Towards High-Level Modelling in Automated 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. The planning task 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 model 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 and enhancing the overall capability of automated planning systems. Unified-Planning is a Python library offering high-level API to specify planning problems and to invoke automated planners. In this paper, we present an extension of the UP library aimed at enhancing its expressivity for high-level problem modelling. In particular, we have added an array type, an expression to count booleans, and the allowance for integer parameters in actions. We show how these facilities enable natural high-level models of three classical planning problems.

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

Keywords and phrases Automated Planning, Reformulation, Modelling

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Planning is a fundamental activity, impacting our daily lives in many and varied ways. The planning task consists of selecting a sequence of actions to achieve a specified goal from specified initial conditions. This type of problem arises frequently in many contexts, from daily tasks to industrial processes. Examples include scheduling deliveries, organising project schedules, improving manufacturing efficiency, optimising resource allocation, and coordinating transportation routes. Automatically solving planning problems is a central discipline of Artificial Intelligence that involves specifying the desired outcome (the 'what') in a purely declarative manner, leaving it to the planning engine to determine the sequence of actions (the 'how') needed to reach that outcome.

Consider a scenario where a delivery robot operates within an environment that can be represented as a grid of cells, each represented as distinct locations that the robot can occupy. The robot's objective is to transport a package from one position to another. This involves considering various states such as the robot's presence in a cell, the package's location, and whether the robot is holding the package. The planning task requires defining a set of possible actions: the robot can move between cells, pick up and drop off packages. In the initial state, both the robot and package are located in specific cells, and the goal is to find a sequence of actions where the robot successfully delivers the package to the desired cell.

In such scenarios, it is natural to represent the grid as a matrix. However, traditional planning frameworks do not support these grid-like structures directly, which poses challenges when modelling the problem. This motivates our work to develop new planning constructs and extensions that can effectively model problems in such structured environments.

The difficulty of solving planning problems grows rapidly with their size in terms of the number of states and possible actions considered. Over many years, a great deal of effort by a number of different research groups has resulted in the development of highly efficient AI planners [26]. These planners serve as the computational engines that apply problem-solving algorithms to generate optimal or satisfactory plans given specific problem domains. However, it is beneficial to consider automated modelling as much a part of the process of solving a planning problem as the search for a solution. The choice of a model has a significant effect on the performance of state-of-the-art AI planning systems [8, 6], similarly to the importance of modelling to Constraint Programming. Constraint Programming has been successfully used to solve planning problems [4, 5] and is particularly well suited to planning problems when the problem requires a certain level of expressivity, such as temporal reasoning or optimality [27, 3].

The Planning Domain Definition Language (PDDL) [17] is the leading language used in the field of automated planning to model planning problems and the domains in which they occur. It provides a formal way to concisely describe the problem in terms of objects, predicates, actions and functions with parameters. PDDL was created in an effort to standardise the input for AI planners, facilitating the solving of planning problems.

Previous work [22] highlighted the limitations of PDDL, particularly in terms of its expressivity, prompting the development of abstraction techniques to extract higher level concepts from PDDL models. Our interest lies in facilitating the handling of complex problems and enhancing the overall capability of automated planning systems. The incorporation of high-level concepts, akin to those available in Essence [14] via Conjure [1] in Constraint Programming, such as functions, relations, arrays, (multi)sets, and sequences, can significantly enrich the modelling scope and flexibility.

Unified-Planning (UP) [28] is a Python library offering high level API to specify planning problems and to invoke planning engines. This open-source library has gained substantial recognition within the research community due to its extensive adoption and continuous development, making it an excellent candidate on which to base our research efforts, offering ample opportunity for exploration and experimentation.

In this paper, we present an extension of the UP library aimed at enhancing its expressiveness for high-level problem modelling. The new high-level implementations we are introducing include:

1. A new UP type: Array
2. A new UP expression: Count
3. Support for Integers as Parameters in Actions

Furthermore, we have developed three new UP compilers, each dedicated to removing one of the high-level concepts we've implemented. These compilers automate the translation of these constructs into simpler ones, employing advanced problem transformations similar to those used for other features within the library.

In the experimental part of this work, we model three classical planning problems — Plotting [11], Rush Hour [12], 8-Puzzle [21] — to demonstrate the effectiveness of our extended framework. These problems are particularly suitable for array-based modelling due to their grid-based nature, which aligns well with our new array type implementation.

2 Background and Related Work

A classical planning problem is typically formalised as a tuple $\sqcap = \langle F, A, I, G \rangle$, where F is a set of propositional state variables, A is a set of actions, I is the initial state, and G is the goal. A *state* is a variable-assignment (or valuation) function over state variables F, which maps each variable of F into a truth value. An action $a \in A$ is defined as a tuple $a = \langle \text{Pre}_a, \text{Eff}_a \rangle$, where Pre_a refers to the preconditions and Eff_a to the effects of the action. Preconditions (Pre) and the goal G are first-order formulas over propositional state variables. Action effects (Eff) are sets of assignments to propositional state variables. An action a is applicable in a state s only if its precondition is satisfied in s ($s \models \text{Pre}_a$). The outcome after the application of an action a will be the state where variables that are assigned in Eff_a take their new value, and variables not referenced in Eff_a keep their current values. A sequence of actions $\langle a_0, \ldots, a_{n-1} \rangle$ is called a plan. We say that the application of a plan starting from the initial state I brings the system to a state s_n. If each action is applicable in the state resulting from the application of the previous action and the final state satisfies the goal (i.e., $s_n \models G$), the sequence of actions is a *valid plan*. A planning problem has a solution if a valid plan can be found for the problem.

A formalisation for the previously mentioned delivery robot scenario, set within a 2x2 grid with positions named *P00*, *P01*, *P10*, and *P11*, could be as follows:

$$
\begin{aligned}
&\sqcap = \langle \\
&\quad F = \{\text{at\_robot}(p), \text{at\_package}(p), \text{holding\_package}\}, \\
&\quad A = \{ \\
&\qquad \text{Move}(p1, p2) = \langle \text{at\_robot}(p1), \{\text{at\_robot}(p2), \neg\text{at\_robot}(p1)\} \rangle, \\
&\qquad \text{PickUp}(p) = \langle \text{at\_robot}(p) \wedge \text{at\_package}(p), \{\text{holding\_package}, \neg\text{at\_package}(p)\} \rangle, \\
&\qquad \text{DropOff}(p) = \langle \text{at\_robot}(p) \wedge \text{holding\_package}, \{\text{at\_package}(p), \neg\text{holding\_package}\} \rangle, \\
&\quad \}, \\
&\quad I = \{\text{at\_robot}(P00), \text{at\_package}(P10), \neg\text{holding\_package}\}, \\
&\quad G = \{\text{at\_package}(P11)\} \\
&\rangle
\end{aligned}
$$

Is important to note that some of these propositional variables F are actually first-order atoms over unquantified variables. For instance, the atom *at_robot(p)* is not a propositional variable by itself. Here, p is a parameter that represents a position in the grid and can be substituted with specific grid locations such as *P00, P01, P10 and P11*. During the grounding process, this predicate generates multiple propositional variables corresponding to each specific position in the grid: *at_robot_P00, at_robot_P01, at_robot_P10 and at_robot_P11*. The actions A define the tasks the robot can perform, such as *Move(p1, p2)*, which allows the robot to move from position *p1* to position *p2* if it is currently in position *p1*. So when the action is executed, the robot moves to position *p2* and it is no longer in position *p1*. The initial state I is the in *P00*, the package is in *P10*, and the robot not holding the package. The goal state G is to have the package in *P11*.

The Planning Domain Definition Language (PDDL) [17] initially supported only Boolean types. Over the years, it has evolved significantly to include support for features such as numeric types, temporal constraints, hierarchical types, durative actions, derived predicates and conditional effects. This evolution has greatly enhanced the expressiveness and flexibility of PDDL, allowing for more sophisticated and detailed modelling of planning problems.

Despite these advancements, there remains a significant lack of expressiveness in PDDL, particularly when it comes to representing more complex planning scenarios [22].

The UP library simplifies the processes of both formulating planning problems and utilising automated planners. The library allows users to define problems in a simple and intuitive manner and solve them using any of the wide array of supported solvers. Additionally, it provides functionality for exporting and importing problems in PDDL or ANML [24] format, and executing advanced problem transformations such as simplification, grounding, and removal of conditional effects.

In the field of automated planning, various approaches have been explored to enhance planning modelling methodologies. This section offers an overview of recent advancements in planning techniques and related works that have inspired or guided our research.

Geffner's Functional STRIPS [16] introduced an extension of the STRIPS planning language by incorporating first-class function symbols. This addition allowed for greater flexibility in representing planning problems, enabling more efficient encodings and supporting complex tasks with minimal action definitions. Recent work by Geffner and Frances [13] explores how to address the computational challenges posed when solving problems expressed in Functional STRIPS by using a Constraint Satisfaction Problem to compute an heuristic that guides the search.

Planning Modulo Theories (PMT) [18], inspired by SAT Modulo Theories (SMT), offers a flexible modelling language and framework where arbitrary first-order theories can be treated as parameters. Although further work has been done in the context of PMT [7], no concrete implementations have been released.

Elahi and Rintanen [10] proposed a modelling language supporting complex data types like Booleans, numeric types, enumerated types, records, unions, arrays, sets, and lists, which are reduced to a Boolean representation. This Boolean representation is further reduced to PDDL, allowing existing domain-independent planners to solve problems specified in the richer modelling language. While this approach effectively enhances PDDL's expressivity through the use of complex data types, our work aims to extend these capabilities further by leveraging the UP library. By directly integrating complex data types into UP, we benefit from Python's simplicity and readability, along with its extensive libraries and community support. This results in intuitive and concise models that are easier to understand and manipulate. Inspired by Essence and Conjure, our method aims to introduce similar high-level modelling expressivity to automated planning, providing a more intuitive and fluent framework.

3 Pipeline

In this section, we provide an overview of the pipeline within the Unified-Planning framework. The framework is designed to streamline the process of transforming planning problems into formats that various planners can understand and solve. The different stages of this pipeline are depicted in Figure 1.

The first step in defining a planning problem is to create a new instance that serves as a container for all the elements that constitute the problem: the fluents, actions, objects, initial state, and goals. Similarly to PDDL, objects typically represent entities in the problem domain, each with a type. Note that the term *fluent* has been historically used to refer to state variables that may change over time. UP also uses a lifted representation of the problem, with state variables and actions having parameters, enabling a concise definition of the problem.

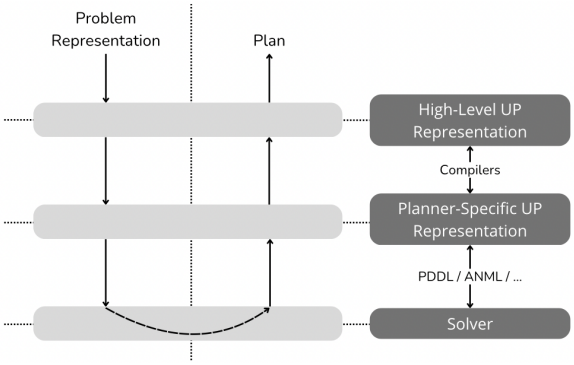


Figure 1 Automated Planning Modelling Pipeline

We further distinguish between two representation levels to clearly delimitate when compilers need to be used on elements of the problem that might need to be transformed to ensure compatibility with the planners. It's important to note that if the original problem does not contain any high-level features or the chosen planner fully supports all the problem's features, compilers may not be necessary.

High-Level UP Representation Initially, the planning problem can be defined at a high-level, including complex features such as conditional effects, quantifiers, or 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 types, count expressions, integer parameters in actions — as high-level representations.

Planner-Specific UP Representation After the potential application of compilers, the problem is expressed in a simplified format. This version retains the semantics of the original problem instance while transforming any unsupported features by the targeted planner.

Specific Planning Languages Representation

Unified-Planning offers the ability to transform the problem representation into various specific planning languages such as PDDL and ANML. Whether the problem is translated to PDDL, ANML, or another specific planning language before passing to the solver, depends on the requirements of the chosen planner. Some planners necessitate translating the problem into a specific format before they can process it, essentially when they are designed to exclusively work with these languages. Alternatively, if the selected planner can directly understand the UP representation of the problem, this can straightforwardly interface with the planner without an additional compilation step.

Solver

Finally, the converted planning problem is fed into a planner for execution. UP provides access to a variety of planners, each with distinct capabilities, which can be utilised to solve different types of planning problems. 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

4.1 Array Type

The new Array Type class is designed to represent arrays consisting of a specified number of elements of a given type. It relies on two main parameters: `size` and `elements_type`. The `size` parameter signifies the number of elements contained within the array and must be an integer (Python class int) with a predefined value greater than one. On the other hand, `elements_type` represents the type of the elements within the array. It is optional and it defaults to None, meaning no specific type is assigned. In this case, the array will be assumed to be of Boolean type. The construction of the class is shown in Listing 1.

Listing 1 Construction of the Array Type.

```
def ArrayType(size: int, elements_type: Type=None) -> unified_planning.model.types.Type
```

This type empowers us to represent tables or matrices effectively. With this implementation, we can now define *ArrayType Fluents*, giving us the ability to access the array's elements individually, treating each as a fluent, while also knowing the position of each within the array. Moreover, given that arrays are considered types themselves and the `elements_type` parameter represents a type, we can create arrays of arrays and so forth, enabling the creation of nested arrays.

Assume the initial example of a robot that operates in a 3x3 grid, where the robot can only move to adjacent cells —left, right, up, or down—, but not diagonally. Without using arrays, it is necessary to define all the relationships between the different cells to determine if they are neighbours. However, by utilising arrays, their position and relationship becomes implicit. For example, we can define the grid cells using a double array-type fluent (matrix) that encapsulates Booleans, indicating whether the robot is in each cell of the grid. The same approach can be used for the package. Listing 2 shows the definition of the initial state and fluents. Note that all positions that are not specified in `initial_values` default to false.

Array Type Compiler

Given that this new implementation is not compatible with the planners, we've developed an *ArraysRemover* compiler to transform arrays into individual elements. For each array-type fluent, we create a series of fluents that correspond to its individual elements. These new fluents maintain the original name but are suffixed with '_' followed by the respective position indices of the array elements. For example, the i-th element of the array my_ints, accessed using $my\_ints[i]$, transforms into a new fluent named my_ints_i. Moreover, this methodology extends to multidimensional arrays. For each dimension, an index is added to represent all the elements. Considering the fluent `at_robot` of the previous example, the elements are separated into the nine new fluents with the format depicted in Listing 3. Furthermore, in all actions and goals, we not only replace array accesses, as previously illustrated, but also manage the entire array comprehensively. For instance, when encountering an expression such as `at_robot[0]` $\iff$ `[False, False, False]`, we decompose it into: `at_robot_0_0` $\iff$ `False` $\wedge$ `at_robot_0_1` $\iff$ `False` $\wedge$ `at_robot_0_2` $\iff$ `False`.

Undefinedness

Introducing arrays poses a significant issue when accessing positions outside the arrays' defined range. The concept of undefinedness in planning is not extensively addressed because

Listing 2 UP representation of `at_robot` fluent using Array-Type.

```
fluents = [
  array[3, array[3, bool]] at_robot
]
initial fluents default = [
  array[3, array[3, bool]] at_robot := [
    [false, false, false],
    [false, false, false],
    [false, false, false]
  ]
]
initial values = [
  at_robot[0][0] := true
]
```

Listing 3 UP representation of derived fluents from `at_robot` after applying the Array Type Compiler.

```
fluents = [
  bool at_robot_0_0
  bool at_robot_0_1
  bool at_robot_0_2
  ...
  bool at_robot_2_2
]
initial fluents default = [
  bool at_robot_0_0 := false
  bool at_robot_0_1 := false
  bool at_robot_0_2 := false
  ...
  bool at_robot_2_2 := false
]
initial values = [
  at_robot_0_0 := true
]
```

Listing 4 UP representation of the `move_right` action using Integer-Type Parameters.

```
action move_right(integer[0, 2] r, integer[0, 1] c) {
  preconditions = [
    at_robot[r][c]
  ]
  effects = [
    at_robot[r][(c + 1)] := true
    at_robot[r][c] := false
  ]
}
```

traditional planning models typically require well-defined states and actions. In our new implementation, we have been inspired by how undefinedness is handled in constraint programming, based on a study proposing three approaches [15]. Two of these methods involve introducing a new truth value, which we find impractical for the library due to substantial modification overhead. The third approach transforms expressions containing undefined values to False. For example, in an array-type fluent my_ints containing three integers, accessing $my\_ints[3]$ (where the array has elements only at indices 0, 1, and 2) would evaluate the expression $(my\_ints[3] == 2)$ as $False$.

Our solution is a hybrid approach comprising two modes: restrictive and permissive.

Restrictive encountering an out-of-range access triggers an error, halting the program with a message indicating the undefined element.

Permissive handles out-of-range accesses differently depending on whether they occur in preconditions or effects. In preconditions, akin to constraint handling, if the fluent is a Boolean element, it evaluates itself to False. But, if the fluent is not Boolean, the simplest Boolean expression evaluates to False. In effects, an attempt to access out-of-range simply removes that action, notifying the user via a message.

We illustrate in Listings 5 and 6 two different examples to demonstrate how the Permissive approach manages various scenarios within the preconditions.

Listing 5 UP representation simplifying expressions for Boolean undefined fluents in permissive mode.

```
And(at_robot[2][2],at_robot[2][3])
And(at_robot[2][2],False)
False
```

Listing 6 UP representation simplifying expressions for non-Boolean undefined fluents in permissive mode.

```
Or(Equals(Plus(my_ints[3],2),2),Equals(my_ints[2],0))
Or(False,Equals(my_ints[2],0)
Equals(my_ints[2],0)
```

4.2 Integer Type Parameters in Actions

We identified support for bounded integer parameters to be passed to actions as a highly useful implementation when working with arrays.

We will illustrate this approach with the previous example. To properly control the possible movements, we create four actions, each corresponding to one direction. For instance, with the `move_right` action, shown in Listing 4, we ensure that the robot cannot move outside the grid by allowing this movement only when the robot is in one of the first two columns. This means that only the integers 0 and 1 will be passed as parameters for the columns, preventing the robot from moving right when it is in the rightmost column.

This approach allows for more flexible and concise problem descriptions in planning scenarios. It enables us to refer to any element within the array by indexing this integer in the array, eliminating the need to create an action for each individual element. However, it is crucial to ensure that the bounded integer values fall within the specified domain range, as values outside this range may lead to undefined behaviour. It is crucial to understand how undefined values will be handled according to the strategies outlined above, ensuring the implementation appropriately manages any out-of-range values introduced. Additionally, integers can be utilized not only for indexing arrays but also as values in preconditions or effects, as well as in arithmetic operations, enhancing the model's flexibility and expressiveness.

Integer Type Parameters in Actions Compiler

This compiler overcomes the limitations of the previous implementation, as planners could not comprehend or process integer parameters or anything other than objects in actions. For each action, the compiler generates new actions for each possible combination of the integer parameters. These new actions do not include the integer parameters, as they are replaced by their respective values within the preconditions and effects. Retaining the original name, they are suffixed with '_' followed by the current integer parameter values separated by '_'.

For instance, when the compiler processes the `move_right` action described in Listing 4, it will transform it into six new actions, one for each possible combination of the parameters `r` and `c` — $(0,0),(0,1),(1,0),(1,1),(2,0),(2,1)$ —. In Listing 7, the action generated for the parameters (0,1) illustrates how the parameters `r` and `c` in both preconditions and effects are substituted with their specific values. Furthermore, this compiler simplifies expressions wherever possible during parameter substitution with integer values. This simplifier handles arithmetic operations at runtime, reducing the size and complexity of the generated expressions. For example, in the previous instance with `at_robot[0][2]`, the value 2 results from the addition $(1+1)$, which is computed automatically during compilation. This optimisation streamlines execution, enhances efficiency, and simplifies the problem description for the planner.

Listing 7 UP representation of `move_right(r=0, c=1)` action after applying the Integer-Type Parameters Compiler.

```
action move_right_0_1 {
  preconditions = [
    at_robot[0][1]
  ]
  effects = [
    at_robot[0][2] := true
    at_robot[0][1] := false
  ]
}
```

Listing 8 Construction of the Count Expression.

```
def Count(self,
  *args: Union[BoolExpression, Iterable[BoolExpression]]
) -> unified_planning.model.fnode.FNode
```

Listing 9 UP representation of a goal ensuring only one room is occupied by robots using the Count Expression.

```
goals = [
  (Count(at_robot[0][0], at_robot[0][1], at_robot[0][2],
    at_robot[1][0], at_robot[1][1], at_robot[1][2],
    at_robot[2][0], at_robot[2][1], at_robot[2][2]) == 1)
]
```

4.3 Count Expression

When encoding problems, we encounter situations where evaluating the number of True statements among multiple Boolean expressions becomes necessary. To address this, we have developed the `Count` expression, specifically designed to manage Boolean n-arguments efficiently. The purpose of this function is to return an integer representing the number of True expressions among multiple Boolean expressions. The construction of this expression is shown in Listing 8, and can be formulated in two ways: $Count(a, b, c)$ or $Count([a, b, c])$, where a, b, c represent Boolean arguments.

We can exemplify the function with an illustration of a possible goal from the previous example. Suppose our problem involves multiple robots, and the goal is for all robots to end up in the same room, leaving only one room occupied. To achieve this, we need to count how many cells are occupied by any robot. This is demonstrated in Listing 9.

Count Expression Compiler

This compiler translates each *Count* expression into a new set of expressions that the planner can comprehend, involving the creation of a new function (known as an integer fluent in Unified-Planning) for each argument of every *Count* expression in the problem. These functions are designed to represent the Boolean value of each argument: they take on a value of 0 if the expression is False and 1 if it is True. The new fluents will be named sequentially as $count\_0, count\_1, count\_2$, and so on, for each argument of the expressions in the problem. When different *Count* expressions share an argument with the same expression, they will be assigned the same name. Moreover, substitutes each argument in the expression with its corresponding function, and replaces the *Count* operator expression with the well-known *Plus* operator, which adds up various integer expressions. This way, we sum up the new functions created, each corresponding to its original Boolean expression. This is shown in Listing 10. These functions are initialised depending on the initial value of the fluents and the evaluation of the expression. As depicted in Listing 12, in our example, only the position $(0, 0)$ is set to True, resulting in the function related to this Boolean expression, `count_0`, having an initial value of 1.

Furthermore, for each action, if any fluent is modified, those expressions (arguments from Count expressions) containing that fluent will also be evaluated to potentially change the value of the corresponding function based on the evaluation result. As illustrated in Listing 10, the Integer Type Parameters in Actions compiler and the Array Type Compiler have already been applied. It must be in this order, as discussed in Section 4.4. The

Listing 10 UP representation of the previous goal after applying the Count compiler.

```
goals = [
  (Plus(count_0, count_1, count_2, count_3, count_4,
        count_5, count_6, count_7, count_8) == 1)
]
```

Listing 11 UP representation of `move_right(r=0, c=0)` action effects after applying the Count compiler.

```
action move_right_0_0 {
  preconditions = [
      at_robot_0_0
  ]
  effects = [
    at_robot_0_1 := true
    at_robot_0_0 := false
    count_0 := 0
    count_1 := 1
  ]
}
```

Listing 12 UP representation of `at_robot` initial values after applying the Count compiler.

```
initial values = [
  at_robot_0_0 := true
  at_robot_0_1 := false
  ...
  at_robot_2_2 := false
  count_0 := 1
  count_1 := 0
  count_2 := 0
  count_3 := 0
  count_4 := 0
  count_5 := 0
  count_6 := 0
  count_7 := 0
  count_8 := 0
]
```

action `move_right_0_0` modifies both `at_robot_0_0` and `at_robot_0_1`. Consequently, the corresponding functions, `count_0` and `count_1`, are re-evaluated to their new values: 0 if the Boolean evaluates to False and 1 if it evaluates to True. If the effect is conditional, the function's effect will adhere to the same condition. Moreover, if the effect's fluent has parameters, a condition will be added in this new effect, checking whether these parameters match those of the same fluent in the *Count* argument.

4.4 Sequence of Compiler Application

To ensure the proper utilisation of the compilers we've developed, it is crucial to follow a specific sequence in their application. The required order, assuming all three implementations are included in a problem, is as follows:

1. Integer Parameters in Actions Compiler
2. Array Type Compiler
3. Count Expression Compiler

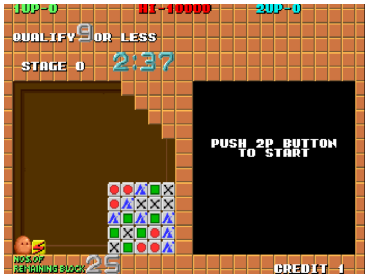
When a problem includes integer parameters in actions it is crucial to apply the *Integer Parameters in Actions Compiler* before any other compiler. Not applying this compiler first could lead to issues if our problem also involves arrays. For example, accessing specific elements like `my_ints[i]` where `i` is an integer parameter requires knowing its value to determine the array element. Applying the *Array Type Compiler* first without this information would result in errors due to the undefined value of `i`.

The *Count Expression Compiler* should be applied after resolving complexities related to arrays using the other two compilers. This sequence is critical because the expressions it handles may reference array elements. Additionally, to effectively adjust the new functions associated with each argument of count expressions, it is essential to first evaluate the changes affecting the fluents within the action effects. Without substituting integers and removing arrays beforehand, we would not know which fluents are affected by action changes.

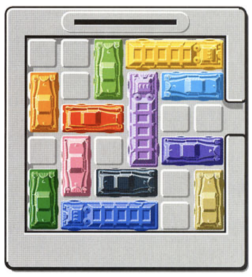
Consequently, we would be unable to implement the required adjustments to our functions when evaluating expressions with the updated values.

5 Experiments

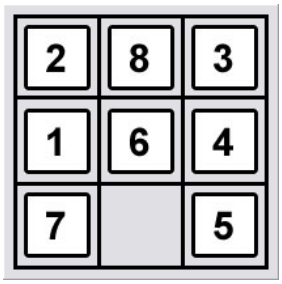
In this section, we evaluate our new UP-extended implementation encoding three typical planning games: Plotting, Rush-Hour, and 8-Puzzle, against existing PDDL models. These experiments were conducted using a cluster comprising 20 nodes, each equipped Intel(R) Xeon(R) E-2234 CPUs @ 3.60 GHz with 16GB of RAM. The solvers used in our experiments are *Enhsp-opt* [23] for our UP-extended model and *Fast-Downward* [19] with the *seq-opt-lmcut* heuristic for the selected PDDL models. We employ two different planners to leverage their respective strengths: *Enhsp* (version 20) supports numeric extensions crucial for our model, while *Fast-Downward* (version 23.06+), although lacking numeric support, provides robust heuristic-based search capabilities. Our aim was to measure the combined preprocessing and solving time, with a configured timeout of 1 hour (all results are presented in seconds), and see if the proposed UP extensions had a reasonable cost regarding solving time. For each problem we will highlight the advantages in modelling provided by some of the features that we have developed in UP. The models with UP-extended, along with the implementation of the proposed extensions, can be found on GitHub at *https://github.com/stacs-cp/unified-planning/tree/new_types2*.



(a) Example of a 5x5 grid Initial Plotting Instance



(b) Example of an Initial Rush Hour Instance



(c) Example of an Initial 8-Puzzle Instance

Figure 2 Examples of different instances of Plotting, Rush-Hour and 8-Puzzle games.

5.1 Plotting

Plotting is a tile-matching puzzle video game published by Taito (Figure 2a illustrates an instance of the game). The objective of the game is to remove at least a certain number of coloured blocks from a grid by sequentially shooting blocks into the same grid. The interest and difficulty of Plotting is due to the complex transitions after every shot: various blocks are affected directly, while others can be indirectly affected by gravity.

Listing 13 depicts the grid represented as a double array, with each cell indicating the color of a block. The initial grid configuration is defined using a Python nested list.

We define several actions that manipulate the grid of colored blocks based on the shot. These actions utilise integer parameters to reference different blocks, and we ensure these parameters are within valid ranges to prevent out-of-bounds access. One such action, detailed in Listing 14, is `shoot_partial_row`, which clears blocks of a colour `p` from a row `r` up to the last column `l` of that row, stopping when the next block is of a different color.

Listing 13 UP extended: Defining the Plotting game problem instance using Array Type.

```
grid = [[R,R,B,G,Y],[R,B,Y,Y,Y],[B,G,B,G,B],[G,Y,G,R,B],[Y,G,R,R,B]]
blocks = Fluent('blocks', ArrayType(rows, ArrayType(columns, Colour)))
plotting_problem.add_fluent(blocks)
plotting_problem.set_initial_value(blocks, grid)
```

Listing 14 UP extended: Defining the Shoot-Full-Row action.

```
spr = unified_planning.model.InstantaneousAction('shoot_partial_row', p=Colour,
    r=IntType(0, rows-1), l=IntType(0, columns-2))
spr.add_precondition(Not(Or(Equals(p, W), Equals(p, N))))
for c in range(0, columns-1):
  spr.add_precondition(Or(GT(c,l), Equals(blocks[r][c], p), Equals(blocks[r][c], N)))
spr.add_precondition(Or(
    *[And(Equals(blocks[r][c], p), LE(c,l)) for c in range(columns-1)]
))
spr.add_effect(hand, blocks[r][l+1])
spr.add_effect(blocks[r][l+1], p)
for c in range(0, columns-1):
  spr.add_effect(blocks[0][c], N, LE(c,l))
  for a in range(1, rows):
    spr.add_effect(blocks[a][c], blocks[a-1][c], And(LE(c,l), LE(a,r)))
```

In Listing 14, we highlight some of the most interesting preconditions and effects of this action, particularly to demonstrate the effectiveness of Python functions, such as *for loops*, for iterating over different elements. The first precondition ensures that the next block of the column `l` is different from `p` and not `N`, where `N` indicates no block is present. The first loop ensures that all elements in the row until the column l are either `p` or `N`. And the following precondition confirms that at least one of these elements (i.e., among all elements being cleared) is `p`, preventing unnecessary actions when all blocks are `N`. The effects described detail how the action modifies the grid. Initially, the `hand` takes on the value of the next block in sequence (the first block different), while this position saves the value of `p`. Additionally, the nested loops update the grid's elements in accordance with gravity, ensuring the correct movement of blocks above the affected row.

As shown in Listing 15, the objective of the game is to have no more than a specified number of blocks remaining, in this example, 4 or fewer. Our new implementation of the *Count* expression is particularly useful for this purpose as it facilitates the counting of blocks that are different from `N`, indicating how many blocks are left on the grid.

Listing 15 UP extended: Defining the Goal.

```
remaining = [Not(Equals(blocks[i][j], N)) for i in range(rows) for j in range(columns)]
plotting_problem.add_goal(LE(Count(remaining), 4))
```

For the experimental part, we compared our UP-extended model with the PDDL extracted from the paper 'Challenges in Modelling and Solving Plotting with PDDL' by J. Espasa et al. [2]. We utilised the instances from the database of the same work, which consists of 522 instances. In Figure 3, we show results for several instances based on grid size, number of colours, and number of remaining blocks. Our model successfully solves 216 instances out of the total, whereas the PDDL model solves only 78 instances. Our high-level implementations has resulted in a more concise, understandable, and manageable model in comparison to the

Grid	Rows	Columns	Colours	Remaining Blocks	UP-extended	PDDL
RRRG,RGGG	2	4	2	1	2.362	Timeout
RRR,GGR,RRR	3	3	2	1	4.651	89.270
RGG,BBG,GBR	3	3	3	2	4.604	1441.097
RRRG,RGRR,RGRG	3	4	2	1	40.254	Timeout
RGBB,GBRB,BGGR	3	4	3	2	40.779	1336.587
RGBBG,GGBRB,BGRBR	3	5	3	2	2057.253	Timeout
RGGBY,YYYRB,YBYBY	3	5	4	3	2050.072	Timeout
RG,GR,RG,GG	4	2	2	1	3.368	Timeout
RGR,GGG,RRR,RGR	4	3	2	1	48.402	Timeout
RRG,BGR,GGR,RRG	4	3	3	2	46.706	1748.482
RGB,RBB,BBR,GGR,BGR	5	3	3	2	2765.453	Timeout
RGR,RGG,BYB,GYY,YGY	5	3	4	3	2777.459	Timeout

Figure 3 Comparison of Plotting models: UP-extended vs PDDL.

one of [2] where the authors needed to simulate cell positions by ad-hoc encoding numbers and relational predicates (as $\leq$) into PDDL.

5.2 Rush Hour

Rush Hour is a sliding block puzzle game set on a 6x6 grid, where blocks represent vehicles stuck in a traffic jam (Figure 2b illustrates an instance of the game). The objective is to move a special vehicle, the red car, to the exit located at the right edge of the grid. However, the movement of vehicles is restricted: they can only move forwards or backwards in a straight line and cannot cross over each other.

In modelling the Rush Hour game, we utilise a 6x6 grid represented by a double array, where each cell denotes a letter representing a vehicle, as depicted in Listing 16. This grid instantiation process utilises Michael Fogleman's database [12], which provides string representations where 'o' denotes an empty cell, and each letter represents a distinct vehicle. Vehicles include cars, which occupy 2 cells, and trucks, which occupy 3 cells. Python's ability to manage strings as iterable sequences allows for a concise representation of the grid and enhances flexibility in modifying its configuration.

Listing 16 Defining the Rush Hour game problem instance using Array Type.

```
occupied = Fluent('occupied', ArrayType(6, ArrayType(6, Vehicle)))
rush_hour_problem.add_fluent(occupied)
grid = 'GBBoLoGHIoLMGHIAAMCCCKoMooJKDDEEJFFo'
for i, char in enumerate(grid):
    r, c = divmod(i, columns)
    if char == '.':
        rush_hour_problem.set_initial_value(occupied[r][c], none)
    else:
        obj = Object(f'{char}', Vehicle)
        if not rush_hour_problem.has_object(char):
            rush_hour_problem.add_object(obj)
            rush_hour_problem.set_initial_value(is_car(obj), grid.count(char) == 2)
        rush_hour_problem.set_initial_value(occupied[r][c], obj)
```

Utilising the permissive mode when handling undefinedness (See Undefinedness above) is highly beneficial in encoding the Rush Hour problem. It eliminates the requirement to specify exact movement possibilities for each vehicle position while ensuring they remain

Listing 17 Defining the Move-Horizontal-Car action.

```
mhc = unified_planning.model.InstantaneousAction('move_horizontal_car', v=Vehicle,
    r=IntType(0,rows-1), c=IntType(0,columns-2), m=IntType(-(columns-2), columns-2))
```

Instance	Reachable States	Steps	Moves	UP-extended	PDDL
BBBKLMHCCKLMHoAALMDDJooooIJEEooIFFGG	24132	92	49	70.446	3.659
BBBKLMHCCKLMHoAALoDDJooooIJEEooIFFGG	37740	89	48	100.161	4.862
HoBBBMHCCJLMAAIJLoDDIKLooooKEEoFFGGo	16930	77	43	68.247	3.339
oBBBKMCCoIKMAAoILNGDDJLNGoHJEEFFHooo	23979	70	41	81.336	2.763
HBBBoMHCCJoMAAIJLMDDIKLooooKEEFFGGoo	20592	73	40	77.392	3.246
BBBJKLGCCJKLGoAAKoDDIooooHIooooHEEFF	28257	71	37	64.295	3.262

Figure 4 Comparison of Rush Hour models: UP-extended vs PDDL

within the grid. Depending on the vehicle's location, it may have a varying number of possible moves without exiting the grid. By using this mode, we can define the range of movements from a minimum of `1` up to the maximum possible movement. For instance, in the `move_horizontal_car` action, illustrated in Listing 17, this maximum distance is `columns-2` when the car starts at one edge and moves towards the opposite edge. The permissive mode automatically filters out actions that exceed defined movement ranges, ensuring that actions are generated only for scenarios where vehicles can move within specified cell limits without leaving the grid. This optimization streamlines action creation by defining movement ranges at a higher level, eliminating the need to specify possible moves for each individual cell.

We compare our UP-extended model with an adapted version of ehajdin's PDDL model, available on GitHub [9]. Originally designed to restrict vehicle movements to one step per move, we modified it to enable full vehicle mobility, allowing movements ranging from 1 to 4 steps per move. From Fogleman's database, we selected the 43 most complex instances based on factors influencing difficulty outlined in the undergraduate thesis [25]. These instances are used to evaluate and compare the effectiveness of both models. From this selection, we highlight the 6 most difficult instances in Figure 4. While the model does not show improved efficiency compared to the PDDL implementation, it notably enhances problem description fluency and clarity, and significantly reduces model size.

5.3 8-Puzzle

The N-puzzle is a classic sliding puzzle game consisting of a $k * k$ grid with $((k * k) - 1)$ numbered tiles (N) and one blank space (Figure 2c illustrates an instance of the game). The objective is to rearrange the tiles by sliding them horizontally or vertically into the blank space, with the goal of achieving a specific configuration, often the ordered sequence of numbers from 1 to N.

Listing 18 Defining the 8-Puzzle grid.

```
puzzle = Fluent('puzzle', ArrayType(k, ArrayType(k, IntType(0,8))))
```

The definition of the grid, depicted in Listing 18, is represented as a 2D array where each cell contains an integer value ranging from 0 to n. The value 0 represents the empty space.

The representation of `slide_up` action is shown in Listing 19, which moves a tile up into the empty space in the grid. The parameters `r` and `c` are integers representing the row and

Initial Instance	Goal Instance	Plan (moves)	UP-extended	PDDL
876041253	012345678	31	20.882	0.774
806547231	012345678	31	20.213	0.810
856723410	012345678	30	20.762	0.844
854763210	012345678	30	19.999	0.792

Figure 5 Comparison of 8-puzzle models: UP-extended vs PDDL

column indices of the grid. Note that the range for `r` is from 1 to $k-1$, since you cannot slide up from the first row. The first precondition ensures that the tile above the current position (`r`, `c`) is the empty space, making the slide-up move possible. The effects describe the result of the action: the tile at the current position moves up and original position is now empty.

Listing 19 Defining the Slide-Up action.

```
su = unified_planning.model.InstantaneousAction('slide_up', r=IntType(1,k-1),
    c=IntType(0,k-1))
su.add_precondition(Equals(puzzle[r-1][c], 0))
su.add_effect(puzzle[r-1][c], puzzle[r][c])
su.add_effect(puzzle[r][c], 0)
```

To evaluate the performance of our model, we selected the two instances requiring the highest number of steps for a solution, both needing 31 steps, and the two configurations with the highest number of solutions, each having 64 solutions and requiring 30 steps, as detailed in the paper [21]. We obtained the PDDL model for these instances from the GitHub repository of the user mazina [20].

We observe the results in Figure 5. As with the Rush Hour model, our implementation did not outperform the PDDL implementation in terms of solving time. However, it also significantly enhanced the clarity and natural representation of the problem, while also reducing the model size.

6 Conclusions and Future Work

This proposed extension allows plans to be expressed more naturally, facilitating the management of complex problems. By making the modelling process more intuitive, it significantly reduces the manual effort and time required to select an optimal modelling approach. One significant advantage of this Python-based framework is its ability to effectively utilise Python functions, such as *for loops*, for iterating over different elements, allowing better manipulation of complex structures. This capability enhances the flexibility and power of the modelling process, making it easier to handle intricate scenarios. A key benefit of this pipeline is the significant modelling flexibility it provides, enabling the use of different compilers depending on the planner, which illustrates the diverse methods to model, transform and solve the same problem. As we have seen, arrays have been particularly useful in our examples.

Looking ahead, we aim to implement more high-level concepts such as functions, relations, (multi)sets, and sequences to further enhance the modelling capabilities of the framework. We also want to explore alternative compilers for each feature obtaining different possible encoding paths from our high-level UP extensions to PDDL (as done in constraint programming with Conjure [1] between Essence and Essence Prime). We also plan to conduct an experimental analysis of the solving time cost of using our proposed high-level UP representations.

References

1. Özgür Akgün, Alan M Frisch, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022.
2. Joan Espasa Arxer, Ian James Miguel, Peter Nightingale, András Z Salamon, and Mateu Villaret. Challenges in modelling and solving plotting with pddl. In *Knowledge Engineering for Planning and Scheduling*, 2023.
3. Behrouz Babaki, Gilles Pesant, and Claude-Guy Quimper. Solving classical AI planning problems using planning-independent CP modeling and search. In *13th International Symposium on Combinatorial Search (SOCS)*, pages 2–10. AAAI Press, 2020.
4. Roman Barták, Miguel A Salido, and Francesca Rossi. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing*, 21(1):5–15, 2010.
5. Roman Barták and Daniel Toropila. Reformulating constraint models for classical planning. In *21st FLAIRS*, pages 525–530, 2008.
6. Roman Barták and Jindrich Vodrázka. The effect of domain modeling on efficiency of planning: Lessons from the nomystery domain. In *TAAI*, pages 433–440, 2015.
7. Miquel Bofill, Joan Espasa, and Mateu Villaret. Relaxing non-interference requirements in parallel plans. *Log. J. IGPL*, 29(1):45–71, 2021.
8. Lukás Chrpa. Modeling planning tasks: Representation matters. In *Knowledge Engineering Tools and Techniques for AI Planning*, pages 107–123, 2020.
9. ehajdini. Rush hour - pddl. March 2019. URL: `https://github.com/ehajdini/AI/blob/master/RushHour_PDDL/domain1.pddl`.
10. Mojtaba Elahi and Jussi Rintanen. Planning with complex data types in PDDL. *CoRR*, abs/2212.14462, 2022.
11. Joan Espasa, Ian Miguel, and Mateu Villaret. Plotting: A planning problem with complex transitions. In *28th International Conference on Principles and Practice of Constraint Programming, CP*, volume 235 of *LIPIcs*, pages 22:1–22:17, 2022.
12. Michael Fogleman. Solving rush hour, the puzzle. July 2018. URL: `https://www.michaelfogleman.com/rush/`.
13. Guillem Francès and Hector Geffner. Effective planning with more expressive languages. In *25th International Joint Conference on Artificial Intelligence, IJCAI*, pages 4155–4159, 2016.
14. Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.
15. Alan M. Frisch and Peter J. Stuckey. The proper treatment of undefinedness in constraint languages. In *15th CP*, volume 5732 of *LNCS*, pages 367–382. Springer.
16. 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.
17. Alfonso Emilio Gerevini. An introduction to the planning domain definition language (PDDL): book review. *Artif. Intell.*, 280:103221, 2020.
18. Peter Gregory, Derek Long, Maria Fox, and J. Christopher Beck. Planning modulo theories: Extending the planning paradigm. In *22nd ICAPS*, 2012.
19. 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`.
20. mazina. Domains-planning-domain-definition-language/pddl. 2012. URL: `https://github.com/SoarGroup/Domains-Planning-Domain-Definition-Language/blob/master/pddl/eight01x.pddl`.
21. Alexander Reinefeld. Complete solution of the eight-puzzle and the benefit of node ordering in IDA. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 248–253. Morgan Kaufmann, 1993. URL: `http://ijcai.org/Proceedings/93-1/Papers/035.pdf`.

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

23 Enrico Scala. The enhsp planning system, 2023. URL: `https://gitlab.com/enricos83/ENHSP-Public/-/tree/enhsp-20`.

24 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.

25 Carla Davesa Sureda. Rush hour, 2023. URL: `https://dugi-doc.udg.edu/handle/10256/24566`.

26 Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. The 2023 International Planning Competition. *AI Magazine*, 45(2):280–296.

27 Vincent Vidal and Héctor Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. volume 170, pages 298–335. Elsevier, 2006.

28 Zheng Zang, Jiarui Song, Yaomin Lu, Xi Zhang, Yingqi Tan, Zhiyang Ju, Haotian Dong, Yuanyuan Li, and Jianwei Gong. A unified framework integrating trajectory planning and motion optimization based on spatio-temporal safety corridor for multiple agvs. *IEEE Trans. Intell. Veh.*, 9(1):1217–1228, 2024.