A Formally Verified Validator for Classical Planning Problems and Solutions

Mohammad Abdulaziz Peter Lammich
Technical University of Munich

Abstract—In this paper we present a formally verified validator for planning problems and their solutions. We formalise the semantics of a fragment of PDDL (\(\land, \neg, \rightarrow, =\) in the preconditions, typing and constants) in the Higher-Order Logic theorem prover Isabelle/HOL. We then construct an efficient plan validator and mechanically prove it correct w.r.t. our semantics. We argue that our approach provides a superior compromise in constructing validators where one can have the best of two worlds: (i) clear and concise semantics w.r.t. which the validator is built thus helping to avoid bugs (unlike existing validators, which we show have bugs) and (ii) an optimised implementation whose performance is competitive with mainstream unverified validators.

I. INTRODUCTION

Since their earliest days, AI planning systems have had a lot of interaction with theorem provers. First-Order Logic (FOL) theorem provers were used in [1], [2], and, more recently, propositional satisfiability solvers in [3], [4]. However, most of that work was focused on using theorem provers to solve planning problems, i.e. to produce plans.

In this work we argue that theorem proving technology can excel in filling needs for the planning community, other than producing plans. One obvious such need is validating whether a plan actually solves a planning problem. Since in many cases large transition systems underlie planning problems, and (candidate) plans can be substantially long, manual validation is infeasible. Furthermore, the need for validation is exacerbated if the application in which those plans are used is safety-critical. This motivates the need for automatic plan validators that verify whether a sequence of actions indeed achieves the goal of a planning problem.

This issue is addressed by validators for different planning formalisms, most notably VAL [5], which validates plans for problems specified in PDDL2.1 [6]. It was later extended [7] to cover processes and events. However, VAL’s C++ implementation uses many performance oriented optimisations that sometimes obfuscate the relation between the semantics of PDDL and the actual implementation, which might cause the implemented semantics to not be equivalent to the intended semantics. One attempt to remedy this is Patrik Haslum’s INV AL validator for PDDL. It is written in Common Lisp, with the intent of closely resembling the semantics of PDDL [6], [8]. This, however, comes at the cost of performance: in our experiments INV AL is substantially slower than VAL. Moreover, although Common Lisp may be more concise than C++ in describing the semantics of PDDL, it is still a programming language. Thus, it requires the semantics to be specified as executable programs instead of the mathematically more concise and abstract specifications which are used in pen-and-paper specification. Again, this has the potential of introducing differences between the implemented and the intended semantics.

Indeed, there are bugs in both VAL and INV AL. There are domains, instances and buggy plans that are mistakenly accepted as correct plans by VAL. Furthermore, there are domains, instances and plans that are all correct, but trigger non-termination for INV AL and segmentation faults for VAL, thus rendering them practically incomplete.²

In this work we argue that theorem provers and formal methods technology are an excellent fit to alleviate: (i) having to choose between clear semantics and efficient implementations, and (ii) having to trust the equivalence of the implemented semantics and the pen-and-paper semantics. Languages used by theorem provers, like Higher-Order Logic (HOL) [9], [10] and Dependent Type Theory [11], [12], allow for the use of standard mathematical constructs that are not necessarily executable, like set comprehensions. Thus, they enable a formal specification of the semantics that is far more concise and elegant than what is possible in any programming language (see Table I). This reduces the chance of having bugs in the specified semantics. More importantly, theorem provers allow for proving that sanity checking properties hold for the specified semantics. It also allows mechanising soundness theorems associated with pen-and-paper semantics. Having those properties mechanised is substantially more reliable than mere visual inspection, and thus gives a much stronger indication on the equivalence between the formally specified and the pen-and-paper semantics.

Another advantage of using theorem provers for building validators is that the implementation of the validator needs not be the same as the specified semantics. Indeed, an optimised

1https://github.com/patrikhaslum/INV AL

2We describe the bugs we found in more detail in the experiments section.

<table>
<thead>
<tr>
<th>Semantic Definition / Size</th>
<th>VAL</th>
<th>INV AL</th>
<th>Isabelle/HOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply Action</td>
<td>75</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>Action Enabled in State</td>
<td>67</td>
<td>69</td>
<td>15</td>
</tr>
<tr>
<td>Well-Formed Action Instance</td>
<td>87</td>
<td>77</td>
<td>25</td>
</tr>
<tr>
<td>Plan is Valid</td>
<td>224</td>
<td>163</td>
<td>50</td>
</tr>
</tbody>
</table>

Table I: Sizes in lines of code (LOC) of the specification of different definitions. This table is only indicative, since different validators support different features. However, whenever we could, we only count LOC contributing to the STRIPS fragment in VAL and INV AL. E.g. in INV AL, we count no lines that apply relative or assign effects.
implementation of the validator with low level performance oriented trickery can be used. The equivalence between the implementation and the specification is formally proved within the theorem prover. This allows for the runtimes of our validator to compete with VAL, and be much faster than INVAL, despite our semantics specification in HOL being substantially more concise than that of INVAL, let alone VAL.

Contributions: (i) (Section II) In Isabelle/HOL, we formalise the semantics of a propositional STRIPS like formalism for ground actions, but we add to it equality, negation, disjunctions and implication in the preconditions. We roughly follow the semantics of STRIPS by Lifschitz [13], adapting it to the extra features that we support and removing from it practically irrelevant features like non-atomic effects. (ii) (Section III) We define the semantics of a fragment of PDDL (i.e. the notion of action schemata and typing) on top of our formalisation of ground actions. Our fragment includes the PDDL flags :strips, :typing, :negative-preconditions, :disjunctive-preconditions, :equality, :constants, and :action-costs. (iii) (Section IV) We prove theorems about the formalised semantics that serve as sanity checks. (iv) (Section V) We specify an optimised validator in Isabelle/HOL, and prove it correct w.r.t. our formalised semantics. Using Isabelle’s code generator [14], [15], we extract an executable validator as Standard ML [16] program. It is much faster than INVAL and competitive with VAL.

To the best of our knowledge, this is the first formally verified plan validator. The Isabelle sources of our development can be downloaded at: http://home.in.tum.de/~mansour/verval.html.

A. Isabelle/HOL

Isabelle/HOL [10] is a theorem prover based on Higher-Order Logic. Roughly speaking, Higher-Order Logic can be seen as a combination of functional programming with logic. Isabelle/HOL supports the extraction of the functional fragment to actual code in various languages [14].

Isabelle’s syntax is a variation of Standard ML combined with (almost) standard mathematical notation. Function application is written infix, and functions are usually Curried (i.e., function $f$ applied to arguments $x_1 \ldots x_n$ is written as $f \hspace{1pt} x_1 \ldots x_n$ instead of the standard notation $f(x_1, \ldots, x_n)$). We explain non-standard syntax in the paper where it occurs.

Isabelle is designed for trustworthiness: following the LCF approach [17], 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.

II. Mechanising the Semantics of Ground Actions

In this section, we formalise a semantics of ground actions, i.e. STRIPS extended with disjunction, negation, and equality in the preconditions and goals. We follow the definitions by Lifschitz [13], whenever possible. In Section III, we use this formalisation as a basis for our PDDL semantics.

A. Formalising the Abstract Syntax

First, we define basic concepts like atoms, predicates, and formulae. These concepts are also used for defining PDDL semantics, so we follow Kovacs’ [18] PDDL grammar where appropriate. In Isabelle/HOL, Kovacs’ abstract syntax can elegantly be modelled as algebraic data types. E.g. consider the following abstract syntax rules from Kovacs’.

Those rules are modelled as follows in Isabelle/HOL as follows.

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type_synonym</code></td>
<td><code>name = string</code></td>
</tr>
<tr>
<td><code>datatype</code></td>
<td><code>predicate = Pred (name: name)</code></td>
</tr>
<tr>
<td><code>datatype</code></td>
<td><code>PredicateAtom = Predicate (atom: Predicate)</code></td>
</tr>
<tr>
<td><code>definition</code></td>
<td><code>Eq : (PredicateAtom) × (PredicateAtom) ⇒ bool</code></td>
</tr>
</tbody>
</table>

That is, our abstract syntax does not detail the structure of names, but models them as strings. For predicates, we use a datatype with a single constructor `Pred`, which contains a `name`. The Isabelle notation `name` defines a selector function, i.e. `name` is the name of predicate `p`. For atomic formulae, which we call atoms here, we use a datatype with two constructors, one per rule in the grammar. Moreover, the parameter `t` in Kovacs’ grammar is modelled as a type parameter `t`. E.g. `object_atom` is the type of ground atoms, and `term_atom` is the type of terms over terms.

To model formulae, we use an existing Isabelle/HOL formalisation of propositional logic by Michael and Nipkow [19]. Formulae are parameterised over atoms, and consist of the standard connectives $\wedge$, $\vee$, $\rightarrow$, $\neg$ and $\bot$ (i.e. falsum). For example, the type `object_atom_formula` represents ground formulae.

An effect is a pair of sets of formulae to be added and deleted. We restrict the formulae to predicate atoms only, since non-atomic effects are practically irrelevant.

B. The Semantics of STRIPS + Negation and Equality

We define a world model to be a set of ground formulae.

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type_synonym</code></td>
<td><code>world_model = &quot;object_atom_formula_set&quot;</code></td>
</tr>
</tbody>
</table>

States resulting from action execution are “basic” world models containing only predicate atoms instead of arbitrary formulae.

| Definition   | `"wm_basic M ≡ ∀a ∈ M. is_predAtom a"` |

To define entailment, the basic world model is closed by adding the negations of all predicates not in the basic model, as well as all equalities and inequalities:

| Definition   | `"close_world :: "world_model ⇒ world_model" where "close_world M =` |
A domain consists of a list of type declarations, which are pairs of declared types and supertypes, as well as predicate declarations, constant declarations, and action schemas. A problem refers to a domain, and additionally contains object declarations, as well as an initial state and a goal. A well-formedness condition will restrict the formulae of the initial state to be only predicate atoms\(^3\).

**B. Well-Formedness**

The next step towards specification of a PDDL semantics is to define well-formedness criteria. For example, we require that action names are unique and that all predicates occurring in formulæ or effects have been declared and are applied to type-compatible terms.

While many of the well-formedness conditions are straightforward, some leave room for ambiguities that have to be resolved in a formal specification. For example, Kovacs’ grammar formally allows for types being declared with an Either-supertype. The semantics of this is not clear; different validators seem to have different interpretations of what this means, and thus, we do not support this feature. In our semantics of typing each type declaration \((tn – tn')\) introduces an edge \(tn \leftarrow tn'\) in a subtype relation, where \(tn\) and \(tn'\) are type names. A term of (Either-) type \(oT\) can be substituted at a parameter position expecting type \(T\) iff every type name in \(oT\) is reachable from some type name in \(T\) via the following subtype relation.

definition of_type :: "type ⇒ type ⇒ bool" where
"of_type oT T ≡ (set (primitives T) ⊆ subtype_rel " set (primitives T))"

Here, \((*)\) denotes reflexive transitive closure, and \((\ast)\) is the image of a set under a relation.

**C. Execution Semantics**

Finally, we define the semantics of plans within our fragment of PDDL: by instantiating the action schema referenced by a plan action we obtain a ground action, which is then applied to the world model using the execution semantics defined in Section II. Additionally, we check that each plan action is well-formed, i.e. that its arguments’ types can be substituted for the action schema’s parameter types. Note that we assume an implicitly fixed PDDL instance \(P\) with domain \(D\) for the following definitions.

definition plan_action_path :: "world_model ⇒ plan_action list ⇒ world_model ⇒ bool" where
"plan_action_path P πs M' ≡ ((∀π ∈ set πs. wf_plan_action π)
∧ ground_action_path P (map Resolve_Instanciate πs) M')"

In this definition, we first ensure that every plan action is well-formed. Then, we resolve and instantiate the actions of the plan, obtaining a sequence of ground actions, which is checked against the basic semantics for ground actions. With the above definition, it is straightforward to define a valid plan as a plan that transforms the initial world model \(I\) to a world model \(M'\) that entails the problem’s goal:

definition valid_plan :: "plan ⇒ bool" where
"valid_plan πs M ≡ (∃M'. plan_action_path πs M' ∧ M' \models (goal P)"

Kovacs grammar also allows negated atoms and equalities here, which, however, is meaningless under our closed world assumption.

\(^3\)We sometimes use slightly different names than Kovacs.
IV. PROVING PROPERTIES ABOUT THE SEMANTICS:
SANITY CHECKING THEOREMS

Traditionally, validators specify semantics (usually in the
form of an implementation) and trust that the specified sem-
antics are what they are supposed to mean at face value;
the only check one could do is visual inspection. Here, we
demonstrate that specifying semantics in a theorem prover
allows proving sanity checking theorems within the theorem
prover about the specified semantics, instead of mere visual
inspection. Admittedly, the theorem statements are still trusted
only visually, but it is established as a plausible premise that
proving such sanity checking theorems reduces the chance of
bugs remaining unnoticed, e.g. Norrish’s seminal formalisation
of the C-language’s semantics [20].

One sanity checking theorem we prove in Isabelle/HOL
concerns the soundness of closing the world-model. It states
that for a formula without negation, implication, nor equality,
our entailment is the same as entailment without closing the
model. This shows that our ground action semantics generalise
standard STRIPS without negation and equality.

Another sanity check shows that executing a well-typed plan
preserves well-formedness of the world model. Here, a world
model is well-formed if it is basic, and all its predicates are
declared and applied to type-compatible declared objects. This
is a sanity check for our well-formedness conditions: it proves
that they are strong enough to ensure nothing odd will happen
to the world model during execution.

A more involved sanity check confirms that our execution
semantics of ground actions are sound, in the sense used by
Lifschitz [13]. Lifschitz introduced the notions of abstract
states and abstract actions, and derived conditions on ground
actions and world models allowing their execution to be
simulated by executing abstract actions on abstract states.
Since the execution semantics of abstract actions are an
alternative formulation of the execution semantics of ground
actions, then showing a simulation between the two semantics
is a form of validation reducing the possibility of having
bugs. We model abstract states in Isabelle/HOL as 
valuations, i.e. functions from atoms to truth values. While Lifschitz
assumed that the concept of an abstract state satisfying a
formula is readily defined, we formalise that concept using the
syntactic entailment operator defined in the logic of Michael
and Nipkow [19], in which they use \( \models \) to denote that a
valuation \( \langle s \rangle \) entails a formula \( \varphi \). However, we extend
the entailment operator of their logic to include equalities and
denote this new entailment by \( \models_{=} \). Lifschitz, and we,
define abstract actions as partial functions from abstract states

To demonstrate the simulation, we define mappings from
world-models to abstract states and from ground actions to
abstract actions. A world-model \( \mathcal{M} \) is mapped to an abstract
state \( \langle s \rangle \) if \( \forall \varphi \in \text{close-world } \mathcal{M}, s \models_{=} \varphi \). Let \( \alpha \) be a ground
action with precondition \( \text{pre} \) and effects \( \text{add} \) and \( \text{del} \). \( \alpha \)
is mapped to an abstract action \( f \) if for any abstract state \( s \),
\( (s) \models_{=} \text{pre} \) implies that there is an \( s' \) s.t. \( f(s) = s' \) (i.e.
the abstract action is well-defined on \( s \) ), and that for any
atom \( \langle \text{atm} \rangle \) (i) if \( \langle \text{atm} \rangle \notin \text{del} \) and \( (s) \models_{=} \text{atm} \), then \( (s') \models_{=} \text{atm} \), and (ii) if \( \langle \text{atm} \rangle \notin \text{add} \) and \( (s) \models_{=} \neg \text{atm} \), then \( (s') \models_{=} \neg \text{atm} \), and (iii) if \( (\text{atm} \in \text{add}) \), then \( (s') \models_{=} \text{atm} \wedge w \),
and (iv) if \( \langle \text{atm} \in \text{del} \rangle \) and \( \langle \text{atm} \notin \text{add} \rangle \), then \( (s') \models_{=} \neg \text{atm} \). Those two mappings are analogous to the mappings
by Lifschitz. However, since, unlike Lifschitz, we support
negations, implications and equalities in the precondition we
use an entailment operator with equality, and we add the
articles ii and iii to the action mapping. Also since we restrict
ground action effects to atoms, we omit the condition stating
that every non-atomic effect in \( \langle \text{add} \rangle \) is a tautology from
the action mapping. Using our mappings we show the required
simulation between ground actions/world models and abstract
actions/states as a theorem in Isabelle/HOL equivalent to the
theorem in Section 4 in Lifschitz's paper.

As a last sanity check, we show that our semantics of types
and action schema instantiation do not affect soundness, i.e.
we show that a well-formed instantiation of a well-formed
domain can be simulated by abstract states and actions. A
minor challenge is devising concrete mappings from PDDL
world models to abstract states and the other way around, as
well as as from PDDL plan actions to abstract actions.

V. A FORMALLY VERIFIED PLAN VALIDATOR

Using the Semantics of the fragment of PDDL defined in
Section III, we concisely write down a PDDL plan validator.
Given a problem \( P \) and plan \( \pi \), the validator checks that
the problem is well-formed and the plan is valid:
\[
\text{validator } P \pi \equiv \text{wf_problem } P \land \text{valid_plan } P \pi
\]

However, this specification depends on several abstract
mathematical concepts that we used in our semantics, e.g. sets
and reflexive transitive closures. While these are well-suited
for an elegant specification, they cannot be directly executed.
Thus, we prove that our abstract specification is equivalent to a
concrete, algorithmic one, for which Isabelle/HOL can extract
actual code. This refinement approach gives us the best of
two worlds: a concise semantics and an efficient checker that
provably implements the semantics.

Our refinement proceeds in two steps. First, we replace
abstract specifications by algorithms defined on abstract math-
ematical types like sets. Second, we replace the abstract types
by concrete (verified) data structures like red-black trees. The
second step is automated by the Containers Framework [15].
We exemplify some of the refinements below.

For reflexive transitive closure, we use a (general purpose)
DFS algorithm. We define
\[
\text{definition of_type_impl } G \circ T \\
≡ (\forall pt \in \text{set primitives } oT). \text{dfs_reachable } G (\text{op}=pt) (\text{primitives } T)
\]

Here, the \( G \) parameter will be instantiated by the tabulated
successor function of the subtype graph, which is only com-
puted once and passed as extra parameter to all subsequent
functions (\( \lambda \)-lifting). We prove
\[
\text{lemma of_type_impl_correct: of_type_impl STG oT } T \leftrightarrow \text{of_type } oT \ T
\]

Above, \( \text{STG} \) is the actual tabulated subtype graph defined
w.r.t. the implicitly fixed domain \( D \).

Another crucial refinement summarises the enabledness
check and effect application of a plan action. This way, the
action has to be instantiated only once. Moreover, we use an error monad to report human-readable error messages:

```markdown
definition en_exE2 where
  en_exE2 G mp \equiv (\lambda (PAction n args) \Rightarrow \lambda M. do {
    a ← resolve_action_schemaE n;
    check (action_params_match2 G mp a args) (ERRS "Par. mism.");
    let ai = instantiate_action_schema a args;
    check (holds M (precondition ai)) (ERRS "Precondition not satisfied");
    Error_Monad.return (apply_effect (effect ai) M)};
```

Here, `(G)` will again be instantiated by the subtype relation, and `(mp)` will be instantiated by a map from object names to types. The following lemma justifies the refinement:

```markdown
lemma (in wf_ast_problem) en_exE2_return_iff:
  assumes wm_basic M
  shows en_exE2 STG mp_objT a M = Inr M'
  \iff plan_action_enabled a M \land M' = execute_plan_action a M
```

That is, for a basic world model `(M)` and plan action `(a)`, the function `en_exE2 STG mp_objT a M` will return `Inr M'`, if and only if the action is enabled and its execution yields `M'`. Otherwise, it returns `Inl msg` with a human readable error message. Again, `(STG)` is the actual subtype relation, and `(mp_objT)` is the actual mapping from objects to types.

Finally, we obtain a refined checker `(check_plan)` and prove the following theorem that it is sound w.r.t. the semantics.

```markdown
lemma check_plan_return_iff:
  check_plan P ns = Inr () \iff wf_problem P \land valid_plan P ns
```

For a problem `P` and an action sequence `ns`, our executable plan checker `check_plan` returns `Inr ()` if and only if the problem and its domain are well-formed, and the plan is valid.\(^5\) Otherwise, it returns `Inl msg` with some error message.

During the refinement steps, we prove correct many algorithms and data structures. However, these details are irrelevant for understanding the final correctness theorem: if one believes that the right hand side of the equivalence, which only depends on the abstract semantics, correctly specifies a well-formed problem and valid plan, and believes in the soundness of Isabelle, then this theorem states that the left hand side (the checker) is a plan validator.

Isabelle/HOL’s code generator can generate implementations of computable functions in different languages. We generate a Standard ML [16] implementation of `check_plan` and combine it with a parser and a command line interface (CLI) to obtain an executable plan validator. Note that the parser and CLI are trusted parts of the validator, i.e. there is no formal correctness proof that the parser actually recognises the desired grammar and produces the correct abstract syntax tree, nor that the CLI correctly forwards the arguments to `check_plan` and correctly displays the result.

### A. Empirical Evaluation

Our validator currently supports the PDDL flags `:strips`, `:typing`, `:negative-preconditions`, `:disjunctive-preconditions`, `:equality`, `:constants`, and `:action-costs`, which are enough to validate the vast majority of the International Planning Competition benchmarks and their solutions. We use our validator, in addition to VAL and INVAL, to validate PDDL problems and plans from previous International Planning Competitions (IPC). We validate plans generated by Fast-Downward [21].

Table II shows the runtimes of our validator, VAL, and INVAL for different IPC benchmark domains, and for each domain shows how many plans were labelled as valid. There are two main goals for this experiment. First, we investigate whether there are differences in the validation outcomes of VAL, INVAL, and our validator, and whether such differences are due to bugs. For the IPC benchmarks, observed differences were primarily due to segmentation faults by VAL, which happened with 111 instances. Another difference is that both VAL and INVAL allow an empty list of object declarations of the form `(- type)` in the problem, while our validator expects at least one object, which conforms to Kovacs’ grammar. This showed up in a problem from the WoodWorking domain.

However, for non-IPC benchmarks, we found bugs in both VAL and INVAL that did not show up in the IPC benchmarks: VAL erroneously identifies distinct atoms during its precondition check. Consider a domain with a predicate `P` and an instance of that domain with objects `OA`, `OB`, `O`, and `AOB`. An action with precondition `P O AOB` will be satisfied by an atom `(P O AO B)` in the state. Based on that, we construct examples where VAL would report an incorrect plan to be valid. Another issue with VAL is that it sometimes reports a valid plan to be invalid, if it has an action with no preconditions. INVAL does not terminate for domains with cyclic type dependencies if its type-checker is enabled.

The fact that these bugs are not detected via testing on the IPC benchmarks strengthens the argument for the use of formal verification to develop AI planning tools, especially if the main purpose of those tools is to add confidence in the correctness of plans and planning systems, as is the case with validators.\(^6\)

The second purpose of our experiments is comparing the performance. Table II shows that our validator is much faster than INVAL on all benchmarked domains. The runtimes of our validator are even comparable to VAL. This, however, is rather impressive when compared to other pieces of verified software. For example, the verified model-checker in [22] is about 400 times slower than the unverified Spin model-checker [23]. This performance is the result of several rounds of profiling to identify hotspots and performance leaks, and adjusting the refinements (and their correctness proofs) accordingly.

### VI. RELATED WORK AND CONCLUSIONS

In this work we provide the first formalisation, to our knowledge, of STRIPS or any of its extensions in any theorem prover. In the theorem prover Isabelle/HOL, we formalise an extended version of STRIPS that allows for negations, implications and equalities in the preconditions and on top of that we formalise the semantics of a fragment of PDDL. To our knowledge, the closest work to that in the planning literature is McCarthy’s formalisation of STRIPS in Situation Calculus [24]. Using stepwise refinement techniques, we created a competitive plan validator, for a PDDL fragment containing most IPC problems, which is mechanically proved

\(^5\)Note: the problem `P` is now an explicit parameter.

\(^6\)We note that all of these bugs were reported to the relevant bug trackers.
Table II: A table showing the maximum and minimum runtimes of different validators on instances in different IPC domains and the number of plans labelled as valid. The column headed by * is the one for our validator.

correct w.r.t. the abstract PDDL semantics. To the best of our knowledge, this is the first formally verified plan validator.

Stepwise refinement approaches were successfully used in a variety of applications, e.g. the verified C compiler CompCert [25], the verified kernel Sel4 [26], and a verified theorem prover [27]. Applications closer to the work of this include a verified SAT solver [28], a verified certificate checker for SAT [29], and a verified model-checker [30].

A logical extension of this paper would be to include more PDDL features like axioms [8] and durative actions [6]. However, the lack of consensus on their semantics in the planning community poses a challenge to formalising them. Other applications of our semantics include proving the equi-solvability of planning problems, or that certain decompositions hold for planning problems, or that certain decompositions hold for temporal planning domains, 

Acknowledgements

The first author is supported by the DFG Koselleck Grant NI 491/16-1.

References