Formally Verified Certification of Unsolvability of Temporal Planning Problems

David Wang[1] and Mohammad Abdulaziz[1]

King's College London, London, UK
{david.wang,mohammad.abdulaziz}@kcl.ac.uk

Abstract. We present an approach to unsolvability certification of temporal planning. Our approach is based on encoding the planning problem into a network of timed automata, and then using an efficient model checker on the network followed by a certificate checker to certify the output of the model checker. Our approach prioritises trustworthiness of the certification: we formally verify our implementation of the encoding to timed automata using the theorem prover Isabelle/HOL and we use an existing certificate checker (also formally verified in Isabelle/HOL) to certify the model checking result.

Keywords: Temporal planning · Timed automata · Model checking · Unsolvability certification · Formal verification · Isabelle/HOL

1 Introduction

In AI planning, given a model of the world describing its dynamics in terms of *actions* and its *state* using state variables, the aim is to find a sequence of actions that reaches a *goal* from an *initial state*. Since research on this problem started in the 1960s [71], the performance and expressiveness of planning systems have improved substantially, as the planning competitions show [72,41,27,87,83], and state-of-the-art planners now scale to real-world sequential decision problems.

Despite this progress, the trustworthiness of planners' outputs can still be improved, given the complexity of state-of-the-art planning algorithms at an abstract level as well as at an implementation optimisation level. Much of the work in this direction develops *certificate checkers*, i.e. (relatively) small, independent programs that check the output of a planner, such as plan validators [59,49], which confirm that a candidate solution solves a given planning problem. Validators have also been formally verified, i.e. proved correct with respect to a semantics of the planning formalism, for classical planning [4] and for temporal planning [2], which adds action durations on a dense timeline and concurrent execution of plan actions.

Certifying that a problem is unsolvable, or that a plan is optimal, is much harder than validating a candidate plan. A plan is usually succinct and is itself a certificate, checked by executing its steps, while a certificate of unsolvability or optimality can, for most relevant planning formalisms, be exponentially long in the size of the planning task in the worst case (unless NP = PSPACE [7,

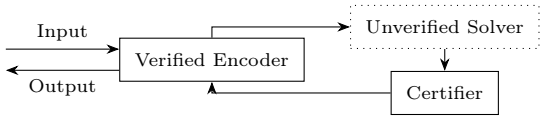


Fig. 1: The planner architecture proposed by Abdulaziz and Kurz [3], combining verified (solid) and unverified (dotted) modules. The input is a grounded planning problem, which is encoded into an automaton by a verified encoder, the automaton is checked by an efficient, unverified model checker, and the result is certified with a verified certificate checker.

Chapter 4]). In practice, such certificates are exponentially long unless they are very carefully designed, like those of Eriksson et al. [34], which can be succinct for large classes of classical planning problems.

We consider certifying unsolvability of temporal planning [39] (see Example 1 and Definition 23). Making state-of-the-art temporal planners produce succinct certificates would require substantial changes to their algorithms and optimised implementations. We thus follow another approach (Figure 1): we encode the planning problem into timed automata (TA) model checking, for which a practical certifying algorithm exists. The certificate then implies unsolvability of the planning problem only if the encoding and its implementation are correct. However, that encoding is complex (Example 2) and can be a source of bugs/untrustworthiness. Although temporal planning, like model checking, is defined on state transition systems, planning languages have richer constructs: durative invariants (i.e. conditions that must hold throughout an action's execution) and numeric fluents with linear comparison guards (Example 1 and Figure 2) cannot be modelled natively in common TA formalisms, e.g. TA networks, which makes an efficient encoding too complex to trust without verification. We therefore formally verify, using an interactive theorem prover (ITP), the correctness of the encoding and its implementation.

More specifically, our contributions are: (Contribution i) a practical encoding of grounded temporal planning problems into networks of timed automata as formalised by Wimmer [89,88], for use with TChecker [56], which combines and extends the encodings of Heinz et al. [50] and Gigante et al. [43] (Section 3), in particular with numeric fluents, whose finite bounds we infer by interval analysis and widening with thresholds (Section 3.1), (Contribution ii) a more general notion of plan validity and unsolvability than earlier encodings, with dense duration intervals $[l, u]$ (like Gigante et al. [43], unlike Heinz et al. [50]) and concurrent execution of non-interfering snap actions without a global lock (Figure 2), (Contribution iii) a detailed correctness proof of the encoding (Section 4, Appendix C), which accounts for the order in which transitions are applied and shows that variable bounds are respected throughout runs, closing gaps in previous informal proofs, (Contribution iv) a formal verification in Isabelle/HOL of this proof, of an efficient executable implementation of the encoding (Section 5), and of the soundness of the system that combines it with the verified certificate checker of Wimmer and von Mutius [90] and an unverified certifying TA

model checker, and (Contribution v) an experimental evaluation showing that our system is practical (Section 6).

2 Background and Definitions

Definition 1 (Boolean expression). *Sets of well-formed arithmetic $\mathcal{E}(\mathcal{V})$ and Boolean $\mathcal{B}(\mathcal{V})$ expressions are constructed from a set of variables $\mathcal{V}$. e and b are in $\mathcal{E}(\mathcal{V})$ and $\mathcal{B}(\mathcal{V})$ respectively, if $e ::= x \mid d \mid e_h \circledast e_i$ and $b ::= e_j \bowtie e_k \mid b_j \wedge b_k \mid \top$, where $\bowtie \in CmpOps \equiv \{<, \leq, =, \geq, >\}$, $\circledast \in ArOps \equiv \{+, -, \cdot, /\}$, $x \in \mathcal{V}$, $d \in \mathbb{D}$, $\{e_h, e_i\} \subseteq \mathcal{E}(\mathcal{V})$, and $\{b_j, b_k\} \subseteq \mathcal{B}(\mathcal{V})$ for some h, i, j, and k.*

Definition 2 (Variable assignment). *Let 2^S denote the powerset of a set S. A* variable assignment *$v : \mathcal{V} \to \mathbb{D}$ is a mapping from a set of variables $\mathcal{V}$ to members of a numeric type $\mathbb{D}$ (e.g. $\mathbb{Q}$). Expressions $e, f \in \mathcal{B}(\mathcal{V}) \cup \mathcal{E}(\mathcal{V})$ are evaluated with respect to a variable assignment v according to the rules $eval(v, x) \equiv v(x)$, $eval(v, d) \equiv d$, $eval(v, \top) \equiv \top$ and $eval(v, e \circ f) \equiv e' \circ f'$, where $x \in \mathcal{V}$, $d \in \mathbb{D}$, $e' \equiv eval(v, e)$, $f' \equiv eval(v, f)$, and $(\{e, f\}, \circ) \in 2^{\mathcal{E}(\mathcal{V})} \times (ArOps \cup CmpOps)$ or $(\{e, f\}, \circ) \in 2^{\mathcal{B}(\mathcal{V})} \times \{\wedge\}$. $v \vDash_b g$ denotes the* satisfaction *of a Boolean expression $g \in \mathcal{B}(\mathcal{V})$ by a variable assignment v, i.e. its evaluation to $\top$. Note that a division by 0 or the presence of a variable $y \notin \mathcal{V}$ in a subexpression of g evaluates to an undefined value $\mathbf{u}$, which falsifies $v \vDash_b g$.*

Definition 3 (Variable updates). *An* update *u for a variable $z \in \mathcal{V}$ is a pair $\langle z, e \rangle \in \mathcal{V} \times \mathcal{E}(\mathcal{V})$, also denoted as $z := e$. $writes(u)$ is the set of written variables $\{z\}$ and $reads(u)$ the set of read variables in e. Similarly, $writes(U)$ and $reads(U)$ are the sets of written and read variables in a set of updates U.*

Timed Automata

Now follow the concepts of timed automata [6] as formalised by Wimmer [89,88], which we present in simplified form. Their work builds on Larsen et al.'s work [67]. A distinctive feature is that it has explicit semantics for variable assignments and concerns networks of timed automata, as opposed to the standard formulation, which concerns individual automata [6].

Definition 4 (Clocks). *A* clock constraint *δ_x is defined with respect to a clock variable x in a set of clock variables $\mathcal{X}$ and a constant $d \in \mathbb{Q}$ as $\delta_x \equiv x \bowtie d$, where $\bowtie \in \{<, \leq, =, \geq, >\}$.*

A clock valuation *c is a mapping from $\mathcal{X}$ to $\mathbb{Q}_{\geq 0}$. A constraint δ_x can be evaluated against a valuation c using the rule: $ccval(c, x \bowtie d)$ if and only if $c(x) \bowtie d$. c satisfies a set of clock constraints g if and only if each constraint δ_x in g is satisfied by c. We denote the* satisfaction *(respectively* falsification*) of a set of constraints as: $c \vDash_c g$ (respectively $c \nvDash_c g$).*

Definition 5 (Clock updates). *A clock $x \in \mathcal{X}$ can only be updated explicitly by resetting its value in a clock valuation c to 0. We denote single resets as $x := 0$ and multiple resets using the set of clocks $r \subseteq \mathcal{X}$ they apply to.*

Definition 6 (Timed Automaton). *A* timed transition relation $\Delta \subseteq \mathcal{L} \times \mathcal{B}(\mathcal{V}) \times \mathcal{C}(\mathcal{X}) \times 2^{\mathcal{V} \times \mathcal{E}(\mathcal{V})} \times 2^{\mathcal{X}} \times \mathcal{L}$ *is a set of* transitions $t \equiv \langle l, b, g, u, r, l' \rangle$ *consisting of (i) an outgoing location l; (ii) a condition b; (iii) a guard g; (iv) a set of updates u; (v) a set of resets r; and (vi) an incoming location l'. In diagrams (e.g. Figure 8 and Definition 19), we omit locations (e.g. $\langle l, b, g, u, r, l' \rangle \simeq \langle b, g, u, r \rangle$), and as a shorthand, list conditions and guards, as well as updates and resets together. We write each condition or guard as a query $(b)?$, and each update or reset as $z \coloneqq e$. A* timed automaton $\mathfrak{T} = \langle \mathcal{L}, \Delta \rangle$ *is a tuple consisting of: (i) a set of locations: $\mathcal{L}$; and (ii) a transition relation: Δ. See Figure 8 for an example of a timed automaton's transitions.*

Definition 7 (Network of Timed Automata). *A* network of timed automata $\mathcal{T} \equiv \langle \vec{\mathcal{T}}, \mathcal{V}, \mathcal{X}, \mathcal{D}, \langle urg_0, \ldots, urg_n \rangle \rangle$ *is: (i) a list of timed automata $\vec{\mathcal{T}} = \langle \mathcal{T}_0, \ldots, \mathcal{T}_n \rangle$, where $\mathcal{T}_i = \langle \mathcal{L}_i, \Delta_i \rangle$ for $0 \leq i \leq n$, with pairwise disjoint sets of locations; (ii) a set of variables $\mathcal{V}$; (iii) a set of clocks $\mathcal{X}$; (iv) a map $\mathcal{D}$ from variables $z \in \mathcal{V}$ to finite interval domains $\mathcal{D}_z = [\mathcal{D}_z^{\text{lo}}, \mathcal{D}_z^{\text{hi}}] \subseteq \mathbb{Z}$; and (v) a subset of locations of each automaton $\mathcal{T}_i$ declared to be* urgent*: $urg_i \subseteq \mathcal{L}_i$. We mark urgent locations in diagrams with exclamation marks: e.g. "loc_1!".*

Definition 8 (Network of Timed Automata Transitions).

The semantics of a network of timed automata are defined with respect to configurations *($\langle L, v, c \rangle$), where (i) $L = \langle l_0, \ldots, l_n \rangle$ is a list of the same length as $\vec{\mathcal{T}}$, recording with each l_i a location for $\mathcal{T}_i$; (ii) v is a variable assignment for $\mathcal{V}$ with values in $\mathbb{Z} = \mathbb{D}$ (see Definition 2); and (iii) c is a clock valuation for $\mathcal{X}$ (see Definition 4). We denote an* initial configuration *as $\langle L_I, v_I, c_I \rangle$, where $v_I(z) \in \mathcal{D}_z$ must hold for all $z \in \mathcal{V}$.*

The transition relation between configurations *is characterised by two kinds of transitions, namely (1)* Delay*: $\langle L, v, c \rangle \longrightarrow_\delta \langle L, v, c' \rangle$, which represents the passage of a time period δ by updating c to c', with the following conditions and updates: (1.a) Condition: No locations l_j in L are* urgent *(see Definition 7) and $\delta \geq 0$; and (1.b) Update: $c'(x) = c(x) + \delta$ for all clocks $x \in \mathcal{X}$, which we denote as $c' = c \oplus \delta$; and (2)* Internal*: $\langle L, v, c \rangle \longrightarrow_t \langle L', v', c' \rangle$, which applies an update to the location of a single automaton $\mathcal{T}_i$ according to the following rules: (2.a) Conditions: (2.a.i) $t \equiv \langle l, b, g, u, r, l' \rangle$ is in $\mathcal{T}_i$'s transition relation Δ_i; (2.a.ii) the location l_i of automaton $\mathcal{T}_i$ matches l; (2.a.iii) the condition on variables is satisfied ($v \vDash_b b$); (2.a.iv) the guard is satisfied ($c \vDash_c g$); and (2.a.v) $eval(v, e) \in \mathcal{D}_z$ for $\langle z, e \rangle \in u$ [1]; and (2.b) Updates: (2.b.i) $L' = \langle l_0, \ldots, l', \ldots, l_n \rangle$, with l' at position i; (2.b.ii) if $\langle z, e \rangle \in u$ then $v'(z) = eval(v, e)$ else $v'(z) = v(z)$ (that is simultaneous updates)[2]; and (2.b.iii) if $x \in r$ then $c'(x) = 0$ else $c'(x) = c(x)$ for all $z \in \mathcal{V}$ and $x \in \mathcal{X}$.*

We write $\longrightarrow$ for the transition relation with unspecified labels and $\longrightarrow^*$ for its reflexive transitive closure (i.e. any sequence of $\longrightarrow_\delta$ and $\longrightarrow_t$).

[1] In Isabelle/HOL and Munta, division is integer division, which rounds down, and falls back to 0 on division by zero [89]. In our checker, we disallow division to avoid unintended behaviour.

[2] Only one update per variable and transition is allowed.

Intuitively, when a *configuration* contains an urgent (Definition 8) location, the network cannot make a transition that represents the passage of time.

Definition 9 (Run). *A* run *consists of alternating* $\longrightarrow_t$ *and* $\longrightarrow_\delta$ *transitions. A finite run is modelled by Wimmer [88] as an inductive list of configurations* $cfgs = \langle\langle L_0, v_0, c_0\rangle, \ldots, \langle L_m, v_m, c_m\rangle\rangle$*. cfgs satisfies the conditions to be a run (run(cfgs)), if for each* $0 \leq k < m$ *there exists a* c'_k*, such that* $\langle L_k, v_k, c_k\rangle \longrightarrow_\delta \langle L_k, v_k, c'_k\rangle \longrightarrow_t \langle L_{k+1}, v_{k+1}, c_{k+1}\rangle$*. When not otherwise specified, we use the term* run *to refer to finite runs.*

Model checkers for timed automata can check that properties expressed as [Timed] Computation Tree Logic ([T]CTL) formulas hold for a network of timed automata. These are asserted on some or all runs beginning at the initial configuration [6]. For modern treatments of the subject, we refer to [24,17].

One limitation is that model checkers need a finite discrete state space, which can be impossible to find when arbitrary integer variables are involved [51]. Hence the need for a finite domain for each variable in Definition 8. We use techniques from abstract interpretation to precompute over-approximations of the domains of variables produced when encoding planning instances (see Section 3.1).

Temporal Planning

Temporal planning is a formalism that can logically represent problems with continuous time and discrete states and actions. Planners explore the space of possible action sequences to achieve specified goals. We illustrate the concepts using instances of a planning domain (Definition 23) as running examples throughout the paper. In this domain, all broken fuses must be mended in a dark cellar using the light provided by matches. The agent only has one free hand to mend one fuse at a time and hence cannot perform multiple actions simultaneously.

We defer the full details to Section 3, but introduce the basic idea of encoding actions shown in Figure 2 and Example 1, like lighting a match or mending a fuse, that can change a propositional and numeric state of the world as timed automata in a network, propositions, such as "fuse 1 is broken", as binary (integer) variables, and quantities, like the number of matches used, as integer variables.

Real-world planning software works on abstract syntax of languages such as PDDL [42] and ANML [82], which represent actions as templates (or *schemas*), propositions as predicates and numeric fluents as functions to be instantiated with constants (or *objects*). PDDL also allows the representation of state-dependent duration constraints and continuous change, which are not supported by standard timed automata. The semantics of PDDL are complex enough that checking plans output by planning software by eye is hard; a significant amount of research has been dedicated to producing tools that perform the check automatically and improving their reliability [59,49], and even formally verifying such checkers [1,2].

We base our temporal planning formalism on a fragment of the formal semantics by Abdulaziz and Koller [2], except that our preconditions here are limited to conjunctions. This roughly corresponds to a subset of grounded PDDL. We also refer to the work of Gigante et al. [43], who assume that a planning prob-

lem has been preprocessed to use a set-theoretic formulation of propositional planning, with conditions being minimal sets of true propositions.

Definition 10 (Valuation). *A valuation $\mathcal{A}$ is a mapping from a set of propositions P to the set $\{0,1\}$. A valuation $\mathcal{A}$ is a model for a set of propositions ϕ, written $\mathcal{A} \vDash_b \phi$, if and only if $\mathcal{A}(p) = 1$ for all propositions $p \in \phi$. A valuation $\mathcal{A} : P \to \{0,1\}$ is interpreted as the set $\{v \mid \mathcal{A}(v) = 1\}$ and a set of propositions Q is interpreted as a valuation which maps any $p \in Q$ to 1, and everything else to 0. For example, a set Q_1 in Example 1 can include the proposition p, which indicates that "the hand is busy". Also, in the rest of this paper a* propositional state *M is synonymous with a valuation.*

As all propositional formulas in this paper are conjunctions of positive atoms (i.e. propositions), we characterise them using sets of their propositions.

Definition 11 (Temporal planning problem). *A pair of a propositional valuation M_P and a variable assignment M_F with values in $\mathbb{Q} = \mathbb{D}$ (see Definitions 2 and 10) forms a* state *$M \equiv \langle M_P, M_F \rangle$. A planning problem Π is a tuple $\langle P, F, A, I, G \rangle$, where (i) P is a set of state-characterising propositions; (ii) F is a set of state-characterising numeric fluents (i.e. numeric variables); (iii) $A \equiv \{\pi_i \mid 1 \leq i \leq N\}$: an indexed set of N actions, each of which is a tuple $\langle \pi_{start}, \pi_{end}, \pi_{inv}, \pi_{ninv}, \pi_l, \pi_u \rangle$ where (1) π_{start}, π_{end} are start and end* snap actions*; (2) π_{inv} is a propositional invariant, a propositional formula defined over the propositions P; (3) π_{ninv} is a numeric invariant, a Boolean expression defined over the fluents F; and (4) $\pi_l \equiv \langle l_\pi, \lhd_{l\pi} \rangle$ and $\pi_u \equiv \langle u_\pi, \lhd_{u\pi} \rangle$ are lower and upper bounds on the duration any instance of π can take, expressed as pairs in $\mathbb{Q} \times \{<, \leq\}$. A snap action α is a tuple $\langle \alpha_{pre}, \alpha_{nc}, \alpha_{add}, \alpha_{del}, \alpha_{upd} \rangle$ where (a) α_{pre} is its propositional precondition, a formula using propositions P; (b) α_{nc} is its numeric precondition, a Boolean expression over fluents F; (c) $\alpha_{add} \subseteq P$ are its positive effects; (d) $\alpha_{del} \subseteq P$ are its negative effects; and (e) α_{upd} is a set of variable updates over the fluents F; (iv) I is an initial state; and (v) $G \equiv \langle G_P, G_F \rangle$, where G_P is a propositional condition and G_F is a Boolean expression, characterising the goal. A full example of a temporal planning problem can be found in Example 1.*

Example 1. The first instance of our running example can be seen in Figure 2 and is formally declared in Definition 24. It has one fuse f_0 and one match m_0. It extends MatchCellar with some numeric fluents initialised by I, namely: (i) *spare-fuses* for the number of spare fuses decremented when one is used by $\mathit{mend\text{-}fuse}(f_0, m_0)_{start} \equiv \mathit{mf}_{00\,start}$; (ii) *matches-lit* for the total number of lit matches, which is incremented when one is lit by $\mathit{light\text{-}match}(m_0)_{start} \equiv \mathit{lm}_{0\,start}$; and (iii) $\mathit{dur\text{-}lit\text{-}lb}(m_0)$, $\mathit{dur\text{-}lit\text{-}ub}(m_0)$ and $\mathit{dur\text{-}mend}(f_0)$, which are lower and upper bounds for the time durations of the action that keeps a match lit lm_0 and the action that mends a fuse mf_{00}. We write these as intervals $[\mathit{dur\text{-}lit\text{-}lb}(m_0), \mathit{dur\text{-}lit\text{-}ub}(m_0)]$ and $[\mathit{dur\text{-}mend}(f_0), \mathit{dur\text{-}mend}(f_0)]$ respectively.

We denote actions, propositions, and fluents using schematic actions, predicates and functions. Typically, the first step to solving a planning problem is

grounding [53], which instantiates a set of these schematic actions with type-correct objects, such as *light-match*(x) with m_0 for x, which yields the start effect *matches-lit* := *matches-lit* + 1 of *light-match*$(m_0)_{start}$. A naive grounding would enumerate all type-correct permutations, which is why good grounding is hard [29] and recently lifted planners try to avoid grounding altogether [30,28].

matches-lit is a non-trivially bounded numeric fluent, which appears unbounded, because its upper bound is implicit in the total number of actions that increment its value and can execute at most once each. Static analysis could find the bound. Compiling objects into counters could also work [40,77]. Lifted planners could also use this knowledge.

In principle, truly unbounded fluents make PDDL [39] more expressive than timed automata and planning undecidable [51]. The bound for *spare-fuses* can be inferred using interval arithmetic and abstract interpretation. See Section 3.1.

Numeric values assigned to variables in planning are taken to be real-valued [39], which means that they cannot be represented in timed automata. In fact, continuous planning semantics are based on ODEs and correspond to reachability in hybrid automata [39]. We present temporal planning with integer fluents. The benchmarks by Panjkovic et al. [74] include domains with rational and constant duration constraints and provide versions with durations normalised to integers, except for PAINTER, which our unverified grounder normalises. Normalisation of constant duration constraints with respect to rationals is standard in timed automata [6] but makes it hard to relate the original and normalised values. Duration constraints in planning can also depend on the value of arithmetic operations over fluents and vice versa, in which case normalisation is not trivial or even possible [39,69].

Definition 12 (Effect application). *The application of a set of snap actions B on a state M is defined as $M[B] \equiv M' \equiv \langle {M'}_P, {M'}_F \rangle$, where ${M'}_P = (M_P - \bigcup_{\alpha \in B} \alpha_{del}) \cup \bigcup_{\alpha \in B} \alpha_{add}$ and ${M'}_F = M_F$ except for every written fluent v, which is assigned $eval(M_F, e)$, where $(v, e) \in \bigcup_{\alpha \in B} \alpha_{upd}$. For M' to be defined, no $eval(M_F, e)$ is* $\mathbf{u}$*, where $(v, e) \in \bigcup_{\alpha \in B} \alpha_{upd}$.*

Definition 13 (Interference). *Snap actions α_1 and α_2 are interfering ($mutex(\alpha_1, \alpha_2)$) if and only if (i) $\alpha_{1pre} \cap (\alpha_{2add} \cup \alpha_{2del}) \neq \emptyset$; (ii) $\alpha_{2pre} \cap (\alpha_{1add} \cup \alpha_{1del}) \neq \emptyset$; (iii) $\alpha_{1add} \cap \alpha_{2del} \neq \emptyset$; (iv) $\alpha_{2add} \cap \alpha_{1del} \neq \emptyset$; (v) $reads(\alpha_1) \cap writes(\alpha_2) \neq \emptyset$; (vi) $reads(\alpha_2) \cap writes(\alpha_1) \neq \emptyset$; or (vii) $writes(\alpha_1) \cap writes(\alpha_2) \neq \emptyset$, where $reads(\alpha_i) = reads(\alpha_{i_{upd}}) \cup reads(\alpha_{i_{inc}})$ and $writes(\alpha_i) = writes(\alpha_{i_{upd}})$ for $i \in \{1, 2\}$.*

If two snap actions α_1 and α_2 interfere, then $(M - \alpha_{2del}) \cup \alpha_{1add}$ is not necessarily equivalent to $(M \cup \alpha_{1add}) - \alpha_{2del}$, and $\alpha_{2pre} \subseteq M$ does not imply $\alpha_{2pre} \subseteq M - \alpha_{1del}$. Neither is the sequential application of α_{1upd} and α_{2upd} guaranteed to produce the same result as α_{2upd} and then α_{1upd}.

Planning abstracts away from *commonsense reasoning* about interfering actions by prohibiting such interference in the semantics.

Our interference condition is more restrictive than that of Fox and Long [39], which allows simultaneous writes of the same fluent f by additive effects of the form $f \mathrel{+}= e$ or $f \mathrel{-}= e$. Ours treats additive effects as assignments of the form $f \coloneqq f + e$ or $f \coloneqq f - e$ and interprets these as interfering.

In contrast, transitions in the timed automata formalism (Definition 8) are applied sequentially, which is only order-independent, if a set of such transitions do not interfere.

See Figures 5 and 7 for an example of mutual exclusion preventing unintuitive effects of actions.

Example 2. Our next example is MatchCellar with: one match m_0, 3 fuses f_i for $0 \leq i < 3$, a fixed 5-second burn duration ($\mathit{dur\text{-}lit\text{-}lb}(m_0)$, $\mathit{dur\text{-}lit\text{-}ub}(m_0)$), 2-second mend duration $\mathit{dur\text{-}mend}(f_i)$ for $0 \leq i < 3$, and no notion of lit matches or spare fuses. This is equivalent to instance 2 from Panjkovic et al.'s benchmarks [74]. It is formally defined in Definition 25. It demonstrates the complexity of manually inspecting an encoding from planning to timed automata: (1) each of the 3 mend actions $\mathit{mend\text{-}fuse}(f_i, m_0) \equiv \mathit{mf}_{i0}$, where $0 \leq i < 3$, depends on the hand being free (written *handfree*) $\mathit{handfree} \in \mathit{mf}_{i0\mathit{startpre}}$, occupies it $\mathit{handfree} \in \mathit{mf}_{i0\mathit{startdel}}$, and frees it again $\mathit{handfree} \in \mathit{mf}_{i0\mathit{endadd}}$, which makes their starts interfere with starts and ends of other mf_{j0}, where $0 \leq j < 3$, and not executable in the same instant (this needs 27 clock constraints to encode as we and Gigante et al. [43] do); and (2) the durative invariant that the match must be burning ($\mathit{light}(m_0)$) throughout a mend action ($\mathit{mend\text{-}fuse}(f_i, m_0)$) is enforced by a lock counter, by preventing deletions of the proposition (assignments to 0) while the counter is above 0 (see Definition 20).

Definition 14 (Plan). *A plan is a list of tuples $\langle\langle \pi_1, t_1, d_1\rangle, \ldots, \langle \pi_n, t_n, d_n\rangle\rangle$, where, for $1 \leq i \leq n$, $\pi_i \in A$ is an action, $t_i \in \mathbb{Q}_{\geq 0}$ and $d_i \in \mathbb{Q}_{\geq 0}$ are rational numbers, to which we refer as the starting time point and the duration, respectively.*

For a plan $\vec{\pi}$, we call a sorted sequence $t_0, \ldots, t_m$ of the set of rational numbers $\{t \mid \langle a, t, d\rangle \in \vec{\pi}\} \cup \{t + d \mid \langle a, t, d\rangle \in \vec{\pi}\}$ the happening time points (htps) of the plan, and we denote it by $\mathit{htps}(\vec{\pi})$.

Definition 15 (Valid state sequence). *For $t \in \mathbb{Q}_{\geq 0}$ and a plan $\vec{\pi}$, let the* induced parallel plan *be $B_t \equiv \{\pi_{\mathit{start}} \mid \langle \pi, t, d\rangle \in \vec{\pi}\} \cup \{\pi_{\mathit{end}} \mid \langle \pi, t - d, d\rangle \in \vec{\pi}\}$ and $I_t \equiv \{\pi \mid \langle \pi, t', d\rangle \in \vec{\pi} \wedge t' < t \leq t' + d\}$. Also, let $t_0, \ldots, t_m$ be the sorted happening time points of $\vec{\pi}$.*

For a sequence of states $M_0, \ldots, M_{m+1}$, we say the sequence of states is valid with respect to a plan $\vec{\pi}$ if and only if, for every happening time point t_i of $\vec{\pi}$, we have: (i) $(M_i)_P \vDash_b \pi_{\mathit{inv}}$ and $(M_i)_F \vDash_b \pi_{\mathit{ninv}}$, for every $\pi \in I_{t_i}$; (ii) $(M_i)_P \vDash_b \alpha_{\mathit{pre}}$ and $(M_i)_F \vDash_b \alpha_{\mathit{nc}}$, for every $\alpha \in B_{t_i}$; (iii) for every t_j in $\mathit{htps}(\vec{\pi})$, $B_{t_i} \cup B_{t_j}$ is pairwise non-interfering when $t_i = t_j$ (0-separation) or $0 < |t_i - t_j| < \epsilon$ (ϵ-separation); and (iv) $M_{i+1} = M_i[B_{t_i}]$. Here, 0-separation means that no snap actions at the same time point can interfere and ϵ-separation means that the minimum separation between two interfering snap actions is at least some ϵ, which is fixed to define the semantics. If it is set to zero, this is

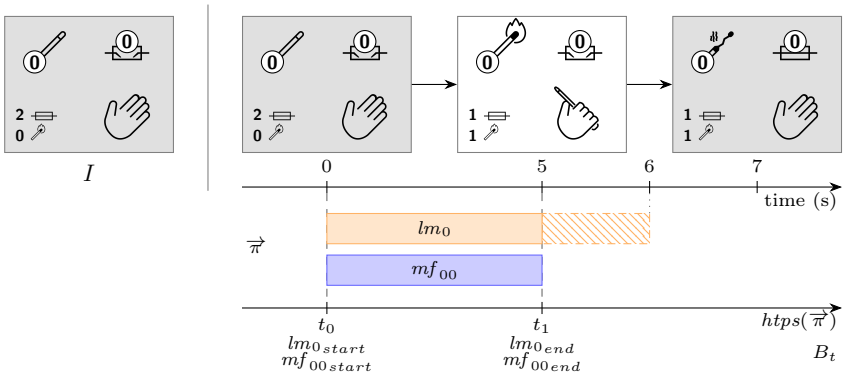


Fig. 2: Left: I: (i) m_0 is not lit, (ii) f_0 is not mended, (iii) the hand is free *handfree* $= 1$, (iv) 2 spare fuses *spare-fuses*, and (v) 0 lit matches *matches-lit*. Right: A valid state sequence showing states at time points before applying effects. Bottom: A valid plan $\vec{\pi}$ progresses the state sequence from I to a state satisfying G after 5 s. lm_0 (lighting a match) and mf_{00} (mending a fuse) are both executed for 5 s at $t_0 = 0$. A timeline shows alternative valid durations for lm_0 (e.g. the interval $[5, 6]$).

equivalent to 0*-separation between actions. (See Figures 6 and 9b for example state sequences.)*

Definition 16 (Valid plan). *Let, for a plan* $\vec{\pi}$*, no-self-overlap*$(\vec{\pi})$ *denote that for all distinct* i *and* j*, where* $\langle a_i, t_i, d_i\rangle$ *and* $\langle a_j, t_j, d_j\rangle$ *are the* i*-th and* j*-th elements of* $\vec{\pi}$*, there is either no* overlap*, i.e.* $\neg(t_i \leq t_j \leq t_i + d_i)$*, in the closed intervals* $[t_i, t_i + d_i]$ *and* $[t_j, t_j + d_j]$*, or* $a_i \neq a_j$*.*

$\vec{\pi}$ *is a valid plan for a problem* Π *if and only if: (i) there is a state sequence* $M_1, \ldots, M_{m+1}$ *such that* $I, M_1, \ldots, M_{m+1}$ *is valid with respect to* $\vec{\pi}$*; (ii)* $(M_{m+1})_P \vDash_b G_P$ *and* $(M_{m+1})_F \vDash_b G_F$*; (iii) any tuple* $\langle a_i, t_i, d_i\rangle \in \vec{\pi}$ *satisfies duration constraints* $l_\pi \lhd_{l\pi} d_i \lhd_{u\pi} u_\pi$ *of the durative action* $\pi \equiv a_i$*, where* $\pi_l \equiv \langle l_\pi, \lhd_{l\pi}\rangle$*,* $\pi_u \equiv \langle u_\pi, \lhd_{u\pi}\rangle$ *and* $\{\lhd_{l\pi}, \lhd_{u\pi}\} \subseteq \{<, \leq\}$*; and (iv) no-self-overlap*$(\vec{\pi})$ *holds. We write dur-c-sat*(π, d) *to indicate the satisfaction of duration constraints by a duration* d*.*

Figure 2 shows a valid plan for Example 1. For an informal illustration of mutual exclusivity in relation to valid state sequences, see Figures 4, 6 and 9a and Example 6.

Note: we add the *no self-overlap* condition to plans for validity because Gigante et al. [43] showed that, without it, planning is undecidable under 0-separation (which is what Abdulaziz and Koller [2] use in their separation constraint) and EXPSPACE-complete under ϵ-separation, vs. PSPACE-complete without self-overlap.

ITPs and Isabelle/HOL Isabelle/HOL [73] is an ITP for the development and checking of natural deduction proofs in Higher-Order Logic (HOL). Isabelle

combines automated proof checking with an interactive proof development environment.

Definitions and statements about those definitions which are to be proven can be declared by the user in a formal language that is mainly based on Church's simple type theory [22,76]. Isabelle/HOL's language, however, has a number of features that are either added to or built upon simple type theory, like locales [60], type classes, and facilities to define and reason about algebraic datatypes [85] and recursive functions [62]. Proofs for claims in Isabelle/HOL are either fully automatically constructed by the ITP or, if that fails, the user provides higher-level arguments for the proofs and the ITP tries to bridge the gaps in those arguments. The process is continued until a full formal proof, down to the logical axioms, is constructed.

ITPs have been used in many applications, ranging from the verification of low-level software, like the seL4 [61] OS kernel; algorithms for planning [3], model checking [89], and SAT solving [15]; all the way to formally verifying proofs in pure mathematics [8,44,45,19,13,48,20] —here they facilitate large collaborative efforts.

3 Encoding

As shown in Examples 1 and 2, even features of planning that can be modelled are hard to model in TA networks. E.g. (i) non-trivially bounded (or unbounded) fluents (Example 1 and Listing 1.1), (ii) mutex constraints (Example 2), (iii) durative invariants over numeric fluents (which we only encode as an over-approximation), (iv) durations that depend on fluents (Listing 1.1; see also Long and Fox [69]), (v) simultaneous snap action execution (Definition 15 and urgent locations in Definition 20), (vi) no self-overlap (Definition 16) which is implicit in the structure of the automaton for each action (Definition 20), and (vii) no actions running at the end (Definition 17).

The correctness of these features is hard to assert by manual inspection—in particular the correctness of the inferred bounds in Section 3.1. Therefore, we argue for placing trust in Isabelle, as opposed to the code itself; we do this for the concepts in this section by formalising Section 4 in Isabelle with respect to Wimmer's semantics [89,88] and a modified version of Abdulaziz and Koller's semantics [2] to add numeric fluents.

This section describes a practical encoding of temporal planning problems into a network of timed automata based on the work of Heinz et al. [50] and Gigante et al. [43]. Heinz et al. [50] use integer variables and Bogomolov et al. [16] a global lock automaton to allow one snap action per ϵ interval. This enforces mutex constraints (Definition 13), but prevents non-interfering snap actions from executing in the same instant, unlike ours and Gigante et al.'s [43], which both use clock constraints. Hence, some transitions must query a number of clocks, which scales linearly with the number of actions (see Example 2 and Definition 20). Our approach thus checks and proves a more general notion of unsolvability than that of Heinz et al. [50], and we provide a formal proof of correctness.

Our encoding (detailed below) has one automaton per action, and one main automaton. The main automaton encodes the planning problem by initialising a propositional state and checking the satisfaction of a goal, and each action automaton encodes a unique durative action. An action automaton checks that a snap action's preconditions have been satisfied, before transitioning to the corresponding urgent location and resetting a unique clock. Moreover, using a set of pre-compiled clock constraints, an action automaton checks that no interfering snap actions have done the same in the same instant. Like Heinz et al. [50], we use a simple counter assigned to each propositional variable to ensure that no invariants are violated when an action is running. Effects are applied to a propositional state encoded using a set of binary variables. When any action automaton is in an urgent location (Definition 8), the instant must correspond to a happening time point.

Definition 17 (Integer variables $\mathcal{V}$). *We obtain a set of variables $\mathcal{V}$ by adding the following variables and bounds to the set of fluents F in the planning problem Π: (i) vp_p with bounds $[0, 1]$ (*binary variable*) for every proposition $p \in P$, which records p's truth value encoded as 0 or 1; (ii) lp_p with bounds $[0, N]$ (*lock counter*) for every proposition $p \in P$, which records the number of active actions that require p to be satisfied as invariant; (iii) aa with bounds $[0, N]$ (*active action counter*) records the number of active actions; and (iv) ps with bounds $[0, 2]$ (*planning state*) records one of: 0 (not started), 1 (planning), 2 (done). These must be fresh with respect to F. Note: by construction, no transition assigning a value to ps or vp_p for any $p \in P$ outside of their bounds is possible. For the bounds of lp_p and aa we refer to Theorem 1, and for those of F we refer to Section 3.1.*

Definition 18 (Action clocks). *We assign two clocks per durative action π: (i) a* start clock $x_{\pi_{start}}$ *for the time since π_{start}; and (ii) an* end clock $x_{\pi_{end}}$ *for the time since π_{end} has happened. When the kind (start/end) of a snap action α_1 is unspecified, we use x_{α_1} to denote such a clock.*

Definition 19 (Main Automaton). *The conditions and effects are: (i) $e_{1M} \equiv \langle \emptyset, \emptyset, \{(ps \coloneqq 1)\} \cup \{(vp_p \coloneqq 1) \mid p \in I_P\} \cup \{(f \coloneqq I_F(f)) \mid f \in F\}, \emptyset \rangle$; (ii) $e_{2M} \equiv \langle \{(aa = 0)?\} \cup \{(vp_p = 1)? \mid p \in G_P\} \cup \{G_F\}, \emptyset, \{(ps \coloneqq 2)\}, \emptyset \rangle$; and (iii) $e_{3M} \equiv \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$. An illustration of this automaton can be seen in Figure 3.*

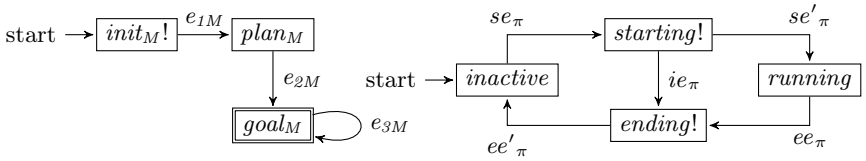


Fig. 3: The main automaton $\mathcal{T}_M$ and an automaton $\mathcal{T}_\pi$ for some action π.

Definition 20 (Action Automata). *Figure 3 shows the structure of an automaton $\mathcal{T}_\pi$, which encodes an action $\pi \in A$. The structure of four locations to record four states of actions is inspired by the work of Heinz et al. [50]. First, we characterise a few abbreviations and functions (i) $mutex_e(\alpha_1)$ checks clocks to ensure that no scheduled snap actions interfere: $\{(0 < x_{\alpha_2})?, (\epsilon \leq x_{\alpha_2})? \mid \alpha_2 \in intf(\alpha_1)\}$, where $intf(\alpha_1)$ is the set of all snap actions α_2 in the problem, such that $mutex(\alpha_1, \alpha_2)$; when $\epsilon = 0$, this is 0-separation semantics (see Definitions 13 and 15 and Gigante et al. [43]); (ii) $dur_e(\pi)$ is satisfied when dur-c-sat$(\pi, c(x_{\pi_{start}}))$ is (see Definition 16): i.e. $\{(l_\pi \lhd_{l\pi} x_{\pi_{start}})?, (x_{\pi_{start}} \lhd_{u\pi} u_\pi)?\}$ where $\pi_l \equiv \langle l_\pi, \lhd_{l\pi}\rangle$ and $\pi_u \equiv \langle u_\pi, \lhd_{u\pi}\rangle$; (iii) $eff_e(\alpha_1)$ encodes the effects of a snap action α_1 on propositional variables; it is defined as the union of $\{(vp_p \coloneqq 0) \mid p \in \alpha_{1del} \wedge p \notin \alpha_{1add}\}$ and $\{(vp_p \coloneqq 1) \mid p \in \alpha_{1add}\}$; (iv) $pre_e(\alpha_1)$ encodes the preconditions as a Boolean condition: $\{(vp_p = 1)? \mid p \in \alpha_{1pre}\} \cup \{\alpha_{1nc}\}$; and (v) $inv_e(\alpha_1)$ checks that no effect of α_1 invalidates an active invariant: $\{(lp_p = 0)? \mid p \in \alpha_{1del} - \alpha_{1add}\}$. The transitions' relevant conditions and effects are: (i) $se_\pi \equiv \langle pre_e(\pi_{start}) \cup inv_e(\pi_{start}), mutex_e(\pi_{start}), eff_e(\pi_{start}) \cup (\pi_{start})_{upd} \cup \{(aa \coloneqq aa + 1)\}, \{x_{\pi_{start}}\}\rangle$; (ii) $se'_\pi \equiv \langle \{(vp_p = 1)? \mid p \in \pi_{inv}\} \cup \{\pi_{ninv}\}, \emptyset, \{(lp_p \coloneqq lp_p + 1) \mid p \in \pi_{inv}\}, \emptyset\rangle$; (iii) $ee_\pi \equiv \langle \{\pi_{ninv}\}, mutex_e(\pi_{end}) \cup dur_e(\pi), \{(lp_p \coloneqq lp_p - 1) \mid p \in \pi_{inv}\}, \{x_{\pi_{end}}\}\rangle$; (iv) $ee'_\pi \equiv \langle pre_e(\pi_{end}) \cup inv_e(\pi_{end}), \emptyset, eff_e(\pi_{end}) \cup (\pi_{end})_{upd} \cup \{(aa \coloneqq aa - 1)\}, \emptyset\rangle$; and (v) $ie_\pi \equiv \langle \emptyset, mutex_e(\pi_{end}) \cup dur_e(\pi), \emptyset, \{x_{\pi_{end}}\}\rangle$. Moreover, every transition in $\mathcal{T}_\pi$ is conditioned on $(ps = 1)$, which ensures that no automaton $\mathcal{T}_\pi$ starts a simulated execution of an action $\pi \in A$ before all variables have been initialised by e_{1M}, which applies $(ps \coloneqq 1)$. $(ps = 1)$ also prevents further changes once e_{2M} applies $ps \coloneqq 2$.*

This encoding is not exact, but it is sound, which is needed to check unsolvability. It only checks the numeric invariant satisfaction at the start and end of the action, which means that runs simulating invalid plans could be admitted, but none that simulate a valid plan will be rejected.

On the other hand, propositional invariants p in π_{inv} are exactly modelled by lp_p on the condition that they are conjunctions of propositions.

The defining features of the encoding in Definition 20 for an action π are the encoding of the conditions of π_{start} and π_{end} into a location and two transitions each. The *urgency* (Definition 8) of $(starting_\pi!)$ and $(ending_\pi!)$ ensures that all transitions necessary to simulate the execution of a snap action are applied in the same instant. Encoding a snap action as two transitions is necessary to capture the semantics of π_{2inv} conditions in a valid state sequence (Definition 15). See Examples 4 and 5 and Figures 4, 6, 9a and 10 for examples. Every update of the form $(vp_p \coloneqq 0)$ in an ee'_π or se_π edge is accompanied by a $(lp_p = 0)?$ check. This check can require lp_p for some $p \in P$ to be decremented by some ee_{π_2} for which $p \in \pi_{2inv}$. In other cases se'_{π_2} can only increment lp_p after the $(lp_p = 0)?$ check has occurred in some se_{π_3} or ee'_{π_3}.

Most importantly, by adding ie_π and using our relaxed definition of a valid plan (Definition 16), which admits durations $d = 0$, we use durative actions to encode instantaneous snap actions. ie_π is applicable only if $\pi \in A$ has a lower

duration bound of $\pi_l \equiv \langle 0, \leq \rangle$, because $starting_\pi$ is urgent (Definition 8) and any transition to $starting_\pi$ resets the corresponding $x_{\pi_{start}}$ to 0.

Definition 21 (Network for a planning problem). *For a planning problem Π, the list of automata $\vec{\mathcal{T}}$ is $\langle \mathcal{T}_M, \mathcal{T}_{\pi_1}, \ldots, \mathcal{T}_{\pi_N} \rangle$.*
Initial configuration. *The initial configuration $\langle L_I, v_I, c_I \rangle$ is characterised as follows. The initial locations are $L_I \equiv \langle init_M, inactive_{\pi_1}, \ldots, inactive_{\pi_N} \rangle$. They are marked with arrows in Figure 3. The variable assignment and clock valuation are initialised to satisfy $v_I(z) = \mathcal{D}_z^{\text{lo}}$ for any variable $z \in \mathcal{V}$ and $c_I(x) = 0$ for any clock $x \in \mathcal{X}$.*
Acceptance condition. *See Figure 3. The acceptance condition is the reachability of a configuration in which $\mathcal{T}_M$ is in the $goal_M$ location, i.e. $\mathcal{T} \vDash \text{EF}(loc(\mathcal{T}_M) = goal_M)$ [5,24,17].*

Remark 1. $\text{EF}\phi$ *means that some run eventually reaches a configuration satisfying ϕ, which in this case asserts that $L_i \equiv \langle goal_M, \ldots \rangle$, where $\langle L_i, v_i, c_i \rangle$ is the i-th configuration of some run.*

3.1 Numeric Fluent Bounds

We encode fluents directly as a subset of variables of the timed automaton network. Note that the planning semantics use unbounded $\mathbb{Q}$-ranged variables, and to guarantee termination and a finite reachable state space of discrete states, the implementation restricts variables and operations to integers with finite bounds. We reject planning problems that contain non-integer constants, except in rational duration constraints, which can be evaluated exactly independent of the state (Definition 11). This is sufficient to precisely encode propositional states, and addition, subtraction and multiplication involving numeric fluents. However, integer division is not exact, so we also reject planning problems involving integer division for now. Our encoding can, however, be extended to admit problems involving only exact integer division.

For unbounded numeric fluents in the planning problem, we infer approximate, sound bounds in the automaton. We use widening with thresholds [47,14], a technique from abstract interpretation [31], with a carefully deduced set of thresholds. The inference starts with the interval of each fluent f set to $[I_F(f), I_F(f)]$. In each iteration, every snap action is applied to the current intervals: its numeric precondition first restricts the intervals of the fluents it depends on. Then its numeric effects are evaluated with interval arithmetic. The current intervals are extended to cover the results. If an interval grows, its new bound is widened to the nearest threshold beyond it, or to $\pm\infty$ if there is none. The thresholds are collected from the initial values, the constants in the actions' preconditions and effects, and the constants and initial values of the expressions compared with a fluent in a precondition, shifted by the increment that an effect of the same action applies to that fluent. The iteration stops once no interval grows, which happens after finitely many iterations limited by the set of thresholds (see Proposition 1). See also Appendix E for the pseudocode and Lemma 1 for how these bounds are used in the proof of Theorem 1.

4 Correctness

Contribution iii is a detailed proof that the existence of a plan is sufficient to demonstrate the existence of a valid run in the encoding of temporal planning as timed automata in Section 3. Proving this is what is needed to show the encoding is useful for certifying unsolvability. In addition to being formally verified, we believe our proof's level of detail, which is higher than all those by earlier authors, and its auxiliary concepts and lemmas will be of general interest for others working on encoding temporal planning into automata.

The main idea is that we construct a run which simulates the remainder of an induced parallel plan (see Definition 15). Then we combine this with a run beginning at the initial configuration $\langle L_I, v_I, c_I\rangle$ and append a final transition into an accepting state $\langle L_G, v_G, c_G\rangle$. We first define conditions under which each part of such a run can be created.

Definition 22 (Encoded states). *runs-at$(\pi, t_i, \triangleleft)$, where $\triangleleft \in \{\leq, <\}$ encodes whether an action π is running at or after time point t_i: runs-at$(\pi, t_i, \triangleleft)$ is true if and only if there exist s and d such that $\langle \pi, s, d\rangle \in \overrightarrow{\pi}$ and $s \triangleleft t_i$ and $\neg(s + d \triangleleft t_i)$.*

ran-for-at$(\alpha_1, t_i, \triangleleft)$ is the time between t_i and the last time a snap action α_1 has been applied, including t_i: ran-for-at$(\alpha_1, t_i, \triangleleft)$ is $t_i - s$, where s is the greatest time point for which $\alpha_1 \in B_s$ and $s \triangleleft t_i$ hold. If no such time point exists, then ran-for-at$(\alpha_1, t_i, \triangleleft)$ evaluates to a value $t_i - t_0 + \kappa + \epsilon$ where κ is fixed and $0 < \kappa$.

enc-after$(\langle L_I, v_I, c_I\rangle, i, \langle L_i, v_i, c_i\rangle)$ denotes the encoded state of a planning problem. When a happening time point sequence is not empty,[3] *enc-after$(\langle L_I, v_I, c_I\rangle, i, \langle L_i, v_i, c_i\rangle)$ is satisfied if for the i-th happening time point $t_i \in$ htps$(\overrightarrow{\pi})$, all of the following hold: (i) For any action $\pi_j \in A$: (i.a) L_i records the location l_j of $\mathcal{T}_\pi$ as inactive$_\pi$ if $\neg$runs-at$(\pi, t_i, \leq)$; (i.b) L_i records the location l_j as running$_\pi$ if runs-at$(\pi, t_i, \leq)$; and (i.c) $c_i(x_x)$ evaluates to ran-for-at$(x, t_i, \leq)$, for all $x = \pi_{j\,start}$ or $x = \pi_{j\,end}$; (ii) For any proposition $p \in P$: (ii.a) $v_i(vp_p)$ evaluates to 1 if $p \in (M_{i+1})_P$ and 0 if $p \notin (M_{i+1})_P$; and (ii.b) $v_i(lp_p)$ evaluates to $|\{\pi \in A \mid p \in \pi_{inv} \wedge$ runs-at$(\pi, t_i, \leq)\}|$; (iii) For any fluent $f \in F$, $v_i(f)$ evaluates to $(M_{i+1})_F(f)$; (iv) $v_i(aa)$ evaluates to $|\{\pi \in A \mid$ runs-at$(\pi, t_i, \leq)\}|$; and (v) $v_i(ps)$ evaluates to 1.*

Theorem 1. *If a plan $\overrightarrow{\pi}$ is valid for a planning problem Π and $\mathcal{T}$ is the network for Π (Definition 21), then $\mathcal{T} \vDash \mathrm{EF}(loc(\mathcal{T}_M) = goal_M)$ holds.*

Proof sketch (Appendix C has a detailed proof). It is sufficient to show the existence of a run *cfgs* beginning at $\langle L_I, v_I, c_I\rangle$ and ending in $\langle L_G, v_G, c_G\rangle$, where L_G records $\mathcal{T}_M.goal_M$. *cfgs* is constructed in three stages. First, $\langle L_I, v_I, c_I\rangle$ must transition to a configuration $\langle L'_0, v'_0, c'_0\rangle$. The action transition e_{1M} can update

[3] When the htp sequence is empty, we define *enc-after$(\langle L_I, v_I, c_I\rangle, i, \langle L_i, v_i, c_i\rangle)$* to encode the propositional and numeric conditions of the initial state after a small delay has elapsed. See Definition 27.

v'_I of $\langle L'_I, v'_I, c'_I\rangle$ to encode I of Π from which a delay $\delta = \kappa + \epsilon$ (Definition 22) will result in a configuration $\langle L_0, v_0, c_0\rangle$ which satisfies mutex conditions on c_0. By applying transitions which encode all snap actions $\alpha \in B_{t_0}$, an initial run $\langle\langle L_I, v_I, c_I\rangle, \langle L'_I, v'_I, c'_I\rangle, \langle L_0, v_0, c_0\rangle, \ldots, \langle L'_0, v'_0, c'_0\rangle\rangle$ is obtained, where *enc-after*$(\langle L_I, v_I, c_I\rangle, 0, \langle L'_0, v'_0, c'_0\rangle)$ is satisfied. Then we can prove by induction on the length of $\mathit{htps}(\overrightarrow{\pi})$ that there is some $\langle L_k, v_k, c_k\rangle$ such that $\mathit{run}([\langle L'_0, v'_0, c'_0\rangle, \ldots, \langle L_k, v_k, c_k\rangle])$ and *enc-after*$(\langle L_I, v_I, c_I\rangle, k, \langle L_k, v_k, c_k\rangle)$ are satisfied, where $k \leq m$. Note that our construction ensures that all variable bounds are respected throughout the run (see Lemma 1). For the details of the inductive step, which involves a specific order of transitions that must be applied, we refer to Appendix C, Figures 4, 6 and 10, and Example 5. If $\langle L_k, v_k, c_k\rangle$ satisfies *enc-after*$(\langle L_I, v_I, c_I\rangle, k, \langle L_k, v_k, c_k\rangle)$, where $k = m$, and L_k records plan_M for $\mathcal{T}_M$, then e_{2M} can be applied to $\langle L_k, v_k, c_k\rangle$ (Lemma 4) to reach $\langle L_G, v_G, c_G\rangle$, where $L_G \equiv \langle \mathit{goal}_M, \ldots\rangle$.

5 Verification and Implementation

The effort in the encoding and its correctness proof is substantial given the complexity of the definitions involved. Indeed, the total size of the implementation and the associated proofs is 43K lines of Isabelle/HOL scripts (of which 23K are hand-written), which is much larger than similar projects. E.g. Abdulaziz and Kurz's certifying SAT-based planner [3] is 17.5K lines; the difference is primarily due to the relative complexity of the concepts and thus the proofs for temporal planning and timed automata, compared to classical planning and SAT formulas (see Table 2 for more comparisons). We now describe our approach to the verification and implementation.

Implementation We use stepwise refinement [91] of datatypes and definitions to produce provably correct executable code from abstract definitions. In this methodology, one proves the correctness of an algorithm in a number of steps: starting with an abstract version of the algorithm, where the proofs focus on mathematical ideas, and adding and proving correctness of the implementation in further steps that replace mathematical functions and relations with datatypes and executable code.

Proofs For the proof of Theorem 1, we define a list of transitions *ts*, from which we obtain a list of functions *fs*, such that each fs_i is associated with a precondition P_i and postcondition Q_i, which match those of the corresponding transition $\mathit{ts}_i \in \mathit{ts}$. We iteratively remove the first element of *fs*, apply it to the final element in *cfgs*, and append the result, beginning from $\mathit{cfgs} = \langle\langle L_I, v_I, c_I\rangle\rangle$. In the inductive step, we prove that each P_i implies Q_i after applying fs_i to cfgs_i, and that each Q_i implies P_{i+1}. Each pre- and postcondition includes the bounds for all variables. We also prove that the addition of numeric conditions and updates to the network preserves the correctness of the run at each step to obtain the full result involving numeric fluents. The additional assumptions necessary are obtained from Lemma 1.

Domain (#inst)	ours		T-CTP		T-FTP		OPTIC		nuXmv		UPPAAL	
MatchCellar (9)	(5/1/0)	**3**	(2/5/0)	**2**	(7/0/0)	**2**	(0/6/0)	**3**	(7/0/0)	**2**	(6/0/0)	**3**
SYNC (8)	(5/0/0)	**3**	(3/0/0)	**5**	(0/8/0)	**0**	(2/6/0)	**0**	(2/0/0)	**6**	(4/0/0)	**4**
PAINTER (12)	(10/1/0)	**1**	(7/0/0)	**5**	(2/10/0)	**0**	(8/2/0)	**2**	(8/0/0)	**4**	(8/0/0)	**4**
MAJSP-1 (128)	(72/51/5)	**0**	(56/60/0)	**12**	(84/44/0)	**0**	(84/43/0)	**1**	(106/0/0)	**22**	(127/0/0)	**1**
MAJSP-2 (240)	(84/82/6)	**68**	(86/66/0)	**88**	(212/28/0)	**0**	(103/37/0)	**100**	(128/0/0)	**112**	(185/0/1)	**54**
Total (397)	(176/135/11)	**75**	(154/131/0)	**112**	(305/90/0)	**2**	(197/94/0)	**106**	(251/0/0)	**146**	(330/0/1)	**66**

Table 1: For each domain/method pair: in parentheses the number of timeouts, memouts, and unknowns (crash or no verdict), followed by the number of instances proved unsolvable (**bold**, right-aligned). Here, we have our pipeline (ours) and OPTIC run on PDDL; TAMER run on ANML [82] with their CTP (T-CTP) and FTP (T-FTP) algorithms; and nuXmv and UPPAAL run on the respective encodings emitted by TAMER.

We define a reusable library of *automation lemmas* to simplify the proofs of these steps. Each postcondition Q_i is proven by proving that all assertions in a list $\langle Q_{i,0}, Q_{i,1}, \ldots \rangle$ hold. Moreover, from each precondition P_i we can deduce the truth of a list of facts $\langle P_{i,0}, P_{i,1}, \ldots \rangle$. Automation lemmas guide Isabelle's automated proof methods towards proofs, by pairing postconditions with only the relevant preconditions, i.e. trimming the search space. See Table 2 for the lines of code in each part and our supplementary material for a mapping of concepts to Isabelle definitions and lemmas.

Trust Our verified unsolvability checker (i.e. our entire pipeline: (i) unverified grounding (written based on Fast Downward [52] and Temporal Fast Downward [37]); (ii) verified encoding; (iii) unverified model checking using TChecker [56]; and (iv) verified certificate checking) places trust in: (i) the unverified grounder, which we plan to verify; (ii) Isabelle's kernel, which we trust to be sound; (iii) the code generator [46], which we trust to generate correct code with respect to abstract functional programs written in HOL; (iv) the MLton compiler, hardware and runtime, which we trust to compile and execute the generated code correctly; and (v) the PDDL parser, which we trust to produce correct abstract syntax trees (ASTs). The certificate checking stage involving printing the network in Munta's format, translating it to TChecker's format, running TChecker, translating the certificate back, and parsing the certificate does not need to be trusted, because the certificate's correctness is verified with respect to the AST of the network in memory.

Our system reports "unsolvable" only if the certificate checker accepts the certificate as sufficient to demonstrate that the grounded problem has no plan; this is machine-checked by Isabelle's kernel, based on the contrapositive of Theorem 1 together with the formally verified correctness of the certificate checker.

6 Experimental Evaluation

The experimental evaluation compares the performance of our verified unsolvability checker with other unverified planners and model checkers. We com-

pare against two unsolvability variants of the planner TAMER [86], as well as its encodings to UPPAAL [11,10] and nuXmv [21,23], which are available online [75]; and OPTIC [12]. We run the planners on the domains used by Panjkovic et al. [74], on a few identical servers (AMD EPYC 9554P, 64c/128t, 1511 GiB RAM), with a per-run limit of 48 GiB RAM and a timeout of 1800 seconds. See Table 1 and Appendix F.

The domains each present their own difficulty: (1) MatchCellar seems to be hard, because the unsolvability argument either needs to detect symmetries between mend actions [33,38,54], or relies on domain-specific information [40,77] (see Example 1), neither of which any solver uses. (2) PAINTER seems to be hard for every method, because the argument for unsolvability needs to show that the four preconditions of the join action cannot hold at the same time: nuXmv on average needs to search for more frames in PAINTER than in other domains before it finds an inductive invariant, planners like OPTIC cannot find this invariant as a heuristic and therefore cannot immediately deduce unsolvability, and ours, UPPAAL and TAMER-CTP unfold state spaces that scale multiplicatively with the number of treatments. (3) SYNC's difficulty seems to scale with the number of reachable discrete states and the number of possible interleavings of action executions. The cause of unsolvability in this domain is an impossible scheduling/time constraint, which means that a full exploration of the discrete state space is needed for methods that explore state spaces. (4) MAJSP-1 and -2 seem to be difficult, because they lead to large combinations of potentially concurrent actions, which leads to large state spaces. MAJSP-1 instances seem to be harder than MAJSP-2, because the robots can usually reach treatments in MAJSP-1 before running out of battery, which scales the number of potentially concurrent actions with the number of pallets and treatments. The cause of unsolvability is in the relationship between initial battery levels and the cost of moving.

The difficulty of MAJSP shows mainly in our method and UPPAAL, which explore states. OPTIC and nuXmv reason about battery levels symbolically, with a static numeric check and an inductive invariant respectively. We hypothesise that, unlike UPPAAL, TAMER-CTP's direct use of ANML avoids an increase in the hardness of the problem added by the translation to model checking. OPTIC's preprocessor can correctly identify 101 instances of MAJSP as unsolvable using "move cost > initial battery level", before exploring the state space. nuXmv searches for longer proofs for unsolvability in the form of an inductive invariant over comparisons between real variables [18] without exploring the state space [21], which makes it perform particularly well in SYNC and MAJSP. However, it seems to struggle to find something equivalent to $2 \cdot m < f$ for the unsolvability proof for MatchCellar, where m and f are the sizes of the sets of objects of types match and fuse, respectively. See Example 2.

PAINTER seems to be harder for our method and OPTIC than for TAMER-CTP, nuXmv and UPPAAL, because both use PDDL, which does not support the intermediate effects of the treatment action and needs four actions to represent it, unlike ANML, which these three solvers use.

7 Discussion

We presented a formally verified, executable, practical encoding of numeric temporal planning problems as networks of timed automata. Using Theorem 1 we have proven, once and for all, that an encoding from grounded planning problems can be formally verified and made executable, in a way that is sound for checking and verifying the unsolvability of any planning problem which satisfies some assumptions (namely: no self-overlap of actions, no disjunctive invariants, integer fluents, trusted grounding; see Section 5), and that in some cases it performs on par with unverified state-of-the-art planning software. Combined with a certificate checker for timed automata [90], our work presents a fully verified pipeline for showing unsolvability.

We used Heinz et al.'s encoding [50] as our starting point. Most notably, we extend it to work for problems with numeric fluents (since numeric planning with unbounded fluents is undecidable [51], we decide unsolvability for problems for which we can infer verified bounds). We also generalised its notion of unsolvability with ideas from Gigante et al. [43]. We note that there are others who used related bounding techniques in planning: Scala et al. [78] use interval arithmetic to compute a heuristic value for a state, which is also sound for reachability, Kuroiwa et al. [63] infer bounds of numeric variables and use them in heuristics for optimal linear numeric planning, and Coles et al. [26] compute bounds statically.

Other authors have designed encodings of temporal planning into timed automata [32,68,50] and explored planning using model checking [58]. However, in all these cases the main focus was solvable planning problems. For unsolvability, Panjkovic et al. [74] recently adapted the TAMER planning system [86] for detecting unsolvability using model checking techniques as well as by integrating the nuXmv [21,23] model checker. Earlier approaches that could be used to detect unsolvability are the temporal version of Fast Downward [36], a temporal version of the landmark heuristics [70], and the OPTIC planner [12].

ITPs have been used to verify related software, e.g. SAT-based planning [3], SAT encodings [25], classical and temporal plan validation [4,2], and certification for combinatorial problems [64,84,92]. We show that combining a verified encoding with a verified certificate checker, first devised by Abdulaziz and Kurz [3] for SAT encodings of classical planning, scales to source and target formalisms substantially more complex than those previously handled with ITPs in planning. The alternative is to verify an entire solver, as Esparza et al. [35] did for an executable model checker; this rules out crashes due to bugs, which can occur in our approach if the unverified model checker crashes, but verifying a solver that performs on par with unverified state-of-the-art solvers is most likely infeasible, due to the number of optimisations it uses.

As future work, we would like to verify an unsolvability checker for other richer planning formalisms, e.g. for hybrid planning, which involves verifying a certificate checker for hybrid automata [55]. Another direction is verifying a practical grounding algorithm, e.g. one based on Fast Downward's grounder [52]: until now all previous encodings assume grounded problems as input.

8 Data Availability

Detailed proofs and examples are in the appendix; the certifier, its Isabelle/HOL proofs, and the benchmark data are in the supplementary material and will be published online. We have used AI (Claude Opus 4.8 to 5.5 and Fable 5.0 to 5.1) to make minor changes and structural moves in the original propositional formalisation, and extensively to implement and prove the correctness of the numeric extension. These proofs are checked by Isabelle's kernel against a formal specification. We have also used it to lay out figures and tables in this manuscript.

References

1. Abdulaziz, M.: Plan-length bounds: Beyond 1-way dependency. In: 33rd AAAI Conf. Artif. Intell. AAAI (2019). https://doi.org/10.1609/aaai.v33i01.33017502
2. Abdulaziz, M., Koller, L.: Formal semantics and formally verified validation for temporal planning. In: 36th AAAI Conf. Artif. Intell. AAAI (2022). https://doi.org/10.1609/aaai.v36i9.21197
3. Abdulaziz, M., Kurz, F.: Formally verified SAT-Based AI planning. In: 37th AAAI Conf. Artif. Intell. AAAI (2023). https://doi.org/10.1609/aaai.v37i12.26714
4. Abdulaziz, M., Lammich, P.: A Formally Verified Validator for Classical Planning Problems and Solutions. In: 30th Int. Conf. Tools Artif. Intell. ICTAI (2018). https://doi.org/10.1109/ICTAI.2018.00079
5. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: 1990 Proc. Fifth Annu. IEEE Symp. Log. Comput. Sci. pp. 414–425 (1990). https://doi.org/10.1109/LICS.1990.113766
6. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science (1994). https://doi.org/10.1016/0304-3975(94)90010-8
7. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. 1 edn. (2009). https://doi.org/10.1017/CBO9780511804090
8. Avigad, J.: Mathematics and the formal turn (2023). https://doi.org/10.48550/arXiv.2311.00007
9. Ballarin, C.: Locales: A Module System for Mathematical Theories. J. Autom. Reason. (2014). https://doi.org/10.1007/s10817-013-9284-7
10. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL implementation secrets. In: Damm, W., Olderog, E.R. (eds.) Form. Tech. Real-Time Fault-Toler. Syst. 7th Int. Symp. FTRTFT 2002 Co-Spons. IFIP WG 22 Oldenbg. Ger. Sept. 9-12 2002 Proc. Lecture Notes in Computer Science, vol. 2469, pp. 3–22. Springer (2002). https://doi.org/10.1007/3-540-45739-9_1
11. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) Hybrid Syst. III. pp. 232–243. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). https://doi.org/10.1007/BFb0020949
12. Benton, J., Coles, A.J., Coles, A.: Temporal Planning with Preferences and Time-Dependent Continuous Costs. In: 22nd Int. Conf. Autom. Plan. Sched. ICAPS (2012), http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699
13. Blanchard, J., Briggs, D., Deka, K., Fenner, N., Forster, Y., (Skelet), G.G., House, M.L., Kadziolka, M., Kropitz, P., Ligocki, S., mxdys, Nasciszewski, M., Stérin,

T., Xu, C., Yuen, J., Zimmermann, T.: Determination of the fifth busy beaver value. In: Bhaskara, A., Czumaj, A. (eds.) Proc. 58th Annu. ACM Symp. Theory Comput. STOC 2026 Salt Lake City UT USA June 22-26 2026. pp. 565–574. ACM (2026). https://doi.org/10.1145/3798129.3806376
14. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation: Complexity, Analysis, Transformation, pp. 85–108. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-36377-7_5
15. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. J. Autom. Reason. (2018). https://doi.org/10.1007/s10817-018-9455-7
16. Bogomolov, S., Magazzeni, D., Podelski, A., Wehrle, M.: Planning as model checking in hybrid domains. In: Proc. Natl. Conf. Artif. Intell. vol. 3 (2014). https://doi.org/10.1609/aaai.v28i1.9037
17. Bouyer, P., Laroussinie, F., Markey, N., Ouaknine, J., Worrell, J.: Timed temporal logics. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models Algorithms Log. Tools - Essays Dedic. Kim Guldstrand Larsen Occas. His 60th Birthd. Lecture Notes in Computer Science, vol. 10460, pp. 211–230. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_11
18. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D.A. (eds.) Verification Model Checking Abstr. Interpret. - 12th Int. Conf. VMCAI 2011 Austin TX USA January 23-25 2011 Proc. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
19. Buzzard, K., Commelin, J., Massot, P.: Formalising perfectoid spaces. In: Proc. 9th ACM SIGPLAN Int. Conf. Certif. Programs Proofs. pp. 299–312. Cpp 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373830
20. Buzzard, K., Taylor, R.: A lean proof of fermat's last theorem. Tech. rep., Imperial College of London (2024), https://imperialcollegelondon.github.io/FLT/blueprint.pdf
21. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: Comput. Aided Verification - 26th Int. Conf. CAV 2014 Held Part Vienna Summer Log. VSL 2014 Vienna Austria July 18-22 2014 Proc. (2014). https://doi.org/10.1007/978-3-319-08867-9_22
22. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**(2), 56–68 (1940). https://doi.org/10.2307/2266170
23. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: Extending nuXmv with Timed Transition Systems and Timed Temporal Properties. In: Dillig, I., Tasiran, S. (eds.) Comput. Aided Verification. pp. 376–386. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_21
24. Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction to model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 1–26. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_1
25. Codel, C., Avigad, J., Heule, M.: Verified Encodings for SAT Solvers. In: Formal Methods in Computer-Aided Design (2023). https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_22

26. Coles, A.J., Coles, A., Fox, M., Long, D.: A hybrid LP-RPG heuristic for modelling numeric resource flows in planning. J. Artif. Intell. Res. **46**, 343–412 (2013). https://doi.org/10.1613/JAIR.3788
27. Coles, A.J., Coles, A., Olaya, A.G., Celorrio, S.J., López, C.L., Sanner, S., Yoon, S.: A Survey of the Seventh International Planning Competition. AI Mag. (2012). https://doi.org/10.1609/aimag.v33i1.2392
28. Corrêa, A.B., Giacomo, G.D.: Lifted planning: Recent advances in planning using first-order representations. In: Proc. Thirty-Third Int. Jt. Conf. Artif. Intell. IJCAI 2024 Jeju South Korea August 3-9 2024. pp. 8010–8019. ijcai.org (2024), https://www.ijcai.org/proceedings/2024/886
29. Corrêa, A.B., Hecher, M., Helmert, M., Longo, D.M., Pommerening, F., Woltran, S.: Grounding Planning Tasks Using Tree Decompositions and Iterated Solving. Proc. Int. Conf. Autom. Plan. Sched. ICAPS **33**, 100–108 (Jul 2023). https://doi.org/10.1609/icaps.v33i1.27184
30. Corrêa, A.B., Pommerening, F., Helmert, M., Francès, G.: Lifted successor generation using query optimization techniques. In: Beck, J.C., Buffet, O., Hoffmann, J., Karpas, E., Sohrabi, S. (eds.) Proc. Thirtieth Int. Conf. Autom. Plan. Sched. Nancy Fr. Oct. 26-30 2020. pp. 80–89. AAAI Press (2020), https://ojs.aaai.org/index.php/ICAPS/article/view/6648
31. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang. pp. 238–252. Popl '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973
32. Dierks, H., Behrmann, G., Larsen, K.G.: Solving Planning Problems Using Real-Time Model Checking. AIPS 2002 Workshop Plan. Model Checking (2002)
33. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Form. Methods Syst. Des. (1996). https://doi.org/10.1007/BF00625970
34. Eriksson, S., Röger, G., Helmert, M.: Unsolvability certificates for classical planning. In: Barbulescu, L., Frank, J., Mausam, Smith, S.F. (eds.) Proc. Twenty-Seventh Int. Conf. Autom. Plan. Sched. ICAPS 2017 Pittsburgh Pa. USA June 18-23 2017. pp. 88–97. AAAI Press (2017), https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/view/15734
35. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A Fully Verified Executable LTL Model Checker. In: 25th Int. Conf. Comput. Aided Verification CAV (2013). https://doi.org/10.1007/978-3-642-39799-8_31
36. Eyerich, P., Mattmüller, R., Röger, G.: Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. Proc. Int. Conf. Autom. Plan. Sched. (2009). https://doi.org/10.1609/icaps.v19i1.13373
37. Eyerich, P., Mattmüller, R., Röger, G.: Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning. In: Towards Service Robots for Everyday Environments - Recent Advances in Designing Service Robots for Complex Tasks in Everyday Environments (2012). https://doi.org/10.1007/978-3-642-25116-0_6
38. Fox, M., Long, D.: The detection and exploitation of symmetry in planning problems. In: Proc. Sixt. Int. Jt. Conf. Artif. Intell. IJCAI 99 Stockh. Swed. July 31 - August 6 1999 2 Vol. 1450 Pages (1999)
39. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. J. Artif. Intell. Res. (2003), http://arxiv.org/abs/1106.4561

40. Fuentetaja, R., de la Rosa, T.: Compiling irrelevant objects to counters. Special case of creation planning. Ai Commun. **29**(3), 435–467 (2016). https://doi.org/10.3233/AIC-150692
41. Gerevini, A., Haslum, P., Long, D., Saetti, A., Dimopoulos, Y.: Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. Artif. Intell. **173**(5-6), 619–668 (2009). https://doi.org/10.1016/J.ARTINT.2008.10.012
42. Gerevini, A., Long, D.: Plan Constraints and Preferences in PDDL3
43. Gigante, N., Micheli, A., Montanari, A., Scala, E.: Decidability and complexity of action-based temporal planning over dense time. Artificial Intelligence (2022). https://doi.org/10.1016/j.artint.2022.103686
44. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In: Kapur, D. (ed.) Comput. Math. 8th Asian Symp. ASCM 2007 Singap. Dec. 15-17 2007 Revis. Invit. Pap. Lecture Notes in Computer Science, vol. 5081, p. 333. Springer (2007). https://doi.org/10.1007/978-3-540-87827-8_28
45. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A machine-checked proof of the odd order theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interact. Theorem Proving. pp. 163–179. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
46. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) Funct. Log. Program. 10th Int. Symp. FLOPS 2010 Sendai Jpn. April 19-21 2010 Proc. Lecture Notes in Computer Science, vol. 6009, pp. 103–117. Springer (2010). https://doi.org/10.1007/978-3-642-12251-4_9
47. Halbwachs, N., Proy, Y.E., Roumanoff, P.: Verification of Real-Time Systems using Linear Relation Analysis. Form. Methods Syst. Des. **11**(2), 157–185 (Aug 1997). https://doi.org/10.1023/A:1008678014487
48. Hales, T., Adams, M., Bauer, G., Dang, T.D., Harrison, J., Hoang, L.T., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, Q.T., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, T.H.A., Tran, N.T., Trieu, T.D., Urban, J., Vu, K., Zumkeller, R.: A Formal Proof of the Kepler Conjecture. Forum Math. Pi **5**, e2 (2017). https://doi.org/10.1017/fmp.2017.1
49. Haslum, P.: Patrikhaslum/INVAL (2024), https://github.com/patrikhaslum/INVAL
50. Heinz, A., Wehrle, M., Bogomolov, S., Magazzeni, D., Greitschus, M., Podelski, A.: Temporal Planning as Refinement-Based Model Checking. In: Proc. Twenty-Ninth Int. Conf. Autom. Plan. Sched. ICAPS 2018 Berkeley CA USA July 11-15 2019 (2019), https://ojs.aaai.org/index.php/ICAPS/article/view/3476
51. Helmert, M.: Decidability and undecidability results for planning with numerical state variables. In: Ghallab, M., Hertzberg, J., Traverso, P. (eds.) Proc. Sixth Int. Conf. Artif. Intell. Plan. Syst. April 23-27 2002 Toulouse Fr. pp. 44–53. AAAI (2002), http://www.aaai.org/Library/AIPS/2002/aips02-005.php
52. Helmert, M.: The Fast Downward Planning System. J. Artif. Intell. Res. (2006). https://doi.org/10.1613/jair.1705
53. Helmert, M.: Concise finite-domain representations for PDDL planning tasks. Artif. Intell. (2009). https://doi.org/10.1016/j.artint.2008.10.013
54. Hendriks, M., Behrmann, G., Larsen, K.G., Niebert, P., Vaandrager, F.W.: Adding symmetry reduction to uppaal. In: Larsen, K.G., Niebert, P. (eds.) Form. Model. Anal. Timed Syst. First Int. Workshop FORMATS 2003 Marseille Fr. Sept. 6-7

2003 Revis. Pap. Lecture Notes in Computer Science, vol. 2791, pp. 46–59. Springer (2003). https://doi.org/10.1007/978-3-540-40903-8_5
55. Henzinger, T.: The theory of hybrid automata. In: Proc. 11th Annu. IEEE Symp. Log. Comput. Sci. (1996). https://doi.org/10.1109/LICS.1996.561342
56. Herbreteau, F., Point, G., Sankur, O.: Ticktac-project/tchecker. TickTac Project (2023), https://github.com/ticktac-project/tchecker
57. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. Inf. Comput. **251**, 67–90 (2016). https://doi.org/10.1016/j.ic.2016.07.004
58. Hoffmann, J., Hermanns, H., Klauck, M., Steinmetz, M., Karpas, E., Magazzeni, D.: Let's learn their language? A case for planning with automata-network languages from model checking. In: Thirty-Fourth AAAI Conf. Artif. Intell. AAAI 2020 Thirty-Second Innov. Appl. Artif. Intell. Conf. IAAI 2020 Tenth AAAI Symp. Educ. Adv. Artif. Intell. EAAI 2020 N. Y. NY USA Febr. 7-12 2020. pp. 13569–13575. AAAI Press (2020). https://doi.org/10.1609/AAAI.V34I09.7083
59. Howey, R., Long, D., Fox, M.: VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In: 16th IEEE Int. Conf. Tools Artif. Intell. ICTAI (2004). https://doi.org/10.1109/ICTAI.2004.120
60. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales - A Sectioning Concept for Isabelle. In: Theorem Proving High. Order Log. 12th Int. Conf. TPHOLs99 Nice Fr. Sept. 1999 Proc. (1999). https://doi.org/10.1007/3-540-48256-3_11
61. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd ACM Symp. Oper. Syst. Princ. 2009 SOSP (2009). https://doi.org/10.1145/1629575.1629596
62. Krauss, A.: Automating Recursive Definitions and Termination Proofs in Higher-Order Logic. Ph.D. thesis, Technical University Munich (2009), http://mediatum2.ub.tum.de/doc/681651/document.pdf
63. Kuroiwa, R., Shleyfman, A., Beck, J.C.: Extracting and exploiting bounds of numeric variables for optimal linear numeric planning. In: Gal, K., Nowé, A., Nalepa, G.J., Fairstein, R., Radulescu, R. (eds.) ECAI 2023 - 26th Eur. Conf. Artif. Intell. Sept. 30 - Oct. 4 2023 Kraków Pol. - 12th Conf. Prestig. Appl. Intell. Syst. PAIS 2023. Frontiers in Artificial Intelligence and Applications, vol. 372, pp. 1332–1339. IOS Press (2023). https://doi.org/10.3233/FAIA230409
64. Lammich, P.: The GRAT Tool Chain - Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees. In: 20th Int. Conf. Theory Appl. Satisf. Test. SAT (2017). https://doi.org/10.1007/978-3-319-66263-3_29
65. Lammich, P.: Refinement to Imperative HOL. J. Autom. Reason. (2019). https://doi.org/10.1007/s10817-017-9437-1
66. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Interact. Theorem Proving (2010). https://doi.org/10.1007/978-3-642-14052-5_24
67. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1-2) (1997). https://doi.org/10.1007/s100090050010
68. Li, Y., Sun, J., Dong, J.S., Liu, Y., Sun, J.: Planning as Model Checking Tasks. In: 2012 35th Annu. IEEE Softw. Eng. Workshop. pp. 177–186 (Oct 2012). https://doi.org/10.1109/SEW.2012.25
69. Long, D., Fox, M.: The 3rd international planning competition: Results and analysis. J. Artif. Intell. Res. **20**, 1–59 (2003). https://doi.org/10.1613/JAIR.1240
70. Marzal, E., Sebastia, L., Onaindia, E.: Detection of unsolvable temporal planning problems through the use of landmarks. In: Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N.M. (eds.) ECAI 2008 - 18th Eur. Conf. Artif. Intell. Patras

Greece July 21-25 2008 Proc. Frontiers in Artificial Intelligence and Applications, vol. 178, pp. 919–920. IOS Press (2008). https://doi.org/10.3233/978-1-58603-891-5-919
71. McCarthy, J., Hayes, P.J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. Mach. Intell. 4 (1969)
72. McDermott, D.V.: The 1998 AI Planning Systems Competition. AI Mag. (2000). https://doi.org/10.1609/aimag.v21i2.1506
73. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic (2002). https://doi.org/10.1007/3-540-45949-9
74. Panjkovic, S., Micheli, A., Cimatti, A.: Deciding Unsolvability in Temporal Planning under Action Non-Self-Overlapping. In: 36th AAAI Conf. Artif. Intell. AAAI (2022). https://doi.org/10.1609/AAAI.V36I9.21225
75. Panjkovic, S., Micheli, A., Cimatti, A.: Deciding Unsolvability in Temporal Planning under Action Non-Self-Overlapping (artifact) (2022), https://es-static.fbk.eu/people/amicheli/resources/aaai22/
76. Paulson, L.C.: A formulation of the simple theory of types (for Isabelle). In: Martin-Löf, P., Mints, G. (eds.) COLOG-88 Int. Conf. Comput. Log. Tallinn USSR Dec. 1988 Proc. Lecture Notes in Computer Science, vol. 417, pp. 246–274. Springer (1988). https://doi.org/10.1007/3-540-52335-9_58
77. Riddle, P., Douglas, J., Barley, M., Franco, S.: Improving performance by reformulating PDDL into a bagged representation. In: ICAPS 2016 Workshop Heuristics Search Domain-Indep. Plan. (2016)
78. Scala, E., Haslum, P., Thiébaux, S., Ramírez, M.: Interval-based relaxation for general numeric planning. In: Kaminka, G.A., Fox, M., Bouquet, P., Hüllermeier, E., Dignum, V., Dignum, F., van Harmelen, F. (eds.) ECAI 2016 - 22nd Eur. Conf. Artif. Intell. 29 August-2 Sept. 2016 Hague Neth. - Prestig. Appl. Artif. Intell. PAIS 2016. Frontiers in Artificial Intelligence and Applications, vol. 285, pp. 655–663. IOS Press (2016). https://doi.org/10.3233/978-1-61499-672-9-655
79. Schäffeler, M., Abdulaziz, M.: Markov decision processes with rewards (Dec 2021), https://isa-afp.org/entries/MDP-Rewards.html
80. Schäffeler, M., Abdulaziz, M.: Verified algorithms for solving markov decision processes (Dec 2021), https://isa-afp.org/entries/MDP-Algorithms.html
81. Schäffeler, M., Abdulaziz, M.: Formally Verified Approximate Policy Iteration. In: 38th AAAI Conf. Artif. Intell. AAAI (2025). https://doi.org/10.48550/arXiv.2406.07340
82. Smith, D.E., Frank, J., Cushing, W.: The ANML language. In: KEPS (2008)
83. Taitler, A., Alford, R., Espasa, J., Behnke, G., Fi*v*ser, D., Gimelfarb, M., Pommerening, F., Sanner, S., Scala, E., Schreiber, D., Segovia-Aguas, J., Seipp, J.: The 2023 International Planning Competition. AI Mag. (2024). https://doi.org/10.1002/aaai.12169
84. Tan, Y.K., Yang, J., Soos, M., Myreen, M.O., Meel, K.S.: Formally Certified Approximate Model Counting. In: 38th Int. Conf. Comput. Aided Verification CAV (2024). https://doi.org/10.1007/978-3-031-65627-9_8
85. Traytel, D., Popescu, A., Blanchette, J.C.: Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In: 27th Annu. IEEE Symp. Log. Comput. Sci. LICS (2012). https://doi.org/10.1109/LICS.2012.75
86. Valentini, A., Micheli, A., Cimatti, A.: Temporal Planning with Intermediate Conditions and Effects. Proc. AAAI Conf. Artif. Intell. (2020). https://doi.org/10.1609/aaai.v34i06.6553

87. Vallati, M., Chrpa, L., Grzes, M., McCluskey, T.L., Roberts, M., Sanner, S.: The 2014 International Planning Competition: Progress and Trends. AI Mag. (2015). https://doi.org/10.1609/aimag.v36i3.2571
88. Wimmer, S.: Timed Automata (AFP). Archive of Formal Proofs (2016), https://isa-afp.org/entries/Timed_Automata.html
89. Wimmer, S.: Munta: A verified model checker for timed automata. In: Form. Model. Anal. Timed Syst. 17th Int. Conf. FORMATS 2019 Amst. Neth. August 27–29 2019 Proc. pp. 236–243. Springer-Verlag, Berlin, Heidelberg (2019). https://doi.org/10.1007/978-3-030-29662-9_14
90. Wimmer, S., von Mutius, J.: Verified Certification of Reachability Checking for Timed Automata. In: 26th Int. Conf. Tools Algorithms Constr. Anal. Syst. TACAS (2020). https://doi.org/10.1007/978-3-030-45190-5_24
91. Wirth, N.: Program Development by Stepwise Refinement. Commun. ACM (1971). https://doi.org/10.1145/362575.362577
92. Yang, J., Tan, Y.K., Soos, M., Myreen, M.O., Meel, K.S.: Efficient Certified Reasoning for Binarized Neural Networks. In: 28th Int. Conf. Theory Appl. Satisf. Test. SAT (2025). https://doi.org/10.4230/LIPICS.SAT.2025.32

A MatchCellar Example

Definition 23 (MatchCellar Template). *The template is parameterised by (i) an indexed set N_m of matches, (ii) an indexed set N_f of fuses, (iii) a duration for the actions involving matches and fuses, namely d^{lb}_{lit} and d^{ub}_{lit} for the duration range a match burns and d_{mend} for the duration needed to mend a fuse, and (iv) a number of spare fuses n_{spare}. This is in fact possible in standard PDDL. See Listing 1.1.*

We identify objects using shorthand notation and indices. We obtain a planning problem $\Pi \equiv \langle P, F, A, I, G\rangle$, where (1) $P = \{$handfree, unused(m_i), light(m_i), mended$(f_j)\}$, for $i \in N_m$ and $j \in N_f$; (2) $F = \{$dur-lit-lb(m_i), dur-lit-ub(m_i), dur-mend(f_j), spare-fuses, matches-lit$\}$; (3) $A = \{(lm_i \equiv$ light-match$(m_i))$, $(mf_{ji} \equiv$ mend-fuse$(f_j, m_i))\}$, for $i \in N_m$ and $j \in N_f$, with conditions and effects (3.a) $lm_i \equiv \langle\langle\{$unused$(m_i)\}$, $\top$, $\{$light$(m_i)\}$, $\{$unused$(m_i)\}$, $\{$matches-lit $:=$ matches-lit $+ 1\}\rangle$, $\langle\emptyset$, $\top$, $\emptyset$, $\{$light$(m_i)\}$, $\emptyset\rangle$, $\emptyset$, $\top$, $\langle$dur-lit-lb(m_i), $\leq\rangle$, $\langle$dur-lit-ub(m_i), $\leq\rangle\rangle$; and (3.b) $mf_{ji} \equiv \langle\langle\{$handfree$\}$, spare-fuses > 0, $\emptyset$, $\{$handfree$\}$, $\{$spare-fuses $:=$ spare-fuses $- 1\}\rangle$, $\langle\emptyset$, $\top$, $\{$handfree, mended$(f_j)\}$, $\emptyset$, $\emptyset\rangle$, $\{$light$(m_i)\}$, $\top$, $\langle$dur-mend(f_j), $\leq\rangle$, $\langle$dur-mend(f_j), $\leq\rangle\rangle$; (4) the initial state $I \equiv \langle I_P, I_F\rangle$ is characterised by (4.a) $I_F($spare-fuses$) = n_{spare}$; (4.b) $I_F($matches-lit$) = 0$; (4.c) $I_F($dur-lit-lb$(m_i)) = d^{lb}_{lit}$ and $I_F($dur-lit-ub$(m_i)) = d^{ub}_{lit}$ for $i \in N_m$; (4.d) $I_F($dur-mend$(f_j)) = d_{mend}$ for $j \in N_f$; and (4.e) $I_P \equiv \{$handfree, unused$(m_i) \mid i \in N_m\}$; and (5) the goal is $G \equiv \langle\{$mended$(f_j) \mid j \in N_f\}, \top\rangle$.

Definition 24 (MatchCellar Example Instance). *This can be constructed based on Definition 23 by setting (i) both the set of identifiers for matches N_m and that for fuses N_f to $\{0\}$, (ii) the number of spare fuses n_{spare} to 2, and (iii) the burn duration bounds d^{lb}_{lit} to 5 and d^{ub}_{lit} to 6, and the mend duration d_{mend} to 5.*

Definition 25 (MatchCellar Unsolvable Instance). *This example can be derived from Definition 23. Disregard the number of spare fuses (or set it to a value greater than 3) and the number of lit matches, as these are irrelevant for the planning instance.*

Set the remaining parameters to: 1 match, 3 fuses, a burn duration of 5 ($d^{lb}_{lit} = d^{ub}_{lit} = 5$), and a mend duration of 2.

Listing 1.1: MatchCellar domain (Definition 23)

```
(define (domain matchcellar)
    (:requirements :typing :durative-actions :numeric-fluents
        )
    (:types
        match fuse
    )
    (:predicates
        (handfree)
```

```
        (unused ?match - match)
        (mended ?fuse - fuse)
        (light ?match - match)
    )
    (:functions
        (dur-lit-lb ?x - match)
        (dur-lit-ub ?x - match)
        (dur-mend ?x - fuse)
        (spare-fuses)
        (matches-lit)
    )

    (:durative-action LIGHT_MATCH
        :parameters (?match - match)
        :duration (and
            (>= ?duration (dur-lit-lb ?match))
            (<= ?duration (dur-lit-ub ?match)))
        :condition (and
            (at start (unused ?match)))
        :effect (and
            (at start (not (unused ?match)))
            (at start (light ?match))
            (at start (assign (matches-lit) (+ (matches-lit)
                1)))
            (at end (not (light ?match))))
    )

    (:durative-action MEND_FUSE
        :parameters (?fuse - fuse ?match - match)
        :duration (= ?duration (dur-mend ?fuse))
        :condition (and
            (at start (handfree))
            (over all (light ?match))
            (at start (> (spare-fuses) 0)))
        :effect (and
            (at start (not (handfree)))
            (at start (decrease (spare-fuses) 1))
            (at end (mended ?fuse))
            (at end (handfree)))
    )
)
```

Listing 1.2: Instance of Example 1

```
(define (problem instance)
 (:domain matchcellar)
 (:objects
    match0 - match
    fuse0 - fuse
)
```

```
 (:init
  (handfree)
  (unused match0)
  (= (dur-lit-lb match0) 5)
  (= (dur-lit-ub match0) 6)
  (= (dur-mend fuse0) 5)
  (= (spare-fuses) 2)
  (= (matches-lit) 0)
)
 (:goal
  (and
     (mended fuse0)
))
 (:metric minimize (matches-lit))
)
```

Listing 1.3: Instance of Example 2

```
(define (problem instance)
 (:domain matchcellar)
 (:objects
    match0 - match
    fuse0 fuse1 fuse2 - fuse
)
 (:init
  (handfree)
  (unused match0)
  (= (dur-lit-lb match0) 5)
  (= (dur-lit-ub match0) 5)
  (= (dur-mend fuse0) 2)
  (= (dur-mend fuse1) 2)
  (= (dur-mend fuse2) 2)
  (= (spare-fuses) 3)
  (= (matches-lit) 0)
)
 (:goal
  (and
     (mended fuse0)
     (mended fuse1)
     (mended fuse2)
))
 (:metric minimize (matches-lit))
)
```

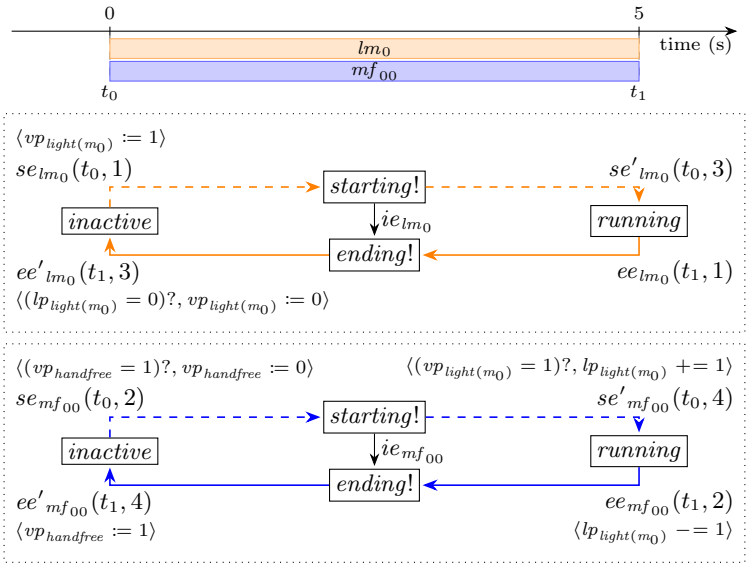


Fig. 4: Automata for the actions of the plan in Figure 2. Dashed edges are applied to simulate t_0 (----) and solid ones to simulate t_1 (——). The order is indicated with respect to each time point t_i as (t_i, n).

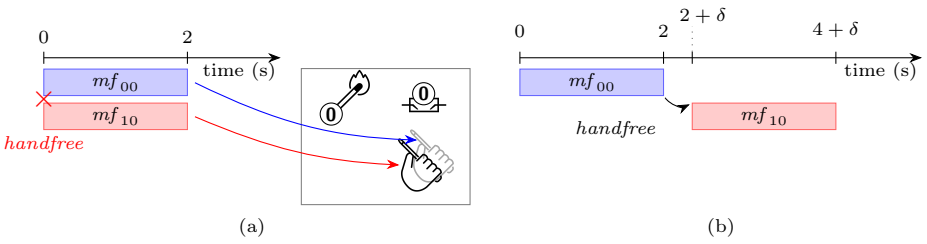


Fig. 5: Interfering actions of Example 2. (a) The start snap actions of two mend actions both require and delete *handfree*, so they interfere and cannot be scheduled at the same time point. (b) The second mend can start δ after the first has ended and added *handfree* again.

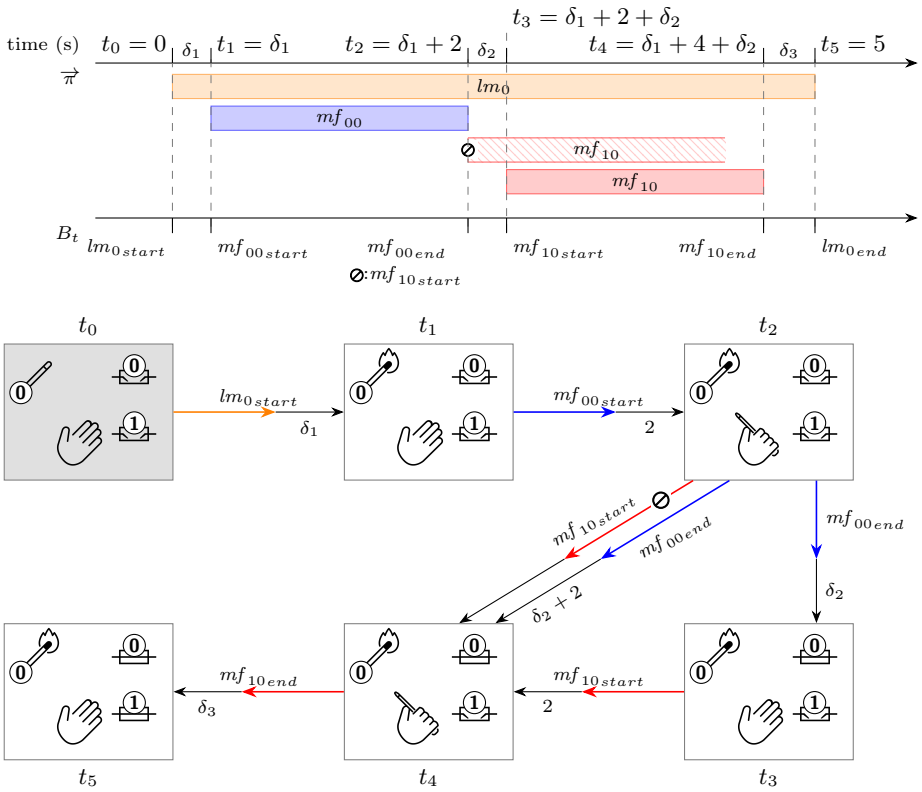


Fig. 6: Timeline (top) and partial state space (bottom) of Example 2 with transitions representing executions of durative actions and time delays δ_1, δ_2, δ_3. Each state is shown at its time point before the snap actions of that time point are applied. Starting mf_{10} at t_2, when mf_{00} ends, is forbidden according to Definition 13 (hatched box in the timeline; in the state space, the pair of transitions from t_2 to t_4): $mf_{00_{end}}$ adds *handfree*, which $mf_{10_{start}}$ requires, so the second mend can only start after a delay δ_2 (t_3).

B Propositional Example

Example 3 (Rooms, Robots, Doors, Blocks (Intuition)). We provide an example of a temporal planning problem concerning rooms (denoted as rm), robots (denoted as rb), doors (denoted as d), and blocks (denoted as b).

We refer to propositions in text by treating a descriptive word as a function and objects as arguments in a function call. For example, *"robot number 1 is holding block number 1"* is $holds(rb_1, b_1)$.

Similarly, we treat actions as functions and use objects as arguments. For example, the action of *"robot number 1 picking up the block from within room 1"* is: $pick\text{-}up(rb_1,\ b,\ rm_1) \equiv \langle\langle\{idle(rb_1),\ in(b,\ rm_1),\ battery\text{-}level(rb_1) - cost\text{-}pick\text{-}up(b) \geq 0\}, \emptyset, \{idle(rb_1), in(b, rm_1)\}\rangle, \{in(rb_1, rm_1)\}, \langle\emptyset, \{idle(rb_1), holds(rb_1, b)\}, \emptyset\rangle, \langle 4, \leq\rangle, \langle 4, \leq\rangle\rangle$.

This resembles PDDL's syntax with commas and parentheses instead of spaces to separate arguments.

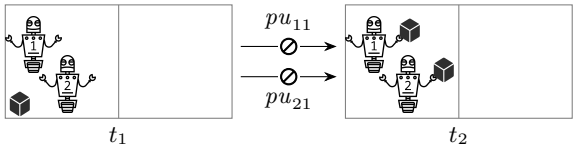


Fig. 7: Interfering actions in Example 4. Two robots picking up the same block. Both starting snap actions are conditioned on $in(b, rm_1)$ and delete $in(b, rm_1)$.

Example 4 (Rooms, Robots, Doors, Blocks (Formalism)). We identify objects using shorthand notation and indices, and thereby identify some propositions. The set of relevant propositions P in our problem is: $\{idle(rb_i),\ closed(d),\ open(d),\ in(rb_i,\ rm_j),\ in(b,\ rm_j),\ holds(rb_i,\ b),\ connects(d,\ rm_i,\ rm_j)\}$, for $i \in \{1, 2\}$ and $j \in \{1, 2\}$.

The set of numeric fluents in our problem is: $\{battery\text{-}level(rb_i),\ cost\text{-}move(rm_j,\ rm_k),\ cost\text{-}pick\text{-}up(b),\ cost\text{-}open(d)\}$, for $i \in \{1, 2\}$, $j \in \{1, 2\}$ and $k \in \{1, 2\}$, where $cost\text{-}move(rm_j, rm_k) = 2$.

The set of actions A is: $\{(pu_{ij} \equiv pick\text{-}up(rb_i, b, rm_j)), (pd_{ij} \equiv put\text{-}down(rb_i,\ b,\ rm_j)),\ (od_{ij} \equiv open\text{-}door(rb_i,\ d,\ rm_j)),\ (cd_{ij} \equiv close\text{-}door(rb_i,\ d,\ rm_j)),\ (mv_{ijk} \equiv move(rb_i, d, rm_j, rm_k))\}$, for $i \in \{1, 2\}$, $j \in \{1, 2\}$ and $k \in \{1, 2\}$.

The precise conditions and effects are:
$\{(od_{ij} \equiv \langle\langle\{idle(rb_i),\ closed(d),\ battery\text{-}level(rb_i) - cost\text{-}open(d) \geq 0\},\ \emptyset,\ \{idle(rb_i),\ closed(d)\}\rangle,\ \{in(rb_i,\ rm_j)\},\ \langle\emptyset,\ \{idle(rb_i),\ open(d)\},\ \emptyset\rangle,\ \langle 3,\ \leq\rangle,\ \langle 3,\ \leq\rangle\rangle),\ (pd_{ij} \equiv \langle\langle\{idle(rb_i),\ holds(rb_i,\ b)\},\ \emptyset,\ \{idle(rb_i),\ holds(rb_i,\ b)\}\rangle,\ \{in(rb_i,\ rm_j)\},\ \langle\emptyset,\ \{idle(rb_i),\ in(b,\ rm_j)\},\ \emptyset\rangle,\ \langle 4,\ \leq\rangle,\ \langle 4,\ \leq\rangle\rangle),\ (cd_{ij} \equiv \langle\langle\{idle(rb_i),\ open(d)\},\ \emptyset,\ \{idle(rb_i),\ open(d)\}\rangle,\ \{in(rb_i,\ rm_j)\},\ \langle\emptyset,\ \{idle(rb_i),\ closed(d)\},$

$\emptyset\rangle$, $\langle 3, \leq\rangle$, $\langle 3, \leq\rangle\rangle$), ($mv_{ijk} \equiv \langle\langle\{idle(rb_i),\ in(rb_i,\ rm_j),\ connects(d,\ rm_j,\ rm_k),$ $battery\text{-}level(rb_i) - cost\text{-}move(rm_j,\ rm_k) \geq 0\},\ \emptyset,\ \{idle(rb_i),\ in(rb_i,\ rm_j)\}\rangle,$ $\{open(d)\}, \langle\emptyset, \{idle(rb_i), in(rb_i, rm_k)\}, \emptyset\rangle, \langle 2, \leq\rangle, \langle 5, \leq\rangle\rangle$)

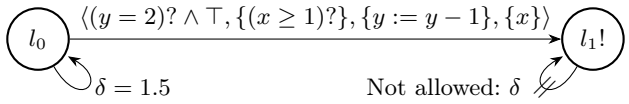
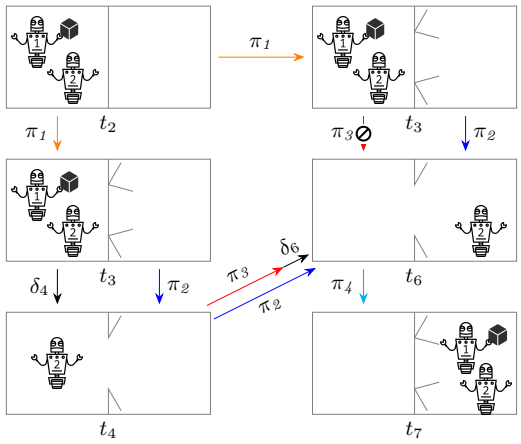


Fig. 8: A timed automaton with named locations. The action transition checks the condition $(y = 2)? \wedge \top$ and the guard $\{(x \geq 1)?\}$, applies the update $\{y := y - 1\}$, and resets $\{x\}$.

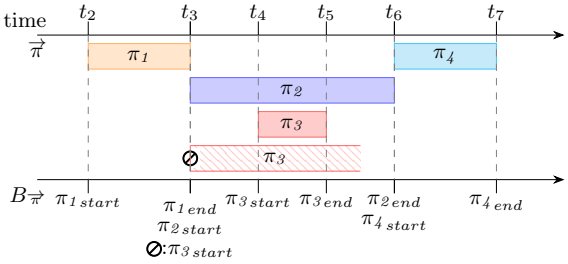
Example 5 (Rooms, Robots, Doors, Blocks (Encoding)). Continuing our example, we assign an index i to some actions to uniquely identify them:
$(\pi_1 \equiv od_{21} \equiv open\text{-}door(rb_2, d, rm_1)), (\pi_2 \equiv mv_{112} \equiv move(rb_1, rm_1, rm_2, d)),$
$(\pi_3 \equiv mv_{212} \equiv move(rb_2, rm_1, rm_2, d)), (\pi_4 \equiv cd_{22} \equiv close\text{-}door(rb_2, d, rm_2))$.

The top right state in Figure 9a characterises the instant in which π_1 ends. Because $\pi_{1\,end}$ has not been applied, neither $open(d)$ nor $closed(d)$ is satisfied. π_3 cannot start executing when π_1 ends, because $\pi_{1\,end}$ adds $idle(rb_2)$ while $\pi_{3\,start}$ deletes $idle(rb_2)$, which is invalid according to Definition 13. This invalidates any immediate application of $\pi_{3\,start}$ to the top right state labelled t_3 to eventually obtain the state at t_6. A valid alternative is shown in the left of Figure 9a. A delay of $\delta_4 \equiv t_4 - t_3$ must separate $\pi_{1\,end}$ from $\pi_{3\,start}$, which can be seen in the transition from t_3 to t_4 in the left. Similarly, a delay of $\delta_6 \equiv t_6 - t_5$ must separate the end of π_3 from the start of π_4. On the other hand, since π_2 checks $open(d)$ in $\pi_{2\,inv}$ but neither in $\pi_{2\,start}$ nor $\pi_{2\,end}$'s precondition, π_2's execution can be started at t_3 and ended at t_6.

Example 6 (Rooms, Robots, Doors, Blocks (Transition Order)). In Figure 10 we see an example of a few transitions' conditions and effects in automata for actions from Figure 9. We can see that some transitions to simulate snap action executions at a time point must occur in a certain order. ee'_{π_1} sets $vp_{open(d)}$ to 1 at t_2 and $vp_{open(d)}$ must equal 1 before se'_{π_2} can be taken. Since se'_{π_2} has incremented $lp_{open(d)}$ by 1 it must be decremented by ee_{π_2} before se_{π_4} can be taken at t_6.



(a) Partial state space with transitions representing executions of durative actions π and time delays δ. A forbidden transition according to Definition 13 representing the start of π_3 is shown at t_3 in the top right.



(b) Timeline illustrating valid and invalid schedules for actions.

Fig. 9: Parts of candidate plans for Example 4.

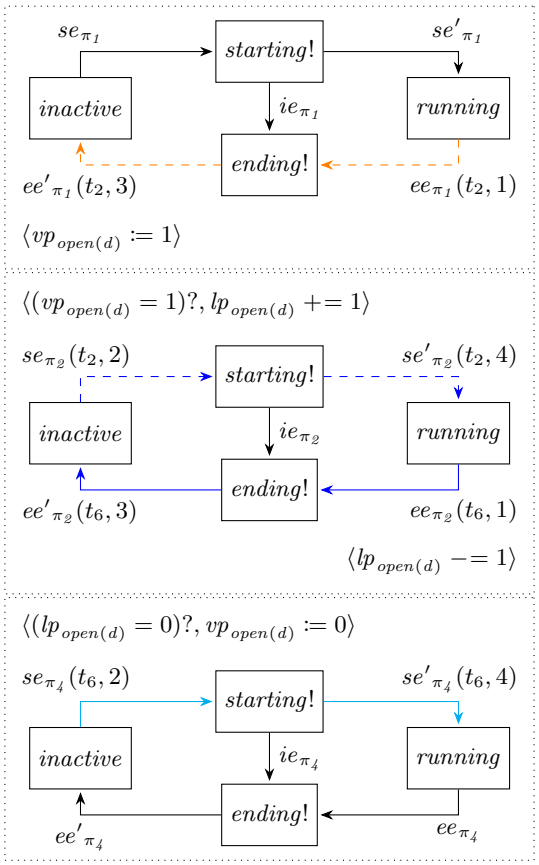


Fig. 10: Examples of automata for actions from Figure 9. Dashed edges are applied to simulate t_2 (----) and solid ones to simulate t_6 (——). The order is indicated with respect to each time point.

C Proof of Theorem 1

We now construct a run $\langle\langle L_{i;0}, v_{i;0}, c_{i;0}\rangle, \ldots, \langle L_{i;l_i}, v_{i;l_i}, c_{i;l_i}\rangle\rangle$ for every happening time point t_i, which satisfies *encodes-before*$(\langle L_I, v_I, c_I\rangle, i, \langle L_{i;0}, v_{i;0}, c_{i;0}\rangle)$ and *enc-after*$(\langle L_I, v_I, c_I\rangle, i, \langle L_{i;l_i}, v_{i;l_i}, c_{i;l_i}\rangle)$, where l_i indicates the length of the run.

Furthermore, if $0 < i$, then applying a delay of $\delta_i = t_i - t_{i-1}$ to $\langle L_{i-1;l_{i-1}}, v_{i-1;l_{i-1}}, c_{i-1;l_{i-1}}\rangle$ creates $\langle L'_{i-1;l_{i-1}}, v'_{i-1;l_{i-1}}, c'_{i-1;l_{i-1}}\rangle$, which is equivalent to $\langle L_{i;0}, v_{i;0}, c_{i;0}\rangle$ according to the conditions in Definition 22. E.g. $\langle L_{i-1;l_{i-1}}, v_{i-1;l_{i-1}}, c_{i-1;l_{i-1}}\rangle \longrightarrow_{\delta_i} \langle L_{i;0}, v_{i;0}, c_{i;0}\rangle$.

We first give a more precise characterisation of encoding the state of a world just before a set of snap actions at a time point t_i is applied.

Definition 26 (Encoded states before). *From enc-after*$(\langle L_I, v_I, c_I\rangle, i, \langle L, v, c\rangle)$ *(Definition 22) we derive encodes-before*$(\langle L_I, v_I, c_I\rangle, i, \langle L, v, c\rangle)$ *by substituting* M_i *for* M_{i+1}*, runs-at*$(\pi, t_i, <)$ *for runs-at*$(\pi, t_i, \leq)$*, and ran-for-at*$(\alpha, t_i, <)$ *for ran-for-at*$(\alpha, t_i, \leq)$*.*

Definition 27 (Encoded initial state). *When the htp sequence is empty, enc-after*$(\langle L_I, v_I, c_I\rangle, i, \langle L_i, v_i, c_i\rangle)$ *holds for any* i *if the following hold: (i) For any* $\pi_j \in A$*: (i.a)* L_i *records* l_j *as inactive*$_\pi$*; and (i.b)* $0 < c_i(x_{\pi_{j_{start}}})$ *and* $0 < c_i(x_{\pi_{j_{end}}})$*; (ii) For any* $p \in P$*: (ii.a)* $v_i(vp_p)$ *evaluates to* 1 *if* $p \in I_P$ *and* 0 *if* $p \notin I_P$*; and (ii.b)* $v_i(lp_p)$ *evaluates to* 0*. (iii) For any fluent* $f \in F$*: (iii.a)* $v_i(f)$ *evaluates to* $I_F(f)$*.*

Lemma 1. *Take box to be the environment returned by* infer-bounds *(Listing 9). Take a numeric state* M_F *whose assignment* $M_F(f) \in box(f)$ *within the bounds* $box(f)$ *for any fluent* $f \in F$*. Assume* M_F *satisfies the preconditions* h_{nc} *of a snap action* h*.*

Claim 1. Take any numeric effect $(f \coloneqq e) \in h_{upd}$*. Evaluating* e *against* M_F *results in a value* f' *that is within* $box(f)$*.*

Claim 2. Evaluating all $(f_i \coloneqq e_i) \in h_{upd}$ *in any order against* $(M'_i)_F$*, where* $(M'_0)_F = M_F$ *and* $(M'_{i+1})_F(g) = (M'_i)_F(g)$ *where* $g \neq f_i$ *and* $(M'_{i+1})_F(f_i) = eval((M'_i)_F, e_i)$ *otherwise maintains that* $(M'_{i+1})_F(f) \in box(f)$ *for any* $f \in F$ *where* $0 \leq i < |h_{upd}|$*.*

Claim 3. The initial numeric state I_F *is also subsumed by the box.*

Proof (Sketch). Take the final iteration of infer-bounds. We know that *env* has reached a pre-fixed point. We know that $env \sqsupseteq env'' \sqsupseteq env'$ because of widening and the loop's condition. guarded-asteps only updates fluents when the conditions of h are satisfied. $env' \sqsupseteq env$, because guarded-asteps is monotone. Hence $env'' = env' = env$.

Also, for any single action whose numeric preconditions are satisfied by *env*, evaluating all numeric effects with respect to *env* and applying them at once, cannot have made the accumulator wider than *env* in guarded-asteps, since the union of intervals in the accumulator is monotonic. In fact, applying any single one of them could not have made the accumulator wider. Thus follows Claim 1.

We also know that these numeric effects do not interfere. Hence evaluating any one e_i against some $(M'_i)_F$ results in $eval((M'_0)_F, e_i) = eval(M_F, e_i)$, which maintains $(M'_{i+1})_F(f) \in box(f)$ for all $f \in F$. Thus follows Claim 2.

Claim 3 follows from monotonicity of the repeat-until loop and the fact that *env* starts with all assignments in $[I_F(f), I_F(f)]$.

Lemma 2. *Given a configuration enc-after*$(\langle L_I, v_I, c_I\rangle, 0, \langle L_0, v_0, c_0\rangle)$*, it is possible to prove the existence of a configuration* $\langle L_k, v_k, c_k\rangle$ *and a run* $\langle\langle L_0, v_0, c_0\rangle, \ldots, \langle L_k, v_k, c_k\rangle\rangle$*, such that run*$([\langle L_0, v_0, c_0\rangle, \ldots, \langle L_k, v_k, c_k\rangle])$ *and enc-after*$(\langle L_I, v_I, c_I\rangle, k, \langle L_k, v_k, c_k\rangle)$ *are satisfied, where* $k \leq m$.

The proof of Lemma 2 proceeds by induction on k. The base case holds since $run([\langle L_0, v_0, c_0\rangle])$ holds. From *enc-after*$(\langle L_I, v_I, c_I\rangle, k, \langle L_k, v_k, c_k\rangle)$ we can deduce *encodes-before*$(\langle L_I, v_I, c_I\rangle, k{+}1, \langle L_k, v_k, c_k \oplus \delta_{k+1}\rangle)$, where $\delta_{k+1} = t_{k+1} - t_k$, $\{t_k, t_{k+1}\} \subseteq htps(\overrightarrow{\pi})$, and there exists no s in $htps(\overrightarrow{\pi})$, such that $t_k < s < t_{k+1}$. This is because no snap action in the induced parallel plan can change the discrete state of the world between t_k and t_{k+1}. From *encodes-before*$(\langle L_I, v_I, c_I\rangle$, $k+1, \langle L_k, v_k, c_k \oplus \delta_{k+1}\rangle)$ all automata $\mathcal{T}_\pi$, where $\langle t_{k+1}, \pi_{start}\rangle \in B_{\overrightarrow{\pi}}$ and/or $\langle t_{k+1}, \pi_{end}\rangle \in B_{\overrightarrow{\pi}}$, can apply the transitions encoding the conditions and effects of the start π_{start} and/or end π_{end} of π at t_{k+1}. Moreover, each action (and hence each automaton) can at most add 1 to the total number of active actions, encoded by $v(aa)$ and one to each of $v(lp_p)$, which means that both must stay beneath N. When an update decrements it, the 1 must have been contributed, so $v(aa) > 0$ and $v(lp_p) > 0$ hold. When an update increments it, the action does not yet contribute 1, which means that $v(aa) < N$ and $v(lp_p) < N$ hold.

From a configuration $\langle L_I, v_I, c_I\rangle$, which encodes the initial state I of the planning problem at t_0, it is possible to reach a configuration $\langle L_k, v_k, c_k\rangle$, which encodes the state of the world after the final happening time point t_k of the plan. To prove this, we can use Lemma 2. To use Lemma 2, we need a $\langle L_0, v_0, c_0\rangle$, which encodes the state of the planning problem after t_0. This $\langle L_0, v_0, c_0\rangle$ can be reached from $\langle L_I, v_I, c_I\rangle$, which we prove in Lemma 3.

Lemma 3. *There exists a finite run* $\langle\langle L_I, v_I, c_I\rangle, \ldots, \langle L_0, v_0, c_0\rangle\rangle$ *from the initial configuration* $\langle L_I, v_I, c_I\rangle$ *to some configuration* $\langle L_0, v_0, c_0\rangle$*, which satisfies enc-after*$(\langle L_I, v_I, c_I\rangle, 0, \langle L_0, v_0, c_0\rangle)$.

This holds, because e_{1M} encodes the updates necessary to initialise the discrete components of the configuration of the network L'_I and v'_I to encode the initial state I of the planning problem. Another $\longrightarrow_{\delta_I}$ transition can initialise c'_I, where $\delta_I > 0$. If there are no entries $\langle t_0, \alpha\rangle$ in the induced parallel plan $B_{\overrightarrow{\pi}}$, then *encodes-before*$(\langle L_I, v_I, c_I\rangle, 0, \langle L'_I, v'_I, c'_I\rangle) \iff$ *enc-after*$(\langle L_I, v_I, c_I\rangle$, $0, \langle L'_I, v'_I, c'_I\rangle)$. If there is at least one entry, then all transitions to check the conditions of any α, where $\langle t_0, \alpha\rangle \in B_{\overrightarrow{\pi}}$, must be applicable in some order. After applying all transitions, some $\langle L_0, v_0, c_0\rangle$ can be reached, where *enc-after*$(\langle L_I, v_I, c_I\rangle, 0, \langle L_0, v_0, c_0\rangle)$. Again $v(aa)$ and $v(lp_p)$ respect the bounds, using a similar argument to Lemma 2.

A configuration $\langle L_G, v_G, c_G\rangle$ satisfying the goal condition can be reached from $\langle L_k, v_k, c_k\rangle$ in one transition, if $\langle L_k, v_k, c_k\rangle$ encodes the state of the planning problem after the final happening time point $t_k \in \mathit{htps}(\vec{\pi})$.

Lemma 4. *We assume that a configuration $\langle L_k, v_k, c_k\rangle$ satisfies enc-after$(\langle L_I, v_I, c_I\rangle, k, \langle L_k, v_k, c_k\rangle)$, where t_k is the final happening time point in htps$(\vec{\pi})$. We also assume that L_k records plan$_M$ for $\mathcal{T}_M$. Then there exists a valid run $\langle\langle L_k, v_k, c_k\rangle, \ldots, \langle L_G, v_G, c_G\rangle\rangle$ ending in a configuration in which $L_G \equiv \langle \mathit{goal}_M, \ldots\rangle$.*

It is sufficient to prove that e_{2M} can be applied to $\langle L_k, v_k, c_k\rangle$, because applying e_{2M} to any $\langle L, v, c\rangle$ results in a $\langle L', v', c'\rangle$, where $L' \equiv \langle \mathit{goal}_M, \ldots\rangle$. The conditions of e_{2M} are satisfied, because *enc-after*$(\langle L_I, v_I, c_I\rangle, k, \langle L_k, v_k, c_k\rangle)$ is a stronger assumption.

C.1 Proof of Lemma 2

Before continuing on to the proof of Lemma 2, we refer to Example 5 and Figures 4, 6 and 10, to informally illustrate our argument for the order of applied transitions.

Proof. The proof is by induction on k. ***Base case:*** $k = 0$.

Since a single configuration is a run, we obtain the fact: $\mathit{run}([\langle L_0, v_0, c_0\rangle])$. From the assumption *enc-after*$(\langle L_I, v_I, c_I\rangle, 0, \langle L_0, v_0, c_0\rangle)$ and Definition 22 we trivially deduce *enc-after*$(\langle L_I, v_I, c_I\rangle, 0, \langle L_0, v_0, c_0\rangle)$, which completes this case. ***Inductive case:*** *$k = i + 1$ for an arbitrary i.*

We can assume the existence of some i and $\langle L_i, v_i, c_i\rangle$, which satisfy *enc-after*$(\langle L_I, v_I, c_I\rangle, i, \langle L_i, v_i, c_i\rangle)$ and $\mathit{run}([\langle L_I, v_I, c_I\rangle, \ldots, \langle L_i, v_i, c_i\rangle])$ from the inductive hypothesis.

We can prove that $\langle L'_i, v'_i, c'_i\rangle$ exists, where *encodes-before*$(\langle L_I, v_I, c_I\rangle, i + 1, \langle L'_i, v'_i, c'_i\rangle)$ and $\langle L_i, v_i, c_i\rangle \longrightarrow_{\delta_{i+1}} \langle L'_i, v'_i, c'_i\rangle$ where $\delta_n = t_n - t_{n-1}$. This is because t_{i+1} is the immediate successor time point of t_i according to Definition 14, and, therefore, only the durations recorded by c_i must be incremented to satisfy *encodes-before*. The properties encoded by v and L refer to propositions and states of actions, which can only change at happening time points, of which there are none between t_i and t_{i+1}.

Hence, we know of the existence of some $\langle L'_i, v'_i, c'_i\rangle$ that satisfies *encodes-before*$(\langle L_I, v_I, c_I\rangle, i + 1, \langle L'_i, v'_i, c'_i\rangle)$. From Definition 16, we can deduce that this permits the network to simulate the execution of all snap actions at t_{i+1} in at least one order.

An action can either be scheduled to *start* at t_{i+1}, *end*, or both. Due to the no self-overlap condition (Definition 16), if an action is scheduled to start and end, it must be scheduled with a duration of 0, i.e. be *instantaneous*.

All automata $\mathcal{T}_\pi$ for all *ending* actions π must transition from $\mathit{running}_\pi$ to ending_π. This can decrement some lp_p for $p \in \pi_{inv}$ to 0. This then permits ${ee'}_{\pi_2}$ and se_{π_2} transitions to be taken without violating some $(\mathit{lp}_p = 0)$? conditions, if π_{2start} and π_{2end} are scheduled at t_{i+1} in $B_{\vec{\pi}}$.

Automata for *instantaneous* actions π do not take their ${se'}_\pi$ and ee_π transitions at all, skipping them by taking their ie_π transition. ${se'}_\pi$'s guard, which checks $v(vp_p) = 1$ for $p \in \pi_{inv}$, is not guaranteed to be satisfied, because π_{inv} conditions are not active in the instants an action starts or ends.

All *instantaneous* actions π must take se_π, ie_π, and ${ee'}_\pi$ transitions.

Then all *starting* snap actions' automata must take their se_π and all *ending* snap actions' automata must take their ${ee'}_\pi$ transitions. This applies their effects.

Finally, any automaton $\mathcal{T}_\pi$ for a *starting* snap action π_{start} can take its ${se'}_\pi$ transition to lock the invariants π_{inv} by incrementing lp_p for all $p \in \pi_{inv}$.

Transitions in this order are possible and lead to a configuration satisfying *enc-after*$(\langle L_I, v_I, c_I\rangle, i+1, \langle L_{i+1}, v_{i+1}, c_{i+1}\rangle)$ because: (i) no mutex constraints (see Definition 13) in transitions' guards are violated, due to non-interference of snap actions in a valid plan (see Definition 15); (ii) the precondition on vp_p where $p \in \alpha_{1pre}$ of every snap action α_1 scheduled at t_i is satisfied by every intermediate v' in this part of the run, because any precondition α_{1pre} of a scheduled snap action α_1 is already satisfied by M_i (see Definition 15) and mutex constraints ensure that neither α_{2add} nor α_{2del} include any p in α_{1pre} if α_2 and α_1 are scheduled at t_i; (iii) no $x_{\pi_{start}}$ or $x_{\pi_{end}}$ clock that is reset by any transition interferes with any other transition's guard, because no mutually exclusive snap actions can be scheduled at t_i (see Definitions 15 and 16); (iv) duration constraints are satisfied (*dur-c-sat*$(\pi, c(x_{\pi_{start}}))$) for all *ending* snap actions; and (v) every update stays within the bounds, either by construction or because each active action π (and automaton $\mathcal{T}_\pi$) can only contribute 1 to the value $v(z)$ bounded by $[0, N]$—when the transition increments, the π (and $\mathcal{T}_\pi$) cannot have been contributing to that value and when the transition decrements, they must have been. This is illustrated in Figures 4 and 10.

This completes this part of the run and lets us obtain $\langle L_{i+1}, v_{i+1}, c_{i+1}\rangle$, where *enc-after*$(\langle L_I, v_I, c_I\rangle, i+1, \langle L_{i+1}, v_{i+1}, c_{i+1}\rangle)$ and *run*$([\langle L'_i, v'_i, c'_i\rangle, \ldots, \langle L_{i+1}, v_{i+1}, c_{i+1}\rangle])$. Furthermore, with $\langle L_i, v_i, c_i\rangle \longrightarrow_{\delta_{i+1}} \langle L'_i, v'_i, c'_i\rangle$ we deduce *run*$([\langle L_i, v_i, c_i\rangle, \ldots, \langle L_{i+1}, v_{i+1}, c_{i+1}\rangle])$. From the induction hypothesis we obtain the fact *run*$([\langle L_0, v_0, c_0\rangle, \ldots, \langle L_i, v_i, c_i\rangle])$. By combining these two runs we obtain the fact *run*$([\langle L_i, v_i, c_i\rangle, \ldots, \langle L_{i+1}, v_{i+1}, c_{i+1}\rangle])$, which completes the proof with *enc-after*$(\langle L_I, v_I, c_I\rangle, i+1, \langle L_{i+1}, v_{i+1}, c_{i+1}\rangle)$.

C.2 Proof of Lemma 3

Proof (by cases on the size of htps$(\vec{\pi})$*).*
The transition $\langle L_I, v_I, c_I\rangle \longrightarrow_{e_{1M}} \langle L'_I, v'_I, c_I\rangle$ is possible for some $\langle L'_I, v'_I, c_I\rangle$. L'_I and v'_I satisfy the conditions in *encodes-before*$(\langle L_I, v_I, c_I\rangle, 0, \langle L'_I, v'_I, c_I\rangle)$ in Definition 26, because no action is active before t_0 and all $v(vp_p)$ for $p \in P$ encode the propositional state I.

Since no snap actions have been scheduled before t_0, for all $c_I(x_\alpha)$ to satisfy *ran-for-at*$(\alpha, t_0, <)$, the values need to be incremented by $\delta_0 = \kappa + \epsilon$ (Definition 22). This is achievable using a $\langle L'_I, v'_I, c_I\rangle \longrightarrow_{\delta_0} \langle L'_I, v'_I, c'_I\rangle$ transition.

Therefore, there exists a configuration $\langle L', v', c'\rangle$ such that *run*$([\langle L_I, v_I, c_I\rangle, \ldots, \langle L'_I, v'_I, c'_I\rangle])$ and *encodes-before*$(\langle L_I, v_I, c_I\rangle, 0, \langle L'_I, v'_I, c'_I\rangle)$.

We proceed with the proof by splitting the cases.
Case 1*. $htps(\vec{\pi})$ is empty.* In this case, *encodes-before*$(\langle L_I, v_I, c_I\rangle, 0, \langle L'_I, v'_I, c'_I\rangle)$ is equivalent to *enc-after*$(\langle L_I, v_I, c_I\rangle, 0, \langle L'_I, v'_I, c'_I\rangle)$, due to the definition of both in case $htps(\vec{\pi})$ is empty, which concludes the proof. See Definitions 22 and 26.
Case 2*. $htps(\vec{\pi})$ is not empty.* In this case, *encodes-before*$(\langle L_I, v_I, c_I\rangle, 0, \langle L'_I, v'_I, c'_I\rangle)$ permits the construction of a run $\langle\langle L'_I, v'_I, c'_I\rangle, \ldots, \langle L_0, v_0, c_0\rangle\rangle$ such that *enc-after*$(\langle L_I, v_I, c_I\rangle, 0, \langle L_0, v_0, c_0\rangle)$ and $run([\langle L'_I, v'_I, c'_I\rangle, \ldots, \langle L_0, v_0, c_0\rangle])$ are satisfied using reasoning like in Lemma 2. This run can be combined with $\langle\langle L_I, v_I, c_I\rangle, \ldots, \langle L'_I, v'_I, c'_I\rangle\rangle$ to construct $\langle\langle L_I, v_I, c_I\rangle, \ldots, \langle L_0, v_0, c_0\rangle\rangle$.

C.3 Proof of Lemma 4

Proof. $\langle L_k, v_k, c_k\rangle \longrightarrow_{e_{2M}} \langle L_G, v_G, c_G\rangle$ is possible if *enc-after*$(\langle L_I, v_I, c_I\rangle, k, \langle L_k, v_k, c_k\rangle)$ holds.

Every starting snap action π_{start} scheduled before or at t_k can be paired with an ending snap action π_{end} before or at t_k.

Therefore, the number of se and ee' edges traversed is equal. Hence, $v_k(aa) = 0$ and $v_k(lp_p) = 0$ for all p.

Moreover, $v_k(vp_p)$ equals 1 for all p in the final propositional state $(M_{k+1})_P$ and $v_k(f)$ equals $(M_{k+1})_F(f)$ for all $f \in F$, where M_{k+1} is the final state of the valid state sequence (see Definitions 15 and 22). Since $G_P \subseteq (M_{k+1})_P$ and $(M_{k+1})_F \vDash_b G_F$ (see Definition 16), we can deduce $v_k(vp_p) = 1$ for all $p \in G_P$ and that v_k satisfies G_F. Hence, the conditions of e_{2M} are satisfied.

C.4 Proof of Theorem 1

Proof. It is sufficient to prove the existence of a run *cfgs* and configuration $\langle L_G, v_G, c_G\rangle$ of the automata network, such that: (i) *cfgs* is a valid run: $run(cfgs)$; (ii) *cfgs* contains $\langle L_G, v_G, c_G\rangle$: $cfgs \equiv \langle\langle L_I, v_I, c_I\rangle, \ldots, \langle L_G, v_G, c_G\rangle, \ldots\rangle$; and (iii) L_G records the main automaton's location as $goal_M$: i.e. $L_G \equiv \langle goal_M, \ldots\rangle$.

The proof is completed by fixing $k = m$ in Lemma 2 and combining the run obtained from this fact with those from Lemmas 3 and 4.

D Details of Formalisation

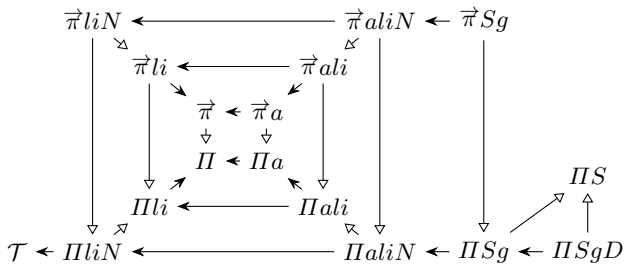


Fig. 11: Locale extensions ($\uparrow$) add and sublocales ($\twoheadrightarrow$) replace definitions and assumptions: timed automata ($\mathcal{T}$), planning problem (Π), plan ($\overrightarrow{\pi}$), abstract syntax (S), unique naming functions (N), executable definitions (D), groundedness (g), annotated actions as snap actions (a), lists as sets (l), machine integers as clock constraints (i).

Locales The first aspect of formally verifying our encoding is the approach to formalising the definitions involved in Isabelle/HOL's logic. Since the definitions are numerous and intricately connected, a naive approach could lead to unneeded verification effort and a badly structured model of the encoding. We thus use Isabelle/HOL locales [9] to structure our library of definitions. A locale can be seen as a formal mathematical context that introduces constants and assumptions, in which one develops definitions and theorem statements that implicitly recognise these constants and assumptions. The most fundamental locale (see Listing 1.4) is that defining constants and assumptions on a planning problem (Π). Locales can help structure a library of definitions by representing two main types of relationships between mathematical concepts. First is *extension*, where one concept can be seen as another concept, but with more variables and assumptions added. E.g. a valid plan ($\overrightarrow{\pi}$) (see Listing 1.5) presumes a valid planning problem, extended with a sequence of actions and assumptions on those actions, such that they all come from the planning problem. The other type of relationship is that of a *sublocale*: a mathematical concept satisfies the assumptions of another, under a suitable mapping of one concept's variables to another's. E.g. Π is a sublocale of a network of timed automata ($\mathcal{T}$), after encoding, i.e. our encoding always gives a valid automaton. Figure 11 shows the relationships between the main concepts formalised as locales in our project.

As shown in Figure 11, the core concept of a planning problem Π, which uses abstract mathematical concepts like sets and dense linear orders, is where much of the abstract theory about temporal planning is proved. It is refined into

more concrete variants; e.g. the Πa locale replaces snap actions in Π with annotated actions, to make it easier to identify actions, and hence transitions that need to be taken to simulate a snap action's execution, in the proof. The Πali locale fixes types to represent sets as lists, to make it easier to order transitions, and clock constraints as integers (as opposed to dense linear orders) for simpler proofs and executability. Within the $\Pi aliN$ locale, we provide an executable definition of the encoding in Section 3. Another refinement step is to deal with parsed problems: ΠSg is defined over abstract syntax trees of grounded PDDL. ΠSgD refines ΠSg, and is the most refined variant of a planning problem. It has executable datatypes such as native strings, red-black trees, and arbitrary-length integers, and it has the most efficient encoding that we run.

We reuse from the work of Abdulaziz and Koller [2] the abstract syntax (and a semantics) of PDDL in ΠS, a minimal trusted Standard ML parser for that abstract syntax, and executable code to check well-formedness. We also use the Collections framework by Lammich and Lochbihler [66] to refine to efficient data structures. Moreover, our code that interfaces with the formalisation of timed automata by Wimmer and von Mutius [90] uses the Imperative Refinement Framework by Lammich [65] for efficient imperative implementations, since Wimmer and von Mutius's code [90] is developed within that framework.

Listing 1.4: Locale for a temporal planning problem

Important locales

```
locale temp-planning-problem = action-defs at-start at-end
     over-all lower upper pre adds dels
  for at-start:: 'action  ⇒ 'snap-action
  and at-end::  'action ⇒ 'snap-action
  and over-all:: 'action  ⇒ 'proposition set
  and lower:: 'action  ⇀ ('time::time) lower-bound
  and upper:: 'action  ⇀ 'time upper-bound
  and pre:: 'snap-action ⇒ 'proposition set
  and adds:: 'snap-action ⇒ 'proposition set
  and dels:: 'snap-action ⇒ 'proposition set  +
fixes init:: 'proposition set
  and goal:: 'proposition set
  and ϵ:: 'time::time
assumes eps-ran:  0 ≤ ϵ
```

Listing 1.5: Locale for a valid temporal plan

```
type-synonym ('i, 'a, 't) temp-plan = 'i ⇀ ('a × 't × 't)
locale temp-plan-defs = temp-planning-problem  at-start
    at-end over-all lower upper pre adds dels init goal ϵ
  for at-start:: 'action  ⇒ 'snap-action
  and at-end::  'action ⇒ 'snap-action
  and over-all:: 'action  ⇒ 'proposition set
  and lower::
```

```
      'action  ⇀ ('time::time) lower-bound
    and upper::
      'action  ⇀ 'time upper-bound
    and pre::
      'snap-action ⇒ 'proposition set
    and adds::
      'snap-action ⇒ 'proposition set
    and dels::
      'snap-action ⇒ 'proposition set
    and init:: 'proposition set
    and goal:: 'proposition set
  and ϵ:: 'time  +
fixes π:: ('i, 'action, 'time) temp-plan

locale valid-temp-plan = temp-plan-defs +
  assumes vp: valid-plan
```

Mathematical Induction To obtain a goal of the form $esc(i, j, q_{i;j}) \rightarrow esc(i, j + 1, q_{i;j+1})$, we reason about lists. Instead of structural induction over the constructors of lists, we induct over indices of elements, i.e. natural numbers.

For the formalisation we separate the preconditions of these transitions from their effects. We define lists of effects as lists of functions.

Listing 1.6 defines two functions. `seq_apply` applies a list of functions beginning at an initial element. `ext_seq` takes a function that creates a list from an initial element and a list. It applies the function to the final element of the list and appends the result. `((ext_seq ∘ seq_apply) fs)` can be composed with itself using arbitrary `fs` arguments to build longer lists.

Listing 1.6: Lists and Functions

```
definition ext-seq f xs ≡
    xs @ f ( last xs)
definition seq-apply fs x ≡
    map ( λi. ( fold (id) ( take i fs) x))
        [1..<length fs + 1]
```

We also use standard functions, such as `fold`, `take` and `nth`. `take` returns the first n elements of a list. `nth`, denoted as `!`, obtains the n-th element by index.

Consider a list of functions $fs = \langle f_0, \ldots, f_{n-1} \rangle$. Consider the list $xs = \langle x_1, \ldots, x_n \rangle$ of states obtained through: `seq_apply` fs x_0. It is apparent x_i `= ( last (seq-apply (take` i fs`)` x_0`))` for $i < n$ and x_{i+1} `=last (seq-apply (take` $(i + 1)$ fs`)` x_0`)`.

The $(n + 1)$-th element is always obtained by applying the n-th function to the n-th element. We define the induction-like inference rule in Listing 1.7, which allows us to prove properties of individual states and the beginning and ends of lists obtained from `seq_apply`. R and S are properties independent of the index. When `x` is replaced with `(last ys')` for some `(ys' = (ext_seq ∘ seq_apply fs) ys)`, then we need to know some `R (last ys')`, which can be proven with the

same lemma. `S` must be instantiated with `R` and `R` must be instantiated with some `R'. R (last (y # seq_apply fs ys))` for some `y` is proven under the assumption of `(R' y)`. Thus, we compose proofs.

Listing 1.7: Final state lemma

```
lemma seq-apply-pre-post:
  assumes PQ: ...
  and QP: ...
  and Rx: R x
  and len0: ∀x. 0 =length fs
    ⟶ ( R x ⟶ S x)
  and RP0: ∀x. 0 < length fs
    ⟶ ( R x ⟶ P 0 x)
  and QSl: ∀x. 0 < length fs
    ⟶ ( Q ( length fs - 1) x ⟶ S x)
    shows
    S ( last ( x # seq-apply fs x))
```

For the induction-like proof of Lemma 2, we need to assert that the entire list is a run. This is proven using Listing 1.8, by instantiating `LP` with the *run* predicate. When a state satisfies `(P i)` and its successor satisfies `(Q i)`, then they form a run consisting of one step. A run ending with a configuration `(last xs)` can be prepended to a run beginning at `(last xs)`.

Listing 1.8: List predicate lemma

```
lemma ext-seq-comp-seq-apply-list-prop:
  assumes PQ: ...
  and QP: ...
  and R (last xs)
  and RP0: ...
  and LPi: ∀i s. i < length fs
    ⟶ P i s ⟶ Q i (( fs ! i) s)
    ⟶ LP [s, ( fs ! i) s]
  and base: ∀x. LP [x]
  and step: ∀xs ys. LP xs
    ⟶ LP ( last xs # ys)
    ⟶ LP ( xs @ ys)
  and LPxs: LP xs
  shows LP (( ext-seq ∘ seq-apply) fs xs)
```

With rules analogous to Listings 1.7 and 1.8, we prove Theorem 1 and Lemmas 2 to 4.

Table 2: Size of formalisations (percentages rounded to sum to 100 %)

Size of formalisation				
Component	Lines of code (thousands) by attribution			
	Human	Human + AI	AI	Total
Propositional				
Abstract temporal planning semantics	6.9 (98 %)	<0.1 (<1 %)	0.1 (2 %)	7.0
Reduction in Section 3	1.1 (97 %)	<0.1 (3 %)	<0.1 (<1 %)	1.1
Technical setup for proofs	4.1 (95 %)	0.2 (5 %)	<0.1 (<1 %)	4.3
Proofs of Theorem 1 including lemmas, conditions and rules	3.7 (80 %)	0.9 (19 %)	<0.1 (1 %)	4.6
Reduction from PDDL AST to abstract planning	2.8 (47 %)	0.7 (12 %)	2.4 (41 %)	6.0
Refinement to executable code	1.0 (90 %)	<0.1 (4 %)	0.1 (6 %)	1.1
Verified certificate check and code export	1.2 (95 %)	<0.1 (3 %)	<0.1 (2 %)	1.3
Miscellaneous lemmas and definitions	1.8 (94 %)	—	0.1 (6 %)	1.9
Total (propositional)	22.6 (83 %)	2.0 (7 %)	2.8 (10 %)	27.4
Numeric extension				
Abstract semantics	—	—	0.8 (100 %)	0.8
Reduction in Section 3	—	—	0.5 (100 %)	0.5
Technical setup for proofs	—	—	2.7 (100 %)	2.7
Proofs of Theorem 1	—	—	5.8 (100 %)	5.8
Reduction from PDDL AST	—	<0.1 (1 %)	0.4 (99 %)	0.4
Refinement to executable code	—	—	1.2 (100 %)	1.2
Numeric bounds: inference	—	<0.1 (2 %)	1.8 (98 %)	1.9
Numeric bounds: carrying	—	—	1.9 (100 %)	1.9
Certificate check and code export	—	<0.1 (6 %)	0.6 (94 %)	0.6
Miscellaneous lemmas and definitions	—	—	<0.1 (100 %)	<0.1
Total (numeric extension)	—	0.1 (1 %)	15.7 (99 %)	15.8
Total (ours)	22.6 (52 %)	2.0 (5 %)	18.5 (43 %)	**43.2**
Formal PDDL semantics (with others)				
Basics: syntax, happening semantics	5.7 (100 %)	—	—	5.7
State-sequence semantics	—	1.8 (100 %)	—	1.8
Total	5.7 (76 %)	1.8 (24 %)	—	7.5
Grounding (with others)				
Total	1.6 (59 %)	0.8 (28 %)	0.4 (13 %)	2.7
Third-party				
Interval domain from HOL-IMP	—	—	—	0.4
Two lemmas copied from Munta	—	—	—	<0.1
Total	—	—	—	0.4
Total	30.3 (56 %)	4.6 (9 %)	18.8 (35 %)	**53.9**

Size of related formalisations	
Formalisation	Lines of code
Verified SAT-based planner [3]	$\sim$ 17500
Formal semantics and a verified plan validator for temporal PDDL [2]	$\sim$ 6500
Formal semantics and a verified plan validator for classical PDDL [4]	$\sim$ 3000
Formal semantics of Markov Decision Processes (MDPs), a verified solver for MDPs and certificate checker for linear programming solutions [81]	$\sim$ 35000
Verified MDP algorithms [80]	$\sim$ 10000
Formalised MDP theory [79]	$\sim$ 5000

E Threshold Inference

```
type IntOrInf = ℤ | ∞ | −∞
type Interval = {lo : IntOrInf, hi : IntOrInf}
type Env = Map⟨𝒱, Interval⟩
```

```
function add-int-or-inf(a : IntOrInf, b : IntOrInf)                ▷ overloads (+)
    if a = ∞ OR b = ∞ then return ∞
    else if a = −∞ OR b = −∞ then return −∞
    else
        return a + b
```

```
function add-intv(e₁ : Interval, e₂ : Interval)                    ▷ overloads (+)
    return [e₁.lo + e₂.lo, e₁.hi + e₂.hi]
```

```
function negate-intv(e₁ : Interval)                                ▷ overloads (unary −)
    return [−e₁.hi, −e₁.lo]
```

```
function sub-intv(e₁ : Interval, e₂ : Interval)                    ▷ overloads (−)
    return add-intv(e₁, −e₂)
```

Listing 1: Data structures, types, and operations.

```
function approx-eval-intv(env : Env, e : Expr)
    if e = Const(c) then return [c, c]
    else if e = Var(v) then return env[v]
    else if e = Add(e₁, e₂) then return approx-eval-intv(env, e₁) +
    approx-eval-intv(env, e₂)
    else if e = Sub(e₁, e₂) then return approx-eval-intv(env, e₁) −
    approx-eval-intv(env, e₂)
    else if e = Mul(e₁, e₂) then return [−∞, +∞]
    else if e = Div(e₁, e₂) then return [−∞, +∞]
```

Listing 2: Approximate interval evaluation.

```
type CompOp = {<, ≤, =, ≥, >}
type Expr = Const(c : ℤ) | Var(v : 𝒱)
      | Add(e₁ : Expr, e₂ : Expr) | Sub(e₁ : Expr, e₂ : Expr)
      | Mul(e₁ : Expr, e₂ : Expr) | Div(e₁ : Expr, e₂ : Expr)
type BExp = Comp(e₁ : Expr, op : CompOp, e₂ : Expr)
      | And(b₁ : BExp, b₂ : BExp) | True
type Effect = {l : 𝒱, r : Expr}
type snap-action = {nc : BExp, upd : Set⟨Effect⟩}
```

```
function increments(f : 𝒱, exp : Expr)
    if exp = Add(f, exp′) then return {exp′}
    else if exp = Sub(f, exp′) then return {Sub(Const(0), exp′)}
    else if exp = Add(exp′, f) then return {exp′}
    else
        return ∅

function comps(b : BExp)
    if b = True then return ∅
    else if b = And(b₁, b₂) then return comps(b₁) ∪ comps(b₂)
    else if b = Comp(e₁, op, e₂) then return {(e₁, op, e₂)}
```

Listing 3: Expressions and guards.

```
function shifted-consts(env : Map⟨𝒱, ℤ⟩, e : Expr, offset : ℤ)
    ks ← ∅
    for all k ∈ consts(e) ∪ {eval(env, e)} do
        ks ← ks ∪ {k + offset}
    return ks

function snap-fluent-thrs(env : Map⟨𝒱, ℤ⟩, f : 𝒱, a : snap-action)
    thrs ← ∅
    for all eff ∈ a_upd do
        for all increment ∈ increments(f, eff.r) do
            offset ← eval(env, increment)
            for all (e₁, op, e₂) ∈ comps(a_nc) do
                if f = e₁ then
                    thrs ← thrs ∪ shifted-consts(env, e₂, offset)
                else if f = e₂ then
                    thrs ← thrs ∪ shifted-consts(env, e₁, offset)
    return thrs

function snap-thrs(env : Map⟨𝒱, ℤ⟩, fluents : Set⟨𝒱⟩, a : snap-action)
    cs ← consts(a_nc) ∪ consts(a_upd)
    thrs ← ∅
    for all f ∈ fluents do
        thrs ← thrs ∪ snap-fluent-thrs(env, f, a)
    return thrs ∪ cs
```

```
function init-thresholds(fluents : Set⟨V⟩, inits : Map⟨V, ℤ⟩, S : Set⟨snap-action⟩)
    snaps-thrs ← ∅
    for all a ∈ S do
        snaps-thrs ← snaps-thrs ∪ snap-thrs(inits, fluents, a)
    init-thrs ← ∅
    for all f ∈ fluents do
        init-thrs ← init-thrs ∪ {inits[f]}
    return snaps-thrs ∪ init-thrs
```

Listing 4: Threshold initialisation.

```
function widen(thrs : List⟨ℤ⟩, cur : Interval, nxt : Interval)
    nl ← cur.lo
    nu ← cur.hi
    thrs ← sort(concat(thrs, [−∞, ∞]))
    if nxt.lo < cur.lo then
        nl ← −∞
        for i ← 0; thrs[i] ≤ nxt.lo AND i < length(thrs); i++ do
            nl ← thrs[i]
    if nxt.hi > cur.hi then
        nu ← ∞
        for i ← length(thrs) − 1; thrs[i] ≥ nxt.hi AND i ≥ 0; i−− do
            nu ← thrs[i]
    return [nl, nu]
```

```
function widen-all(fluents : Set⟨V⟩, thrs : List⟨ℤ⟩, env : Env, env′ : Env)
    acc ← env
    for all f ∈ fluents do
        acc[f] ← widen(thrs, env[f], env′[f])
    return acc
```

Listing 5: Widening.

```
function astep-updates(effects : Set⟨Effect⟩, env : Env)
    env′ ← env
    for all eff ∈ effects do
        env′[eff.l] ← approx-eval-intv(env, eff.r)
    return env′
```

```
function unguarded-astep(S : Set⟨snap-action⟩, env : Env)
    env′ ← env
    for all a ∈ S do
        env′ ← env′ ⊔ astep-updates(a_upd, env)
    return env′
```

Listing 6: Updates for snap actions (astep updates) and unguarded updates for all snap actions.

```
function comp-intv(op : CompOp, intv : Interval)
    if intv.hi < intv.lo then return intv
    else if op = ≤ then return [−∞, intv.hi]
    else if op = < then return [−∞, intv.hi − 1]
    else if op = ≥ then return [intv.lo, ∞]
    else if op = > then return [intv.lo + 1, ∞]
    return intv
```

```
function flip-op(op : CompOp)
    if op = ≤ then return ≥
    else if op = < then return >
    else if op = ≥ then return ≤
    else if op = > then return <
    else
        return =
```

```
function refine-comp((e1, op, e2) : (Expr, CompOp, Expr), env : Env)
    e2′ ← approx-eval-intv(env, e2)
    e1′ ← approx-eval-intv(env, e1)
    if e1 = Var(f) then
        env[f] ← env[f] ⊓ comp-intv(op, e2′)
    if e2 = Var(f) then
        env[f] ← env[f] ⊓ comp-intv(flip-op(op), e1′)
    return env
```

```
function refine-guards(env : Env, G : Set⟨BExp⟩)
    env′ ← env
    for all g ∈ G do
        for all c ∈ comps(g) do
            env′ ← refine-comp(c, env′)
    return env′
```

Listing 7: Refining guards.

```
function interval-inter-bexp((e_1, op, e_2) : (Expr, CompOp, Expr), env : Env)
    e'_1 ← approx-eval-intv(env, e_1)
    e'_2 ← approx-eval-intv(env, e_2)
    i_1 ← e'_1 ⊓ comp-intv(op, e'_2)
    i_2 ← e'_2 ⊓ comp-intv(flip-op(op), e'_1)
    if i_1.hi < i_1.lo then return false
    else if i_2.hi < i_2.lo then return false
    return true
```

```
function guarded-astep-updates(a : snap-action, env : Env)
    env' ← env
    for all c ∈ comps(a_nc) do
        if NOT interval-inter-bexp(c, env) then return env
        env' ← refine-comp(c, env')
    return astep-updates(a_upd, env')
```

```
function guarded-asteps(env : Env, S : Set⟨snap-action⟩)
    acc ← env
    for all a ∈ S do
        acc ← acc ⊔ guarded-astep-updates(a, env)
    return acc
```

Listing 8: Guarded steps.

```
function infer-bounds(fluents : Set⟨V⟩, inits : Map⟨V, ℤ⟩, S : Set⟨snap-action⟩)
    env ← λv. [0, −1]                                   ▷ empty interval
    for all f ∈ fluents do
        env[f] ← [inits[f], inits[f]]
    thrs ← init-thresholds(fluents, inits, S)
    env'' ← env
    repeat
        env ← env''
        env' ← guarded-asteps(env, S)
        env'' ← widen-all(fluents, thrs, env, env')
    until env'' ⊑ env
    return env
```

Listing 9: Main loop.

Proposition 1. *We argue that termination of* infer-bounds *(Listing 9) is guaranteed, because in each iteration which does not satisfy* $env'' \sqsubseteq env$*, the finite set of thresholds for a variable* $v \in F$ *not within* v*'s bounds must shrink or one of a finite number of bound variables is irreversibly set to* $+\infty$ *or* $-\infty$*. When the final environment contains an infinite bound, we reject the planning instance with no verdict.*

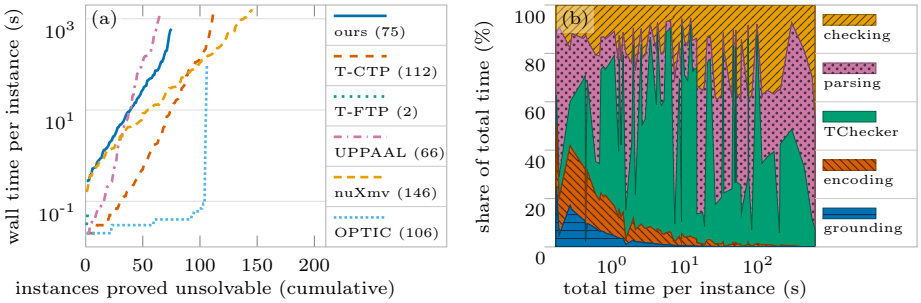


Fig. 12: (a) Cumulative number of instances (of 397) proved unsolvable within a given wall time (timeout 1800 s); total in legend. (b) Per-stage times of our pipeline for the 75 certified-unsolvable instances, normalised per instance.

F Additional Data

Table 1 and Figure 12a show how the different planners compare to one another. The data shows that our pipeline performs on par with the other unverified planners (3 vs. 2, 3) on MatchCellar, is worse than TAMER-CTP, nuXmv, and UPPAAL on SYNC (3 vs. 5, 6, 4) and PAINTER (1 vs. 5, 4, 4); OPTIC also outperforms ours on PAINTER with 2 solved instances, but does not solve any SYNC instance. Our pipeline does not solve any instance of MAJSP-1, while TAMER-CTP and nuXmv solve 12 and 22 respectively; OPTIC and UPPAAL solve 1 each. In MAJSP-2, our pipeline outperforms UPPAAL (68 vs. 54), while being outperformed by TAMER-CTP and nuXmv (88 and 112 respectively); OPTIC detects 100 (out of 240) as unsolvable while preprocessing and grounding the problem, while timing out on 103 and running out of memory on 37 of the remaining instances.

The dominant failure mode of our pipeline in MatchCellar, SYNC, and PAINTER is time, while failures in MAJSP-1 and MAJSP-2 are roughly evenly split between time and memory. See Figure 12b. We point out another thing: in our pipeline, the execution time and memory needed are wholly dominated by TChecker, certificate parsing and verified certificate checking, i.e. our verified encoding usually needs negligible resources compared to the other stages.

In 11 MAJSP instances, our pipeline failed because TChecker was unable to infer clock bounds. UPPAAL segfaulted in one MAJSP-2 instance.

TChecker can prove one more SYNC instance unsolvable when using an LU-abstracted zone subsumption algorithm [57] to ensure termination (see Table 3). This brings the performance of our approach on SYNC on par with UPPAAL and closer to TAMER and nuXmv. However, certificates in the form of LU-abstracted zone graphs are not currently supported by our certificate checker.

	tck-reach (s)			end-to-end (s)			certificate (MB)		
Instance	covreach	aLU	×	covreach	aLU	×	covreach	aLU	×
MatchCellar 02	0.016	0.015	1.1	0.29	0.27	1.1	<1	<1	–
MatchCellar 03	0.031	0.025	1.2	0.55	0.43	1.3	<1	<1	–
MatchCellar 04	140	14.4	9.7	427	92	4.6	835	242	3.5
SYNC 1_2	0.016	0.017	0.9	0.28	0.27	1.0	<1	<1	–
SYNC 1_3	0.181	0.091	2.0	1.66	0.94	1.8	2	1	2.0
SYNC 1_4	7.47	2.26	3.3	34.1	10.5	3.3	49	17	2.9
SYNC 1_5	timeout	242	–	timeout	359	–	–	240	–

Table 3: The table shows the times for TChecker's reachability check and the end-to-end pipeline, and the size of the emitted certificates. "covreach" here refers to TChecker not exploring states covered in the zone graph twice and aLU refers to detecting coverage using LU-abstracted zone graphs. Note that the end-to-end figure stops when the certificate is rejected for the aLU algorithm, which might overrepresent the time savings, but not by much, given the distribution of time taken for each stage (see Figure 12b) and the clearly smaller state space to check.