Leveraging CP for Numeric-to-Classical Planning Compilations

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

Numeric planning extends classical planning with integer-valued state variables, supporting compact models for many real-world problems, but restricting which planners can be used. Compiling numeric tasks into classical ones is therefore an attractive way to reuse the mature ecosystem of classical planning, and recent work has shown that, with the right encoding, classical planners can be competitive with native numeric planners. We propose an alternative compilation strategy that delegates arithmetic reasoning to a constraint-programming solver at compilation time. From each of these assignments, a new instantiated action with plain Boolean preconditions and effects is produced. We implement this idea in two compilations: *Integer Representation* (IR), which encodes integer values as objects, and *Logarithmic Representation* (LR), which encodes them as bits. Both compilations rely on a per-instance bound analysis that can produce tighter compiled problems than the uniform domain bounds used in prior work. On the numeric domains of the latest International Planning Competition, both IR and LR compilations are competitive with prior classical-planning compilations.

1 Introduction and Related Work

Numeric planning extends classical planning with state variables that take numeric values, allowing compact and natural models of resources, capacities, and quantities that pervade real-world problems (Fox and Long 2003). Formally, a numeric planning problem is a tuple $\Pi = \langle \mathcal{F}, \mathcal{N}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{F}$ is a set of Boolean fluents, $\mathcal{N}$ is a set of numeric fluents, $\mathcal{A}$ is a set of actions, $\mathcal{I}$ is the initial state, and $\mathcal{G}$ is the goal. A state is a valuation over $\mathcal{F} \cup \mathcal{N}$ that assigns each fluent a value in its domain. Each action $a = (\mathit{Pre}, \mathit{Eff}) \in \mathcal{A}$ consists of a precondition Pre, a conjunction of literals over $\mathcal{F}$ and (in)equalities over $\mathcal{N}$ (e.g., $f \geq 5$) that must hold for a to be applicable, and an effect Eff, a set of assignments $\langle v, \mathit{exp} \rangle$ specifying the next-state value (exp) of each updated fluent (v); for example, $\langle f, f + 1 \rangle$ encodes `increase` f 1. Executing a in a state s yields the state $a(s)$ in which the fluents touched by Eff take the new values and the rest are unchanged. A goal $\mathcal{G}$ is a conjunction of literals over $\mathcal{F}$ and (in)equalities over $\mathcal{N}$. A sequential plan is a sequence $a_1; \ldots; a_n$ such that $a_n(\ldots(a_1(\mathcal{I}))\ldots) \models \mathcal{G}$. We focus on the bounded integer fragment, in which every numeric fluent ranges over a finite integer domain; on this fragment, numeric planning is decidable (Gigante and Scala 2023). Over this simple representation of the problem, we will also consider a lifted representation (Corrêa and Giacomo 2024), allowing parameterized families of variables and actions with finite types. In addition, to aid our transformations we will also make use of conditional effects (Nebel 2000) and axioms (Thiébaux, Hoffmann, and Nebel 2005).

Native numeric planners such as ENHSP (Scala, Haslum, and Thiébaux 2016) can solve such tasks directly. At the same time, decades of work on classical planning have produced a rich and mature ecosystem of heuristics, search algorithms, and engineering-grade implementations, and compiling numeric problems into classical ones is an attractive way to bring this broad toolbox to bear on numeric domains. Recent work has made this idea concrete: Gigante and Scala (2023) showed that bounded numeric planning admits a polynomial-size compilation into classical planning that preserves plan length, and Bonassi, Percassi, and Scala (2025) subsequently introduced three practical compilations within the Unified Planning (UP) framework (Micheli et al. 2025): ONE-HOT, which encodes each integer value as a Boolean fluent, and two binary encodings, BLAST and BLAST$_{\mathcal{X}}$, which represent integer values as bit vectors and simulate arithmetic through conditional effects and axioms. Their experiments show that with the right representational choices, classical planners can become competitive with native numeric ones. A common feature of these compilations is that they encode arithmetic *propositionally* inside the compiled actions, materializing operations such as addition as a full adder over Boolean fluents.

We propose a different strategy: using CP-SAT, the Google OR-Tools constraint programming solver (Perron and Furnon 2024), we enumerate at compilation time all consistent value assignments to the fluents that each action touches. Each solution becomes an instantiated action whose preconditions and effects are plain Booleans, with no remaining trace of the original arithmetic. The compiled problem can then be solved by any classical planner supporting axioms (Thiébaux, Hoffmann, and Nebel 2005), a feature we exploit to encode complex goals.

We implement this idea in two compilations, both implemented as compilation passes within UP: *Integer Rep-*

resentation (IR), which encodes integer values as objects, and *Logarithmic Representation* (LR), which encodes them as bits. Both rely on a per-instance bound analysis that can produce tighter compiled problems than the uniform domain bounds used in prior work. We evaluate the two compilations on the numeric domains of the latest International Planning Competition (IPC-23) (Taitler et al. 2024), comparing against the three compilations of Bonassi, Percassi, and Scala (2025) and the native numeric planner ENHSP. Our results show that delegating arithmetic to CP-SAT at compile time is not only a practical design choice, but also competitive with existing approaches.

2 Compilations

We present two compilations from numeric to classical planning, both implemented as compilation passes within the UP framework. Our goal is to produce, for each numeric instance, an equivalent classical planning problem in which preconditions and effects contain no arithmetic. The key idea is to delegate arithmetic reasoning to OR-Tools CP-SAT at compilation time: for every action containing arithmetic, CP-SAT enumerates all consistent value assignments to the fluents involved, and from each solution we instantiate a particular classical action with the corresponding values, yielding plain Boolean preconditions and effects.

We first describe a naive version of this compilation (Section 2.1) and then present a sequence of refinements that make it practical (Section 2.2).

2.1 Naive compilation

The two compilations, *Integer Representation* (IR) and *Logarithmic Representation* (LR), share the same pipeline and differ in how integer fluents are represented and, as a consequence, in how some effects are handled (Sections 2.3 and 2.4). Before compilation, we eliminate any numeric fluent that does not appear in preconditions, goals, or as the source of any other fluent's value: such fluent values do not affect plan validity. We say that an arithmetic expression is *compound* if it involves arithmetic operators or numeric comparisons. The naive compilation transforms the numeric problem into three steps:

Fluent transformation Each integer fluent is replaced by a set of object fluents (in IR) or Boolean fluents that represent the bits of the numeric fluent (in LR). Non-integer fluents are copied unchanged.

Action transformation For each action a containing any compound arithmetic expression, we build a constraint satisfaction problem (CSP) with all its preconditions, effects, and bounds of the numeric fluents occurring in the action. CP-SAT enumerates every assignment to the fluents and parameters involved that satisfy the built CSP. From each CP-SAT solution σ we generate an instantiation of a, denoted by a_σ, by 1) replacing the precondition of a with a conjunction of the assignments stated in σ; 2) replacing the effects $\langle v, e\rangle$ by $\langle v, \sigma(e)\rangle$, where $\sigma(e)$ is the evaluation of e under σ, with expressions resolved and simplified at compilation time whenever possible. Actions without any compound arithmetic expressions are just updated with their fluents transformed into the chosen representation.

For instance, consider an action $a = (f \geq 3, \langle f, f-1\rangle)$, where f has bounds $[0, 5]$. CP-SAT enumerates the assignments $f \in \{3, 4, 5\}$, resulting in three instantiated actions $a_{f=3}, a_{f=4}, a_{f=5}$. Each instantiated action has a precondition that fixes the value of f and an assignment effect that sets f to its decremented value (e.g., $a_{f=3} = (f = 3, \langle f, 2\rangle)$).

Goal transformation We deal with goals that are a conjunction of subgoals. Atomic subgoals, consisting of a single Boolean fluent, are left unchanged. All other subgoals are compiled into derived Boolean fluents via PDDL axioms as follows: For each non-atomic subgoal, we distinguish two cases. If it contains no compound arithmetic expressions, we introduce an axiom whose body is the subgoal directly. If it does contain compound arithmetic expressions, we construct a CSP from the subgoal together with the bounds of the numeric fluents it involves. We then enumerate all satisfying assignments using CP-SAT and introduce an axiom whose body is the disjunction of those assignments. In both cases, the original subgoal is replaced by its corresponding derived fluent. The resulting goal is therefore a conjunction of atomic subgoals and derived fluents. We found empirically that this representation benefits the classical planner.

This naive scheme is correct, but produces compiled problems much larger than necessary.

2.2 Refinements

We introduce two refinements that reduce the size of the compiled problem while preserving equivalence. Algorithm 1 shows the resulting pipeline.

Integer fluent equalities Treating an equality between two integer fluents $f = g$ as arithmetic would force CP-SAT to enumerate all pairs of consistent values, which could lead to a blow-up in the size of the compiled problem. We exclude such equalities from the *arithmetic* category and keep them as direct fluent comparisons between their corresponding representations: object-typed in IR, bit-by-bit in LR.

Classifying effects into dependent or independent In the naive scheme, every fluent appearing in an effect of an action with arithmetic is instantiated by the solutions given by CP-SAT, even the fluents of the effects whose next-state value has no relation to any precondition. Consider an action with the precondition $f > 3$ and effects $\langle f, f-1\rangle$ and $\langle g, g+1\rangle$, where f and g are integer fluents. The naive compilation produces $|dom(f)| \times |dom(g)|$ instantiated actions, while only the values of f are actually constrained by the precondition. This is wasteful: CP-SAT multiplies solutions by the range of fluents that appear only in the effects, producing many instantiated actions that differ only in values irrelevant to the precondition. To address this, we classify the effects of each action as *dependent* if some numeric fluent appearing in the next-state value also appears in some precondition, and *independent* otherwise.

This classification opens the door to obtaining a more succinct way of instantiating actions. The two compilations ex-

Algorithm 1: Numeric-to-Classical Compilation

Require: Numeric planning problem $\Pi = \langle \mathcal{F}, \mathcal{N}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$
Ensure: Classical planning problem Π'

$\Pi' \leftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$
▷ *Fluent transformation*
for all $f \in \mathcal{F} \cup \mathcal{N}$ **do**
  if f is a bounded integer fluent **then**
    $[lb, ub] \leftarrow$ BOUNDSOF(f)
    IR: add object-typed fluent f' over $\{\mathtt{n}_{lb}, \ldots, \mathtt{n}_{ub}\}$ to Π'
    LR: add $k = \lceil \log_2(ub - lb + 1) \rceil$ Boolean fluents $f_0, \ldots, f_{k-1}$ to Π'
  else
    add f to Π'
  end if
end for
▷ *Action transformation*
for all $a \in \mathcal{A}$ **do**
  if a has no compound arithmetic expressions **then**
    add a to Π' with transformed fluents
  else
    $dep\_eff, ind\_eff \leftarrow$ CLASSEFF(a)
    $arith\_pre, non\_arith\_pre \leftarrow$ CLASSPREC(a)
    IR: $\Sigma \leftarrow$ CPSATENUM($arith\_pre, dep\_eff$)
    LR: $\Sigma \leftarrow$ CPSATENUM($arith\_pre, dep\_eff \cup ind\_eff$)
    for all $\sigma \in \Sigma$ **do**
      create instantiated action a_σ
      add non_arith_pre to a_σ
      IR: expand ind_eff as conditional effects
      add a_σ to Π'
    end for
  end if
end for
▷ *Goal transformation*
for all $g \in \mathcal{G}$ **do**
  if g is atomic **then**
    add g to Π''s goal
  else
    ▷ *Axiom body: CP-SAT solutions for g if g is arithmetic*
    add derived fluent AXIOMFOR(g) to Π''s goal
  end if
end for
return Π'

ploit it differently: IR removes independent effects from the CP-SAT call and expands them as conditional effects on the instantiated actions, which is compact under an object-based representation. LR keeps independent effects inside the CP-SAT call, since expanding them as conditional effects over bit assignments would be too costly for the classical planner.

2.3 Integer Representation (IR)

IR maps each integer fluent with domain $[lb, ub]$ to an object-typed fluent over a new user-type `Number`, where objects $\mathtt{n}_i$ represent integer values $i \in [lb, ub]$. A global range covering all integer fluents in the problem is precomputed statically, with one object per used value to keep the set compact.

Because integer values are represented as objects, independent effects are expanded as conditional effects. For example, with f in $[0, 3]$ and the effect $\langle f, f + \delta \rangle$, IR generates the following conditional effects: $f' = \mathtt{n}_0 \triangleright \langle f', \mathtt{n}_1 \rangle$, $f' = \mathtt{n}_1 \triangleright \langle f', \mathtt{n}_2 \rangle$, and $f' = \mathtt{n}_2 \triangleright \langle f', \mathtt{n}_3 \rangle$.

After IR, an additional pass using the UP USERTYPEFLUENTSREMOVER compiler is required to eliminate the `Number` object-typed fluents, since most classical planners do not natively support fluents of user types.

2.4 Logarithmic Representation (LR)

LR maps each integer fluent f with domain $[lb, ub]$ to $k = \lceil \log_2(ub - lb + 1) \rceil$ Boolean fluents $f_0, \ldots, f_{k-1}$. The value v is encoded as the binary representation of $v - lb$, shifting the domain to start at zero. For example, with f in $[2, 5]$, LR uses $k = 2$ Boolean fluents f_0, f_1. The value $v = 4$ is encoded as the binary representation of $4 - 2 = 2$, namely $(f_0, f_1) = (0, 1)$.

2.5 Comparison to prior compilations

Our two compilations relate naturally to the three encodings of Bonassi, Percassi, and Scala (2025): IR adopts an object-based representation similar to their ONE-HOT, while LR adopts a binary representation similar to BLAST and BLAST$_\mathcal{X}$. The key difference lies in how arithmetic is handled. BLAST and BLAST$_\mathcal{X}$ encode each numeric effect as a set of conditional effects (and axioms, in BLAST$_\mathcal{X}$) that explicitly simulate a full adder over the bit representation. In contrast, we delegate arithmetic reasoning to CP-SAT at compilation time: each new action is instantiated from a single CP-SAT solution and contains plain Boolean preconditions and effects. The second difference concerns fluent bounds. We compute these bounds per instance through a static analysis of the PDDL: for each numeric fluent, the bound is the maximum of its initial value, any threshold appearing in the goal, and the largest constant operand of its modifying actions. This is a conservative approximation of the reachable values from the initial state of each instance: smaller instances of a domain produce smaller bounds and more compact compiled problems. By contrast, Bonassi, Percassi, and Scala (2025) use uniform domain bounds derived from the largest constant appearing in any instance of the domain.

3 Experimental Evaluation

The experiments were conducted on an AMD EPYC 9755 processor at 2.70 GHz, with a time limit of 1800 seconds and a memory limit of 8 GB per run (for joint compilation and solving), matching the experimental setup of (Bonassi, Percassi, and Scala 2025).

We evaluate our compilations on the numeric domains from the latest International Planning Competition (IPC-23). This benchmark contains a mix of strongly numeric (SN) and mildly numeric (MN) domains, based on the proportion of numeric variables relative to the total number of variables, indicated by $\mathfrak{N}$. We compare our two compilations, IR and LR, against the three compilations of Bonassi, Percassi, and Scala (2025): ONE-HOT (O), BLAST (B) and BLAST$_\mathcal{X}$ (B$_\mathcal{X}$), which are obtained by running their publicly available implementation. We also include ENHSP with the M(3h‖3n) configuration from the work by (Chen and Thiébaux 2024) as a reference native numeric planner.

Table 1: Coverage results and domain features. $\mathfrak{N}$: domain numericity, $|A_{\mathbb{Z}}|$: arithmetic lifted actions, $|F_{\mathbb{Z}}|$: numeric fluents, max: largest absolute integer constant in the input problem. $|A_{\mathbb{Z}}|$, $|F_{\mathbb{Z}}|$, and max reported as a range [min, max] when they vary across the 20 instances, otherwise as a single number. Bold indicates overall best, underline the best among compilations. *Domain provided as PDDL with grounded actions.

					LAMA-First					
Domain (#)	$\mathfrak{N}$	$\lvert A_{\mathbb{Z}}\rvert$	$\lvert F_{\mathbb{Z}}\rvert$	max	O	B	B$_{\mathcal{X}}$	IR	LR	M
BGrouping (20)	1.0	4	2	[5, 100]	2	2	2	**19**	11	**18**
Counters (20)	1.0	2	1	[8, 80]	6	5	5	**15**	6	10
Watering (20)	1.0	10	7	[29, 124]	0	0	4	2	0	**20**
Farmland (20)	1.0	2	2	[1400, 12600]	0	0	4	0	0	**20**
MTrader (20)	0.94	4	3	10000	0	0	0	0	0	**20**
Sailing (20)	0.88	8	2	[740, 966]	0	0	5	0	0	**20**
Pathways* (20)	0.7	[44, 767]	[27, 447]	[4, 8]	0	0	**13**	3	3	4
Sugar* (20)	0.64	[126, 167]	[36, 41]	[15, 30]	**18**	13	14	14	10	13
Σ_{SN} (160)	–	–	–	–	23	20	47	53	30	**125**
Settlers (20)	0.41	24	6	10	16	**17**	**17**	14	**17**	5
Expedition (20)	0.39	4	2	1000	0	0	4	0	2	**7**
HPower (20)	0.32	2	3	[20200, 41000]	0	0	8	0	0	**20**
Rovers (20)	0.12	9	1	80	6	6	15	14	15	**16**
MPrime (20)*	0.08	[68, 17595]	[6, 37]	[3, 12]	**20**	**20**	**20**	10	11	19
Delivery (20)	0.02	2	1	[5, 20]	19	**20**	**20**	**20**	**20**	**20**
Σ_{MN} (120)	–	–	–	–	64	63	84	58	65	**87**
Σ (280)	–	–	–	–	87	83	131	111	95	**212**

The runtime of each compilation method includes both the compilation time and the planner search time. In all experiments based on compilations from numeric to classical planning, we use Fast Downward (Helmert 2006) (LAMA-first) (Richter and Westphal 2010) as the classical planner.

Table 1 summarizes key domain characteristics that help explain the results: the number of arithmetic actions ($|A_{\mathbb{Z}}|$), the number of numeric fluents ($|F_{\mathbb{Z}}|$) and the maximum fluent bound per instance. The relative coverage of IR and LR depends less on the strongly/mildly numeric distinction than on two structural properties of the instance: the size of the fluent bounds and the number of lifted arithmetic actions.

In domains where both factors are manageable, our compilations achieve strong coverage. BGrouping and Counters combine very small bounds with compact action sets, allowing IR to solve nearly all instances. On these domains, per-instance bounds are particularly effective: smaller instances of a domain produce smaller bounds, keeping the compiled problem compact. In mildly numeric domains, BLAST$_{\mathcal{X}}$ leads overall, but IR and LR remain competitive in Settlers, Rovers, and Delivery, where the combined size of bounds and actions stays within reach of the CP-SAT enumeration.

The domains where IR and LR compilations fail reveal two distinct bottlenecks. First, large fluent bounds make CP-SAT enumeration produce problems too large for the planner within the time limit. This affects Farmland, HPower, Sailing, MTrader, Watering, and Expedition, where IR and LR solve few or no instances. Second, the number of actions with numeric fluents determines how many times CP-SAT must run during compilation. On MPrime, these grow substantially with instance size, leading to combinatorial blow-up despite small bounds. Pathways and Sugar show a similar pattern on a smaller scale.

4 Conclusions and Future Work

We presented two compilations from numeric to classical planning that delegate arithmetic reasoning to a constraint solver at compilation time, achieving competitive coverage on the IPC-23 numeric benchmarks.

A reachability-based bound analysis (Kuroiwa, Shleyfman, and Beck 2023) would produce tighter per-instance bounds and make the compilations tractable in more domains. A second direction for future work focuses on how arithmetic comparisons are encoded in LR. Currently, CP-SAT expands comparisons such as $a > b$ into the disjunction of all bit assignments that satisfy them, which grows exponentially with the bit width of the fluents involved. A more efficient encoding would compare the bits of a and b, from the most significant onward, producing a much smaller encoding.

Finally, our compilations apply the same refinements to every domain and instance, but the best strategy may vary. For example, when bounds are small, enumerating all consistent value assignments via CP-SAT is cheap; when bounds are large, encoding via derived fluents may scale better. A natural direction is to make these decisions adaptive, choosing per domain, or even per action, which refinement to apply, based on a static analysis of the instance.

More broadly, our approach opens the door to expressing constraints that are difficult or impossible to handle with current planners, by leveraging the full expressivity of the constraint solver at compilation time.

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

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

Chen, D. Z.; and Thiébaux, S. 2024. Novelty Heuristics, Multi-Queue Search, and Portfolios for Numeric Planning. *Proceedings of the International Symposium on Combinatorial Search*, 17(1): 203–207.

Corrêa, A. B.; and Giacomo, G. D. 2024. Lifted Planning: Recent Advances in Planning Using First-Order Representations. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*, 8010–8019. ijcai.org.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res.*, 20: 61–124.

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

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

Kuroiwa, R.; Shleyfman, A.; and Beck, J. C. 2023. Extracting and Exploiting Bounds of Numeric Variables for Optimal Linear Numeric Planning. In *ECAI*, 1332–1339.

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.

Nebel, B. 2000. On the Compilability and Expressive Power of Propositional Planning Formalisms. *J. Artif. Intell. Res.*, 12: 271–315.

Perron, L.; and Furnon, V. 2024. OR-Tools.

Richter, S.; and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

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.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fiser, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; Segovia-Aguas, J.; and Seipp, J. 2024. The 2023 International Planning Competition. *AI Mag.*, 45(2): 280–296.

Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of PDDL axioms. *Artif. Intell.*, 168(1-2): 38–69.