Formal Semantics and Formally Verified Validation for Temporal Planning

Mohammad Abdulaziz[1,2] and **Lukas Koller**[1]

[1]Techniche Universität München, Germany

[2]King's College London, United Kingdom

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 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 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$\cdots$', 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 $\phi_1 \wedge \phi_2$, or (v) a disjunction of propositional formulae $\phi_1 \vee \phi_2$. 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 $\mathcal{A} \models \phi_2$, and (v) if ϕ is a disjunction of propositional formulae $\phi_1 \vee \phi_2$, then $\mathcal{A} \models \phi_1$ or $\mathcal{A} \models \phi_2$.*

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.*

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 $P \equiv \bigcup\{\{$*(el-at e_i f_j)*, *(p-at p_k f_j)*, *(in-el p_k e_i)*, *(el-op e_i)*$\} \mid 0 \leq i, j, k \leq 1\}$. The propositions *(el-at e_i f_j)* and *(p-at p_k f_j)* encode at which floor an elevator or a passenger currently is. The proposition *(in-el p_k e_i)* encodes whether a passenger is in an elevator or not. The proposition *(el-op e_i)* encodes whether an elevator door is open. The initial state is $\mathcal{I} \equiv \{$*(el-at e_0 f_0)*, *(el-at e_1 f_1)*, *(p-at p_0 f_1)*, *(p-at p_1 f_0)*, *(el-op e_0)*$\}$ and its goal is $\mathcal{G} \equiv$ *(p-at p_0 f_0)* $\wedge$ *(p-at p_1 f_1)*. 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 *(op e_1)* $\equiv \langle\langle\neg$*(el-op e_1)*$, \emptyset, \emptyset\rangle, \langle\top, \{$*(el-op e_1)*$\}, \emptyset\rangle, \top\rangle$, to have each of the passengers enter one of the elevators *(en p_0 e_1 f_1)* $\equiv \langle\langle$*(p-at p_0 f_1)* $\wedge$ *(el-at e_1 f_1)*$, \emptyset, \emptyset\rangle, \langle\top, \{$*(in-el p_0 e_1)*$\}, \{$*(p-at p_0 f_1)*$\}\rangle,$ *(el-op e_1)*$\rangle$ and *(en p_1 e_0 f_0)* $\equiv \langle\langle$*(p-at p_1 f_0)* $\wedge$ *(el-at e_0 f_0)*$, \emptyset, \emptyset\rangle, \langle\top, \{$*(in-el p_1 e_0)*$\}, \{$*(p-at p_1 f_0)*$\}\rangle,$ *(el-op e_0)*$\rangle$, and to close an elevator's door *(cl e_0)* $\equiv \langle\langle$*(el-op e_0)*$, \emptyset, \emptyset\rangle, \langle\top, \emptyset, \{$*(el-op e_0)*$\}\rangle, \top\rangle$. 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 $\langle \pi_0, t_0, d_0\rangle, \ldots, \langle \pi_n, t_n, d_n\rangle$, 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 $\mathit{htps}(\vec{\pi})$.*

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

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)* $\mathrm{atoms}(\pi^1{}_{pre}) \cap (\pi^2{}_{add} \cup \pi^2{}_{del}) = \emptyset$*, (ii)* $\mathrm{atoms}(\pi^2{}_{pre}) \cap (\pi^1{}_{add} \cup \pi^1{}_{del}) = \emptyset$*, (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 snap 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 an action with precondition π_{inv} and no effects, which is scheduled to execute in the simple plan multiple times. It executes once

[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 the semantics of durative actions.

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 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 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.

[2]DOI:10.5281/zenodo.5784579

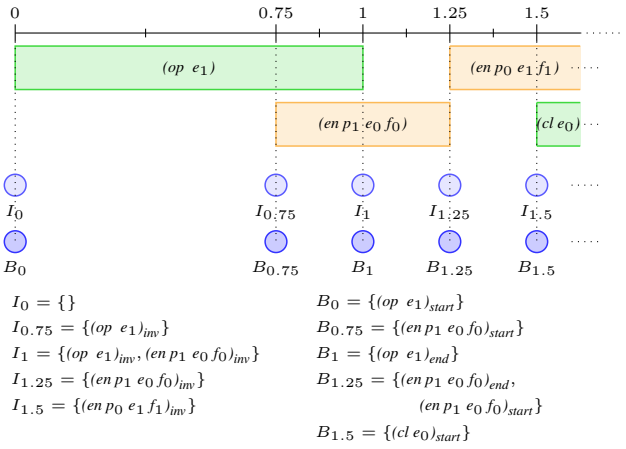

Figure 1: Concepts from Def. 5 for the elevator example.

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

Definition 6 (Valid Plan)**.** *Plan $\vec{\pi}$ 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 $\vec{\pi}$ 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.

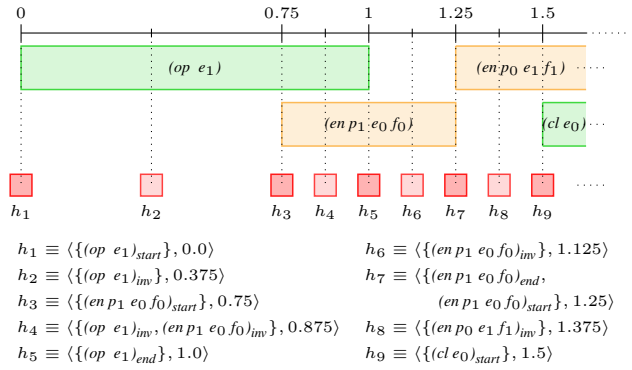

Figure 2: Illustration for the beginning of an induced happening sequence (Def. 8) for the elevator-running example.

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 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 $\langle A_0, r_0\rangle, \ldots, \langle A_m, r_m\rangle$ is an induced happening sequence for a plan $\vec{\pi}$ with happening time points $t_0, \ldots, t_n$ iff, for all $0 \leq i \leq m$, we have that $A_i \subseteq \bigcup\{\{\langle \pi_{inv}, \emptyset, \emptyset\rangle, \pi_{start}, \pi_{end}\} \mid \langle \pi, t, d\rangle \in \vec{\pi}\}$ and, for all $\langle \pi, t, d\rangle \in \vec{\pi}$, (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 e1)*$, 0, 1\rangle$, the start snap action *(op e1)*$_{start}$ is placed at the start of the action, at time point 0 and collected in happening h_1, whereas the end snap action *(op e1)*$_{end}$ 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 to be checked. Therefore, happening h_2 contains the invariant snap action for the first plan action *(op e1)*. In between the consecutive happening time points 0.75 and 1 the action *(op e1)* is running as well as the action *(en p1 e0 f0)*, hence the happening h_4 contains the invariant snap actions for both *(op e1)* and *(en p1 e0 f0)*.

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 $\vec{\pi}$ is valid for a planning problem Π iff $\vec{\pi}$ 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 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_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 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 Π, a plan $\vec{\pi}$ 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 $\vec{\pi}$ after being sorted in ascending order.
($\Rightarrow$) From Def. 9, $\vec{\pi}$ 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 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 $\langle A_j, r_j\rangle$, 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 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 $\langle \pi, t, d\rangle \in \vec{\pi}$, there is an action $\langle \pi_{inv}, \emptyset, \emptyset\rangle$ 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 $\vec{\pi}$ 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 Def. 7. It is easy to see that conjunct (ii) of Def. 7 holds for 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 $\mathcal{I}, M_1, M_1, \ldots, M_{n+1}, M_{n+1}$[3] is the witness: • Conjunct (i) of Def. 7 holds for $\mathcal{I}, M_1, M_1, \ldots, M_{n+1}, M_{n+1}$ because conjunct (i) of Def. 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 Def. 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 Def. 7 holds for $\mathcal{I}, M_1, M_1, \ldots, M_{n+1}, M_{n+1}$ because conjunct (iii) of Def. 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 $\vec{\pi}$, 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.

[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, $\vec{\pi}$ 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, π_{start})
 $H :=$ insert-action($H, t + d, \pi_{end}$)
 for each $0 \leq i < n$
  if $t \leq t_i < t_{i+1} \leq t + d$
   $H :=$ insert-action($H, \frac{r_i+r_j}{2}, \langle \pi_{inv}, \emptyset, \emptyset\rangle$)
 ret H
function simplify-plan($\vec{\pi}$)
 $H := \emptyset$
 for each $\langle \pi, t, d\rangle \in \vec{\pi}$
  simplify-action(*htps*($\vec{\pi}$), $\langle \pi, t, d\rangle, H$)
 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 $\exists \pi \in A_i . M \not\models \pi_{pre}$
   ret *False*
  $M := \left(M - \bigcup_{\pi \in A_i} \pi_{del}\right) \cup \bigcup_{\pi \in A_i} \pi_{add}$
 if $M \models \mathcal{G}$
  ret *True*
 ret *False*
function check-plan($\Pi, \vec{\pi}$)
 $H :=$ simplify-plan($\vec{\pi}$)
 if valid-hap-seq(H, Π)
  ret "valid Plan"
 ret "error"

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

Lemma 1. *Let* $\vec{\pi}$ *be a plan and* H *and* H' *be induced happening sequences for* $\vec{\pi}$*. 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*

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 $\vec{\pi}$, and let $M_1, M_2, \ldots, M_{m+1}$ ($M'_1, M'_2, \ldots, M_{m'+1}$) be the induced state sequences of $\mathcal{I}$ and H (H'). Because of the fourth conjunct of Def. 8, we have a monotonically increasing mapping f (f') from $\{0, 1, \ldots, n\}$ to $\{0, 1, \ldots, m\}$ ($\{0, 1, \ldots, m'\}$), 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. $A_j \subseteq \{\langle \phi, \emptyset, \emptyset \rangle \mid \phi \text{ is propositional formula}\}$. 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 $\vec{\pi}$, $\text{simplify-plan}(\vec{\pi})$ is an induced happening sequence for the plan $\vec{\pi}$.*

Proof sketch. This follows from Def. 8. □

Lemma 3. *For any happening sequence H and planning problem Π, $\text{valid-hap-seq}(H, \Pi)$ is true iff H is a valid happening sequence wrt. $\mathcal{I}$.*

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, 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 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 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.

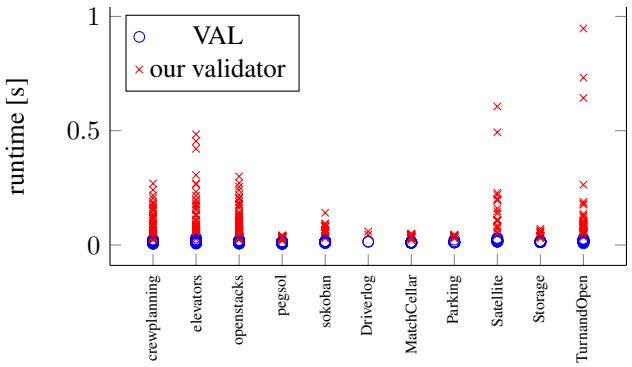

Figure 3: Validation running times for IPC 2014 domains.

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 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 (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 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 *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 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 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 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.

[4] Interested readers should consult the formalisation.

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 *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, 4:1–4:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Abdulaziz, M.; and Lammich, P. 2018. A Formally Verified Validator for Classical Planning Problems and Solutions. In *IEEE 30th International Conference on Tools with Artificial Intelligence, ICTAI 2018, 5-7 November 2018, Volos, Greece*, 474–479. IEEE.

Abdulaziz, M.; Norrish, M.; and Gretton, C. 2018. Formally Verified Algorithms for Upper-Bounding State Space Diameters. *J. Autom. Reason.*, 61(1-4): 485–520.

Cimatti, A.; Micheli, A.; and Roveri, M. 2017. Validating Domains and Plans for Temporal Planning via Encoding into Infinite-State Linear Temporal Logic. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, 3547–3554. AAAI Press.

Claßen, J.; Hu, Y.; and Lakemeyer, G. 2007. A Situation-Calculus Semantics for an Expressive Fragment of PDDL. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, 956–961. AAAI Press.

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 Mag.*, 33(1): 83–88.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability Certificates for Classical Planning. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, 88–97. AAAI Press.

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 *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 1151–1156. AAAI Press / The MIT Press.

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 *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, 9859–9866. AAAI Press.

Haftmann, F.; and Nipkow, T. 2007. A Code Generator Framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, 294–301. IEEE Computer Society.

Jung, R.; Jourdan, J.-H.; Krebbers, R.; and Dreyer, D. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.*, 2(POPL): 66:1–66:34.

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 *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, 179–192. ACM.

Lakemeyer, G.; and Levesque, H. J. 2004. Situations, Si! Situation Terms, No! In *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, June 2-5, 2004*, 516–526. AAAI Press.

Lochbihler, A. 2013. Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, 116–132. Springer.

Long, D.; Kautz, H. A.; Selman, B.; Bonet, B.; Geffner, H.; Koehler, J.; Brenner, M.; Hoffmann, J.; Rittinger, F.; Anderson, C. R.; Weld, D. S.; Smith, D. E.; and Fox, M. 2000. The AIPS-98 Planning Competition. *AI Mag.*, 21(2): 13–33.

McCarthy, J.; and Hayes, P. J. 1981. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Readings in Artificial Intelligence*.

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*, volume 2283 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-540-43376-7.

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. *J. Artif. Intell. Res.*, 53: 541–632.

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. *Artif. Intell.*, 170(12-13): 1031–1080.

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 *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, 596–605. IEEE Computer Society.

Vallati, M.; Chrpa, L.; Grzes, M.; McCluskey, T. L.; Roberts, M.; and Sanner, S. 2015. The 2014 International Planning Competition: Progress and Trends. *AI Mag.*, 36(3): 90–98.

Wimmer, S.; and von Mutius, J. 2020. Verified Certification of Reachability Checking for Timed Automata. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, 425–443. Springer.

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

Appendix: Running Example

The problem is specified in PDDL by the PDDL-domain `temp-elevators` and PDDL-problem `temp-elevators -prob1` in Listing 1 and 2.

Listing 1: PDDL-domain for elevator planning problem

```
(define (domain temp-elevators)
  (:requirements :typing :negative-preconditions
       :durative-actions :duration-inequalities)
  (:types floor - object elevator - object passenger - object)
  (:predicates
      (el-at ?e - elevator ?f - floor)
      (p-at ?e - passenger ?f - floor)
      (in-el ?p - passenger ?e - elevator)
      (el-op ?e - elevator)
  )
  (:functions (el-dur ?from - floor ?to - floor) - number)
  (:durative-action mv
      :parameters (?e - elevator ?from - floor ?to - floor)
      :duration (= ?duration (el-dur ?from ?to))
      :condition (and (at start (el-at ?e ?from))
                      (over all (not (el-op ?e))))
      :effect (and (at start (not (el-at ?e ?from)))
                   (at end (el-at ?e ?to)))
  )
  (:durative-action op
      :parameters (?e - elevator)
      :duration (= ?duration 1)
      :condition (at start (not (el-op ?e)))
      :effect (at end (el-op ?e))
  )
  (:durative-action cl
      :parameters (?e - elevator)
      :duration (= ?duration 1)
      :condition (at start (el-op ?e))
      :effect (at end (not (el-op ?e)))
  )
  (:durative-action en
      :parameters (?p - passenger ?e - elevator ?f - floor)
      :duration (<= ?duration 1)
      :condition (and (at start (and (p-at ?p ?f) (el-at ?e ?f))
           )
                      (over all (el-op ?e)))
      :effect (and (at start (not (p-at ?p ?f)))
                   (at end (in-el ?p ?e)))
  )
  (:durative-action ex
      :parameters (?p - passenger ?e - elevator ?f - floor)
      :duration (<= ?duration 1)
      :condition (and (at start (and (in-el ?p ?e) (el-at ?e ?f)
           ))
                      (over all (el-op ?e)))
      :effect (and (at start (not (in-el ?p ?e)))
                   (at end (p-at ?p ?f)))
  )
)
```

Listing 2: PDDL-problem for elevator planning problem

```
(define (problem temp-elevators-prob1)
(:domain elevators)
(:objects f0 f1 - floor p0 p1 - passenger e0 e1 - elevator)
(:init (el-at e0 f0)
    (el-at e1 f1)
    (el-op e0)
    (p-at p0 f1)
    (p-at p1 f0)
    (= (el-dur f0 f1) 1)
    (= (el-dur f1 f0) 1))
(:goal (and (p-at p0 f0) (p-at p1 f1))))
```

In the running example, there are two passengers `p0` and `p1`, who want to use two elevators `e0` and `e1`. Passenger `p0` wants to move from floor `f1` to floor `f0`, whereas the passenger `p1` wants to move from floor `f0` to floor `f1`. The PDDL-domain (listing 1) specifies actions to move an elevator (*mv*), to enter and exit an elevator (*en* and *ex*), and to open and close and elevator door (*op* and *cl*). Each one of the actions has the expected preconditions and effects; e.g. *mv* requires the elevator door to be closed during the entire move actions. For a passenger to enter an elevator, *en* requires the elevator door to be open. A PDDL-domain defines action schemas, which are instantiated with arguments to obtain executable ground actions. For the instantiation of an action schema the conditions and effects of are partitioned by their annotation (`at start`, `at end`, and `over all`) to produce the snap actions π_{start}, π_{end}, and the invariants π_{inv}. The snap action π_{start} contains all instantiated conditions and effects annotated with `at start`, whereas the snap action π_{end} contains all instantiated conditions and effects annotated with `at end`, and π_{inv} contains all instantiated invariants.

Listing 3: Valid plan for elevator planning problem

```
0: (op e1)[1]
1.25: (en p0 e1 f1)[0.5]
2: (cl e1)[1]
3: (mv e1 f1 f0)[1]
4: (op e1)[1]
5.25: (ex p0 e1 f0)[0.5]
0.75: (en p1 e0 f0)[0.5]
1.5: (cl e0)[1]
2.5: (mv e0 f0 f1)[1]
3.5: (op e0)[1]
4.75: (ex p1 e0 f1)[0.5]
```

Listing 3 shows a valid plan for the running example. A plan for a temporal planning problem is a schedule of actions to execute. The plan specifies the starting time point and a duration for each action. In PDDL the starting time points are denoted in front of a colon (`:`) at the beginning of each line. The duration of an action is denoted in brackets (`[d]`, with duration d) at the end of a line.

In the elevator example the end snap action for the action instantiation of *cl e0* and the start snap action for the action instantiation of *op e0* are interfering ground actions.

- $(cl\ e0)_{end} = \langle \top, \emptyset, \{(el\text{-}op\ e0)\} \rangle$
- $(op\ e0)_{start} = \langle \neg(el\text{-}op\ e0), \emptyset, \emptyset \rangle$

The end snap action for *cl e0* contains the negative effect *el-op e0*, which is a precondition of the start snap action of *op e0*. This violates the first condition for non-interference between ground actions (Definition 4). Intuitively this means, there has to be a non-zero, but arbitrarily small, time interval where the elevator door is open before it can be closed again.

Appendix: Isabelle/HOL Listings

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 $f(x_1, \ \ldots\ , x_n)$.

Formalising the Semantics

The following code snippet shows our formalization of the ground action interference according to Definition 4.

Listing 4: Ground Action interference and Happening Sequence validity

```
    definition acts_non_intrf :: "ground_action ⇒ ground_action ⇒
         bool" where
2     "acts_non_intrf a b ⟷
      (let
4       add_a = set(adds(effect a));
        del_a = set(dels(effect a));
6       pre_a = Atom ` atoms (precondition a);
        add_b = set(adds(effect b));
8       del_b = set(dels(effect b));
        pre_b = Atom ` atoms (precondition b) in
10      pre_a ∩ (add_b ∪ del_b) = {} ∧
```

```
      pre_b ∩ (add_a ∪ del_a) = {} ∧
      add_a ∩ del_b = {} ∧
      add_b ∩ del_a = {})"
```

The formalization of the ground action interference is straight forward. The function `acts_non_intrf` returns a boolean value indicating whether the given ground actions are non-interfering according to Definition 4.

The following code snippet shows our formalization of Definition 5.

Listing 5: New Semantics

```
definition acts_of_plan_at :: "time ⇒ plan ⇒ ground_action set
      " where
  "acts_of_plan_at t_i π s =
    {a_π. ∃π. (t_i,π) ∈ simple_acts π s ∧ Some a_π = res_inst π}
  ∪ {a_start. ∃π. (t_i,π) ∈ durative_acts π s
  ∧ Some a_start = res_inst_snap_action π At_Start}
  ∪ {a_end. ∃t' π. (t',π) ∈ durative_acts π s
  ∧ t_i = t' + duration π
  ∧ Some a_end = res_inst_snap_action π At_End}"

definition invs_of_plan_at :: "time ⇒ plan ⇒ (object atom)
      formula set" where
  "invs_of_plan_at t_i π s =
    {inv. ∃t_π π. (t_π,π) ∈ durative_acts π s
  ∧ t_π < t_i ∧ t_i ≤ t_π + (duration π)
  ∧ Some inv = res_inst_inv π}"

fun apply_eff :: "ground_action set ⇒ world_model ⇒
      world_model" where
  "apply_eff A_i M =
    (M - ⋃ (set ` dels ` effect ` A_i)) ∪ ⋃ (set ` adds `
        effect ` A_i)"

fun valid_state_seq :: "world_model ⇒ time list ⇒ plan ⇒
      world_model ⇒ bool" where
  "valid_state_seq M [] π s M' ⟷ (M = M')"
| "valid_state_seq M (t_i#ts) π s M' ⟷
  (let A_i = acts_of_plan_at t_i π s in
  (∀i ∈ invs_of_plan_at t_i π s. M ^c ⊨= i)
  ∧ (∀a ∈ A_i. M ^c ⊨= precondition a)
  ∧ (∀a ∈ A_i. ∀b ∈ A_i. a ≠ b ⟶ acts_non_intrf a b)
  ∧ valid_state_seq (apply_eff A_i M) ts π s M')"
```

Note that in our formalization we make a distinction between durative and non-durative (instantaneous) actions. Therefore, the function `simple_acts` return all non-durative plan actions in a given plan, and the function `durative_acts` return all durative plan actions in a given plan. The function `res_inst_snap_action` instantiates the snap action of a given durative plan action for a given temporal annotation: `At_Start` or `At_End`, corresponding to π_{start} and π_{end}. The function `res_inst` instantiates the ground action for a non-durative plan action. The function `res_inst_inv` instantiates the invariant formula (π_{inv}) for a given durative plan action. Given a time point `t` and a plan π`s`, the function call `acts_of_plan_at t` π`s` returns the set B_t from Definition 5. Similarly, given a time point `t` and a plan π`s`, the function call `invs_of_plan_at t` π`s` returns the set I_t from Definition 5.

The function `apply_eff` applies the effects of a set of ground actions to a given state and returns the resulting state. The function `valid_state_seq` recursively formalizes the validity of a state sequence according to Definition 5. The state sequence is never explicitly constructed. In each recursion step it is checked that (i) all invariants for the current time point are satisfied by the current state, (ii) the precondition of each action for the current time point is satisfied by the current state, and (iii) all actions for the current time point are pairwise non-interfering.

The following code snippet then shows our formalization of plan validity according to Definition 6.

Listing 6: New Semantics

```
lemma "valid_plan π s ≡ wf_plan π s
  ∧ (∃htps M'. htps_seq π s htps
  ∧ valid_state_seq I htps π s M' ∧ M' ^c ⊨= (goal P))"
  unfolding valid_plan_def valid_plan_from_def by auto
```

The predicate `wf_plan` characterizes a well-formed plan, and the predicate `htps_seq` characterizes the sequence of happening time points for a plan. The predicate `valid_plan` characterizes the plan validity of a plan according to Definition 6.

Formalising the Refined Semantics

Next we are showing our formalization of our refined semantics, which are based around the induced happening sequence (Definition 8 and 7).

The following code snippet shows our formalization of the happening execution (`apply_happ`) and the validity of a happening sequence (`valid_happ_seq`).

Listing 7: Induced Happening Sequence and Plan validity

```
fun apply_happ :: "happening ⇒ world_model ⇒ world_model"
      where
  "apply_happ (t_i, A_i) M =
    (M - ⋃ (set (map (set o dels o effect) A_i))) ∪ ⋃ (set (map
        (set o adds o effect) A_i))"

fun valid_happ_seq :: "world_model ⇒ happening list ⇒
      world_model ⇒ bool" where
  "valid_happ_seq M [] M' ⟷ (M = M')"
| "valid_happ_seq M ((t_i,A_i)#hs) M' ⟷
  (∀a∈ set A_i. M ^c ⊨= precondition a)
  ∧ (∀a ∈ set A_i. ∀b ∈ set A_i. a ≠ b ⟶ acts_non_intrf a b)
  ∧ valid_happ_seq (apply_happ (t_i,A_i) M) hs M'"
```

The function `apply_happ` returns the state after applying the effects of a given happening to a given state. The validity of a happening sequence (Definition 7) is formalized with the function `valid_happ_seq`, which is recursive on the given happening sequence. The state sequence is only constructed implicitly through the recursion. In each recursion step it is checked that (i) the actions within the happening are pairwise non-interfering and (ii) the preconditions of each action in the happening are satisfied by the current state.

The following code snippet shows our formalization of the predicate that characterizes induced happening sequences (Definition 8).

Listing 8: Induced Happening Sequence

```
definition ind_happ_seq :: "plan ⇒ happening list ⇒ bool"
      where
  "ind_happ_seq π s hs ⟷
  (strict_sorted (map fst hs)
  ∧ (∀(t_π,π) ∈ simple_acts π s.
      let g_a = the (res_inst π) in
      ∃A. (t_π,A) ∈ set hs ∧ g_a ∈ set A)
  ∧ (∀(t_π,π) ∈ durative_acts π s.
      let π_start = the (res_inst_snap_action π At_Start);
        π_end = the (res_inst_snap_action π At_End);
        π_inv = the (res_inst_snap_action π Over_All) in
      (∃A. (t_π,A) ∈ set hs ∧ π_start ∈ set A) ∧
      (∃A. (t_π + (duration π),A) ∈ set hs ∧ π_end ∈ set A) ∧
      (∀t_i t_j. (consec_htps π s t_i t_j
  ∧ t_π ≤ t_i ∧ t_j ≤ t_π + (duration π))
      ⟶ (∃(t',A) ∈ set hs.
        t_i < t' ∧ t' < t_j ∧ π_inv ∈ set A)))
  ∧ (∀(t_i,A_i) ∈ set hs. A_i ≠ [] ∧
      (∀a ∈ set A_i. ∃(t_π,π) ∈ set π s.
        inst_of_plan_action π s (t_π,π) (t_i,a))))"
```

An induced happening sequence is characterized by the following constraints: (i) the induced happening sequence is strictly sorted by the time points of each happening, (ii) for each plan action the induced happening sequence contains the correct snap actions (according to Definition 8), and (iii) every action in the induced happening sequence is a grounded and instantiated snap action for a plan action. The predicate `ind_happ_seq` formalizes these constraints and consists of four conjuncts. The first conjunct ensures that an induced happening sequence is strictly sorted by the time points of happenings. The second and third conjuncts specify the placement of the instantiated ground actions for each plan action according to Definition 8. The fourth and final conjunct uses the function `inst_of_plan_action` to ensure that all ground actions in the induced happening sequence are a instantiated snap actions for a plan action.

The following code snippet then shows our formalization of plan validity according to Definition 9.

Listing 9: Plan validity (II)

```
definition plan_happ_path :: "world_model ⇒ plan ⇒ world_model
      ⇒ bool" where
  "plan_happ_path M πs M' ⟷
  (∃hs. ind_happ_seq πs hs ∧ valid_happ_seq M hs M')"

lemma "valid_plan2 πs ≡ wf_plan πs
  ∧ (∃M'. plan_happ_path I πs M' ∧ M' ⌢ ⊨= goal P)"
  unfolding valid_plan2_def valid_plan_from2_def by auto
```

The formalization of the plan validity `valid_plan2` uses an existential quantifier for a valid induced happening sequence.

The next code snippet shows the Lemma that proves the equivalence between the two plan validity definitions (Definition 6 and 9).

Listing 10: Equivalence Proof

```
lemma
  assumes "wf_problem"
  shows "valid_plan πs ⟷ valid_plan2 πs"
  unfolding valid_plan_def valid_plan2_def
  using valid_plan_from2_iff[OF assms] by blast
```

Under the assumptions that the given planning problem and plan are well-formed (`wf_problem`) both formalizations (`valid_plan` and `valid_plan2`) are equivalent.

Formalization of Executable Plan Validator

First, executable refinements for the semantics of ground actions and happenings are implemented.

Listing 11: Enabled-ness Execution of a Happening

```
definition en_exE :: "happening ⇒ world_model ⇒ _+world_model"
      where
  "en_exE ≡ λ(t_i,A_i) ⇒ λs. do {
  check_allm (λa. check (holds s (precondition a))
    (ERRS ''Precondition not satisfied'')) A_i;
  check_pairwise (λa b.
    check (a ≠ b ⟶ acts_non_intrf a b)
      (ERRS ''Actions in happening interfering'')) A_i;
  Error_Monad.return (apply_happ_exec (t_i,A_i) s)}"
```

The function `en_exE` combines the execution of a happening with an enabled-ness check. A happening h is *enabled* in a state if (i) the precondition of every ground action in the happening h is satisfied in the state M, and (ii) no two ground actions in the happening h are interfering. If a given happening h is enabled in a given state M then `en_exE` return the resulting state after applying the effects of the happening h. The function `apply_happ_exec` is an executable refinement of the function `apply_happ`.

The following lemma justifies the refinement for the happening execution.

Listing 12: Justification Enabled-ness Execution of a Happening

```
lemma (in wf_ast_problem) en_exE_return_iff:
  assumes "wm_basic s"
  and "∀a ∈ set A_i. wf_ground_action a"
  shows "en_exE (t_i,A_i) s = Inr s' ⟷ happ_enabled (t_i,A_i) s
        ∧ s' = apply_happ (t_i,A_i) s"
  unfolding en_exE_def
  using assms holds_for_wf_fmlas[OF <wm_basic s>]
  symmetric_pred_check_pairwise[OF acts_non_intrf_symmetric]
  apply_happ_exec_refine
  by auto
```

The assumptions for this lemma are that the world model is *basic* and all ground actions in the happening are well-formed. A *basic* world model only contains predicate atoms. The lemma then states, that the function `en_exE` only returns `Inr s'` if the given happening `(t_i,A_i)` is enabled in the state `s` and the application of the effects yield the world model `s'`. Otherwise the function will return `Inl msg` with an error message `msg`.

The validity of a happening sequence (Definition 7) is implemented recursively and combined with the enabled-ness execution for happenings. Moreover, the entailment of the goal state specifications is directly checked in the base case of the recursion.

Listing 13: Validity of a Happening Sequence

```
fun valid_happ_seq_fromE :: "nat ⇒ world_model ⇒ happening
      list ⇒ _+unit" where
  "valid_happ_seq_fromE si s [] =
  check (holds s (goal P))
    (ERRS ''Postcondition does not hold'')"
| "valid_happ_seq_fromE si s (h#hs) = do {
  s ← en_exE h s <+? (λe _. shows ''at step '' o shows si o
        shows '': '' o e ());
  valid_happ_seq_fromE (si+1) s hs}"
```

The argument `si` is the index of the current execution step. The implementation is justified with the following lemma.

Listing 14: Justification for Validity of a Happening Sequence

```
lemma (in wf_ast_problem)
  assumes "wm_basic M" and "wf_happ_seq hs"
  shows "valid_happ_seq_fromE k M hs = Inr () ⟷
  (∃ M'. valid_happ_seq M hs M' ∧ M' ⌢ ⊨= (goal P))"
  using assms valid_happ_seq_from_refine
        valid_happ_seq_fromE_return_iff by auto
```

The function `valid_happ_seq_fromE` only returns `Inr ()` if there exists a state `M'`, such that the given happening sequence `hs` is valid from the starting state `M` to the state `M'` and `M'` entails the goal state specifications.

Next the executable refinements for the semantics of temporal plan validity are described.

The following code snippet shows the implementation of the function `simplify_plan`, which is an executable function that produces an induced happening sequence.

Listing 15: Executable Construction of an Induced Happening Sequence

```
fun insort_happ :: "happening ⇒ happening list ⇒ happening
      list" where
  "insort_happ (t_i,A_i) [] = [(t_i,A_i)]"
```

```
  | "insort_happ (t_i,A_i) ((t_j,A_j)#hs) =
    (if t_i < t_j then (t_i,A_i)#(t_j,A_j)#hs
    else if t_i = t_j then (t_j,A_i @ A_j)#hs
    else (t_j,A_j)#(insort_happ (t_i,A_i) hs))"

fun insort_mult_happs :: "happening list ⇒ happening list ⇒
     happening list" where
  "insort_mult_happs [] hs_2 = hs_2"
| "insort_mult_happs (h#hs_1) hs_2 = insort_happ h (
     insort_mult_happs hs_1 hs_2)"

definition consec_htps_in_interval :: "time list ⇒ time ⇒ time
     ⇒ (time × time) list" where
  "consec_htps_in_interval htps t_i t_j =
  filter (λ(t,t'). t_i ≤ t ∧ t' ≤ t_j) (zip (butlast htps) (tl
       htps))"

fun simplify_action :: "time list ⇒ (time × plan_action) ⇒ (
     time × ground_action) list" where
  "simplify_action htps (t,Simple_Plan_Action n args) = (
  let a = the (resolve_action_schema n) in
    [(t, instantiate_action_schema a args)])"
| "simplify_action htps (t,Durative_Plan_Action n args d) = (
  let a = the (resolve_action_schema n) in
  (t,inst_snap_action a args At_Start) #
  (t+d,inst_snap_action a args At_End) #
  (map (λ(t_i,t_j).((t_i+t_j) / 2,inst_snap_action a args
       Over_All)) (consec_htps_in_interval htps t (t+d))))"

fun simplify_plan :: "time list ⇒ plan ⇒ happening list" where
  "simplify_plan htps [] = []"
| "simplify_plan htps (π#πs) =
  insort_mult_happs (map (λ(t_a,a). (t_a,[a])) (simplify_action
       htps π)) (simplify_plan htps πs)"
```

The function `simplify_plan` takes two arguments: a sequence of happening time points and a plan. The sequence of happening time points is needed to place the invariant snap actions. The invariant snap actions are placed at $\frac{t_i+t_{i+1}}{2}$ for two consecutive happening time points t_i and t_{i+1}. The function `simplify_plan` is recursive on the given plan. In each recursion step alls snap actions for the current plan action (according to Definition 8) are inserted into the existing happening sequence with the function `insort_mult_happs`. The function `insort_mult_happs` uses the function `insort_happ` to inserts multiple happenings into a existing happening sequence. The function `simplify_action` return a list of all snap actions for a given plan action (according to Definition 8). The function `consec_htps_in_interval` returns all intervals of consecutive happening time points that lie in between two given time points.

The following lemma proves the correctness of the function `simplify_plan`.

Listing 16: Correctnes of the Construction of an Induced Happening Sequence

```
lemma (in wf_ast_problem)
  assumes "wf_plan πs"
  shows "ind_happ_seq πs (simplify_plan (htps_exec πs) πs)"
  using assms by (rule simplify_plan_correct)
```

The lemma proves, that given a well-formed plan (`wf_plan πs`) the function `simplify_plan` constructs an induced happening sequence for the given plan `πs`. The function `htps_exec` constructs the sequence of happening time points for a given plan.

Efficiency Refinement of Executable Plan Validator

The following code snippet shows the more efficient refinement of the function `simplify_plan`, that uses an AVL-tree instead of a list.

Listing 17: Efficient Construction of the Induced Happening Sequence with an AVL-tree

```
fun insert_happ_to_tree :: "happening ⇒ happening tree_ht ⇒
     happening tree_ht" where
  "insert_happ_to_tree (t_i,A_i) ⟨⟩ = avl_node ⟨⟩ (t_i,A_i) ⟨⟩"
| "insert_happ_to_tree (t_i,A_i) ⟨l,((t_j,A_j),h),r⟩ = (
  if t_i < t_j then avl_balL (insert_happ_to_tree (t_i,A_i) l) (t
       _j,A_j) r
  else if t_i = t_j then avl_node l (t_j,A_i @ A_j) r
  else avl_balR l (t_j,A_j) (insert_happ_to_tree (t_i,A_i) r))"

fun insert_timed_ground_acts_to_tree :: "(time × ground_action
     ) list ⇒ happening tree_ht ⇒ happening tree_ht" where
  "insert_timed_ground_acts_to_tree [] htree = htree"
| "insert_timed_ground_acts_to_tree ((t_a,a)#as) htree =
  insert_happ_to_tree (t_a,[a]) (
       insert_timed_ground_acts_to_tree as htree)"

fun simplify_planE_avl' :: "_ ⇒ (object, type) mapping ⇒ (
     object, rat) mapping ⇒ time list ⇒ plan ⇒ _+happening
     tree_ht" where
  "simplify_planE_avl' stg mp mp_fe htps [] = do {
       Error_Monad.return ⟨⟩ }"
| "simplify_planE_avl' stg mp mp_fe htps (π#πs) = do {
  as ← simplify_actionE G mp mp_fe htps π;
  htree ← simplify_planE_avl' G mp mp_fe htps πs;
  Error_Monad.return (insert_timed_ground_acts_to_tree as
       htree)}"

fun simplify_planE_avl :: "_ ⇒ (object, type) mapping ⇒ (
     object, rat) mapping ⇒ time list ⇒ plan ⇒ _+happening
     list" where
  "simplify_planE_avl stg mp mp_fe htps πs = do {
  htree ← simplify_planE_avl' G mp mp_fe htps πs;
  Error_Monad.return (avl_inorder htree)}"
```

The function `simplify_planE_avl'` constructs an AVL-tree containing all happenings for the constructed induced happening sequence. The function `simplify_planE_avl` then simply uses an inorder-traversal (`avl_inorder`) on the AVL-tree to obtain the induced happening sequence.

The argument `stg` is the instantiated subtype relation for an implicitly fixed domain and the argument `mp` is a map from object names to types. For a given plan action the function `simplify_actionE` returns all snap actions and checks the well-formedness of the given plan action. The implementations for the efficient type checking and its verification are reused from (Abdulaziz and Lammich 2018) without any modifications. The functions `insert_timed_ground_acts_to_tree` and `insert_happ_to_tree` are used to insert elements to the AVL tree.

Listing 18: Correctnes of the Construction of the Induced Happening Sequence

```
lemma (in wf_ast_problem)
  assumes "wf_domain"
  and "simplify_planE_avl STG mp_objT mp_Fevl (htps_exec πs)
       πs = Inr hs"
  shows "ind_happ_seq πs hs" and "wf_plan πs"
  using assms simplify_planE_avl_equiv
       simplify_planE_return_iff
  htps_exec_sorted htps_exec_distinct simplify_plan_correct
  by (auto simp: strict_sorted_iff)
```

This lemma proves that if `simplify_planE_avl` returns `Inr hs` then the returned happening sequence `hs` is an induced happening sequence and the plan `πs` is well-formed.

Finally, the construction of the induced happening sequence (`simplify_planE_avl`) and the implementation for the validity a happening sequence (`valid_happ_seq_fromE`) are combined in the function `check_plan`, which gives us the desired validator.

Listing 19: Implementation of Validtor function

```
definition "check_plan P πs ≡ do {
  let stg = ast_domain.STG (ast_problem.domain P);
  let conT = ast_domain.mp_constT (ast_problem.domain P);
  let mp = ast_problem.mp_objT P;
  let mp_fe = ast_problem.mp_Fevl P;
  check_wf_problem P stg conT mp;
  hs ← ast_problem.simplify_planE_avl stg mp mp_fe (htps_exec
       πs) πs;
  ast_problem.valid_happ_seq_fromE 1 (ast_problem.I P) hs
} <+? (λe. String.implode (e () ''''))"
```

The following theorem proves our validator correct.

Listing 20: Correctness of our Validator

```
theorem "check_plan P πs = Inr () ⟷ ast_problem.wf_problem
     P ∧ ast_problem.valid_plan P πs"
  by (rule check_plan_return_iff)
```

Our validator `check_plan` only returns `Inr ()` if the problem is well-formed and the plan is valid.