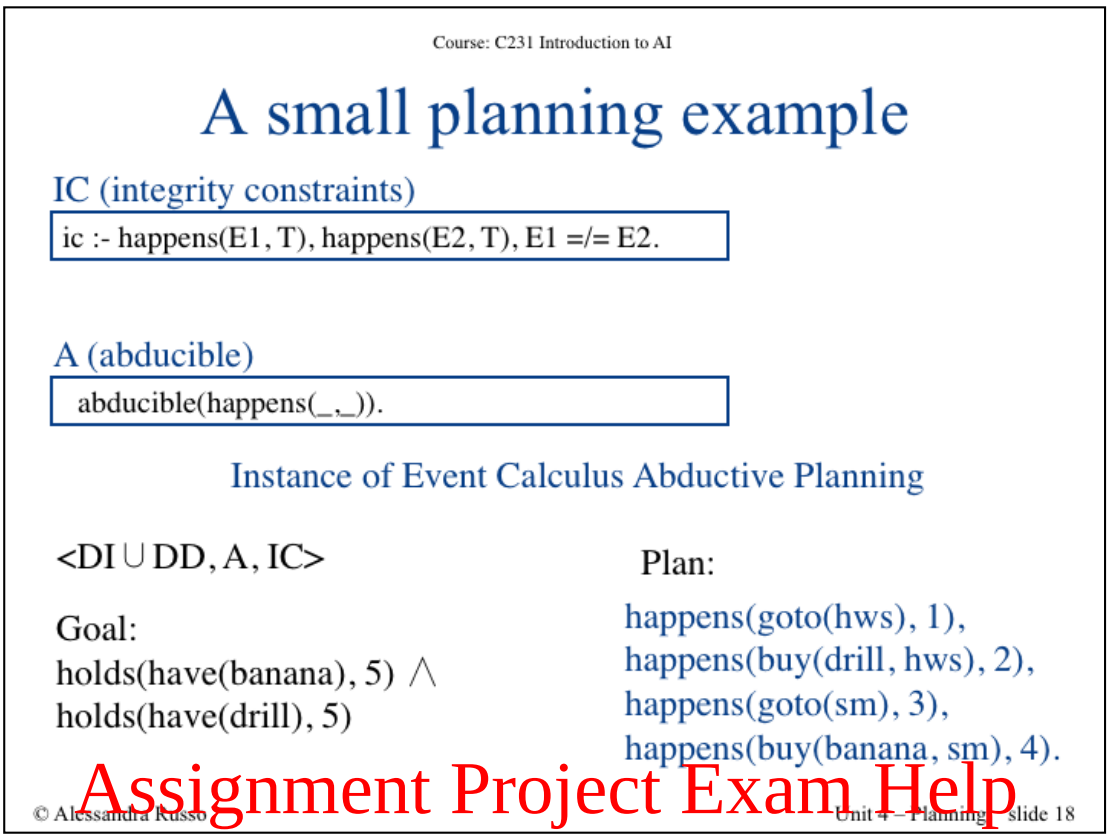
You could easily see how the above axiomatisation corresponds to a STRIPS representations:

goto(x, y)
Precond: at(x)
Effect: ¬at(x) ∧ at(y)

buy(x, store)
Precond: at(store) ∧ sells(store, x)
Effect: have(x)

where the preconditions are captured in the conditions in the "initiates" axioms and the effects are expressed by the head atoms of the "initiates" and "terminates" rules.

The axiomatisation given in this slide includes also a finite domain membership constraint of the form X in Min..Max, that specifies the range of the type "time" used in the specification.

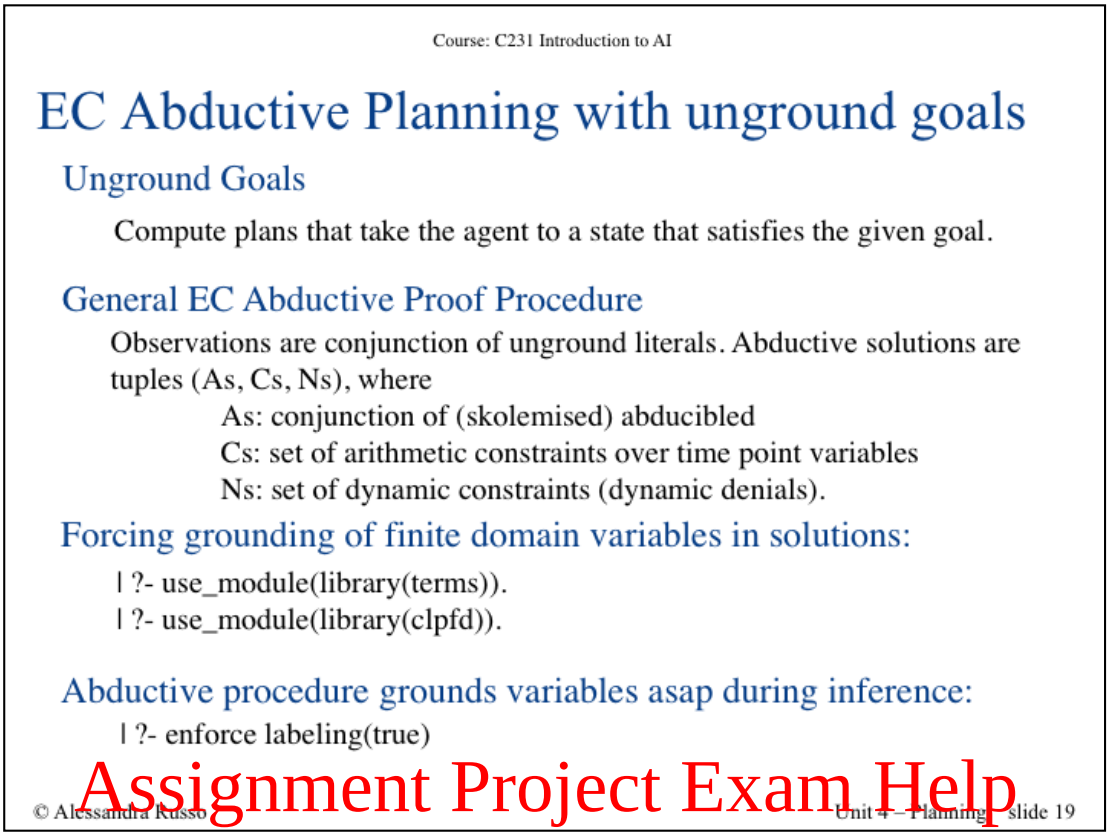


The third component includes **domain specific integrity constraints**. These are effective when the include abducibles, i.e. they express constraints on the possible events that can be assumed to happen as part of a plan. In our example, we have only one general integrity constraint, expressing the fact that only one event can happen at each time.

Abducibles are instances of the predicate "happens", and this predicate is included in the given integrity constraint.

The full Event Calculus Abductive Planning task is then defined by the tuple <DI ∪ DD, A, IC> defined so far, and a given goal. An example of a goal is given in this slide.

Ab abductive solution would be in this case be a sequence of events that takes the agent from the initial state to a state at time point 5 where the agent has both the banana and the drill, as required by the given goal.



The example of EC Abductive Planning that we have considered in the previous slides uses ground goals and plans are computed so that specific goal states are reached. In real word applications it is not easy to predict the specific time points in which given goals states are satisfied. A more natural plan query would be to ask if there is a plan that would take at some time the agent to a state satisfying certain properties. This type of goal is normally a conjunction of literals with unground time variable. To compute plans the abductive procedure would have to reason about unground observations.

In the previous lecture we have considered ONLY the abductive proof procedure for ground goals. But more general abductive procedures exist that take goals which are unground. These more general algorithms are outside the scope of this course and therefore not examinable, but a brief summary is given here to allow you to use the abductive system that is available in CATE to solve planning problems. This is because the abductive procedure in CATE is a more general and powerful abductive engine.

When unground goals are considered to in a given EC Abductive Planning problem the reasoning about time variables has to keep track of the ordering relations between (skolemised) time variables. The more abductive proof procedure generates also dynamic constraints to keep try the consistency among unground (negated) literals. The output of the abductive procedure in this case is a tuple:
(As, Cs, Ns), where As is the list of abduced events; Cs is the list of arithmetic constraints over the time variables that appear in the abduced events and Ns is the list of dynamically generated constraints, called **dynamic denials**. These are of the form: *fail(ListOfUniversalVars, ListOfGoals)*. For example the denial constraint *fail([X], [p(X, Y), q(X), r(Y)]), should* be read as
$\exists Y \ \forall X. (\bot \leftarrow p(X, Y), q(X), r(Y))$. All variables in the list of variables are assumed to be universally quantified and any other variable that appear in the constraint literals are existentially quantified.

Consider, for instance, the same example of planning problem given in the previous slide. And let's consider the goal "G = holds(have(banana), T) $\wedge$ holds(have(drill), T).", which means asking for a plan that would take the agent to a state where it has bought banana and a drill. A plan solution would be "*happens(goto(hws),_A),happens(buy(drill,hws),_B),happens(goto(sm),_C),happens(buy(banana,sm),_D)*" with arithmetic constraints including Cs = [_D#<T, _B#<_D, _B#<_C, _B#>=_A, _D#>=_B, _A#<_D, _C#>=_B, _A#<_C, _A#<_B…], with _D in 4..7, T in 5..8, _C in 3..6, _B in 2..5, _A in 1..4.

Conclusion

- Notion of planning
- Forward planning
- Regression planning
- Event Calculus
- Event Calculus Abductive Planning