Exploring Instance Generation for Automated Planning

Ozgür Akgün, Nguyen Dang, Joan Espasa,
Ian Miguel, András Z. Salamon and Christopher Stone

School of Computer Science, University of St Andrews, UK
{ozgur.akgun,nttd,jea20,ijm,Andras.Salamon,cls29}@st-andrews.ac.uk

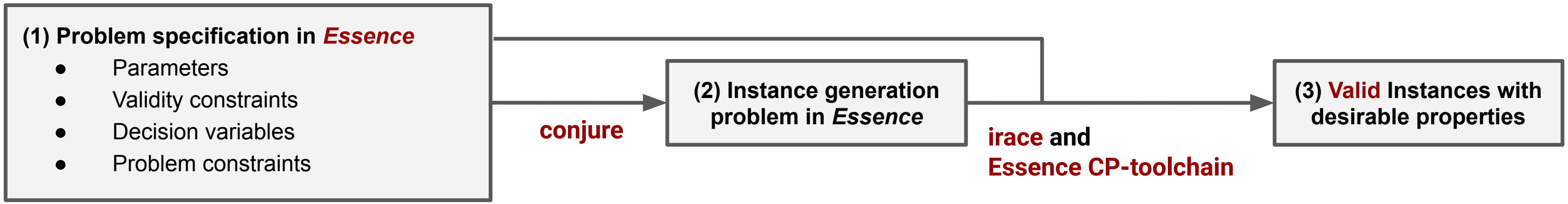
Why Instance Generation?

- Varied benchmarks are **crucial** to evaluate solvers
- Using same benchmarks over and over: **overfitting**
- Better understanding of essential differences between solvers

What we build upon

An automated instance generation system for CP *(Akgün et al 2019, 2020)*

- ***Essence CP-toolchain*** [1] : a high-level constraint modelling pipeline
- ***irace*** [2] *(López-Ibáñez et al 2016)*: an automated parameter tuning tool



Our aim: extend the system to support automated instance generation for AI Planning Problems

[1] https://constraintmodelling.org/ [2] https://iridia.ulb.ac.be/irace/

What is AI Planning?

A ***classical* planning problem** is a tuple $\Pi = \langle V,A,I,G \rangle$:

- **V**: *propositions* (or Boolean variables)
- **A**: *actions*
  - formalized as pairs of <*pre-conditions*, *effects*>
- **I**: *initial state*
- **G**: a formula over **V** that any *goal state* must satisfy.

What is AI Planning?

A ***classical* planning problem** is a tuple $\Pi = \langle V,A,I,G \rangle$:

- **V**: *propositions* (or Boolean variables)
- **A**: *actions*
  - formalized as pairs of <*pre-conditions*, *effects*>
- **I**: *initial state*
- **G**: a formula over **V** that any *goal state* must satisfy.

PDDL: a modelling language for classical planning problems

PDDL Examples

```
(:types robot tile color - object)
(:predicates
      (robot-at ?r - robot ?x - tile)
      (up ?x - tile ?y - tile)
      (down ?x - tile ?y - tile)
      (right ?x - tile ?y - tile)
      (left ?x - tile ?y - tile)
      (clear ?x - tile)
      (painted ?x - tile ?c - color)
      (robot-has ?r - robot ?c - color)
      (available-color ?c - color))
```

```
(:action move_up
        :parameters
          (?r - robot ?from - tile ?to - tile)
        :precondition (and
             (robot-at ?r ?from)
             (up ?to ?from) (clear ?to))
        :effect (and
             (robot-at ?r ?to)
             (not (robot-at ?r ?from))
             (clear ?from) (not (clear ?to))))
```

...

PDDL instance

```
(define (problem toy)
 (:domain floor-tile)
 (:objects
    tile_0-0 tile_0-1
    tile_1-0 tile_1-1 - tile
    robot1 robot2 - robot
    white black - color)
 (:goal (and
     (painted tile_0-0 white)
     (painted tile_1-0 black))))
```

```
(:init
    (robot-at robot1 tile_0-1) (robot-has robot1 white)
    (robot-at robot2 tile_1-1) (robot-has robot2 black)
    (available-color white) (available-color black)
    (clear tile_0-0) (clear tile_1-0)
    (up tile_0-1 tile_1-1) (up tile_0-0 tile_1-0)
    (down tile_1-1 tile_0-1) (down tile_1-0 tile_0-0)
    (right tile_0-1 tile_0-0) (right tile_1-1 tile_1-0)
    (left tile_0-0 tile_0-1) (left tile_1-0 tile_1-1))
```

PDDL instance


```
(define (problem toy)
 (:domain floor-tile)
 (:objects
    tile_0-0 tile_0-1
    tile_1-0 tile_1-1 - tile
    robot1 robot2 - robot
    white black - color)
 (:goal (and
     (painted tile_0-0 white)
     (painted tile_1-0 black))))
```

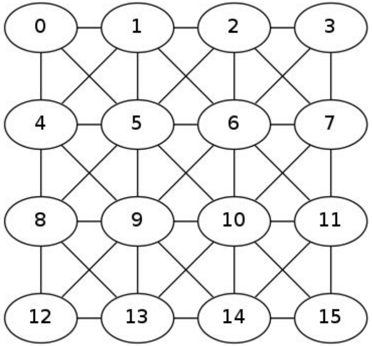
```
(:init
   (robot-at robot1 tile_0-1) (robot-has robot1 white)
   (robot-at robot2 tile_1-1) (robot-has robot2 black)
   (available-color white) (available-color black)
   (clear tile_0-0) (clear tile_1-0)
   (up tile_0-1 tile_1-1) (up tile_0-0 tile_1-0)
   (down tile_1-1 tile_0-1) (down tile_1-0 tile_0-0)
   (right tile_0-1 tile_0-0) (right tile_1-1 tile_1-0)
   (left tile_0-0 tile_0-1) (left tile_1-0 tile_1-1))
```

Validity Constraints

- ***Essence*** uses ***where** constraints* to restrict the input space:

```
given b: int(1..)
given r: int(1..)
where r <= b
```

- ***PDDL*** can't express them in most cases
- Useful to guide the search for graded instances
- **Pivotal**, depending on the assumptions:



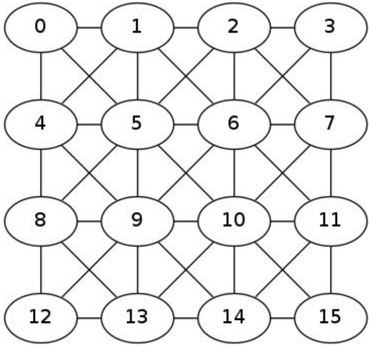
Validity Constraints

- ***Essence*** uses ***where** constraints* to restrict the input space:

```
given b: int(1..)
given r: int(1..)
where r <= b
```

- ***PDDL* can't express them in most cases**
- Useful to guide the search for graded instances
- Pivotal, depending on the assumptions:

Augment PDDL to support those constraints?



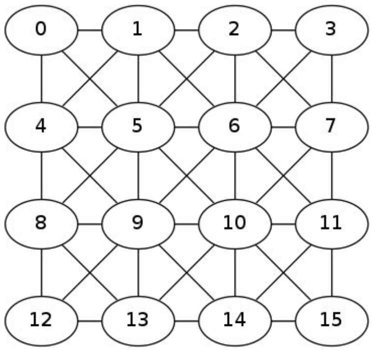
Validity Constraints

- ***Essence*** uses ***where** constraints* to restrict the input space:

```
given b: int(1..)
given r: int(1..)
where r <= b
```

- ***PDDL* can't express them in most cases**
- Useful to guide the search for graded instances
- Pivotal, depending on the assumptions:

Augment PDDL to support those constraints?

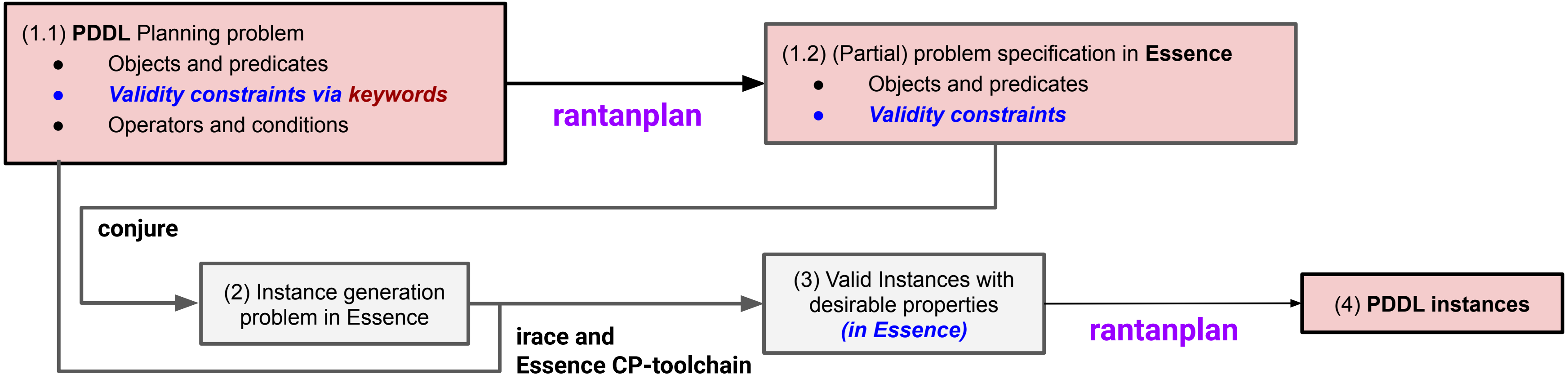


PDDL + **Essence**

Planning problem description

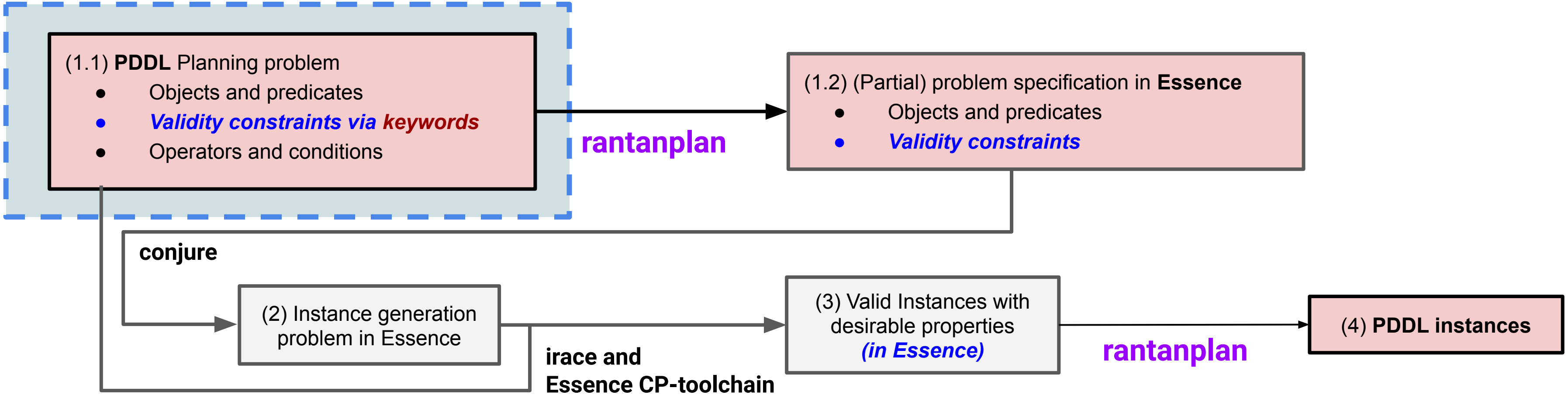
Validity constraints

First approach: Adding new keywords into PDDL



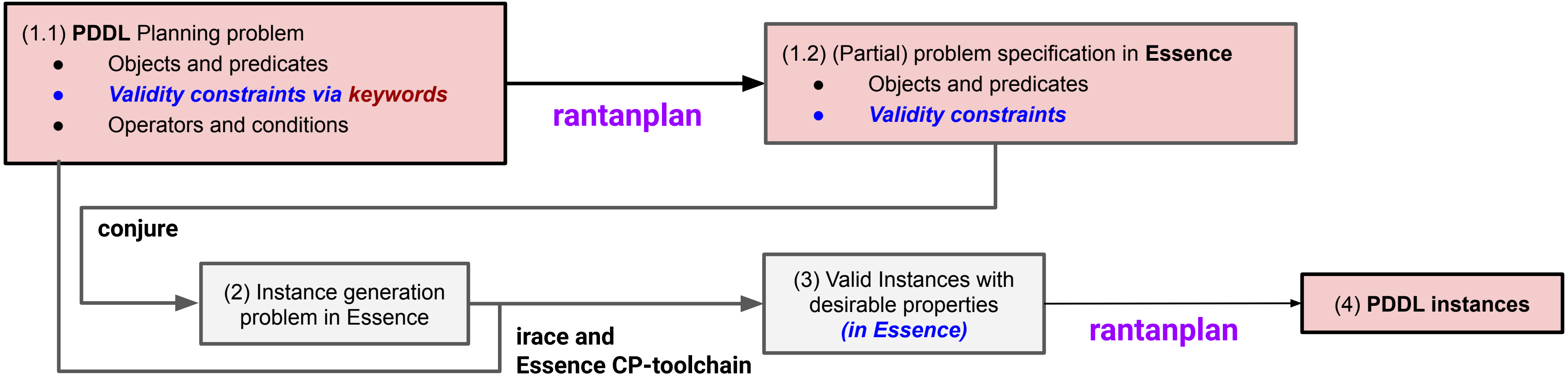
- Keyword examples: *at-least-k, at-most-k, min, max, xor, square-grid*
- Keywords are translated to *validity constraints in Essence* by *rantanplan*

First approach: Adding new keywords into PDDL



One single PDDL input by users

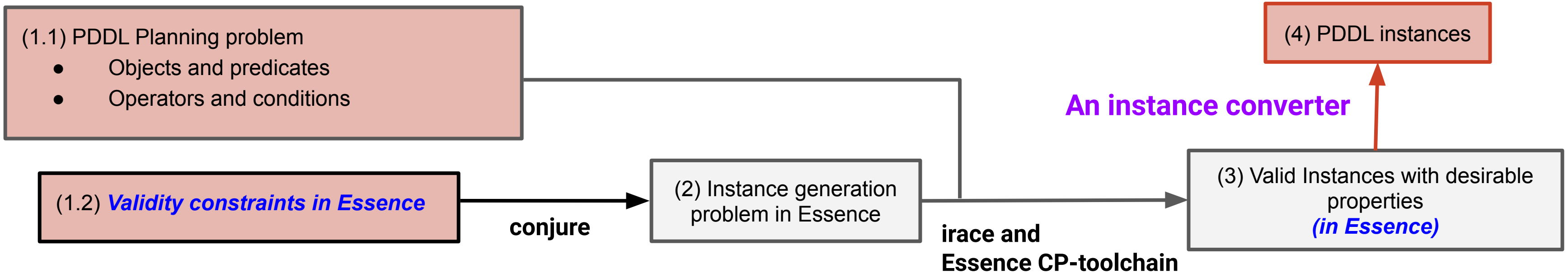
First approach: Adding new keywords into PDDL



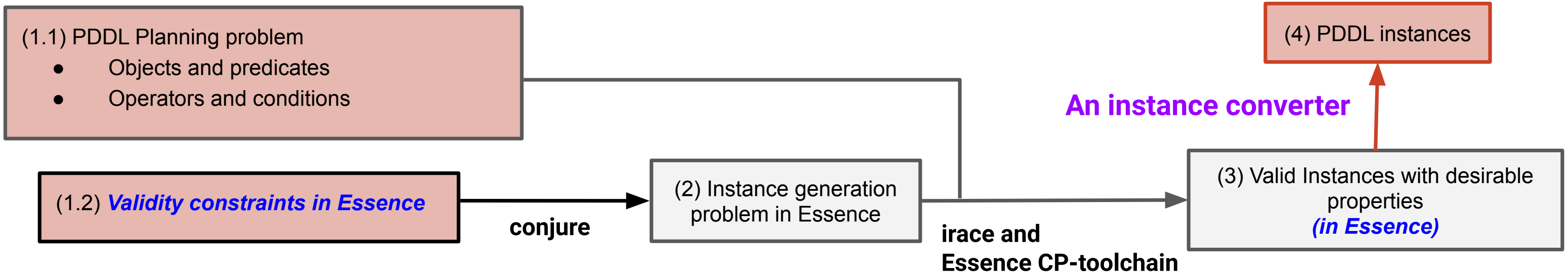
One single PDDL input by users

- **Limited flexibility** in specifying validity constraints
  square-grid: up-down-left-right, northwest-southeast-etc.
  other shapes rather than square-grid?
- **Bad scalability** due to low-level representations

Second approach:
Expressing validity constraint directly using Essence

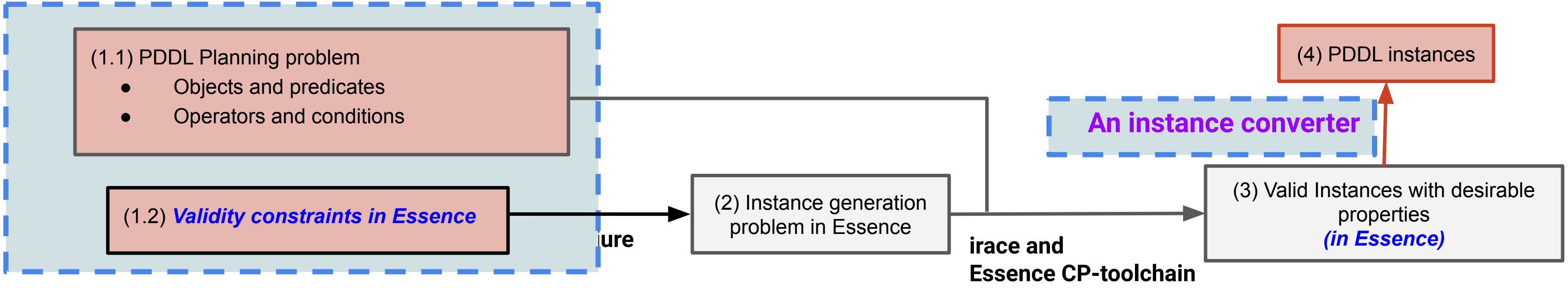


Second approach:
Expressing validity constraint directly using Essence



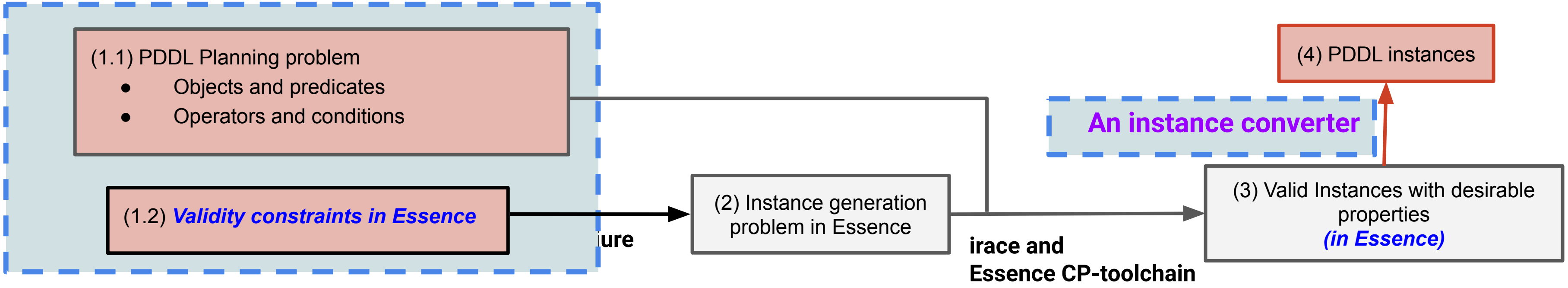
- **Flexibility** in specifying validity constraints
- **Much better scalability**
  thanks to high-level representations

Second approach:
Expressing validity constraint directly using Essence

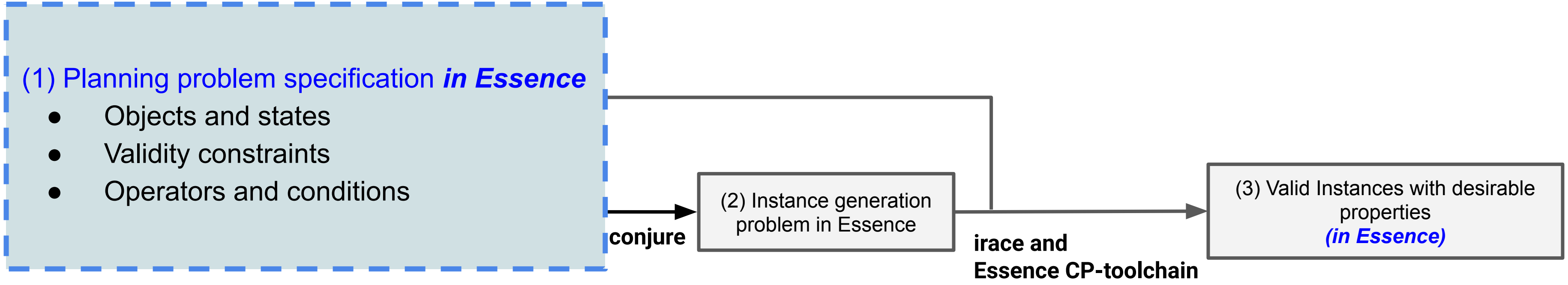


- **Flexibility** in specifying validity constraints
- **Much better scalability**
  thanks to high-level representations

Lack of automation
3 user input components required

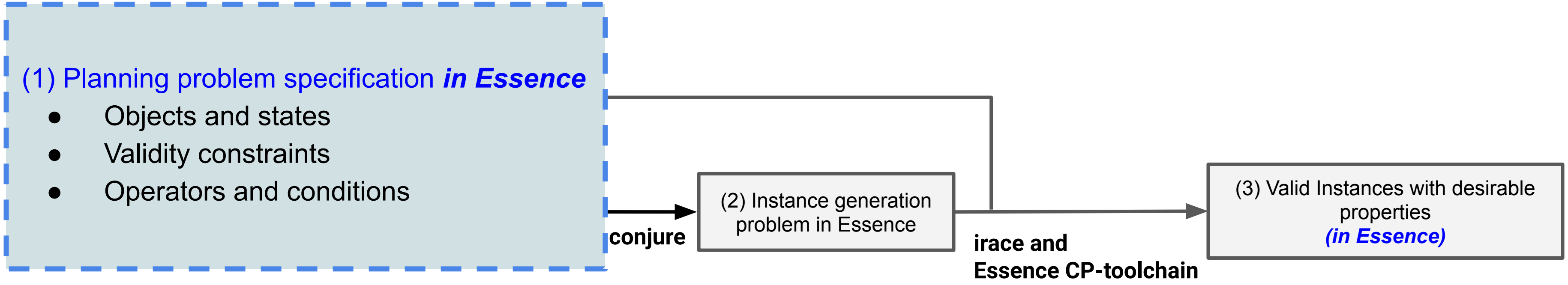


How about using Essence for the whole thing?

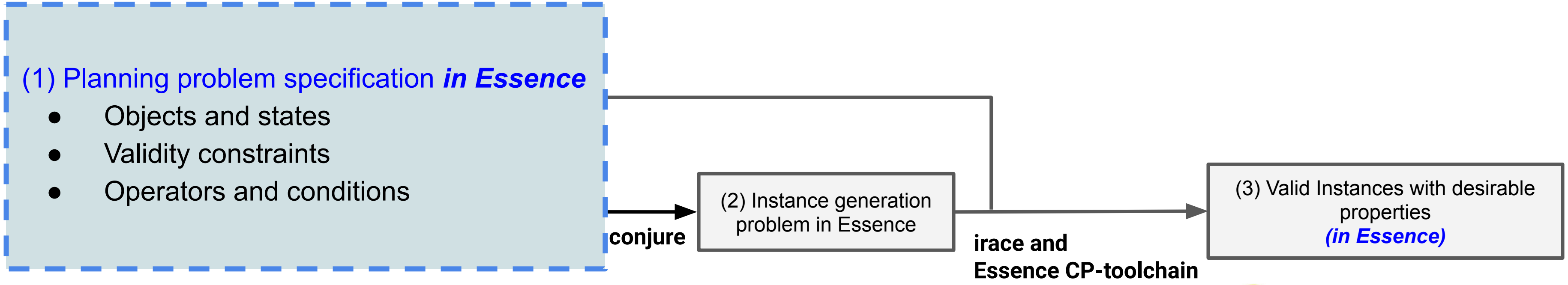


How about using Essence for the whole thing?

Third approach: Expressing planning problems in Essence



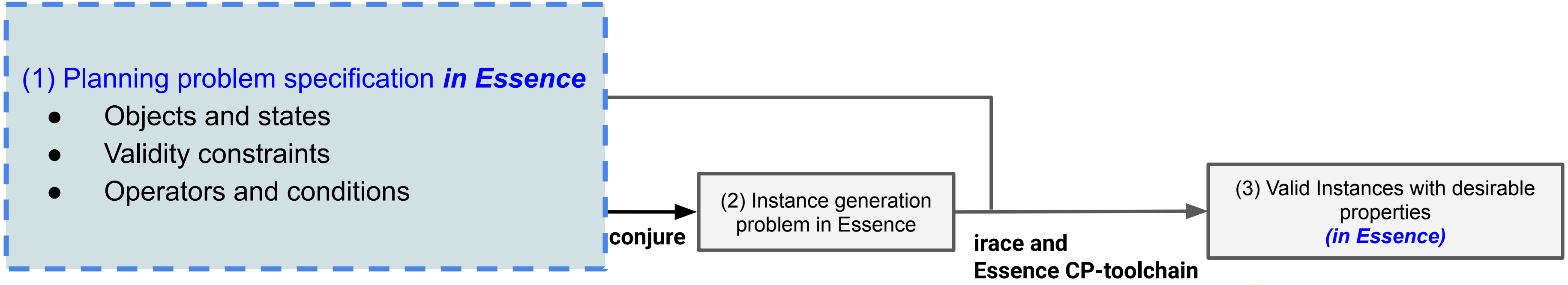
Third approach: Expressing planning problems in Essence



- **Flexibility** & **good scalability**
  - high-level data types allows capture abstract structures easily
    - `sequence, set, relation, function, partition, record, ...`

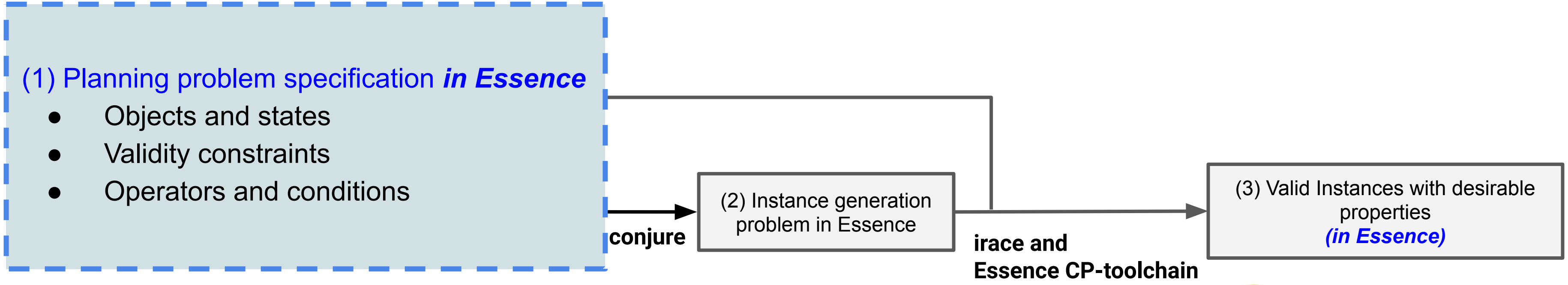
```
letting STATE be domain record {
  robots : sequence (size n_robot) of
                record{row:int, column:int, colour:COLOUR},
  grid : GRID}
```

Third approach: Expressing planning problems in Essence



- **Flexibility** & **good scalability**
  - high-level data types allows capture abstract structures easily
    - `sequence, set, relation, function, partition, record, ...`
- **A single input** by users

Third approach: Expressing planning problems in Essence



- **Flexibility** & **good scalability**
  - high-level data types allows capture abstract structures easily
    - `sequence, set, relation, function, partition, record, ...`
- **A single input** by users
- **Multiple solving paradigms supported**:
  - could be refined to **PDDL**, **CP**, **SAT**, **SMT**, ...

Takeaway

- PDDL has limited expressivity and is not well-suited for automated instance generation for planning
- We believe that expressing planning problem using high-level modelling language such as Essence is the key solution
- Next step: we need to implement the described extension in Essence
  → it’s a lot of work for the implementation, so we want to know if the community would like it :)

Please let us know what you think!

What is AI Planning?

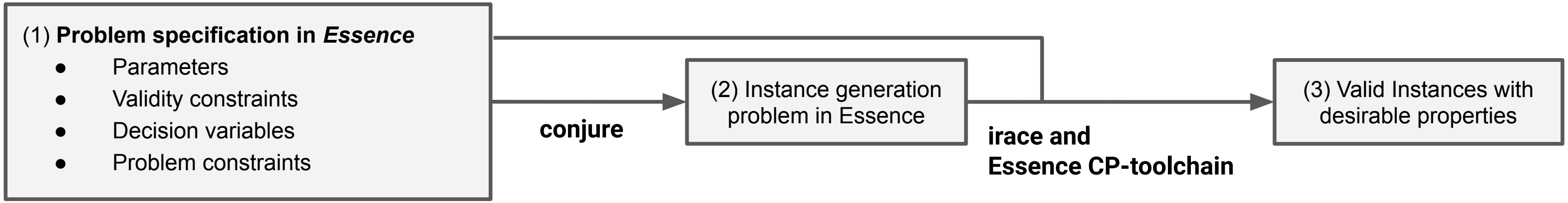
A classical planning problem is defined as a tuple **Π =⟨V,A,I,G⟩**:

- **V** - a set of propositions (or Boolean variables)
- **A** - is a set of actions, formalized as pairs⟨**p**,**e**⟩, where **p** is a set of preconditions and **e** a set of effects
- **I** - is the initial state
- **G** - is a formula over V that any goal state must satisfy.

What we build upon

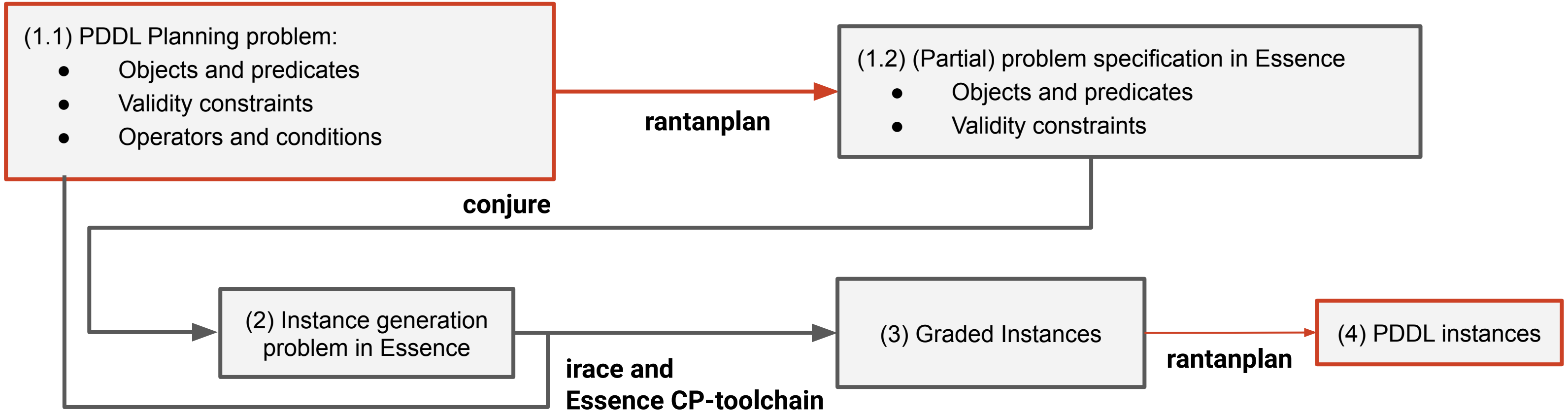
Akgün, Dang, Miguel, Salamon, Stone, *Instance generation via generator instances* (CP 2019)

- Uses the ***Essence CP-toolchain*** and ***irace*** to generate instances
- We treat it as a black box



First approach: Extend PDDL

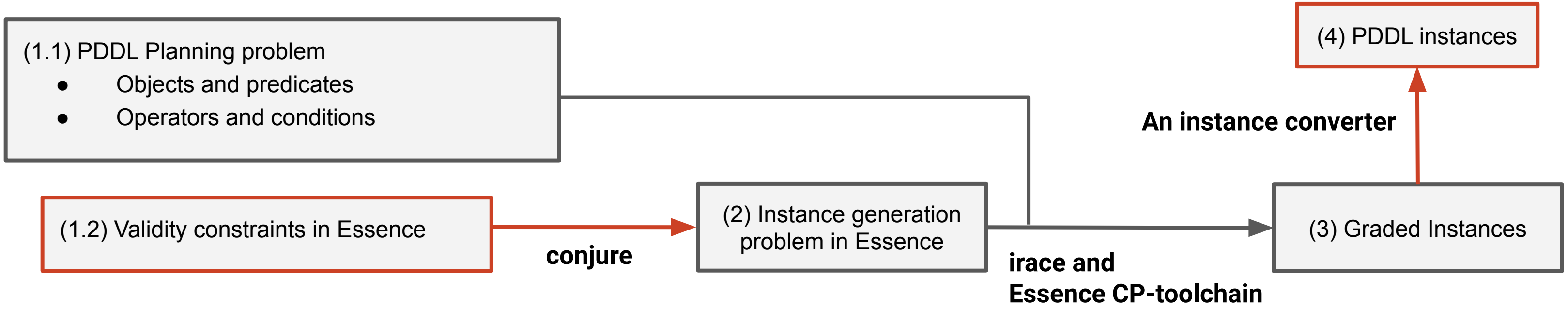
New keywords: **`instance-constraints`**, **`init`**, **`goal`**, **`appear`**, **`min`**, **`max`**, **`exactly-k`**, **`atleast-k`**, **`atmost-k`**, **`xor`** **+** A library of structures: **`isLRUDquareGrid`**



Problem: many structural constraints (such as a graph being connected) cannot be expressed in a purely first-order language like PDDL

Second approach: Use Essence

Using Essence directly would be a solution, giving the user more expressivity.



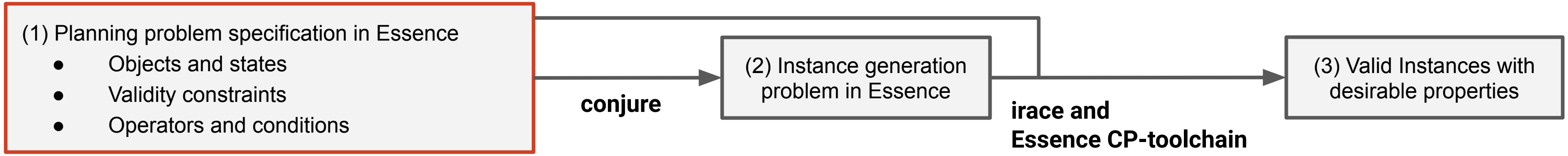
The good: Higher level constructs means better performance

The bad: lack of automation. No easy way of deriving the semantics between the two representations.

Third approach: Extend Essence

- high-level type constructors, such as `set`, `relation` and `function`
- No need to reconstruct the structure from a PDDL description
- Could refine down to PDDL, CP, SAT, SMT, ...

```
Letting STATE be domain record {
     robots : sequence (size n_robot) of record
                         {    row :int,
                              column:int,
                              colour: COLOUR },
     grid : GRID}
```



Takeaways:

- Working system for simple PDDL problems
- PDDL has limited expressivity for what we need
- Proposal of an elegant solution