Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)

On-Demand Mutex Constraints for Numeric Planning as SMT

Mustafa F. Abdelwahed[1], Miquel Bofill[2], Joan Espasa[1], Mateu Villaret[2]

[1]University of St Andrews, St Andrews, UK
[2]Universitat de Girona, Girona, Spain
ma342@st-andrews.ac.uk, miquel.bofill@udg.edu, jea20@st-andrews.ac.uk, mateu.villaret@udg.edu

Abstract

Planning as Satisfiability Modulo Theories (SMT) has been shown to be a competitive approach for solving numeric planning problems, leveraging the expressiveness of SMT to handle numeric variables and constraints. Being able to set more than one action in a single timestep is crucial for performance, allowing short makespan plans and reducing the number of solver queries required. However, encoding action interference constraints can cause significant formula size blow-up, with mutex constraints dominating the encoding size and limiting scalability. In this paper we present a tool that implements a lazy approach to handling action interference in planning as SMT, exploiting recent developments in solver technology to integrate custom propagators directly into the SMT solver's search process. Rather than eagerly encoding all interference constraints upfront, our propagators add mutex clauses on demand, generating them only when the solver attempts to execute two mutually interfering actions in parallel. Experimental evaluation on standard numeric planning benchmarks demonstrates that our lazy approach is significantly more scalable than the eager approach while maintaining correctness, leading to improved solver performance and making this tool a competitive planner in the numeric setting.

Code — https://github.com/udg-lai/ICAPS26-Mutex-On-Demand-SMT

Introduction and Related Work

Planning as SAT (Kautz, Selman, and others 1992) is a classic approach for solving planning problems by translating planning problem instances into propositional Boolean formulas. This approach has been extended to use Satisfiability Modulo Theories (SMT) (Barrett and Tinelli 2018), which generalizes SAT by supporting reasoning over first order theories, making it particularly suitable for numeric planning problems with continuous or discrete numeric variables. Over the years, numerous systems have adopted planning as SMT (Hoffmann et al. 2007; Cardellini, Giunchiglia, and Maratea 2024; Bofill, Espasa, and Villaret 2016; Scala et al. 2016; Leofante 2023; Sapena, Onaindia, and Marzal 2024; Tosello, Valentini, and Micheli 2025).

A critical design decision in planning as SAT/SMT is whether to allow multiple actions to execute in parallel within a single timestep. Permitting parallel actions significantly improves performance by reducing plan makespan and the number of solver queries required. However, this requires the explicit encoding of *interference constraints*, commonly called mutex constraints, to prevent interfering actions from executing simultaneously. A standard approach (Rintanen, Heljanko, and Niemelä 2006) is to precompute an interference graph between all pairs of actions and encode all possible interference constraints upfront, sometimes leading to a formula size blow-up that can dominate the encoding and limit scalability, particularly for domains with many potentially interfering actions.

A different approach is possible using *lemmas on demand* (De Moura, Rueß, and Sorea 2002; Barrett, Dill, and Stump 2002). Rather than eagerly encoding all constraints, we can start with a relaxation of the problem and iteratively refine it by adding constraints only when violated by candidate solutions. Recent advances in SAT/SMT solver technology, notably IPASIR-UP (Fazekas et al. 2023) and Z3's user propagator interface (Bjørner, Eisenhofer, and Kovács 2023), now enable fine-grained integration of custom reasoning procedures directly into the solver's search process, making lazy constraint generation practical and efficient.

In this paper, we present a lazy approach for handling action interference in planning as SMT, exploiting custom propagators to monitor the solver's assignments and generate mutex clauses on demand. Instead of eagerly computing and encoding the full interference graph as done in (Bofill, Espasa, and Villaret 2016), our propagators allow us to lazily compute interferences and add mutex constraints only when the solver attempts to schedule two interfering actions in the same timestep. This strategy reduces formula size dramatically while maintaining soundness and completeness. We concentrate on numeric planning, where determining interference between actions is more difficult than in the propositional case, and hence our approach can lead to a greater improvement. Our main contributions are a novel planning-as-SMT solver for PDDL2.1 (excluding temporal extensions and metric optimisations) that uses custom propagators to generate mutex constraints lazily rather than encoding them upfront, and an experimental evaluation demonstrating that this lazy approach significantly outperforms the eager baseline in both scalability and solving performance.

Background

A numeric planning problem can be defined as a tuple (V, A, I, G), where V is a set of state variables (either numeric or Boolean), A is a set of actions, I is the initial state and G is the goal. A state is a valuation over V, i.e., a function mapping each variable $v \in V$ to a value in its domain ($\{true, false\}$ in the Boolean case). The goal is a set of states, usually defined as a set of propositions that a goal state must satisfy. Actions $a = \langle Pre, Eff \rangle \in A$ are defined as pairs of preconditions and effects. Preconditions describe which are the requirements on the state to execute the action, whilst effects describe how the state is updated after its execution. Preconditions and effects are typically given as sets of literals in the Boolean case. In the numeric setting, preconditions are generalized to sets of (in)equalities and effects are generalized to sets of assignments $\langle v, exp \rangle$, where v is a variable and exp is an expression of the corresponding type. For example, increasing a variable v by one is represented by the pair $\langle v, v + 1 \rangle$, indicating that $v + 1$ is the value that v will hold in the next state. The state resulting from executing an action a in state s is denoted by $a(s)$. The new state is defined by assigning new values to the variables according to the effects, and retaining the values of the variables not updated by any of the effects. A sequential plan of length n for a planning problem is a sequence of actions $a_1; a_2; \ldots; a_n$ such that $a_n(\ldots(a_2(a_1(I)))\ldots) \models G$.

In the planning as SAT/SMT approach, a planning problem is solved by considering a sequence of formulas ϕ_0, ϕ_1, ϕ_2, …, where ϕ_i encodes the existence of a plan of length i. The solving procedure proceeds by testing the satisfiability of ϕ_0, ϕ_1, ϕ_2, and so on, until a satisfiable formula ϕ_n is found. In this approach, variables need to be replicated for each timestep, e.g., a^t denotes if action a is executed at time t. Then, the standard encoding of a plan of length T goes as follows (Rintanen 2021). First of all, it is stated that the execution of an action at timestep t implies its preconditions to be satisfied at time t and its effects to be satisfied at time $t + 1$: for every $a = \langle Pre, Eff \rangle \in A$ and $t \in 0..T - 1$, we have $a^t \rightarrow Pre^t$ and $a^t \rightarrow Eff^{t+1}$, where Pre^t and Eff^{t+1} denote the corresponding encodings for the precondition and the effects (with equalities like $v^{t+1} = v^t + 1$ for assignments $\langle v, v + 1 \rangle$) on the time-indexed state variables. Moreover, a change in the value of a state variable v at timestep $t + 1$ can occur only if an action that can change this value is executed at timestep t: for every variable $v \in V$ and $t \in 0..T - 1$, we have the frame axiom $v^t \neq v^{t+1} \rightarrow \bigvee\{a^t \mid a = \langle Pre, Eff \rangle \in A, \langle v, exp \rangle \in Eff\}$. To ensure that exactly one action is executed in each timestep t, an *exactly-one* constraint over the action execution variables of each timestep is also added. Finally, it is stated that the initial state holds at time 0 and that the goal holds at time T.

A parallel plan of length n can be defined similarly to a sequential plan where, instead of having a sequence of actions, we have a sequence of sets of actions $\sigma_1; \sigma_2; \ldots; \sigma_n$ such that $order(\sigma_1); order(\sigma_2); \ldots; order(\sigma_n)$ is a sequential plan, where $order(\sigma_i)$ is an ordering function which serializes the set σ_i into a sequence of actions. Actions in the same set σ_i are said to occur in parallel. Then, the planning as SAT/SMT approach for parallel plans is the same as for sequential plans, but now each formula encodes the existence of a parallel plan of certain length where, at each step, several actions can occur simultaneously (the *exactly-one* constraint over the action execution variables of each timestep is removed). However, it is required that actions scheduled in parallel can still be serialized.

In the $\forall$-*step semantics* (Kautz and Selman 1996) parallel actions can be ordered with respect to any total order, i.e., no two actions a_j, a_k in each σ_i are interfering (that is, executing a_j neither falsifies the precondition of a_k nor changes any of its effects, and vice versa). In the numeric setting, a less restrictive notion of interference can be defined based on the commutativity of effects on numeric variables (Bofill, Espasa, and Villaret 2021). To ensure serialization of actions scheduled in parallel, a mutex clause $\neg a_j^i \vee \neg a_k^i$ is added for each pair of interfering actions a_j and a_k and timestep i. The $\exists$-*step semantics* (Dimopoulos, Nebel, and Koehler 1997; Rintanen, Heljanko, and Niemelä 2006) weakens the $\forall$-step requirements, by only requiring the existence of some ordering of the actions that results in a valid sequential plan.

There are other semantics, such as the *relaxed* $\exists$*-step semantics* (Wehrle and Rintanen 2007) and the *relaxed relaxed* $\exists$*-step semantics* (Balyo 2013), which progressively relax the applicability requirements on parallel actions.

Interference between actions is usually determined by inspecting the literals that occur in their preconditions and effects at compile time. However, such syntactic checks tend to overestimate interference, especially when numeric functions are present. Recent works (Bofill, Espasa, and Villaret 2021) alleviate this problem by checking interference semantically, looking for (un)satisfiability of interference conditions by calling a SMT solver at compile time. Note that the potential number of interferences between actions is quadratic in the number of actions, so it can result in a quadratic number of mutexes. In numeric planning, when using a semantic notion of interference, the number of interferences can be significantly reduced. Interferences between actions can be represented in a graph.

Definition 1 (Disabling Graph) *Given a planning problem $P = (V, A, I, G)$, a* disabling graph *for P is a directed graph $\mathcal{G} = (V, E)$, where $V = A$ and $(a_j, a_k) \in E$ if a_j interferes with a_k.*

In the previous definition, the notion of interference must be adapted for the semantics considered but, in any case, an edge from a_j to a_k will mean that the action a_j cannot immediately precede a_k in a valid sequential plan. Under the $\forall$-step semantics, a mutex clause is added for every pair of interfering actions, i.e., if $(a_j, a_k) \in E$ or $(a_k, a_j) \in E$, then a mutex $\neg a_j^i \vee \neg a_k^i$ is added for each timestep i. However, under the $\exists$-step semantics, if $\mathcal{G}$ has no cycles, then a set of actions scheduled in parallel can be ordered to form a valid sequential plan according to any topological sort of the reverse of $\mathcal{G}$. Therefore, mutexes can be added only to prevent cycles among actions in the same timestep. To this end, a simple approach is to set an arbitrary order between actions and add a mutex for a pair of actions a_j and a_k if $(a_j, a_k) \in E$ and $a_j < a_k$ (Rintanen, Heljanko, and Niemelä 2006).

Lazy Mutexes and Interferences

The number of mutexes typically dominates the size of the formula. In this section we describe two propagators that avoid this overhead by generating mutexes on-demand during search, rather than encoding them all in advance. The propagators adhere to the API provided by the Z3 SMT solver (Bjørner, Eisenhofer, and Kovács 2023). More concretely, the propagators implement the `fixed` function of the API, receiving a callback whenever the solver decides to assign a value to an action variable a^t, and keep track of the set of active actions $\mathcal{A}^t_{active} = \{a \in A \mid a^t = true\}$ for each timestep t, by using the `push` and `pop` endpoints.

Lazy Forall Propagator Since in the $\forall$-step semantics no two actions in the same timestep can interfere, after each assignment to an action variable a^t to $true$, we check if there is either an outgoing or incoming edge in the disabling graph $\mathcal{G} = (V, E)$ between a and any of the already active actions in timestep t. If that is the case, a conflict is thrown. The solver will then process the conflict, learn a clause and backtrack. Algorithm 1 illustrates this approach.

Algorithm 1: Forall Propagator

Require: Action variable assignment (a^t, v); current active actions $\mathcal{A}^t_{active}$; disabling graph $\mathcal{G} = (V, E)$
Ensure: Conflicts generated if interference detected

if $v = false$ **then**
  return $\emptyset$
end if
$\mathcal{A}^t_{active} \leftarrow \mathcal{A}^t_{active} \cup \{a\}$
for $a_i \in \mathcal{A}^t_{active} \setminus \{a\}$ **do**
  if $(a, a_i) \in E \vee (a_i, a) \in E$ **then**
    generate_conflict$(\{a^t, a^t_i\})$
  end if
end for

Exists Propagator To enforce the $\exists$-step semantics, the exists propagator implements existential quantification semantics: actions can execute in parallel if there exists at least one ordering under which they do not interfere. This is more permissive than forall semantics. The propagator uses cycle detection in the disabling graph to identify when no valid ordering exists among a set of active actions.

After assigning an action variable a^t to $true$, we add it to the set $\mathcal{A}^t_{active}$. Then, given the static (i.e., built at compile time) disabling graph $\mathcal{G} = (V, E)$, we dynamically build the disabling digraph $\mathcal{G}^t = (\mathcal{A}^t_{active}, E^t)$ where $E^t = \{(a_j, a_k) \in E \mid a_j, a_k \in \mathcal{A}^t_{active}\}$. Finally, a conflict is generated if a cycle is detected in $\mathcal{G}^t$.

Algorithm 2 illustrates this approach. A subgraph of the disabling graph $\mathcal{G}$ induced by the active actions at timestep t is built by the **build_disabling_graph** function. Then, the function **detect_cycle_dfs** performs a depth-first search to find cycles. If one exists, no valid serialisation of the active actions is possible, and a conflict is generated with all the actions present in the cycle. Note that an advantage of this approach is that cycles are broken optimally during search. That is, while the eager approach commits to arbitrary mutex constraints upfront, potentially excluding valid parallel executions, this propagator lets the solver explore all possible ways to break the cycles, preserving step-optimality.

Algorithm 2: Exists Propagator

Require: Action variable assignment (a^t, v); current active actions $\mathcal{A}^t_{active}$; disabling graph $\mathcal{G} = (V, E)$
Ensure: Conflict generated if cycle detected

if $v = false$ **then**
  return $\emptyset$
end if
$\mathcal{A}^t_{active} \leftarrow \mathcal{A}^t_{active} \cup \{a\}$
$\mathcal{G}^t \leftarrow$ **build_disabling_graph**$(\mathcal{G}, \mathcal{A}^t_{active})$
$cycle \leftarrow$ **detect_cycle_dfs**$(\mathcal{G}^t)$
if $cycle \neq \emptyset$ **then**
  generate_conflict$(\{a^t_i \mid a_i \in cycle\})$
end if

Lazy Interference In the previous algorithms we assumed that the disabling graph was already precomputed. However, we can remove this restriction and consider different strategies for when this computation needs to occur. We distinguish two approaches: the *eager* and the *lazy* approaches. In the *eager* approach, the disabling graph is computed a priori between all pairs of actions before search begins. This, in turn, enables two possibilities: either the complete set of mutex constraints is added directly to the encoding (the classical methodology adopted by previous work such as (Rintanen, Heljanko, and Niemelä 2006; Bofill, Espasa, and Villaret 2016)), or a propagator can query the precomputed graph during search using constant-time edge lookups. In contrast, the *lazy* approach starts with an empty graph and computes an edge only when the solver assigns two actions to the same timestep. This defers the cost of interference checking to search time, avoiding the upfront quadratic computation and reducing memory usage when only a fraction of action pairs are ever considered together.

Experimental Results

Our planner has been implemented on top the Unified Planning library (Micheli et al. 2025) and the Z3 SMT solver (De Moura and Bjørner 2008). We used an AMD EPYC 7763 64-Core Processor@2.4GHz and, to stress-test scalability, we restricted each task to one CPU core, 1 hour timeout and 2GB of memory. We use the 2023 Numeric IPC domains (Taitler et al. 2024) plus *depots*, *petrobras*, *satellite*, *settlers* and *zenotravel*, capped at 20 instances each. After filtering trivial instances (solved by all planners in under 5 seconds) and unsolvable ones (not solved by any planner), we obtain 417 challenging instances across 24 domains. We compare against ENHSP (Scala et al. 2020), run via Unified Planning with default configuration (GBFS with h^{add}), and Patty (Cardellini, Giunchiglia, and Maratea 2024), as a representative sample of state-of-the-art numeric planners. Since our benchmark extends beyond Patty's original evaluation, its parser encountered 180 errors across several domains, mainly due to unsupported equality predicates be-

tween objects. We nonetheless retain Patty given its relevance and performance.

For both the ∀ and the ∃-step semantics, in our experiments we consider the following variants: **eager** uses the syntactic notion of interference with the disabling graph computed a priori and all mutexes encoded at the start, implementing the method adopted by previous works (Rintanen, Heljanko, and Niemelä 2006; Bofill, Espasa, and Villaret 2016) and serving as a baseline; **lazy** also uses the syntactic notion of interference but computes each interference on demand, starting a graph with no mutexes and adding them during propagation; **eager-semantic-chain** employs the semantic notion of interference and encoding from (Bofill, Espasa, and Villaret 2021), with the disabling graph computed a priori and all mutexes encoded at the start; finally, **lazy-semantic-chain** follows the same approach but with lazy computation. We remark that all **eager** approaches reported do not use the propagators. Table 1

Planner	Solved	Error	TO	MO
forall-eager	113	0	183	121
forall-lazy	142	0	243	32
forall-eager-semantic-chain	137	0	190	90
forall-lazy-semantic-chain	**147**	0	192	78
exists-eager	130	0	197	90
exists-lazy	147	0	234	36
exists-eager-semantic-chain	156	0	187	74
exists-lazy-semantic-chain	**163**	0	177	77
ENHSP	**167**	0	234	16
Patty	134	180	49	54

Table 1: Accumulated results for 417 instances. Solver indicates solved instances; Error refers to parsing errors; TO and MO denote time-outs and memory-outs, respectively.

summarises the results. Our lazy approach shows competitive performance with state-of-the-art numeric planners. The best-performing variant, exists-lazy-semantic-chain, solves 163 instances, only 4 instances behind ENHSP. Patty solves 134 instances, but it should be noted that Patty's parser does not support certain PDDL constructs present in our benchmark set, preventing it from attempting all considered instances. On the domains it can parse, Patty demonstrates strong performance, particularly excelling on families such as *block-grouping* and *hydropower* where its pattern-based approach proves highly effective. The lazy mutex generation consistently improves performance over the eager baseline across all configurations. In the ∃ semantics, the lazy variant solves 147 instances compared to 130 for the eager. The improvement is even more pronounced for the ∀-step semantics (142 versus 113). This pattern holds when combined with semantic interference checking: 163 versus 156 on the ∃-step semantics, and 147 versus 137 with the ∀-step semantics.

An additional advantage of the lazy approach lies in memory consumption, where the eager ∀-step semantics generates 121 memory-outs (MO) compared to only 32 for the lazy approach. Similarly, the lazy ∃-step produces 90 versus 36 memory-outs. This reduction in memory usage explains the coverage improvements: by avoiding the up-

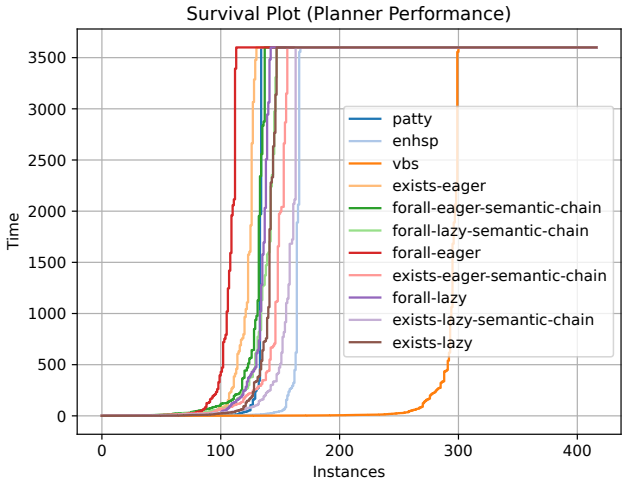

Figure 1: Survival plot showing cumulative solving time across instances, ordered by difficulty. Lazy variants consistently outperform their eager counterparts.

front encoding of the complete interference graph, the lazy approach can handle larger problem instances that would otherwise exhaust available memory. A key advantage of our propagator-based approach is that mutex constraints become part of the solver's conflict analysis and clause learning mechanism, allowing the solver to forget some of the clauses when they are no longer relevant. Figure 1 illustrates the performance characteristics of the different approaches. Our best variant establishes a competitive numeric planner with existing planners, where a virtual best solver (**vbs**) would substantially outperform either in isolation. More concretely, if we consider an oracle that always chooses the best performing planner from our encodings and ENHSP, the resulting planner would solve 300 instances.

When comparing lazy semantic with syntactic eager ∃-step, most domains see only 0.02% to 19% of the mutexes that would be generated eagerly. This reduction is achieved by generating constraints only when needed and eliminating spurious mutexes through semantic interference checking. Note though that most cycles detected by the ∃-step propagator have sizes between 2 and 5 actions.

Conclusions

The main contribution of this work lies in a scalable implementation of established semantics for parallelism (Rintanen, Heljanko, and Niemelä 2006; Bofill, Espasa, and Villaret 2021), achieved through a lazy approach to mutex constraint generation for numeric planning as SMT, exploiting custom propagators to add the interference constraints on demand. Our evaluation demonstrates consistent improvements over eager variants, with dramatic reductions in both mutex constraints generated and memory exhaustion. The lazy approach also preserves step-optimality by deferring cycle-breaking decisions to the solver.

Acknowledgements

This work was 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

Balyo, T. 2013. Relaxing the Relaxed Exist-Step Parallel Planning Semantics. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, 865–871.

Barrett, C.; and Tinelli, C. 2018. Satisfiability Modulo Theories. In Clarke, E. M.; Henzinger, T. A.; Veith, H.; and Bloem, R., eds., *Handbook of Model Checking*, 305–343. Springer International Publishing. ISBN 978-3-319-10574-1 978-3-319-10575-8.

Barrett, C. W.; Dill, D. L.; and Stump, A. 2002. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In Brinksma, E.; and Larsen, K. G., eds., *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, 236–249. Springer.

Bjørner, N.; Eisenhofer, C.; and Kovács, L. 2023. Satisfiability Modulo Custom Theories in Z3. In Dragoi, C.; Emmi, M.; and Wang, J., eds., *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, 91–105. Springer Nature Switzerland. ISBN 978-3-031-24950-1.

Bofill, M.; Espasa, J.; and Villaret, M. 2016. The RANTANPLAN planner: system description. *Knowl. Eng. Rev.*, 31(5): 452–464.

Bofill, M.; Espasa, J.; and Villaret, M. 2021. Relaxing non-interference requirements in parallel plans. *Logic Journal of the IGPL*, 29(1): 45–71. Publisher: Oxford University Press.

Cardellini, M.; Giunchiglia, E.; and Maratea, M. 2024. Symbolic Numeric Planning with Patterns. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI*, 20070–20077. AAAI Press.

De Moura, L.; and Bjørner, N. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08. Springer-Verlag. ISBN 3-540-78799-2.

De Moura, L.; Rueß, H.; and Sorea, M. 2002. Lemmas on Demand for Satisfiability Solvers. *Proc. SAT*, 2: 244–251.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding Planning Problems in Nonmonotonic Logic Programs. In *Recent Advances in AI Planning, Fourth European Conference on Planning (ECP'97)*, volume 1348 of *LNCS*, 169–181. Springer.

Fazekas, K.; Niemetz, A.; Preiner, M.; Kirchweger, M.; Szeider, S.; and Biere, A. 2023. IPASIR-UP: User Propagators for CDCL. In Mahajan, M.; and Slivovsky, F., eds., *26th International Conference on Theory and Applications of Satisfiability Testing, SAT*, volume 271 of *LIPIcs*, 8:1–8:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Hoffmann, J.; Gomes, C. P.; Selman, B.; and Kautz, H. A. 2007. SAT Encodings of State-Space Reachability Problems in Numeric Domains. In *20th International Joint Conference on Artificial Intelligence (IJCAI)*, 1918–1923.

Kautz, H. A.; and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI*, 1194–1201.

Kautz, H. A.; Selman, B.; and others. 1992. Planning as Satisfiability. In *ECAI*.

Leofante, F. 2023. OMTPlan: A Tool for Optimal Planning Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 14(1): 17–23. Publisher: IOS Press.

Micheli, A.; Bit-Monnot, A.; Röger, G.; Scala, E.; Valentini, A.; Framba, L.; Rovetta, A.; Trapasso, A.; Bonassi, L.; and Gerevini, A. E. 2025. Unified Planning: Modeling, manipulating and solving AI planning problems in Python. *SoftwareX*, 29: 102012.

Rintanen, J. 2021. Planning and SAT. In Biere, A.; Heule, M.; Maaren, H. v.; and Walsh, T., eds., *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, 765–789. IOS Press.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13): 1031–1080.

Sapena, O.; Onaindia, E.; and Marzal, E. 2024. A hybrid approach for expressive numeric and temporal planning with control parameters. *Expert Systems with Applications*, 242.

Scala, E.; Ramírez, M.; Haslum, P.; and Thiébaux, S. 2016. Numeric Planning with Disjunctive Global Constraints via SMT. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS*, 276–284. AAAI Press.

Scala, E.; Saetti, A.; Serina, I.; and Gerevini, A. E. 2020. Search-guidance mechanisms for numeric planning through subgoaling relaxation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 226–234.

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.

Tosello, E.; Valentini, A.; and Micheli, A. 2025. Temporal Task and Motion Planning with Metric Time for Multiple Object Navigation. In *AAAI*.

Wehrle, M.; and Rintanen, J. 2007. Planning as Satisfiability with Relaxed Exists-Step Plans. In *AI 2007: Advances in Artificial Intelligence, 20th Australian Joint Conference on Artificial Intelligence, Gold Coast, Australia, December 2-6, 2007, Proceedings*, 244–253.