A Simple Yet Efficient Lifted Formulation for Hard-to-Ground Planning Problems

Miquel Bofill[1] Cristina Borralleras[2] Josu Oca[1]

[1]Universitat de Girona

[2]Universitat de Vic - Universitat Central de Catalunya

The 25th workshop on Constraint Modelling and Reformulation (ModRef 2026)

Lisbon, Portugal
July 19th, 2026

Summary

Work in progress on **Planning as SMT**

based on **backward built lifted planning graphs**

with promising results on **hard-to-ground** problems

Planning Domains

Planning domains are defined in **first-order logic**

```
(define (domain blocksworld)
(:requirements :strips)
(:predicates (clear ?x)
             (on-table ?x)
             (arm-empty)
             (holding ?x)
             (on ?x ?y))

(:action pickup
  :parameters (?x)
  :precondition (and (clear ?x) (on-table ?x) (arm-empty))
  :effect (and (holding ?x) (not (clear ?x))
               (not (on-table ?x)) (not (arm-empty))))

(:action stack
  :parameters (?x ?y)
  :precondition (and (clear ?y) (holding ?x))
  :effect (and (arm-empty) (clear ?x) (on ?x ?y)
               (not (clear ?y)) (not (holding ?x)))))
```

Grounded Representations

But problems usually solved on **grounded representations**

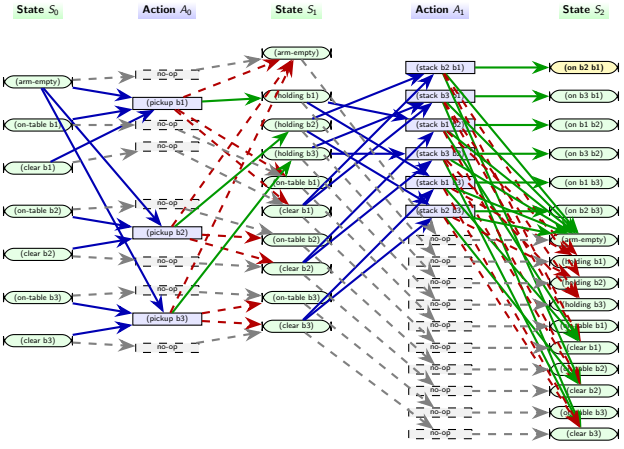
E.g., in planning as SAT, given objects b_1, b_2, b_3, ... we would have a formula like

$$arm\_empty^0$$
$$\wedge\ clear^0_{b_1} \wedge clear^0_{b_2} \wedge clear^0_{b_3} \ldots$$
$$\wedge\ on\_table^0_{b_1} \wedge on\_table^0_{b_2} \wedge on\_table^0_{b_3} \ldots$$
$$\wedge\ \neg holding^0_{b_1} \wedge \neg holding^0_{b_2} \wedge \neg holding^0_{b_3} \ldots$$
$$\wedge\ \neg on^0_{b_1,b_1} \wedge \neg on^0_{b_1,b_2} \wedge \neg on^0_{b_1,b_3} \wedge \cdots \wedge \neg on^0_{b_2,b_1} \ldots$$
$$\neg clear^1_{b_1} \rightarrow \neg clear^0_{b_1} \vee pickup^0_{b_1} \vee stack^0_{b_2,b_1} \vee stack^0_{b_3,b_1} \vee \ldots$$
$$\ldots$$

with lots of variables and constraints $\rightsquigarrow$ **combinatorial explosion**

Planning Graphs

Combinatorial explosion also occurs in **planning graphs**



Blocks world example, with putdown and unstack operators removed

Mitigating Combinatorial Explosion

Combinatorial explosion of grounding typically addressed by

- **relaxed reachability analysis + heuristics** (state space pruning), or
- **grounding on demand**, or
- working directly with a **lifted representation**

Mitigating Combinatorial Explosion

Combinatorial explosion of grounding typically addressed by

- **relaxed reachability analysis + heuristics** (state space pruning), or
- **grounding on demand**, or
- working directly with a **lifted representation**

Satisfiability Modulo Theories (SMT) is the problem of solving first-order logic formulas with respect to background theories

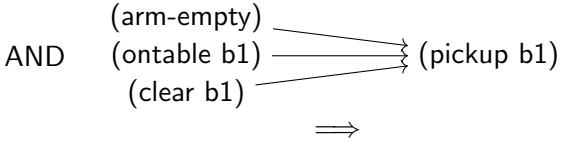
In SMT we can have formulas like

$$\neg f(x) \vee g(x) \vee (h(y, z) \wedge y = x)$$

Can this formalism help solve hard-to-ground planning problems?

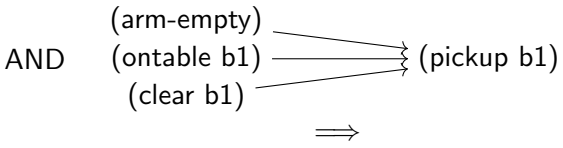
Forward vs Backward Planning Graph Construction

When building a planning graph, from ground facts we are going to get ground actions:

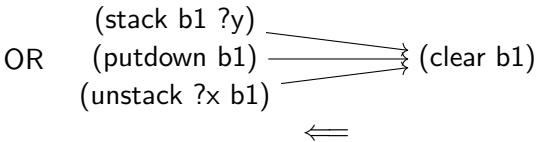


Forward vs Backward Planning Graph Construction

When building a planning graph, from ground facts we are going to get ground actions:

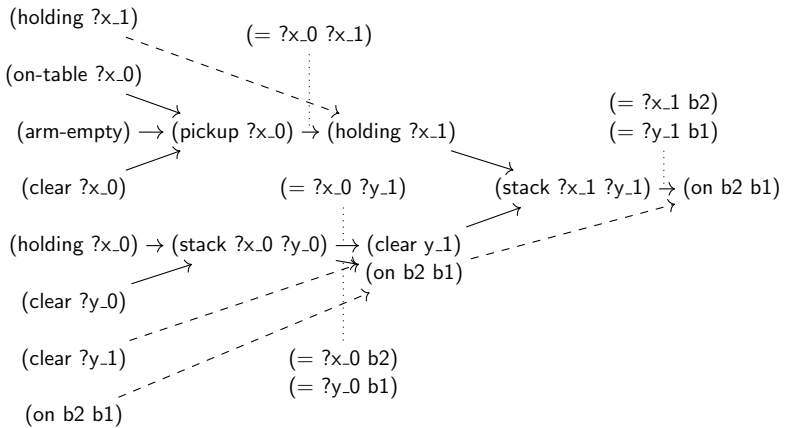


But **building the graph backwards** easily allows to get atoms with **free variables**:



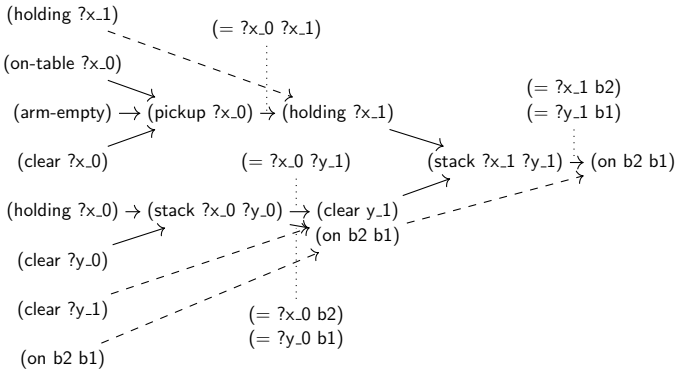
Backward Lifted Planning Graphs

This allows for **much compact planning graphs**

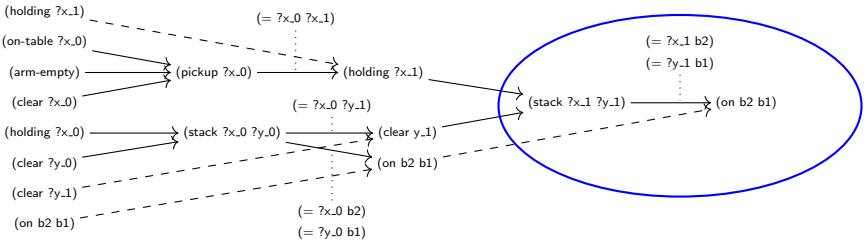


Translation into SMT: functions and variables

```
;; Predicates
(declare-fun clear_0 (Int) Bool)
(declare-fun clear_1 (Int) Bool)
(declare-fun clear_2 (Int) Bool)
(declare-fun on-table_0 (Int) Bool)
(declare-fun on-table_1 (Int) Bool)
(declare-fun on-table_2 (Int) Bool)
(declare-fun arm-empty_0 () Bool)
(declare-fun arm-empty_1 () Bool)
(declare-fun arm-empty_2 () Bool)
(declare-fun holding_0 (Int) Bool)
(declare-fun holding_1 (Int) Bool)
(declare-fun holding_2 (Int) Bool)
(declare-fun on_0 (Int Int) Bool)
(declare-fun on_1 (Int Int) Bool)
(declare-fun on_2 (Int Int) Bool)

;; Actions variables
(declare-const _pickup_?x_0 Bool)
(declare-const _stack_?x_?y_0 Bool)
(declare-const _stack_?x_?y_1 Bool)

;; Objects' variables
(declare-const ?x_1 Int)
(declare-const ?x_0 Int)
(declare-const ?y_1 Int)
(declare-const ?y_0 Int)

;; Goal
(assert (and (on_2 2 1)))
```

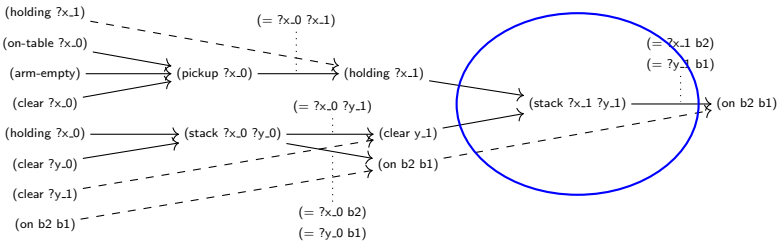


Constraints



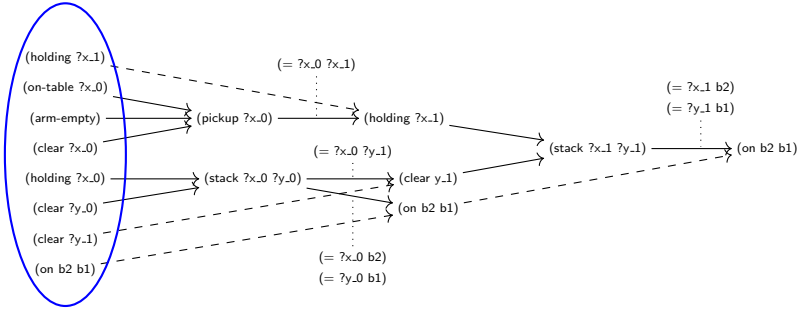

```
;; Planning graph
;; Level 1

;; Frame axioms
(assert (or (not (on_2 2 1)) (on_1 2 1)
            (and _stack_?x_?y_1 (= ?x_1 2) (= ?y_1 1))))

;; At most one action per level

;; At least one action per level
(assert (or _stack_?x_?y_1))
```

Constraints




```
;; Preconditions
(assert (=> _stack_?x_?y_1
            (and (holding_1 ?x_1) (clear_1 ?y_1))))
;; Add effects
(assert (=> _stack_?x_?y_1
            (and (on_2 ?x_1 ?y_1) arm-empty_2 (clear_2 ?x_1))))
;; Del effects
(assert (=> _stack_?x_?y_1
            (and (not (holding_2 ?x_1)) (not (clear_2 ?y_1)))))
```

Constraints



PDDL

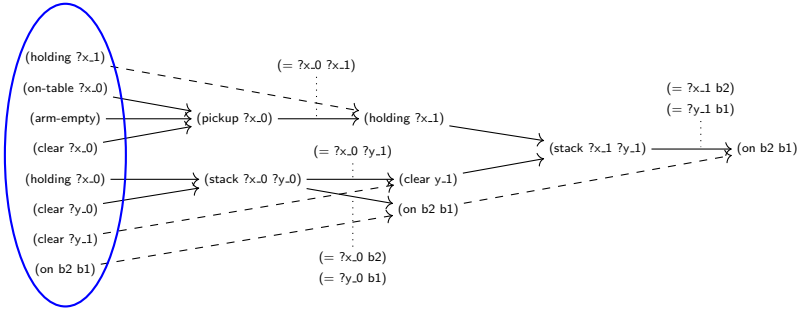
```
(:objects b1 b2 b3)
(:init
   (arm-empty)
   (on-table b1)
   (clear b1)
   (on-table b2)
   (clear b2)
   (on-table b3)
   (clear b3)
)
```

SMT

```
;; Initial facts
(assert (and (arm-empty_0) (on-table_0 1) (clear_0 1) (on-table_0 2)
             (clear_0 2) (on-table_0 3) (clear_0 3)))
;; Closed world
(assert (and (not (holding_0 1)) (not (holding_0 2)) (not (holding_0 3))
             (not (on_0 1 1)) (not (on_0 1 2)) (not (on_0 1 3))
             (not (on_0 2 1)) (not (on_0 2 2)) (not (on_0 2 3))
             (not (on_0 3 1)) (not (on_0 3 2)) (not (on_0 3 3))))
```

Closed world $\Rightarrow$ **combinatorial explosion** (can be 99% of formula)

Constraints



Closed-world combinatorial explosion avoided w/equality and uninterpreted functions!

```
(:objects b1 b2 b3)
(:init
  (arm-empty)
  (on-table b1)
  (clear b1)
  (on-table b2)
  (clear b2)
  (on-table b3)
  (clear b3)
)
```

```
(assert (not (holding_0 ?x_1)))
(assert (or (not (on-table_0 ?x_0)) (= ?x_0 1) (= ?x_0 2) (= ?x_0 3)))
(assert arm-empty_0)
(assert (or (not (clear_0 ?x_0)) (= ?x_0 1) (= ?x_0 2) (= ?x_0 3)))
(assert (not (holding_0 ?x_0)))
(assert (or (not (clear_0 ?y_0)) (= ?y_0 1) (= ?y_0 2) (= ?y_0 3)))
(assert (or (not (clear_0 ?y_1)) (= ?y_1 1) (= ?y_1 2) (= ?y_1 3)))
(assert (not (on_0 2 1)))
```

Results

Table: #solved instances, `github.com/abcorrea/htg-domains` (STRIPS), 16GB, 1h.

	ALPS	LiSAT	Q-Planner	Madagascar	Fast Downward
blocksworld-large-simple (40)	**40**	**40**	1	3	12
childsnack-contents (144)	12	**24**	8	1	12
logistics-large-simple (40)	29	**35**	0	0	21
pipesworld-tankage-nosplit (50)	19	**25**	0	6	11
visitall-multidimensional (180)	99	**111**	10	40	78
Coverage (454)	199	**235**	19	50	134

ALPS, domain-**A**gnostic **L**ifted **P**lanning **S**ystem

Conclusions & Future Work

- We have introduced a simple but powerful SMT formulation for hard-to-ground planning problems
- Prototype not (yet) competitive with state-of-the art, showing very low memory consumption
- **Future work**
  - Add support for negative goals and preconditions
  - Add support for equality predicates
  - **Devise parallel encodings based on these ideas**
  - **Extend to numeric planning**
  - Integrate with SMT solver through API
  - Explore a CP approach

A straightforward MiniZinc encoding

- Use **arrays** for **uninterpreted functions**. E.g.

  ```
  (declare-fun on-table_0 (Int) Bool)
  ```

  becomes

  ```
  array[1..nBlocks] of var bool: onTable_0;
  ```

A straightforward MiniZinc encoding

- Use **arrays** for **uninterpreted functions**. E.g.

  ```
  (declare-fun on-table_0 (Int) Bool)
  ```

  becomes

  ```
  array[1..nBlocks] of var bool: onTable_0;
  ```

- Use **built-in equality** for variable binding constraints

A straightforward MiniZinc encoding

- Use **arrays** for **uninterpreted functions**. E.g.

  ```
  (declare-fun on-table_0 (Int) Bool)
  ```

  becomes

  ```
  array[1..nBlocks] of var bool: onTable_0;
  ```

- Use **built-in equality** for variable binding constraints
- Use **set** constraints for **unary** predicates in initial state. E.g.

  ```
  (assert (or (not (on-table_0 ?x_0)) (= ?x_0 1) (= ?x_0 2) (= ?x_0 3)))
  ```

  becomes

  ```
  constraint onTable_0[x_0] -> x_0 in {1,2,3};
  ```

A straightforward MiniZinc encoding

- Use **arrays** for **uninterpreted functions**. E.g.

```
(declare-fun on-table_0 (Int) Bool)
```

becomes

```
array[1..nBlocks] of var bool: onTable_0;
```

- Use **built-in equality** for variable binding constraints
- Use **set** constraints for **unary** predicates in initial state. E.g.

```
(assert (or (not (on-table_0 ?x_0)) (= ?x_0 1) (= ?x_0 2) (= ?x_0 3)))
```

becomes

```
constraint onTable_0[x_0] -> x_0 in {1,2,3};
```

- Use **table** constraints for **non-unary** predicates in initial state. E.g.

```
(assert (or (not (on_0 ?x_0 ?y_0))
(and (= ?x_0 1) (= ?y_0 2)) (and (= ?x_0 2) (= ?y_0 3))))
```

becomes

```
constraint on_0[x_0,y_0] -> table([x_0,y_0], [|1,2|2,3|]);
```

Conclusions & Future Work

Thank you!