Compiling Expressive Planning with Data Types

Carla Davesa[1], Joan Espasa[1], Ian Miguel[1], Mateu Villaret[2]

[1]School of Computer Science, University of St Andrews, UK
[2]Departament d'Informàtica, Matemàtica Aplicada i Estadística, Universitat de Girona, Spain
cds26@st-andrews.ac.uk, jea20@st-andrews.ac.uk, ijm@st-andrews.ac.uk, mateu.villaret@udg.edu

Abstract

Planning involves selecting action sequences to achieve goals from initial conditions, a fundamental task arising in contexts ranging from daily tasks to industrial processes. The Planning Domain Definition Language (PDDL) is the standard formalism for describing such problems, yet previous work has highlighted some of its limitations in expressivity. In particular, many real-world planning problems inherently exhibit structural patterns such as sets, arrays, and counting. These are features of the problem domain itself that are cumbersome to model directly in PDDL. We present a high-level modeling language that compiles to PDDL, supporting richer data structures and operations including arrays, sets, Boolean counting expressions, integer range variables, and bounded integer parameters in actions. This allows users to express problems in terms closer to their natural structure, without committing to a particular low-level encoding. From a single high-level specification, our framework can automatically generate and explore multiple PDDL encodings with different performance trade-offs in solving time, memory usage, and plan quality. We demonstrate our approach on a set of benchmark problems involving complex structural patterns. Our results show that compiled models are competitive with established handcrafted models from the literature, and can sometimes outperform them.

Code — https://github.com/carladavesa/unified-planning

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. Examples of planning problems arise in many contexts, such as business process management (Marrella 2019), robotics (Karpas and Magazzeni 2020) or cybersecurity (Choi et al. 2021). The difficulty of solving planning problems grows rapidly with their size in terms of the number of states and possible actions considered, though considerable effort has resulted in the development of highly efficient AI planners (Taitler et al. 2024). The Planning Domain Definition Language (PDDL) (Gerevini 2020) is the de-facto leading language used in automated planning to model planning problems, providing a formal way to describe problems in terms of objects, predicates, actions, and functions with parameters. A given planning problem typically has multiple possible PDDL models, and the chosen model has a significant effect on the performance of state-of-the-art AI planning systems (Chrpa 2020; Barták and Vodrázka 2015). Hence, it is important to recognize the importance of modeling and consider it as much a part of the process of solving a planning problem as the search for a solution. However, previous work has highlighted the limitations of PDDL, particularly its low level of abstraction (Rintanen 2015). This is especially challenging when modeling problems with inherent structural patterns, such as arrays, sets, and counting operations. To illustrate the challenges of modeling in PDDL, consider the well-known 15-Puzzle: a 4×4 grid where numbered tiles slide into a blank space. Since standard PDDL lacks integer parameters, representing the grid requires encoding indices as objects (`idx`) and using explicit successor predicates (`next`) to simulate arithmetic operations like `c+1` (Listing 1). Ideally, one would express this directly using a 2D integer array with bounded integer action parameters to access the affected positions (Listing 2).

We present a high-level modeling language that enables such natural specifications. Our main contribution is a language supporting richer constructs including arrays, sets,

```
(:types tile idx)
(:predicates (grid ?t - tile ?r ?c - idx)
   (empty ?r ?c - idx) (next ?c1 ?c2 - idx))
(:action move-right
 :parameters (?t - tile ?r ?c ?nc - idx)
 :precondition (and (grid ?t ?r ?c)
   (empty ?r ?nc) (next ?nc ?c))
 :effect (and (grid ?t ?r ?nc) (empty ?r ?c)
   (not (grid ?t ?r ?c)) (not (empty ?r ?nc))))
```

Listing 1: 15-Puzzle domain in PDDL

```
fluents = [array[4, array[4, integer[0, 15]]] grid]
action move_right(integer[0, 3] r, integer[0, 2] c):
 preconditions = [(grid[r][c+1] == 0) and
                  (not (grid[r][c] == 0))]
 effects = [grid[r][c+1] := grid[r][c], grid[r][c] := 0]
```

Listing 2: 15-Puzzle with high-level constructs

Boolean counting expressions, integer range variables, and bounded integer parameters in actions, combined with automated compilation to PDDL. We integrate our work within the Unified Planning (UP) library (Micheli et al. 2025), a Python framework that provides a unified interface to various planning engines and formalisms. By providing multiple compilation strategies, our framework can automatically generate alternative PDDL encodings with different performance trade-offs in solving time, memory usage, and plan quality from a single high-level specification. This removes the burden of manual low-level modeling while maintaining compatibility with existing planning engines. We demonstrate how these features enable natural high-level models of benchmark problems involving complex structural patterns. Our empirical results demonstrate that the models we generate are competitive with, and sometimes outperform, handcrafted models from the literature.

2 Background and Related Work

The Planning Domain Definition Language (Gerevini 2020) (PDDL) provides a modeling language to declaratively define planning problems. PDDL allows for lifted representations, where actions can be represented by action templates with parameters. Over the years, PDDL has evolved to include support for features such as numeric types, temporal constraints, durative actions or derived predicates. However, some of these extensions are "conditioned" by the leading search-based solving mechanisms, which do not allow functionalities such as integer parameters in action templates, nor support complex data structures such as arrays or sets. Several works in the literature highlight the significant lack of expressiveness in PDDL, particularly when it comes to representing more complex planning scenarios (Geffner 2003; Rintanen 2015; Espasa et al. 2024b).

Languages such as ANML (Smith, Frank, and Cushing 2008) can natively express structured types, but no planner supports this feature. Semantic Attachments (Dornhege et al. 2009) offer another potential approach to supporting richer types, allowing external modules to evaluate conditions and effects beyond standard PDDL, although this requires modifications to the underlying planner, unlike our approach. An alternative and well-established solution is reformulation (Alarnaouti, Baryannis, and Vallati 2023). That is, more expressive features can be compiled away to semantically equivalent representations, allowing existing solvers to tackle problems formulated in more expressive fragments without requiring any change to the underlying search algorithms. This idea has been applied to existing PDDL features such as numeric planning (Bonassi, Percassi, and Scala 2025; Gigante and Scala 2023), trajectory constraints (Percassi and Gerevini 2019), PDDL+ (Percassi, Scala, and Vallati 2021), conditional effects (Gerevini, Percassi, and Scala 2024), conformant planning (Palacios and Geffner 2009) and temporal planning (Rintanen 2017), enabling researchers to model using rich problem representations while leveraging existing solvers. Elahi and Rintanen (2022) proposed a modeling language that supports complex data types, including records, tuples, unions, and lists, reduced to a Boolean PDDL representation, leveraging existing classical planners. Our approach differs in several key aspects: while their framework commits to a single compilation pathway targeting only classical PDDL, ours provides multiple pathways targeting both classical and numeric PDDL. Regarding data types, we support counting expressions and integer range variables, which they do not, while they allow complex types as action parameters, which we do not. For arrays, they allow variable indices at runtime via disjunctions, whereas ours ensures constant indices before compilation. For sets, both use Boolean membership predicates, but we additionally support cardinality expressions via DNF or auxiliary integer fluents.

3 Problem Definition

Our source planning problem considers a lifted representation of actions, functions and predicates. Functions may return object-typed elements, bounded integers, arrays, or sets. We assume the equality predicate is available for all types. We support the interpreted types of bounded integers, arrays, and sets, with the common constants, operators, and predicates for them detailed below.

Definition 1 (Types) *We consider the* Boolean type*:* $Bool$*;* atomic types*: Object*[1] *and bounded integers* $Int\langle lb, ub\rangle$*; and* compound types*:* $Array(size, \tau)$*, arrays of* `size` *elements of type* τ *indexed from* 0 *to* $size - 1$*, and* $Set(\tau)$*, sets of non-repeated elements of object type* τ*.*
From these types we can also construct functional types[2] *of the form:* $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ *referring to types of functions with* n *parameters of types* $\tau_1 \ldots \tau_n$ *and returning elements of Boolean, atomic or compound type* τ*.*

Definition 2 (Signature) *We define a signature* Σ *as the tuple* (T, F, P, X)*, where* T *is a finite set of types,* F *is a finite set of functionally typed symbols not returning* $Bool$*,* P *is a finite set of functionally typed symbols returning* $Bool$ *(i.e. predicates), and* X *is a possibly infinite set of atomic typed variables grouped in subsets* X_τ *for* $\tau \in T$*. To specify the type* τ *of each symbol* s *we use* $s : \tau$*.*

Definition 3 (Terms) *Given a signature* $\Sigma = (T, F, P, X)$ *we define well-typed* Σ*-terms (or term, for short) with their type, overloading the* $:$ *symbol as follows:*

$$
\begin{array}{ll}
x : \tau & \text{for all } x \in X_\tau \\
f(t_1, \ldots, t_n) : \tau' & \text{for all } f : \tau_1 \times \ldots \times \tau_n \rightarrow \tau' \in F \\
p(t_1, \ldots, t_n) : Bool & \text{for all } p : \tau_1 \times \ldots \times \tau_n \rightarrow Bool \in P
\end{array}
$$

whenever for all $i \in \{1..n\}$*,* $t_i : \tau_i$ *and* τ_i *and* τ' *are atomic or compound types. Boolean terms are also called* atoms*. The nullary symbol case is already covered when* $n = 0$*.*

Definition 4 (Formula) *Given a signature* $\Sigma = (T, F, P, X)$*, formulas, (or well-formed formulas) are defined recursively: every atom is a formula, if* ϕ *is a formula then* $not(\phi)$ *is a formula, if* ϕ_1 *and* ϕ_2 *are formulas then* $and(\phi_1, \phi_2)$*,* $or(\phi_1, \phi_2)$*,* $implies(\phi_1, \phi_2)$*,*

[1] Object types can be organized in a subtype hierarchy (as in PDDL) with root type $object$.

[2] Atomic and compound types are particular cases of functional types with 0 parameters.

$if f(\phi_1, \phi_2)$ are formulas, and if ϕ is a formula and $x \in X$ a variable, then $forall(x : \tau, \phi)$ and $exists(x : \tau, \phi)$ are formulas if x appears free in ϕ. $FreeVars(\phi)$ is the set of free variables occurring in ϕ.

To define the semantics of formulas, we need to specify which domains we are considering, and then how to interpret the symbols and evaluate terms and formulas. Therefore, we first define the domains of our interpretations.

Definition 5 (Domain) *A domain for an atomic or compound type is the set of elements that inhabit that type. For object types τ, Dom_τ is a non-empty finite set of elements. For all numeric types of the form Int$\langle _, _ \rangle$ there is only a domain $Dom_{Int<lb..ub>}$ that will be a (sufficiently large) range of integers $lb..ub$. For array types of the form $Array(n, \tau)$, the domains are the possible arrays of n elements of type τ indexed from 0 to $n-1$. For set types of the form $Set(\tau)$, the domains are the possible sets of elements from 2^{Dom_τ}.*

Definition 6 (Interpretation) *Given a signature $\Sigma = (T, F, P, X)$, an* interpretation *$\mathcal{I}$ for Σ assigns a domain Dom_τ to each of the atomic and compound types $\tau \in T$; a function $f^{\mathcal{I}} : Dom_{\tau_1} \times ... \times Dom_{\tau_n} \to Dom_\tau$ to each function symbol $f : \tau_1 \times ... \times \tau_n \to \tau \in F$; a function $p^{\mathcal{I}} : Dom_{\tau_1} \times ... \times Dom_{\tau_n} \to \{true, false\}$ to each predicate symbol $p : \tau_1 \times ... \times \tau_n \to Bool \in P$; and an element $x^{\mathcal{I}} \in Dom_\tau$ to each variable $x : \tau \in X$.*

Since interpretations will capture the notion of state in a planning problem, we classify symbols in a signature according to whether their interpretation can change during plan execution or not and according to whether the semantics is given by a particular interpreted type or not.

Definition 7 (Built-in Symbols, Objects and Fluents) *Given a signature $\Sigma = (T, F, P, X)$, we partition the symbols in $F \cup P$, into* built-in *(fixed interpretation for integers, arrays and sets),* objects *(nullary symbols of object type, representing constants in the domain; by $objects(\tau)$ we refer to the objects of type τ), and* fluents *(symbols whose interpretation varies across states). From fluent symbols we can construct* state variables*: $f(o_1, \ldots, o_n)$ where $f \in fluents(\Sigma)$ and $o_i \in objects(\Sigma)$.*

Undefinedness When evaluating state variables or more complex expressions, we may encounter situations where some terms are undefined. For example, accessing an array element outside the defined bounds or dividing by zero. To handle these cases uniformly, we introduce the special value $\perp$ to represent undefinedness in all types, including $Bool$, thereby extending all domains to include $\perp$. This treatment follows the approach used in constraint programming (Frisch and Stuckey 2009) and our previous work (Davesa et al. 2025).

Interpreted Types - Signature and Semantics We now provide the signatures and fixed semantics for our interpreted types, with their treatment of undefinedness.

Integers For a given signature $\Sigma = (T, F, P, X)$ with numeric types in T, we have built-in function symbols $F_{int} = \{plus, minus, prod, div\} \cup Nums \subseteq F$ where $Nums$ is a finite set of numbers in a range $lb..ub$, and predicate symbols $P_{int} = \{lt, le, gt, ge, eq\} \subseteq P$. All function symbols have type $Int\langle lb, ub\rangle \times Int\langle lb, ub\rangle \to Int\langle lb, ub\rangle$, and predicate symbols have type $Int\langle lb, ub\rangle \times Int\langle lb, ub\rangle \to Bool$. The interpretation of these symbols with the associated domain $Dom_{Int<lb,ub>}$ is the obvious correspondence with the integer operators. and works as follows: whenever any of the parameters is $\perp$, the result is $\perp$, otherwise it is the resulting number; finally, the division by zero is $\perp$.

We also have an additional function symbol to count how many atoms in a list are true, $F_{count} = \{count\} \subseteq F$. The semantics of $count : \overbrace{Bool \times \ldots \times Bool}^{n} \to Int\langle lb, ub\rangle$ is defined as follows:

$$count^{\mathcal{I}}_n(b_1, \ldots, b_n) = \begin{cases} \perp & \text{if } \forall_{i \in 1..n}.b_i = \perp \\ |\{b_i \mid b_i = true\}| & \text{otherwise} \end{cases}$$

Arrays For a given signature $\Sigma = (T, F, P, X)$ with array types in T, we have built-in function symbols $F_{Array} = \{read, write\} \subseteq F$ and a predicate symbol $P_{Array} = \{eq\} \subseteq P$. In some scenarios, certain positions may be inaccessible due to obstacles or intentional exclusion. To model this, the array fluents support an *undefined_positions* parameter that marks specific indices as permanently undefined. We also refer to these as out-of-bounds. In practice, we use syntactic sugar for array operations: $a[i]$ denotes $read(a, i)$, and array element assignment $a[i] := v$ is represented as a full array assignment $a := write(a, i, v)$. The interpretation of $read : Array(n, \tau) \times Int\langle 0, n-1\rangle \to \tau$ and $write : Array(n, \tau) \times Int\langle 0, n-1\rangle \times \tau \to Array(n, \tau)$ are defined as follows:

$$read^{\mathcal{I}}(a, i) = \begin{cases} a[i] & \text{if } i \in \{0..n-1\} \\ \perp & \text{otherwise} \end{cases}$$

$$write^{\mathcal{I}}(a, i, v) = \begin{cases} a' \text{ where } a' : Array(n, \tau) \text{ and} \\ \quad \forall_{p=0}^{n-1} a'[p] = \begin{cases} v & p{=}i \\ a[p] & \text{else} \end{cases} \\ \quad \text{if } i \in \{0..n{-}1\} \wedge a[i] \neq \perp \\ \perp \text{ otherwise} \end{cases}$$

The interpretation of $eq : Array(n, \tau) \times Array(n, \tau) \to Bool$ is defined as follows:

$$eq^{\mathcal{I}}(a_1, a_2) = \begin{cases} a_1 = a_2 & \text{if } a_1[i], a_2[i] \neq \perp \\ & \quad \forall i \in \{0..n-1\} \\ \perp & \text{otherwise} \end{cases}$$

Notice that to be able to use arrays we need also numbers since we are accessing elements by indexing with numeric values, therefore we allow integers to be in $Dom_{Int<lb..ub>}$.

Sets For a given signature $\Sigma = (T, F, P, X)$ with set types in T, we have built-in function symbols $F_{set} = \{\{\}, add, remove, union, intersect, difference, card\} \subseteq F$ and predicate symbols $P_{Set} = \{member, subseteq, disjoint, eq\} \subseteq P$. The interpretation of $\{\} : Set(\tau)$, $add, remove : Set(\tau) \times \tau \to Set(\tau)$; $union, intersect, difference : Set(\tau) \times Set(\tau) \to Set(\tau)$; $card : Set(\tau) \to Int\langle lb, ub\rangle$; $member : Set(\tau) \times \tau \to$

$Bool$; $subseteq, disjoint, eq : Set(\tau) \times Set(\tau) \rightarrow Bool$ is the obvious correspondence with the set operators: whenever any of the parameters is $\perp$, the result is $\perp$; otherwise, the operator is applied normally.

We can now define how to evaluate terms and formulas.

Definition 8 (Term and Atom Evaluation) *Given a signature $\Sigma = (T, F, P, X)$, an interpretation for it $\mathcal{I}$ and a Σ-term (or a Σ-atom) t we inductively define $[\![t]\!]_{\mathcal{I}}$ as follows:*

$$[\![x]\!]_{\mathcal{I}} = x^{\mathcal{I}} \quad \text{for all variables } x \in X$$

$$[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{I}} = f^{\mathcal{I}}([\![t_1]\!]_{\mathcal{I}}, \ldots, [\![t_n]\!]_{\mathcal{I}}) \quad \text{for all } f \in F \text{ (or } f \in P)$$

Notice that n can be zero.

Definition 9 (Formula evaluation) *Formula evaluation propagates undefinedness with strict semantics. Logical operators return $\perp$ if any operand is $\perp$, except for or and $exists$, which ignore $\perp$ operands and return $\perp$ only when all operands are $\perp$. We say that an interpretation $\mathcal{I}$ satisfies ϕ, notated as $\mathcal{I} \models \phi$, iff $[\![\phi]\!]_{\mathcal{I}} = true$.*

We can now define the planning problems we consider.

Definition 10 (Soundly Inhabiting Interpretation) *We say that an interpretation $\mathcal{I}$, for $\Sigma = (T, F, P, X)$,* soundly inhabits it*, if and only if, for all atomic types $\tau \in T$, $\mathcal{I}$ establishes a one-to-one correspondence between objects of type τ and $Dom_\tau \setminus \{\perp\}$. Moreover, the interpretations for the built-in symbols of the signature are those previously described in the corresponding interpreted types.*

Definition 11 (Actions) *Given a signature $\Sigma = (T, F, P, X)$ an* action schema *is characterized by its name and a tuple with parameters, precondition, and effects: $a = \langle params, pre, eff \rangle$ where $params$ is a list of object or numeric typed variables, pre is a formula with $FreeVars(pre) \subseteq params$, and eff is a list of conditional effects $\langle C \triangleright t := t' \rangle$ and unconditional effects $\langle t := t' \rangle$:*

- *C is a formula such that $FreeVars(C) \subseteq params$,*
- *t is a term of the form $f(p_1, \ldots, p_n)$ where $f \in fluents(\Sigma)$, $p_i \in params \cup objects(\Sigma)$ and,*
- *t' is a well-typed expression of the same type as t, or a formula (if $t : Bool$)*

We call $t := t'$ an assignment*. Effects can be quantified using $forall$: $forall(x : \tau, e)$, represents one effect per object of type τ (or number in $lb..ub$ if $\tau = Int\langle lb, ub \rangle$), directly unfolded.*

An action *a is the result of grounding an action schema where parameters of object type τ have been replaced by objects of type τ and numeric typed parameters of type $int\langle l, u \rangle$ have been replaced by numbers within $l..u$, hence it has the form: $a = \langle pre, eff \rangle$ where no free variables (parameters) occur in pre nor in eff. Moreover, the quantified effects $forall$ have been unfolded.*

Definition 12 (Action applicability) *Given a signature Σ and an interpretation $\mathcal{I}$ that soundly inhabits it, and given an action $a = \langle pre, eff \rangle$, we say that a* is applicable *in $\mathcal{I}$ iff: $\mathcal{I} \models pre$; no two effects (unconditional or conditional such that $\mathcal{I}$ satisfies their condition) assign to the same state variable; for every unconditional effect $\langle l := r \rangle$: $[\![r]\!]_{\mathcal{I}} \neq \perp$, and if $l : Int\langle lb, ub \rangle$ then $[\![r]\!]_{\mathcal{I}} \in \{lb..ub\}$; for every conditional effect $\langle C \triangleright l := r \rangle$ where $\mathcal{I} \models C$: $[\![r]\!]_{\mathcal{I}} \neq \perp$, and if $l : Int\langle lb, ub \rangle$ then $[\![r]\!]_{\mathcal{I}} \in \{lb..ub\}$; for every assignment $\langle l := r \rangle$ (conditional or unconditional) where $l : Array(n, \tau)$, arrays l and r* share the same undefined positions*, i.e., for all $i \in 0..n-1$, $[\![l[i]]\!]_{\mathcal{I}} = \perp \leftrightarrow [\![r[i]]\!]_{\mathcal{I}} = \perp$.*

The application of a to $\mathcal{I}$, denoted as $a(\mathcal{I})$, updates $\mathcal{I}$ by applying all unconditional effects plus all conditional effects $\langle C \triangleright l := r \rangle$ where $\mathcal{I} \models C$.

Definition 13 (Planning Problem) *A planning problem Π is a tuple $\langle \Sigma, \mathcal{I}_0, \mathcal{A}, g \rangle$ where Σ is a signature, $\mathcal{I}_0$ is an interpretation that* soundly inhabits *Σ, $\mathcal{A}$ is a set of action schemes over Σ and g is a Goal formula.*

Let $\mathcal{G} = ground_{\mathcal{I}_0}(\mathcal{A})$ be all the actions that result from grounding all action schemes in $\mathcal{A}$ with all possible combinations of objects and numeric values. Let $\pi = (a_1, \ldots, a_n)$ be a sequence of actions where each $a_i \in \mathcal{G}$, and let $\mathcal{I}_i = a_i(\mathcal{I}_{i-1})$ for all $i \in 1..n$. We say that π is a valid plan for Π if each action a_i is applicable on $\mathcal{I}_{i-1}$ and $\mathcal{I}_n \models g$.

Π has a solution if and only if there exists a plan for it.

4 The Pipeline

Our planning framework is implemented in the Unified Planning (UP) library (Micheli et al. 2025). UP streamlines the process of transforming planning problems into formats suitable for input to various planners. Similarly to PDDL, UP uses a lifted representation with state variables and parameterized actions, providing support for features such as conditional effects or user-type fluents, which might not be supported by all planning engines. UP provides their own compilers to transform the problem into a planner-specific representation by removing unsupported features. We have extended the UP syntax to provide support for our interpreted types, integer parameters in actions, and quantification over bounded integers. We have implemented several compilers to transform this high-level representation of the problem to representations supported by the target planning engine. We use the 15-Puzzle as a running example throughout this section. UP then translates this *planner-specific UP representation* into languages such as PDDL and ANML (Smith, Frank, and Cushing 2008), depending on the planner's requirements. Finally, the converted planning problem is fed to the planner. UP provides access to a variety of planners.

Language Fragment We consider the source planning problem from Definition 13 over the following fragment:

Fluent symbols can only have object-typed parameters. *Set element types* are restricted to object types. *Nested set operations* are disallowed, except that $card$ may contain one level of set operations (e.g., $card(add(A, e))$). Array indices in $read$ and $write$ cannot contain fluents. Indices can only be constants, parameters, and their arithmetic combinations. Array writes cannot be nested and can only appear on the right-hand side of effect assignments. The $count$ function cannot appear on the right-hand side of an assignment and must be compared with a constant or an integer parameter (e.g., $gt(count(\ldots), 2)$).

New compilers

We introduce six new compilers that eliminate our extended constructs. All compilers follow the structure of those already in the UP library, ensuring consistency and compatibility. In some cases, we provide multiple compilation strategies for the same construct, enabling the exploration of alternative models from a single high-level representation. To understand the impact of each transformation, we provide a cost analysis based on the number of fluents, objects, or actions introduced.

To ensure correctness, compilers need to deal with action applicability and therefore they need to deal with undefinedness carefully. For the sources of undefinedness that can be detected at compilation time, we simply remove them. For the sources that cannot be detected a priori (such as out-of-bound assignments to integer fluents), we add the appropriate preconditions in the actions to ensure correctness.

Integer Parameter Actions Remover (IPAR) Eliminates integer parameters and quantification over integer ranges from action schemas through partial grounding. Each action schema is partially grounded into multiple action schemes by instantiating all integer-typed parameters with all values in their declared bounds. Quantifiers over integer ranges are unfolded, and arithmetic expressions are simplified.

Since indexing expressions only contain constants, integer quantified variables or parameters, here is where out-of-bounds array accesses can occur in $read$ and $write$ operations. This compilation phase also prevents all possible arithmetic errors (such as division by zero), and integer assignments that would violate declared bounds.

Action instances are discarded if a precondition or any right-hand side on an unconditional effect evaluates to $\bot$. Conditional effects $\langle C \triangleright l := r \rangle$ are handled as follows: if the condition C evaluates to $\bot$ or false at compilation time, the effect is removed. Otherwise, if the right-hand side r evaluates to $\bot$ at compile time, the effect is removed and $not(C)$ is added to the preconditions to prevent the action from being applicable when the effect would violate applicability conditions. For effects assigning to integer fluents $l : \text{Int}\langle lb, ub\rangle$, where r is not known at compilation time, $(and(leq(lb, r), leq(r, ub)))$ is added to the precondition (or, $or(not(C), and(leq(lb, r), leq(r, ub))))$ for conditional effects). For division operations, if the divisor is a fluent or expression that could evaluate to zero, a precondition is added ensuring it never equals zero. This ensures all generated action instances respect the applicability conditions. For instance, in the 15-Puzzle problem shown in Listing 2, IPAR grounds `move_right(r,c)` into $4{\times}3 = 12$ action instances, simplifying `c+1` to a constant in each, also handling bound checking, as illustrated below.

Example 4.1 *Consider an action with integer parameters $x : Int\langle 1, 5\rangle$ and $y : Int\langle 2, 6\rangle$ containing the assignment $x{:=}plus(2, y)$. IPAR adds preconditions to ensure bounds are respected: the precondition $and(leq(1, plus(2, y)), leq(plus(2, y), 5))$ is added to prevent out-of-bounds assignments (restricting y to values 2 or 3 in this case).*

Proposition 1 *Let A be the set of actions that contain integer parameters. For each action $a \in A$, let $IP(a)$ denote its set of integer parameters. For each $p \in IP(a)$, let $lb(p)$ and $ub(p)$ be its lower and upper bounds, respectively. The maximum number of grounded actions that can be generated: $\sum_{a \in A} \prod_{p \in IP(a)} (ub(p) - lb(p) + 1)$.*

Arrays Remover (AR) Eliminates arrays by replacing array fluents with multi-parameter fluents indexed by a new object type. Requires that integer parameters have been eliminated (IPAR must be applied first), ensuring all array accesses use constant indices. A new object type $Index$ is introduced with objects $i_0, \ldots, i_{n-1}$ where $n = \max\{n_1, \ldots, n_d\}$ across all array dimensions in the problem. Each d-dimensional array fluent $f : Array(n_1, \ldots, Array(n_d, \tau)\ldots)$ is replaced by $f' : \overbrace{Index \times \ldots \times Index}^{d} \to \tau$ (preserving any original parameters of f). Note that this allows syntactically valid but semantically undefined combinations (e.g., $f'(i_3, i_0)$ when $n_1 = 2$), but such accesses have already been eliminated by IPAR. Array accesses $f[c_1]\ldots[c_d]$ are replaced with $f'(i_{c_1}, \ldots, i_{c_d})$, array assignments $f[c_1]\ldots[c_d] := v$ become $f'(i_{c_1}, \ldots, i_{c_d}) := v$, and array comparisons are expanded by an element-wise comparison. For instance, the 15-Puzzle `grid` fluent becomes `grid'(i_r, i_c) : Number` with `Index` objects $\{i_0, i_1, i_2, i_3\}$, and `grid[1][2]` becomes `grid'(i_1, i_2)`.

Proposition 2 *Let F_A be the set of array fluents. For each fluent $f \in F_A$, let $L(f)$ denote the list of lengths of its dimensions, and let $U(f)$ be the set of positions marked as undefined for f. The compiler introduces one new object type ($Index$) with $\max_{f \in F_A, l \in L(f)} l$ objects, and the total number of fluent instances is: $\sum_{f \in F_A} \left(\left(\prod_{l \in L(f)} l \right) - |U(f)| \right)$.*

Arrays Logarithmic Remover (ALR) Transforms integers and integer (multi)arrays into a purely Boolean representation using binary encoding. Requires that integer parameters have been eliminated (IPAR applied first). It currently supports (in)equalities between integer terms (fluents or constants). Full arithmetic support is focus of future work. A new object type $Position$ is introduced with objects $p_{j_1,\ldots,j_d}$ for each valid index combination in the arrays, excluding positions marked as undefined. Each integer fluent $f : Int\langle lb, ub\rangle$ or array fluent $f : Array(n_1, \ldots, Array(n_d, Int\langle lb, ub\rangle)\ldots)$ is replaced by $b = \lceil \log_2(ub - lb + 1) \rceil$ Boolean fluents $bit^f_0, \ldots, bit^f_{b-1}$ (preserving any original parameters of f), where array fluents have an additional parameter $Position$. Equality comparisons $f = v$ transform into conjunctions over bit fluents; less-than comparisons use lexicographic bit comparison; assignments $f := v$ set each bit fluent according to the binary representation of v.

Example 4.2 *The fluent `grid : Array(4, Array(4, Int⟨0,15⟩))` becomes: 16 `Position` objects $\{p_{0,0}, \ldots, p_{3,3}\}$, and 4 Boolean fluents $bit_0, \ldots, bit_3 : Position \to Bool$ ($\lceil \log_2 16 \rceil = 4$). An assignment `grid[1][2] := 5` (binary 0101_2)*

becomes: $\texttt{bit}_0(p_{1,2})$:= *false,* $\texttt{bit}_1(p_{1,2})$:= *true,* $\texttt{bit}_2(p_{1,2})$:= *false,* $\texttt{bit}_3(p_{1,2})$:= *true.*

Proposition 3 *Let F_{AI} be the set of integer and integer array fluents. For each fluent $f \in F_{AI}$, let $L(f)$, $U(f)$, $ub(f)$ denote its dimension lengths (empty for integer fluents), undefined positions, and integer upper bound. The compiler introduces: $\sum_{f \in F_{AI}} \lceil \log_2(ub(f)+1) \rceil$ Boolean fluent symbols. For array fluents, it also introduces $\sum_{f \in F_A} (\prod_{l \in L(f)} l - |U(f)|)$ Position objects, where $F_A \subseteq F_{AI}$ is the subset of array fluents.*

Sets Remover (SR) Transforms set fluents into Boolean predicates representing membership. Currently supports only sets of object types. Each fluent $S : Set(\tau)$ becomes $S'(e : \tau) : Bool$ (the element parameter first, preserving any original parameters of S) where $S'(e) = true$ iff $e \in S$. Built-in symbols are expanded element-wise: $member(S, e)$ becomes $S'(e)$; $eq(S_1, S_2)$, with $S_1, S_2 : Set(\tau)$ transformed to $forall(x : \tau, iff(S'_1(x), S'_2(x)))$; $disjoint(S_1, S_2)$ becomes $forall(x : \tau, not(and(S'_1(x), S'_2(x))))$. Effect assignments are expanded as follows: $S := add(S, e)$ becomes $S'(e) := true$; Other operations ($remove$, $union$, $intersect$, $difference$, $subseteq$) are expanded similarly. Cardinality expressions $card(S)$ are transformed to $count$: $count([S'(e) \mid e \in objects(\tau)])$.

Proposition 4 *Let F_S be the set of set fluents. For each fluent $f \in F_S$ of type $Set(\tau)$, the compiler replaces it with a Boolean fluent with an additional parameter of type τ. The total number of Boolean fluent instances is $\sum_{f \in F_S} |Dom_\tau| \cdot \prod_{p \in params(f)} |Dom_p|$ where $params(f)$ denotes any parameters f has.*

Count Remover (CR) Eliminates $count$ expressions by expanding them into disjunctive normal form (DNF). Since $count$ expressions always appear within comparisons (e.g., $lt(count(a, b, c), 2)$), the compiler explicitly enumerates all satisfying assignments.

Proposition 5 *The compiler does not introduce new fluents, objects, or actions. The size of the resulting formula for a comparison $op(count(x_1, \ldots, x_n), k)$ (where $op \in \{eq, lt, leq, gt, geq\}$) is $O\left(\binom{n}{k}\right)$ in the worst case.*

Count Integer Remover (CIR) Transforms $count$ expressions into integer arithmetic by introducing auxiliary integer fluents for each count subexpression. For each argument arg_i in $count(arg_1, \ldots, arg_n)$, a new fluent $count_i : Int\langle 0, 1\rangle$ is added that tracks whether arg_i is true ($count_i = 1$) or false ($count_i = 0$). If arg_i contains parameters (e.g., fluents or variables), $count_i$ preserves those parameters. If the same subexpression appears in multiple $count$ expressions, the fluent is reused. The $count$ expression becomes $plus(plus(plus(count_0, count_1), \ldots), count_{n-1})$. When an action effect modifies a fluent f appearing in arg_i, the compiler adds effects to update $count_i$: if the assignment makes arg_i evaluable to a constant, an unconditional effect sets $count_i$ accordingly (preserving any original effect conditions); otherwise, two conditional effects are added to set $count_i$ based on whether the updated arg_i evaluates to true or false. In the initial state, each $count_i$ is set by evaluating its corresponding arg_i.

Proposition 6 *Let C be the set of count expressions in the problem. For each $c \in C$, let $A(c)$ be the set of subexpressions. The total number of integer fluent symbols introduced is $\sum_{c \in C} |A(c)|$.*

Integers Remover (IR) Transforms integer fluents into object fluents by treating each integer value as a distinct object. Since all numeric types have finite bounds, we can determine exactly which integer values any expression can reach. This allows us to use a single precomputed integer domain $Dom_{\text{Int}\langle l,u\rangle}$ consisting precisely of these reachable values. A single object type $Number$ is introduced with objects $(n_l, \ldots, n_u)$. Each integer fluent $f : \text{Int}\langle lb, ub\rangle$ is replaced by $f' : Number$ (preserving any original parameters). Since all integer fluents share the same $Number$ type, this allows syntactically valid but semantically out-of-bounds assignments (e.g., $f'_1 = n_5$ when $f_1 : Int\langle 0, 2\rangle$); however, the compilation adds constraints to preconditions ensuring such assignments never occur in applicable actions. When integer fluents appear in arithmetic expressions or comparisons in preconditions or goals, the compiler uses OR-Tools CP-SAT solver (Perron and Didier 2025) to compute all satisfying value assignments for the integer variables involved and generates a DNF formula enumerating these valid assignments. Effects with arithmetic operations $f : \text{Int}\langle lb, ub\rangle := g$ are expanded as follows: for each possible value v in f declared bounds $lb..ub$, the compiler uses CP-SAT to find combinations of the fluent values appearing in g (within their respective bounds) that make the expression equal to v. This generates conditional effects where *conditions* specify the required fluent values. Recall that fluents are restricted to their declared bounds by IPAR. Increase/decrease effects are expanded similarly. Continuing Example 4.1, with $x := plus(2, y)$ where $x \in \text{Int}\langle 1, 5\rangle$ and $y \in \text{Int}\langle 2, 6\rangle$, IR expands this to conditional effects: $\langle eq(y, n_2) \triangleright x := n_4\rangle$ and $\langle eq(y, n_3) \triangleright x := n_5\rangle$.

Proposition 7 *Let I be the set of integer fluents with domains $D(i) = \langle l_i, u_i\rangle$. IR introduces $\left|\bigcup_{i \in I} \{l_i, \ldots, u_i\}\right|$ $Number$ objects.*

Compilation Order The compilers have dependencies that determine their safe application order. IPAR must be applied first, as it eliminates integer parameters and ensures array indices are constants, which AR, ALR, and IR require. SR operates independently on set fluents and can be applied before or after IPAR. CR and CIR eliminate count expressions and must be applied after SR, since SR expands cardinality into count. IR must be applied last, as it eliminates integer fluents produced by other compilers.

Example To illustrate one possible compilation path, we apply `uti` (IPAR, AR, IR) to the 15-Puzzle from Listing 2, generating $4 \times 3 = 12$ action instances. The resulting fluent and one such instance are shown in Listing 3, where IPAR has instantiated the integer parameters (r=0, c=0), AR has replaced the array by an indexed fluent with `Index` objects, and IR has replaced integer values by `Number` objects.

```
types = [Index, Number]
objects = [Index: [i0, ..., i3], Number: [n0, ..., n14]]
fluents = [Number puzzle[i_1 = Index, i_2 = Index]]
action move_right_0_0:
 preconditions = [(puzzle(i0, i1) == n0) and
                  (not (puzzle(i0, i0) == n0))]
 effects = [puzzle(i0, i1) := puzzle(i0, i0),
            puzzle(i0, i0) := n0]
```

Listing 3: 15-Puzzle after `uti` compilation

Compilers Correctness Only IPAR introduces new actions, doing so in a semantics-preserving manner through the instantiation of integer parameters. All other compilers maintain a one-to-one correspondence (modulo removal of inapplicable actions) between the source and target actions. After the compilation chain, no state variable or expression is evaluated as $\perp$ during the execution of any valid plan. IPAR eliminates all integer parameters, making array indices constant and detecting out-of-bounds accesses statically: actions with such accesses in preconditions or unconditional effect RHS are removed; conditional effects with such accesses in conditions or RHS are removed. For effects on integer fluents with non-constant RHS, bound checks are added to preconditions/conditions, ensuring assignments stay within bounds. AR and ALR operate on constant indices only (IPAR precondition), hence no runtime index errors. SR, CR, CIR, and IR do not introduce new sources of undefinedness. For each compiler, if a ground action a is applicable in state $\mathcal{I}$, then the corresponding compiled actions are applicable in the compiled $\mathcal{I}$. Conversely, if a is not applicable (due to precondition failure or undefinedness), then either the corresponding action is removed during compilation, or the compiled precondition evaluates to false in the compiled version. In IPAR, action instances where a precondition or unconditional effect RHS evaluates to $\perp$ are discarded. For integer assignments where the right side is non-constant, a bounds check is added to the precondition, ensuring the target action is applicable iff the source action satisfies the bounds requirement. AR, ALR, and SR perform syntactic transformations that preserve the truth value of preconditions under the compilation correspondence. CR and CIR preserve the integer value of count expressions, hence comparisons involving count evaluate identically. In IR, integer fluents are replaced by object-based representations with predicates encoding arithmetic; the compiled preconditions are logically equivalent under the compilation.

5 Experimental Evaluation

Our empirical hypothesis is twofold: (i) the extended features enable more natural modeling of a range of problems, and (ii) our alternative compilation paths provide variety in the set of output models with complementary strengths. We model and solve nine domains: **15-Puzzle** (100 instances from (Asai and Fukunaga 2015); compared against the handcrafted model): a 4×4 array of Int$\langle 0, 15\rangle$ with 4 action schemas each taking 2 Int$\langle 0, 3\rangle$ parameters. **Pancake Sorting** (Gates and Papadimitriou 1979) (50 instances, 10 per size: 5–25 (n) pancakes; no existing PDDL model was available, we handcrafted one for comparison): a 1D array of Int$\langle 0, n-1\rangle$, 1 action schema with 1 Int$\langle 1, n-1\rangle$ parameter, and range variables. **Plotting** (85 instances of grids up to 7×7, compared against the published model (Espasa et al. 2024b); several bugs in the handcrafted model were identified and fixed, which highlights the difficulty of manually modeling complex problems in PDDL): a $n\times n$ array of object type ($n\leq7$), 4 action schemas with up to 2 Int$\langle 0, n-1\rangle$ parameters, range variables, and a count expression in the goal. **Puzznic** (39 instances from (Espasa et al. 2024a), grids up to $\sim10\times7$; both the handcrafted model and ours use derived predicates, a feature we integrated through UP framework support (Speck 2024)): an $n\times m$ array of object type with 3 action schemas taking 2 Int$\langle 0, n-1\rangle$ and Int$\langle 0, m-1\rangle$ parameters, and range variables. **Rush Hour** (10 hardest instances from the game; compared against the handcrafted model (ehajdini 2019)): a 7×7 array of object type with 12 action schemas each taking 2 Int$\langle 0, 6\rangle$ parameters. **Sokoban** (39 instances with grids up to $\sim10\times10$ from the IPC 2011 benchmark (Potassco Team 2011); compared against the handcrafted model): an $n\times m$ array of object type with 8 action schemas each taking 2 Int$\langle 0, n-1\rangle$ and Int$\langle 0, m-1\rangle$ parameters. **Labyrinth** (40 instances with grids up to 5×5 from the IPC 2023 benchmark (IPC - Classical Track 2023); compared against the handcrafted model): an $n\times n$ array of object type with 8 action schemas (4 movement with 2 Int$\langle 0, n-1\rangle$ parameters, 4 pushing with 1 Int$\langle 0, n-1\rangle$ parameter) and range variables. The final two domains are **Dump Trucks** (5 instances with 10–20 packages, 2 locations, 2 trucks) and **Storytellers** (11 instances with 5–20 stories, 2 audiences, 2 storytellers). We tried to mimic the problems from (Gregory et al. 2012). As the software is not publicly available, direct performance comparison cannot be conducted.

The experiments were run on Intel(R) Xeon(R) E-2234 CPUs@3.60 GHz with 16GB of RAM. We used the planners integrated into the UP framework to ensure a consistent evaluation setting and to demonstrate the compatibility of our compilation pipeline with standard planning tools. Specifically, we used[3] Fast Downward (Helmert 2006) (v24.06) and SymK (Speck, Mattmüller, and Nebel 2020) (v1.3.1) as our main classical planners, and ENHSP (Scala, Haslum, and Thiébaux 2016) (v20) for problems involving numbers. All planners were used with their default configurations and a timeout of 30 minutes. We ran each planner in three modes: *OneShot* to obtain an initial solution, *Anytime* to identify the best solution possible given a time bound, and *Optimal* to guarantee the optimal solution, if feasible. We applied several compilation strategies to each of our models to obtain different versions to compare with handcrafted PDDL models (`hc`). The classical strategies are: `up` (IPAR, AR), `log` (IPAR, ALR), `uti` (IPAR, AR, IR), `c` (IPAR, AR, CR), `ci` (IPAR, AR, CIR, IR), `sc` (SR, CR), `sci` (SR, CIR, IR); and the numeric strategies (requiring a numeric planner): `int` (IPAR, AR), `cin` (IPAR, AR, CIR), `scin` (SR, CIR). After applying our compilers, UP's built-in compilers are ap-

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

plied as needed to produce a fully compatible representation (e.g., removing object fluents, which are not supported by most planning engines). This illustrates the ability of our system to explore different low-level models from a single high-level representation, and indeed we will see that different strategies perform better depending on the problem. The new extensions add negligible compilation time: most domains compile in under 5 seconds. The exception is count expressions with many arguments, where the resulting formula grows combinatorially (e.g., Dump Trucks and Storytellers with 20 objects take over 600 seconds to compile). To evaluate both coverage and plan quality, we will use the IPC2023 score for each configuration on all benchmarks. Table 1 reports the total scores in all domains and operation modes.

In general, our compiled models consistently matched or outperformed handcrafted models. The most substantial improvements were observed in the IPC 2023 labyrinth domain, where our `up` compilation with SymK substantially outperformed the handcrafted model in both anytime and oneshot modes, solving significantly more instances. Similar results were achieved in Plotting, where all our compilations outperformed the handcrafted model across all three modes, with the `cin` compilation demonstrating particularly strong performance with ENHSP. In Pancake Sorting, the `uti` compilation scaled considerably better than the handcrafted model in anytime mode, while `uti` with Fast Downward performed exceptionally well in oneshot mode. For optimal planning, all compilations outperformed the handcrafted model, with SymK showing excellent scalability. The 15-Puzzle domain showed more varied results: in anytime mode, the `uti` and `log` compilations matched the handcrafted model. Our `int` compilation scaled better in oneshot mode, and the `log` encoding proved superior for optimal planning. Both Rush Hour and Sokoban exhibited similar patterns, with the `up` compilation scaling slightly better than the handcrafted model across all modes. Particularly noteworthy was an order-of-magnitude improvement achieved in solving times by our compilation when paired with SymK in both domains. Finally, in Puzznic, our `up` compilation scaled marginally better than the handcrafted model across all modes, though performance remained broadly comparable.

		OneShot			Anytime			Optimal		
		F	S	E	F	S	E	F	S	E
15-Puzzle (100)	hc	**68.4**	32.0	44.6	**99.2**	32.0	24.9	19	32	0
	uti	55.9	0.0	39.0	88.3	0.0	20.1	0	0	0
	log	50.3	38.0	9.2	89.0	41.0	4.3	0	**37**	0
	int	x	x	26.1	x	x	17.2	x	x	0
Pancake Sorting (50)	hc	11.2	10.0	10.9	17.1	10.0	10.0	0	10	9
	uti	**32.3**	20.0	10.7	**40.5**	18.8	19.6	0	20	20
	log	11.8	21.0	9.3	21.5	23.0	19.8	0	**21**	20
	int	x	x	12.1	x	x	21.0	x	x	20
Plotting (85)	hc	32.5	10.0	3.4	36.9	10.0	4.6	0	10	5
	c	56.6	32.0	64.4	62.8	32.0	69.0	0	32	68
	ci	58.8	21.0	66.4	64.8	21.0	70.9	0	21	x
	cin	x	x	**74.3**	x	x	**84.0**	x	x	**79**
Puzznic (39)	hc	2.0	12.2	x	2.0	11.2	x	0	13	x
	up	2.7	**14.1**	x	2.8	**14.1**	x	0	**15**	x
Rush Hour (10)	hc	6.1	**10.0**	1.3	**10.0**	**10.0**	1.0	**10**	**10**	2
	up	6.3	**10.0**	7.5	**10.0**	**10.0**	9.5	**10**	**10**	**10**
Sokoban (20)	hc	12.3	16.0	12.5	18.8	15.0	14.7	18	16	12
	up	13.5	**20.0**	11.5	18.0	**19.0**	12.9	17	**20**	12
Labyrinth (40)	hc	23.4	0.0	19.3	22.9	0.0	20.7	0	0	0
	up	**33.1**	0.0	0.0	**38.4**	0.0	0.0	0	0	0
Dump Trucks (5)	sc	0.0	0.0	2.0	0.0	0.0	0.0	0	0	0
	sci	0.0	0.0	0.0	0.0	0.0	0.0	0	0	0
	scin	x	x	**4.1**	x	x	0.0	x	x	**5**
Story-tellers (11)	sc	0.0	0.0	**10.0**	0.0	0.0	**10.0**	0	0	**10**
	sci	0.0	0.0	6.0	0.0	0.0	0.0	0	0	0
	scin	x	x	0.0	x	x	0.0	x	x	0

Table 1: Performance scores for each compilation-solving configuration across planning domains. For OneShot and AnyTime settings, scores are $\sum(C^*/C)$, where C is the found plan cost and C^* is the best known cost; unsolved instances score 0. For Optimal settings, scores represent instances solved. "x" indicates untested configurations. Solvers: F = Fast-Downward, S = Symk, E = ENHSP.

Overall, results show that we can generate models that scale at least as well as handcrafted models, with substantial advantages in several domains, particularly when paired with appropriate planners. Crucially, the automation provided by our approach enables rapid exploration of different modeling choices, allowing us to identify models that are meaningfully more performant than their handcrafted counterparts without requiring manual reformulation.

6 Conclusions and Further Work

We presented a framework for modeling planning problems using high-level data structures and operations, implemented as extensions to the Unified Planning library. Our approach supports multidimensional arrays, sets, bounded integer parameters in actions, integer range variables, and Boolean counting expressions, which are features that arise naturally in many planning domains but are cumbersome to express directly in PDDL.

We developed six composable compilers that transform high-level specifications into planner-compatible representations. This enables automatic exploration of multiple PDDL encodings from a single source model, with different trade-offs in solving time, memory usage, and plan quality. Our framework also treats numeric planning as a first-class citizen, supporting it as both source and target in transformations, a distinction from prior work. Our experiments show that compiled models can match or outperform handcrafted PDDL models, with different compilation strategies excelling on different problems, validating the utility of automatic reformulation exploration.

As further work, additional compilers for both current and new data types can be added. In addition, given the observed diversity, we would like to automate the selection of the best performing strategy and solver available.

Acknowledgments

This work has been partially funded by MCIN/MICIU/AEI/10.13039/ 501100011033 and by ERDF A way of making Europe (grants PID2021-122274OB-I00 and PID2024-157625OB-I00).

References

Alarnaouti, D.; Baryannis, G.; and Vallati, M. 2023. Reformulation techniques for automated planning: a systematic review. *Knowl. Eng. Rev.*, 38.

Asai, M.; and Fukunaga, A. 2015. Solving large-scale planning problems by decomposition and macro generation. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, volume 25, 16–24.

Barták, R.; and Vodrázka, J. 2015. The effect of domain modeling on efficiency of planning: Lessons from the No-mystery domain. In *TAAI*, 433–440.

Bonassi, L.; Percassi, F.; and Scala, E. 2025. Towards Practical Classical Planning Compilations of Numeric Planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 26472–26480. Issue: 25.

Choi, T.; Ko, R. K.; Saha, T.; Scarsbrook, J.; Koay, A. M.; Wang, S.; Zhang, W.; and St Clair, C. 2021. Plan2Defend: AI Planning for Cybersecurity in Smart Grids. *2021 IEEE PES Innovative Smart Grid Technologies-Asia (ISGT Asia)*, 1–5.

Chrpa, L. 2020. Modeling Planning Tasks: Representation Matters. In *Knowledge Engineering Tools and Techniques for AI Planning*, 107–123.

Davesa, C.; Espasa, J.; Miguel, I.; and Villaret, M. 2025. Undefinedness in Planning. In *The 24th International Workshop on Constraint Modelling and Reformulation (ModRef)*.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic Attachments for Domain-Independent Planning Systems. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS)*, 114–121.

ehajdini. 2019. Rush Hour - PDDL. GitHub repository.

Elahi, M.; and Rintanen, J. 2022. Planning with Complex Data Types in PDDL. *CoRR*, abs/2212.14462.

Espasa, J.; Gent, I. P.; Miguel, I.; Nightingale, P.; Salamon, A. Z.; and Villaret, M. 2024a. Cross-Paradigm Modelling: A Study of Puzznic. In *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, 89–95.

Espasa, J.; Miguel, I.; Nightingale, P.; Salamon, A. Z.; and Villaret, M. 2024b. Plotting: a case study in lifted planning with constraints. *Constraints An Int. J.*, 29(1-2): 40–79.

Frisch, A. M.; and Stuckey, P. J. 2009. The Proper Treatment of Undefinedness in Constraint Languages. In Gent, I. P., ed., *Principles and Practice of Constraint Programming - CP*, volume 5732 of *Lecture Notes in Computer Science*, 367–382. Springer.

Gates, W. H.; and Papadimitriou, C. H. 1979. Bounds for Sorting by Prefix Reversal. *Discrete Mathematics*, 27(1): 47–57.

Geffner, H. 2003. PDDL 2.1: Representation vs. Computation. *J. Artif. Intell. Res.*, 20: 139–144.

Gerevini, A. E. 2020. An Introduction to the Planning Domain Definition Language (PDDL): Book review. *Artif. Intell.*, 280: 103221.

Gerevini, A. E.; Percassi, F.; and Scala, E. 2024. An effective polynomial technique for compiling conditional effects away. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 20104–20112.

Gigante, N.; and Scala, E. 2023. On the Compilability of Bounded Numeric Planning. In *IJCAI*, 5341–5349.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 22, 65–73.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26: 191–246.

IPC - Classical Track. 2023. Labyrinth Domain.

Karpas, E.; and Magazzeni, D. 2020. Automated planning for robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 3: 417–439.

Marrella, A. 2019. Automated planning for business process management. *Journal on data semantics*, 8(2): 79–98.

Micheli, A.; Bit-Monnot, A.; Röger, G.; Scala, E.; Valentini, A.; Framba, L.; Rovetta, A.; Trapasso, A.; Bonassi, L.; Gerevini, A. E.; Iocchi, L.; Ingrand, F.; Köckemann, U.; Patrizi, F.; Saetti, A.; Serina, I.; and Stock, S. 2025. Unified Planning: Modeling, manipulating and solving AI planning problems in Python. *SoftwareX*, 29: 102012.

Palacios, H.; and Geffner, H. 2009. Compiling Uncertainty Away in Conformant Planning Problems with Bounded Width. *J. Artif. Intell. Res.*, 35: 623–675.

Percassi, F.; and Gerevini, A. E. 2019. On Compiling Away PDDL3 Soft Trajectory Constraints without Using Automata. In Benton, J.; Lipovetzky, N.; Onaindia, E.; Smith, D. E.; and Srivastava, S., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS*, 320–328. AAAI Press.

Percassi, F.; Scala, E.; and Vallati, M. 2021. Translations from Discretised PDDL+ to Numeric Planning. In Biundo, S.; Do, M.; Goldman, R.; Katz, M.; Yang, Q.; and Zhuo, H. H., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS*, 252–261. AAAI Press.

Perron, L.; and Didier, F. 2025. OR-Tools CP-SAT Solver.

Potassco Team. 2011. Sokoban domain from the IPC 2011 Benchmark Suite.

Rintanen, J. 2015. Impact of Modeling Languages on the Theory and Practice in Planning Research. In *29th AAAI*, 4052–4056.

Rintanen, J. 2017. Temporal Planning with Clock-Based SMT Encodings. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, 743–749.

Scala, E.; Haslum, P.; and Thiébaux, S. 2016. Heuristics for Numeric Planning via Subgoaling. In Kambhampati, S., ed., *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 3228–3234. IJCAI/AAAI Press.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, volume 31.

Speck, D. 2024. Axioms Support in Unified Planning. https://github.com/speckdavid/up-symk. UP-SymK extension.

Speck, D.; Mattmüller, R.; and Nebel, B. 2020. Symbolic Top-k Planning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI*, 9967–9974. AAAI Press.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fišer, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; Segovia-Aguas, J.; and Seipp, J. 2024. The 2023 International Planning Competition. *AI Magazine*, aaai.12169.