Modeling Planning Domains

Roman Barták, Lukáš Chrpa

Introduction

Action planning deals with the problem of finding a sequence of actions (a plan) to transfer the world from the current state to a desired state.

- There are **causal relations** between actions (pick-up is done before put-down).
- A formal model of actions is required so planning is a **model-based approach**.

This tutorial is about how to model planning problems.

Part I: Introduction and Background
- *AI Planning*
- *Formal models (STRIPS, control rules, HTNs)*

Part II. Planning Domain Modelling Languages and Tools
- *Modelling languages*
- *Modelling tools*
- *Lessons from ICKEPS*

Part III. Designing and Developing a Domain Model
- *15-puzzle, Nomystery problem*
- *Road Traffic Accident Management*

Part IV. Development of Real-World Planning Application
- *Petrobras*
- *Task Planning for Autonomous Underwater Vehicles*

Part V. Closing Remarks and Open Problems

Part I:

INTRODUCTION AND BACKGROUND

Planning deals with **selection and organization of actions** that are changing world states.

System Σ modelling states and transitions:

- **set of states S** (recursively enumerable)
- **set of actions A** (recursively enumerable)
  - actions are controlled by the planner!
  - no-op
- **set of events E** (recursively enumerable)
  - events are out of control of the planner!
  - neutral event ε
- **transition function γ**: $S \times A \times E \rightarrow 2^S$
  - actions and events are sometimes applied separately
    $\gamma: S \times (A \cup E) \rightarrow 2^S$

A **planning task** is to find which actions are applied to world states to reach some goal from a given initial state.

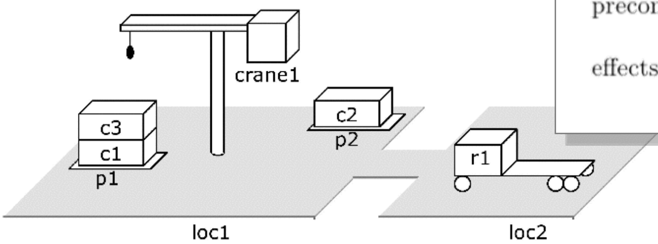
What is a goal?

- **goal state** or a set of of goal states
- **satisfaction of some constraint** over a sequence of visited states
  - for example, some states must be excluded or some states must be visited
- **optimisation of some objective function** over a sequence of visited states (actions)
  - for example, maximal cost or a sum of costs

Representing **world states** as sets of atoms (factored representation).

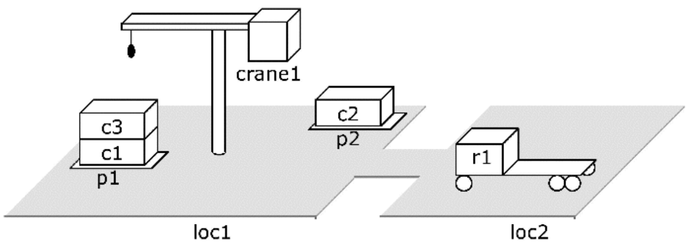
Representing **actions** as entities changing validity of certain atoms.

```
take(crane1,loc1,c3,c1,p1)
  ;; crane crane1 at location loc1 takes c3 off c1 in pile p1
  precond: belong(crane1,loc1), attached(p1,loc1),
           empty(crane1), top(c3,p1), on(c3,c1)
  effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1),
           ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)
```



{attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

State is a set of instantiated atoms (no variables). There is a finite number of states!



{attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

- The truth value of some atoms is changing in states:
  - **fluents**
  - *example: at(r1,loc2)*
- The truth value of some state is the same in all states
  - **rigid atoms**
  - *example: adjacent(loc1,loc2)*

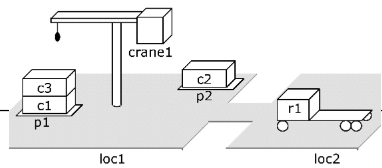
We will use a classical **closed world assumption**.
An atom that is not included in the state does not hold at that state!

operator o is a triple (name(o), precond(o), effects(o))

- **name(o): name of the operator** in the form $n(x_1,\dots,x_k)$
  - n: a symbol of the operator (a unique name for each operator)
  - $x_1,\dots,x_k$: symbols for variables (operator parameters)
    - Must contain all variables appearing in the operator definition!
- **precond(o):**
  - literals that must hold in the state so the operator is applicable on it
- **effects(o):**
  - literals that will become true after operator application (only fluents can be there!)

take(k, l, c, d, p)
;; crane k at location l takes c off of d in pile p
precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)
effects: holding(k, c), $\neg$ empty(k), $\neg$ in(c, p), $\neg$ top(c, p), $\neg$ on(c, d), top(d, p)

An action is a fully instantiated operator
– substitute constants to variables



operator

take(k, l, c, d, p)
;; crane k at location l takes c off of d in pile p
precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)
effects: holding(k, c), $\neg$ empty(k), $\neg$ in(c, p), $\neg$ top(c, p), $\neg$ on(c, d), top(d, p)

action

take(crane1,loc1,c3,c1,p1)
;; crane crane1 at location loc1 takes c3 off c1 in pile p1
precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)
effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

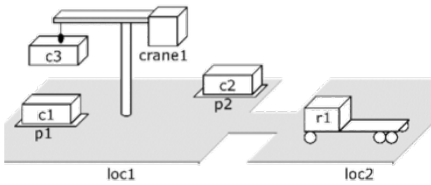
Notation:

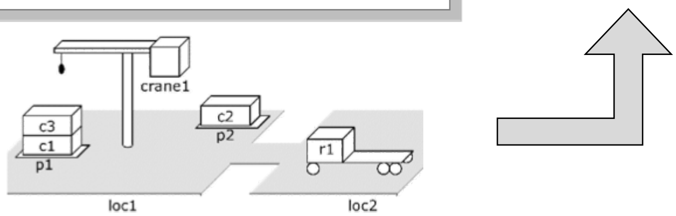
– S^+ = {positive atoms in S}
– S^- = {atoms, whose negation is in S}

Action **a** is **applicable** to state **s** if any only
$\text{precond}^+(\mathbf{a}) \subseteq \mathbf{s} \;\wedge\; \text{precond}^-(\mathbf{a}) \cap \mathbf{s} = \emptyset$

The result of application of action a to **s** is
$\gamma(\mathbf{s},\mathbf{a}) = (\mathbf{s} - \text{effects}^-(\mathbf{a})) \cup \text{effects}^+(\mathbf{a})$

```
take(crane1,loc1,c3,c1,p1)
  ;; crane crane1 at location loc1 takes c3 off c1 in pile p1
  precond: belong(crane1,loc1), attached(p1,loc1),
           empty(crane1), top(c3,p1), on(c3,c1)
  effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1),
           ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)
```





The **planning problem** is given by a triple (O,s_0,g).

– O defines the the operators and predicates used (this is also called a **domain model**)

– s_0 is an initial state, it provides the particular constants (objects)

– g is a set of instantiated literals

- state **s** satisfies the goal condition **g** if and only if $\mathbf{g}^+ \subseteq \mathbf{s} \wedge \mathbf{g}^- \cap \mathbf{s} = \emptyset$
- $S_\mathbf{g} = \{\mathbf{s} \in S \mid \mathbf{s} \text{ satisfies } \mathbf{g}\}$ – a set of goal states

Blockworld: classical representation

Constants

– blocks: a,b,c,d,e

Predicates:

– **ontable(*x*)**
  block *x* is on a table
– **on(*x*,*y*)**
  block *x* is on *y*
– **clear(*x*)**
  block *x* is free to move
– **holding(*x*)**
  the hand holds block *x*
– **handempty**
  the hand is empty

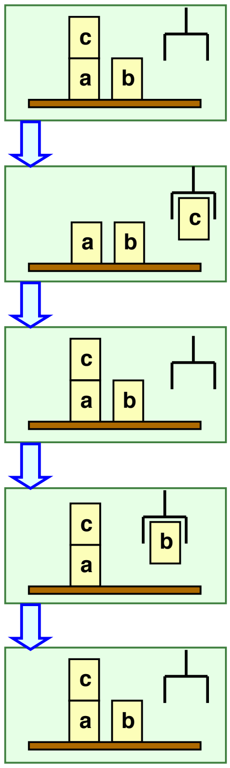
Operators

unstack(*x*,*y*)
Precond: on(*x*,*y*), clear(*x*), handempty
Effects: ¬on(*x*,*y*), ¬clear(*x*), clear(*y*), ¬handempty, holding(*x*),

stack(*x*,*y*)
Precond: holding(*x*), clear(*y*)
Effects: ¬holding(*x*), ¬clear(*y*), on(*x*,*y*), clear(*x*), handempty

pickup(*x*)
Precond: ontable(*x*), clear(*x*), handempty
Effects: ¬ontable(*x*), ¬clear(*x*), ¬handempty, holding(*x*)

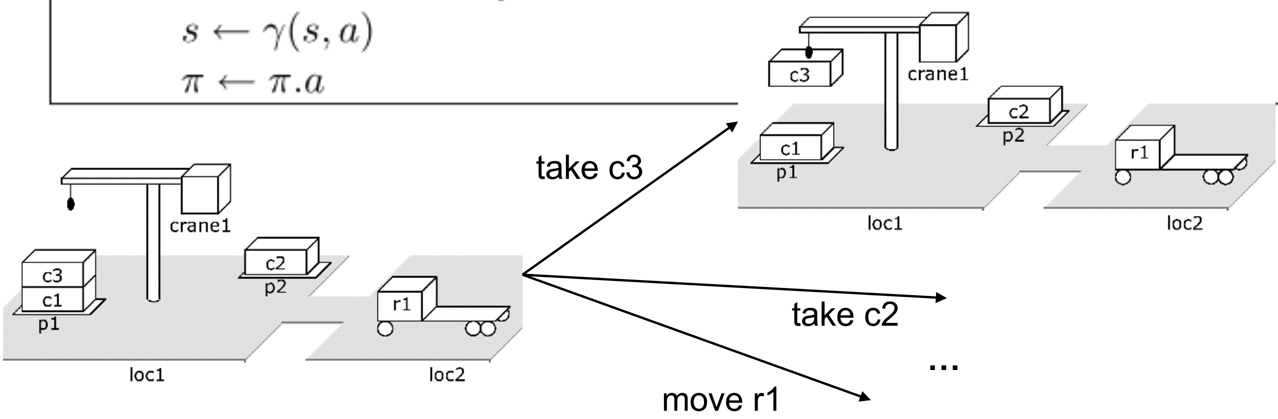
putdown(*x*)
Precond: holding(*x*)
Effects: ¬holding(*x*), ontable(*x*), clear(*x*), handempty



Forward planning

Forward-search(O, s_0, g)
  $s \leftarrow s_0$
  $\pi \leftarrow$ the empty plan
  loop
    if s satisfies g then return π
    $E \leftarrow \{a | a$ is a ground instance of an operator in O,
        and $\text{precond}(a)$ is true in $s\}$
    if $E = \emptyset$ then return failure
    nondeterministically choose an action $a \in E$
    $s \leftarrow \gamma(s, a)$
    $\pi \leftarrow \pi.a$

Heuristics suggest which action to select



Heuristics guide the planner towards a goal state by ordering alternative plans. They do not solve the problem with the **large number of alternatives**.

Example (blockworld)

- If a block is placed correctly (consistent with the goal) then any action that moves that block just enlarges the plan.
- If a block is on a wrong place and there is an action that moves it to the correct place then any action that moves the block elsewhere just enlarges the plan.

It is possible to describe desirable/forbidden sequences of states using linear temporal logic.

- **control rules**

It is possible to describe expected plans via task decompositions.

- **hierarchical task networks**

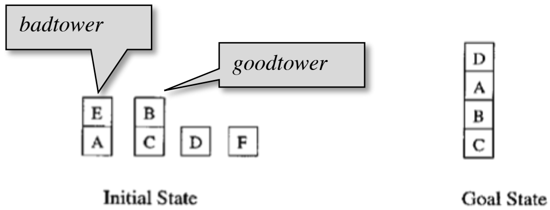
We need a formalism to express relations between the current world state and future states.

Simple temporal logic

- extension of first-order logic by **modal operators**
  - $\phi_1 \cup \phi_2$ (until) ϕ_1 is true in all states until the first state (if any) in which ϕ_2 is true
  - $\Box\, \phi$ (always) ϕ is true now and in all future states
  - $\Diamond\, \phi$ (eventually) ϕ is true now or in any future state
  - $\bigcirc\, \phi$ (next) ϕ is true in the next state
  - GOAL(ϕ) ϕ (no modal operators) is true in the goal state
- ϕ is a logical formula expressing relations between the objects of the world (it can include modal operators)

Control rules: an example

Goodtower is a tower such that no block needs to be moved.
Badtower is a tower that is not good.



$$goodtower(x) \triangleq clear(x) \land \neg \mathrm{GOAL}(holding(x)) \land goodtowerbelow(x)$$

$$\begin{aligned} goodtowerbelow(x) \triangleq\ & (ontable(x) \land \neg\exists[y{:}\mathrm{GOAL}(on(x,y))]\,)) \\ & \lor \exists[y{:}on(x,y)]\ \neg\mathrm{GOAL}(ontable(x)) \land \neg\mathrm{GOAL}(holding(y)) \land \neg\mathrm{GOAL}(clear(y)) \\ & \land \forall[z{:}\mathrm{GOAL}(on(x,z))]\ z = y \land \forall[z{:}\mathrm{GOAL}(on(z,y))]\ z = x \\ & \land goodtowerbelow(y) \end{aligned}$$

$$badtower(x) \triangleq clear(x) \land \neg goodtower(x)$$

Control rule:

$$\begin{aligned} \Box\big(\forall[x{:}clear(x)]\ & goodtower(x) \Rightarrow \bigcirc(clear(x) \lor \exists[y{:}on(y,x)]\ goodtower(y)) \\ & \land badtower(x) \Rightarrow \bigcirc(\neg\exists[y{:}on(y,x)]\,) \\ & \land (ontable(x) \land \exists[y{:}\mathrm{GOAL}(on(x,y))]\ \neg goodtower(y)) \\ & \quad \Rightarrow \bigcirc(\neg holding(x))\big) \end{aligned}$$

goodtower remains *goodtower*

do not put anything on *badtower*

do not take a block from a table until you can put it on a *goodtower*

Planning with control rules

Forward state-space planning guided by control rules.

– If a partial plan S_π violates the control rule progress(ϕ, S_π), then the plan is not expanded.

```
STL-plan(O, s0, g, φ)
    s ← s0
    π ← the empty plan
    loop
        if φ = false then return failure
        if s satisfies g then return π
        A ← {a | a is a ground instance of an operator in O
                  and precond(a) is true in s}
        if A = Ø then return failure
        nondeterministically choose an action a ∈ A
        s ← γ(s, a)
        π ← π.a
        φ ← progress(φ, s)
```

a partial plan violates the control rule ϕ

a complete plan found

actions applicable to state s

a new state after the action

control rule progression ϕ

Classical planning assumes primitive actions connected via causal relations.

In real-life we can frequently use "**recipes**" to solve a particular task.

– recipe is a set of operations to achieve a sub-goal

HTN planning is based on performing a set of tasks (instead of achieving goals).

– **primitive task**: performed by a classical planning operator
– **non-primitive task**: decomposed by a **method** to other tasks (can use recursion)

How to describe a recipe to perform a given task?

– specify sub-tasks and their relations

A **task network** is a pair (U,C), where U is a set of tasks and C is a set of constraints.

– **tasks** are named similarly to operators: $t(r_1,\ldots,r_n)$
– constraints are in the form:
  - **precedence constraint**: u < v (task u is performed before task v)
  - **before-constraint**: before(U',l) (literal l is true right before the set of tasks U')
  - **after-constraint**: after(U',l) (literal l is true right after the set of tasks U')
  - **between-constraint**: between(U',U'',l) (literal l must be true right after U', right before U'' and in all states in between)

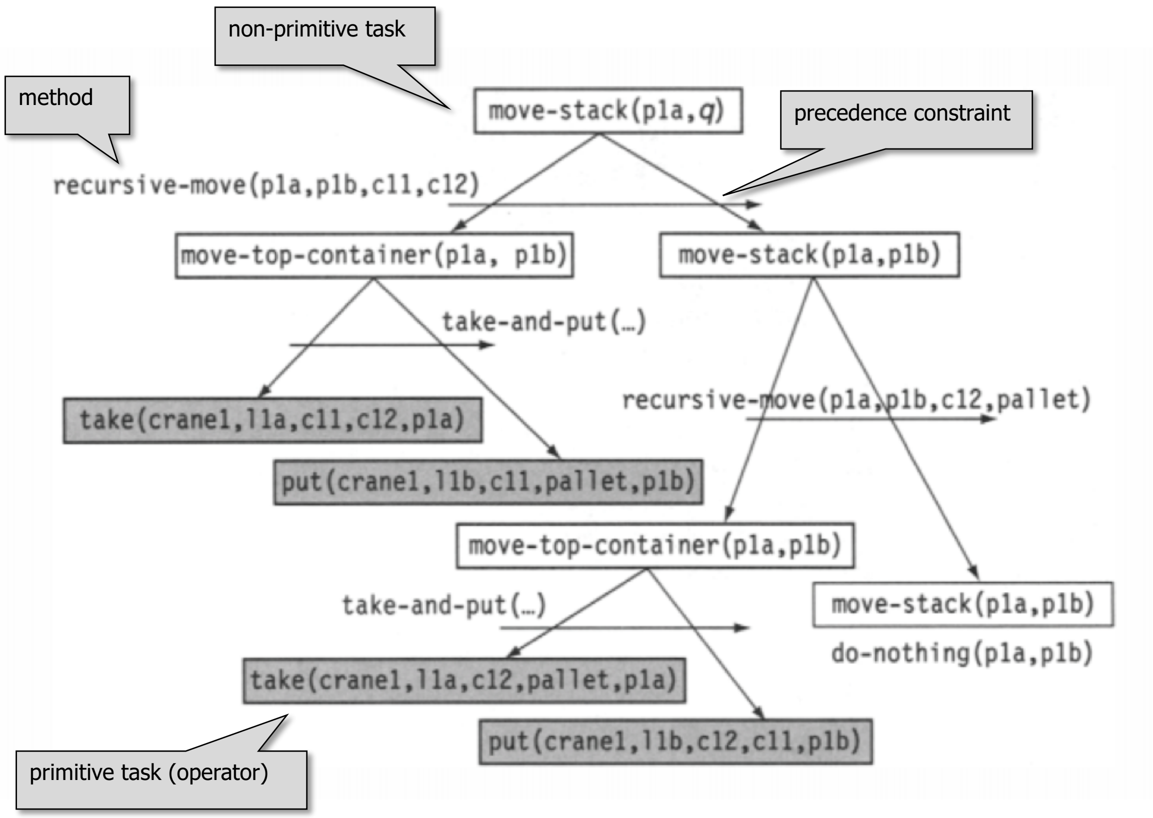
To perform non-primitive tasks, we need to decompose them to other tasks using a method.

An **HTN method** is a tuple
m = (*name, task, subtasks, constr*)

- *name* is $n(x_1,\ldots,x_n)$, where $\{x_1,\ldots,x_n\}$ are all variables in *m* and n is a unique name of the method,
- *task* is a non-primitive task,
- (*subtasks*, *constr*) is a task network.

There may be more methods for a single non-primitive task.

Task decomposition



HTN Problem

Now, the planning problem is specified somehow differently from classical planning as a process to obtain a plan from decomposition of tasks in a given task network.

An **HTN planning domain** is a pair (O,M)

- O is a set of operators
- M is a set of HTN methods

An **HTN planning problem** is a 4-tuple (s_0,w,O,M)

- s_0 is the initial state
- w is the initial task network
- (O,M) is the HTN planning domain

HTN Planning

```
Abstract-HTN(s, U, C, O, M)
    if (U, C) can be shown to have no solution
        then return failure
    else if U is primitive then
        if (U, C) has a solution then
            nondeterministically let π be any such solution
            return π
        else return failure
    else
        choose a nonprimitive task node u ∈ U
        active ← {m ∈ M | task(m) is unifiable with t_u}
        if active ≠ ∅ then
            nondeterministically choose any m ∈ active
            σ ← an mgu for m and t_u that renames all variables of m
            (U', C') ← δ(σ(U, C), σ(u), σ(m))
            (U', C') ← apply-critic(U', C') ;; this line is optional
            return Abstract-HTN(s, U', C', O, M)
        else return failure
```

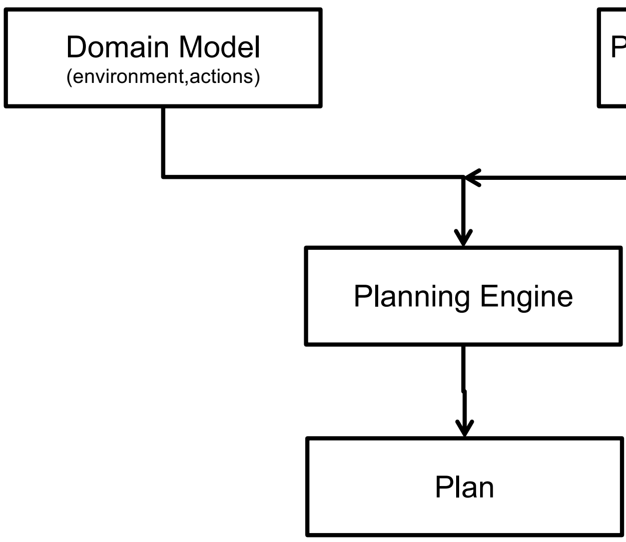
decomposition of a task

performing application-specific computations

Part II.

PLANNING DOMAIN MODELLING LANGUAGES AND TOOLS

Domain-independent planning concept



Domain-independent planning concept

- A (description) **language**
  - Describe domain model and problem specification (usually one domain model for a class of problems)
- A **planning engine**
  - must support the language
  - should be efficient for the given domain model
- **Plans** interpreting

PDDL [McDermott et al, 1998]

- Planning Domain Definition Language (PDDL)
- Inspired by the STRIPS and ADL languages
- Most widespread
- Official language of International Planning Competitions (IPCs)

```
(define (domain blocksworld)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
       (ontable ?x - block)
       (clear ?x - block)
       (handempty)
       (holding ?x - block)
       )
  (:action pick-up
     :parameters (?x - block)
     :precondition (and (clear ?x)
                        (ontable ?x)
(handempty))
     :effect (and (not (ontable ?x))
                        (not (clear ?x))
                        (not (handempty))
                        (holding ?x))
)
…
```

Versions of PDDL

- **PDDL 1.2**
  - Predicate centric (i.e., classical representation)
  - Object types
  - ADL features (e.g., conditional effects, equality)
- **PDDL 2.1**
  - Numeric Fluents
  - Durative Actions
- **PDDL 2.2**
  - Timed-initial literals
  - Derived Predicates
- **PDDL 3.0**
  - State-trajectory constraints (hard constraints for the planning process)
  - Preferences (soft constraints for the planning process)
- **PDDL 3.1**
  - Object Fluents

Extensions of PDDL

- **PDDL+**
  - Continuous processes
  - Exogenous events
- **PPDDL**
  - Probabilistic action effects
  - Reward fluents
- **MA-PDDL**
  - Multi-agent planning

NDDL [Frank & Jonsson, 2002]

- NASA's response to PDDL
- Variable representation
- Timelines/activities
- Constraints between activities

```
class Instrument
{
    Rover rover;
    InstrumentLocation location;
    InstrumentState state;

    Instrument(Rover r)
    {
             rover = r;
             location = new InstrumentLocation();
               state = new InstrumentState();
    }

     action TakeSample{
              Location rock;
             eq(10, duration);
     }
     …
}

Instrument::TakeSample
{
       met_by(condition object.state.Placed on);
       eq(on.rock, rock);

    contained_by(condition object.location.Unstowed);

       equals(effect object.state.Sampling sample);
       eq(sample.rock, rock);

       starts(effect object.rover.mainBattery.consume tx);
       eq(tx.quantity, 120); // consume battery power
}
```

https://github.com/nasa/europa/wiki/Example-Rover

ANML [Smith et al., 2008]

- Combines aspects from NDDL and PDDL
  - Actions and states (PDDL)
  - Variable representation (NDDL)
  - Temporal Constraints (NDDL)
- Hierarchical methods

```
action Pickup (crew ev, object item)
{
duration := 5 ;
[start] located(ev) == located(item);
[all] possesses(ev,item) ==
FALSE:->TRUE ;
[end] located(item) := POSSESSED ;
}

action Putaway (crew ev, object item,
location stowage)
{
Duration := 10 ;
[start] located(ev) == stowage ;
[all] possesses(ev, item) ==
TRUE:->FALSE ;
[end] located(item):= stowage ;
}
```

[Boddy & Bonasso, 2010]

RDDL [Sanner, 2011]

- became the official language of the probabilistic track of the IPC since 2011
- models partial observability
- efficient description of (PO)MDPs

```
domain wildfire_mdp {

types {
x_pos : object;
y_pos : object;
};

pvariables {

// Action costs and penalties
COST_CUTOUT        : {non-fluent, real, default =   -5 }; //
Cost to cut-out fuel from a cell
COST_PUTOUT        : {non-fluent, real, default =  -10 }; //
Cost to put-out a fire from a cell
PENALTY_TARGET_BURN : {non-fluent, real, default = -100 }; //
Penalty for each target cell that is burning
PENALTY_NONTARGET_BURN : {non-fluent, real, default =   -5 };
// Penalty for each non-target cell that is burning
…..
}

cpfs{
burning'(?x, ?y) =  if ( put-out(?x, ?y) ) // Intervention to
put out fire?
                     then false
         // Modification: targets can only start to burn if at
least one neighbor is on fire
            else if (~out-of-fuel(?x, ?y) ^ ~burning(?x, ?y))
// Ignition of a new fire? Depends on neighbors.
               then [if (TARGET(?x, ?y) ^ ~exists_{?x2: x_pos,
?y2: y_pos} (NEIGHBOR(?x, ?y, ?x2, ?y2) ^ burning(?x2, ?y2)))
                    then false
                    else Bernoulli( 1.0 / (1.0 + exp[4.5 -
(sum_{?x2: x_pos, ?y2: y_pos} (NEIGHBOR(?x, ?y, ?x2, ?y2) ^
burning(?x2, ?y2)))]) ) ]
               else
               burning(?x, ?y); // State persists
…
}
```

https://cs.uwaterloo.ca/~mgrzes/IPPC_2014/

Domain-independent planners

- Dozens of classical planners
  - support typed STRIPS
  - newer planners support action costs, and some ADL features
  - many of them are optimal
- Several temporal planners
  - support durative actions
  - few support numeric fluents or timed-initial literals
  - few fully support concurrency
  - very few are optimal
- Several probabilistic planners
  - (PO)MDP
  - FOND
- A few continuous planners
- ....

Language expressiveness vs. planning engines

"*It is almost a law in PDDL planning that for every language feature one adds to a domain definition, the number of planners that can solve (or even parse) it, and the efficiency of those planners, falls exponentially*" [anonymous reviewer]

Motivate **development of more expressive** planning engines

Reduce the number of **features** in models

Picat

Picat is a logic-based multi-paradigm language that integrates logic programming, functional programming, constraint programming, and scripting.

- logic variables, unification, backtracking, pattern-matching rules, functions, list/array comprehensions, loops, assignments
- tabling for dynamic programming and **planning**
- **constraint solving** with CP (constraint programming), SAT (satisfiability), and MIP (mixed integer programming).

Forward planning in Picat language (using tabling):

```
table (+,-,min)
plan(S,Plan,Cost),final(S) =>
    Plan=[],Cost=0.
plan(S,Plan,Cost) =>
    action(S,S1,Action,ActionCost),
    plan(S1,Plan1,Cost1),
    Plan = [Action|Plan1],
    Cost = Cost1+ActionCost.
```

Cost optimization done via:

- iterative deepening
- branch-and-bound

Goal condition

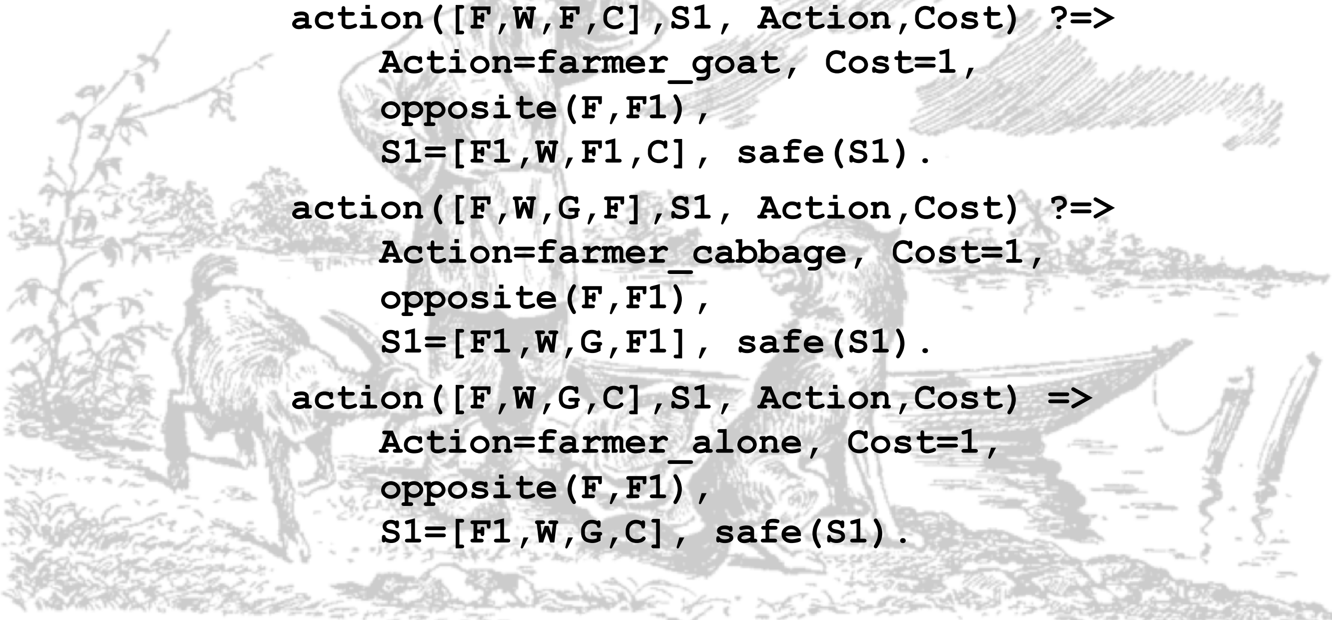
```
final(+State) => goal_condition.
```

Action description

```
action(+State,-NextState,-Action,-Cost),
   precondition,
   [control_knowledge]
?=>
   description_of_next_state,
   action_cost_calculation,
   [heuristic_and_deadend_verification].
```

Locations of
Farmer, Wolf, Goat, and Cabbage

```
action([F,F,G,C],S1, Action,Cost) ?=>
    Action=farmer_wolf, Cost=1,
    opposite(F,F1),
    S1=[F1,F1,G,C], safe(S1).
action([F,W,F,C],S1, Action,Cost) ?=>
    Action=farmer_goat, Cost=1,
    opposite(F,F1),
    S1=[F1,W,F1,C], safe(S1).
action([F,W,G,F],S1, Action,Cost) ?=>
    Action=farmer_cabbage, Cost=1,
    opposite(F,F1),
    S1=[F1,W,G,F1], safe(S1).
action([F,W,G,C],S1, Action,Cost) =>
    Action=farmer_alone, Cost=1,
    opposite(F,F1),
    S1=[F1,W,G,C], safe(S1).
```



KE Tools for Planning Domain Modelling

Purpose of KE tools

Assist in domain developing process

- Support development cycle (as in SW engineering)
- Visualize (parts of) the model
- Verification and Validation support (e.g. consistency check)
- …

Usable by non-experts (but with basic knowledge of planning)

GIPO [Simpson et al., 2007]

- GIPO (Graphical Interface for Planning with Objects) won the ICKEPS 2005 competition
- Based on the **OCL** (Object-Centred Language)
- Define **life histories** of **objects**
- Supports **"classical" PDDL** (limitedly also "durative" actions)
- Supports **HTN** (HyHTN planner is integrated) [McCluskey et al., 2003]

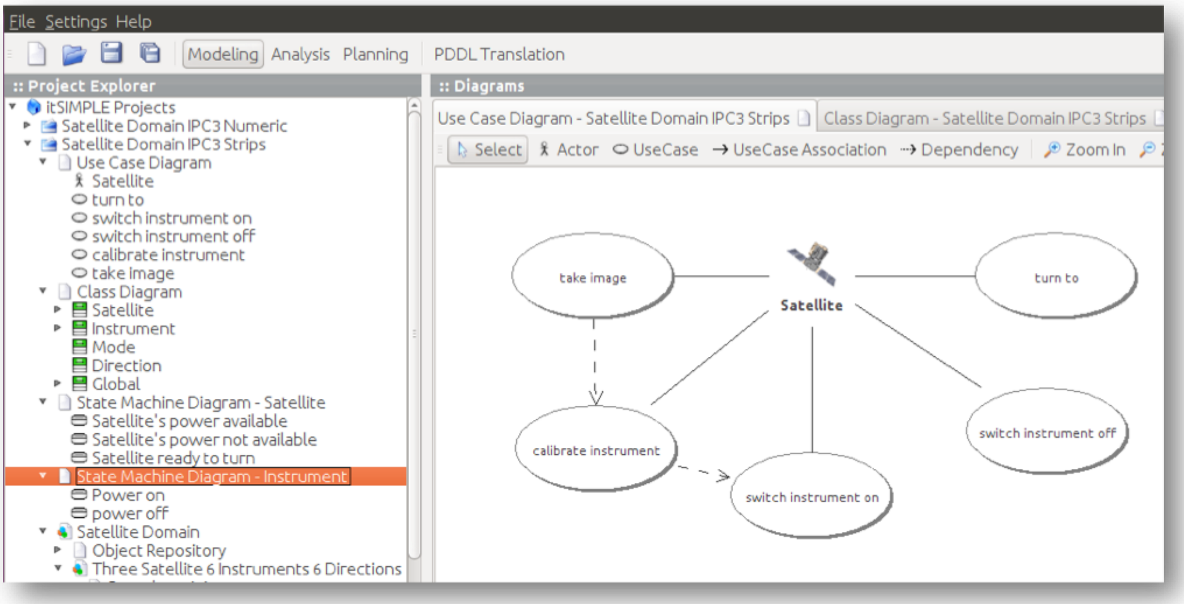
ItSimple [Vaquero et al., 2007;2012]

- Supports development cycle
- Exploits **UML** for domain modelling
- Exploits **Petri Nets** for dynamic analysis of **state machines** (e.g. reachability analysis)
- Supports **PDDL 3.1**
- Project webpage

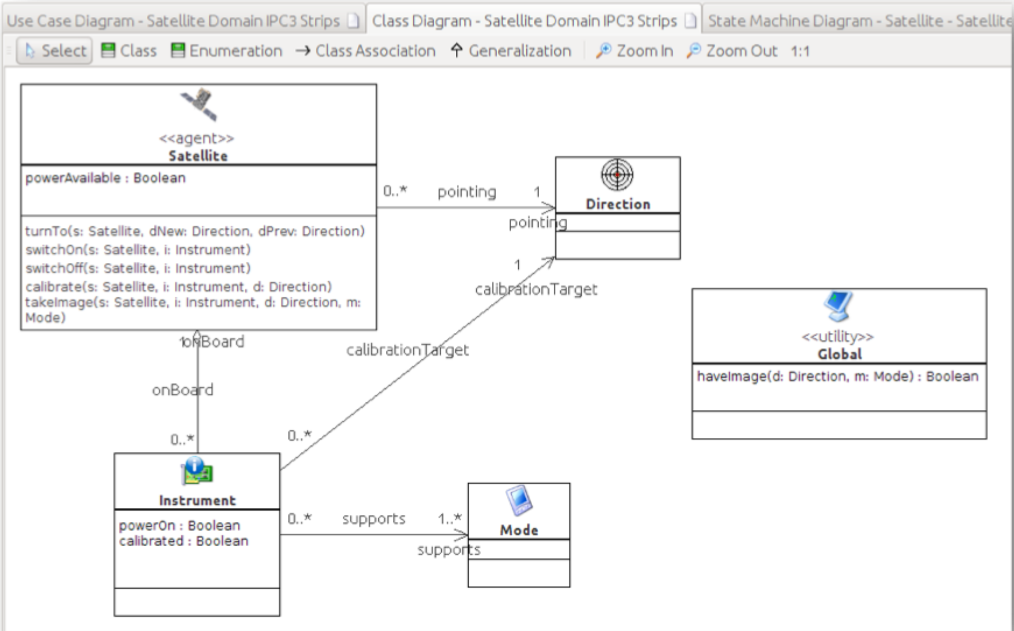
  https://code.google.com/archive/p/itsimple/
- Tutorial on domain modelling in ItSimple by Chris Muise

  http://www.youtube.com/watch?feature=player_embedded&v=FGBhvBnzyvo

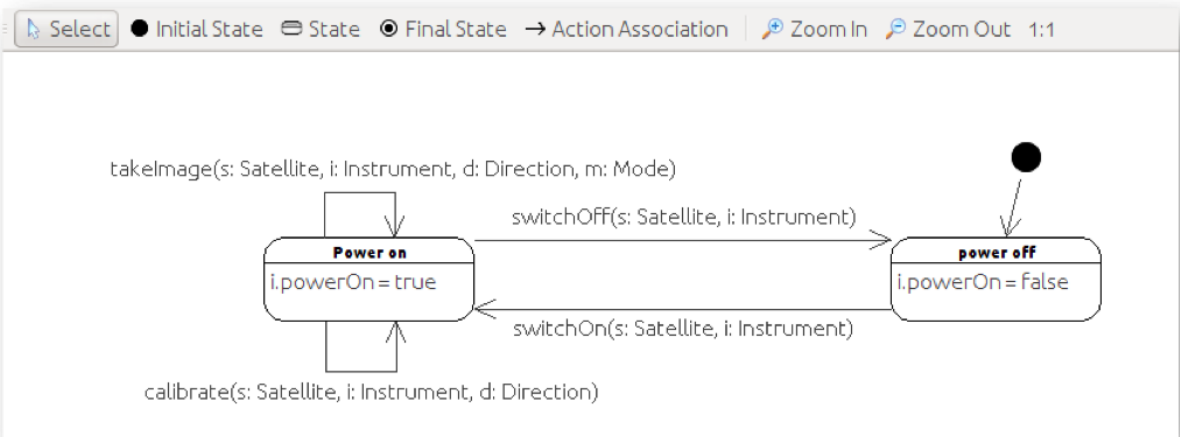
ItSimple – sample use case



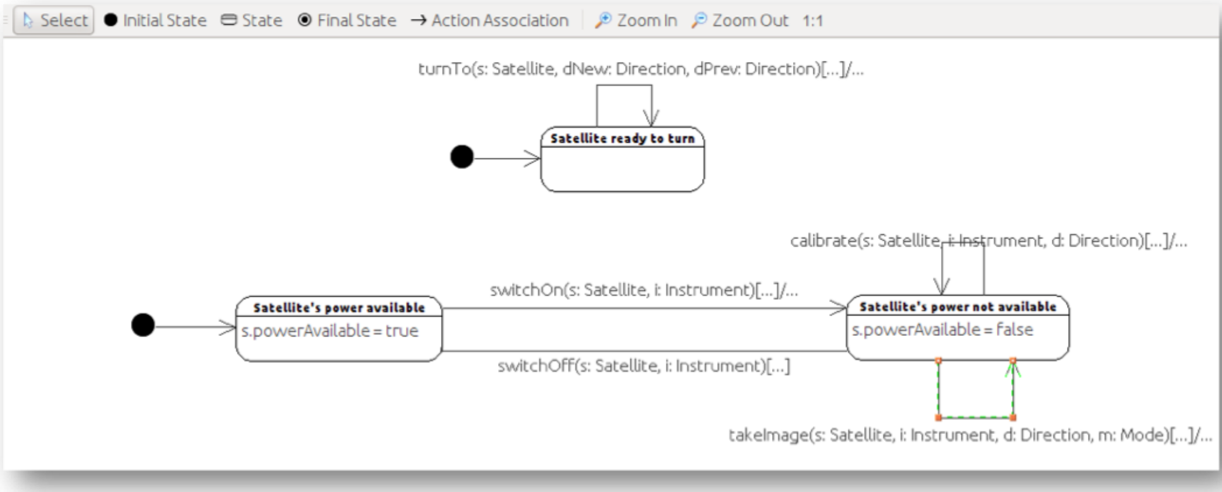
ItSimple – sample class diagram



ItSimple – sample state machine (Satellite)



ItSimple – sample state machine (Instrument)



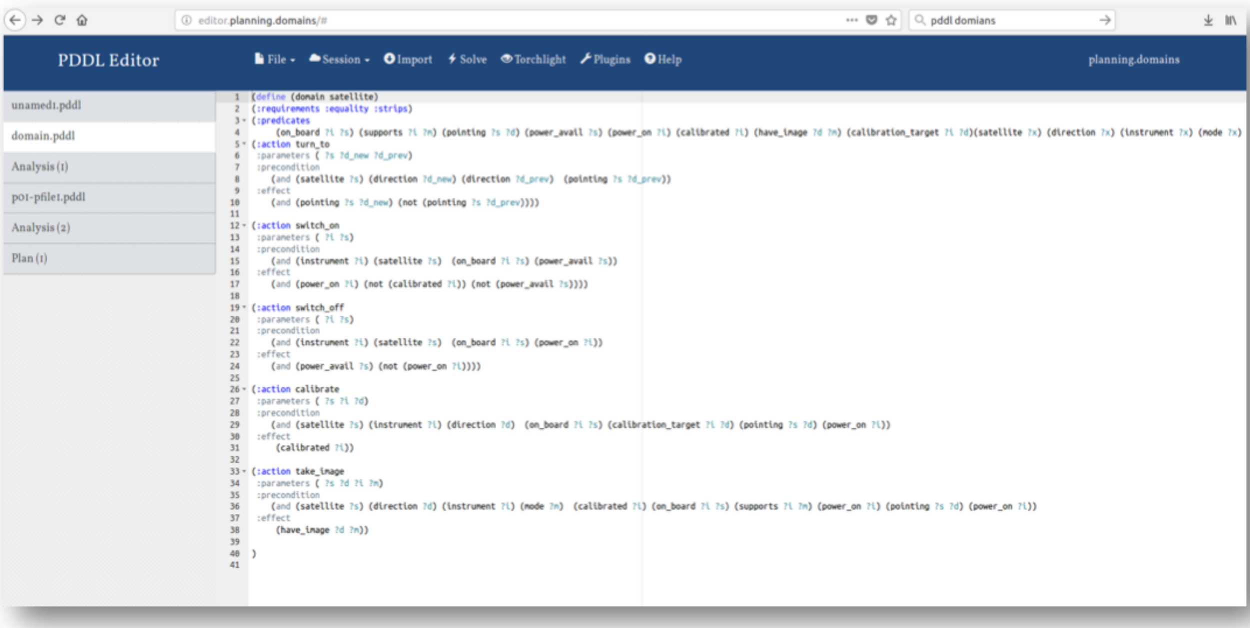
Some other KE frameworks

- **EUROPA** [Barreiro et al., 2012]
  - Framework supporting NDDL and ANML
- **JABBAH** [Gonzalez-Ferrer et al., 2009]
  - Supports HTN
- **KEWI** [Wickler et al., 2014]
  - Object Centred (including inheritance)
  - Web Application (supports collaboration)
- **VIZ** [Vodrážka & Chrpa, 2010]
  - A “light-weight” KE tool

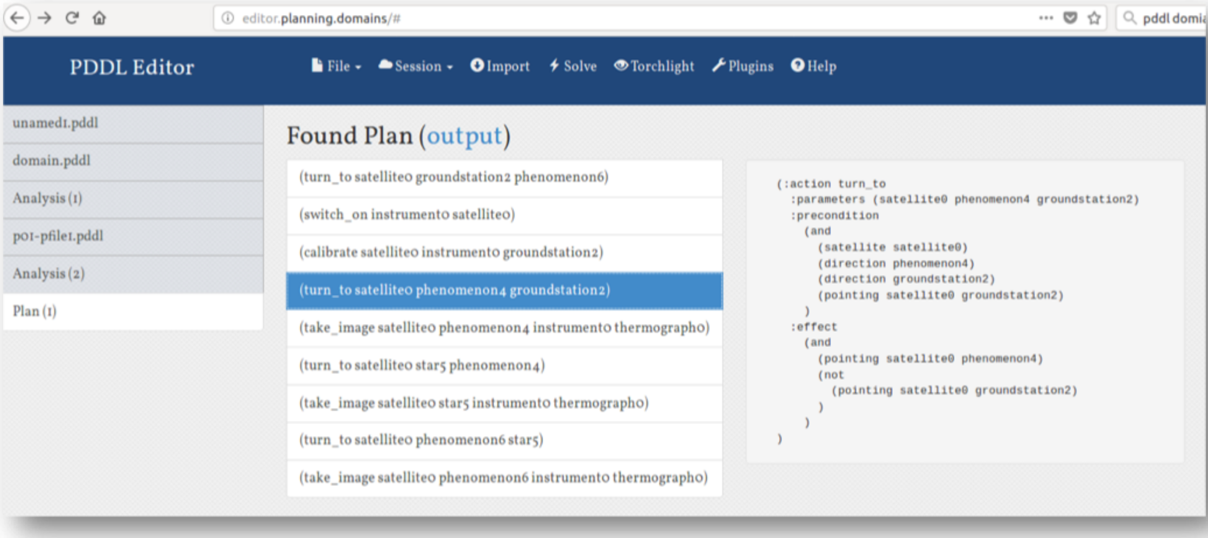
Planning.Domains

- “A Collection of Tools for Working with Planning Domains” [Muise]
- Web application
- Rich editor (syntax highlighting, autocomplete, etc.)
- Plug-in support
- Repository of all domains and problems from the IPCs

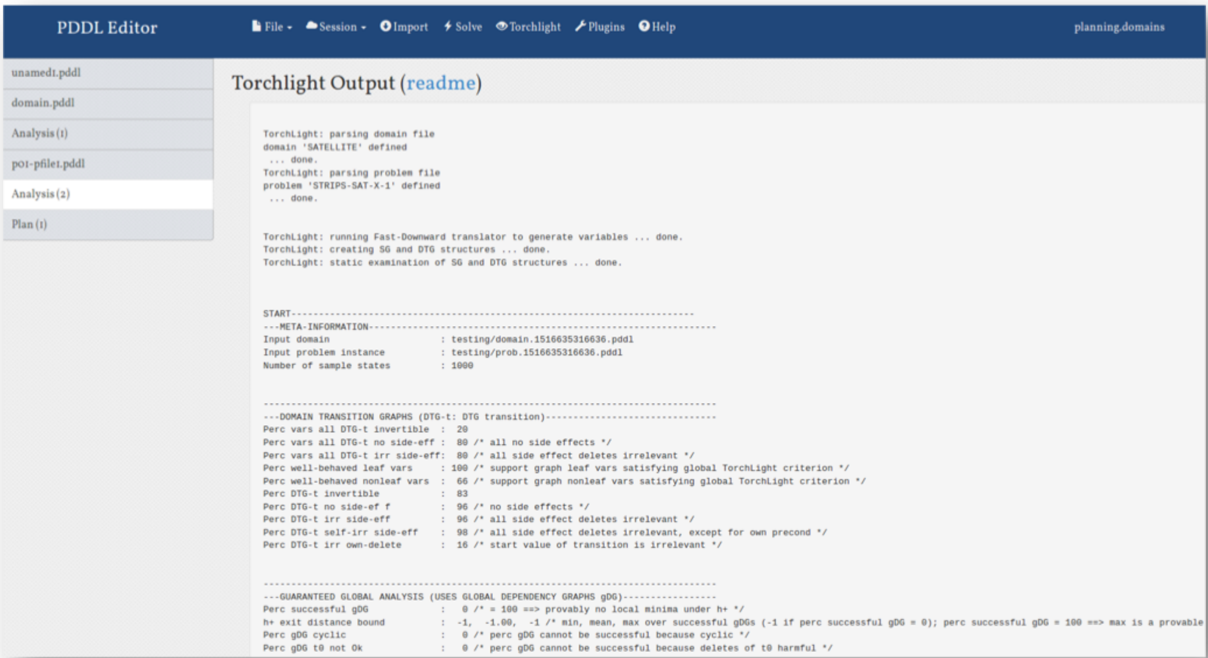
Planning.Domains – sample domain (Satellite)



Planning.Domains – sample plan (Satellite domain)



Planning.Domains – analysis (by TorchLight)



The Fifth International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS 2016)

ICKEPS mission

“Promote the knowledge-based and domain modelling aspects of AI P&S, to accelerate knowledge engineering research, to encourage the development and sharing of prototype tools or software platforms that promise more rapid, accessible, and effective ways to construct reliable and efficient P&S systems”

ICKEPS history

- **ICKEPS 2005** (San Francisco) - Tools and Tools Environments for KE
- **ICKEPS 2007** (Providence) - teams working (offline) on KE tasks and application scenarios
- **ICKEPS 2009** (Thessaloniki) - Tools for translating into planner-ready language from application-oriented language
- **ICKEPS 2012** (Sao Paulo) - teams working (offline) on KE tasks and application scenarios
- **ICKEPS 2016** (London) teams working (online) on KE tasks and application scenarios

ICKEPS 2016 roadmap

- **Pre-competition**
  - Organizers prepared 4 scenarios
    - 2 temporal (Star-trek, Roundabout)
    - 2 classical (RPG, Match Three Harry)
  - Organizers composed competition rules and evaluation criteria
- **On-site modelling**
  - Teams up to 4 members
  - 6 hours time limit for modelling
- **Demonstration**
  - 10 minutes per team to present their KE process
- **Board of Judges**
  - Deciding the winners

ICKEPS 2016 evaluation criteria

- **KE process**
  - Use of KE tools
  - Teamwork
- **Models**
  - Correctness
  - Generality
  - Readability
  - Planners' performance

ICKEPS 2016 key observations

- **It was fun !**
- Teams often selected easier domains to tackle (e.g. classical ones)
- Provided models were **different**, in some cases quite considerably
- Interesting modelling approaches – e.g. analysing domain transition graph to identify "bad" states
- Not many KE tools were exploited
  - The winning team (Muise & Lipovetzky) exploited the Planning.Domains framework

RPG domain – some observations

- According to the specification the hero dies if:
  - does not have a sword and enters a room with a monster
  - destroys the sword in a room with a monster
  - in a room with a trap, the hero performs any other action than “disarm” (for this action the hero must be empty handed)

- The competitors observed:
  - the hero must have a sword in order to enter a room with a monster
  - the hero must be empty handed to enter a room with a trap

RPG domain – some observations

- The models do not explicitly consider hero's death
- Some Planning Operators encoded in the models:
  - move-without-sword
  - move-with-sword
  - destroy-sword-move-disarm
  - …
- Models were rather **“planner-friendly”** than “user-friendly”

- **Modelling oriented** rather than KE tools oriented
- **Practical applications**
  - Combine offline and on-site modelling
- Get more competing teams
  - 6 teams competed on ICKEPS 2016
- Automatize the model evaluation process
- Attract interest outside “planning” community
  - “expert bias” can be mitigated
- ...

Part III.

DESIGNING AND DEVELOPING A DOMAIN MODEL

15-Puzzle

4		3	6
12	1	11	7
9	5	10	15
13	8	14	2

Initial state

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Goal state

State representation

Position of gap — Position of 1 — Position of 2

```
main =>
    Init = [(1,2),(2,2),(4,4),(1,3),(1,1),(3,2),(1,4),(2,4),
            (4,2),(3,1),(3,3),(2,3),(2,1),(4,1),(4,3),(3,4)],
    best_plan(Init,Plan).

final(S) => S = [(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),
                 (3,1),(3,2),(3,3),(3,4),(4,1),(4,2),(4,3),(4,4)].
```

15-Puzzle: actions

```
action([P0@(R0,C0)|Tiles],NextS,Action,Cost) =>
    Cost = 1,
    (R1 = R0-1, R1 >= 1, C1 = C0, Action = up;
     R1 = R0+1, R1 =< 4, C1 = C0, Action = down;
     R1 = R0, C1 = C0-1, C1 >= 1, Action = left;
     R1 = R0, C1 = C0+1, C1 =< 4, Action = right),
    P1 = (R1,C1),
    slide(P0,P1,Tiles,NTiles),
    NextS = [P1|NTiles].

% slide the tile at P1 to the empty square at P0
slide(P0,P1,[P1|Tiles],NTiles) =>
    NTiles = [P0|Tiles].
slide(P0,P1,[Tile|Tiles],NTiles) =>
    NTiles=[Tile|NTilesR],
    slide(P0,P1,Tiles,NTilesR).
```

15-Puzzle: heuristics and performance

Heuristic function

```
heuristic(Tiles) = Dist =>
    final([_|FTiles]),
    Dist = sum([abs(R-FR)+abs(C-FC) :
             {(R,C),(FR,FC)} in zip(Tiles,FTiles)]).
```

Performance

- Picat planner easily solves 15-puzzle instances
- It can even solve some hard 24-puzzle instances if a better heuristic is used

NoMystery problem

A truck moves between locations to pickup and deliver packages while consuming fuel during moves.

- setting:
  - initial locations of packages and truck
  - goal locations of packages
  - initial fuel level, fuel cost for moving between locations
- possible actions: **load**, **unload**, **drive**
- assumption: track can carry any number of packages

Nomystery: state representation

Factored representation

– state = a set of atoms that hold in that state (a vector of values of state variables)

```
{at(p0,l2),at(p1,l2),at(p2,l1),at(t0,l2),
 in(p3,t0),in(p4,t0),in(p5,t0),
 fuel(t0,level84)}
```

Structured representation

– state = a term describing objects and their relations

objects represented by properties rather than by names to break object symmetries

```
s(l2, level84, [l2,l2,l4], [[l1|l3],[l2|l3],[l2|l4]])
```

truck location

fuel level

destinations of loaded packages

current and desired locations of waiting packages

Nomystery: actions

Factored representation

```
action(S,NextS,Act,Cost),
   truck(T), member(at(T,L),S),
   select(at(P,L),S,RestS), P != T
?=>
   Act = load(L,P,T), Cost = 1,
   NewS = insert_ordered(RestS,in(P,T)).
```

Structured representation

```
action(s(Loc,Fuel,LPs,WPs),NextS,Act,Cost),
   select([Loc|PkGoal],WPs,WPs1)
?=>
   Act = load(Loc,PkGoal), Cost = 1,
   LPs1 = insert_ordered(LPs,PkGoal),
   NextS = s(Loc,Fuel,LPs1,WPs1).
```

Estimate distance to goal

Precise heuristic for Nomystery domain:

- each package must be loaded and unloaded
- each place with packages to load or unload must be visited

```
action(S,NextS,Act,Cost),
  truck(T), member(at(T,L),S),
  select(at(P,L),S,RestS), P != T
?=>
  Act = load(L,P,T), Cost = 1,
  NewS = insert_ordered(RestS,in(P,T)),
  heuristics(NewS) < current_resource().
```


Tell the planner what to do at a given state based on the goal

- unload all packages destined for current location (and only those packages)

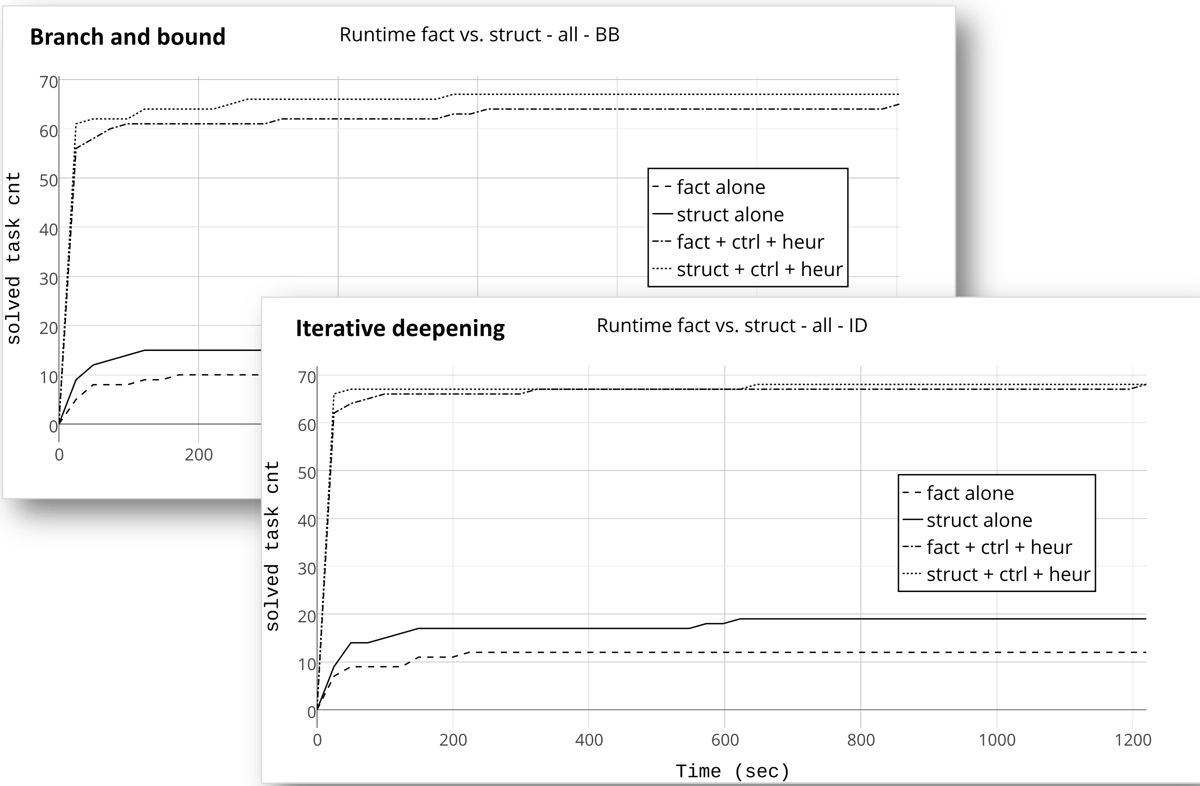
```
action(s(Loc,Fuel,LoadedPks,WaitPks), NextState, Action, Cost),
       select(Loc,LoadedPks,LoadedPks1)
=>
       Action = unload(Loc,Loc),
       NextState = s(Loc,Fuel,LoadedPks1, WaitPks),
       Cost = 1.
```

- load all undelivered packages at current location
- move somewhere
  - move to a location with waiting package or to a destination of some loaded package

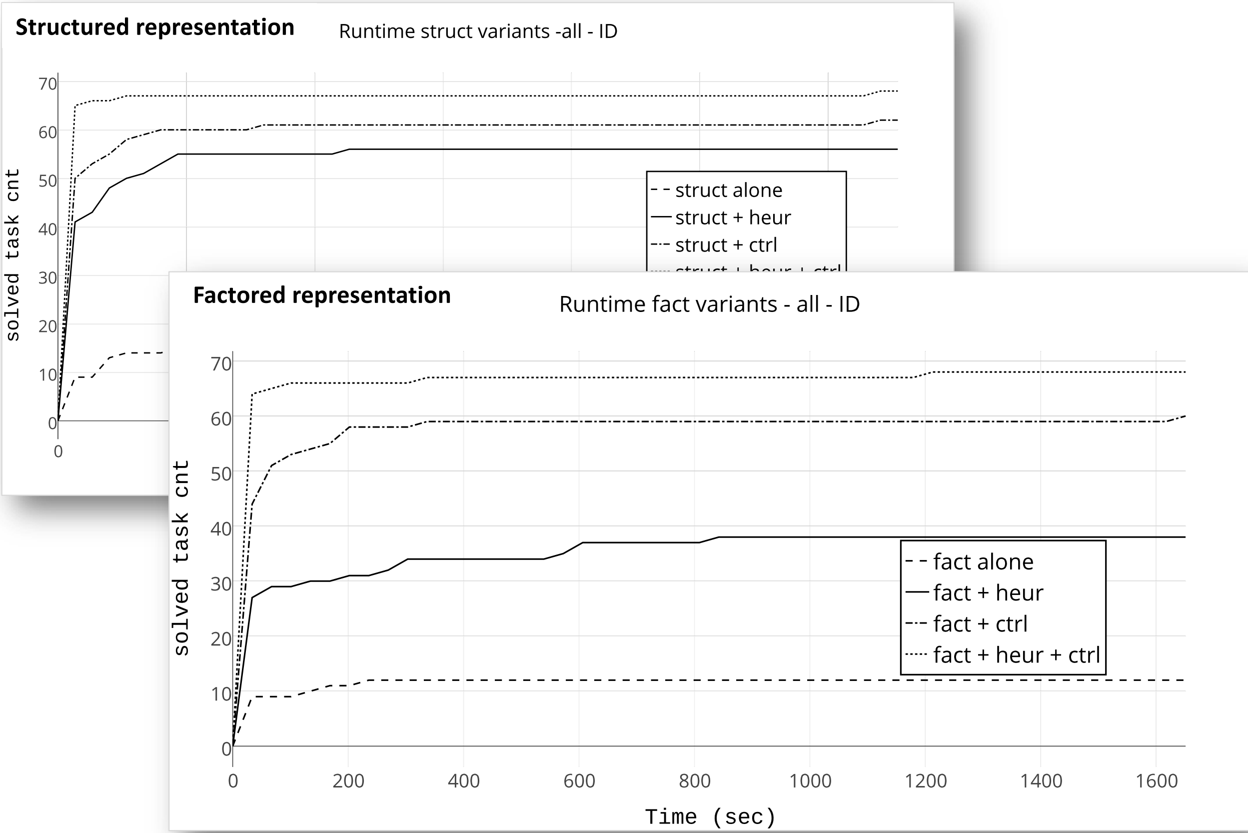
NoMystery model

```
action(s(Loc,Fuel,LoadedCGs,Cargoes), NextState, Action, Cost),
    select(Loc,LoadedCGs,LoadedCGs1)
=>
    Action = unload(Loc,Loc),
    NextState = s(Loc,Fuel,LoadedCGs1,Cargoes), Cost = 1.

Action(s(Loc,Fuel,LoadedCGs,Cargoes), NextState, Action, Cost),
    select([Loc|CargoGoal],Cargoes,Cargoes1)
=>
    insert_ordered(CargoGoal,LoadedCGs,LoadedCGs1),
    Action = load(Loc,CargoGoal),
    NextState = s(Loc,Fuel,LoadedCGs1,Cargoes1) , Cost = 1.

Action(s(Loc,Fuel,LoadedCGs,Cargoes), NextState, Action, Cost)
?=>
    Action = drive(Loc,Loc1),
    NextState = s(Loc1,Fuel1,LoadedCGs,Cargoes),
    fuelcost(FuelCost,Loc,Loc1),
    Fuel1 is Fuel-FuelCost,
    Fuel1 >= 0, Cost = 1.
```

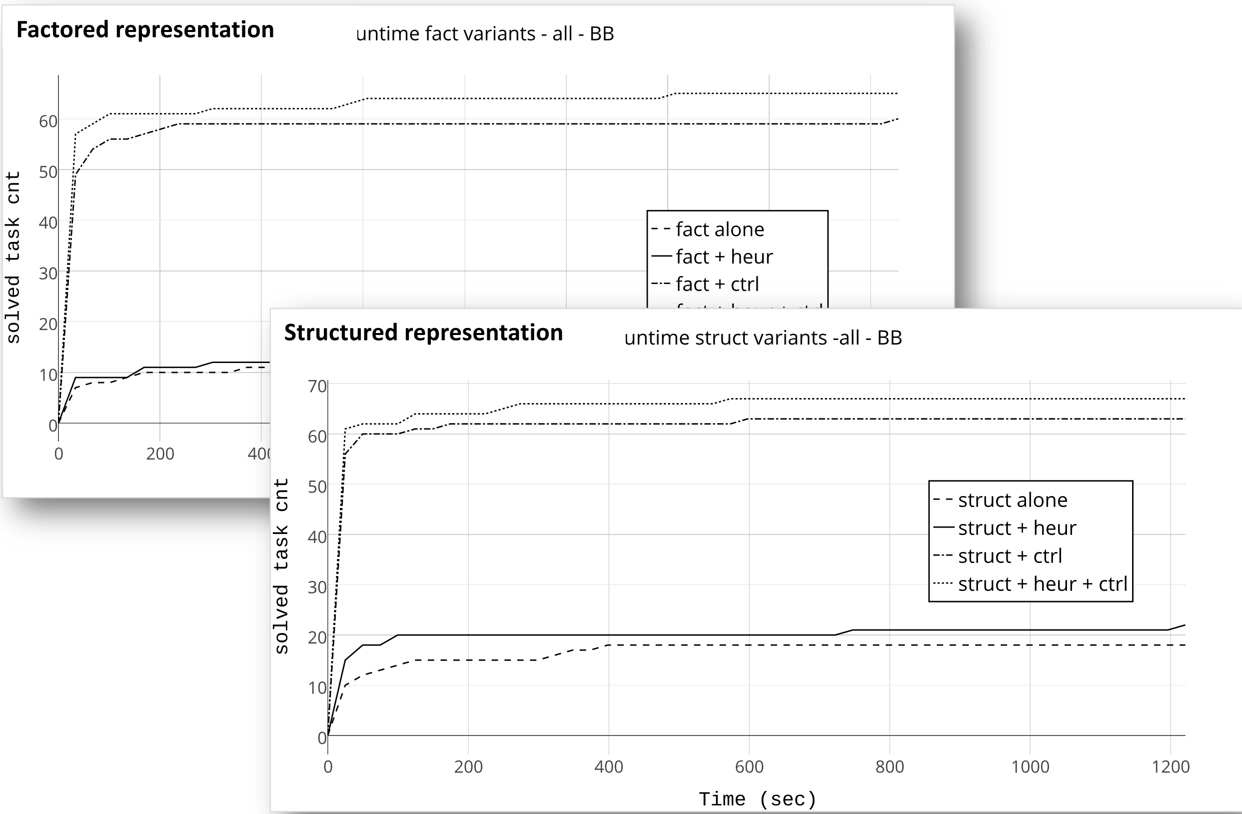
Factored vs. structured representations



Heuristics vs. control knowledge (ID)



Heuristics vs. control knowledge (B-and-B)



- using **structured representation** of states instead of factored representation
  - object symmetry breaking
- **control knowledge** helps more than **heuristics**
- **heuristics** are more important for iterative-deepening than for branch-and-bound
- **control knowledge** is critical for branch-and-bound

Modelling Road Traffic Accident Management Domain: Exploring KE Strategies

In collaboration with University of Huddersfield

[Shah et al., 2013]

Motivation for the study

- **No standard** modelling **procedure** (so far)
- **Domain modelling** is **ad-hoc** and depends on **planning expert's knowledge**
- Little knowledge about **how** existing **KE tools influence the modelling process**
- We **investigated two KE methods**
  - Hand-coding
  - Using KE tool (ItSimple)

Road Traffic Accident Management (RTAM) domain

- RTAM domain deals with situations that raise immediately after traffic accident(s) are reported
- Requirements
  - The accident site has to be secured
  - Accident victims have to be released from damaged vehicles and taken to hospitals
  - Damages vehicles have to be towed away
- Situations have to be "sorted out" asap

Method A: hand-coding

- A **planning expert** uses a (plain) **text editor** to encode the RTAM domain
- **Validate** the model on **several** (easy) **problem instances**
- In case of any issue (e.g. incorrect plan or no plan at all), **fix the model**
- Repeat until no issue remains

Method B: ItSimple

- **ItSimple** [Vaquero et al, 2007] uses **UML standards** for **domain modelling**
- Design of **class diagrams**
- Definition of **state machines**
- Translation of **UML models** to **PDDL**
- Validation of the model on several (easy) problem instances (as in Method A)

Formal conceptualization - objects

- Assets (*X*)
  - Static Assets (X_S)
  - Mobile Assets (X_M)
- Artifacts (*Y*)
- Locations (*L*)
- Properties (*P*)
  - Characterizing a state of assets and/or artifacts

Formal conceptualization – relations and invariants

- *loc:* X→ L ∪ {⊥} (asset's location)
- *connected* ⊆ L×L (locations are directly connected)
- *in:* Y→ X ∪ L ∪ {⊥} (artifact "attached" to an asset or a location)
- cap: X→ ℕ (asset's capacity)
- *Property* ⊆ X ∪ Y× P (properties of assets and artifacts)

- An asset *x* can have at most *cap(x)* artifacts attached to it at the same time.
- Static assets have constant location.

- **Move**
  - Moves a mobile asset from one location to another
- **Attach**
  - Attaches an artifact to an asset
- **Detach**
  - Detaches an artifact to an asset
- **Interact**
  - Changes properties of assets/artifacts
  - First-aid, Extinguish-fire, Secure-location, Release-victim

Move (x_m, l_1, l_2)

Precondition:

At start: $loc(x_m)=l_1$

Over all: $(l_1, l_2) \in connected$

Effects:

At start: $loc(x_m)=\perp$

At end: $loc(x_m)=l_2$

Attach (y,x,l)

Precondition:

At start: *in(y)=l*

Over all: loc(x)=l, $|\{y' \mid in(y')=x\}| \leq cap(x)$

Effects:

At start: in*(y)*=⊥

At end: in*(y)= x*

Formal conceptualization - actions

Detach (y,x,l)

Precondition:

At start: *in(y)=x*

Over all: *loc(x)=l*

Effects:

At start: *in(y)*=⊥

At end: *in(y)= l*

Formal conceptualization - actions

Interact $(e_1,e_2,l,p_1,p_2,p_3,p_4,p_5,p_6)$

Precondition:

At start: $(e_1,p_1) \in$ *property*, $(e_2,p_2) \in$ *property*

Over all: $loc(e_1)=l \lor in(e_1)=l$, $loc(e_2)=l \lor in(e_2)=l$

Effects:

At start: $(e_1,p_1) \notin$ *property*, $(e_2,p_2) \notin$ *property*, $(e_1,p_3) \in$ *property*, $(e_2,p_4) \in$ *property*

At end: $(e_1,p_3) \notin$ *property*, $(e_2,p_4) \notin$ *property*, $(e_1,p_5) \in$ *property*, $(e_2,p_6) \in$ *property*

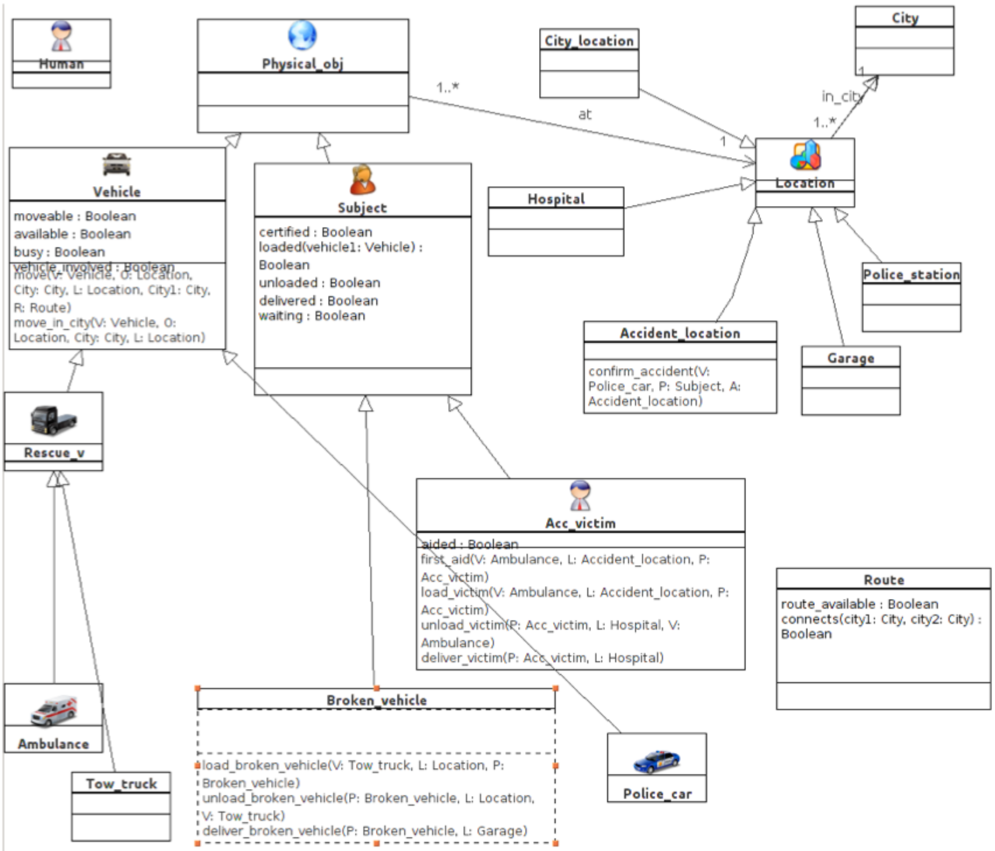
Considered models for evaluation

- “Classical” model
  - Typed Strips PDDL
- Temporal model
  - PDDL 2.1
  - Durative actions

A sample action in PDDL 2.1

```
(:durative-action release-victim
        :parameters (?V - fire_brigade ?P - acc_victim
                          ?A - accident_location)
        :duration (= ?duration (releasing-time))
        :condition (and
             (at start (at ?P ?A))
             (at start (at ?V ?A))
             (at start (certified ?P))
             (at start (available ?V))
             (at start (waiting ?P))
             (at start (trapped ?P))
        )
        :effect (and
             (at start (not (available ?V)))
             (at end (not (trapped ?P)))
             (at end (available ?V))
         )
)
```

Class diagram in ItSimple



Evaluation

- Inspired by **software engineering** evaluation criteria
- **Process**
  - From the domain conceptualization to the final domain model
- **Project**
  - Project execution and resources needed
- **Product**
  - Quality of the produced domain model

Process

Method A

- Depends on skills and judgment of the expert
- Hard to replicate
- Can be used with any language (e.g. PDDL, ANML, Picat)

Method B

- ItSimple supports a “disciplined” design cycle
- The process can be repeated
- Can be used only with limited number of languages/features

Project

Method A

- Domain modelling took around 2 man-days
- Issues in the model and hard to spot
- Most of the time was spent on debugging

Method B

- Domain modelling took around 3 man-days
- Issues in the model are easier to spot
- Most of the time was spent on model design

Product

Method A

- More preconditions and effects per operator
- Harder to "read"
- Slightly less "planner-friendly" (LPG was slower)

Method B

- More operators (releasing victims and extinguishing fire was split into two operators each)
- Easier to "read" (the UML diagrams, not the generated PDDL !)
- Slightly more "planner-friendly" (LPG was faster)

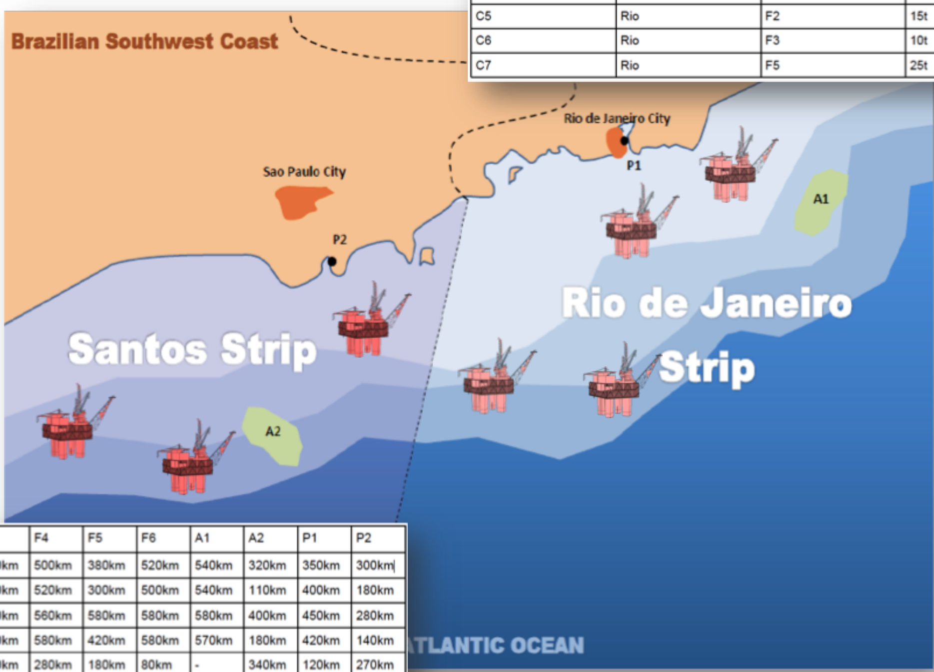
- The **best strategy** for generating readable and easy to maintain models is the use of **KE tools**
  - Limited language/features support
  - “Expert bias”
- **Decision** what **language** and what **features** will be used must be done **early** (before formal conceptualization)
- There is **no strategy** (yet) for developing **“planner-friendly” domain models**

Part IV.

DEVELOPMENT OF REAL-WORLD PLANNING APPLICATION

- one of the challenge problems at ICKEPS 2012
- transporting **cargo** items between **ports** and petroleum **platforms** while assuming limited capacity of **vessels** and fuel consumption during transport
- basic operations:
  - navigating, docking/undocking, loading/unloading, refueling
- objectives:
  - fuel consumption, makespan, docking cost, waiting queues, the number of ships

cargo specification	loading port	delivering point	weight
C1	Rio	F1	20t
C2	Rio	F6	5t
C3	Rio	F4	15t
C4	Santos	G4	8t
C5	Rio	F2	15t
C6	Rio	F3	10t
C7	Rio	F5	25t



	F1	F2	F3	F4	F5	F6	A1	A2	P1	P2
G1	468km	580Km	420km	500km	380km	520km	540km	320km	350km	300km
G2	580km	468km	380km	520km	300km	500km	540km	110km	400km	180km
G3	588km	600km	420km	560km	580km	580km	580km	400km	450km	280km
G4	600km	588km	580km	580km	420km	580km	570km	180km	420km	140km
A1	200km	40km	320km	280km	180km	80km	-	340km	120km	270km
A2	340km	380km	370km	340km	280km	300km	340km	-	270km	100km
P1	300km	160km	280km	200km	160km	130km	120km	270km	-	200km
P2	380km	290km	320km	340km	270km	300km	270km	100km	200km	-

Petrobras - existing approaches

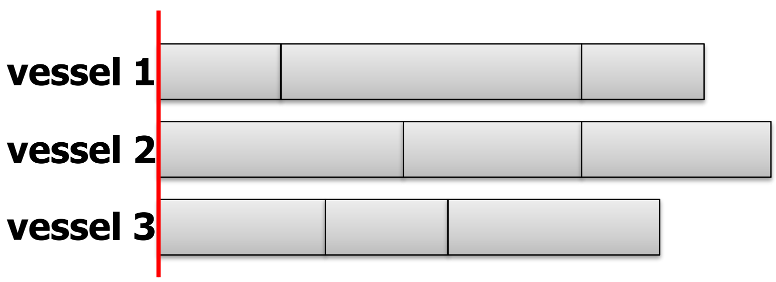
- **Classical planning**
  - the planning part (decision of actions) modeled in PDDL 2.1 and solved by SGPlan (optimize fuel)
  - the scheduling part (time allocation) solved in post-processing
- **Temporal planning**
  - modeled completely in PDDL 2.1 (durative actions and resources)
  - solved using the Filuta planner (optimize makespan)
- **Monte Carlo Tree Search**
  - using abstract actions (Load, Unload, Refuel, GoToWaiting)
  - solved using MCTS (optimize "usedFuel + 10 ∗ numActions + 5 ∗ makespan")

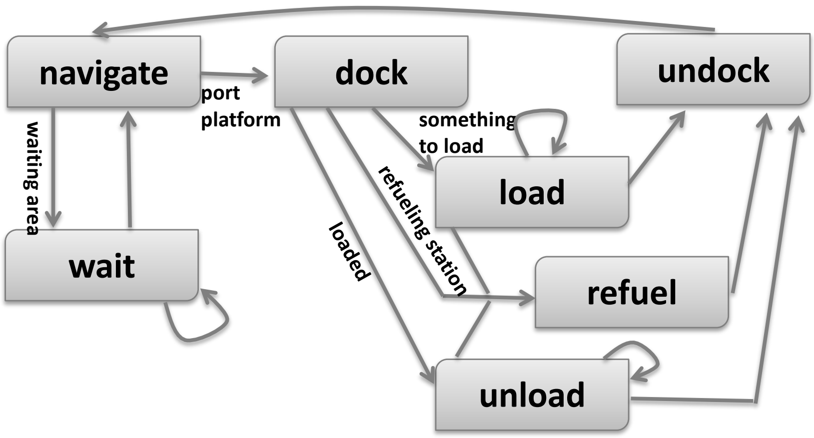
Petrobras integrated (B-Prolog) – states

- Each vessel modeled separately as a *timeline* (sequence of actions)

  ```
  [Start,Fuel,Action,Loc,LoadedCargo,Dur]
  LoadedCargo = [Weight,CargoLoc,Dest]
  ```

- left-to-right scheduling with rolling horizon

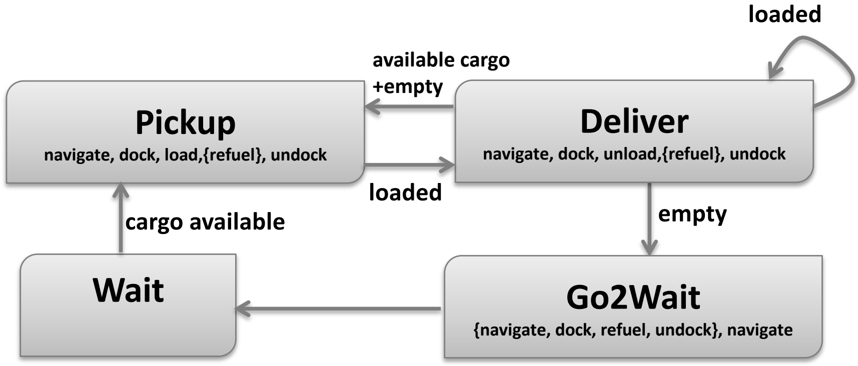




This does not work!

- more vessels heading for the same cargo (but only the first vessel will load it)
- useless planned actions (just to do something – refueling)

Exploiting *macro actions*, *landmarks* (cargo must be picked up), *control rules*, *heuristics*



Petrobras separated (Picat) - states

- **Solving approach:**
  - separate planning (fuel optimization) from scheduling (time allocation, makespan)
  - separate route selection from cargo-to-deliver selection
- **State representation:**
  - cargo Items: `[[OriginLoc, [DestinationLoc, Weight1,Weight2,...]], ...]`
  - vessels: `[[Location, FuelLevel1, FuelLevel2,...], ...]`

  Removes symmetries between items and vessels.

Petrobras separated - main loop

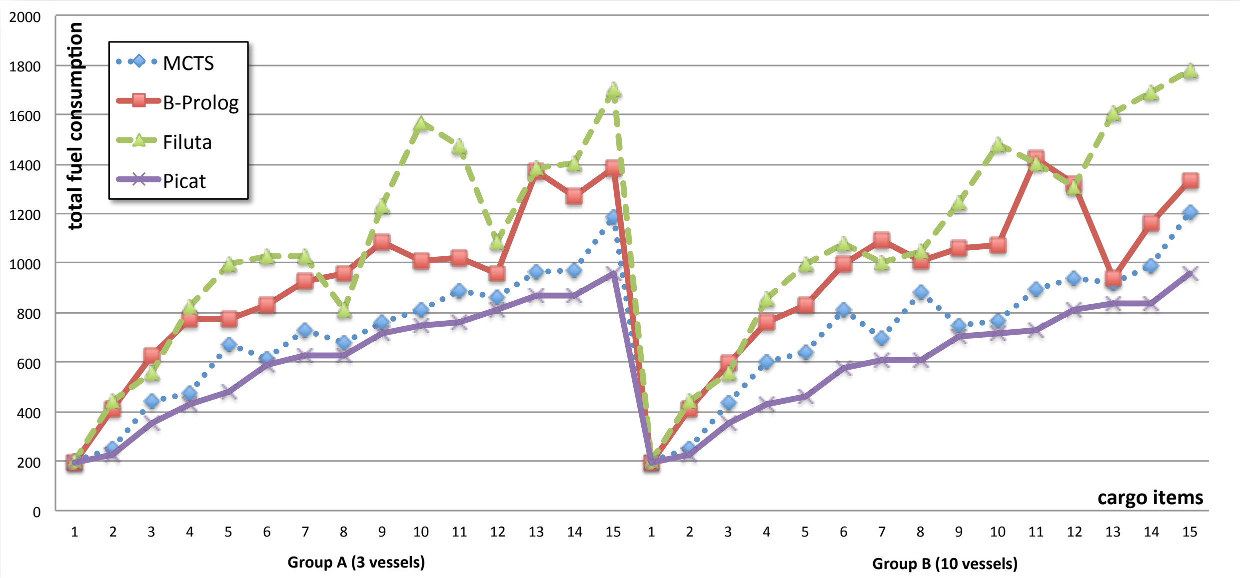
```
table (+,+, -,min)
plan([], _Vessels, Plan, Fuel) =>
    Plan = [], Fuel = 0.
plan(Cargo, Vessels, Plan, Fuel) =>
    select_port(Cargo, Port, PortCargo, RestCargo),
    select_cargo(PortCargo,Destinations,FreeCap,RestPortCargo),
    select_and_move_vessel(Vessels, Port, FuelLevel1,
    RestVessels, Plan1, Fuel1),
    load_at_other_ports(RestCargo, Port, FreeCap, FuelLevel1,
    Destinations2, RestCargo2, Port2, FuelLevel2, Plan2,
    Fuel2),
    path_plan(Port2, FuelLevel2, Destinations ++ Destinations2,
        FinalLoc, FinalLevel, Plan3, Fuel3),
    plan(addCargo(RestCargo2, Port, RestPortCargo),
        addVessel(RestVessels, FinalLoc, FinalLevel),Plan4,Fuel4),
    Plan = Plan1 ++ $[load(Port),undock(Port)] ++ Plan2
            ++ Plan3 ++ Plan4,
    Fuel = Fuel1 + Fuel2 + Fuel3 + Fuel4.
```

- The challenge problem from ICKEPS 2012
  - 10 vessels with fuel capacity 600l, 15 cargo items
- Random problems from ICTAI 2012
  - varying the number of vessels, fuel capacity:
    - ***Group A* – 3 vessels, fuel tank capacity 600 liters**
    - ***Group B* – 10 vessels, fuel tank capacity 600 liters**
  - varying the number of items (1-15) in each group
- Comparison of
  - temporal planner FILUTA
  - MCTS planner
  - B-Prolog planner
  - Picat planner

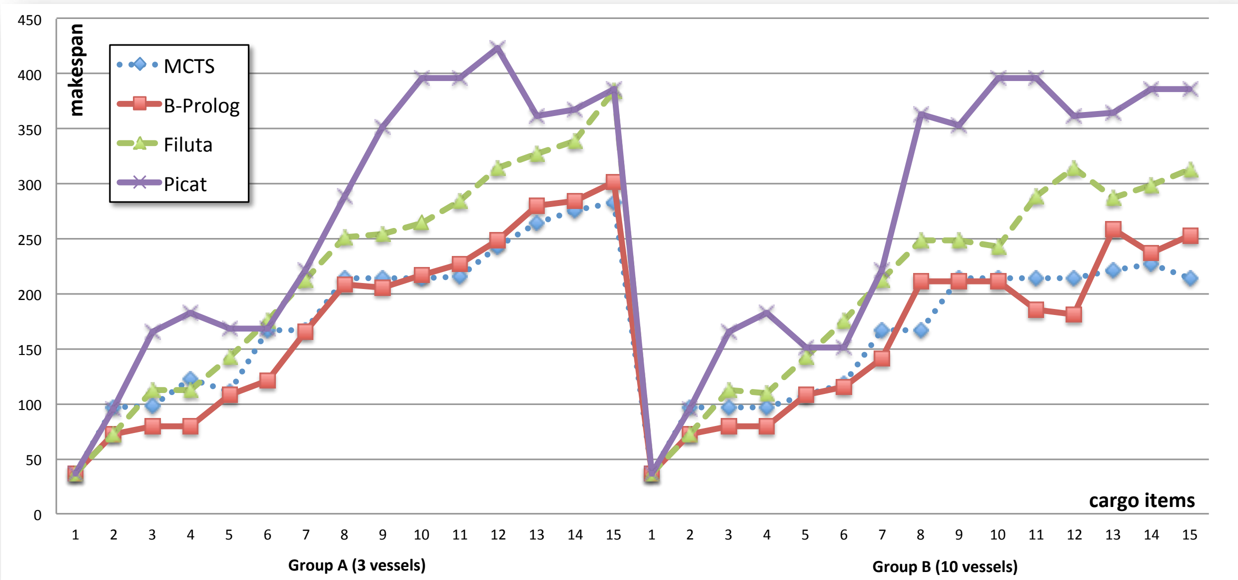
System	Optimization Criteria			
	Fuel (l)	Makespan (h)	Vessels	Runtime (ms)
B-Prolog	1263	**162**	4	~60 000
Filuta	1989	263	4	~600 000
MCTS	887	204	5	~600 000
Picat	**812**	341	**3**	**813**

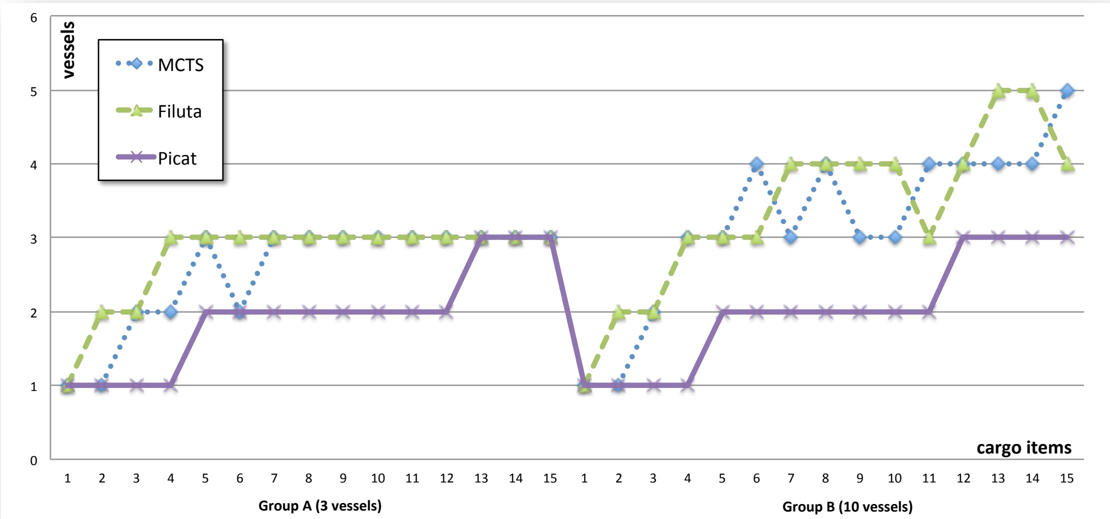
10 vessels with fuel capacity 600l, 15 cargo items

Petrobras results: fuel consumption



Petrobras results: makespan





Mixed-initiative Task Planning for Autonomous Underwater Vehicles

In collaboration with LSTS lab, University of Porto

[Chrpa et al., 2015;2017]

System requirements [Chrpa et al., 2015]

- Necessity to control multiple heterogeneous Autonomous Underwater Vehicles (AUVs)
- An operator (human) specifies high-level tasks (e.g. “sample an object with ctd camera”)
- Task assignment to each AUV should be automatized

How task assignment can be automatized ?

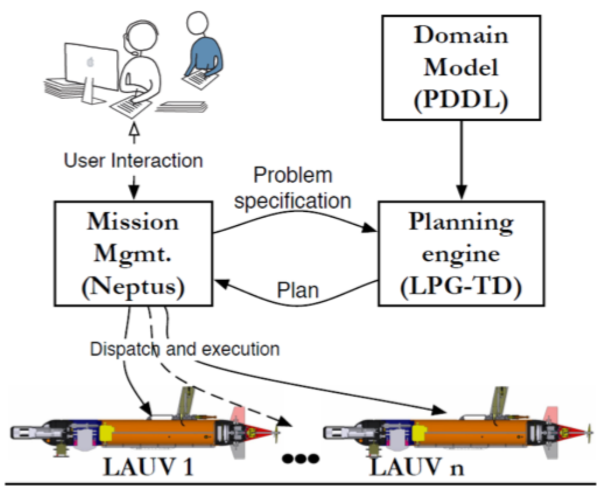
- Each **task** has specific **requirements**
- Each **vehicle** has specific **capabilities**
- For completing tasks AUVs have to perform certain sequences of **actions**
- Hence, we need to find **a plan** that if executed, the AUVs will complete all given tasks

Available "machinery"

- In LSTS, AUVs are controlled via **NEPTUS** (a decision support tool with GUI) and **DUNE** (onboard vehicle control) → **"low-level" control**
- Domain-independent **AI planning** (i.e., finding a sequence of actions that achieves a defined goal) → **"high-level" task planning**
  - **PDDL**, a language for specifying planning domain models and problem instances
  - **LPG-td**, a planning engine accepting domain and problem descriptions in PDDL and returning a plan (if exists)

Modular architecture

- **User specifies tasks** in NEPTUS
- NEPTUS **generate a planning problem** and sends it to LPG-td
- LPG-td **returns a plan** to NEPTUS
- NEPTUS **distributes the plan** to each of the vehicles



"High-level" specification

- Each **AUV** has certain **payloads** attached to it
- Each task must be completed by using a certain **payload** (e.g. camera, sidescan)
- Each AUV has a limited amount of **energy** that is consumed by executing actions
- Collected data can be **communicated** while an AUV is in its "depot" (a "safe spot" close to shore/ship)
- Two (or more) AUVs cannot be at the same **location** or perform the same **task** simultaneously

Formal conceptualization - objects

- Vehicles (*V*)
- Payloads (*P*)
- Phenomenons (*X*)
- Tasks (*T*)
- Locations (*L*)

- *at* $\subseteq V \times L$ (vehicle's location)
- *base* $\subseteq V \times L$ (vehicle's "depot")
- *has* $\subseteq V \times P$ (attached payloads to the vehicle)
- *at-phen* $\subseteq X \times L$ (phenomenon's location)
- *task* $\subseteq T \times X \times P$ (task description)
- *sampled* $\subseteq T \times V$ (acquired task data by vehicle)
- *data* $\subseteq T$ (acquired task data by the control centre)

- *dist:* $L \times L \rightarrow \mathbb{R}^+$ (distance between locations)
- *survey-dist:* $L \times L \rightarrow \mathbb{R}^+$ (length of survey)
- *speed:* $V \rightarrow \mathbb{R}^+$ (vehicle's speed)
- *battery-level:* $V \rightarrow \mathbb{R}^+$ (vehicle's battery level)
- *battery-use:* $V \cup P \rightarrow \mathbb{R}^+$ (vehicle's or payload's energy consumption)

Move (v,l1,l2)

Duration: *d=dist(l1,l2)/speed(v)*

Precondition:

At start: *(v,l1)∈at, battery-level(v)≥ d*battery-use(v)*

At end: *∄v'≠v: (v',l2)∈at*

Effects:

At start: *(v,l1)∉at, battery-level(v)=battery-level(v)-d*battery-use(v)*

At end: *(v,l2)∈at*

Sample (v,t,x,p,l)

Duration: *d=60* (constant duration)

Precondition:

At start: *battery-level(v)≥ d*battery-use(p)*

Overall: *(v,l)∈at, (x,l)∈at-phen, (v,p)∈has, (t,x,v)∈task*

Effects:

At start: *battery-level(v)=battery-level(v)-d*battery-use(p)*

At end: *(t,v)∈sampled*

Survey (v,t,x,p,l1,l2)

Duration: *d=survey-dist(l1,l2)*

Precondition:

At start: *(v,l1)∈at, battery-level(v)≥ d*(battery-use(v)+battery-use(p))*

Overall: *(x,l1)∈at-phen, (x,l2)∈at-phen, (v,p)∈has, (t,x,v)∈task*

Effects:

At start: *(v,l1)∉at,*
battery-level(v)=battery-level(v)-d(battery-use(v)+battery-use(p))*

At end: *(v,l2)∈at, (t,v)∈sampled*

No concurrent survey action can be executed over *x*

Collect-data (v,t,l)

Duration: *d=60* (constant duration)

Precondition:

Overall: *(v,l)∈at, (v,l)∈base,(t,v)∈sampled*

Effects:

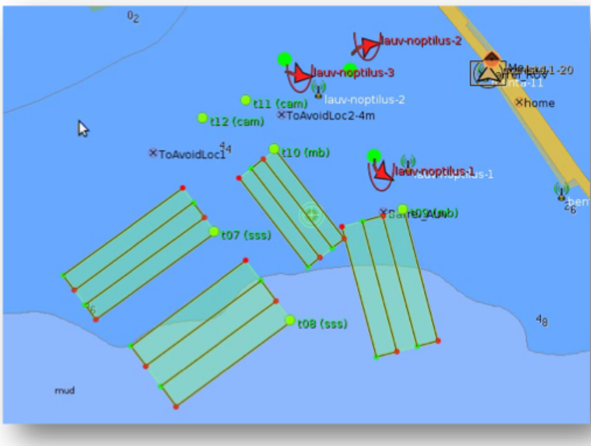
At end: *t∈data*

PDDL model of the Sample action

```
(:durative-action sample
 :parameters (?v - vehicle ?l - location ?t -task
              ?o - phenomenon ?p - payload)
 :duration (= ?duration 60)
 :condition (and (over all (at-phen ?o ?l))
                 (over all (task ?t ?o ?p))
                 (over all (at ?v ?l))
                 (over all (has ?p ?v))
                 (at start (>= (battery-level ?v)
                               (* (battery-use ?p) 60))))
 :effect (and (at end (sampled ?t ?v))
              (at start (decrease (battery-level ?v)
              (* (battery-use ?p) 60)))) )
```

Execution of the model: settings

- Evaluated in Leixões Harbour, Porto
- 3 light AUVs carrying different payloads
- In phase one, areas of interest were surveyed
- In phase two, contacts identified in phase one were explored



Planned vs. execution time

- The plans were **executable**
- **High discrepancies**, especially for move and survey actions
- **Rough time predictions** that were done only on distance and type of vehicle

Vehicle	Action	Time Difference
Noptilus-1	move survey sample communicate	47.80 ± 49.11 23.15 ± 23.26 1.33 ± 0.58 0.16 ± 0.17
Noptilus-2	move survey sample communicate	39.57 ± 35.66 107.88 ± 141.10 N/A 0.25 ± 0.07
Noptilus-3	move survey sample communicate	59.90 ± 57.05 24.00 ± 0.00 9.57 ± 13.64 0.11 ± 0.16

Additional assumptions [Chrpa et al., 2017]

1) Users can add, remove or modify tasks during the mission

2) Vehicles might fail to execute an action

3) Communication with the control center is possible only when a vehicle is in its “depot”

Additional requirements for the system

- System has to be **flexible** (e.g. a user can add a new task) and **robust** (e.g. handling vehicles' failures)
- Dynamic Planning, Execution and Re-planning
  - Automatized response on task changes by user and/or exceptional circumstances during plan execution
- How the "one shot" model has to be changed?

Model amendments

- Removed *battery constraints*
  - vehicles' battery levels were much higher than duration of operations
- Added *maximum "away" time constraints*
  - Vehicles have to come to their depots to establish communication (if they are "away" communication might not be possible)
- Split the *move* action into *move-to-sample, move-to-survey, move-to-base,* the former two must be succeeded by *sample* and *survey* action respectively
- Optimizing plans (vehicles cannot go to locations they do not have anything to do)
- Modified representation of *phenomenons* (objects and areas of interests are explicitly distinguished)

Maximum "away" time constraints

- Numeric fluents
  - *from-base:* $V \rightarrow \mathbb{R}^+$ (how long the vehicle is "away")
  - *max-to-base:* $V \rightarrow \mathbb{R}^+$ (maximum "away"time)
- Preconditions (at start) of the *move, sample, survey* actions contain *(d* – action duration)*:*
  - *from-depot(v) ≤ max-to-depot(v) – d*
- Effects (at end) of the *move, sample, survey* actions contain *(d* – action duration)*:*
  - *from-depot(v) = from-depot(v) + d*
- Effects (at end) of the *move-to-base* action contain:
  - *from-depot(v)=0*

PDDL model of amended sample action

```
(:durative-action sample
:parameters (?v - vehicle ?l - location ?t -task ?o – oi
             ?p - payload)
:duration (= ?duration 60)
:condition (and (over all (at-oi ?o ?l))
                (over all (task ?t ?o ?p))
                (over all (at ?v ?l))
                (over all (has ?p ?v))
                (at start (<= (from-base ?v)
                              (- (max-to-base ?v) 60)))
           )
:effect (and (at end (sampled ?t ?v))
             (at end (can-move ?v))
             (at start (increase (from-base ?v) 60))
        )
)
```

Considered models

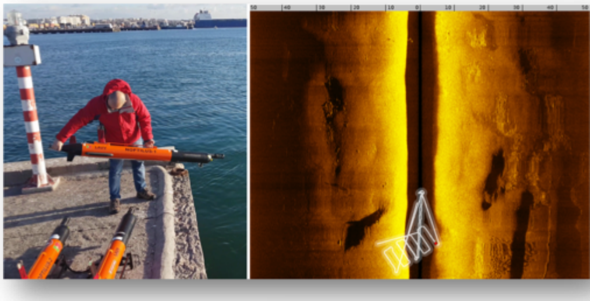
- **All Tasks**
  - Allocates all specified tasks to the vehicles
  - Minimizes the plan execution time and the number of vehicles' returns to their depots
- **One Round**
  - Allocates only tasks for the next "round" (i.e., after vehicles return to their depots they cannot move)
  - Maximizes the number of completed tasks

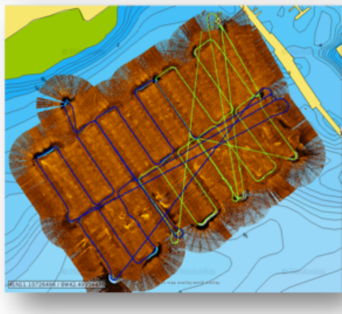
Execution

- **Preprocessing**
  - Splitting large surveillance areas into smaller ones
- **Planning**
  - NEPTUS generates a problem specification in PDDL, runs LPG-td, then processes and distributes the plan among the vehicles
- **Execution**
  - Each vehicle is responsible for executing its actions
  - Move actions are translated into timed-waypoints for mitigating the differences between planned and actual times
  - When in depots vehicles communicate status of completed tasks (success/failure) – failed tasks are "re-inserted"
- **Replanning**
  - If a new planning request comes (e.g. a user added a new task), vehicles continue to execute their current plans until they come back to their depots, then they receive new plans

Execution of the models: settings

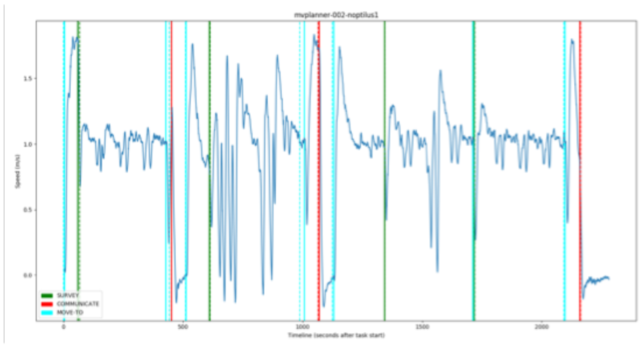
- Evaluated in Leixões Harbour, Porto
- Mine-hunting scenario was used
- 3 light AUVs, 2 carried sidescan, one carried camera
- In phase one, areas of interest were surveyed
- In phase two, contacts identified in phase one sampled to identify them as mines, or false positives





Results of the models execution

- Both models produced correct plans that were successfully executed
- During one of the executions one AUV (Noptilus 3) failed (depth sensor fault) – tasks were automatically re-inserted and allocated to a different AUV, which completed them
- All Tasks model produces better quality plans (for larger scenarios, however, One Round model might be more efficient)



- Most planned/actual differences are quite small (less than 3 seconds).
- Around time 1000 a noticeable difference occurred (vehicle had to ascend during the survey). The delay was eliminated by accelerating during the following move action.

Part V.

CLOSING REMARKS AND OPEN PROBLEMS

Summary

- **Domain model** is the key component for domain-independent planning
  - User-friendly (e.g., human readable)
  - Planner-friendly (e.g. planners are efficient)
- We have **languages** to describe **domain models**
- We have **planning engines** supporting those languages
- We have (some) **KE tools** supporting domain modelling

Good news

- Planning succeeded in many real-world applications
  - Space Exploration
  - Manufacture Planning
  - Narrative Generation
  - Task Planning for Autonomous Robots
  - Urban Traffic Control
  - …...

Not so good news

- A limited number of expressive planning engines
  - In IPC 2014, 67 planners participated, out of which only 6 competed in temporal track
- Domain modelling is still a **“black art”**
  - “Expert bias”
  - No guidelines (e.g. how to make model planner-efficient)
  - Limited tool support (e.g. debugging is still manual)
  - Lack of interest from the community

Questions to ask ourselves

- Do researches outside the planning community use domain-independent planning ?
- If not, why ?
  - Lack of guidelines for domain modelling
  - Lack of efficient and expressive planning engines
  - Lack of awareness
  - ….
- How can we motivate researches outside the planning community to use domain-independent planning in their research ?

Challenges

- The notion of **quality** of domain models
  - What it exactly stands for
  - How to assess it
- **KE tool** support
  - Debugging
  - Dynamic testing
  - Planner efficiency assessing
  - …
- Adopting **SW engineering** principles
  - Development life cycle
  - Collaboration
  - Maintenance
  - ….