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 et al. 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” 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 (HOL) 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 PDDL 2.1 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 is practical and compare it with 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 · · · ’, 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 prob-
lems. 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 Formule). A propositional for-
matula \( \phi \) 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 \( \phi_1 \land \phi_2 \), or (v) a disjunction of propositional formulae
\( \phi_1 \lor \phi_2 \). A valuation \( A \) is a mapping of \( V \) to the set \{0, 1\}. A
valuation \( A \) is a model for a formula \( \phi \), written \( A \models \phi \),
iff (i) \( \phi \) is the verum, (ii) if \( \phi \) is an atom, then \( A(v) = 1 \),
(iii) if \( \phi \) is a negated formula \( \neg \phi \), then \( A \notmodels \phi \), (iv) if \( \phi \) is a
conjunction \( \phi_1 \land \phi_2 \), then \( A \models \phi_1 \) and \( A \models \phi_2 \), and (v) if \( \phi \)
is a disjunction of propositional formulae \( \phi_1 \lor \phi_2 \), then
\( A \models \phi_1 \) or \( A \models \phi_2 \).

Note: sometimes, for notational economy, we treat a valua-
tion \( A \) as \( A(\top) = 1 \) and \( A(\bot) = 0 \) 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 \( \Pi \) is a
tuple \( (P, \delta, \mathcal{I}, \mathcal{G}) \), where (i) \( P \) is a set of atomic
each of which is a state characterising proposition, (ii) \( \delta \) is
the set of actions, each of which is a tuple \( (\pi_{\text{start}}, \pi_{\text{end}}, \pi_{\text{pre}}) \)
where \( \pi_{\text{start}}, \pi_{\text{end}} \) are start and end snapshot actions, and \( \pi_{\text{pre}} \)
is a formula defined over the propositions \( P \). A snap action \( \pi \) is a tuple
\( (\pi_{\text{pre}}, \pi_{\text{add}}, \pi_{\text{del}}) \) where \( \pi_{\text{pre}} \) is its precondition, a formula
using propositions \( P \). \( \pi_{\text{add}} \subseteq P \) are its positive effects, and
\( \pi_{\text{del}} \subseteq P \) 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 \( P \).

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 el-
evators (\( e_0 \) and \( e_1 \)) to change floors (\( f_0 \) and \( f_1 \)). The set of state
characterising propositions for this planning problem is
\( P = \{\{\text{el-at } e_i f_j\}, \{\text{in-el } e_i \}\} \mid 0 \leq i, j, k \leq 1 \} \).
The propositions \( \{\text{el-at } e_i f_j\} \)

\(^1\)In the formalisation by Abdulaziz and Lammich, on which we
base our work, there is support for equalities. This is done by mod-
eLLing states as sets of formulae. We oMMit these details here since
they are orthogonal to the the semantics of durative actions.

and \( \{\text{in-el } e_i\} \) encode at which floor an elevator or a pas-
enger currently is. The proposition \( \{\text{in-el } p_k e_i\} \) encodes whether
a passenger is in an elevator or not. The proposition \( \{\text{el-op } e_i\} \)
encodes whether an elevator door is open. The initial state
is \( \mathcal{I} = \{\{\text{el-at } e_0 f_0\}, \{\text{el-at } e_1 f_1\}, \{\text{in-el } e_0 \}\} \), \( \{\text{el-op } e_0\} \) and its goal is \( \mathcal{G} = \{\{\text{in-el } p_0 e_0\} \land \{\text{el-op } e_0\} \}
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 \( p_1 \)
want 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 (\( \{\text{op } e_1\} \equiv \langle\langle\{\text{op } e_1\}, \emptyset, \emptyset, \{\top, \{\text{el-op } e_1\}, \emptyset, \top\rangle\rangle \), \( \{\text{op } e_0 \}\) and
\( \{\text{in-el } p_0 e_1\}\), \( \{\text{in-el } p_0 e_1\}\), \( \{\text{el-op } e_1\} \), and to close
an elevator’s door (\( \{\text{cl } e_i\} \equiv \langle\langle\{\text{cl } e_i\}, \emptyset, \emptyset, \emptyset, \{\top, \emptyset, \{\text{el-op } e_i\}\rangle\rangle \). Each one of the actions has the expected
preconditions and effects; e.g. moving the elevator requires
its door to be closed during the entire move action.

Definition 3 (Plan). A plan is a sequence of tuples
\( \langle p_{0}, t_{0}, d_{0} \rangle, \ldots, \langle p_{n}, t_{n}, d_{n} \rangle \), where, for \( 1 \leq i \leq n \), \( p_{i} \) is an action, \( t_{i} \in Q_{\geq 0} \) and \( d_{i} \in Q_{\geq 0} \) are rational numbers,
which to we refer as the starting time point and the duration,
respectively. For a plan \( \mathcal{P} \), we call a sorted sequence
\( t_{0}, \ldots, t_{n} \) of the set of rational numbers \{1 \mid (a, t, d) \in \mathcal{P} \} \) the
time happening time points of the plan, and we denote it by \( bhts(\mathcal{P}) \).

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

A central question when it comes to the semantics of tem-
poral 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 \( \pi^1 \) and \( \pi^2 \)
are non-interfering iff (i) \( \text{atoms}(\pi^1) \cap \pi^2) \) \( \{\text{pre}\} \subseteq \emptyset \)
(ii) \( \text{atoms}(\pi^1) \cap \{\text{pre}\} \subseteq \emptyset \) \( \{\text{pre}\} \subseteq \emptyset \)
and (iv) \( \pi^1 \cap \pi^2 \subseteq \emptyset \).

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 de-
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 sim-
lar to the semantics of \( \mathcal{V} \)-step parallel plans (Rintanen, Hel-
janko, 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 \( \pi \) starting at a time point \( t \) and which has duration
\( \delta \) is compiled to three snap actions with duration zero.
The first action is \( \pi_{\text{start}} \) and it is scheduled to execute at \( t \) in
the simple plan. The second action is \( \pi_{\text{end}} \) and it is scheduled
to execute at \( t + \delta \) in the simple plan. The third is an action
with precondition \( \pi_{\text{pre}} \) and no effects, which is scheduled
to execute in the simple plan multiple times. 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 HOL.
Roughly speaking, HOL 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 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
error messages when the kernel refuses a proof. It does not lead to inconsistent theorems being proved, but only to
cornerstone formalism, 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
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.

---

![Figure 1: Concepts from Def. 5 for the elevator example.](image)

**Definition 5 (Valid State Sequence).** For \( t \in \mathbb{Q}_{\geq 0} \) and a plan \( \mathcal{P} \), let \( B_t \equiv \{ \pi_{\text{start}} | (\pi, t, d) \in \mathcal{P} \} \cup \{ \pi_{\text{end}} | (\pi, t - d, d) \in \mathcal{P} \} \) and \( I_t \equiv \{ \pi_{\text{inv}} | (\pi, t', d) \in \mathcal{P} \wedge t' < t < t' + d \} \). Also, let \( t_0, \ldots, t_n \) be the happening time points of \( \mathcal{P} \). For a sequence of states \( M_0, \ldots, M_{n+1} \), we say the sequence of states is valid wrt a plan \( \mathcal{P} \) iff for every happening time point \( t_i \) of \( \mathcal{P} \), we have:

1. \( M_i \models \pi_{\text{inv}} \), for every \( \pi_{\text{inv}} \in I_t \),
2. \( M_i \models \pi_{\text{pre}} \), for every \( \pi \in B_t \),
3. \( B_{t+1} = (B_t - \bigcup_{\pi \in B_t} \pi_{\text{del}}) \cup \bigcup_{\pi \in B_t} \pi_{\text{add}} \).

**Definition 6 (Valid Plan).** Plan \( \mathcal{P} \) is a valid plan for a problem \( \Pi \) iff there is a state sequence \( M_1, \ldots, M_{n+1} \) s.t. \( \Pi, M_1, \ldots, M_{n+1} \) is valid wrt \( \mathcal{P} \) and \( M_{n+1} \models \mathcal{G} \). Note: above, simultaneous execution of instantaneous ground actions is only allowed for non-interfering ground actions. Otherwise, simultaneous execution might result in a not well-defined 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 Def. 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.

---

^1DOI:10.5281/zenodo.5784579
Definition 7 (Valid Happening Sequence). A happening \( h \) is a pair \((A, r)\), where \( A \) is a set of snap actions and \( r \in \mathbb{Q}_{\geq 0} \) is the starting time point. For a happening sequence \((A_0, r_0), \ldots, (A_n, r_n)\) 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 \( \pi \) iff for every \( 0 \leq i < m \) (i) \( M_i \models \pi_{\mathrm{pre}} \) for every \( \pi \in A_i \), (ii) \( A_i \) is pairwise non-interfering, and (iii) \( M_{i+1} = (M_i - \bigcup_{\pi \in A_i} \pi_{\mathrm{del}}) \cup \bigcup_{\pi \in A_i} \pi_{\mathrm{add}} \) for \( 0 \leq i < n \). A happening sequence is valid wrt some state iff they induce a valid 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 \((A_0, r_0), \ldots, (A_m, r_m)\) is an induced happening sequence for a plan \( \overline{\pi} \) with happening time points \( t_0, \ldots, t_n \) iff for all \( 0 \leq i \leq m \), we have that \( A_i \subseteq \bigcup_{\pi \in \overline{\pi}} \{ \pi_{\mathrm{im}}, \emptyset, \emptyset \} \cup \{ \pi_{\mathrm{start}}, \pi_{\mathrm{end}} \} \) and, for all \( (\pi, t, d) \in \overline{\pi} \), (i) there is a happening \((A_i, r_i)\) with \( r_i = t \) and \( \pi_{\mathrm{start}} \in A_i \), (ii) there is a happening \((A_j, r_j)\) with \( r_j = t + d \) and \( \pi_{\mathrm{end}} \in A_j \), for each \( 0 \leq i < n \) with \( t \leq t_i < t + d \) there is a happening \((A_k, r_k)\) with \( r_k < t_i < t_{i+1} \) and \( (\pi_{\mathrm{mv}}, \emptyset, \emptyset) \in A_k \), and (iv) the starting time points \( t_0, \ldots, t_m \) 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 \((op \ e_1), 0, 1\), the start snap action \((op \ e_1)_{\mathrm{start}}\) is placed at the start of the action, at time point 0 and collected in happening \( h_1 \), whereas the end snap action \((op \ e_1)_{\mathrm{end}}\) is placed at the end of the action, at time point 1 and collected in happening \( h_5 \). For every two consecutive happening points the invariants of all currently running actions need 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. Def. 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 \( \overline{\pi} \) is valid for a planning problem \( \Pi \) iff \( \overline{\pi} \) has an induced happening sequence \( h_0, \ldots, h_n \) s.t. the happening sequence is valid wrt \( \mathcal{I} \) and \( M_{n+1} \models \sum, \) where \( M_{n+1} \) is the last state in the induced state sequence.

At a higher-level, the contrast between Def. 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, Def. 6 uses \( B_I \) and \( t_I \), which denote the snap actions executing at time \( t \) and the set of invariants which should hold at time \( t \), respectively. Accordingly, for Def. 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 Def. 9. The two definitions are equivalent as shown below.

Theorem 1. For a planning problem \( \Pi \), a plan \( \overline{\pi} \) is valid according to Def. 9 iff it is valid according to Def. 6.

Proof sketch. Let \( t_0, \ldots, t_n \) be the happening time points of \( \overline{\pi} \) after being sorted in ascending order. (⇒) From Def. 9, \( \overline{\pi} \) has an induced happening sequence \((A_0, r_0), \ldots, (A_m, r_m)\), 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 Def. 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 \((A_j, r_j)\), s.t. \( B_i = A_j \) and \( t_i = r_j \). Since this induced happening sequence is also a valid happening sequence, the conjuncts (ii), (iii), and (iv) of Def. 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 Def. 8 asserts that, for each \((\pi, t, d) \in \overline{\pi}\), there is an action \((\pi_{\mathrm{mv}}, \emptyset, \emptyset)\) between each two happening.
that happen during the execution of an action \( \pi \). The preconditions of this action ensure that the invariants of the action \( \pi \) 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 \( \bar{\pi} \) has an induced happening sequence, which is valid wrt \( I \), from a given valid state sequence \( I, M_1, \ldots, M_{n+1} \). Consider the happening sequence \( \langle B_{t_0}, t_0 \rangle, \langle I_{t_1}, \frac{t_1+t_2}{2} \rangle, \langle B_{t_2}, t_1 \rangle, \langle I_{t_2}, \frac{t_2+t_3}{2} \rangle, \ldots, \langle B_{t_{n-1}}, t_{n-1} \rangle, \langle I_{t_n}, \frac{t_n+t_{n+1}}{2} \rangle, \langle B_{t_{n+1}}, t_n \rangle \). We now need to show that this happening sequence is a valid one, according to Def. 7. It is easy to see that conjunct (ii) of Def. 7 holds for \( I \) of this happening sequence. To show that the other two conjuncts of Def. 7 hold, we first need to provide a witness state sequence to which those conjuncts apply. The state sequence \( I, M_1, M_1, \ldots, M_{n+1}, M_{n+1} \) is the witness: • Conjunct (i) of Def. 7 holds for \( I, M_1, M_1, \ldots, M_{n+1}, M_{n+1} \) because conjunct (i) of Def. 5 holds for \( I, M_1, M_1, \ldots, M_{n+1} \), 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 Def. 5 also holds for \( I, M_1, \ldots, M_{n+1} \), which implies that the preconditions of each happening \( \langle I_{t_i}, \frac{t_i+t_{i+1}}{2} \rangle \) are entailed by the state \( M_{i-1} \). • Conjunct (iii) of Def. 7 holds for \( I, M_1, M_1, \ldots, M_{n+1}, M_{n+1} \) because conjunct (iii) of Def. 7 holds for \( I, M_1, \ldots, M_{n+1} \). The last remaining thing is to show that the happening sequence we constructed is an induced happening sequence for \( \bar{\pi} \), according to Def. 8; • The first two conjuncts of Def. 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.

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 Def. 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. those 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(\( \Pi, \bar{\pi} \)) = “valid Plan” iff \( \bar{\pi} \) is valid for the planning problem \( \Pi \) according to Def. 6.

Lemma 1. Let \( \bar{\pi} \) be a plan and \( H \) and \( H' \) be induced happening sequences for \( \bar{\pi} \). If a state sequence is an induced state sequence by a state \( M_0 \) and \( H \), then there is a state sequence induced by \( M_0 \) and \( H' \), where the last state of the

---

Algorithm 1: The executable specification of plan validity, check-plan, as pseudo-code. In this pseudo-code, \( \Pi \) denotes a planning problem, \( \bar{\pi} \) 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 < 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 \( \langle A_0, r_0 \rangle, \ldots, \langle A_i, r_i \rangle, \{ \pi \}, r \), \( \langle A_{i+1}, r_{i+1} \rangle, \ldots, \langle A_m, r_m \rangle \)

function simplify-action(\( \langle A_0, r_0 \rangle, \ldots, \langle A_n, r_n \rangle, \pi, t, d \), H)
  H := insert-action(H, t, \{ \pi \})
  H := insert-action(H, t + d, \{ \pi \})
  for each 0 < i < n
    if \( t \leq t < t_{i+1} \leq t + d \)
      H := insert-action(H, \( t_{i+1} + d \), \{ \pi \}, t, d \}
  ret H

function simplify-plan(\( \bar{\pi} \))
  H := \{ \}
  for each \( \langle \pi, t, d \rangle \in \bar{\pi} \)
    simplify-action(hps(\( \bar{\pi} \)), \{ \pi, t, d \}, H)
  ret H

function valid-hap-seq(\( \langle A_0, r_0 \rangle, \ldots, \langle A_m, r_m \rangle, \Pi \))
  M := I
  for each 0 < i < m
    if \( i = 0 \)
      if \( \exists \pi^1, \pi^2 \in A_i \) and they are interfering
        ret False
      else False
    ret True
    ret False

function check-plan(\( \Pi, \bar{\pi} \))
  H := simplify-plan(\( \bar{\pi} \))
  if valid-hap-seq(H, \Pi)
    ret “valid Plan”
  ret “error”
```
two sequences is the same.

Proof sketch. Firstly, let \( H (H') \) be \((A_0, r_0), (A_1, r_1), \ldots, (A_m, r_m)\) \((A_0, r_0), (A'_1, r'_1), \ldots, (A_m, r'_m)\) let \( t_0, t_1, \ldots, t_n \) be the happening time points of \( \overline{H} \), and let \( M_1, M_2, \ldots, M_{n+1} \) be the induced state sequences of \( T \) and \( H (H') \). Because of the fourth conjunct of Def. 8, we have a monotonically increasing mapping \( f \) from \( \{0, 1, \ldots, m\} \) to \( \{0, 1, \ldots, m\} \) such that, for \( 0 \leq i \leq n \), \( t_i = r_{f(i)} \) \( (t_i = r_{f(i)}' \) and \( f(n) = m \) \( (f'(n) = m') \). Also, from the third conjunct of Def. 8 we have that, for \( 0 \leq i \leq n \), \( A_{f(i)} (A'_{f'(i)}) \) has no invariant snap actions and, accordingly, \( A_{f(i)} = A'_{f'(i)} \), and for \( j \in \{0, 1, \ldots, m\} \setminus \{f(0), f(1), \ldots, f(n)\} \) \( (j \in \{0, 1, \ldots, m\} \setminus \{f(0), f(1), \ldots, f'(n)\}) \), \( A_j \) has only invariant snap actions, i.e. \( A_j \subseteq \{\phi, 0, 0\} \mid \phi \) is propositional formula. From the two previous statements, we conclude that \( M_{f(i)} = M'_{f'(i)} \), for \( 0 < i \leq n \), which finishes our proof.

Lemma 2. For any plan \( \overline{H} \), simplify-plan(\( \overline{H} \)) is an induced happening sequence for the plan \( \overline{H} \).

Proof sketch. This follows from Def. 8.

Lemma 3. For any happening sequence \( H \) and planning problem \( \Pi \), valid-hap-seq(\( H, \Pi \)) is true iff \( H \) is a valid happening sequence \( \Pi \).

Proof sketch. This follows from Def. 7.

Proof of Theorem 2. The theorem follows from Lemmas 2, 3, and 1, and Theorem 1.

A validator has to be executable and efficient and thus the implementation of a validator is more complicated than the formalisation of the semantics.

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, \( \epsilon \), 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 Def. 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 \( \epsilon \) and thus requires the \( \epsilon \) as an extra parameter. This leads to rejecting, otherwise valid, plans if a too large \( \epsilon \) is given to \( \epsilon \).

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, especially since our validator uses arbitrary precision arithmetic. We also note that 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.

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 (Norris 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 studied by multiple authors. We believe our work improves over all the previous approaches in two aspects: the succinctness of our semantics specificaiton and the trustworthiness 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 “\texttt{\&}” 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. 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 HOL to specify our semantics, while they specify their semantics in terms of linear temporal logic modulo real arithmetic, which is significantly less expressive than HOL.

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 Lakeymeyer 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 HOL since, although our top-level definitions are state transition-based, the mechanisms behind the recursive function definitions and the algebraic data types in HOL are all declarative in terms of the axioms of HOL (Krauss 2009; Traytel, Popescu, and Blanchette 2012). (iii) Has less clear computational properties, since general procedures to reason about HOL 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 HOL and formally showing, within Isabelle/HOL, the correctness of the encoding of PDDL in $\mathcal{ES}$ from Claßen, Hu, and Lakemeyer.

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.

---

4Interested readers should consult the formalisation.
Acknowledgements

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

References


Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. JAIR.


