Modelling Multi-dimensional Arrays in Unified Planning

Carla Davesa ✉
School of Computer Science, University of St Andrews, UK

Joan Espasa
School of Computer Science, University of St Andrews, UK

Ian Miguel
School of Computer Science, University of St Andrews, UK

Mateu Villaret
Departament d'Informàtica, Matemàtica Aplicada i Estadística, Universitat de Girona, Spain

Abstract

Planning is a fundamental activity, arising frequently in many contexts, from daily tasks to industrial processes. It consists of selecting a sequence of actions to achieve a specified goal from specified initial conditions. The Planning Domain Definition Language (PDDL) is the leading language used in the field of automated planning to describe planning problems. Previous work has highlighted the limitations of PDDL, particularly in terms of its expressivity. Our interest lies in facilitating the handling of complex problems to enhance the overall capability of automated planning systems. To pursue this agenda, we must identify patterns in the planning problems that we wish to solve. The present work is motivated by the prevalence of grid structure in planning problems, which is a significant challenge to model in PDDL, especially when complex operations on grid indices are required. We situate our work in Unified Planning (UP), which offers an API to specify planning problems and transform them into inputs suitable for specific planners. We present an extension of the UP library to enhance its expressivity for high-level problem modelling: we have added an array type, an expression to count satisfied Boolean expressions, an integer range variable type, and support for integer parameters in actions. Within the UP framework, we have developed compilers that transform these extensions into acceptable representations for the UP's integrated planners. By providing alternative compilation pathways we enable the automated exploration of a variety of models with different performance characteristics from a single high-level model. We show that our UP extensions enable natural high-level models for six benchmark planning problems, and support the automatic generation of multiple low-level variants. The resulting models are competitive with, and in some cases outperform, hand-crafted models from the literature.

2012 ACM Subject Classification Theory of computation; Computing methodologies → Planning and scheduling

Keywords and phrases Automated Planning, Reformulation, Modelling

1 Introduction

Automated Planning is a branch of Artificial Intelligence that focuses on selecting sequences of actions to achieve desired goals from specified initial conditions. Planning problems arise frequently in many contexts, from daily tasks to industrial processes.

The Planning Domain Definition Language (PDDL)[9] is the leading language used in automated planning to model planning problems. It provides a formal way to describe a problem in terms of objects, predicates, actions, and functions with parameters. A study [16] discussed the limitations of PDDL, highlighting its low level of abstraction, which forces the modeller to encode ubiquitous n-valued state variables (e.g. bounded integer variables) as sets of Boolean variables. We aim to provide a more expressive modelling language, combined with automated methods for compilation to PDDL, both to remove the burden from the

user of manual modelling in PDDL and to support the automatic exploration of alternative models. To pursue this agenda, we need to identify patterns in the planning problems that we wish to solve. The present work is motivated by the prevalence of grid structure in planning problems, which is a significant challenge to model in PDDL [3], especially when complex operations on grid indices are required.

We situate our work in Unified Planning (UP) [14], a Python library with an API to specify planning problems and can transform them into PDDL. Our main contribution is a set of UP extensions, including a new Array type for multidimensional structures, full support for bounded integer parameters in actions, a new integer range variable type, and an expression to evaluate the number of satisfying statements among multiple Boolean expressions. We have also developed six compilers that can automatically compile these features away, maintaining compatibility with the supported planning engines and infrastructure. By providing alternative compilation pathways we also enable the exploration of a variety of models with different performance characteristics from a single high-level model. We show how our extensions support natural modelling of six benchmark planning problems and demonstrate that the compiled models are competitive with, and sometimes outperform, handcrafted models from the literature.

2 Background and Related Work

A classical planning problem consists of finding a *plan*: a sequence of actions applicable from the initial state to a goal state. *States* are typically defined by Boolean state variables (*fluents*) and *actions* by their *preconditions* that define in which states an action is *applicable*, and *effects* that define the changes to perform on the values of the state variables once the action has been applied. The *goal* is usually a (partial) valuation over the fluents. State changes occur only through action applications, and effects are fully deterministic.

The Planning Domain Definition Language (PDDL) [9] provides a declarative language for modelling planning problems. Over the years, PDDL has evolved to include support for features such as numeric types, temporal constraints, and conditional effects. However, some of these extensions are "conditioned" by the limitations of search-based solvers, which do not allow functionalities such as integer parameters in action templates, nor support complex data structures such as arrays. Several works remark the significant lack of expressiveness in PDDL, particularly when it comes to representing more complex planning scenarios [8, 16, 3].

Previous works in the literature have explored the idea of enhancing planning languages with more expressive representations. Functional STRIPS [7] or Planning Modulo Theories (PMT) [11], for example, could support multi-dimensional arrays. Unfortunately, these approaches are either unreleased, or no longer maintained. Hence, by considering the available and well supported planners, we are limited to using state-of-the-art planners based on PDDL despite their limited expressivity. Languages such as ANML [18] can express structured types, but no planner supports it. Recent work [1] proposed a modelling language supporting complex data types which are reduced to a Boolean representation and then PDDL, leveraging existing classical planners.

Our approach differs by treating numeric planning as a first-class citizen, supporting it as both source and target in our transformations. In addition, we integrate these capabilities directly into the UP library [14], taking advantage of all the facilities that the library provides. That is, our work extends the foundation of the library by using the simplicity of Python and extensive ecosystem to create more intuitive models.

3 The Unified Planning Framework Pipeline

Unified Planning (UP) [14] streamlines the process of transforming planning problems into formats suitable for input to various planners.

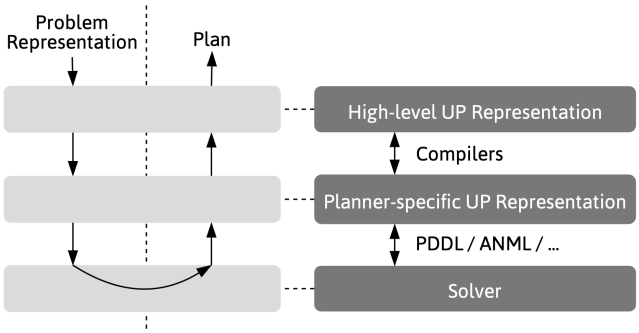

Figure 1 Automated Planning Modelling Pipeline

Figure 1 shows the stages of this pipeline. The first step is to define a planning problem instance, which serves as a container for fluents, actions, objects, initial state, and goals. Objects are typed entities in the domain. Like PDDL, UP uses a lifted representation of the problem, with parameterised state variables and actions, enabling a concise definition of the problem.

We distinguish two representation levels to clarify when compilation is needed. If the problem contains no high-level features or the planner already supports them, no compilation is required.

High-Level UP Representation: Planning problems can be modelled using high-level features like conditional effects, quantifiers, and user-type fluents. These features might not be supported by all planners, requiring the use of compilers to transform the problem into a compatible low level. We categorise the proposed implementations, array type, count expression, range variable, and integer parameters in actions, as high-level representations.

Planner-Specific UP Representation: After applying the necessary compilers, the problem is simplified to match the features supported by the target planner, while preserving its original semantics. UP can then transform the representation of problems into various specific planning languages such as PDDL and ANML [18].

Solver: Finally, the converted planning problem is fed into a planner for solution. UP provides access to a variety of planners, each supporting particular fragments of PDDL or ANML. The planner employs search algorithms and planning techniques to find a plan that transitions from the initial state to the goal state.

4 The Proposed UP Extensions

We present four extensions to the Unifed Planning framework that enhance modelling flexibility: support for array types, integer parameters in actions, a new expression to count satisfied Boolean conditions, and a new variable type for quantification over integer ranges. We illustrate these contributions through two running examples: n-Puzzle and Plotting.

Array Type We introduce the `ArrayType` class in UP to represent arrays with a fixed number of elements of a specified type. It takes two parameters: `size`, specifying the number of elements; and `elements_type`, defining the type of each element (defaulting to Boolean if not provided, consistent with UP's default fluent type behaviour). By allowing fluents to have an `ArrayType` as their type, each element in the array can be accessed and treated as an individual fluent, while their relative positions within the array are inherently preserved. Furthermore, since arrays are considered types themselves, it becomes possible to define nested arrays, such as tables or matrices.

```
# Declaring types, objects and fluents
T = UserType("Tile")
t0 = Object("t0", T) ... t8 = Object("t8", T)
g = Fluent("grid", ArrayType(3,ArrayType(3,T)))
p.add_fluent(g, default_initial_value=t0)
# Specifying the initial and goal state
p.set_initial_value(g, [[t1,t2,t3],[t5,t6,t8],[t4,t7,t0]])
p.add_goal(g, [[t1,t2,t3],[t4,t0,t5],[t6,t7,t8]])
```

Listing 1 Defining the 8-Puzzle problem of Figure 2.

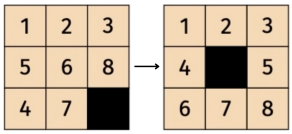

Figure 2 8-Puzzle instance example.

```
(:action move-right
  :parameters (?r ?c ?nc - idx ?t - tile)
  :precondition (and (grid ?t ?r ?c)
    (next ?nc ?c) (empty ?r ?nc))
  :effect (and
    (not(empty ?r ?nc)) (grid ?t ?r ?nc)
    (not(grid ?t ?r ?c)) (empty ?r ?c)))
```

Listing 2 8-Puzzle `move-right` in PDDL.

```
mr = InstantaneousAction("move-right",
        r=IntType(0,2), c=IntType(0,1))
mr.add_precondition(Equals(grid[r][c+1], t0))
mr.add_effect(grid[r][c+1], grid[r][c])
mr.add_effect(grid[r][c], t0)
```

Listing 3 8-Puzzle `move-right` using Integer Parameters.

The n-Puzzle (see Figure 2) is a classic AI problem where tiles must be moved into an adjacent empty cell to reach a goal configuration. Without arrays, modelling requires explicitly defining all adjacency relations, resulting in a long and error-prone model. Arrays make these relations implicit: the grid is modelled as a 2D fluent (Listing 1), and initial and goal states can be compactly expressed as 2D arrays.

Integer-Type Parameters in Actions PDDL only supports object parameters [5], so arrays are often modelled using fluents indexed by coordinate objects. For example, the 8-Puzzle `move_right` action must include positions and values as parameters (Listing 2). To simplify this, we introduce bounded integer parameters in actions, allowing natural indexing and arithmetic over arrays. For example, the `move-right` action in Listing 3 uses a bounded column parameter `IntType(0,1)` to restrict moves within valid columns. This leads to simpler and more intuitive action definitions.

Count Expression To support cardinality constraints, we introduce the `Count` expression, which returns how many of its Boolean arguments are satisfied. It can be written as `Count(a, b, c)` or `Count([a, b, c])`, where each argument is a Boolean expression. We illustrate the use of this expression with Plotting, where the goal is to reduce the number of blocks in the grid to a target number or fewer. The avatar shoots blocks into the grid. If the shot block hits one of the same pattern, that block is removed. State changes are complex because multiple blocks may be removed in a single shot and gravity affects the blocks in the grid. In our model, the fluent `blocks` is a 2D array of `Colour` values indexed by integer `row` and `column`. The condition in Listing 4 ensures that the number of non-empty cells (those not equal to `N`) does not exceed a threshold (e.g., 2). Python list comprehensions provide a concise way to construct the Boolean expressions counted in this condition.

Integer Range Variables With the introduction of arrays and integer parameters in actions, it becomes natural to express properties over collections using iteration, particularly in quantifiers for preconditions and effects. However, in classical planning, quantifiers such as *forall* and *exists* are restricted to user-defined types, so variables can only range over declared objects. This restriction makes it impossible to directly quantify over a sequence of integers, requiring manual unrolling or Python checks, which is both cumbersome and error-prone.

Figure 3 2x3 Plotting instance.

```
plotting.add_goal(LE(Count([Not(Equals(blocks[i][j], N))
    for i in range(2) for j in range(3)]), 2))
```

Listing 4 Plotting goal condition with Count.

To address this limitation, we introduce the `RangeVariable` type to represent a bounded integer variable, defined by a lower and upper bound, which can be constants or parameters.

5 Compilers

This section presents six compilers that translate high-level constructs into low-level representations compatible with standard languages like PDDL. While the output may be verbose, it reveals the complexity hidden in high-level models—complexity that would be tedious to encode manually. Some constructs offer multiple compilation options, supporting the exploration of alternative models from a single specification. All compilers follow the UP library structure to ensure consistency and tool compatibility.

To correctly apply the new compilers they must follow a specific order. Figure 4 illustrates the required sequence based on the features present in the problem, guiding the transformation towards a representation compatible with classical planners. Since planner capabilities vary, not all features need to be removed for every problem–planner combination.

IntParameterActionsRemover (IPAR) Handles actions with integer parameters by converting them into a set of (partially) grounded actions, one per each possible combination of parameter values within their bounds. For each such action, the compiler generates new versions in which the integer parameters are replaced by specific values within the preconditions and effects. During this process, arithmetic expressions involving integer parameters are also simplified.This compiler also processes `RangeVariable` constructs by expanding integer quantifiers into equivalent logical expressions.

ArraysRemover (AR) Arrays are eliminated by replacing each element with a separate fluent or value. In the 8-Puzzle example, the original multidimensional array fluent *grid* becomes nine new fluents, one for each cell in the 3×3 grid. All array accesses in the problem are then replaced by references to the corresponding fluent.

ArraysLogarithmicRemover (ALR) Converts multidimensional integer array fluents into a Boolean encoding compatible with classical planners. For each bounded integer fluent with range $[0, n]$, it introduces $\lceil \log_2(n+1) \rceil$ Boolean fluents, representing the bits of its binary encoding. Each array position becomes an object of a new `Position` user type, used as a parameter in the resulting Boolean fluents. For example, in the 8-Puzzle (Listing 1), using `IntType(0,8)` encodes each tile with 4 bits, generating fluents `g_b0`–`g_b3` and `Position` objects like `p_0_0`, ..., `p_2_2`. A tile value of 1 at $(0, 0)$ is thus encoded as 0001.

CountRemover (CR) Transforms the cardinality constraints involving `Count` expressions by replacing them with an equivalent disjunction that explicitly enumerates all satisfying assignments. Although the size of the resulting formula grows combinatorially with the number of variables involved, this approach guarantees compatibility with classical planners.

CountIntRemover (CIR) Since our encoding now supports numeric fluents, we preserve cardinality constraints by counting how many Boolean expressions in a `Count` evaluate to true. For each such expression, the compiler creates an integer fluent set to 1 if the expression holds, and 0 otherwise. Repeated subexpressions reuse the same fluent to avoid redundancy. The original `Count` is replaced by a `Plus` over these integer fluents. Their initial values reflect the evaluation of the corresponding expressions in the initial state. When any fluent involved

in a `Count` changes, the compiler adds conditional effects to update the associated integer fluent accordingly, maintaining consistency.

IntegersRemover (IR) Replaces integer values with objects of a new user type `Number`, assuming a finite range. Integer values in preconditions and effects are replaced by these objects, and arithmetic operations are converted into Boolean predicates, generated only as needed and precomputed over all combinations. *Increase* and *Decrease* are handled via conditional effects that assign the correct next value. Finally, the **UserTypesFluentRemover (UFR)** compiler converts user-type fluents into Boolean formulas for classical planner compatibility.

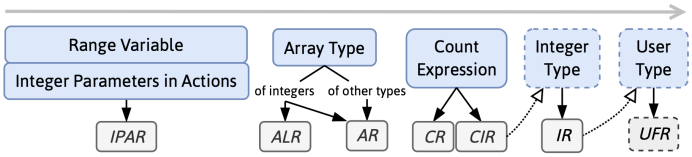

Figure 4 Compiler sequence by features. Boxes - blue: features; grey: compilers; dashed: existing components. Arrows - grey: order; black: applicable compilers; dashed: introduced.

6 Experiments

We evaluate our new UP-extended implementation by encoding and solving six domains: 15-Puzzle, Pancake Sorting, Plotting, Rush Hour, Sokoban, and Puzznic alongside existing handcrafted PDDL models. All the models employing our extensions are more natural and concise than their low-level PDDL equivalents, which tend to be long and prone to error. We applied several compilation strategies to each of our models to obtain different compiled versions to compare with the handcrafted PDDL models (Table 2). This illustrates the ability of our system to explore different low-level models from a single high-level representation, and indeed we will see that different strategies perform better depending on the problem. To ensure fair comparison, we converted the handcrafted models into UP's internal representation using its PDDL parser, so that all variants were processed uniformly and submitted to the planners in the same format.

We analysed the impact of our UP extensions by comparing them with handcrafted PDDL versions. To ensure consistency and demonstrate compatibility with standard planning tools, we used the UP-integrated planners: Fast Downward [12] (v24.06) and SymK [19] (v1.3.1), and ENHSP [17] (v20) for numeric problems [1]. All planners were used with their default configurations and a timeout of 30 minutes. We ran three operation modes: *OneShot* to obtain an initial solution, *Anytime* for the best solution found within the time limit; and *Optimal*, to guarantee optimality if found in time. Since Fast Downward's default heuristic within UP in optimal mode does not support conditional effects, we applied UP's conditional effects compiler when required. All experiments were run on a 20-node cluster with Intel(R) Xeon(R) E-2234 CPUs @ 3.60 GHz with 16GB of RAM. Table 1 shows the percentage of solved instances per problem, compilation, and mode.

15-Puzzle We have introduced the n-Puzzle as a well-known running example. Due to the negligible solving times of the 8-Puzzle, we focused our evaluation on the more challenging 15-Puzzle. We used the 100 instances from the ICAPS 2015 benchmark repository [13], and the handcrafted model available there, which serves as our baseline for comparison. We

[1] Other available engines were excluded due to lack of support for required features or installation issues.

		OneShot			Anytime			Optimal		
		F	S	E	F	S	E	F	S	E
15-Puzzle	handcrafted	100	32	x	100	32	x	20	32	x
	up	100	0	x	100	0	x	0	0	x
	ut-integers	100	0	x	100	0	x	0	0	x
	logarithmic	100	38	x	100	41	x	0	37	x
	integers	x	x	100	x	x	100	x	x	0
pancake	handcrafted	40	20	x	40	20	x	0	20	x
	up	84	40	x	84	40	x	0	40	x
	ut-integers	84	40	x	84	40	x	0	40	x
	logarithmic	50	42	x	42	48	x	0	42	x
	integers	x	x	50	x	x	50	x	x	40
plotting	handcrafted	43	11	x	43	11	x	0	11	x
	count	72	37	x	72	36	x	0	37	x
	count-int	68	24	x	68	24	x	0	24	x
	count-int-num	x	x	97	x	x	97	0	x	91
pzn	handcrafted	5	33	x	5	31	x	0	0	x
	up	3	8	x	3	10	x	0	0	x
rh	handcrafted	100	100	x	100	100	x	100	100	x
	up	100	100	x	100	100	x	100	100	x
sok	handcrafted	95	80	x	95	75	x	18	80	x
	up	90	95	x	90	95	x	18	95	x

Table 1 Percentage of instances solved for problem-compilation versions and solving methods, rounded to the nearest integer. An "x" indicates that the specific compilation-solving configuration was not tested. The solvers are denoted as follows: 'F' = Fast Downward, 'S' = SymK, 'E' = ENHSP.

Compilation Strategies	Compilers Sequence
up	*IPAR*, *AR*, *UFR*
logarithmic	*IPAR*, *ALR*
ut-integers	*IPAR*, *AR*, *IR*, *UFR*
count	*IPAR*, *AR*, *CR*, *UFR*
count-int	*IPAR*, *AR*, *CIR*, *IR*, *UFR*
count-int-num	*IPAR*, *AR*, *CIR*, *UFR*
integers	*IPAR*, *AR*

Table 2 Compilation strategies with their compilers application sequence. All strategies are compatible with classical planners, except *count-int-num* and *integers*, which preserve numeric fluents and require a numeric planner.

experimented with two versions of our UP-extended model: a numeric version, where tile values are represented as integers, and a non-numeric version, where tiles are represented as user-types objects. From the numeric model, we derived several compilation strategies: *integers*, *logarithmic*, and *ut-integers*, while the non-numeric model led to the *up* version.

Plotting [3] was used as a running example, with its goal expression using our extension `Count`. As this can be compiled in multiple ways, we evaluated three strategies: *count*, *count-int*, and *count-int-num*. We compared our UP-extended models with a manually created one [3]. The original repository includes several groups of instances, where each group varies only by colour assignments, with size and structure consistent across instances. We selected all 87 instances from the first group, ensuring sufficient variety. During evaluation, we found and corrected several bugs in the handcrafted model that produced incorrect plans—highlighting the challenges of modelling complex problems manually in PDDL.

Sokoban is a puzzle game in which an agent pushes boxes onto target locations in a grid-based warehouse. The main challenge is to avoid pushing boxes into positions from which they cannot be moved, requiring careful planning. For the handcrafted model, we used the 39 instances and domain available in the IPC 2011 benchmark [15].

Rush Hour is a sliding block puzzle played on a 6×6 grid, where blocks represent vehicles stuck in a traffic jam. The goal is to move the red car to the exit located on the right edge of the grid. Vehicle movements are constrained: they can only move forward or backwards and cannot cross over other vehicles. For our experiments, we selected the 100 instances with the largest minimum plan lengths from Fogleman's database [4]. We used the handcrafted PDDL model provided in the GitHub repository by Hajdini [10].

Puzznic is a tile matching puzzle game in which the player must move blocks within a grid so that identical blocks touch and disappear, following gravity constraints. Solving each

Pancake Sorting [6] involves ordering a stack of pancakes so that the smallest is on top and the largest at the bottom. The only allowed operation is to flip the top k pancakes, making the problem non-trivial. As no PDDL model was available, we created one and compared it with our UP-based model using the same compilation strategies as for the 15-Puzzle. To ensure diversity and scalability, we tested 10 random instances for sizes 5 to 25.

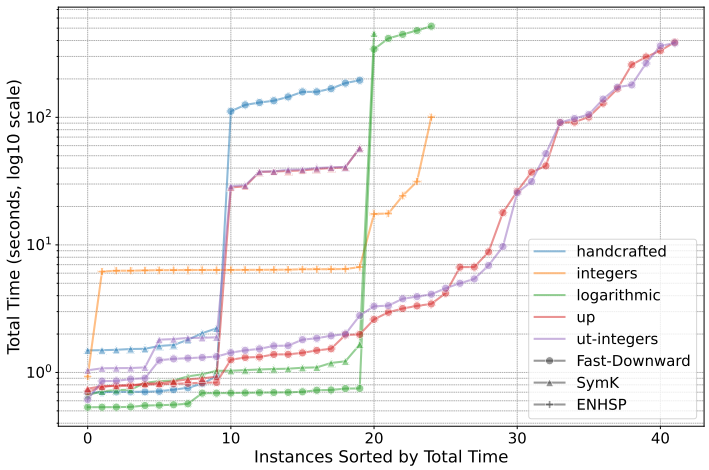

Figure 5 Pancake Problem: Total Time by Compilation and Solving (OneShot). Colours indicate different compilation versions, while line styles and markers distinguish solvers.

level requires planning a sequence of moves to avoid blocking necessary paths or isolating tiles. The complex handcrafted model [2] includes derived predicates and axioms, so to create an extended model incorporating them, we merged our branch with the one developed by Speck, which handles these features. The 39 instances used for our experiments were taken from the Puzznic benchmark [2].

Overall analysis: In 15-Puzzle, Pancake, and Plotting, our models solved more instances and achieved better or comparable plan quality, especially in the Anytime and Optimal modes. The logarithmic encoding was particularly effective in 15-Puzzle, allowing SymK to outperform handcrafted baselines. In Sokoban and Rush Hour, performance was similar across variants, showing our approach generalises well. Rush Hour's simplicity, however, limits performance differentiation. Puzznic was the only domain where our compilations underperformed, with handcrafted SymK consistently superior across modes.

7 Conclusions & Future Work

Our work is motivated by the need to improve modelling facilities for AI Planning, particularly for common structures like grids, which are hard to model in PDDL when index-based operations are involved. We have introduced a set of extensions to the Unified Planning library that support high-level modelling features, enabling natural formulations across a range of domains with array-like structures. These features allow automatic exploration of multiple low-level reformulations from a single high-level model. While each of the rewrites in isolation could be manually implemented by a user, combining various transformations manually is laborious and error-prone. Furthermore, as planners generally support different subsets of features, users would typically need to create specific models for each planner they wished to test. Our experiments show that the compiled models are competitive with, and often outperform, handcrafted alternatives. Future work includes exploring new compilation pathways, automating planner and rewrite selection, and identifying additional useful modelling patterns.

References

1 Mojtaba Elahi and Jussi Rintanen. Planning with complex data types in PDDL. *CoRR*, abs/2212.14462, 2022.

2 Joan Espasa, Ian P. Gent, Ian Miguel, Peter Nightingale, András Z. Salamon, and Mateu Villaret. Cross-paradigm modelling: A study of puzznic. In *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 89–95, 2024. doi:10.1109/ICTAI62512.2024.00021.

3 Joan Espasa, Ian Miguel, Peter Nightingale, András Z. Salamon, and Mateu Villaret. Plotting: a case study in lifted planning with constraints. *Constraints An Int. J.*, 29(1-2):40–79, 2024. URL: https://doi.org/10.1007/s10601-024-09370-x, doi:10.1007/S10601-024-09370-X.

4 Michael Fogleman. Solving rush hour, the puzzle. July 2018. URL: https://www.michaelfogleman.com/rush/.

5 Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124, 2003. URL: https://doi.org/10.1613/jair.1129, doi:10.1613/JAIR.1129.

6 William H. Gates and Christos H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27(1):47–57, 1979. doi:10.1016/0012-365X(79)90068-2.

7 Hector Geffner. Functional strips: A more flexible language for planning and problem solving. In *Logic-Based Artificial Intelligence*, volume 597 of *The Springer International Series in Engineering and Computer Science*. Springer, 2000.

8 Hector Geffner. PDDL 2.1: Representation vs. computation. *J. Artif. Intell. Res.*, 20:139–144, 2003. URL: https://doi.org/10.1613/jair.1995, doi:10.1613/JAIR.1995.

9 Alfonso Emilio Gerevini. An introduction to the planning domain definition language (PDDL): book review. *Artif. Intell.*, 280:103221, 2020.

10 GitHub. Rush Hour - PDDL. March 2019. URL: https://github.com/ehajdini/AI/blob/master/RushHour_PDDL/domain1.pddl.

11 Peter Gregory, Derek Long, Maria Fox, and J. Christopher Beck. Planning modulo theories: Extending the planning paradigm. In *22nd ICAPS*, 2012.

12 Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006. URL: https://doi.org/10.1613/jair.1705, doi:10.1613/JAIR.1705.

13 ICAPS 2015. 15-puzzle domain from the ICAPS 2015 Benchmark Suite, 2015. URL: https://github.com/icaps15-CAP/sigaps-domains/tree/master/15puzzle.

14 Andrea Micheli, Arthur Bit-Monnot, Gabriele Röger, Enrico Scala, Alessandro Valentini, Luca Framba, Alberto Rovetta, Alessandro Trapasso, Luigi Bonassi, Alfonso Emilio Gerevini, Luca Iocchi, Felix Ingrand, Uwe Köckemann, Fabio Patrizi, Alessandro Saetti, Ivan Serina, and Sebastian Stock. Unified planning: Modeling, manipulating and solving ai planning problems in python. *SoftwareX*, 29:102012, 2025. doi:10.1016/j.softx.2024.102012.

15 Potassco Team. Sokoban domain from the IPC 2011 Benchmark Suite, 2011. URL: https://github.com/potassco/pddl-instances/blob/master/ipc-2011/domains/sokoban-sequential-satisficing/instances/instance-1.pddl.

16 Jussi Rintanen. Impact of modeling languages on the theory and practice in planning research. In *29th AAAI*, pages 4052–4056, 2015.

17 Enrico Scala, Patrik Haslum, and Sylvie Thiébaux. Heuristics for numeric planning via subgoaling. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 3228–3234. IJCAI/AAAI Press, 2016. URL: http://www.ijcai.org/Abstract/16/457.

18 David E Smith, Jeremy Frank, and William Cushing. The anml language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, volume 31, 2008.

19 David Speck, Robert Mattmüller, and Bernhard Nebel. Symbolic top-k planning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI*, pages 9967–9974. AAAI Press, 2020. URL: https://doi.org/10.1609/aaai.v34i06.6552, doi:10.1609/AAAI.V34I06.6552.