Formal Semantics and Formally Verified Validation for Temporal Planning

Mohammad Abdulaziz and Lukas Koller
Techniche Universität München, Germany

Abstract

We present a simple and concise semantics for temporal planning. Our semantics are developed and formalised in the logic of the interactive theorem prover Isabelle/HOL. We derive from those semantics a validation algorithm for temporal planning and show, using a formal proof in Isabelle/HOL, that this validation algorithm implements our semantics. We experimentally evaluate our verified validation algorithm and show that it is practical.

Introduction

Although, performance-wise, planning algorithms and systems are very scalable and efficient, as shown by different planning competitions (Long 2000; Coles et al. 2012; Vallati et al. 2015), there is still to be desired when it comes to their trustworthiness, which is crucial to their wide adoption. Consequently, there have been substantial efforts to improve the trustworthiness of planning systems (Howey, Long, and Fox 2004; Fox, Howey, and Long 2005; Eriksson, Röger, and Helmert 2017; Abdulaziz, Norrish, and Gretton 2018; Abdulaziz and Lammich 2018; Cimatti, Micheli, and Roveri 2017; Abdulaziz, Gretton, and Norrish 2019). A basic task when it comes to the trustworthiness of planning systems is that of *plan validation*. In its most basic form, this task is solved by a plan validator, which is a program that, given a planning problem and a candidate plan, confirms whether the candidate plan indeed solves the problem. This boosts the trustworthiness of a plan chiefly because the plan validator should be a simple piece of software that can be more easily inspected than the planning system that computed the plan and, accordingly, less likely to have mistakes.

One challenge to plan validation is that the semantics of planning languages and formalisms can be too complicated. This makes the validator a rather complicated piece of software defeating the trustworthiness appeal of the whole approach. This is especially the case for advanced planning formalisms, like temporal planning (Fox and Long 2003), hybrid planning, and planning problems with processes and events (Fox and Long 2002). This problem is further exacerbated by the low-level languages in which plan validators are usually implemented, e.g. the plan validation system used for most planning competitions, VAL (Howey, Long, and Fox 2004), is implemented in C++. Another challenge to plan validation is that the semantics of planning languages have ambiguities, which lead to different interpretations of what constitutes a correct plan. E.g. there are multiple interpretations of sub-typing using "Either" type expressions in PDDL.

In this work we address the aforementioned challenges using an interactive theorem prover (ITP). In particular, we use the ITP Isabelle/HOL (Nipkow, Paulson, and Wenzel 2002), which implements a formal mathematical system combining higher-order logic and simple type theory. Our first contribution is that we formally specify an abstract syntax for the temporal fragment of PDDL 2.1 in Isabelle/HOL and, based on that, formalise its semantics. Compared to a pen-and-paper semantics, this has the advantage that it removes any room for ambiguity. Furthermore, during formalising this fragment of PDDL, we found that certain parts of the semantics as specified by Fox and Long could be simplified. As our second contribution, we implement an executable plan validator for the temporal part of PDDL2.1 and we formally verify, using Isabelle/HOL, that it correctly implements the semantics which we formalised. Our validator checks (i) if a given problem and the candidate plan are well-formed, and (ii) if the candidate plan is indeed a solution to the problem. Lastly, we experimentally show that this validator performs on-par with the state-of-the-art unverified validator VAL.

Background

In this work we build upon previous work by Abdulaziz and Lammich. In their work, they formalised the syntax and semantics of the STRIPS fragment of PDDL in Isabelle/HOL. The syntax was based on a grammar by Kovacs. Their semantics have two parts: (i) a part defining what it means for a PDDL domain, instance or plan to be well-formed and (ii) a part defining the execution semantics of PDDL. The most interesting aspect of well-formedness has to do with typing: since the grammar of PDDL allows for Either-supertype specifications of the form 'obj - Either obj1 obj2$\cdots$', this leads to ambiguities in interpreting the sub-typing relation when, for instance, instantiating a parameter with an Either-type by an object of an Either-type. In this situation, they took the interpretation that this is a valid substitution if each of the object types is reachable, in the sub-typing relation, from at least one of the parameter types. For the execution semantics, they formalised execution semantics of grounded STRIPS in Isabelle/HOL and, based on that, specified the execution semantics of PDDL by instantiating PDDL action schemata into STRIPS ground actions.

Since most of our work here concerns action execution, which is defined at the level of ground actions, this entire paper discusses ground actions and grounded planning problems. The main change we made at the lifted action/problem level to the formalisation by Abdulaziz and Lammich is that we add an action duration constraints as a syntactic element to the abstract syntax element modelling action schemata. We skip here those (modified) definitions and assume that the ground problems and plans were obtained from well-formed PDDL problems and plans, e.g. all parameters to predicates and action schemata are well-typed and action durations in the plan respect the duration constraints in the action schemata. Interested readers should consult the formalisation scripts.

Definition 1 (Propositional Formulae). *A propositional formula ϕ defined over a set of atoms V is either (i) the verum $\top$, (ii) an atom v, s.t. $v \in V$, (iii) a negated propositional formula $\neg\phi$, (iv) a conjunction of two propositional formulae $\phi_1 \wedge \phi_2$, (v) a disjunction of propositional formulae $\phi_1 \vee \phi_2$, or (vi) an implication of two propositional formulae $\phi_1 \longrightarrow \phi_2$. A valuation $\mathcal{A}$ is a mapping of V to the set $\{0, 1\}$. A valuation $\mathcal{A}$ is a model for a formula ϕ, written $\mathcal{A} \models \phi$, iff (i) ϕ is the verum, (ii) if ϕ is an atom, then $\mathcal{A}(v) = 1$, (iii) if ϕ a negated formula $\neg\phi$, then $\mathcal{A} \not\models \phi$, (iv) if ϕ is a conjunction $\phi_1 \wedge \phi_2$, then $\mathcal{A} \models \phi_1$ and $\mathcal{A} \models \phi_2$, (v) if ϕ is a disjunction of propositional formulae $\phi_1 \vee \phi_2$, then $\mathcal{A} \models \phi_1$ or $\mathcal{A} \models \phi_2$, and (vi) if ϕ is an implication of two propositional formulae $\phi_1 \longrightarrow \phi_2$, then $\mathcal{A} \models \phi_2$ if $\mathcal{A} \models \phi_1$.*

Note: sometimes, for notational economy, we treat a valuation as a set. In such cases, a valuation $\mathcal{A} : V \to \{0, 1\}$ is interpreted as the set $\{v \mid \mathcal{A}(v) = 1\}$ and a set of atoms V is interpreted as a valuation which maps any $v \in V$ to 1, and everything else to 0. Also, in the rest of this paper a *state* is synonymous with a valuation.[1]

Definition 2 (Planning Problem). *A planning problem Π is a tuple $\langle P, \delta, \mathcal{I}, \mathcal{G} \rangle$, where (i) P is a set of atoms, each of which is a state characterising proposition, (ii) δ: set of actions, each of which is a tuple $\langle \pi_{start}, \pi_{end}, \pi_{inv} \rangle$ where • π_{start}, π_{end} are start and end* snap actions, *and • π_{inv} is a formula defined over the propositions P. A snap action π is a tuple $\langle \pi_{pre}, \pi_{add}, \pi_{del} \rangle$ where • π_{pre} is its precondition, a formula using propositions P, • $\pi_{add} \subseteq P$ are its positive effects, and • $\pi_{del} \subseteq P$ are its negative effects. (iii) $\mathcal{I}$ is a valuation over P, modelling the initial state, and (iv) $\mathcal{G}$ is the goal state condition, which is a propositional formula defined over P.*

[1] In the formalisation by Abdulaziz and Lammich, on which we base our work, there is support for equalities. This is done by modelling states as sets of formulae. We omit these details here since they are orthogonoal to the main topic, which is the semantics of durative actions.

As a running example we use a planning problem, which models an elevator control situation. There are two passengers (p_0 and p_1), who want to use two elevators (e_0 and e_1) to change floors (f_0 and f_1). The set of state characterising propositions for this planning problem is $P \equiv \bigcup\{\{\textit{(el-at } e_i\, f_j\textit{)}, \textit{(p-at } p_k\, f_j\textit{)}, \textit{(in-el } p_k\, e_i\textit{)}, \textit{(el-op } e_i\textit{)}\} \mid 0 \leq i, j, k \leq 1\}$. The propositions $\textit{(el-at } e_i\, f_j\textit{)}$ and $\textit{(p-at } p_k\, f_j\textit{)}$ encode at which floor an elevator or a passenger currently is. The proposition $\textit{(in-el } p_k\, e_i\textit{)}$ encodes whether a passenger is in an elevator or not. The proposition $\textit{(el-op } e_i\textit{)}$ encodes whether an elevator door is open. The initial state is $\mathcal{I} \equiv \{\textit{(el-at } e_0\, f_0\textit{)}, \textit{(el-at } e_1\, f_1\textit{)}, \textit{(p-at } p_0\, f_1\textit{)}, \textit{(p-at } p_1\, f_0\textit{)}, \textit{(el-op } e_0\textit{)}\}$ and its goal is $\mathcal{G} \equiv \textit{(p-at } p_0\, f_0\textit{)} \wedge \textit{(p-at } p_1\, f_1\textit{)}$. In the initial state passenger p_0 is on floor f_1 and passenger p_1 is on floor f_0. Both passengers want to change floors: passenger p_0 want to move to floor f_0 and passenger p_1 wants to move to floor f_1. This is specified in the goal state formula. Among many actions, the problem has actions to open one elevator's door $\textit{(op } e_1\textit{)} \equiv \langle\langle \neg\textit{(el-op } e_1\textit{)}, \emptyset, \emptyset\rangle, \langle \top, \{\textit{(el-op } e_1\textit{)}\}, \emptyset\rangle, \top\rangle$, to have each of the passengers enter one of the elevators $\textit{(en } p_0\, e_1\, f_1\textit{)} \equiv \langle\langle \textit{(p-at } p_0\, f_1\textit{)} \wedge \textit{(el-at } e_1\, f_1\textit{)}, \emptyset, \emptyset\rangle, \langle \top, \{\textit{(in-el } p_0\, e_1\textit{)}\}, \{\textit{(p-at } p_0\, f_1\textit{)}\}\rangle, \textit{(el-op } e_1\textit{)}\rangle$ and $\textit{(en } p_1\, e_0\, f_0\textit{)} \equiv \langle\langle \textit{(p-at } p_1\, f_0\textit{)} \wedge \textit{(el-at } e_0\, f_0\textit{)}, \emptyset, \emptyset\rangle, \langle \top, \{\textit{(in-el } p_1\, e_0\textit{)}\}, \{\textit{(p-at } p_1\, f_0\textit{)}\}\rangle, \textit{(el-op } e_0\textit{)}\rangle$, and to close an elevator's door $\textit{(cl } e_0\textit{)} \equiv \langle\langle \textit{(el-op } e_0\textit{)}, \emptyset, \emptyset\rangle, \langle \top, \emptyset, \{\textit{(el-op } e_0\textit{)}\}\rangle, \top\rangle$. Each one of the actions has the expected preconditions and effects; e.g. moving the elevator requires the elevator door to be closed during the entire move actions.

Definition 3 (Plan). *A plan is a sequence of tuples $\langle \pi_0, t_0, d_0 \rangle, \ldots, \langle \pi_n, t_n, d_n \rangle$, where, for $1 \leq i \leq n$, $\pi_i \in \delta$ 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 $t_0, \ldots, t_n$ 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 of the plan, and we denote it by htps$(\vec{\pi})$.*

A valid plan for the elevator running example starts with the following four plan actions: $\langle \textit{(op } e_1\textit{)}, 0, 1\rangle$, $\langle \textit{(en } p_0\, e_1\, f_1\textit{)}, 1.25, 0.5\rangle$, $\langle \textit{(en } p_1\, e_0\, f_0\textit{)}, 2, 1\rangle$, and $\langle \textit{(cl } e_0\textit{)}, 3, 1\rangle$.

A central question when it comes to the semantics of temporal planning is that of *plan validity*. A central notion for defining plan validity is that of action *non-interference*.

Definition 4 (Non-interference). *Snap actions π^1 and π^2 are non-interfering iff (i)* $\text{atoms}({\pi^1}_{pre}) \cap ({\pi^2}_{add} \cup {\pi^2}_{del}) = \emptyset$, *(ii)* $\text{atoms}({\pi^2}_{pre}) \cap ({\pi^1}_{add} \cup {\pi^1}_{del}) = \emptyset$, *(iii)* ${\pi^1}_{add} \cap {\pi^2}_{del} = \emptyset$, *and (iv)* ${\pi^2}_{add} \cap {\pi^1}_{del} = \emptyset$.

The first definition of PDDL 2.1 temporal plan validity was posed by Fox and Long 2003. Here we outline their definitions informally, due to lack of space. In their definitions, a central notion was that of a *simple plan*, which can be thought of as a temporal plan whose actions all have zero duration. Execution semantics of simple plans are similar to the semantics of $\forall$-step parallel plans (Rintanen, Heljanko, and Niemelä 2006): more than one action can execute at the same time, given that the actions are non-interfering. A valid temporal plan is defined one that can be compiled into a valid simple plan. In this compilation, each durative action π starting at a time point t and which has duration d is compiled to three actions with duration zero. The first action is π_{start} and it is scheduled to execute at t in the simple plan. The second action is π_{end} and it is scheduled to execute at $t + d$ in the simple plan. The third is a snap action with precondition π_{inv} and no effects. This last action is scheduled to execute in the simple plan multiple times. In particular, it executes once between every two happening time points of the plan iff the two happening time points are between t and $t + d$, inclusive.

Isabelle/HOL

An ITP is a program which implements a formal mathematical system, i.e. a formal language, in which definitions and theorem statements are written, and a set of axioms or derivation rules, using which proofs are constructed. To prove a fact in an ITP, the user provides high-level steps of a proof, and the ITP fills in the details, at the level of axioms, culminating in a formal proof.

We performed the formalisation and the verification using the interactive theorem prover Isabelle/HOL (Nipkow, Paulson, and Wenzel 2002), which is a theorem prover for Higher-Order Logic. Roughly speaking, Higher-Order Logic can be seen as a combination of functional programming with logic. Isabelle/HOL supports the extraction of the functional fragment to actual code in various languages (Haftmann and Nipkow 2007). Isabelle's syntax is a variation of Standard ML combined with standard mathematical notation. Function application is written infix, and functions can be Curried, i.e. function f applied to arguments $x_1\ \ldots\ x_n$ is written as $f\ x_1\ \ldots\ x_n$ instead of the standard notation $f(x_1,\ \ldots\ ,x_n)$.

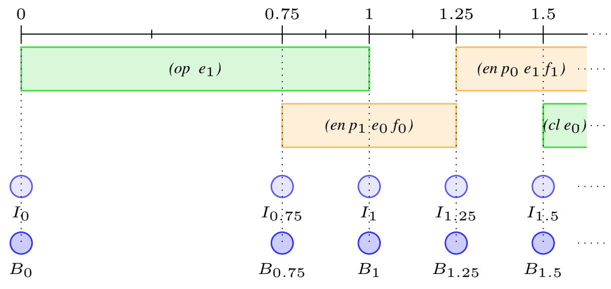
Isabelle is designed for trustworthiness: following the Logic for Computable Functions approach (LCF) (Milner 1972), a small kernel implements the inference rules of the logic, and, using encapsulation features of ML, it guarantees that all theorems are actually proved by this small kernel. Around the kernel there is a large set of tools that implement proof tactics and high-level concepts like algebraic datatypes and recursive functions. Bugs in these tools cannot lead to inconsistent theorems being proved, but only to error messages when the kernel refuses a proof.

All the definitions, theorems and proofs in this paper have been formalised in Isabelle/HOL. The formalisation can be found online[2]. Usually, some definitions are best represented formally in a way which is different from how they represented informally. For instance, a for-loop or a function applied to an indexed sequence in the informal definition are formalised in Isabelle/HOL as recursions over lists. However, there is always a clear resemblance between the formal and the informal definitions and we provide a description associated with the formal definitions.

Semantics of Temporal Planning

One issue with Fox and Long's definition of plan validity is that it is too close to an operational specification of a

[2]DOI:10.5281/zenodo.5784579



$$I_0 = \{\}$$
$$I_{0.75} = \{(\mathit{op}\ \ e_1)_{\mathit{inv}}\}$$
$$I_1 = \{(\mathit{op}\ \ e_1)_{\mathit{inv}}, (\mathit{en}\, p_1\ e_0\, f_0)_{\mathit{inv}}\}$$
$$I_{1.25} = \{(\mathit{en}\, p_1\ e_0\, f_0)_{\mathit{inv}}\}$$
$$I_{1.5} = \{(\mathit{en}\, p_0\ e_1\, f_1)_{\mathit{inv}}\}$$

$$B_0 = \{(\mathit{op}\ \ e_1)_{\mathit{start}}\}$$
$$B_{0.75} = \{(\mathit{en}\, p_1\ e_0\, f_0)_{\mathit{start}}\}$$
$$B_1 = \{(\mathit{op}\ \ e_1)_{\mathit{end}}\}$$
$$B_{1.25} = \{(\mathit{en}\, p_1\ e_0\, f_0)_{\mathit{end}}, (\mathit{en}\, p_1\ e_0\, f_0)_{\mathit{start}}\}$$
$$B_{1.5} = \{(\mathit{cl}\ e_0)_{\mathit{start}}\}$$

Figure 1: Instantiation of Definition 5 with the elevator-running example.

validation algorithm for temporal plans. A negative consequence of that becomes evident when trying to formalise the semantics and pin down all the details: the definitions then become very complicated and unreadable. Although the need for simplifying definitions is generally evident, that need is exacerbated when the definitions are used as specifications against which we formally verify a validator. In that scenario, the semantics should also provide a description of what the validator should do and they should be easily understandable through visual inspection. We resolve that by providing a description of the semantics that abstractly describes what a valid plan is, without appealing to algorithmic constructions like the one of induced happening sequences. We then show that our new definitions are equivalent to the operational definitions of Fox and Long.

Definition 5 (Valid State Sequence)**.** *For $t \in \mathbb{Q}_{\geq 0}$ and a plan $\overrightarrow{\pi}$, let $B_t \equiv \{\pi_{\mathit{start}} \mid \langle \pi, t, d\rangle \in \overrightarrow{\pi}\} \cup \{\pi_{\mathit{end}} \mid \langle \pi, t - d, d\rangle \in \overrightarrow{\pi}\}$ and $I_t \equiv \{\pi_{\mathit{inv}} \mid \langle \pi, t', d\rangle \in \overrightarrow{\pi} \wedge t' < t < t' + d\}$. Also, let $t_0, \ldots, t_n$ be the happening time points of $\overrightarrow{\pi}$. For a sequence of states $M_0, \ldots, M_{n+1}$, we say the sequence of states is valid wrt a plan $\overrightarrow{\pi}$ iff, for every happening time point t_i of $\overrightarrow{\pi}$, we have: (i) $M_i \models \pi_{\mathit{inv}}$, for every $\pi_{\mathit{inv}} \in I_{t_i}$, (ii) $M_i \models \pi_{\mathit{pre}}$, for every $\pi \in B_{t_i}$, (iii) B_{t_i} is pairwise non-interfering, and (iv) $M_{i+1} = (M_i - \bigcup_{\pi \in B_{t_i}} \pi_{\mathit{del}}) \cup \bigcup_{\pi \in B_{t_i}} \pi_{\mathit{add}}$.*

Definition 6 (Valid Plan)**.** *Plan $\overrightarrow{\pi}$ is a valid plan for a problem Π iff there is a state sequence $M_1, \ldots, M_{n+1}$ s.t. $\mathcal{I}, M_1, \ldots, M_{n+1}$ is valid wrt $\overrightarrow{\pi}$ and $M_{n+1} \models \mathcal{G}$.*

Note: above, simultaneous execution of instantaneous ground actions is only allowed for non-interfering ground actions since allowing any ground actions to be executed simultaneously might result in an ambiguous state. We also use the same ground action interference condition defined by Fox and Long.

Figure 1 illustrates the beginning of the instantiation of the elevator running example for Definition 5. At the top of the illustration a timeline is depicted. Below the timeline the first four actions from the valid plan are shown. At the bottom of the illustration the individual sets needed for the state sequence are shown.

Refining the Semantics Towards Executability

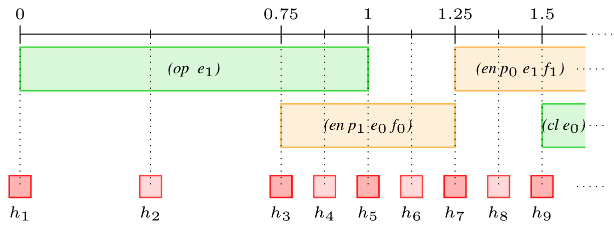
A main goal of this paper is to construct a plan validator which is formally verified wrt the semantics. We do that by following a step-wise refinement approach (Wirth 1971), where we start from the abstractly specified semantics and refine that specification towards an executable program which fulfils those abstractly specified semantics. The next step to refine our semantics is to obtain a version that is closer to the executable program. In this version, we closely follow the semantics given by Fox and Long. A central concept in defining the semantics of temporal plans is that of *happening sequences*. Intuitively, these are the instantaneous changes that happen over the course of plan execution.

Definition 7 (Valid Happening Sequence)**.** *A happening h is a pair $\langle A, r\rangle$, where A is a set of snap actions and $r \in \mathbb{Q}_{\geq 0}$ is the starting time point. For a happening sequence $\langle A_0, r_0\rangle, \ldots, \langle A_n, r_n\rangle$ and a state M_0, we call a state sequence $M_1, \ldots, M_{n+1}$ to be induced by M_0 and the happening sequence iff for every $0 \leq i < m$ (i) $M_i \models \pi_{pre}$, for every $\pi \in A_i$, (ii) A_i is pairwise non-interfering, and (iii) $M_{i+1} = \left(M_i - \bigcup_{\pi \in A_i} \pi_{del}\right) \cup \bigcup_{\pi \in A_i} \pi_{add}$, for $0 \leq i \leq n$. A happening sequence is valid wrt some state iff they induce a state sequence.*

A happening sequence which models the effects and executability of a temporal plan is called an *induced happening sequence*. The validity of a temporal plan is defined as the validity of the induced happening sequence.

Definition 8 (Induced Happening Sequence)**.** *A happening sequence $\langle A_0, r_0\rangle, \ldots, \langle A_m, r_m\rangle$ is an induced happening sequence for a plan $\overrightarrow{\pi}$ with happening time points $t_0, \ldots, t_n$ iff, for all $0 \leq i \leq m$, we have that $A_i \subseteq \bigcup\{\{\langle \pi_{inv}, \emptyset, \emptyset\rangle, \pi_{start}, \pi_{end}\} \mid \langle \pi, t, d\rangle \in \overrightarrow{\pi}\}$ and, for all $\langle \pi, t, d\rangle \in \overrightarrow{\pi}$, (i) there is a happening $\langle A_i, r_i\rangle$ with $r_i = t$ and $\pi_{start} \in A_i$, (ii) there is a happening $\langle A_j, r_j\rangle$ with $r_j = t + d$ and $\pi_{end} \in A_j$, (iii) for each $0 \leq l < n$ with $t \leq t_l < t + d$ there is $\langle A_k, r_k\rangle$ with $t_l < r_k < t_{l+1}$ and $\langle \pi_{inv}, \emptyset, \emptyset\rangle \in A_k$, and (iv) the starting time points $r_0, \ldots, r_m$ are strictly sorted in an ascending order.*

Figure 2 illustrates the beginning of an induced happening sequence for the elevator running example. At the top of the illustration, a timeline with the happening time points is shown. Every start- or end-point of a plan action is a happening time point. In this example, the first five happening time points are: $0, 0.75, 1, 1.25$, and 1.5. Below the timeline the first four actions from the valid plan are shown. For each plan action, the snap actions are placed along the timeline and collected in the happenings, which are symbolized as red squares in the illustration. E.g. for the first plan action $\langle$*(op e_1)*$, 0, 1\rangle$, the start snap action *(op e_1)$_{start}$* is placed at the start of the action, at time point 0 and collected in happening h_1, whereas the end snap action *(op e_1)$_{end}$* is placed at the end of the action, at time point 1 and collected in happening h_5. For every two consecutive happening time points the invariants of all currently running actions need



$h_1 \equiv \langle\{(op\ e_1)_{start}\}, 0.0\rangle$

$h_2 \equiv \langle\{(op\ e_1)_{inv}\}, 0.375\rangle$

$h_3 \equiv \langle\{(en\ p_1\ e_0\ f_0)_{start}\}, 0.75\rangle$

$h_4 \equiv \langle\{(op\ e_1)_{inv}, (en\ p_1\ e_0\ f_0)_{inv}\}, 0.875\rangle$

$h_5 \equiv \langle\{(op\ e_1)_{end}\}, 1.0\rangle$

$h_6 \equiv \langle\{(en\ p_1\ e_0\ f_0)_{inv}\}, 1.125\rangle$

$h_7 \equiv \langle\{(en\ p_1\ e_0\ f_0)_{end}, (en\ p_1\ e_0\ f_0)_{start}\}, 1.25\rangle$

$h_8 \equiv \langle\{(en\ p_0\ e_1\ f_1)_{inv}\}, 1.375\rangle$

$h_9 \equiv \langle\{(cl\ e_0)_{start}\}, 1.5\rangle$

Figure 2: Illustration for the beginning of an induced happening sequence (Definition 8) for the elevator-running example.

to be checked. Therefore, happening h_2 contains the invariant snap action for the first plan action *(op e_1)*. In between the consecutive happening time points 0.75 and 1 the action *(op e_1)* is running as well as the action *(en p_1 e_0 f_0)*, hence the happening h_4 contains the invariant snap actions for both *(op e_1)* and *(en p_1 e_0 f_0)*.

The illustration in Figure 2 only shows one possible induced happening sequence for the valid plan. Definition 8 allows invariant snap actions to be placed arbitrarily in between consecutive happening time points. This is more general than the definition of Fox and Long, which arbitrarily restricts the placement of invariant snap actions to be exactly in the middle of happening time points. We use this placement of invariant actions in the next section, where we give an executable definition of plan validity. Based on the notion of valid happening we define the following notion of plan validity, which is closer to the definition of Fox and Long and to executability.

Definition 9 (Valid Plan II)**.** *Plan $\overrightarrow{\pi}$ is valid for a planning problem Π iff $\overrightarrow{\pi}$ has an induced happening sequence $h_0 \ldots, h_n$ s.t. the happening sequence is valid wrt $\mathcal{I}$ and $M_{n+1} \models \mathcal{G}$, where M_{n+1} is the last state in the induced state sequence.*

At a higher-level, the contrast between Definitions 9 and 6 boils down to that the former specifies plan validity in terms of a happening sequence that should be computed, while the latter specifies validity more abstractly. More specifically, instead of referring to happening sequences, Definitions 6 uses B_t and I_t, which denote the snap actions executing at time t and the set of invariants which should hold at time t, respectively. Accordingly, for Definition 9 we only assert the existence of a sequence of valid states, which can be formalised, in Isabelle/HOL, as a simple recursion on the happening time points of a plan, instead of asserting the existence of an induced happening sequence as in the case of Definition 9.

The following theorem shows that the two definitions are equivalent.

Theorem 1. *For a planning problem Π, a plan $\overrightarrow{\pi}$ is valid* *according to Definition 9 iff it is valid according to Definition 6.*

Proof sketch. Let $t_0, \ldots, t_n$ be the happening time points of $\overrightarrow{\pi}$ after being sorted in ascending order.
($\Rightarrow$) From Definition 9, $\overrightarrow{\pi}$ has an induced happening sequence $\langle A_0, r_0\rangle, \ldots, \langle A_m, r_m\rangle$, and that happening sequence is valid wrt $\mathcal{I}$. Note that $m \geq n$. Our goal here is to show that the induced state sequence of this happening sequence is a valid state sequence, according to Definition 5. Since the induced happening sequence is strictly sorted according to the starting time of the happenings, we know that the different happenings have different starting points. Accordingly, we have, for each t_i, where $0 \leq i \leq n$, there is a happening $\langle A_j, r_j\rangle$, s.t. $B_{t_i} = A_j$ and $t_i = r_j$. Since this induced happening sequence is also a valid happening sequence, the conjuncts (ii), (iii), and (iv) of Definition 5 hold for the induced state sequence. What remains is to show that conjunct (i) holds for the induced state sequence, which states that all action invariants hold during action execution. Observe that conjunct (iii) of Definition 8 asserts that, for each $\langle \pi, t, d\rangle \in \overrightarrow{\pi}$, there is an action $\langle \pi_{\mathit{inv}}, \emptyset, \emptyset\rangle$ between each two happenings that happen during the execution of an action π. The preconditions of this action ensure that the invariants of the action π are not violated during its execution. Accordingly, conjunct (i) holds for the induced state sequence.
($\Leftarrow$) To prove this direction, we need to show that $\overrightarrow{\pi}$ has an induced happening sequence, which is valid wrt $\mathcal{I}$, from a given valid state sequence $\mathcal{I}, M_1, \ldots, M_{n+1}$. Consider the happening sequence $\langle B_{t_0}, t_0\rangle$, $\langle I_{t_1}, \frac{t_0+t_1}{2}\rangle$, $\langle B_{t_1}, t_1\rangle$, $\langle I_{t_2}, \frac{t_1+t_2}{2}\rangle, \ldots, \langle B_{t_{n-1}}, t_{n-1}\rangle, \langle I_{t_n}, \frac{t_{n-1}+t_n}{2}\rangle, \langle B_{t_n}, t_n\rangle$. We now need to show that this happening sequence is a valid one, according to Definition 7. It is easy to see that conjunct (ii) of Definition 7 holds for this happening sequence. To show that the other two conjuncts of Definition 7 hold, we first need to provide a witness state sequence to which those conjuncts apply. The state sequence $\mathcal{I}, M_1, M_1, \ldots, M_{n+1}, M_{n+1}$[3] is the witness: • Conjunct (i) of Definition 7 holds for $\mathcal{I}, M_1, M_1, \ldots, M_{n+1}, M_{n+1}$ because conjunct (i) of Definition 5 holds for $\mathcal{I}, M_1, \ldots, M_{n+1}$, which implies that the preconditions in each action in a happening $\langle B_{t_i}, t_i\rangle$ are entailed by the state M_i, and conjunct (ii) of Definition 5 also holds for $\mathcal{I}, M_1, \ldots, M_{n+1}$, which implies that the preconditions of each happening $\langle I_{t_i}, \frac{t_{i-1}+t_i}{2}\rangle$ are entailed by the state M_{i-1}. • Conjunct (iii) of Definition 7 holds for $\mathcal{I}, M_1, M_1, \ldots, M_{n+1}, M_{n+1}$ because conjunct (iii) of Definition 7 holds for $\mathcal{I}, M_1, \ldots, M_{n+1}$. The last remaining thing is to show that the happening sequence we constructed is an induced happening sequence for $\overrightarrow{\pi}$, according to Definition 8: • The first two conjuncts of Definition 8 hold for this happening sequence because from the definition of B and I. • The third conjunct holds due to the way we construct the happening sequence. • The fourth conjunct holds because we have the happening time points already sorted and the way we construct our happening sequence. This finishes our proof. $\square$

[3]This repetition of states is intended: each state M_i occurs first as a result of executing the happening $\langle B_{t_i}, t_i\rangle$ at state M_{i-1} and then second as a result of executing the happening $\langle I_{t_i}, \frac{t_{i-1}+t_i}{2}\rangle$, which has no effects, at state M_i.

Algorithm 1: The executable specification of plan validity, check-plan, as pseudo-code. In this pseudo-code, Π denotes a planning problem, $\overrightarrow{\pi}$ a plan to be checked, H a sequence of happenings, A_i a set of snap actions, r_i a happening starting time point, and t a happening time point.

```
function insert-action($\langle A_0, r_0\rangle, \ldots, \langle A_m, r_m\rangle, t, \pi$)
  for each $0 \leq i < m$
    if $r_i = t$
      ret $\langle A_0, r_0\rangle, \ldots, \langle A_i \cup \{\pi\}, r_i\rangle, \ldots, \langle A_m, r_m\rangle$
    if $r_{i+1} = t$
      ret $\langle A_0, r_0\rangle, \ldots, \langle A_{i+1} \cup \{\pi\}, r_{i+1}\rangle, \ldots,$
          $\langle A_m, r_m\rangle$
    if $r_i < t < r_{i+1}$
      ret $\langle A_0, r_0\rangle, \ldots, \langle A_i, r_i\rangle, \langle \{\pi\}, r\rangle,$
          $\langle A_{i+1}, r_{i+1}\rangle, \ldots, \langle A_m, r_m\rangle$
function simplify-action($t_0, \ldots t_n, \langle \pi, t, d\rangle, H$)
  $H$ := insert-action($H, t, \pi_{\mathit{start}}$)
  $H$ := insert-action($H, t + d, \pi_{\mathit{end}}$)
  for each $0 \leq i < n$
    if $t \leq t_i < t_{i+1} \leq t + d$
      $H$ := insert-action($H, \frac{r_i + r_j}{2}, \langle \pi_{\mathit{inv}}, \emptyset, \emptyset\rangle$)
  ret $H$
function simplify-plan($\overrightarrow{\pi}$)
  $H := \emptyset$
  for each $\langle \pi, t, d\rangle \in \overrightarrow{\pi}$
    simplify-action($\mathit{htps}(\overrightarrow{\pi}), \langle \pi, t, d\rangle, H$)
  ret $H$
function valid-hap-seq($\langle A_0, r_0\rangle, \ldots, \langle A_m, r_m\rangle, \Pi$)
  $M := \mathcal{I}$
  for each $0 \leq i \leq m$
    if $\exists \pi^1, \pi^2 \in A_i$ and they are interfering
      ret False
    if $\exists \pi \in A_i . M \not\models \pi_{\mathit{pre}}$
      ret False
    $M := \big(M_i - \bigcup_{\pi \in A_i} \pi_{\mathit{del}}\big) \cup \bigcup_{\pi \in A_i} \pi_{\mathit{add}}$
  if $M \models \mathcal{G}$
    ret True
  ret False
function check-plan($\Pi, \overrightarrow{\pi}$)
  $H$ := simplify-plan($\overrightarrow{\pi}$)
  if valid-hap-seq($H, \Pi$)
    ret "valid Plan"
  ret "error"
```

An Executable Verified Validator

The last part of our work is regarding implementing an executable specification of the semantics, i.e. a plan validation algorithm, and formally proving that it is equivalent to the unexecutable specification of the semantics in Definition 6. The formalized semantics are defined with unexecutable abstract mathematical types and depend on several mathematical concepts, e.g. sets and quantifiers. To obtain an executable validator these mathematical types and concepts need to be replaced with efficient algorithms. We use step-wise refinement to replace the abstract specifications in the semantics with algorithms. With step-wise refinement efficient implementations of algorithms can be proven correct by using multiple correctness preserving steps to refine an abstract version of the algorithm towards the efficient implementation. This allows us to formalize concise semantics and implement an efficient validator wrt. the formalized semantics.

We do two main refinement steps: first, we replace the abstract specifications of the semantics with algorithms defined on abstract mathematical types like sets. This is shown in the pseudo-code of our validation algorithm in Algorithm 1, where check-plan is the top-level routine. We then prove the following theorem about it.

Theorem 2. check-plan$(\Pi, \vec{\pi})$ = *"valid Plan" iff $\vec{\pi}$ is valid for the planning problem Π according to Definition 6.*

In the next step-wise refinement step, the abstract mathematical types, like the set operations in valid-hap-seq, are replaced with efficient implementation using balanced trees. Since this step is completely automated with the Containers Framework in Isabelle/HOL (Lochbihler 2013), we do not describe the resulting pseudo-code or the proofs of its equivalence to the pseudo-code from Algorithm 1.

Before we close this section we would like to note two points. First, the formal version of Algorithm 1 includes checks related to PDDL-level well-formedness, like the correctness of typing of action arguments, etc. These details are similar to what was done by Abdulaziz and Lammich and we ignore them here as we only focus on grounded problems. Readers interested in the PDDL-level reasoning can consult the associated formalisation. Second, as one of our goals was to simplify the semantics, we do not assert the presence of a concrete minimum separation, ϵ, between plan actions. In our refinement steps, we are able to derive a validation algorithm which uses arbitrary arithmetic on rational numbers and it is formally proved to implement Definition 6. This is an improvement over the approach of Fox and Long, who claimed in their paper that it is necessary to accept that numeric conditions, including time, will have to be evaluated to a certain tolerance. Indeed, VAL (Howey, Long, and Fox 2004) implements this ϵ and thus requires the ϵ as an extra parameter. This leads to rejecting, otherwise valid, plans if a too large ϵ is given to VAL.

Parsing Problems and Code Generation For parsing, we use an open source parser combinator library written in Standard ML. We note that parsing is a trusted part of our validator, i.e. we have no formal proof that the parser actually recognises the desired grammar and produces the correct abstract syntax tree. However, the parsing combinator approach allows to write concise, clean, and legible parsers, which can be relatively easily checked.

Experimental Evaluation Our validator supports the following PDDL requirements: `:strips`, `:equality`, `:typing`, `:negative-preconditions`, `:disjunctive-preconditions`, `:durative-actions`, and `:duration-inequalities`. For the evaluation of our validator, we compare the validation results and running time of our validator to those of VAL (Howey, Long, and Fox 2004). We use IPC 2014 domains. We used the temporal planners ITSAT (Rankooh and Ghassem-Sani 2015) and Temporal Fast Downward (TFD) (Eyerich, Mattmüller, and Röger 2009) to generate plans for the domains and problems. In all test cases, the validation outcome between our validator and VAL is the same. Our validator is consistently slower than VAL, as can be seen in Figure 3. However, it never needs more than one second to validate any plan. This is a practically acceptable performance, escpecially since our validator uses arbitrary precision arithmetic. We also note that where formally verified code is usually orders of magnitude slower than unverified code due to the difficulty of verifying all code optimisations which are liberally used in unverified code.

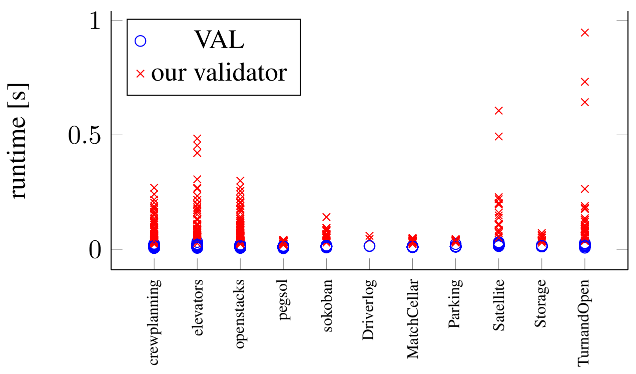


Figure 3: Running times of VAL validator and our validator for some IPC 2014 domains.

Discussion

In this work we presented the first specification of the semantics of the temporal part of PDDL2.1 in a formal mathematical system, namely, Isabelle/HOL. Specifying language semantics in formal mathematical systems has the advantages of removing any ambiguities and providing the basis to build formally verified tool chains to reason about these languages. These advantages of formalising language semantics have been reported by researchers who use ITPs to formalise programming language semantics, e.g. C (Norrish 1998), SML (Kumar et al. 2014), and Rust (Jung et al. 2018). One main purpose of our work was to showcase the merits of this methodology to the planning community.

The semantics and validation of the temporal fragment of PDDL have been extensively studied by multiple authors. We believe our work improves over all the previous approaches in two aspects: the *succinctness* of our specificaiton of the semantics and the *trusworthiness* of our executable validator.

PDDL2.1 was first introduced during the second international planning competition and its semantics were most comprehensively defined by Fox and Long 2003. We base our work on the semantics of Fox and Long. One issue with their semantics noted by earlier authors Claßen, Hu, and Lakemeyer is that it defines plan validity using an executable plan validation algorithm, which is more complicated than what a specification of semantics ought to be. We address that by providing simpler semantics and showing it is equivalent to an executable validator. Our semantics are simpler because they (i) remove the need for a fixed "ϵ" separation between interfering actions, requiring only an arbitrary non-zero separation, (ii) bypass the concept of induced happening sequences, and (iii) do not require that snap actions representing invariants occur exactly between each two happenings which occur while the invariant has to hold. Another difference between our work and that of Fox and Long is that we specify our semantics in Isabelle/HOL wrt abstract syntax which is very close to PDDL syntax.[4] This gives rise to a more detailed specification of the semantics and leaves less room for ambiguities.

Another tangentially related work is that of Gigante et al. 2020. In their work, they studied the complexity of computing plans for different restrictions of the temporal planning as described by Fox and Long.

Another notable planning language which includes temporal elements is ANML (Smith, Frank, and Cushing 2008). The semantics of a language "inspired" by ANML were defined by Cimatti, Micheli, and Roveri 2017. Although Cimatti, Micheli, and Roveri use pen-and-paper definitions, the level of detail of their presentation is closer to ours as they specified an abstract syntax for their language, based on which they defined their semantics. However, our semantics are much more succinct than theirs since we use higher-order logic to specify our semantics, while they specify their semantics in terms of linear temporal logic modulo real arithmetic, which is significantly less expressive than higher-order logic.

Another well-established formalism for studying the semantics of planning and action languages in general is situation calculus (McCarthy and Hayes 1981; Reiter 2001). In that line of work, the work by Claßen, Hu, and Lakemeyer 2007 is the most related to this paper. They showed how to encode a PDDL 2.1 problem as a formula in $\mathcal{ES}$, which is a dialect of first-order logic with interesting computational and meta-theoretic properties introduced by Lakemeyer and Levesque 2004. The main merit of that approach, as stated by Claßen, Hu, and Lakemeyer, is that their semantics are a declarative specification of the semantics of PDDL 2.1 as opposed to the state transition-based semantics of Fox and Long. This has the advantage that all the computational and meta-theoretic properties of $\mathcal{ES}$ apply to it. On the other hand, it has the disadvantage of being less understandable than a state transition-based definition, as one needs to first understand $\mathcal{ES}$. Seen from that perspective, our formalisation three properties:(i) It is clearly state transition-based as our semantics are in terms recursively defined action execution and state transitions. This makes it more readable than the formalisation of Claßen, Hu, and Lakemeyer. (ii) It is also declarative in higher-order logic since, although our top-level definitions are state transition-based, the mechanisms behind the recursive function definitions and the algebraic data types in higher-order logic are all declarative in terms of the axioms of higher-order logic (Krauss 2009; Traytel, Popescu, and Blanchette 2012). (iii) Has less clear computational properties, since general procedures to reason about higher-order logic are all heuristic, since the logic is incomplete. This disadvantage is not an issue, however, in our context given that our goal is to specify a concise semantics for deriving correct by construction software. It can, nonetheless, be remedied by formalising the semantics of $\mathcal{ES}$ in higher-order logic and formally showing, within Isabelle/HOL, the correctness of the encoding of PDDL in $\mathcal{ES}$ from Claßen, Hu, and Lakemeyer.

[4]Interested readers should consult the formalisation.

A lot of work on trustworthiness in planning has focused on plan validation. The state-of-the-art plan validator for temporal plans is VAL (Howey, Long, and Fox 2004). Since VAL implements temporal planning semantics, which is rather involved, in C++, it is difficult to inspect VAL to make sure that it is free of bugs. This, in a sense, defeats one of the main purposes of plan validators: they are supposed to boost trustworthiness by being much simpler than planning systems, making it less likely for them to have bugs and making them easier to inspect. One motivation for our work was to avoid that problem by having a separate concise specification of the semantics which precisely describes what the validator implements. These semantics are then formally connected to an efficient validator. Another approach to temporal plan validation is the one by (Cimatti, Micheli, and Roveri 2017), who compile a given planning problem and a candidate plan into a formula of temporal logic. Plan validation then becomes a satisfiability task for an LTL formula. From a trustworthiness perspective, this approach has the disadvantages that one has to trust the code that implements the compilation to LTL and, more importantly, either one has to trust an LTL model-checker or devise a validator that validates models of LTL formulae. Our approach, on the other hand, trusts a much smaller code base, thanks to the LCF architecture of Isabelle/HOL.

As future work, we would like to connect our formalisation of temporal planning to the formalisation of timed automata by Wimmer and von Mutius 2020. This would enable us to generate formally checkable certificates of unsolvability for temporal planning problems. It would also enable formally verified checking of different properties of a planning domain similar to the ones by Cimatti, Micheli, and Roveri, but with formal guarantees.

Acknowledgements This work was facilitated through the DFG Koselleck Grant NI 491/16-1.

References

Abdulaziz, M.; Gretton, C.; and Norrish, M. 2019. A Verified Compositional Algorithm for AI Planning. In *ITP*.

Abdulaziz, M.; and Lammich, P. 2018. A Formally Verified Validator for Classical Planning Problems and Solutions. In *ICTAI*.

Abdulaziz, M.; Norrish, M.; and Gretton, C. 2018. Formally

Verified Algorithms for Upper-Bounding State Space Diameters. *JAR*.

Cimatti, A.; Micheli, A.; and Roveri, M. 2017. Validating Domains and Plans for Temporal Planning via Encoding into Infinite-State Linear Temporal Logic. In *AAAI*.

Claßen, J.; Hu, Y.; and Lakemeyer, G. 2007. A Situation-Calculus Semantics for an Expressive Fragment of PDDL. In *AAAI*.

Coles, A. J.; Coles, A.; Olaya, A. G.; Celorrio, S. J.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A Survey of the Seventh International Planning Competition. *AI Magazine*.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability Certificates for Classical Planning. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *ICAPS*.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning. *ICAPS*.

Fox, M.; Howey, R.; and Long, D. 2005. Validating plans in the context of processes and exogenous events. In *AAAI*.

Fox, M.; and Long, D. 2002. PDDL+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*.

Fox, M.; and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR*.

Gigante, N.; Micheli, A.; Montanari, A.; and Scala, E. 2020. Decidability and Complexity of Action-Based Temporal Planning over Dense Time. In *AAAI*.

Haftmann, F.; and Nipkow, T. 2007. A code generator framework for Isabelle/HOL. In *TPHOLs*.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *ICTAI*.

Jung, R.; Jourdan, J.; Krebbers, R.; and Dreyer, D. 2018. RustBelt: securing the foundations of the rust programming language. *POPL*.

Kovacs, D. L. 2011. BNF definition of PDDL 3.1. *IPC-2011*.

Krauss, A. 2009. *Automating recursive definitions and termination proofs in higher-order logic*. Ph.D. thesis, Technical University Munich.

Kumar, R.; Myreen, M. O.; Norrish, M.; and Owens, S. 2014. CakeML: a verified implementation of ML. In *POPL*.

Lakemeyer, G.; and Levesque, H. J. 2004. Situations, Si! Situation Terms, No! In *KR*.

Lochbihler, A. 2013. Light-weight containers for Isabelle: efficient, extensible, nestable. In *ITP*.

Long, D. 2000. The AIPS-98 Planning Competition. *AI Magazine*.

McCarthy, J.; and Hayes, P. J. 1981. Some philosophical problems from the standpoint of artificial intelligence. In *Readings in Artificial Intelligence*. Elsevier.

Milner, R. 1972. Logic for computable functions description of a machine implementation. Technical report, Stanford University.

Nipkow, T.; Paulson, L. C.; and Wenzel, M. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*.

Norrish, M. 1998. C formalised in HOL. Technical report, University of Cambridge, Computer Laboratory.

Rankooh, M. F.; and Ghassem-Sani, G. 2015. ITSAT: An Efficient SAT-Based Temporal Planner. *JAIR*.

Reiter, R. 2001. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT Press.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search. *AI*.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML language. In *KEPS*.

Traytel, D.; Popescu, A.; and Blanchette, J. C. 2012. Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *LICS*.

Vallati, M.; Chrpa, L.; Grześ, M.; McCluskey, T. L.; Roberts, M.; Sanner, S.; et al. 2015. The 2014 International Planning Competition: Progress and Trends. *AI Magazine*.

Wimmer, S.; and von Mutius, J. 2020. Verified Certification of Reachability Checking for Timed Automata. In *TACAS*.

Wirth, N. 1971. Program Development by Stepwise Refinement. *Commun. ACM*.