# Planning as Model Checking

*AIPS'02 Tutorial*

Piergiorgio Bertoli    Marco Pistore    Marco Roveri

[bertoli,pistore,roveri]@irst.itc.it

ITC-IRST, Via Sommarive 18, 38055 Povo, Trento, Italy

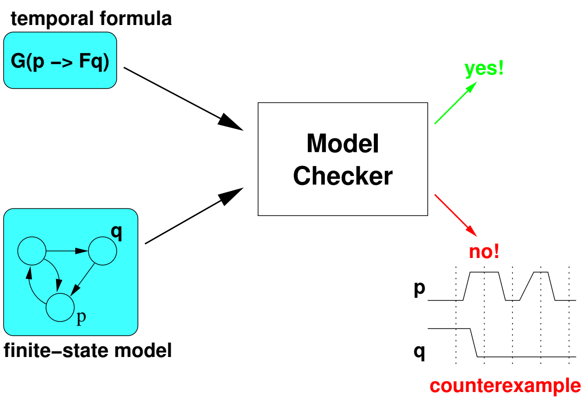


# Introduction



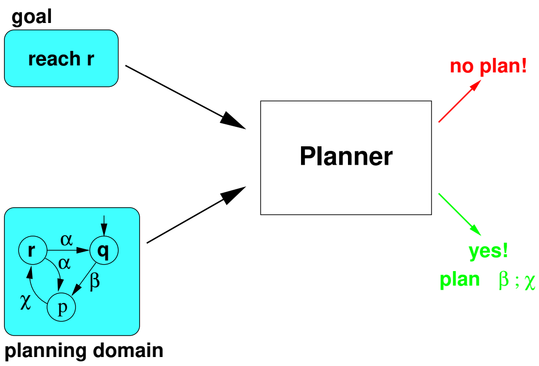
# Model Checking

Model Checking: a technique to *validate* a *formal model* of a system against a *logical specification*.



# Planning as Model Checking

Planning as Model Checking: a technique to *synthesize* a plan from a *formal model of a domain* and a *logical specification of a goal*.






# Planning as Model Checking

Remarkable success of the “Planning as Model Checking” approach:

- Several papers in the most important planning conferences.
- Dedicated sessions on Planning as Model Checking.
- Workshops on this topic (e.g., AIPS’00, IJCAI’01, AIPS’02).
- Wide range of planning systems exploit Model Checking techniques (BDDPlan, CIRCA, MBP, MIPS, Proplan, Simplan, SPUDD, TalPlan, TLPlan, UMOP...).
- Practical application of “Planning as Model Checking” on real problems (design of controllers for autonomous systems, power supply recovery systems,...).



# The “Planning as Model Checking” Tutorial

This tutorial...

- ...is motivated by the strong interest in “Planning as Model Checking”.
- ...is intended to give an overview of “Planning as Model Checking”.
- ...is designed to be *hands-on*: during the hands-on session the participants will confront with the “practical” problems of planning.

In particular:

- we will focus on the approach developed at IRST: use of *Symbolic* Model Checking techniques for planning in non-deterministic domains.
- the MBP planner will be used in the practical sessions.

# Index of the tutorial

1. Introduction.
2. Overview on Planning.
3. Introduction to Model Checking.
4. Planning as (Symbolic) Model Checking.
5. MBP – The Model Based Planner.
6. Hands on!
7. Conclusions.

ITC irst



# Plan

| | |
|---|---|
| 09.00-10.00 | Overview<br>Model Checking<br>An introduction to Planning as Model Checking |
| 10.00-10.30 | Coffee Break |
| 10.30-12.30 | Planning as Model Checking<br>MBP: system architecture & demo |
| 12.30-14.00 | Lunch |
| 14.00-14.30 | nuPDDL: extended planning domain definition language |
| 14.30-15.30 | Assignments & Hands-On, pt.1 |
| 15.30-16.00 | Coffee Break |
| 16.00-18.00 | Hands-on, Pt.2 & Conclusions |

ITC irst



# Acknowledgments

This tutorial is sponsored by PLANET.

Many thanks to the AIPS'02 organizers for the support.

Many thanks to the colleagues at IRST for the helps and the suggestions.

ITC irst

# Overview on Planning



# Planning

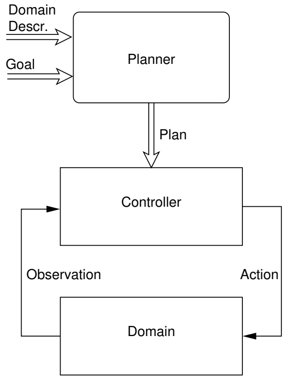
- “Planning” is the problem of building a suitable *plan* for achieving a given *goal* on a given dynamic domain.
- The plan describes the actions to be executed in the domain in order to achieve the goal.

Different approaches to planning differ for:

- the assumptions on the *domain*.
- the assumptions on the *goals*.
- the assumptions on the *plans*.
- the techniques used in the *planning algorithm*.



# The conceptual model




- the *domain* evolves according to performed *actions*.
- the *controller* provides the *actions* according to the *observations* on the domain and to a *plan*.
- the *planner* synthesizes a *plan* from a *domain description* and a *goal*.

*Thanks to “Automated Task Planning: Theory and Practice”, M. Ghallab, D. Nau, P. Traverso*

# Classical planning: assumptions

*Classical* planning relies on simplifying assumptions:

- *finite*: the domain has a finite set of different states.
- *implicit time*: actions are instantaneous state transitions.
- *deterministic*: the initial state is completely specified; each action, if applicable in a state, brings to a single new state.
- *observable*: the state of the system is fully observable to the controller.
- *basic goals*: goals are sets of target states; the objective is to build a plan that leads to one of the goal states.
- *off-line planning*: the plan is built once and for all, before its execution.
- ...

ITC irst



# Classical planning

- A plan is a *sequence of actions* whose execution leads from the initial state of the domain to a goal state.
  - the evolution is completely determined by the executed actions (deterministic domain).
  - no feedback is needed from the domain (observation is not necessary).
- Main difficulty:
  - in spite of simplifying assumptions, search space is huge for practical domains.

ITC irst



# Non-Determinism

The restriction to *deterministic* domains is unrealistic in many practical cases:

- There may be uncertainty on the *initial state*.
- *Actions* may have *non-nominal outcomes* that are highly critical.
- Actions may have *no nominal outcome* (e.g., throwing a dice).
- There may be *exogenous events* or *non-controllable actions*.

In *non-deterministic domains*:

- More than one initial state is possible.
- An action executed in a state may lead to many different states.

Main difficulties:

- Plans cannot simply be sequences of actions.
- Domain models are more complex.
- Lifting most classical planning techniques does not work in practice.

ITC irst

# Non-determinism: probabilistic outcomes

It is often possible to associate probabilities to the outcomes of actions:

- non-nominal outcomes of actions often have a very small probability.
- when throwing a fair dice, all the outcomes have equal probability.
- exogenous events occurs with a given probability.

In planning with probabilistic domains (e.g., MDP-based planning):

- probabilities are associated to the outcomes of actions.
- goals are represented by utility, or value functions.
- planning consists in searching for a plan maximizing the utility function.

Main difficulty:

- *optimality* comes into play.



# Partial observability

In several realistic problems the domain status is only partially visible at run-time to the controller:

- different states of the domain are *indistinguishable* for the controller.
- certain information on the domain is available only after some *sensing actions*.

Particular cases:

- *full observability*.
- *null observability* (conformant planning).

Main difficulty:

- for non-deterministic domains, the controller has a partial knowledge of the domain status.



# Extended goals

The assumption that planning goals are sets of final desired states is unrealistic in many cases:

- *safety conditions* (states to be avoided) may complement the main goal.
- *reactive systems* (infinite plans that react to changes).
- *preferences* among plans and *search control rules* can be specified.

*Temporally extended goals* express conditions on the whole, possibly infinite sequences of states resulting from plan execution.

Main difficulty:

- the plan should trace past history (different “active” goals depending on past events).

# Explicit time

Several planning domains require to represent:

- duration of actions.
- concurrency of actions in accessing resources.
- temporal constraints on the occurrence of events with respect to an absolute time reference.

An *explicit* representation of time is necessary for dealing effectively with these aspects.

Main difficulties:

- planning algorithms have to deal with time and resources.
- time is continuous $\rightarrow$ it cannot be modeled directly in finite-state systems.



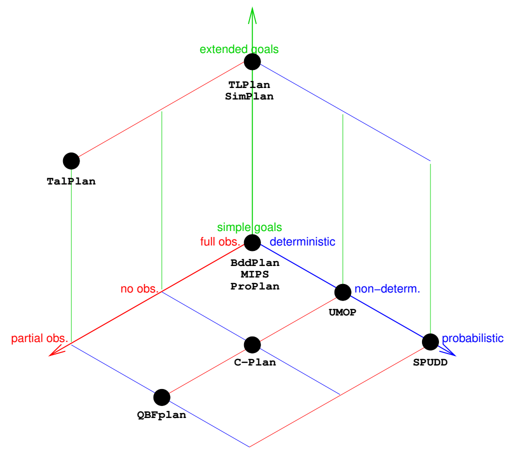
# Plans

Different kinds of plans are necessary for dealing with different assumptions on domains and goals.

- *sequences of actions* are sufficient for *classical planning*.
- *conditional plans* are necessary for dealing with *non-deterministic* domains under *partial observability*.
- *history-dependent* plans are necessary for *extended goals*.
- *explicit time representation* may be necessary in plans.
- ...



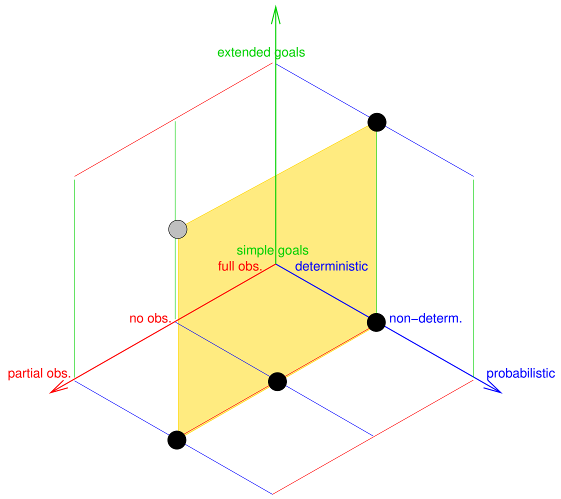
# Planning as Model Checking




Other systems:

- CIRCA: model checking on timed automata.
- ...

# In this tutorial: the MBP system






# A planning system...

...must not be limited to plan synthesis.

- *Plan validation*: verification that a *plan* satisfies a given *property* (not necessarily a goal...).
- *Plan simulation*: interactive simulation of the executions of a *plan* on a given *domain*.
- *Plan synthesis*: automatic generation of the plan from a description of the *domain* and from the *goal*.
- ...



# Introduction to Model Checking

# Talk Overview

Model Checking is a widely used automatic technique for verifying design and for discovering possible bugs.

In this talk . . .

- An introduction to model checking.
- Symbolic representation of FSM based on BDDs.
- An introduction to symbolic model checking algorithms for CTL.



# Model Checking

- Model Checking is a verification technique that solves the problem

$$\mathcal{M} \models \varphi$$

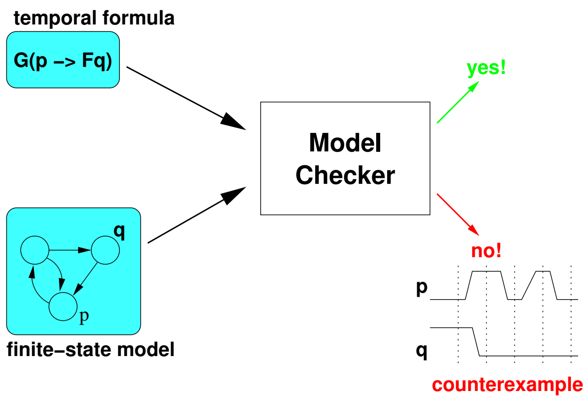
where. . .

- Model $\mathcal{M}$ is represented as a FSM.
- Property $\varphi$ is a temporal logic formula.
- Model checking algorithms traverse the model guided by the property.



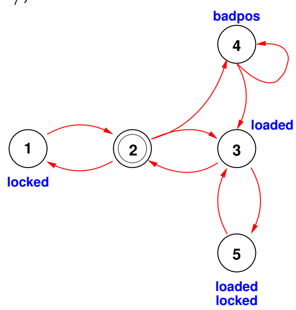
# Model Checker

- Software tool for system verification.



# Formal Model: FSM

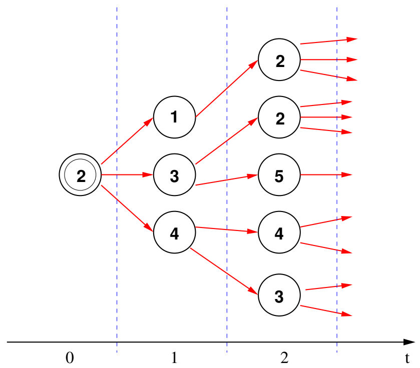
- A model is defined as a tuple $\mathcal{M} = \langle S, S_0, T, P, L \rangle$, where:
  - $S$ is a finite set of states.
  - $S_0$ is the initial set of states.
  - $P$ is a finite set of atomic propositions.
  - $T \subseteq S \times S$ is the transition relation.
  - $L : P \mapsto 2^S$ assigns to each proposition $p \in P$ the set of states where $p$ holds.






# Computation Trees

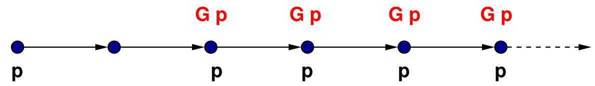
- Computation model is a *tree*:



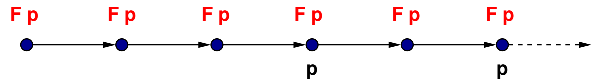



# Temporal Logic: LTL

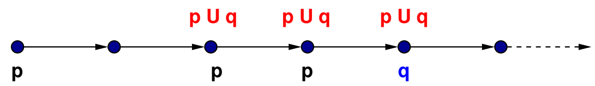
- Express properties of all computation paths.
  - Invariantly $p$: **G** $p$




  - Finally $p$: **F** $p$



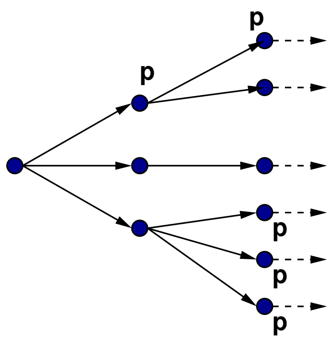

  - $p$ holds until $q$ holds: $p$ **U** $q$



# Temporal Logic: CTL

Express properties on computation paths.

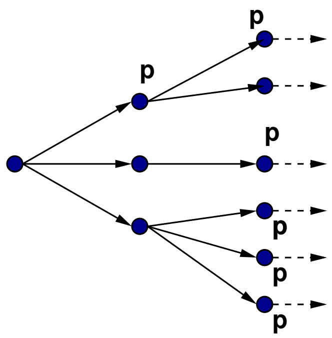
- There exists a path where $p$ finally holds: **EF** $p$






# Temporal Logic: CTL (II)

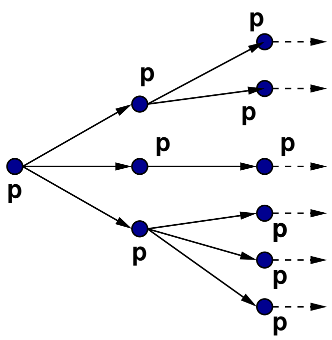
- Property $p$ finally holds for all paths: **AF** $p$





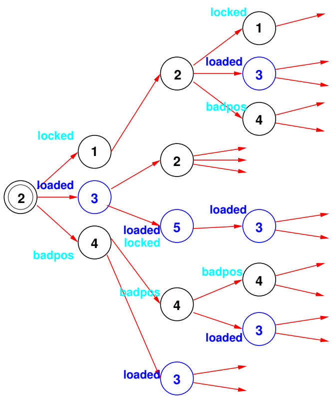

# Temporal Logic: CTL (III)

- Invariant $p$ holds along all paths: **AG** $p$



# CTL Examples: AG EF

- It is always possible to reach a state where *loaded* holds : ***AG EF loaded***






# Symbolic Model Checking

- Symbolic model checking:
  - Face the *state explosion problem* of explicit state model checking.
  - Use the formula $\varphi$ to represent the set of states where $\varphi$ holds.

$$\varphi \rightsquigarrow \{s \mid M, s \models \varphi\}$$

  - Use of Boolean formulas to represent sets and relations.
  - Fix-point characterization of CTL operators.

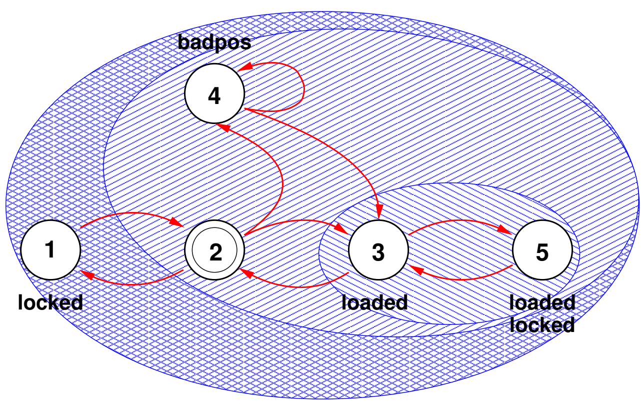
$$\begin{array}{ll}
\mathbf{EF}p \leftrightarrow p \vee \mathbf{EXEF}p & \mathbf{EF}p = \mathit{lfp}_Z\{p \cup \mathbf{EX}Z\} \\
\mathbf{E}[p\mathbf{U}q] \leftrightarrow p \vee \mathbf{EX}(\mathbf{E}[p\mathbf{U}q]) & \mathbf{E}[p\mathbf{U}q] = \mathit{lfp}_Z\{q \cup (p \cap \mathbf{EX}Z)\} \\
\mathbf{AG}p \leftrightarrow p \wedge \mathbf{AXAG}p & \mathbf{AG}p = \mathit{gfp}_Z\{p \cap \mathbf{AX}Z\} \\
\mathbf{EG}p \leftrightarrow p \wedge \mathbf{EXEG}p & \mathbf{EG}p = \mathit{gfp}_Z\{p \cap \mathbf{EX}Z\}
\end{array}$$

$\ldots\ldots$



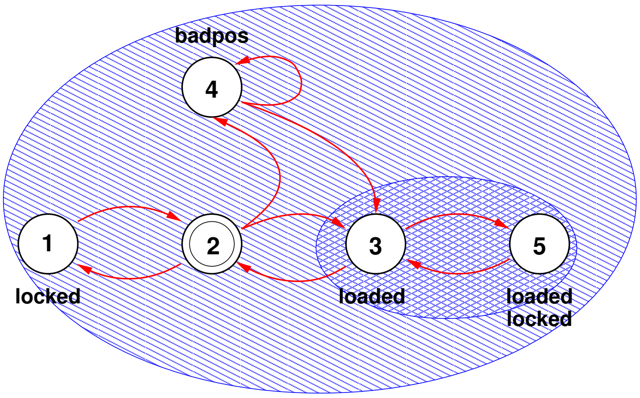
# Fix-point characterization of CTL operators

- $\mathbf{EF}\ \mathit{loaded}$



# Fix-point characterization of CTL operators (II)

- **EG** *loaded*






# Symbolic Representation

- A vector $\boldsymbol{x}$ of Boolean variables where each Boolean variable corresponds to an atomic proposition in $P$.

$$\boldsymbol{x} = \{locked, badpos, loaded\}$$

- A state $s$ is represented with a formula $\xi(s)$ on the propositions:

$$\begin{array}{lll} ① & \rightsquigarrow & locked \wedge \neg badpos \wedge \neg loaded \\ ③ & \rightsquigarrow & \neg locked \wedge \neg badpos \wedge loaded \\ ⑤ & \rightsquigarrow & locked \wedge \neg badpos \wedge loaded \end{array}$$

- A set of states $Q \subseteq S$ represented symbolically as:

$$\xi(Q) = \bigvee_{s \in Q} \xi(s)$$



# Symbolic Representation (II)

- A formula $\varphi$ represents the set of states where the formula holds:

$$\begin{array}{lll} loaded & \rightsquigarrow & \{③,⑤\} \\ \neg badpos & \rightsquigarrow & \{①,②,③,⑤\} \\ loaded \wedge locked & \rightsquigarrow & \{⑤\} \\ badpos \wedge locked \wedge loaded & \rightsquigarrow & \emptyset \\ loaded \vee locked & \rightsquigarrow & \{①,③,⑤\} \end{array}$$

- Set theoretic operations map to propositional operations:

$$\xi(Q_1 \setminus Q_2) = \xi(Q_1) \wedge \neg\xi(Q_2) \qquad \xi(Q_1 \cup Q_2) = \xi(Q_1) \vee \xi(Q_2)$$
$$\xi(Q_1 \cap Q_2) = \xi(Q_1) \wedge \xi(Q_2)$$

# Symbolic Representation (III)

- A new vector $\boldsymbol{x}'$ of Boolean variables to encode next state:

$$\boldsymbol{x}' = \{loaded', locked', badpos'\}$$

- A transition $t = \langle s_s, s_d \rangle$ encoded as

$$\xi(t) = \xi(\langle s_s, s_d \rangle) = \xi(s_s) \wedge \xi'(s_d)$$

$$\langle ①,② \rangle \rightsquigarrow \begin{pmatrix} (locked \wedge \neg badpos \wedge \neg loaded) & \wedge \\ (\neg locked' \wedge \neg badpos' \wedge \neg loaded') & \end{pmatrix}$$

$$\langle ②,③ \rangle \rightsquigarrow \begin{pmatrix} (locked \wedge \neg badpos \wedge \neg loaded) & \wedge \\ (\neg locked' \wedge \neg badpos' \wedge loaded') & \end{pmatrix}$$

- Transition relation $T$ represented symbolically as:

$$\xi(T) = \bigvee_{t \in T} \xi(t)$$



# Symbolic Exploration of the Model

- Images work on sets of states:
  - The forward image $FImg(S)$ of a set $S$ is

$$FImg(S) = \{s' \mid s \in S \wedge \langle s, s' \rangle \in T\} \quad \rightsquigarrow \quad \exists \boldsymbol{x}.(S(\boldsymbol{x}) \wedge T(\boldsymbol{x}, \boldsymbol{x}'))$$

  - The backward image $BImg(S)$ of a set $S$ is

$$BImg(S) = \{s \mid s' \in S \wedge \langle s, s' \rangle \in T\} \quad \rightsquigarrow \quad \exists \boldsymbol{x}'.(S(\boldsymbol{x}') \wedge T(\boldsymbol{x}, \boldsymbol{x}'))$$



# Symbolic CTL model checking

- The set of states where a formula $\varphi$ holds is represented symbolically.
- $M, S_0 \models \varphi$ reduces to $(\neg\varphi \wedge S_0) = \bot$
- Basic CTL operations represented as

$$\mathbf{EX}(\varphi) = \exists \boldsymbol{x}'.(T(\boldsymbol{x}, \boldsymbol{x}') \wedge \varphi(\boldsymbol{x}'))$$
$$\mathbf{AX}(\varphi) = \forall \boldsymbol{x}'.(T(\boldsymbol{x}, \boldsymbol{x}') \rightarrow \varphi(\boldsymbol{x}'))$$

- Fix-point computations as propositional transformations.

$$\textbf{AF}\ \varphi = \mathit{lfp}_Z\{\varphi \cup \mathbf{AX}Z\}$$

$$\begin{aligned} s_0 &= \bot \\ s_1 &= \varphi \vee \mathbf{AX}s_0 = \varphi \\ s_2 &= \varphi \vee \mathbf{AX}s_1 = \varphi \vee \mathbf{AX}\varphi \\ s_3 &= \varphi \vee \mathbf{AX}s_2 = \varphi \vee \mathbf{AX}(\varphi \vee \mathbf{AX}\varphi) \\ &\ldots \\ s_i &= s_{i-1} \end{aligned}$$

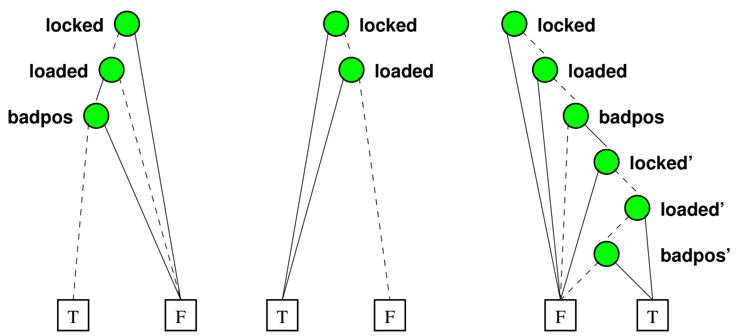
# Binary Decision Diagrams

- The problem in symbolic model checking:
  - The need for efficient and practical representation and manipulation of propositional formulae.
- Binary Decision Diagrams (BDDs).
  - Canonical form for propositional formulae.
  - Efficiently managed by BDD packages.



# Binary Decision Diagrams (II)

{③}  {①,③,⑤}  {⟨④,④⟩,⟨④,③⟩}

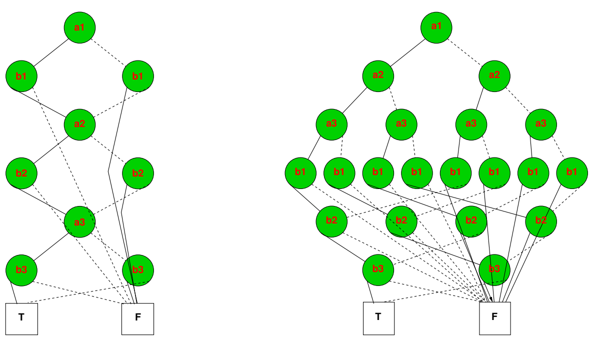





# Binary Decision Diagrams (III)

- BDDs variable ordering is important.

$$(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$$



# Binary Decision Diagrams: Pros and Cons

- Pros
  - Equality test of two BDDs in costant time.
  - Basic propositional operations polynomial on the size of operands.

$$O(\varphi_1 \otimes \varphi_2) = O(|\varphi_1||\varphi_2|)$$

- Cons
  - Quantification exponential in the variables being quantified.
  - BDD size may be exponential in the number of variables.
    - e.g., multiplier.
- They work well in several practical applications.



# Planning as Symbolic Model Checking



# Introduction

The "*Planning as Model Checking*" approach defines a general framework for dealing with:

- non-deterministic domains...
- extended goals...
- partial observability...
- ...and their combination

In "Planning as *Symbolic* Model Checking" BDD-based symbolic techniques are used for:

- a compact representation of domains and plans
- an efficient search in the domain in the planning algorithm

# Talk Overview

In this talk:

1. General framework
2. Reachability goals
3. Extended goals
4. Partial observability
5. Conclusions

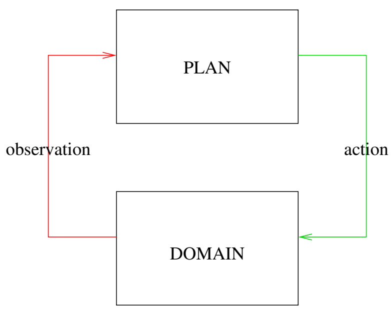


# Planning as Symbolic Model Checking

# General framework

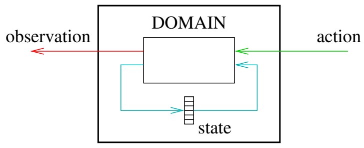


# The general framework




- the *domain* evolves according to performed *actions*.
- the *plan* provides the *actions* according to the *observations* on the domain.

# The model of the domain

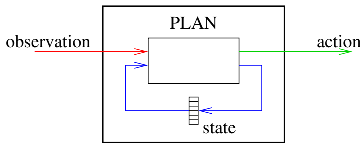



A domain D is a finite-state machine:

- D's input is an *action*.
- D's output is an *observation*.
- D's state represents the *domain state*.
- D's output and next state are non-deterministic.



# Model of the plan

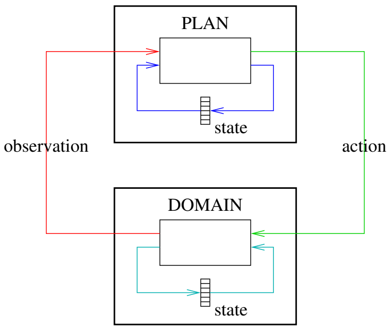



A plan P is a finite-state machine:

- P's input is an *observation*.
- P's output is an *action*.
- P's state represents the *plan state*.
- P's output and next state are deterministic.



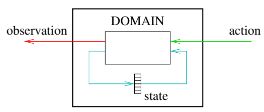
# Plan execution




- The execution of a plan on a domain is the "synchronous execution" of the two machines.
- The state of the execution justaposes the state of the domain and that of the plan.
- Plan validation $\rightarrow$ model checking on execution structures.

# Planning Domain: Formal Definitions

- A *planning domain* is defined as a tuple $\mathcal{D} = \langle S, S_0, A, T, P, L, O, X \rangle$:
  - $S$ is a finite set of states.
  - $S_0$ is the initial set of states.
  - $A$ is a finite set of actions.
  - $T \subseteq S \times A \times S$ is the transition relation.
  - $P$ is a finite set of atomic propositions.
  - $L : P \mapsto 2^S$ assigns to each $p \in P$ the set of states where $p$ holds.
  - $O$ is a finite set of possible observations.
  - $X \subseteq S \times O$ is the observation relation.






# Planning Domains Symbolic Representation

- States represented as in symbolic model checking: $\xi(s)$.
- A vector $\boldsymbol{\alpha}$ of Boolean variables, each corresponding to an action in $A$:

$$\boldsymbol{\alpha} = \{GoWest, GoEast, GoNorth, GoSouth\}$$

- An action represented by assigning *True* to the corresponding Boolean variable:

$$\xi(\texttt{GoWest}) \rightsquigarrow GoWest \qquad \xi(\texttt{GoSouth}) \rightsquigarrow GoSouth$$

- Alternatively, compact logarithmic encoding of actions.
- A transition $t = \langle s_s, a, s_d \rangle$ encoded as

$$\xi(t) = \xi(\langle s_s, a, s_d \rangle) = \xi(s_s) \wedge \xi(a) \wedge \xi'(s_d)$$



# Planning Domains Symbolic Representation (II)

- Observations encoded with a vector $\boldsymbol{o}$ of Boolean variables.

$$\boldsymbol{o} = \{O_{WallN}, O_{WallS}, O_{WallE}, O_{WallO}\}$$

- An observation represented by assigning *True* to the corresponding Boolean variable.

$$\xi(\texttt{O}_{\texttt{WallW}}) \rightsquigarrow O_{WallW} \qquad \xi(\texttt{O}_{\texttt{WallS}}) \rightsquigarrow O_{WallS}$$

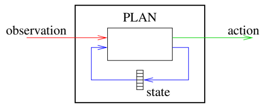
# Symbolic Exploration of the Domains

- Images work on set of states:
  - Forward: set of states reachable from a set of states $S$
    $FImg(S) = \{s' \mid s \in S \wedge \langle s, a, s' \rangle \in T\} \quad \rightsquigarrow \quad \exists \boldsymbol{x}.\boldsymbol{\alpha}.(S(\boldsymbol{x}) \wedge T(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}'))$
  - Backward: set of states from which a set of states $S$ is reachable
    $BImg(S) = \{s \mid s' \in S \wedge \langle s, a, s' \rangle \in T\} \quad \rightsquigarrow \quad \exists \boldsymbol{x}'.\boldsymbol{\alpha}.(S(\boldsymbol{x}') \wedge T(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}'))$



# Plans: Formal Model

- A *plan* $\Pi$ for a planning domain $\mathcal{D}$ is a tuple $\Pi = \langle \Sigma, \sigma_0, \Psi, \Upsilon \rangle$ where:
  - $\Sigma$ is a finite set of *plan states*.
  - $\sigma_0$ is the *initial* plan state.
  - $\Psi : \Sigma \times O \longmapsto A$ associates to a pair $\langle \sigma, o \rangle$ an action $\alpha$ to execute.
  - $\Upsilon : \Sigma \times O \longmapsto \Sigma$ associates to a pair $\langle \sigma, o \rangle$ a new plan state $\sigma'$.




- A configuration for a domain $\mathcal{D}$ and a plan $\Pi$ is a pair $\langle s, \sigma \rangle \in S \times \Sigma$.
- Plan execution is represented with configuration transitions:

$$\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$$



# Planning as Symbolic Model Checking

# Reachability Goals

# Motivations

*Reachability goals*:

- First and “basic” planning problem:
  - find a plan to achieve a desired final situation.
- When tackled within classical planning:
  - solutions are sequences of actions.
  - solutions are guaranteed to achieve the goal.

Adding non-determinism:

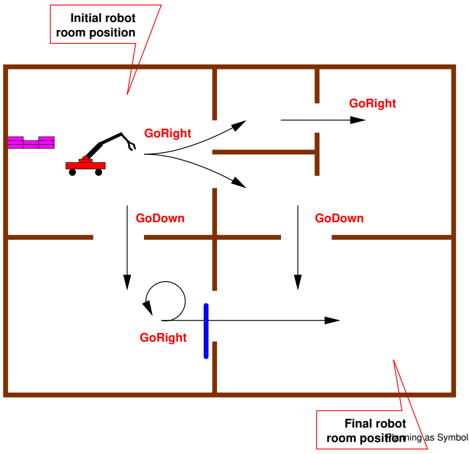
- Plans as sequences of actions are bound to failure.
  - Initial situation might be uncertain.
  - Actions can have non-deterministic effects.
- Plans of different strength, e.g. weak, strong.
- Plans of different structure: sense current state and execute an action.



# Example

The problem . . .

- . . . find a plan from a (set of) initial state(s) to a goal set of states.






# Aim

Objectives:

- Allow for planning under *full observability* in *non-deterministic* domains.
- *Synthesis* of plans of different *strength*.
- Deal *in practice* with *large size domains*.

Problems:

- How can we handle non-deterministic actions?
- Which kind of plans must be generated?
- Which planning algorithms?
- How can planning algorithms handle large domains?

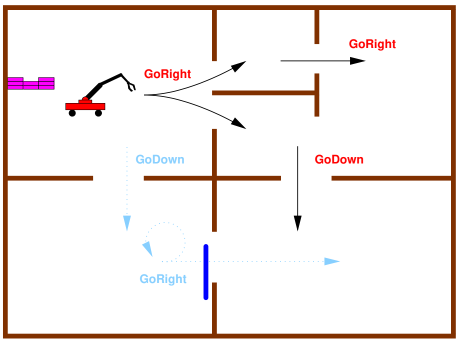
# The PMC approach for Reachability goals

- Solution plans encode conditional behavior based on domain state.
- Planning algorithms exploit . . .
  - symbolic representation of the domain.
  - symbolic representation of solution plans.
- Experimental results show that algorithms work in practice.



# Example

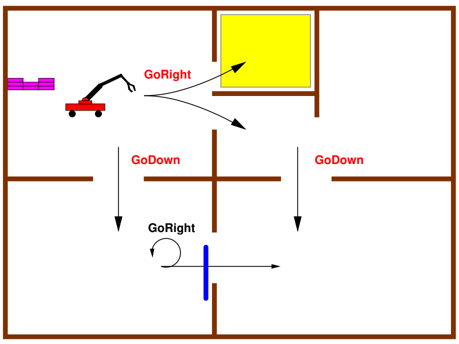
- Strong Solutions: plans that are guaranteed to reach the goal.
  - All executions traces are acyclic and reach the goal.





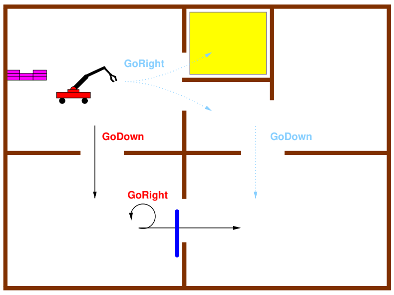

# Example

- Weak Solutions: plans that may achieve the goal.
  - At least one execution trace reaches the goal.



# Example

- Strong Cyclic Solutions: trial and error strategies.
  - The goal is reachable from all states of the execution traces.
  - Solutions are guaranteed to reach the goal under the assumption of "*fair*" execution.






# State-Action Tables

- Solutions are memory-less plans that map states to actions to execute.
- Such solutions can be represented as "state-action tables" (SA).

| State | Action |
|---|---|
| R1 | GoUp |
| R2 | GoRight |
| R3 | GoDown |
| R4 | GoRight |

- State-action tables map in a general plan...
  - ... without plan states.
  - ... switching on the current state.



# Symbolic Representation of SA Tables

- A state-action pair $p = \langle s, a \rangle$ is represented symbolically as:

$$\xi(p) = \xi(s) \wedge \xi(a)$$

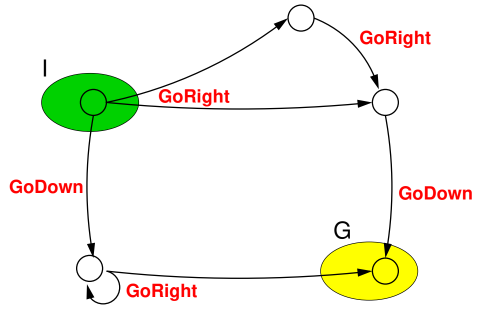
- A state-action table $\pi$ is encoded as symbolically as:

$$\xi(\pi) = \bigvee_{p \in \pi} \xi(p)$$

- Set operations on state-action tables performed as:

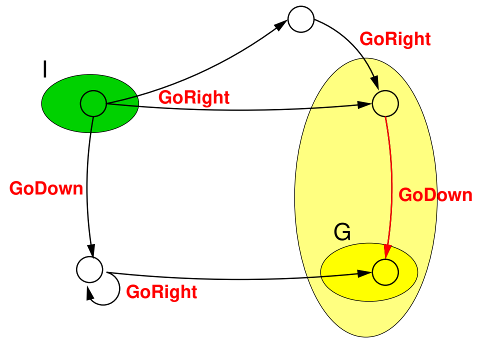
$$\xi(\pi_1 \setminus \pi_2) = \xi(\pi_1) \wedge \neg\xi(\pi_2) \qquad \xi(\pi_1 \cup \pi_2) = \xi(\pi_1) \vee \xi(\pi_2)$$
$$\xi(\pi_1 \cap \pi_2) = \xi(\pi_1) \wedge \xi(\pi_2)$$

# STRONGPLAN: Algorithm intuition



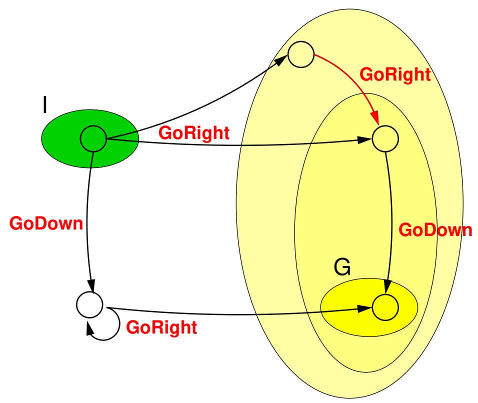



# STRONGPLAN: Algorithm intuition

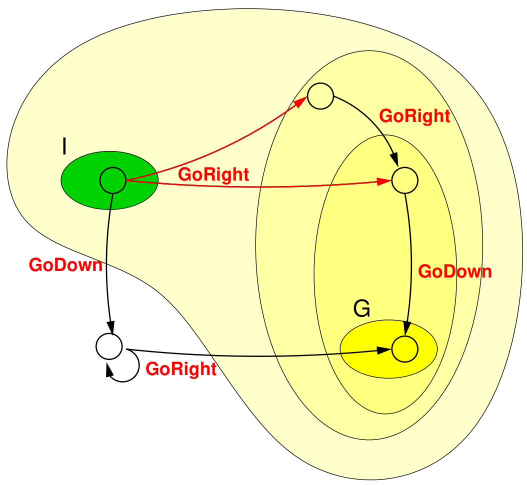





# STRONGPLAN: Algorithm intuition

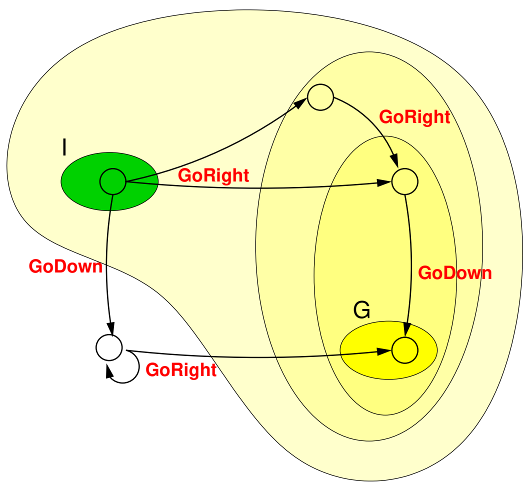


# STRONGPLAN: Algorithm intuition






# STRONGPLAN: Algorithm intuition






# STRONGPLAN: Algorithm

***function*** STRONGPLAN $(I, G)$;
  OSA := FAIL; SA := $\emptyset$;
  ***while*** (OSA $\neq$ SA) ***do***
    PIMG := SPIMG($G \cup$ STOF(SA));
    NSA := PRUNESTATES(PIMG, $G \cup$ STOF(SA));
    OSA := SA;
    SA := SA $\cup$ NSA;
  ***done***;
  ***if*** ($I \subseteq (G \cup$ STOF(SA))) ***then***
    ***return*** SA;
  ***else***
    ***return*** FAIL;
***end***;

***function*** AF $(I, G)$;
  OS := FAIL; S := $\emptyset$;
  ***while*** (OLDS $\neq$ S) ***do***
    PIMG := AX($G \cup$ S);

    OS := S;
    S := S $\cup$ PIMG;
  ***done***;
  ***if*** ($I \subseteq (G \cup$ S)) ***then***
    ***return*** OK;
  ***else***
    ***return*** FAIL;
***end***;

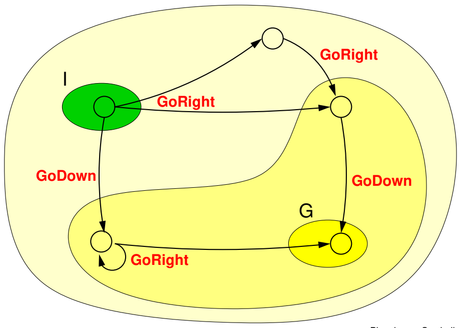
# StrongPlan: properties

- The algorithm terminates.
- The algorithm is correct and complete.
  - Whenever a strong solution exists it finds a state-action table that is a strong solution.
  - Whenever a strong solution does not exist Fail is returned.
- The algorithm computes “optimal” solutions.
  - Optimality defined on the length of the plan execution traces.



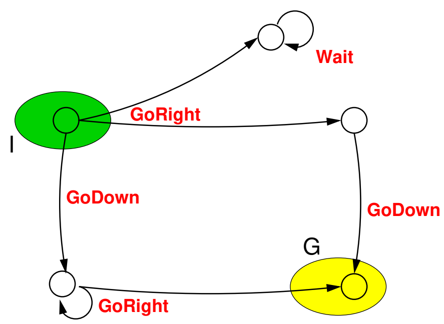
# WeakPlan: algorithm intuitions

- WeakPlan is similar to the StrongPlan algorithm.
  - It performs a *weak pre-image* as basic step.
  - weak pre-image of a set $S$ computes those state-action pairs whose execution may lead to a state $s \in S$.
- WeakPlan similar to the SMC algorithm to compute **EF**.

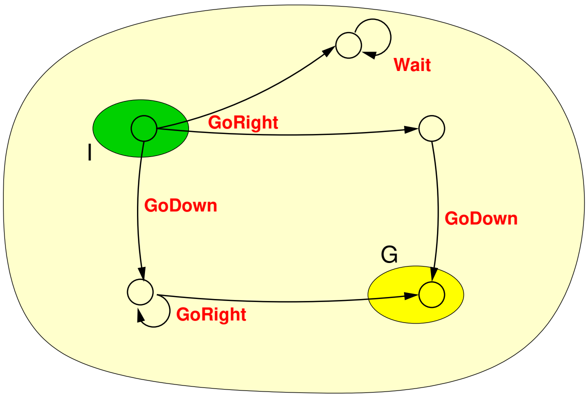





# StrongCyclicPlan: algorithm intuition

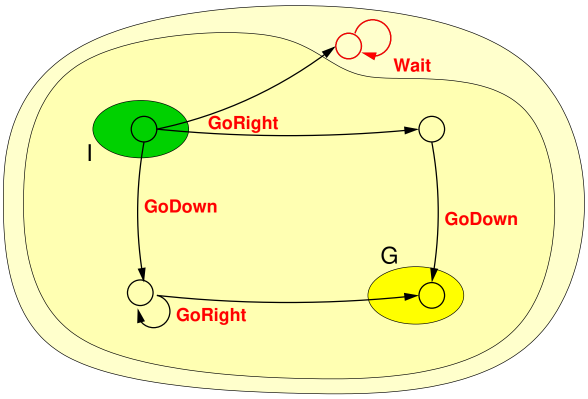


# STRONGCYCLICPLAN: algorithm intuition



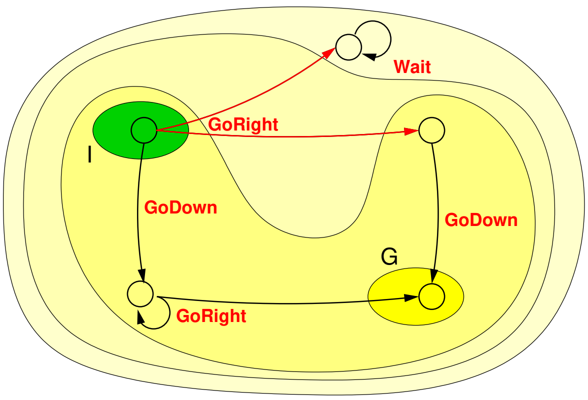



# STRONGCYCLICPLAN: algorithm intuition

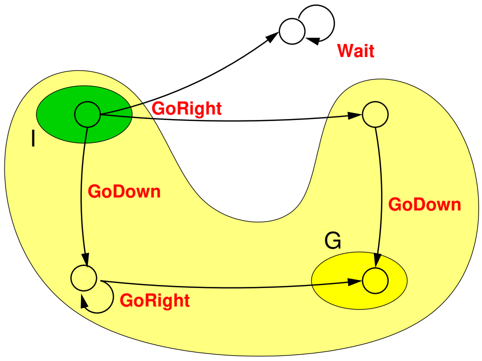





# STRONGCYCLICPLAN: algorithm intuition



# STRONGCYCLICPLAN: algorithm intuition






# STRONGCYCLICPLAN: algorithm

***function*** STRONGCYCLICPLAN $(I, G)$;
  OSA := $\emptyset$;
  SA := UNIVSA;
  ***while*** (OSA $\neq$ SA) ***do***
    OSA := SA;
    SA := PRUNEUNCONNECTED(PRUNEOUTGOING(SA,$G$),$G$);
  ***done***;
  ***if*** $(I \subseteq (G \cup$ STOF(SA))) ***then***
    ***return*** REMOVENONPROGRESS(SA,$G$);
  ***else***
    ***return*** FAIL;
***end***;



# STRONGCYCLICPLAN vs AG EF

- Similarities:
  - Similar meaning (i.e., from all reached states it is possible to achieve the goal).
  - Plan validation performed by verifying **AG EF**$p$ on the synchronous product of domain and plan.
- Differences:
  - The computation of **AG EF** is performed with two *nested distinct* fix point calculations.
  - STRONGCYCLICPLAN *interleaves* the steps of the two fix point calculations.

# STRONGCYCLICPLAN: properties

- The algorithm terminates.
- The algorithm is correct and complete.
  - Whenever a strong cyclic solution exists it finds a state-action table that is a strong cyclic solution.
  - Whenever a strong cyclic solution does not exist FAIL is returned.



# Planning as Symbolic Model Checking

# Extended Goals

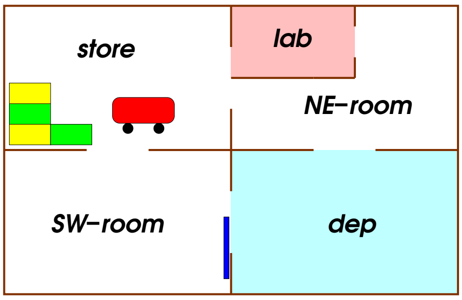


# Motivations

The main motivations for introducing extended goals are:

- *safe planning*:
  - safety conditions (“avoid dangerous states”) complement the main goal.
- *planning for reactive systems*:
  - infinite plan that reacts to events in the environment (mail delivery, elevator system, ...).
- *non-determinism*:
  - need to express (reachability/maintainability) goals of different strength.

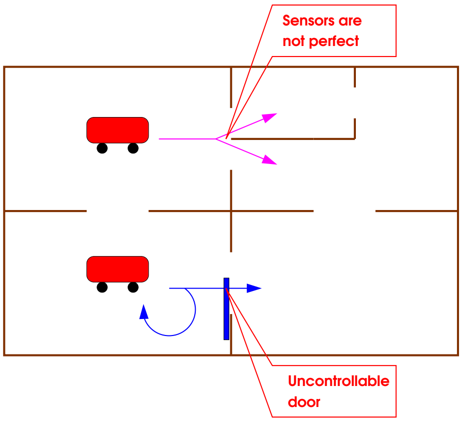
# Example





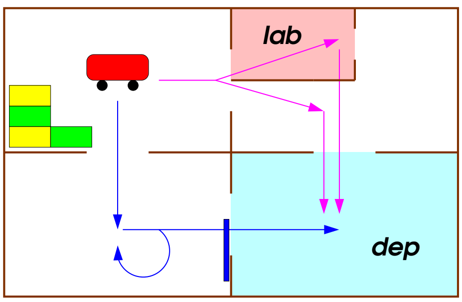

# Example (II)

- Non-determinism: actions may lead to many different states.






# Example (III)




Goal "**Reach *dep* and avoid *lab***"

- "**Do reach *dep* and do avoid *lab***" is *unsatisfiable*.
- "**Do reach *dep* and try avoid *lab***" is satisfiable by route →.
- "**Try reach *dep* and do avoid *lab***" is satisfiable by route →.

# Aim

Objectives:

- Planning for *extended goals*...
- ... in *non-deterministic domains*.
- Dealing *in practice* with *complex goals* and domains of *large size*.

Problems:

- How can we express extended goals?
- Which kind of plans must be generated?
- Planning algorithms?
- How can planning algorithms deal with large domains?



# The PMC approach for Extended Goals

- Extended goals are formulae in a branching-time temporal logics (e.g., CTL):
  → they express temporal conditions that take into account non-determinism.
- Plans can encode conditional, iterative, and history-dependent behaviors:
  → all these features are necessary for extended goals.
- Planning algorithms use BDD-based symbolic model checking techniques:
  → designed to deal with large state spaces.



# Goal language: CTL

*We propose CTL as the language for expressing extended goals.*

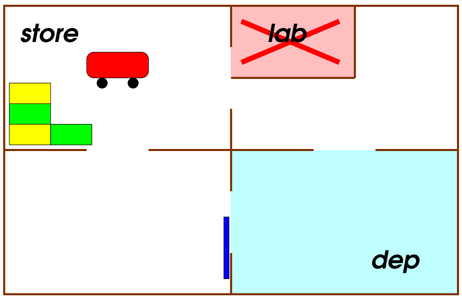
The "**Reach *dep* and avoid *lab***" example:

- **Do reach *dep* and do avoid *lab*** → $\mathrm{AF}\ dep \wedge \mathrm{AG}\ \neg lab$
- **Do reach *dep* and try avoid *lab*** → $\mathrm{AF}\ dep \wedge \mathrm{EG}\ \neg lab$
- **Try reach *dep* and do avoid *lab*** → $\mathrm{EF}\ dep \wedge \mathrm{AG}\ \neg lab$

"Reachability" goals:

- "weak" planning → $\mathrm{EF}$
- "strong" planning → $\mathrm{AF}$
- "strong cyclic" planning → $\mathrm{AG\,EF}$

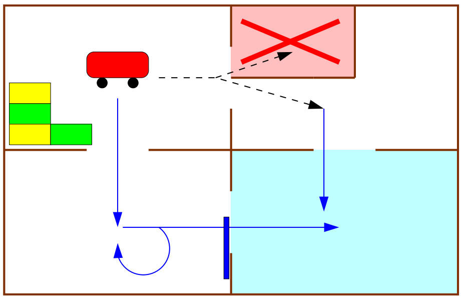
# Plan generation




- The ***lab*** is a dangerous room — it harms the robot.
- The goal is "**Continuously**, **try reach *dep*** and **do reach *store***".
- CTL goal: AG (EF *dep* ∧ AF *store*).



# Plan generation (II)

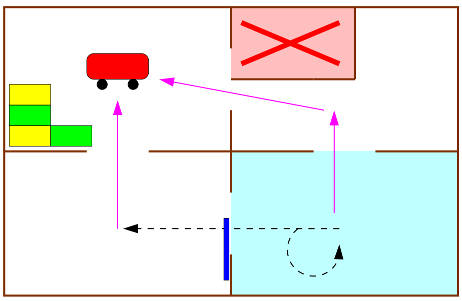


Goal "**Continuously**, **try reach *dep*** and **do reach *store***":

- Satisfying "**try reach *dep***" (EF *dep*). . .



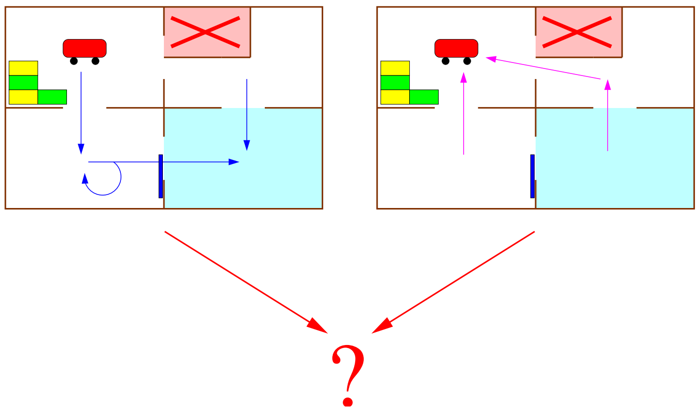
# Plan generation (III)



Goal "**Continuously**, **try reach *dep*** and **do reach *store***":

- Satisfying "**do reach *store***" (AF *store*). . .

# Plan generation (IV)

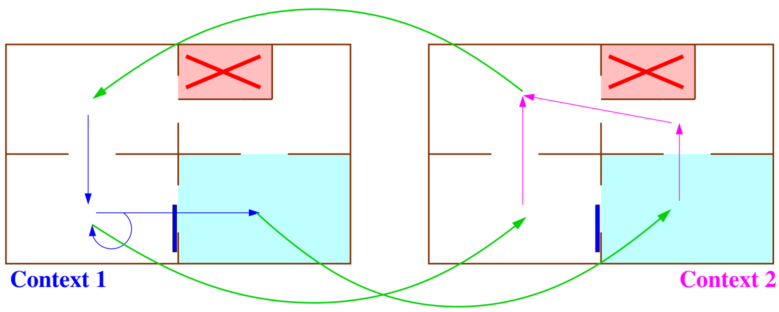





# Plan generation (V)

More *execution contexts* needed for the different intentions of the executor:

- Context 1: "**try reach *dep***".
- Context 2: "**do reach *store***".






# Plans

- A plan for a domain $\mathcal{D}$ is defined by:
  - a set $C$ of (execution) contexts, and an initial context $c_0 \in C$,
  - the action function *act* : $\mathcal{S} \times C \rightharpoonup \mathcal{A}$,
  - the context function *ctxt* : $\mathcal{S} \times C \times \mathcal{S} \rightharpoonup C$.

| state | context | action | next state | next context |
|---|---|---|---|---|
| SW-room | context 1 | go-east | SW-room | context 2 |
| SW-room | context 1 | go-east | dep | context 2 |
| SW-room | context 2 | go-north | store | context 1 |
| ... | ... | ... | ... | ... |

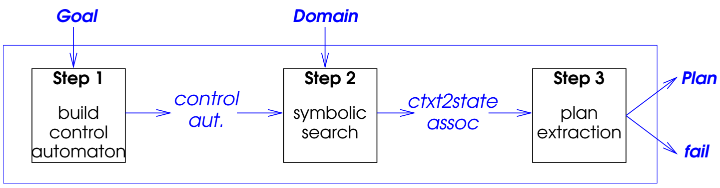
- ... that is, a plan is a FSM (as prescribed by the "general framework").

# Plan validation

- Plan execution:
  - is defined as the *synchronous execution* of the domain and of the plan.
  - the *states* of the execution structure are pairs $(s, c)$, where $s$ is a state of the domain and $c$ is a context of the plan.
  - a *transition* $(s, c) \xrightarrow{a} (s', c')$ exists iff:
    - $a = \textit{act}(s, c)$,
    - $s \xrightarrow{a} s'$,
    - $c' = \textit{ctxt}(s, c, s')$.
- Plan validation:
  - plan $\pi$ satisfies goal $g$ in state $s_0$ iff CTL formula $g$ holds in the initial state $(s_0, c_0)$ of the execution structure.
  - plan validation is reduced to a model checking problem!



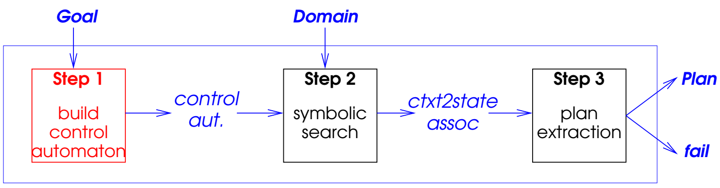
# The symbolic algorithm




1. **Build the control automaton** for the given goal:
   - it represents the possible progresses of the intentions of the executor.
2. **Search in the state space**, guided by the control automaton:
   - finds the sets of states for which a context is satisfiable.
3. **Extract the plan** for the given goal (if it exists):
   - the information on the states associated to the contexts is exploited.



# The symbolic algorithm
# Step 1: build the control automaton




- The control states are the contexts of the plan that is being built.
- The transitions represent the possible evolutions (progresses) of the contexts when actions are executed.

# The symbolic algorithm
# Step 1: build the control automaton (II)

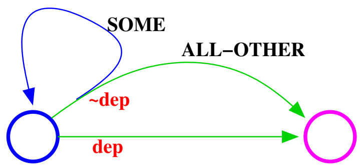
Two contexts are needed for goal $\mathrm{AG}\,(\mathrm{EF}\,dep \wedge \mathrm{AF}\,store)$:

- one corresponding to intention $\mathrm{EF}\,dep$.
- one corresponding to intention $\mathrm{AF}\,store$.

ITC irst



# The symbolic algorithm
# Step 1: build the control automaton (III)



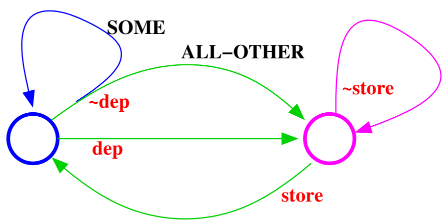

In order to satisfy context $\mathrm{EF}\,dep$, find an action such that:

- if $dep$ holds, then:
  - context $\mathrm{AF}\,store$ is satisfiable for **ALL** the outcomes.
- if $dep$ does not hold then:
  - context $\mathrm{EF}\,dep$ is satisfiable for **SOME** of the outcomes.
  - context $\mathrm{AF}\,store$ is satisfiable for **ALL** the other outcomes.

ITC irst



# The symbolic algorithm
# Step 1: build the control automaton (IV)

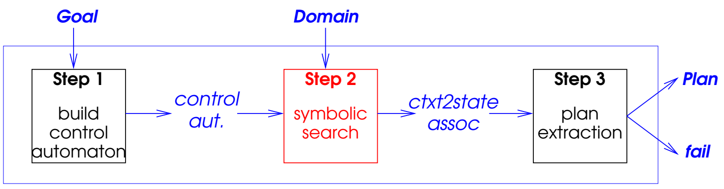



In order to satisfy context $\mathrm{AF}\,store$, find an action such that:

- if $store$ holds, then:
  - context $\mathrm{EF}\,dep$ is satisfiable for **ALL** the outcomes.
- if $store$ does not hold then:
  - context $\mathrm{AF}\,store$ is satisfiable for **ALL** the outcomes.

ITC irst

# The symbolic algorithm
# Step 2: symbolic search

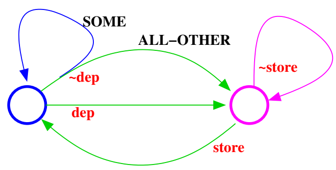


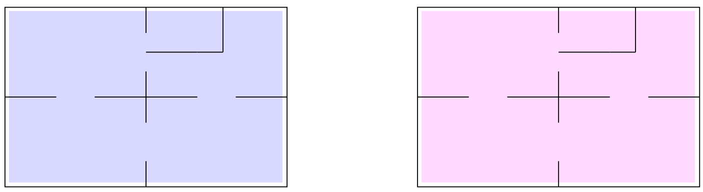

- The states associated to the contexts are obtained via a fix-point computation:
  - least fix-point for reach-like goals.
  - greatest fix-point for maintain-like goals.
- There are mutual dependencies among the contexts:
  - iterative refinements, until a stable association is obtained.
- BDD-based symbolic exploration techniques are exploited



# The symbolic algorithm
# Step 2: symbolic search (II)

- Initially:

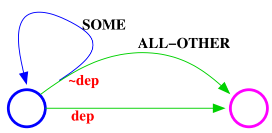


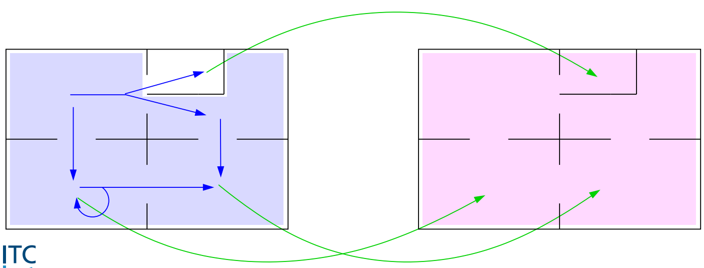





# The symbolic algorithm
# Step 2: symbolic search (III)

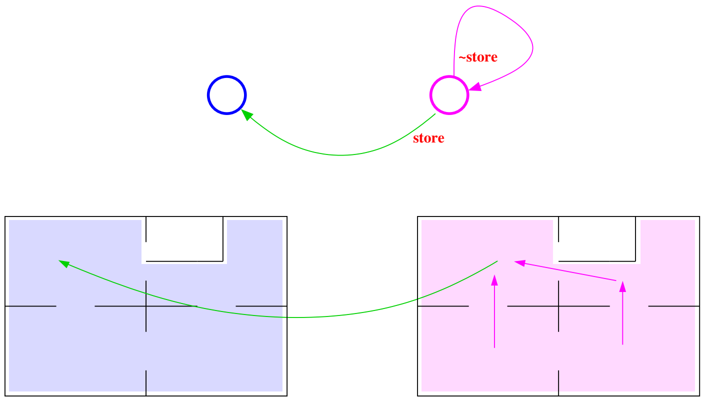
- Refine context EF $dep$:



# The symbolic algorithm
# Step 2: symbolic search (IV)

- Refine context AF $store$:

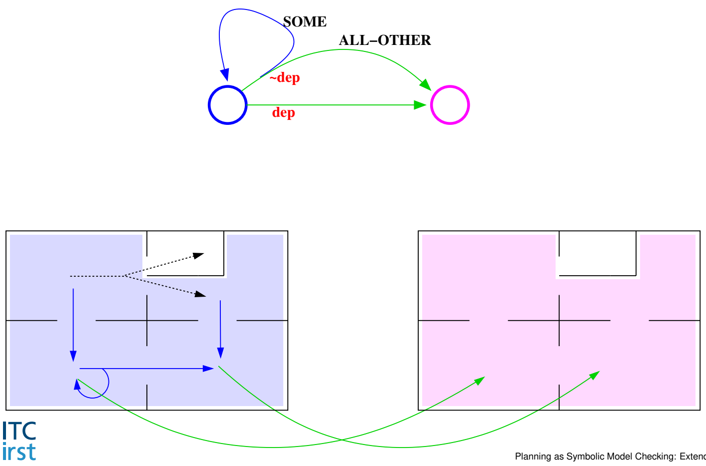



ITC irst



# The symbolic algorithm
# Step 2: symbolic search (V)

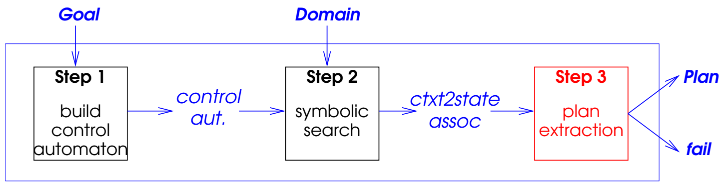
- Refine context EF $dep$:




ITC irst



# The symbolic algorithm
# Step 3: plan extraction

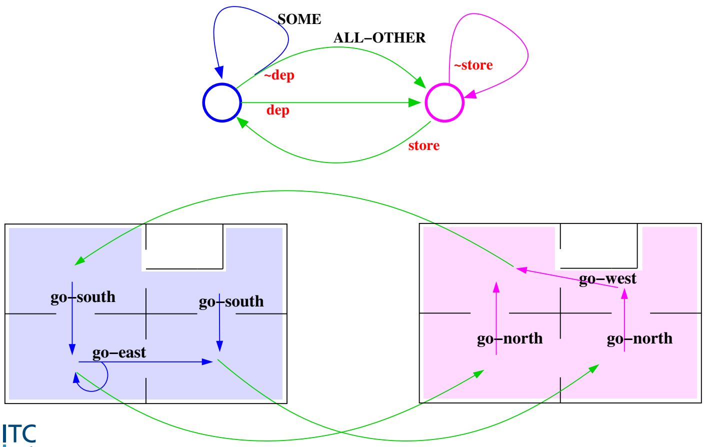



- All the information necessary to extract the plan has been already computed in the search phase.

ITC irst

# The symbolic algorithm
# Step 3: plan extraction (II)

- Find suitable actions:






# Properties of the algorithm

- The algorithm terminates.
- The algorithm is correct and complete:
  - Whenever plans exist, the algorithm finds one.
  - Whenever there is no plan, the algorithm returns FAIL.
- The critical step for performance is "plan search".



# Future work

- Improving the *quality of plans*:
  - rewarding functions, as in MDP-based planning.
- Using extended goals as search *control rules*:
  - control automata define a search strategy.
- Defining a *better language* for goals:
  - see next slides...

# Limits of CTL goals

- CTL goals do not capture *intentionality*:
  - "**Try reach** $p$" is modeled as "$\mathrm{EF}\, p$"...
  - ... but "$\mathrm{EF}\, p$" means "$p$ is reached for some non-deterministic outcomes".
- One has to deal with *failure* of goals:
  - consider goal "**Try maintain** $p$ **Fail Do Reach** $q$"...
  - ... goal "$\mathrm{EG}\, p \vee \mathrm{AF}\, q$" has a different meaning...
  - ... also goal "$\mathrm{AG}\,(\neg p \rightarrow \mathrm{AF}\, q)$" does not work...
  - ... in fact, there is no way for representing the goal in CTL!

ITC irst



# The EAGLE goal language

We are defining EAGLE, a new *Extended Goal Language* that:

- captures the *intentional aspects* of goals.
- can deal with *failure recovery*.

Syntax:

- basic goals: **DoReach** $p$, **DoMaint** $p$, **TryReach** $p$, **TryMaint** $p$.
- conjunction: $g$ **And** $g'$.
- failure: $g$ **Fail** $g'$.
- control operators: $g$ **Then** $g'$, **Repeat** $g$.

Algorithm:

- new algorithm for "control automaton construction".
- "symbolic search" and "plan extraction" can be reused.

ITC irst



# Planning as Symbolic Model Checking

# Partial Observability

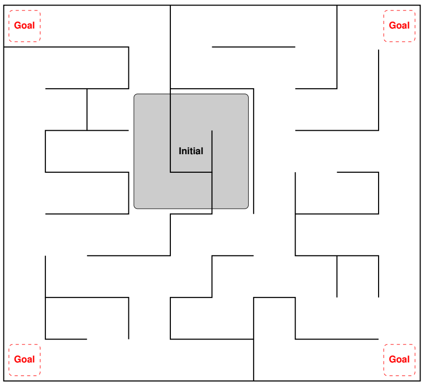
ITC irst

# Motivations

- Realistic:
  - Environment is seldomly fully observable.
  - Some form of sensing is often available.
- General: full observability, conformancy are special cases.
- Related to relevant problems:
  - Diagnosis.
  - Homing/distinguishing problem.
  - Game theory.



# Example: a robot-world with sensing






# Aim

Objectives:

- Allow for planning under *partial observability*...
- in *non-deterministic domains*.
- Deal *in practice* with domains of *large size*.

Problems:

- How can we express partial observability?
- Which kind of plans must be generated?
- Planning algorithms?
- How can planning algorithms deal with large domains?

# The PMC approach for PO

- Partial observability expressed by formulae that encode observation relation.
- Plans encode conditional behavior based on sensing.
- Planning algorithms exploit symbolic representation of observations (as well as actions and states).
- Experimental results show that algorithms work in practice.



# Goals

- Due to nondeterminism and partial observability, the controller has a partial knowledge of the state.
- We consider *strong reachability* goals:
  - At the end of plan execution, the executor *knows* that the goal has been reached...
  - ...in spite of possibly not knowing exactly the state.



# Plans

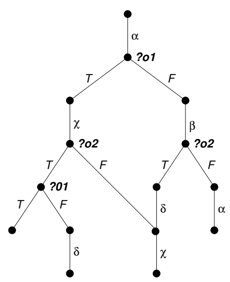
Structure of plans:

- In general: conditional over observations
- Conformant case: sequences
- Full observability case: conditional over states

Expressed as general FSM:

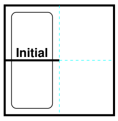
- Plan state: "plan program counter".
- Branching conditions on:
  - "plan program counter".
  - observations.

Validated by model-checking.

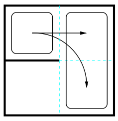


# Plan search: Belief States

- Due to nondeterminism and partial observability, status known to controller can be uncertain:
  - ...because the initial situation is uncertain...




  - ... or because actions may have unpredictable results.

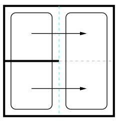


Thus, planning proceeds over sets of states: *belief states (BS)*

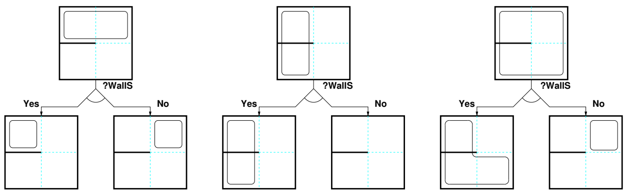


# Plan Search: base steps

- An action transforms a BS into a BS

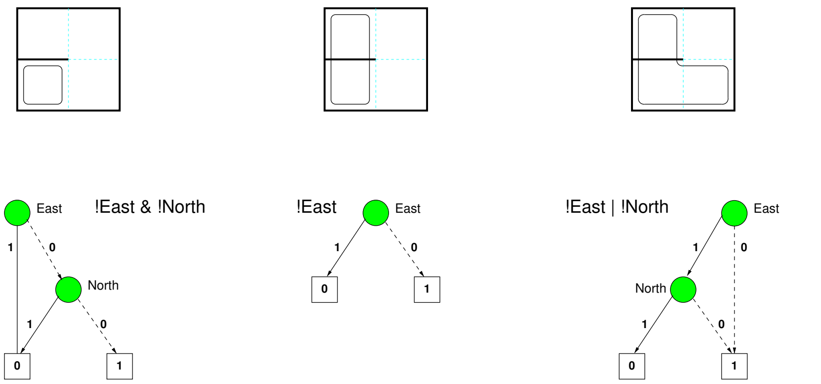


- Observing *may* split a BS

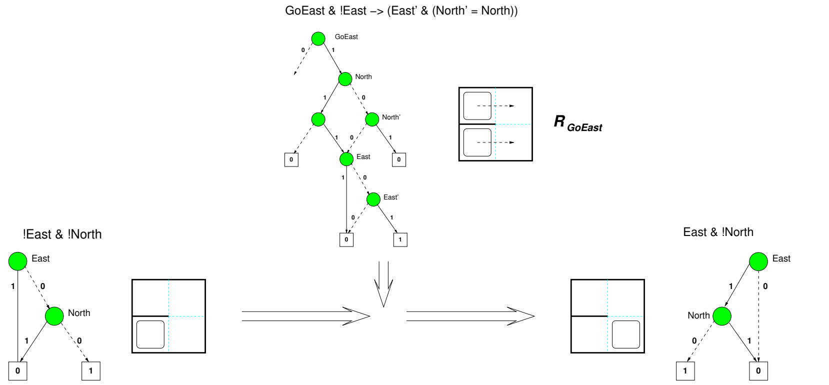





# BDD representation of BS

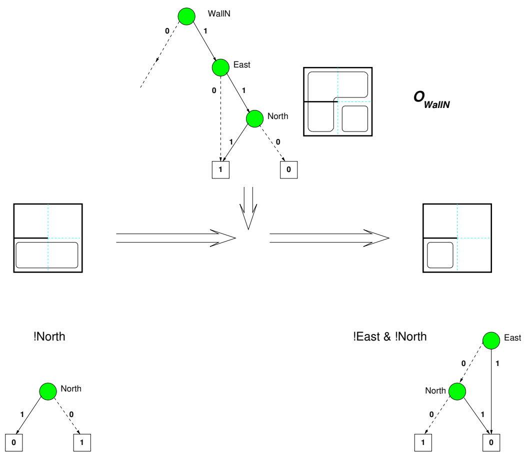


# BDD transformation of BS: acting



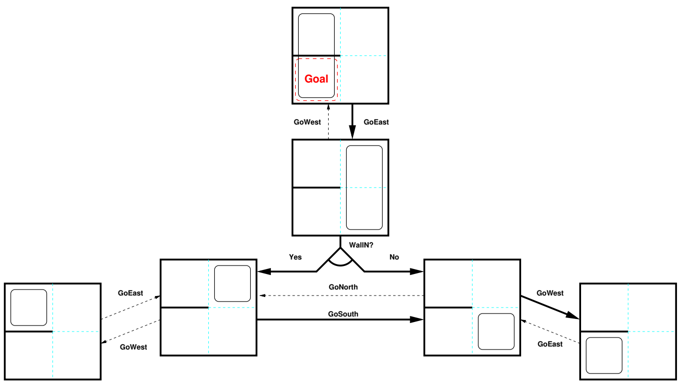



# BDD transformation of BS: observing

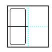





# Search space

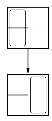


# Search algorithm: a DFS search



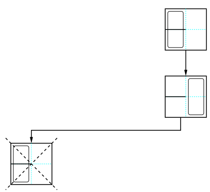


# Search algorithm: a DFS search

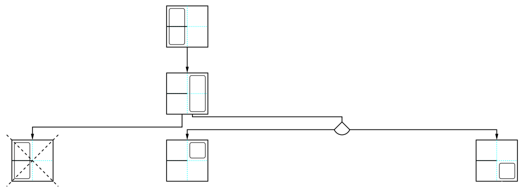




# Search algorithm: a DFS search

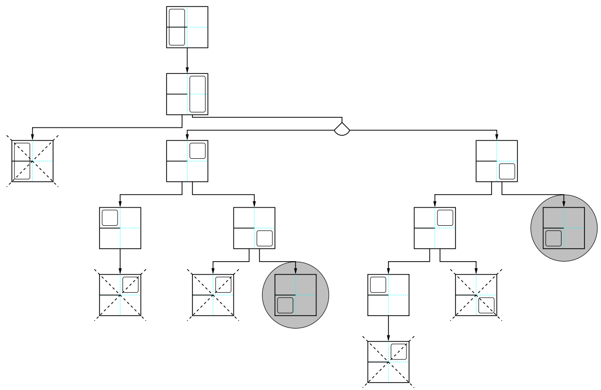


# Search algorithm: a DFS search





# Search algorithm: a DFS search





# Heuristic search

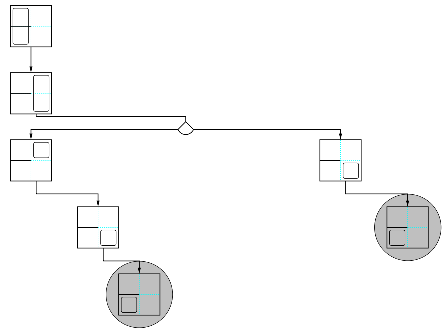
Motivations:

- DFS may produce dumb plans.
- DFS may be not efficient.
- Even simple user-provided heuristics may help a lot.
- Heuristics may be extracted from domain/problem.

Example:

- Favouring observations.
- Avoiding “stepping back”.

# Search algorithm: a Heuristic search





# Conformant algorithms

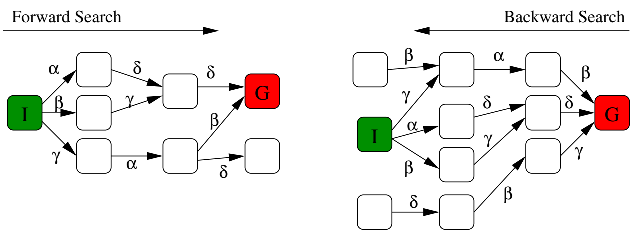
Motivations:

- Relevant applications (e.g. reset sequences).
- Ad-hoc highly efficient planning algorithms.
- Dedicated heuristics and fwd/bwd algorithms, e.g.:
  - breadth-first search in belief space
  - knowledge-driven search
  - ...



# Conformant algorithms (II)

- Search space only features “or” branching.
- BFS search in belief space is feasible...
- ...but it’s not the only way.



# Properties of algorithms

- The algorithms terminate.
- The algorithms are correct and complete:
  - Whenever a strong solution exists they find a plan which is a strong solution.
  - Whenever a strong solution does not exist they return FAIL.



# Future work

- Heuristics in Belief Space.
- Strategies.
- Interleaving planning and execution.
- Diagnosis; homing/distinguishing.
- Realistic case studies.



# Planning as Symbolic Model Checking

# Conclusions

# Combining Extended Goals and Partial Observability

Realistic domains (robot navigation, embedded controllers) combine *partial observability* and *extended goals*.

Challenging problem:

- knowledge goals: introducing "*knows*" in the goal language.
- plan synthesis requires combining plan contexts and belief states.
- no trivial combination of existing PO and EG algorithms.

Work done so far:

- a language for expressing temporally-extended knowledge goals:
  - example: $\mathrm{AG}\, p \wedge \mathrm{AF}\ \mathbf{K}\, q$
- a framework for the validation of plans:
  - a monitor is used for epistemic tracing to trace the belief state.



# Planning in "adversarial" environment

So far we have assumed a "fair" environment:

- all the action outcomes have some chance to occur.

In several applications:

- the non-determinism is due to uncontrollable, adversarial agents:
  - we cannot assume that they will execute actions in a fair way.
  - they can apply a strategy to prevent goal achievement.
- the non-determinism is due to lack of information:
  - some action outcomes may never happen in the "real" domain.

State of the art:

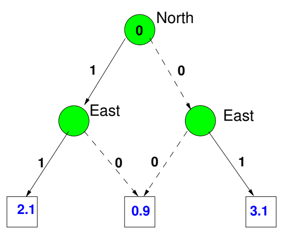
- some forms of *adversarial* planning as model checking addressed by R. Jensen, M. Veloso et al.
- partial observability and extended goals are still open.

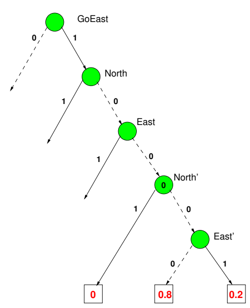


# Planning with probabilities

ADDs (Algebraic Decision Diagrams) can be used for representing:

- expected rewards:
- probabilities of outcomes:




In SPUDD, ADDs are exploited for efficient value iteration in MDP planning.

# Wrap-up

Planning via Model Checking:

- A single, *founded framework* for a variety of planning problems.
- Several specialized algorithms available for plan synthesis.
- Works in practice.
- Plan validation “for free”: Model Checking of plan against domain.



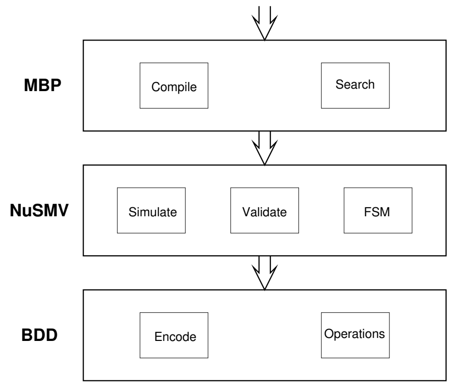
# MBP — The Model Based Planner



# Overview of presentation

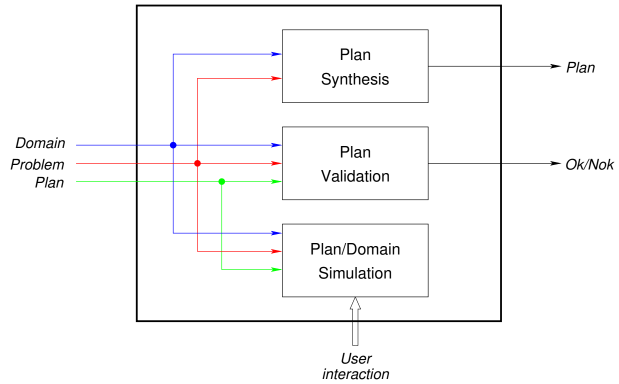
- MBP overview: functions and architecture
- Domain definition language
- Problem definition language
- Plan language
- MBP in action

# MBP: architecture



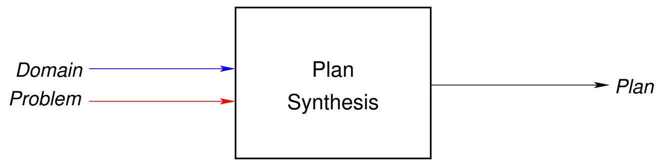



# MBP: main functions



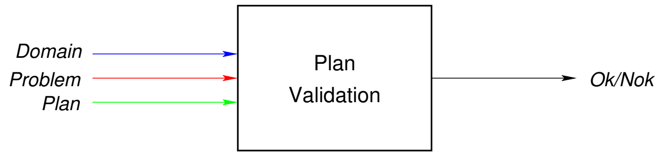



# MBP: plan synthesis




- Input: A domain and a problem.
- Output: a plan (or assessment of nonexistence).
- Plan saved or directly used for validation/simulation.

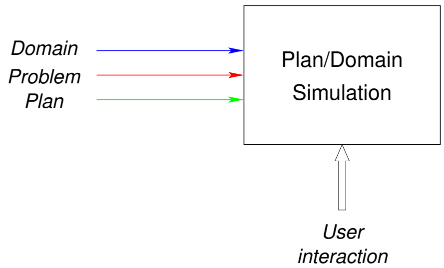
# MBP: plan validation




- Input: a domain, a problem, and a plan.
- Output: yes/no.



# MBP: plan/domain simulation




- Input: a domain, a problem, and a plan.
- Interactive: the user selects action outcomes.
- May terminate with plan success or failure.



# MBP demo

# MBP: languages

Domain/problem languages

- AR: an action description language.
- SMV: a hardware description language.
- *NuPDDL*: extension of PDDL.

NuPDDL Plan language

- captures general framework. . .
- . . . expressive enough for every special cases ...
- ... in particular “classical” sequential plans.



# Standard PDDL features

- Designed for classical planning:
  - Deterministic
  - Full observability
  - Reachability goals
- Based on closed world assumption
- Implicit inertiality
- Parametrized actions
- Typing (enums) allowed
- Layered structure (STRIPS is layer 0)



# A simple PDDL example

```
(define (domain simple_robot)
  (:predicates
       (at_robot_sw) (at_robot_nw)
       (at_robot_se) (at_robot_ne))

  (:action move_robot_sw_se
   :parameters ()
   :precondition (at_robot_sw)
   :effect       (and (not (at_robot_sw)) (at_robot_se)))

   ;....
   ;....
)

(define (problem robot_pb)
        (:domain simple_robot)
        (:init  (at_robot_nw))
        (:goal  (at_robot_sw)))
```

# PDDL: types and parametricity

```
(define (domain simple_robot)
  (:types robot)
  (:predicates
       (at_robot_sw ?r - robot) (at_robot_nw ?r - robot)
       (at_robot_se ?r - robot) (at_robot_ne ?r - robot))

  (:action move_robot_sw_se
   :parameters (?r - robot)
   :precondition (at_robot_sw ?r)
   :effect       (and (not (at_robot_sw ?r)) (at_robot_se ?r)))

   ;....
   ;....
)

(define (problem robot_pb)
        (:domain simple_robot)
        (:objects robot_1 robot_2 - robot)
        (:init   (at_robot_nw robot_1) (at_robot_nw robot_2))
        (:goal   (and (at_robot_sw robot_1) (at_robot_sw robot_2))))
```

ITC irst



# PDDL: conditional effects

```
(define (domain simple_robot)
  (:predicates
       (at_robot_sw) (at_robot_nw)
       (at_robot_se) (at_robot_ne))
  (:action move_robot_east
   :parameters ()
   :precondition (or (at_robot_sw) (at_robot_nw))
   :effect       (and
                   (when (at_robot_sw)
                         (and (not (at_robot_sw)) (at_robot_se)))
                   (when (at_robot_nw)
                         (and (not (at_robot_nw)) (at_robot_ne)))))
   ;....
   ;....
)

(define (problem robot_pb)
        (:domain simple_robot)
        (:init   (at_robot_nw))
        (:goal   (at_robot_sw)))
```

ITC irst



# PDDL: quantifiers

```
(define (domain simple_robot)
  (:types package)
  (:predicates
       (at_robot_sw) (at_robot_nw)
       (at_robot_se) (at_robot_ne)
       (at_sw ?p - package) (at_nw ?p - package)
       (at_se ?p - package) (at_ne ?p - package))
  (:action move_robot_sw_se
   :parameters ()
   :precondition (at_robot_sw)
   :effect       (and
                    (not (at_robot_sw))
                    (forall (?p - package) (not (at_sw ?p)))
                    (at_robot_se)
                    (forall (?p - package) (at_se ?p))))
   ;....
   ;....
)
```

ITC irst

# PDDL2.1: functions

```
(define (domain simple_robot)
  (:types x_coord y_coord)
  (:constants east west   - x_coord
              north south - y_coord)
  (:functions
      (robot_x)
      (robot_y) )

  (:action move_robot_east
     :parameters ()
     :precondition (=  (robot_x) 0)
     :effect       (assign (robot_x) 1))
  ;....
  ;....
)

(define (problem robot_pb)
        (:domain simple_robot)
        (:init   (= (robot_y) 1) (= (robot_x) 0))
        (:goal   (and (= (robot_y) 0) (= (robot_x) 0))))
```



# NuPDDL

- Backward compatibility:
  - Retains closed world assumption, inertiality, parametricity.
  - Includes most of PDDL up ADL layer.
  - Includes PDDL2.1 “functions” extension.
- No layered structure.
- Typing enforced.
- Allows nested quantifications and conditionals.
- Extension: Nondeterminism (initial, action effects).
- Extension: Partial observability.
- Extension: Goal classes.



# Extension n. 1: uncertainty in initial

- “oneof” selects one of several *initial situations*.
- “oneof” is local to the cited elements.
- “unknown” shortcut: “oneof on every possible value”.

```
(define (domain d) ... (:predicates (P1) (P2) (P3) (P4) (P5) (P6)) ...)

(define (problem p)
   ....
   (:init (and (P1) (oneof (and (P2) (P3)) (P4)) (unknown (P5))))
   ....)
```

| P1 | P2 | P3 | P4 | P5 | P6 |
|---|---|---|---|---|---|
| T | T | T | F | T | F |
| T | F | F | T | T | F |
| T | T | T | F | F | F |
| T | F | F | T | F | F |

# Uncertainty in initial (II)

```
(define (problem robot_pb)
        (:domain simple_robot)
        (:init
           (oneof
             (and (= (robot_x) west) (= (robot_y) north))
             (and (= (robot_x) east) (= (robot_y) south))))
        (:goal
                (and (= (robot_y) south)
                     (= (robot_x) west))))


(define (problem robot_pb)
        (:domain simple_robot)
        (:init  (and
                     (= (robot_x) west)
                     (unknown (robot_y))))
        (:goal
                (and (= (robot_y) south)
                     (= (robot_x) west))))
```



# Extension n. 2: nondeterministic effects

- “oneof” selects one of several *transitions*.
- “oneof” is local to the cited elements.
- “unknown” is shortcut as usual.

```
(define (domain d) ... (:predicates (P1) (P2) (P3) (P4) (P5) (P6)) ...)

(define (problem p)
   ....
   (:effect (and (P1) (oneof (and (P2) (P3)) (P4)) (unknown (P5))))
   ....)
```

| P1’ | P2’ | P3’ | P4’ | P5’ | P6’ |
|---|---|---|---|---|---|
| T | T | T | P4 | T | P6 |
| T | P2 | P3 | T | T | P6 |
| T | T | T | P4 | F | P6 |
| T | P2 | P3 | T | F | P6 |



# Nondeterministic effects (II)

```
(define (domain simple_robot)
  (:types x_coord y_coord)
  (:constants east west - x_coord
              north south - y_coord)
  (:functions
      (robot_x) - x_coord
      (robot_y) - y_coord)
  (:action move_robot_east
     :parameters ()
     :precondition (=  (robot_x) west)
     :effect       (and
                      (assign (robot_x) east)
                      (unknown (robot_y))))
     ;....
     ;....
)
```

# Nondeterministic effects (III)

```
(define (domain simple_robot)
  (:types x_coord y_coord)
  (:constants east west - x_coord
              north south - y_coord)
  (:functions
      (robot_x) - x_coord
      (robot_y) - y_coord)
  (:action move_robot_east
     :parameters ()
     :precondition (=  (robot_x) west)
     :effect       (oneof
                     (and
                      (assign (robot_x) east)
                      (assign (robot_y) south))
                     (and
                      (assign (robot_x) west)
                      (assign (robot_y) north))))

     ;....
     ;....
)
```



# Nondeterministic effects (IV)

```
(define (domain simple_robot)
  (:types x_coord y_coord)
  (:constants east west - x_coord
              north south - y_coord)
  (:functions
      (robot_x) - x_coord
      (robot_y) - y_coord)
  (:action move_robot_east
     :parameters ()
     :precondition (=  (robot_x) west)
     :effect       (and
                      (assign (robot_x) east)
                      (when (= (robot_y) north)
                          (unknown (robot_y)))))
  ;....
  ;....
)
```



# Nondeterministic effects (V)

```
(define (domain simple_robot)
  (:types x_coord y_coord package)
  (:constants east west - x_coord
              north south - y_coord)
  (:predicates
    (broken ?p - package))
  (:functions
      (robot_x) - x_coord
      (robot_y) - y_coord)
  (:action move_robot_east
     :parameters ()
     :precondition (=  (robot_x) west)
     :effect       (and
                      (assign (robot_x) east)
                      (when (= (robot_y) north)
                          (and
                            (forall (?p - package)
                              (when (not (broken ?p))
                                (unknown (broken ?p))))
                            (unknown (robot_y))))))
     ;....
     ;....
)
```

# Extension n. 3: Partial Observability

- Introduce boolean observations
- Observations are parametric
- Noisy sensing allowed



# Partial observability (II)

```
(define (domain simple_robot)
  (:types x_coord y_coord)
  (:constants east west - x_coord
              north south - y_coord)
  (:functions
      (robot_x) - x_coord
      (robot_y) - y_coord)

  ;....
  ;....
  (:observation wall_north - boolean
     :parameters ()
     (imply (= wall_north 0)
            (not (or (= (robot_y) north) (= (robot_x) west))))
     (imply (= wall_north 1)
            (or (= (robot_y) north) (= (robot_x) west))))
  ;....
  ;....
)
```



# NuPDDL numbers

- NuPddl allows for finite numeric ranges.
- NuPddl does not allow for unranged `number`s.

```
(define (domain simple_robot)
  (:functions
      (robot_x) - (range 0 1)
      (robot_y) - (range 0 1))

  (:action move_robot_east
     :parameters ()
     :precondition (<  (robot_x) 1)
     :effect       (assign (robot_x) (+ (robot_x) 1)))

  ;....
)

(define (problem robot_pb)
        (:domain simple_robot)
        (:init  (and (= (robot_y) 1) (= (robot_x) 0)))
        (:goal  (and (= (robot_y) 0) (= (robot_x) 0))))
```

# NuPDDL numbers (II)

- Ranges can be instantiated within problem
- Limits of ranges can be accessed via `inf`, `sup`

```
(define (domain simple_robot)
  (:types range_x range_y)
  (:functions
      (robot_x) - range_x
      (robot_y) - range_y)
  (:action move_robot_east
     :parameters ()
     :precondition (< (robot_x) (sup range_x))
     :effect      (assign (robot_x) (+ (robot_x) 1)))
)

(define (problem robot_pb)
        (:domain simple_robot)
        (:typedef
           range_x - (range 0 4)
           range_y - (range 0 5))
        (:init  (and (= (robot_y) 3) (= (robot_x) 2)))
        (:goal  (and (= (robot_y) 0) (= (robot_x) 0))))
```



# Extension n. 4: goal classes

- Weak reachability under full observability.
  `(:weakgoal (at_robot_sw))`
- Strong reachability under full observability.
  `(:stronggoal (at_robot_sw))`
- Strong cyclic reachability under full observability.
  `(:strongcyclicgoal (at_robot_sw))`
- Strong reachability under partial observability.
  `(:postronggoal (at_robot_sw))`
- Strong reachability under null observability.
  `(:conformantgoal (at_robot_sw))`
- Extended goals under full observability.
  `(:ctlgoal (au (not (at_robot_sw)) (at_robot_ne)))`



# NuPDDL: CTL goals

| | |
|---|---|
| Do Reach p ("strong goal"): | `(af p)` |
| Try Reach p ("weak goal"): | `(ef p)` |
| Keep Trying Reach p ("strong cyclic goal"): | `(aw (ef p) p)` |
| Continuously Try Reach p: | `(ag (ef p))` |
| Do Maintain p: | `(ag p)` |
| Try Maintain p: | `(eg p)` |
| Do Maintain p Until q: | `(au p q)` |
| In All Next States p: | `(ax p)` |
| In Some Next States p: | `(ex p)` |
| And: | (and $g_1$ $g_2$ $g_3$...) |
| Or: | (or $g_1$ $g_2$ $g_3$...) |
| Implies: | `(imply p g)` |

# NuPDDL: plan language

Overview:

- Consistently with theory, allows defining an automata.
- Simple plan structures easily captured.
- Syntax style taken from domain definition part of NuPDDL.
- User-friendly imperative-style constructs supported.

ITC irst



# NuPDDL: plan language (II)

- A plan *may* feature a set of typed `:planvars`.
- Plan vars are `:init`ialized (otherwise they assume a default).
- Basic plan steps:
  - `(done)` signals ending of plan.
  - `(fail)` signals plan failure.
  - `(evolve (assign (next(v1) val1)) ... (action (act)))`
  - ..or simply: `(action (act))`
- Steps can be `sequence`d.
  - Branch constructs:
  - `(if (cond) plan1 plan2)`
  - `(switch (case (cond1) plan1) ... (else plan_else))`
- Imperative-style constructs: `label` and `goto`
- Iterations: `while` and `repeat`

ITC irst



# NuPDDL: plan language (III)

```
(define (plan silly_plan)
        (:domain robot_navigation)
        (:problem navigation_problem)
        (:planvars visited_lab - boolean
                   visited_SW_room_no - (range 0 10))
        (:init
            (= (visited_SW_room_no) 0)
            (= (visited_lab) 0)
        )
        (:body
            (sequence
                (while (< (visited_lab) 10)
                    (sequence
                       (evolve (assign
                                  (next (visited_SW_room_no))
                                  (+ (visited_SW_room_no) 1))
                               (action (move_robot_down)))
                       (action (move_robot_up))))
                (action (move_robot_right))
                (label i_am_in_lab (if (= (robot_position) lab)
                    (sequence
                      (evolve
                              (assign (next(visited_lab)) 1)
                              (action (move_robot_down)))
                      (action (move_robot_down)))
                    (action (move_robot_down))))
                (switch
                   (case (= (robot_position) dep) (done))
                   (case (= (robot_position) lab) (goto i_am_in_lab))
                   (case (= (robot_position) store) (fail))
                   (else (fail))))))
```

ITC irst

# NuPDDL: further work...

- NuPDDL designed for backward compatibility
- shall take into account axioms
- shall take into account explicit time
- future work on standardizing:
  - Allow for nesting functions?
  - Allow for probabilities?
  - ....
- **Feedback welcome**.
- NuPDDL: a candidate for possible ND planning competition?

ITC irst



# Hands On!

ITC irst



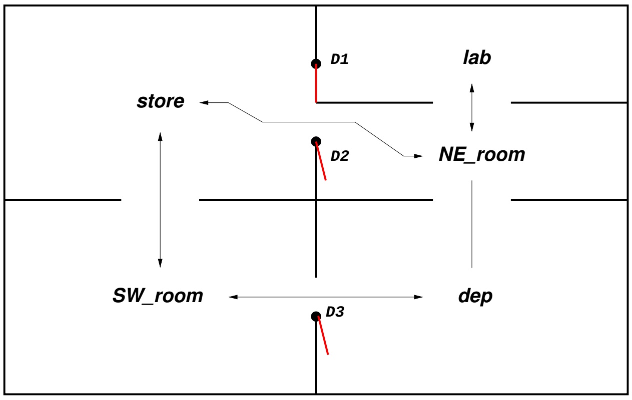
# Practical session on computer

The participants will:

- confront with a set of assignments...
  - design of a planning domain.
  - definition of goals.
  - plan synthesis.
  - plan validation and simulation.
- ...using MBP.

ITC irst

# Starting example: robot navigation






# A nuPDDL domain model: `robot.npddl`

```
(define (domain robot_navigation)
  (:types room)
  (:constants store lab NE_room SW_room dep - room)
  (:functions  (robot_position) - room)

  (:action move_robot_up
     :precondition (or (= (robot_position) SW_room )
                       (= (robot_position) dep     )
                       (= (robot_position) NE_room ))
     :effect (and
                (when (= (robot_position) SW_room ) (assign (robot_position) store   ))
                (when (= (robot_position) dep     ) (assign (robot_position) NE_room ))
                (when (= (robot_position) NE_room ) (assign (robot_position) lab     ))))

  (:action move_robot_down
     :precondition (or (= (robot_position) store   )
                       (= (robot_position) lab     )
                       (= (robot_position) NE_room ))
     :effect (and
                (when (= (robot_position) store   ) (assign (robot_position) SW_room ))
                (when (= (robot_position) lab     ) (assign (robot_position) NE_room ))
                (when (= (robot_position) NE_room ) (assign (robot_position) dep     ))))

  (:action move_robot_right
     :precondition (or (= (robot_position) SW_room )
                       (= (robot_position) store   ))
     :effect (and
                (when (= (robot_position) SW_room ) (assign (robot_position) dep))
                (when (= (robot_position) store   ) (assign (robot_position) NE_room))))

  (:action move_robot_left
     :precondition (or (= (robot_position) dep     )
                       (= (robot_position) NE_room ))
     :effect (and
                (when (= (robot_position) dep     ) (assign (robot_position) SW_room ))
                (when (= (robot_position) NE_room ) (assign (robot_position) store   )))))
```



# Questions

1. Synthesize (and save) a strong plan for reaching *dep* from *store*.
2. Synthesize (and save) a conformant plan for the same problem.

Now suppose D3 is closed (initially and forever):

- Update the domain.
- Check whether the plans generated before are still valid.
- Synthesize:
  - A strong plan for reaching *dep* from *store*.
  - A conformant plan for the same problem.

# A nondeterministic domain

Now suppose that:

1. (initially and forever) every door is open.
2. going east from *store* may lead to either *lab* or *NE_room*.

Then:

- Update the domain. *(Tip: modeling doors is not necessary...)*
- Synthesize a strong plan to reach *lab* from *store*.
- Synthesize a conformant plan for the same problem.
- Is the strong plan valid assuming no observability?



# A nondeterministic domain (II)

Now:

- D3 is uncontrollable, D1 and D2 are open.
- The robot “bounces” on D3 if closed.

Then:

- Update the domain. *(Tip: one can model D3 through the way it affects movements...)*
- Synthesize a strong plan to go from *store* to *dep*.
- Synthesize a conformant plan for the same problem.



# Extended goals

With D3 uncontrollable, D1 and D2 open, suppose *lab* is a dangerous room.

1. Is there a strong plan from *store* to *dep*, admitting passage into *lab*?
2. Is there a strong plan that leads from *store* to *dep* avoiding *lab*?
3. Is there a weak plan for the same problem?
4. Is there a strong cyclic plan for the same problem?

# Problems with Partial Observability

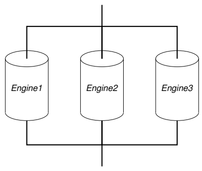
Now suppose that:

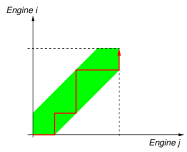
- Exactly one of the doors is open.
- The robot cannot try traversing a locked door.
- The robot can sense whether it can move in one direction or not.

- Is there a strong plan from store to dep? A weak plan?
- Is there a conformant plan? A strong plan using observations?
- Suppose the robot can smell whether it's in the *lab*. Is there a strong plan using observations?



# Advanced assignments: extended goals (I)





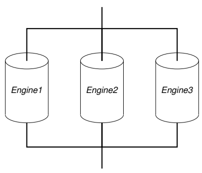

- Three engines, each providing from 0 to 4 levels of power.
- Engines start from being off.
- Problem: reach maximum power while keeping balancing (see figure).

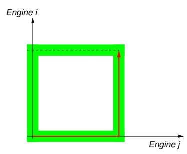
Tasks:

1. Model the domain.
2. Synthesize a strong plan for the problem (if there is one).
3. Simulate the plan.
4. Write a smarter plan, validate and simulate it.



# Advanced assignments: extended goals (II)






- Now problem is: reach maximum power following saturation policy.
- Saturation: at most one engine is "half way through" (see figure).

Tasks:

1. Model the domain.
2. Synthesize a strong plan for the problem.
3. Write a smarter plan and validate it.

# Advanced assignments: extended goals (III)

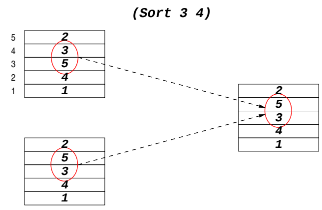
Possible advanced goals:

- Changing request level:
  - the sum of the power provided by the engines should reach a given *request level*;
  - the request level may increase or decrease.
- Alarm:
  - whenever an alarm is raised, all the engines should be turned off;
  - the alarm ends once all the engines are off.
- Switching policy:
  - a request of switch from saturation to balancing or vice-versa can be raised at run-time.

ITC irst



# Advanced assignments: conformant (I)



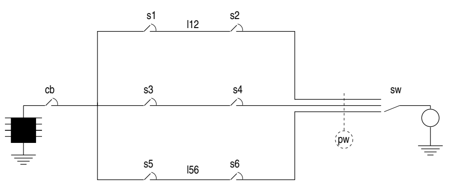

- A stack of 5 numbers, each ranging from 1 to 5.
- An atomic pairwise *(sort x y)* operation.
- Any configuration possible at start.
- The stack must be sorted at the end.

Tasks: Model the domain and find a conformant plan for the problem.

ITC irst



# Advanced assignments: PO (I)




- An electric circuit. Possible actions: open/close *cb*,*s1*,...,*s6*.
- One of *l12* or *l56* has a shortcircuit.
- When *cb* feeds a shortcircuit it automatically reopens.
- Switch *s3* is unreliable: it may not obey.
- Sensor *pw* tells whether power gets to 3-position switch *sw*.
- Goal: turn on light. Initially *cb*,*s1*,...,*s6* are open, *sw* is at position 1.
- Task: model the domain, solve the problem,.
- Task: design a smarter plan, validate and simulate it.

ITC irst

# Conclusions



# Concluding remarks

- Planning: a difficult, real problem.
  - Realistic instances are huge.
  - Realistic solutions must deal with a number of issues (nondeterminism, partial observability, complex goals,...).
- Symbolic Model Checkhing: an advanced validation technique...
  - Clear theoretical framework.
  - A huge amount of existing work on systems and techniques.
- Planning via Symbolic Model Checking:
  - Smart reuse of Model Checking results...
  - Inherits clear theoretical framework...
  - Nice practical results as well!
- MBP: a complex planner based on Symbolic Model Checking.



# Related work: Reachability

- Several approaches deal with "classical" reachability:
  - GraphPlan-based: (GRAPHPLAN,IPP,STAN,...)
    Build a "planning graph" representing effects and mutual dependencies of actions.
  - Enumerative heuristic-based (FF,GRT,HSP[2],...)
  - Symbolic approaches (BDDPLAN,MIPS,PROPPLAN,SATPLAN,...)
    Exploit symbolic representation of problem state to avoid enumeration.
  - Causal planners (UCPOP,...).
  - Hybrid (ALTALT,BLACKBOX,STAN,...).
- Some works deal with nondeterminism and cyclic plans:
  - Logical-based approach (C-PLAN,QBFPLAN,UMOP,...)
    Logical encoding of non-determinism.
  - Stochastic (BURIDAN,GPT,SPUDD,...)
    Exploit Markov Decision Processes tecniques.

# Related works: Extended goals

- Temporal logics have been used:
  - to express extended goals (SIMPLAN,...).
  - to express search control strategies (TLPLAN, TALPLAN,...).
  - to express pre-conditions of actions (in action description languages).

  Most of these approaches are based on LTL.
- Planning with CTL goals is related to CTL synthesis [De Giacomo, Kupferman, Vardi]:
  - synthesis of a control automaton,
  - combination with the domain model.
- Strong relations also with “synthesis of controllers”...

ITC irst



# Related work: Partial Observability

- Other approaches to same problem:
  - Extensions to GRAPHPLAN (CGP,SGP,...)
  - Stochastic (C-BURIDAN,GPT, ...)
  - Symbolic approaches (QBFPLAN, C-PLAN, ...)
- Related problem: reactive planning.
  - Offline planning may be overkill.
  - Replanning necessary in several real-world applications (e.g. robotics).
  - Several systems available (e.g. work on Deep Space missions).
- Related problem: diagnosis.
  - Diagnosis as “planning to remove executor’s uncertainty”.
  - Model-based diagnosis an established field.
  - Several model-based diagnosers available.

ITC irst