Exploring Instance Generation for Automated Planning

Özgür 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

Abstract. Many of the core disciplines of artificial intelligence have sets of standard benchmark problems well known and widely used by the community when developing new algorithms. Constraint programming and automated planning are examples of these areas, where the behaviour of a new algorithm is measured by how it performs on these instances. Typically the efficiency of each solving method varies not only between problems, but also between instances of the same problem. Therefore, having a diverse set of instances is crucial to be able to effectively evaluate a new solving method. Current methods for automatic generation of instances for Constraint Programming problems start with a declarative model and search for instances with some desired attributes, such as hardness or size. We first explore the difficulties of adapting this approach to generate instances starting from problem specifications written in PDDL, the de-facto standard language of the automated planning community. We then propose a new approach where the whole planning problem description is modelled using ESSENCE, an abstract modelling language that allows expressing high-level structures without committing to a particular low level representation in PDDL.

1 Introduction

The planning task consists of selecting a sequence of actions in order to achieve a specified goal from specified starting conditions. This type of problem arises in many contexts. Consider, for example, the delivery of a set of packages by vehicle from a depot to a set of destinations. The allocation of packages and drivers to vehicles must be planned, as well as the route for each vehicle, while respecting package delivery deadlines, vehicle capacities and driver shift restrictions.

Given their importance, the automated solution of planning problems is a central discipline of Artificial Intelligence. The difficulty of solving planning problems grows rapidly with their size in terms of the number of objects and possible actions under consideration. Over many years, a great deal of effort by different research groups has resulted in the development of highly efficient AI planning systems [27]. Testing algorithms across a wide range of problem instances is crucial to ensure the validity of any claim about one algorithm being better than another. However, when it comes to evaluations, typically limited sets of problems are used and thus the full picture is rarely seen. Finding and encoding interesting instances is a time-consuming task, and due to the nature of some problems, is sometimes out of reach from the researcher perspective.

In the International Planning Competitions (IPC) state of the art planning systems are empirically evaluated on a set of benchmark problems. The competitions have, amongst others, a track that focuses on planners that can learn from previous runs. This track uses manually coded problem generators to provide the planners with a varied set of problems. This variety of problems is crucial to ensure that planners can learn and generalise well on new and unseen situations.

The problem of automated instance generation is to make the process of creating benchmark problems more accessible and efficient. Instance generation has been a focus of the SAT community for several decades [22]. Generating random SAT instances while considering different parameters such as the length of clauses, the number of variables, or their connectivity, has helped to illuminate how algorithms behave and how they perform in different circumstances. Having automated tools to generate interesting instances allows algorithm developers to evaluate and compare the algorithms across a wide range of instances, providing a detailed picture of their comparative strengths and weaknesses.

In this work we explore the adaptation of a successful tool [1] that automatically generates instances with desirable properties from a single problem specification in ESSENCE, a high-level modelling language for Constraint Programming (CP) [4]. We discuss two approaches (Section 5 and 6) to make the instance generation process in [1] work with PDDL [18], the de-facto standard language in the automated planning community. Both approaches have their own limitations regarding *flexibility*, *efficiency* and *automation*. We then make a proposal for a new planning modelling language using ESSENCE (Section 7). Thanks to the rich expressiveness of the language, all discussed limitations of the PDDL-based instance generation approaches are overcome.

2 Related Work

Fuentetaja et al. [7] approach the generation of satisfiable planning instances as a planning problem, where users manually write some declarative semantics-related information to describe the generation of different instances. They have pointed out the limitation of PDDL as a representation for describing the instance generation problem, as the task of generating valid initial states is complex and requires information about the meaning of predicates not expressed in the domain definition. PDDL representation is only effective for representing valid states and the transition functions between them. Augmenting PDDL to improve its expressiveness has been proposed several times [14,5,9]. This allows adding extra information into the domain definition, and could potentially lead to more general applications of planning [23].

The usefulness of automated generation of benchmark instances has been demonstrated in various fields, such as combinatorial optimisation [25], SAT [12] and model fitting [20] especially in the context of instance space analysis [24]. In

CP, benchmark instances created using problem-specific generators have been proposed for various problem classes, such as binary Constraint Satisfaction Problem [11,19] and Random Constraint Networks [29]. In Operations Research, several instance generators and benchmarks for well-known combinatorial optimisation problems, such as the Knapsack problem [13] and the Nurse Rostering problem [28], have been provided and widely used. All of those approaches are semi-automated, as the instance generators were manually created.

In discrete optimisation Ullrich [26] developed a tool for the generation of instances in the TSP, Max-SAT and Load Allocation problems. Even in this case generating instances for new problem classes requires some user input, however the formal language they developed simplifies the process of producing instances for new domains.

Recently, a new approach for fully automated instance generation has been proposed for CP [1,2]. The approach allows users to declaratively describe a CP problem and properties of valid instances in a single CP model using the high-level constraint modelling language ESSENCE [4]. An instance generator is then created by the automated modelling tool CONJURE [3]. Finally, instances with desirable properties are generated through a combination of the automated algorithm configuration tool `irace` [17] and the ESSENCE constraint modelling toolchain developed by the CP group at St Andrews [1], which includes CONJURE, Savile Row [21] (a constraint modelling assistant) and `minion` [8] (a CP solver). All steps are done in a completely automated fashion. The whole process of the system is shown in Figure 1.

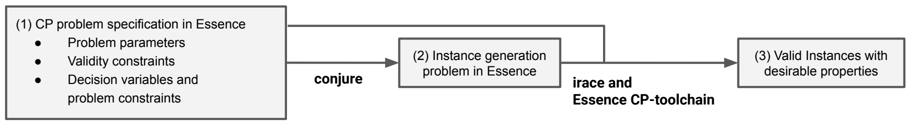
(1) CP problem specification in Essence
Problem parameters
Validity constraints
Decision variables and problem constraints
conjure
(2) Instance generation problem in Essence
irace and Essence CP-toolchain
(3) Valid Instances with desirable properties

Fig. 1: The automated instance generation approach for CP problems proposed in [1,2]

3 Background

A classical planning problem is defined as a tuple $\Pi = \langle V, A, I, G \rangle$ where V is a set of propositions (or Boolean variables), A is a set of actions, I is the initial state and G is a formula over V that any goal state must satisfy.

A state is a total assignment to the variables. Actions are formalized as pairs $\langle p, e \rangle$, where p is a set of preconditions and e a set of effects. More formally, p is a set of Boolean expressions over V, while e is a set of assignments. An action $a = \langle p, e \rangle$ is executable in a given state s if $s \models p$.The state resulting from

[1] https://constraintmodelling.org/

executing action a on state s is denoted by $apply(a, s) = s'$. The new state s' is defined by assigning new values to the variables according to the active effects, and retaining the values of the variables that are not assigned values by any of the active effects. A plan of length n for a planning problem Π is a sequence of actions $a_1; a_2; \ldots; a_n$ such that $apply(a_n, \ldots, apply(a_2, apply(a_1, I)) \ldots) \models G$.

An example planning problem is the `floor-tile` problem used in the International Planning Competition 2014 (IPC-14) [27]. The problem includes a set of robots sharing the task of painting a pattern on floor tiles. The robots move around in four directions (up, down, left and right) and can paint with one colour at a time. They are also able to change the current colour to any other one. However, due to their design robots can only paint the tiles that are in front or behind them. Finally, once a tile has been painted, no robot can stand on it.

Automated planning models are typically expressed in the Planning Domain Definition Language (PDDL) [18]. The user describes the problem in a *domain* file, in terms of predicates and actions with parameters. In turn, these parameters (or free variables) can be instantiated with a set of defined objects. A concrete instance is expressed in a *problem* file, where the initial state, problem objects and goal are defined. Figure 2 shows how predicates and actions are declared in a domain file for the `floor-tile` problem (a problem description model), and Figure 3 depicts an example problem file (an instance).

```
(:predicates ; state variables are defined in the predicates section
        (robot-at ?r - robot ?x - tile) ; at what tile a robot is
        (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))))
...
```

Fig. 2: Snippets of the `floor-tile` PDDL problem description.

4 Validity Constraints for Planning Instances

A planning model is typically an abstraction of a real-world problem, where the modeller has some predefined assumptions on how the real-world works. The correctness of this abstraction derives from a valid initial state, and that the

```
(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)
 (: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)
)
 (:goal (and (painted tile_0-0 white) (painted tile_1-0 black))))
```

Fig. 3: An example `floor-tile` instance.

transitions of the state variables between steps respect the implicit constraints. When a model and an instance respect all the implicit assumptions by the modeller we say that it is a *valid* problem. As an example, some implicit assumptions by the modeller in the `floor-tile` domain (Figure 2) are that in the initial state each robot must be at exactly one tile, or that any given cell can only have one unique cell on top of it. Moreover, in the IPC-14 published instances, the tiles' structure (represented by `up`, `down`, `right` and `left`) always forms a square grid.

In the automated planning community, sometimes when a problem is released, a Python or Java program is included to generate instances automatically. The validity properties of the problem are normally hard-coded in those programs, and therefore appear implicitly in the generated problem instances.

An alternative approach is to allow modellers to express the validity properties of a planning problem declaratively. An instance generator with those validity constraints integrated is then automatically created. Specifying those properties using a declarative modelling language provides flexibility, as users can easily add or update the validity specification without having to modify the generator software's source code directly. This is the approach of the ESSENCE-based automated instance generation system that we propose to build upon.

There is a large variety of validity properties arising in classical planning domains. In this section, we discuss six arbitrarily selected IPC-14 problems (in the Sequential track) and the validity properties of their published benchmark instances. A brief description of these problems follows. We refer to the competition website[2] and the accompanying paper [27] for further details of these benchmarks.

– `city-car`: This problem simulates the impact of road building and demolition on traffic flows. A city is represented as a grid, in which each node is a

[2] https://helios.hud.ac.uk/scommv/IPC-14/

junction and edges are potential roads. Cars start from different positions and have to reach their final destinations. A finite number of roads can be built to connect two junctions and allowing a car to move between them.

- `floor-tile`: A set of robots use colours to paint patterns in floor tiles. The robots can move around and paint with one colour at a time and can also change their colours. Once a tile has been painted, no robot can stand on it.
- `hiking`: This problem simulates a walking trip that lasts several days, where each day one walk is done with a partner. The walks are over a long circular route, without ever walking backwards. Tents and items of luggage can be carried in a car between the start and end of the walking routes if necessary.
- `cave-diving`: There are divers with different skills and confidence, and each can carry tanks of air. These divers must be hired to go into an underwater cave system and either take photos or prepare the way for other divers by dropping full tanks of air. The cave is too narrow for more than one diver to enter at a time. Divers must exit the cave and decompress at the end. They can therefore only make a single trip into the cave. Divers have hiring costs inversely proportional to how hard they are to work with.
- `child-snack`: This involves making and serving sandwiches with various ingredients for a group of children in which some are allergic to gluten.
- `barman`: A robot barman manipulates drink dispensers, glasses, and a shaker. The goal is to find a plan of the robot's actions that serves a desired set of drinks. Robot hands can only grasp one object at a time and glasses need to be empty and clean to be filled.

The properties in these benchmark instances can be roughly divided into two groups. The first group involves constraints between the variables emerging from a single predicate. They typically include three types of constraints: *exactly-k*, *at-most-k* and *at-least-k*, where k is a parameter of the constraints. For example, from the `floor-tile` model an instances illustrated in Figure 2 and 3, we can infer that each robot begins at exactly one tile. This could be represented by, for a given robot, an *exactly-1* constraint on the variables that result from grounding the `robot-at` predicate.

The second group of validity constraints involves *structural constraints* between predicates in a planning problem. The published instances of both `city-car` and `floor-tile` have *square-grid underlying maps*. The maps of `floor-tile` require rectangular grids with up, down, left and right connections, while the maps in `city-car` instances allow cells to be connected horizontally as well as diagonally. In `cave-diving` a cave forms a tree-shaped structure. Sequences are represented in the `hiking` and `barman` domains, while other problems from the competition use stacks or weighted undirected graphs.

If we want to generate valid instances randomly, it is necessary to take these validity constraints into account during the modelling stage. This is because it would be vanishingly unlikely that an instance randomly sampled from the unconstrained instance space would be valid. For example, consider the *floor-tile* domain from Figure 2 and focus on the predicates `up` and `down`, which state if a tile is on top (or bottom) of another. If the `up` predicate states that a tile is on top of another, the `down` predicate should be coherent and correspond with the inverse assignment. If we generate instances randomly, only a small fraction would satisfy this property for any given pair of tiles, and this fraction shrinks exponentially as the instance size grows. Another example involves the `robot-at` predicate, which specifies the tile where a robot is located. It would be unlikely for randomly generated instances to respect the fact that a robot can only be in one place at one time. Almost surely we would end up with instances where a robot is located at various places simultaneously. Combining all the implied constraints used by a modeller, it is clear that the chance of randomly generating valid instances is negligibly small. It is possible to make this claim mathematically precise but we leave this for future work.

5 Augmented PDDL for Generation of Planning Instances

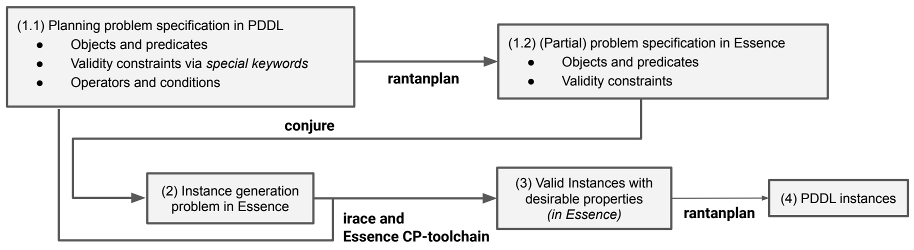
(1.1) Planning problem specification in PDDL
Objects and predicates
Validity constraints via *special keywords*
Operators and conditions
rantanplan
(1.2) (Partial) problem specification in Essence
Objects and predicates
Validity constraints
conjure
(2) Instance generation problem in Essence
irace and Essence CP-toolchain
(3) Valid Instances with desirable properties (in Essence)
rantanplan
(4) PDDL instances

Fig. 4: Automated instance generation for planning using Augmented PDDL

The ESSENCE language allows expressing validity constraints for a Constraint Programming (CP) problem as `where` statements. As shown in Figure 1, the system we build upon starts by receiving the problem description in ESSENCE with validity constraints as input (step 1), and CONJURE will automatically create an instance generator in ESSENCE (step 2), which will then be tuned by `irace` in combination with the CP solver `minion` to generate instances with desirable properties (step 3). In order to employ the same automated methodology for planning problems, we need to be able to create an instance generator automatically from a planning problem description. Ideally we would want to use the planning modelling language PDDL to express those validity constraints inside the problem description in step 1 of Figure 1.

Unfortunately, due to the low-level nature of standard PDDL, the task becomes extremely tedious and error-prone for a human. Firstly, as classical planning only deals with Boolean variables, many-valued variables cannot be directly expressed. This type of variables is essential for modelling common validity constraints such as that a robot can only start at exactly one place. Secondly, many structural constraints (such as a graph being connected) cannot be expressed in a purely first-order language like PDDL [16, Corollary 3.19]. Verifying that a grid specified by adjacency predicates is well-formed, as in our example model, appears also to be impossible to express in pure PDDL, with even restricted versions of this problem still under active investigation [15].

In this section, we will therefore discuss a solution approach (Figure 4) where we augment PDDL with a new declarative section and extra keywords to allow the modeller to express validity constraints in a PDDL problem description (step 1.1 in Figure 4). The real modelling of those constraints is then generated automatically by rantanplan [6], a parser specially developed to translate an augmented PDDL model to the high-level language ESSENCE (step 1.2 in Figure 4). ESSENCE is more expressive than PDDL and allows modelling of structural validity constraints directly.

5.1 Augmented PDDL for single-predicate validity constraints

```
(:instance-constraints
(init (forall (?r - robot)
  (and  (exactly-k (robot-at ?r _) 1 True) ; a robot starts in a tile
        (exactly-k (robot-has ?r _) 1 True)))) ; and has one colour

;nothing starts painted
(init (forall (?t - tile) (exactly-k (painted ?t _) 0 True)))

; we are not interested in the clear predicates in the goal state
(goal (forall (?t - tile) (not (appear (clear ?t)))))
```

Fig. 5: Snippets of the `instance-constraints` section from the `floor-tile` domain.

Validity constraints on the initial and goal states are defined in a new section starting with the new keyword `instance-constraints`. The following new operands are added for single-predicate validity constraints: `init`, `goal`, `xor`, `min`, `max`, `exactly-k`, `atleast-k`, `atmost-k` and `appear`. By default, when considering numeric functions, the range of values generated goes from 0 to `INT_MAX` (a default upper bound for integer variables, which can be specified by users). The modal operators `min` and `max` accept the name of a function and an integer. This further restricts the range of possible values generated. Figure 5 shows an example of the `instance-constraints` section for `floor-tile` using those new keywords.

`init` and `goal` are modal operators that accept a constraint. This constraint will be then applied to the initial or goal state, respectively. `xor` implements the xor logical operation, and has been added for convenience. `exactly-k`, `atleast-k` and `atmost-k` are a family of terms that accept a schematic fluent, a number and a value. As their name imply, they restrict the values taken by the subset of grounded variables generated by that schematic fluent. When specifying these terms, typically we will be interested in one or two parameters of the constrained schematic fluent. For conciseness, the underscore character (_) can be used to indentify parameters that we do not care, acting as a placeholder in a pattern matching style. Finally, the `appear` predicate is used to describe the goal state. The initial state is always a total assignment, as per the closed world assumption, but the goal can be a partial assignment. `appear` can be combined with the `not` operator to force a state variable to not appear in the goal state. It is useful to avoid considering non-interesting goals to the modeller.

Not all the new operands are translated directly to ESSENCE. `min` and `max` determine the size of the integer domains of the related PDDL fluents, while `init` and `goal` control what is constrained. The cardinality constraints follow a pattern `k` {<,>,=} `sum([toInt(x = value) | fluent ])`, where `value` is what we are searching for, and the `fluent` placeholder iterates over the tuples belonging to the fluent, which is represented as a function. Following the `floor-tile` domain, Figure 6 shows how a constraint expressing that a given robot can only be at one place in the initial state is translated to ESSENCE.

```
; a robot starts in one tile
(:instance-constraints
(init (forall (?r - robot)
  (exactly-k
    (robot-at ?r _) 1 True))))
```

(a) Constraint expressed in PDDL

```
; a robot starts in one tile
forAll var_r : robot .
  1 = sum([toInt(value = true)
    | ((p0,_),value) <-
      init[robot_at],var_r = p0 ]
```

(b) Constraint translated to ESSENCE

Fig. 6: The translation of `exactly-k` constraint from PDDL to ESSENCE

5.2 Augmented PDDL for structural constraints

As described in Section 4, another group of validity constraints involves implicit requirements on structures of the underlying map in a planning problem, such as the connections between tiles in the `floor-tile` problem. It is possible to express grids quite simply if the relations used to express the grid structure make use of the geometry of the plane. However, automated instance generation should not restrict the choices made in modelling problems. In the IPC-14 benchmark dataset, all instances of `floor-tile` have the tiles forming a square-grid structure and tiles' connections are represented using adjacency relations `left, right, up` and `down`. Validity constraints to ensure that these adjacency relations express a square-grid map cannot be efficiently modelled using PDDL due to the limited expressiveness of first-order logic. Checking that the adjacency relations form a valid grid requires reconstructing geometric information about placement of tiles on the plane. Although it is possible to express the property that the adjacency relations form a grid for special cases, even this requires solving a challenging tiling problem, and the general case is currently open [15].

```
(:instance-constraints
  init( isLRUDSquareGrid(tile , up, down, left , right) )
  ... ))
```

```
$ ----- Objects and Domains --------
given n_tile: int(1..)
letting tile be domain int(1 .. n_tile)
$ ---- Auxiliaries ------
given tile_size: int(1..)
where n_tile = tile_size * tile_size
$ ----- Initial State --------
given init: record {
  up : function (total) (tile,tile) --> bool,
  down : function (total) (tile,tile) --> bool,
  right : function (total) (tile,tile) --> bool,
  left : function (total) (tile,tile) --> bool }
where
  forAll u,v : tile .
    init[up]((u,v))    <-> u = v + tile_size /\
    init[down]((u,v))  <-> u = v - tile_size /\
    init[left]((u,v))  <-> (u = v + 1) /\
      ((u % tile_size) != 1) /\ $ left: ignore first cell on each row
    init[right]((u,v)) <-> (u = v - 1) /\
      ((u % tile_size) != 0),  $ right: ignore last cell on each row
```

Fig. 7: Expression for the `isLRUDSquareGrid` keyword in ESSENCE

We introduce new PDDL keywords to express those structural constraints, and provide automated translation of those keywords to a CP model in ESSENCE through rantanplan. An example on expressing a square grid using {`left, right, up, down`} predicates is shown in Figure 7. The newly introduced PDDL keyword `isLRUDquareGrid` is used and an auxiliary variable indicating the size of the square grid (variable `tile_size` in the ESSENCE specification) is generated and will be tuned by `irace` during the instance generation process. However, there are two limitations of this automated approach, as we will explain below.

The first limitation is on the *flexibility* to express structural validity constraints. Consider the square-grid structure as an example. There are various ways to define the local connections of cells in a square grid. The choice is normally made based on the specification of planning actions for a specific planning problem. For `floor-tile`, the locality relations {`up, down, left, right`} are used as the moving actions of a robot at each tile can only follow those directions. However, for the `city-car` problem, a car can either move horizontally or diagonally. The underlying square-grid map in `city-car` is then represented using two predicates: `same_line` and `diagonal` for every ordered pair of horizontally and diagonally adjacent cells, respectively.

The second limitation is about *scalability*. As PDDL predicates are basically boolean functions, the size of validity constraints expressed directly using those predicates can grow very quickly. For example, the square-grid structure expressed in Figure 7 needs to use a nested for loop `for u,v: tile`. For a grid size of 20×20, the total number of constraints is 4×20^4. On a computer with Intel

Xeon E5-2640 2.4Ghz CPUs, generating a single instance with such grid size using the CP solver `minion` takes about 20 minutes. The time increase also grows fast, as it takes more than 30 minutes to generate a 22×22 grid.

6 Expressing the Generation of Planning Problems in essence

In this section, we discuss an alternative approach to the augmented-PDDL instance generation proposed in the previous section. This is a hybrid approach where users model validity constraints directly in ESSENCE. A summary of this approach is shown in Figure 8. Compared to the previous approach in Figure 4, step 1.2 is now done manually by users. It offers solutions to the two limitations of the augmented-PDDL approach.

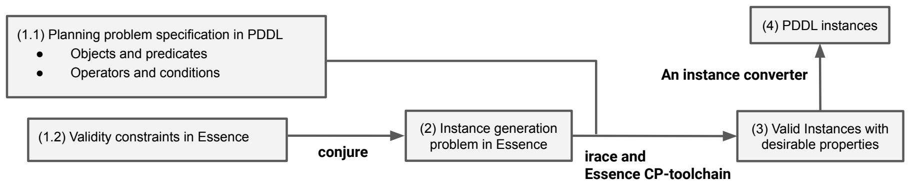
(1.1) Planning problem specification in PDDL
• Objects and predicates
• Operators and conditions
(1.2) Validity constraints in Essence
conjure
(2) Instance generation problem in Essence
irace and Essence CP-toolchain
(3) Valid Instances with desirable properties
An instance converter
(4) PDDL instances

Fig. 8: Automated instance generation for planning problems using a hybrid approach, with problem specification in PDDL and validity constraints in ESSENCE.

Firstly, the new approach allows users to make full use of the high-level modelling language ESSENCE to freely express any validity constraints using constraint modelling, instead of relying on a predefined set of keywords supported by rantanplan. This overcomes the first limitation on flexibility.

Secondly, by modelling validity constraints directly in ESSENCE, users are no longer tied to the relatively low-level boolean representations of PDDL. It is well-known that recovering structure from low-level representation is a difficult and expensive process, as illustrated in the work of Helmert [10] where many-valued variables were detected from PDDL representations with Boolean variables. By expressing the constraints directly in ESSENCE, knowledge about implicit structures present in the problem can be easily expressed, which results in more *efficient* representations. For example, in the square-grid structure of `floor-tile`, each of the `up, down, left, right` relations can be expressed as a function mapping from each tile in the grid to at most one other tile. Compared to the low-level representations using boolean functions as in Figure 4, this new representation significantly reduce the number of validity constraints (from $4 \times n^4$ to $4 \times n^2$ for any $n \times n$ square grid). Figure 9 shows a comparison of the time required by our system to generate square-grid structures for the `floor-tile` problem using the augmented-PDDL approach and the new one described in this section. The results clearly indicate that the high-level representation (the orange line) significantly improve the efficiency of the instance generation process.

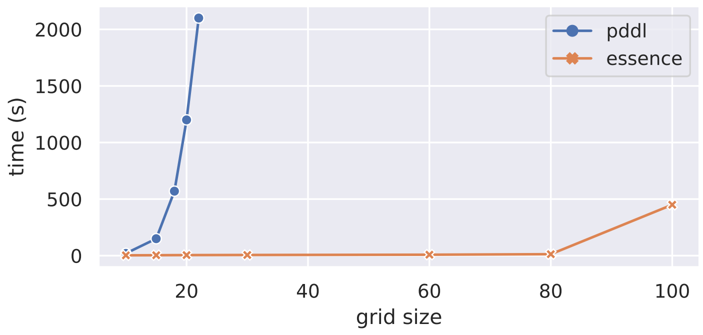

Fig. 9: Time to generate floor-tile square-grid instances using our automated instance generation system. The blue line is the approach described in Section 5, and the orange line is the one described in Section 6.

Another advantage of using high-level representations is that some validity constraints are automatically encoded inside the abstract ESSENCE types themselves without the need of any explicit constraints to express them. An example of the full description of all validity constraints for the `floor-tile` problem is illustrated in Figure 10. As we can see, the first two validity constraints, which require that each robot can be in only one tile and has one colour at a time, are automatically satisfied thanks to their representations as total functions from `robot` to `tile` (`robot_at`) and from `robot` to `color` (`robot_has`).

Despite all the advantages explained above, this approach has a major limitation in terms of *automation*. Instead of writing only one specification for each planning problem of interest, users now have to provide two extra separated inputs to the system. The first one is an ESSENCE specification expressing the validity constraints (step 1.2 in Figure 8), with variable names matched with the predicates in the original PDDL problem description (step 1.1 in Figure 8). As there are several possible PDDL representations from a high-level abstract description of a problem, the second input is a manually written program to convert the instances in ESSENCE back to the PDDL representation specified in Step 1.1 (from step 3 to step 4 in Figure 8). It is extremely difficult to automate the generation of such converter as the system has to recognise which PDDL representation is the right match. In the next section, we propose an elegant approach that can overcome all limitations on flexibility and efficiency discussed so far without any trade-offs on automation. As we will explain, the whole instance generation process can be fully automated and various encodings, including PDDL, could be supported in a completely transparent manner.

```
given n_robot : int(1..)
given tile_size: int(1..)
letting n_tile be tile_size*tile_size
given n_color : int(1..)
letting robot be domain int(1 .. n_robot)
letting tile be domain int(1.n_tile)
letting color be domain int(1..n_color)

$ ----- Initial State --------
given init: record {
  robot_at : function (total) robot --> tile,
  robot_has : function (total) robot --> color,
  up : function tile --> tile,   down : function tile --> tile,
  left : function tile --> tile, right : function tile --> tile,
  clear : set of tile, available_color : set of color}
where
  forAll c: color . c in init[available_color], $ all colors are available
  forAll u : tile .
    u in init[clear], $ all titles are clear
    $ square-grid constraints
    u in defined(init[up]) <-> init[up](u) = u - tile_size,
    u in defined(init[down]) <-> init[down](u) = u + tile_size,
    u in defined(init[left]) <-> (init[left](u) = u - 1)
        /\ (u % tile_size != 1), $ left: ignore first cell on each row
    u in defined(init[right]) <-> init[right](u) = u + 1
        /\ (u % tile_size != 0) $ right: ignore last cell on each row
given goal: record {painted : function (minSize 1) tile --> color}
```

Fig. 10: Validity constraints for the whole `floor-tile` problem expressed directly in ESSENCE

7 Abstract Specification of Planning Problems

This section discusses abandoning a given PDDL description of a planning domain as the starting point for generating instances, and instead extending the ESSENCE language to support the abstract specification of planning problems.

The key feature of the ESSENCE language is the provision of high-level type constructors, such as `set`, `relation` and `function`, which allow a problem to be specified directly in terms of the combinatorial structure to be found. Specifying a planning problem in ESSENCE would remove the considerable difficulty of trying to recover this structure from a PDDL description as discussed in the preceding sections. This would simplify the process of instance generation for a planning problem. By providing a refinement of a planning problem specification to a PDDL model, we can also automate much of the work of producing a PDDL encoding of a problem and ensure that the instances generated are synchronised with the model chosen. These advantages result in a straightforward adaptation of the CP automated instance generation system, as presented in Figure 11.

A simple approach to enabling the specification of planning problems in ESSENCE is to introduce a plan type constructor. In much the same way as PDDL, this would need to support a representation of the objects in the domain, initial and goal states, and plan operators. However, we would gain the far more expressive types available in ESSENCE in order to specify each of these

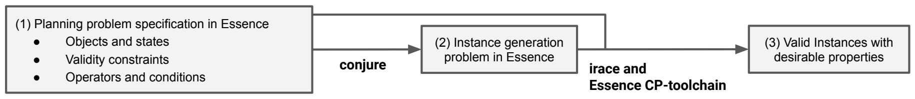
(1) Planning problem specification in Essence
Objects and states
Validity constraints
Operators and conditions
conjure
(2) Instance generation problem in Essence
irace and Essence CP-toolchain
(3) Valid Instances with desirable properties

Fig. 11: Automated instance generation for planning problems in ESSENCE.

```
given n_robot   : int(1..)
given n_colour  : int(1..)
given tile_size : int(1..)
letting CLEAR be 0
letting GRID be domain matrix indexed by [int(1..tile_size), int(1..tile_size)] of
      int(CLEAR..)
letting COLOUR be new type of size n_colour
letting STATE be domain record {
  robots : sequence (size n_robot) of record{row : int, column: int,
                                             colour: COLOUR},
  grid : GRID}
given init : STATE
where $ all tiles are clear at the initial state
    forAll i, j : int(1..tile_size) . init[grid][i,j] = CLEAR
given goal : GRID

find p : plan with state STATE
              with initialState init
              with goalState state[grid] = goal
              with actions [goUp, goDown, goLeft, goRight, paintUp, paintDown,
                   changeColour]
```

Fig. 12: A possible ESSENCE specification of the `floor-tile` problem.

elements. PDDL adopts an implicit frame condition in which all parts of the state not explicitly referenced by an action are assumed to be unchanged, which it also seems sensible to employ in an ESSENCE plan type constructor.

We will illustrate with a hypothetical ESSENCE specification of the `floor-tile` problem. Figure 12 presents the specification of the abstract plan decision variable `p` via a new `plan` type constructor, which expects four arguments. The first is the state of the planning domain, which is specified using the existing `record` type, here capturing the position and colour of each of the robots as well as the current grid state. The initial state is given as a parameter of the same type. The goal, which may only concern a part of the problem state, is flexibly expressed as a set of constraints. For the `floor-tile` domain, the goal is a particular grid configuration, expressed as an equality on the grid part of the plan state.

The final argument is the list of available actions. Figure 13 presents a hypothetical action representing movement upwards on the grid. The `goUp` action is parameterised on the element of the state that must be selected by the planner, in this case which of the robots to move. Preconditions and effects are expressed as constraints on the parts of the state affected, hence benefiting from the abstract types and operators in ESSENCE. As in PDDL, there is an implied frame condition that any parts of the state not mentioned are unchanged.

```
letting goUp(r in robots) be domain
    action { precondition: grid[r[row]-1, r[column]] = CLEAR,
             effects: r[row]' = r[row]-1}
```

Fig. 13: A hypothetical ESSENCE plan action compatible with the plan state defined in Fig. 12. The parameter `r` is to be chosen by the planner from among those robots in the plan state. The operator `'` denotes the state at the subsequent step in the plan. An action such as this could be further annotated with a cost value, if required.

An ESSENCE specification of a planning problem such as the one described in this section could be used for both instance generation and automated modelling. The parameters of the plan domain are apparent in the specification of Figure 12 and, via the types available in ESSENCE, their structure is apparent rather than having to be recovered from a lower level description. Most of the validity constraints for the `floor-tile` problem are implicitly implied in the high-level representations. This simplifies instance generation considerably, and would allow a similar approach to that used for ESSENCE specifications of constraint satisfaction/optimisation problems [1].

Automated modelling could also follow the current practice of refining ESSENCE specifications into constraint models using CONJURE. With PDDL as the target language, refinement rules would have to be written to encode the ESSENCE types and operators describing the plan state and actions. As is the case for constraint models, multiple refinement rules could be written for the same type to enable alternative PDDL encodings to be generated automatically. A further opportunity would be to exploit the existing CONJURE infrastructure to refine the specification to a constraint model of the planning problem, providing alternative solution options via CP, SAT, or SMT through Savile Row.

8 Conclusion and Future Work

We have discussed various approaches for adapting a CP automated instance generation system for planning where the problem descriptions are written in PDDL, the standard modelling language for planning problems. The limitations of those approaches are explained and a new language for describing planning problems using ESSENCE, a high-level constraint modelling language, is proposed. Automated instance generation for planning based on ESSENCE offers greater flexibility, efficiency and expressivity compared with its PDDL-based counterpart. In future work, an implementation of the proposal will be provided and a thorough evaluation of such an instance generation system will be done.

Acknowledgements This work is supported by EPSRC grant EP/P015638/1. Nguyen Dang is a Leverhulme Early Career Fellow.

References

1. Akgün, Ö., Dang, N., Miguel, I., Salamon, A.Z., Stone, C.: Instance generation via generator instances. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 3–19. Springer (2019). https://doi.org/10.1007/978-3-030-30048-7_1
2. Akgün, Ö., Dang, N., Miguel, I., Salamon, A.Z., Spracklen, P., Stone, C.: Discriminating instance generation from abstract specifications: A case study with CP and MIP. In: CPAIOR (2020)
3. Akgün, Ö., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure. In: CP 2013. LNCS, vol. 8124. Springer (2013). https://doi.org/10.1007/978-3-642-40627-0_11
4. Akgün, Ö., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: AAAI 2011. pp. 4–11. AAAI Press (2011), https://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/viewPaper/3687
5. Bacchus, F., Kabanza, F.: Using temporal logics to express search control knowledge for planning. Artificial Intelligence **116**(1-2), 123–191 (2000)
6. Bofill, M., Espasa, J., Villaret, M.: The RANTANPLAN planner: system description. Knowledge Eng. Review **31**(5), 452–464 (2016)
7. Fuentetaja, R., De la Rosa, T.: A planning-based approach for generating planning problems. In: Workshops at AAAI (2012)
8. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI. pp. 98–102. IOS Press (2006), http://ebooks.iospress.nl/volumearticle/2658
9. Haslum, P., Scholz, U.: Domain knowledge in planning: Representation and use. In: Workshop on PDDL at ICAPS (2003)
10. Helmert, M.: Concise finite-domain representations for PDDL planning tasks. Artificial Intelligence **173**(5-6), 503–535 (2009)
11. van Hemert, J.I.: Evolving binary constraint satisfaction problem instances that are difficult to solve. In: The 2003 Congress on Evolutionary Computation, 2003. CEC'03. vol. 2, pp. 1267–1273. IEEE (2003)
12. Horie, S., Watanabe, O.: Hard instance generation for SAT. In: International Symposium on Algorithms and Computation. pp. 22–31. Springer (1997)
13. Julstrom, B.A.: Evolving heuristically difficult instances of combinatorial problems. In: GECCO 2009. pp. 279–286. ACM (2009). https://doi.org/10.1145/1569901.1569941
14. Kautz, H.A., Selman, B.: The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework. In: Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, Pittsburgh, Pennsylvania, USA. pp. 181–189 (1998)
15. Kopczyński, E.: Axiomatizing rectangular grids with no extra non-unary relations. arXiv:1912.09797v1 (2019), https://arxiv.org/abs/1912.09797v1
16. Libkin, L.: Elements of Finite Model Theory. Springer (2004). https://doi.org/10.1007/978-3-662-07003-1
17. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016). https://doi.org/10.1016/j.orp.2016.09.002, http://iridia.ulb.ac.be/irace/
18. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL – the planning domain definition language (1998)

19. Moreno-Scott, J.H., Ortíz-Bayliss, J.C., Terashima-Marín, H., Conant-Pablos, S.E.: Challenging heuristics: evolving binary constraint satisfaction problems. In: Proceedings of the 14th annual conference on Genetic and evolutionary computation. pp. 409–416 (2012)
20. Muñoz, M.A., Villanova, L., Baatar, D., Smith-Miles, K.: Instance spaces for machine learning classification. Machine Learning **107**(1), 109–147 (2018). https://doi.org/10.1007/s10994-017-5629-5
21. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/10.1016/j.artint.2017.07.001
22. Selman, B., Mitchell, D.G., Levesque, H.J.: Generating hard satisfiability problems. Artificial Intelligence **81**(1-2), 17–29 (1996)
23. Shleyfman, A., Karpas, E.: Position paper: Reasoning about domains with PDDL. In: 2018 AAAI Spring Symposia, Stanford University, Palo Alto, California, USA. AAAI Press (2018)
24. Smith-Miles, K., Baatar, D., Wreford, B., Lewis, R.: Towards objective measures of algorithm performance across instance space. Computers & Operations Research **45**, 12–24 (2014)
25. Smith-Miles, K., Lopes, L.: Measuring instance difficulty for combinatorial optimization problems. Computers & Operations Research **39**(5), 875–889 (2012)
26. Ullrich, M., Weise, T., Awasthi, A., Lässig, J.: A generic problem instance generator for discrete optimization problems. In: GECCO 2018. pp. 1761–1768. ACM (2018). https://doi.org/10.1145/3205651.3208284
27. Vallati, M., Chrpa, L., Grześ, M., McCluskey, T.L., Roberts, M., Sanner, S., et al.: The 2014 international planning competition: Progress and trends. AI Magazine **36**(3), 90–98 (2015)
28. Vanhoucke, M., Maenhout, B.: On the characterization and generation of nurse scheduling problem instances. European Journal of Operational Research **196**(2), 457–467 (2009). https://doi.org/10.1016/j.ejor.2008.03.044
29. Xu, K., Boussemart, F., Hemery, F., Lecoutre, C.: Random constraint satisfaction: Easy generation of hard (satisfiable) instances. Artificial Intelligence **171**(8-9), 514–534 (2007). https://doi.org/10.1016/j.artint.2007.04.001