Undefinedness in Planning with Arrays

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

In a recent work, we introduced extensions to the Unified Planning framework to support more expressive modelling, including a new array type to handle multidimensional structures, support for bounded integer parameters in actions, and a new integer range variable type for quantified expressions. These are supported through a compilation process to the standard AI Planning language PDDL. When introducing complex data types such as arrays, accesses out of bounds may be a source of undefinedness. Undefinedness in numeric planning may occur only due to a lack of initialisation in numeric functions or due to ill-formed expressions such as division by zero. Planners should consider formulas containing expressions with undefined values as not satisfied in any state. However, the treatment of the undefinedness introduced through arrays cannot be delegated to the planner because arrays are not expressable in PDDL and hence planners do not natively support them. Therefore, out-of-bounds undefinedness must be addressed within the compilation process. In this work, we analyze several cases that demonstrate how out-of-bounds array access can be treated. There are a range of options, from simply treating a formula that contains an undefined subexpression as unsatisfiable, to a more nuanced treatment based upon where the undefinedness occurs, which provides more modelling convenience.

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

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)[6] 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.

While PDDL is powerful, its low-level abstractions can make it challenging to model certain types of planning problems efficiently [2]. Unified Planning (UP)[8] is a Python library where users can leverage Python's language features and libraries to construct planning models programmatically, and then transform them into PDDL. Many planning domains involve structured relationships and grid-like environments that are naturally represented using multidimensional arrays. To better support these domains, in previous work [10], we extended the UP framework with support for multidimensional arrays, bounded integer parameters in actions to enable direct indexing within these arrays, and range variables to support quantified expressions over integers.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These new modelling capabilities are designed to allow users to describe problems at a high level, without having to manually enumerate all possible cases. However, this introduces new challenges, especially regarding out-of-bounds array accesses, which may arise naturally when referring to neighbouring positions or iterating over structured data.

In the setting of classical planning, this kind of undefinedness is not accounted for, as all states and actions are assumed to be well-defined. This is because PDDL does not support data structures like arrays or lists.

PDDL2.1 introduces the notion of undefinednes in the context of numeric fluents [3]: These can initially be undefined, and any expression or comparison involving an undefined value is itself undefined. Consequently, any precondition or goal that contains such expressions cannot be satisfied, effectively making any plan that relies on undefined numeric values invalid. The Unified Planning (UP) framework claims to follow this same semantics. However, the treatment of undefinedness varies between engines as noted in the UP documentation [11]: "some engines might allow the reference to undefined values in disjunctions where at least one term is true, while the PDDL semantics would consider this expression as ill-formed." This discrepancy reveals that undefinedness is not handled consistently in current tools.

Recent work has explored ways to make planning more accessible by enabling more expressive and natural modelling. For example, several approaches [5, 7, 9, 1] allow users to describe problems at a higher level, often with support for arithmetic or structured data. However, these approaches do not address the treatment of undefined expressions or the potential issues arising from out-of-bound references.

In our previous work [10], we focused on the reformulation of the proposed extensions, since no existing solver supports them directly. In the next section, we summarize how this compilation process is performed. Because array accesses can be checked during compilation, we can detect when an expression refers to a position outside the valid bounds and handle it accordingly.

In this work, we explore strategies that handle such cases during compilation time, ensuring that no out-of-bound expressions are propagated to the solver. Our goal is to preserve the user's modelling intent as accurately as possible. We provide several modelling scenarios where undefined expressions naturally appear and explain how our proposal handles them.

2 Undefinedness in Planning with Arrays

Our extensions to the Unified Planning (UP) framework [10] introduced:

- **Multidimensional Arrays**, enabling the direct representation of grid-like structures. An example is the fluent `blocks` from Listing 2.
- **Bounded Integer Parameters in Actions**, allowing direct indexing and arithmetic operations over arrays. This can be seen in the action defined in Listing 2.
- **Range Variables**, representing bounded integer variables that can be used in quantifiers. An example is shown in Listing 5.

These constructs enable more expressive and structured modelling, but they also introduce a challenge: how to deal with expressions that refer to array positions that are not valid in the context of the domain. For example, accessing `grid[3][0]` in a 3×3 array (with indices ranging from 0 to 2). Such expressions are undefined, and failing to handle them properly can result in invalid models.

We use the term undefinedness to capture two main situations:

(i) *Out-of-bounds* accesses: when an expression refers to a position outside of an array's defined range.

(ii) *Explicitly undefined positions* accesses: when an expression refers to certain valid indices (according to the array's range) which are intentionally excluded from the structure (e.g., to model obstacles or walls). This is a convenient feature we provide in our latest work [*redacted*[1]], via the optional `undefined_positions` parameter in array fluents, which takes a list of tuples matching the structure dimension. This allows "holes" to be defined in the structure, ensuring that no action can interact with those positions. For example, in Listing 3, the positions `(0,0)` and `(5,0)` are excluded from a 6×6 grid to represent a wall, as in a variant of the classical Rush Hour.

Our compiler removes arrays from the problem by replacing them with individual fluents or values that represent each element of the array. Imagine, for instance, having a grid with two "forbidden cells" as illustrated in Figure 1. This would be represented in UP with a fluent `grid` that will be compiled into one fluent per valid cell, while positions specified in the `undefined_positions` parameter are discarded during compilation (see Figure 2).

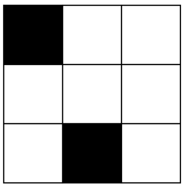

Figure 1 Grid example.

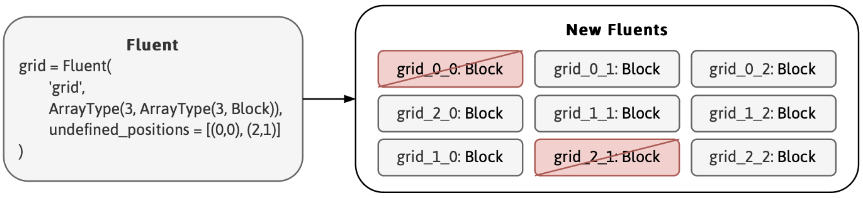

Figure 2 Compilation of a 2D array fluent with undefined positions to individual fluents representing each element.

The compiler also handles action schemes with integer parameters by generating one (partially) grounded action for each valid combination of parameter values within their declared bounds in which the integer parameters are replaced by specific values within the preconditions and effects. Additionally, the compiler simplifies expressions that involve integer arithmetic during the substitution process. An example is shown in Figure 3, where the `move_right` action scheme of a `Block` is expanded into multiple grounded actions. Some action instances are discarded during this process due to how undefined accesses are handled. We explain this behaviour in the next section.

This compiler also processes `RangeVariable` constructs by expanding integer quantifiers into equivalent logical expressions. A `Forall` over a `RangeVariable` is translated into a conjunction (`And`) over all possible values in the range, while an `Exists` over a `RangeVariable` is translated into a disjunction (`Or`) over the same values. Note that these quantifiers involve integer `RangeVariables` and are fully grounded during compilation. Therefore, when we refer to the treatment of undefined expressions within `Forall` and `Exists` in Table 1, we refer to quantifiers that remain in the model after this grounding step, i.e., those that quantify over objects as is usually done in PDDL.

We illustrate this transformation by adding a precondition to the `move_right` action in Figure 3, requiring that the block being moved is different from all blocks above it in the same column. Listing 1 shows this as a `Forall` over a `RangeVariable` ranging from `0` to `r`:

[1] This is a reference to a work submitted under double blind rules.

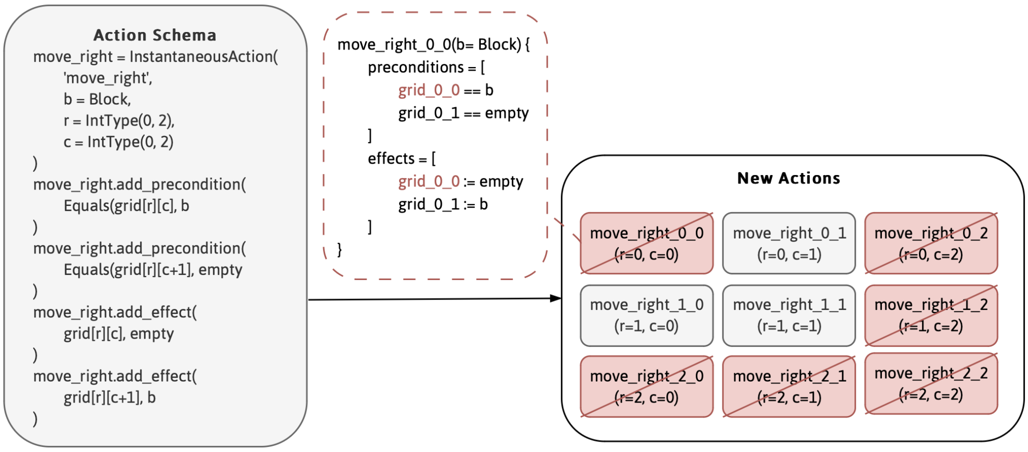

Figure 3 The parameterized action schema `move_right` (left) has two integer parameters to represent the row and column. Those get compiled away by binding them to specific grid positions (middle), and generating nine ground actions (right). Invalid actions (shown in red) are filtered out during compilation, while valid actions are kept.

Listing 1 A `Forall` over a `RangeVariable` that gets expanded during compilation.

```
b = RangeVariable("b", 0, r)
move_right.add_precondition(Forall(Not(Equals(grid[r][c], grid[b][c])), b))
```

At compilation time, this expression is expanded into a conjunction over all valid values of `b` in the given range. For example, in the partially grounded action `move_right(r=2, c=0)`, the condition becomes:

$$\texttt{grid[2][0]} \neq \texttt{grid[0][0]} \land \texttt{grid[2][0]} \neq \texttt{grid[1][0]} \land \texttt{grid[2][0]} \neq \texttt{grid[2][0]}$$

Handling undefined expressions properly is essential to ensure that the generated models are valid. In the following sections, we discuss how undefinedness can propagate in expressions and present our compilation-based approach for detecting and handling such cases, ensuring that the resulting model is valid and well-defined. As a result, modellers can write general action descriptions without having to handle special cases manually.

3 Our Approach

Out-of-range positions can be handled in the following two modes:

- ***Restrictive*** any out-of-bounds access is considered an error and aborts the compilation process.
- ***Permissive*** out-of-bounds accesses are interpreted according to the surrounding operator and context.

In the permissive mode, we propose a semantics for ruling the compilation of undefined expressions that is adapted to the needs of planning. We refer to the *undefined* value as $\perp$. Our approach is operator-aware: instead of propagating undefinedness blindly, it handles it in a way that reflects the role of each logical operator. Internally, undefined subexpressions are treated according to the context in which they appear. As we will illustrate, an out-of-bounds access occurring in a precondition is a distinct situation from a similar occurrence in an effect.

This analysis is performed entirely at compilation time, inside the compilers, which either eliminate the corresponding actions or modify the expressions according to our semantics. Importantly, when we say that an action is discarded, we refer to a specific instantiation generated during (partial) grounding. As a result, some instances of the same action scheme may be eliminated while others are preserved. This selective removal ensures that the resulting model is clean and safe before it is given to the solver, preventing the generation of possible invalid plans due to incomplete actions.

In our previous work [10], we introduced a new expression, `Count`, to allow users to count how many among a list of Boolean arguments evaluate to `True`. The expression can be written as `Count(a,b,c)` or `Count([a,b,c])`, where `a`, `b` and `c` are arbitrary Boolean expressions. When dealing with undefined values, `Count` behaves similarly to `Or`: any undefined arguments are ignored. Only those that are well defined contribute to the count.

Table 1 Propagation of undefined expressions by logical operators

Operator	Behaviour with Undefined Argument	Evaluation
`And`	If any argument is undefined, result is undefined	$\bot \wedge a \Rightarrow \bot$
`Not`		$\neg\bot \Rightarrow \bot$
`Iff`		$a \leftrightarrow \bot \Rightarrow \bot$
$\diamond \in \{=, \neq, <, >, \geq, \leq\}$		$\bot \diamond a \Rightarrow \bot$
`Forall`		$\forall x.\ \bot \Rightarrow \bot$
`Exists`		$\exists x.\ \bot \Rightarrow \bot$
`Or`	Any undefined argument is ignored, unless all are undefined	$\bot \vee a \Rightarrow a$ $\bot \vee \bot \Rightarrow \bot$
`Count`		`Count`$([a, \bot, b]) \Rightarrow$ `Count`$([a, b])$ `Count`$([\,]) \Rightarrow 0$
`Implies`	$a \rightarrow b$ treated as $\neg a \vee b$	$\bot \rightarrow b \Rightarrow \bot \vee b \Rightarrow b$ $a \rightarrow \bot \Rightarrow \neg a \vee \bot \Rightarrow \neg a$

Table 2 Handling of undefined expressions depending on the context

#	Expression Context	Behaviour	Justification
1	Precondition	Discard the entire action	The action is unsafe and invalid.
2	Left-hand side of effect		Writing to an invalid variable is always invalid.
3	Right-hand side of effect		Cannot assign an undefined value.
4	Conditional effect condition	Discard only the effect	The effect cannot happen, but the rest of the action may still be valid.

We illustrate each case from Table 2 with concrete examples and explain the justification for the chosen behaviour. Each example shows how an undefined expression affects the compilation process and why the action or effect is discarded or retained. In addition, these examples help demonstrate how undefined values propagate depending on the logical operators involved, as described in Table 1.

Case 1: Precondition

Any precondition that fully evaluates to $\perp$ or $False$ causes the corresponding action to be removed from the domain. This ensures that trivially inapplicable actions are removed.

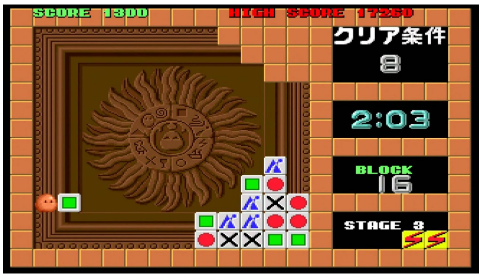

Figure 4 **Plotting** (Taito, 1989). The goal is to reduce the number of blocks in the grid to a target number (8 here) or fewer. The avatar (left) shoots blocks into the grid. If the shot block hits one of the same colour, that block is removed. State changes are complex because multiple blocks may be removed in a single shot and gravity affects the blocks in the grid.

Example 1: We demonstrate the handling of an undefined expression in a precondition with Plotting (Figure 4). In our formulation in Listing 2, the fluent `blocks` is a 2D array of `Colour` type values indexed by `rows` and `cols`, both integers.

Listing 2 Plotting: undefined precondition.

```
rows = 2
cols = 3
Colour = UserType("Colour")
empty = Object("empty")
blocks = Fluent("blocks", ArrayType(rows, ArrayType(cols, Colour)))

# Shoots a block of colour p into column c, clearing blocks up to row l.
sc = InstantaneousAction("shoot_col", p=Colour, c=IntType(0,cols-1), l=IntType(0,rows-1))
# Either l is the last row (we clear the full column), or the next block below (l+1) is
    different from p and is not empty.
sc.add_precondition(Or(Equals(l, last_row), And(Not(Equals(blocks[l+1][c], p)),
                                                Not(Equals(blocks[l+1][c], empty)))))
```

The `shoot-col` action consist in shooting the block held by the avatar against a column `c` given as a parameter of the action. All blocks of the same colour in the column are removed, starting from the top and continuing downwards, until a different colour block is found at row `l+1`, or the bottom of the column is reached. When processing the action, eight new actions are generated, corresponding to all combinations of the integer parameters `c` and `l`. Consider the resulting action generated with parameter values `(c=0, l=1)`. The corresponding precondition includes an array access to `blocks[2][0]`, which is out of bounds and is thus treated as undefined. The compilation step then handles this as follows:

$$
\begin{aligned}
(\texttt{l} = 1) \vee (\neg(\texttt{blocks[l+1][c]} = \texttt{p}) \wedge \neg(\texttt{blocks[l+1][c]} = \texttt{empty})) \\
\Rightarrow (1 = 1) \vee (\neg(\texttt{blocks[2][0]} = \texttt{p}) \wedge \neg(\texttt{blocks[2][0]} = \texttt{empty})) \\
\Rightarrow \texttt{True} \vee (\neg \bot \wedge \neg \bot) \\
\Rightarrow \texttt{True} \vee (\bot \wedge \bot) \\
\Rightarrow \texttt{True} \vee \bot \\
\Rightarrow \texttt{True}
\end{aligned}
$$

In this case, we are checking whether either `l` is the last row, or the block below `l` is of a different colour and not empty. For this specific instantiation, `l` refers to the last row, so the first disjunct holds. Since the overall precondition is a disjunction, the second disjunct, which contains the undefined access, does not need to be evaluated. Thanks to our operator-aware semantics, the undefined expression is safely ignored. As a result, the entire condition evaluates to `True`, and the action instance is preserved, which is the desired behaviour.

Let us now consider a situation where we mistakenly specified the integer parameter as `l = IntType(0, rows)`. This would allow the generation of action instances with `l = 2`, which exceeds the valid row indices and may lead to out-of-bounds accesses.

$$
\begin{aligned}
(\texttt{l} = 1) \vee (\neg(\texttt{blocks[l+1][c]} = \texttt{p}) \wedge \neg(\texttt{blocks[l+1][c]} = \texttt{N})) \\
\Rightarrow (2 = 1) \vee (\neg(\texttt{blocks[3][0]} = \texttt{p}) \wedge \neg(\texttt{blocks[3][0]} = \texttt{N})) \\
\Rightarrow \texttt{False} \vee (\neg \bot \wedge \neg \bot) \\
\Rightarrow \texttt{False} \vee (\bot \wedge \bot) \\
\Rightarrow \texttt{False} \vee \bot \\
\Rightarrow \texttt{False}
\end{aligned}
$$

Our compilation approach detects that the action instance will never be applicable, as its precondition evaluates to `False`. As a result, the action is safely discarded.

Example 2: In this example from the Rush Hour domain (Figure 5), a precondition of the action `move_car_right`, shown in Listing 3, checks that the selected cell is not empty and thus it contains a vehicle that can be moved.

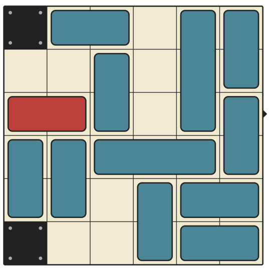

Figure 5 Example of a **Rush Hour** puzzle instance. The red car must be moved to the exit on the right side of the 6×6 grid by sliding vehicles that can only move along their orientation and cannot cross over other vehicles.

Listing 3 Rush Hour: undefined precondition.

```
rows = 6
cols = 6
Vehicle = UserType("Vehicle")
empty = Object("empty", Vehicle)
```

```
at = Fluent("at", ArrayType(rows, ArrayType(cols, Vehicle)),
            undefined_positions=[(0,0), (5,0])
mcr = InstantaneousAction("move_car_right", r=IntType(0,rows-1), c=IntType(0,cols-1))
# there is a vehicle in this position
mcr.add_precondition(Not(Equals(at[r][c], empty)))
```

For the instantiation `(r=0, c=0)`, the position corresponds to a wall and is explicitly marked as undefined, which means that no action can happen there. Our approach detects the access to an undefined position and eliminates the corresponding action during compilation.

$$\neg(\texttt{at[r][c]} = \texttt{empty}) \Rightarrow \neg(\texttt{at[0][0]} = \texttt{empty}) \Rightarrow \neg \perp \Rightarrow \perp$$

Example 3: In this example from the Sokoban domain (Listing 4), the action `move` uses several implications to ensure that the target cell in the intended direction is empty. For each direction (right, left, up, down), a precondition states that if the movement is in a given direction, then the adjacent cell in that direction must be empty.

Listing 4 Sokoban: undefined precondition.

```
Pattern = UserType("Pattern")
P = Object("P", Pattern)  # Player
B = Object("B", Pattern)  # Box
empty = Object("empty", Pattern)
Direction = UserType("Direction")
right = Object("right", Direction)
left = Object("left", Direction)
up = Object("up", Direction)
down = Object("down", Direction)

rows = 3
cols = 3
grid = Fluent("grid", ArrayType(rows, ArrayType(cols, Pattern)))

move = InstantaneousAction("move", d=Direction, r=IntType(0,rows-1), c=IntType(0,cols-1))
move.add_precondition(Implies(Equals(d, right), Equals(grid[r][c+1], empty)))
move.add_precondition(Implies(Equals(d, left), Equals(grid[r][c-1], empty)))
move.add_precondition(Implies(Equals(d, up), Equals(grid[r-1][c], empty)))
move.add_precondition(Implies(Equals(d, down), Equals(grid[r+1][c], empty)))
```

Consider the instantiation `move(d=right, r=1, c=2)`. In a 3×3 grid, the cell `grid[1][3]` is out of bounds. The precondition is evaluated as follows:

$$\begin{aligned}(\texttt{d} = \texttt{right}) \rightarrow (\texttt{grid[1][3]} = \texttt{E}) \Rightarrow (\texttt{d} = \texttt{right}) \rightarrow \perp \Rightarrow \neg(\texttt{d} = \texttt{right}) \vee \perp \\ \Rightarrow \neg(\texttt{d} = \texttt{right})\end{aligned}$$

The implication becomes a negated condition: the action can only be applied if the direction is not `right`, allowing the action to be used in other directions.

Example implies right argument?

Cases 2 and 3: Left-hand and Right-hand Sides of an Effect

When an effect involves undefined expressions, either because the fluent being assigned is invalid (left-hand side) or because the value being assigned is undefined (right-hand side), the action is discarded. Such assignments may lead to incorrect states, so the corresponding action instances are safely removed during compilation to ensure model correctness.

Example 4: Let us revisit the action `shoot_col(c, l)` from Listing 2 in the Plotting domain. This action sets all positions in column `c`, from row 0 to `l`, to `empty`, using a universal quantifier over a range variable `b`:

Listing 5 Plotting: left-hand side of an effect undefined.

```
b = RangeVariable("b", 0, l)
sc.add_effect(blocks[b][c], empty, forall=[b])
```

For the instance `(c=0, l=2)` in the 2×3 grid, this results in the following effects: `blocks[0][0]:=empty`, `blocks[1][0]:=empty`, `blocks[2][0]:=empty`. Since this effect is unconditional and `blocks[2][0]` side is undefined, our compilation approach discards the entire action instance `as expected?`.

Example 5: We now consider a new domain: Pancake Sorting (see Figure 6, where the stack of pancakes is represented using an integer array. The model uses a range variable `b` to capture the pairs of symmetric index involved in the flip.

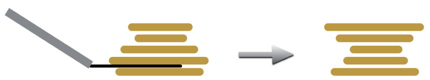

Figure 6 Example of a **Pancake** instance of size 5 where the action `flip(3)` is applied. The main action *flips* the top $f+1$ pancakes, reversing their order. For the action `flip(f)`, the corresponding positions (b) and ($f-b$) are swapped for all $x \in 0 \ldots f$.

Listing 6 Pancake: undefined left-hand and right-hand side of effects.

```
n = 5
pancake = Fluent("pancake", ArrayType(n, IntType(0, 4)))
flip = InstantaneousAction("flip", f=IntType(1, n))
f = flip.parameter("f")
b = RangeVariable("b", 0, f)
flip.add_effect(pancake[b], pancake[f - b], forall=[b])
```

For the instance `flip(5)`, the effect `pancake[b] := pancake[f - b]` expands to:

```
pancake[0] := pancake[5], pancake[1] := pancake[4],
pancake[2] := pancake[3], ..., pancake[5] := pancake[0]
```

Since `pancake[5]` is out of bounds and appears in an unconditional effect, the action is discarded during compilation.

Case 4: Conditional Effect Condition

The condition of the effect cannot be evaluated and the effect will never happen, but the rest of the action may still be valid in other contexts.

Example 6: In this example from the Puzznic domain (see Figure 7) shown in Listing 8, the action `matching` applies a conditional effect to each cell in a 3×3 grid. The effect sets a cell to `F` (free) if it is not already free and at least one of its four adjacent neighbours has the same pattern.

When this is expanded using universal quantification over all grid positions, some resulting conditions may involve undefined positions, either because they lie outside the array bounds or are explicitly marked as walls.

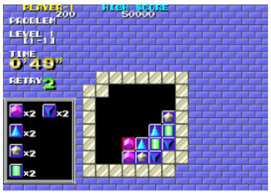

Figure 7 Example of a **Puzznic** puzzle instance. The player must move blocks within a grid so that identical blocks touch and disappear, following gravity constraints. Solving each level requires planning a sequence of moves to avoid blocking necessary paths or isolating tiles.

Listing 7 Puzznic: undefined conditional effect condition.

```
rows = 3
cols 3
Pattern = UserType("Pattern")
empty = Object("empty", Pattern)
at = Fluent("at", ArrayType(3, ArrayType(3, Pattern)), undefined_positions=[(1,1),(2,2)])

matching = InstantaneousAction("matching")
i = RangeVariable("i", 0, rows-1)
j = RangeVariable("j", 0, cols-1)
matching.add_effect(at[i][j], empty,
    condition=And(Not(Equals(at[i][j], empty)),
                  Or(Equals(at[i+1][j], at[i][j]), Equals(at[i-1][j], at[i][j]),
                     Equals(at[i][j+1], at[i][j]), Equals(at[i][j-1], at[i][j]))),
    forall=[i,j])
```

For instance, in the effect `at[1][0] := F`, the condition includes checks on `at[1][1]` (a wall) and `at[1][-1]` (out of range). These are treated as undefined, and because it is a disjunction, the condition is simplified accordingly and the effect is kept.

$$\begin{aligned}
&\neg(\texttt{at[1][0]} = \texttt{F}) \land ((\texttt{at[2][0]} = \texttt{at[1][0]}) \lor (\texttt{at[0][0]} = \texttt{at[1][0]}) \lor \\
&\qquad (\texttt{at[1][1]} = \texttt{at[1][0]}) \lor (\texttt{at[1][-1]} = \texttt{at[1][0]})) \\
&\Rightarrow \neg(\texttt{at[1][0]} = \texttt{F}) \land ((\texttt{at[2][0]} = \texttt{at[1][0]}) \lor (\texttt{at[0][0]} = \texttt{at[1][0]}) \lor \bot \lor \bot) \\
&\Rightarrow \neg(\texttt{at[1][0]} = \texttt{F}) \land ((\texttt{at[2][0]} = \texttt{at[1][0]}) \lor (\texttt{at[0][0]} = \texttt{at[1][0]}))
\end{aligned}$$

Now consider the effect `at[1][1] := F`, where the target position is a wall and explicitly marked as undefined. In this case, the condition itself refers entirely to undefined terms.

$$\begin{aligned}
&\neg(\texttt{at[1][1]} = \texttt{F}) \land ((\texttt{at[2][1]} = \texttt{at[1][1]}) \lor (\texttt{at[0][1]} = \texttt{at[1][1]}) \lor \\
&\qquad (\texttt{at[1][2]} = \texttt{at[1][1]}) \lor (\texttt{at[1][0]} = \texttt{at[1][1]})) \\
&\Rightarrow \neg \bot \land (\bot \lor \bot \lor \bot \lor \bot) \\
&\Rightarrow \bot \land \bot \\
&\Rightarrow \bot
\end{aligned}$$

Since the entire condition is undefined, the effect is removed-even if the left-hand side is also undefined. The action is retained, as it may still apply to other valid positions.

Example 7: Revisiting the `shoot_column` action from Listing 2, this action includes a conditional effect that assigns a new value to `hand` if there is a next block below. If the bottom of the column has been reached, the effect does not apply and the hand colour does not change.

For the instantiation `(c=0, l=1)` in that same grid, the effect becomes:

Listing 8 Plotting: undefined conditional effect condition.

```
shoot_column.add_effect(hand, blocks[l+1][c], condition=LT(l, lr))
```

Our compilation approach first evaluates the condition:

$$1 < 1 \Rightarrow False$$

Although `blocks[2][0]` is undefined (out of bounds), it does not cause an error because the condition is `False`. As a result, the effect is safely discarded, while the action remains valid and kept.

4 Related Work

A related line of work in the field of Constraint Programming explores how to treat expressions that involve undefined values [4] presents three alternative strategies to handle undefined subexpressions in logical formulas. The first two approaches are three-valued: they support an explicit undefined value ($\perp$), and define how it propagates through the different operators. This is similar to our permissive mode, where undefined expressions are also propagated. However, their system handles undefinedness at constraint solving time, whereas we need to resolve it during compilation. Since we do not know the evaluation of the fluents at that point, we must define a fixed behaviour based only on the structure of the expression.

The third approach replaces undefined expressions with `False`. Although this may be convenient in certain contexts, it can lead to unintended behaviour in others. Some of the examples discussed previously show how this strategy may incorrectly allow an action to be applied, simply because an undefined condition is treated as `False`.

In the Plotting example shown in Listing 2, for the first instantiation shown, `(c=0, l=1)`, the entire condition evaluates to `True`, and the action is correctly kept. In this particular case, blindly treating undefined values as `False` produces the correct result by coincidence.

$$
\begin{aligned}
&(\texttt{l} = 1) \lor (\neg(\texttt{blocks[l+1][c]} = \texttt{p}) \land \neg(\texttt{blocks[l+1][c]} = \texttt{empty}))\\
&\Rightarrow (1 = 1) \lor (\neg(\texttt{blocks[2][0]} = \texttt{p}) \land \neg(\texttt{blocks[2][0]} = \texttt{empty}))\\
&\Rightarrow \texttt{True} \lor (\neg\texttt{False} \land \neg\texttt{False})\\
&\Rightarrow \texttt{True} \lor (\texttt{True} \land \texttt{True})\\
&\Rightarrow \texttt{True} \lor \texttt{True}\\
&\Rightarrow \texttt{True}
\end{aligned}
$$

However, for the next instantiation `(c=0, l=2)`, the same treatment leads to accepting an action instance that should be discarded, since it relies on accessing an out-of-bound cell. This contradicts the intended semantics in planning, where all preconditions must be safely evaluable.

$$
\begin{aligned}
&(\texttt{l} = 1) \lor (\neg(\texttt{blocks[l+1][c]} = \texttt{p}) \land \neg(\texttt{blocks[l+1][c]} = \texttt{N}))\\
&\Rightarrow (2 = 1) \lor (\neg(\texttt{blocks[3][0]} = \texttt{p}) \land \neg(\texttt{blocks[3][0]} = \texttt{N}))\\
&\Rightarrow \texttt{False} \lor (\neg\texttt{False} \land \neg\texttt{False})\\
&\Rightarrow \texttt{False} \lor (\texttt{True} \land \texttt{True})\\
&\Rightarrow \texttt{False} \lor \texttt{True}\\
&\Rightarrow \texttt{True}
\end{aligned}
$$

A similar situation occurs in the Rush Hour example shown in Listing 3. The undefined expression is replaced with `False`, and the negation turns it into `True`, causing the precondition to be incorrectly satisfied and the action incorrectly retained.

$$\neg(\texttt{at[r][c]} = \texttt{none}) \Rightarrow \neg(\texttt{at[0][0]} = \texttt{none}) \Rightarrow \neg\texttt{False} \Rightarrow \texttt{True}$$

We now revisit the third example from the Sokoban domain (Listing 4), which uses implications in the precondition. The third strategy handles this by replacing the right-hand side of the implication with `False`, which works correctly in this context, as we also do.

However, if the implication were nested inside other expressions—such as a negation or disjunction—this approach could lead to incorrect results. Blindly replacing undefined values with `False` may cause invalid actions to be mistakenly preserved.

5 Discussion

In all the problems that we have modelled so far, our approach behaves as expected: the generated actions match the intent of the user, and the compilation deals with undefined accesses in a predictable way. However, certain modelling patterns reveal limitations of the current semantics, especially when quantifiers interact with undefined values.

Let us consider an example (see Listing 5) that highlights this issue. Suppose that we have an array of integers of size 5 (indices from 0 to 4) where `a[1]` is undefined, all valid cells are initially set to `0`. We define an action that increments the value at index `c`, but only if all previous values are less than or equal to `a[c]`. We can write this precondition using a `forall` quantifier.

```
array = Fluent("array", ArrayType(5, IntType(0,4)), undefined_positions=[(1)])
increment = InstantaneousAction("increment", c=IntType(0,4))
c = increment.parameter("c")
i = RangeVariable("i", 0, c-1)
increment.add_precondition(Forall(LE(array[i], array[c]), i))
```

If we try to apply the action at `c = 3`, the quantifier is compiled into a conjunction. Since one of the terms is undefined, the whole precondition becomes undefined and the action is discarded:

$$
\begin{aligned}
&\texttt{forall i} \in \texttt{0..2. a[i] <= a[3]}\\
&\Rightarrow (\texttt{a[0]} \leq \texttt{a[3]}) \land (\texttt{a[1]} \leq \texttt{a[3]}) \land (\texttt{a[2]} \leq \texttt{a[3]})\\
&\Rightarrow (\texttt{a[0]} \leq \texttt{a[3]}) \land \bot \land (\texttt{a[2]} \leq \texttt{a[3]})\\
&\Rightarrow \bot
\end{aligned}
$$

If we instead rewrite the same condition using a logically equivalent formulation based on a negated `exists`, the behaviour changes:

$$
\begin{aligned}
&\texttt{not (exists i} \in \texttt{0..2. a[i] > a[3])} \\
\Rightarrow\ &\neg\big((\texttt{a[0]} > \texttt{a[3]}) \lor (\texttt{a[1]} > \texttt{a[3]}) \lor (\texttt{a[2]} > \texttt{a[3]})\big) \\
\Rightarrow\ &\neg\big((\texttt{a[0]} > \texttt{a[3]}) \lor \bot \lor (\texttt{a[2]} > \texttt{a[3]})\big) \\
\Rightarrow\ &\neg\big((\texttt{a[0]} > \texttt{a[3]}) \lor (\texttt{a[2]} > \texttt{a[3]})\big)
\end{aligned}
$$

In this case, the quantifier expands into a disjunction. The undefined term is simply ignored, and the result depends only on the remaining defined comparisons. The two formulations are logically equivalent if all values are defined, but behave differently when some positions are undefined.

In fact, if we look at how similar issues are handled in other declarative modelling paradigms like constraint programming—particularly in MiniZinc—we find the same problem. In MiniZinc, when an array access is out of bounds, the expression is evaluated as `False`, which can lead to inconsistent behaviour. Consider the two formulations in Listing 9, which are logically equivalent but behave differently.

Listing 9 Logically equivalent formulations behave differently when handling out-of-bounds access in MiniZinc.

```
array[1..5] of var int: a;

% UNSAT due to i=5 being out-of-bounds
constraint forall (i in 1..5)(a[i] > a[i+1]);

% SAT
constraint forall (i in 1..5)(not(a[i] <= a[i+1]));
```

Although one could think of removing undefined expressions in quantifiers as the default right choice, we have shown that this could lead to invalid plans (see Listing 6).

We believe that this limitation could be addressed by allowing quantifiers over integer `RangeVariables` to include an optional *annotation* indicating whether undefined elements should be ignored. This would give more control to the modeller and make the behaviour of quantifiers more flexible, aligning better with what users expect when modelling problems that involve inaccessible positions.

References

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

2 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.

3 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.

4 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.

5 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.

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

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

8 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. URL: https://www.sciencedirect.com/science/article/pii/S2352711024003820, doi:10.1016/j.softx.2024.102012.

9 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.

10 Carla Davesa Sureda, Joan Espasa Arxer, Ian Miguel, and Mateu Villaret Auselle. Towards high-level modelling in automated planning, 2024. URL: https://arxiv.org/abs/2412.06312, arXiv:2412.06312.

11 Unified Planning Consortium. Problem representation — unified planning documentation. https://unified-planning.readthedocs.io/en/latest/problem_representation.html. Accessed: 2025-06-03.