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 ($\vee, \neg, \rightarrow, =$ 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.**

Semantic Definition / Size	VAL	INVAL	Isabelle/HOL
Apply Action	75	23	3
Action Enabled in State	67	69	15
Well-Formed Action Instance	87	77	25
Plan is Valid	224	163	50

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 INVAL. E.g. in INVAL, we count no lines that apply relative or assign effects.

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 INVAL[1] 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 INVAL 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 INVAL. 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 INVAL 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 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

[1]https://github.com/patrikhaslum/INVAL

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

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 $x_1\ \dots\ x_n$ is written as $f\ x_1\ \dots\ x_n$ instead of the standard notation $f(x_1,\ \dots\ ,x_n)$). 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 Lifschtiz [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'.

```
<name> ::= <letter> <any char>*
<predicate> ::= <name>
<atomic formula (t)> ::= (<predicate> t*)
<atomic formula (t)> ::= (= t t)
```

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

```
type_synonym name = string
datatype predicate = Pred (name: name)
datatype 't atom = predAtm (predicate: predicate) (arguments: "'t list")
                 | Eq (lhs: 't) (rhs: 't)
```

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 p› 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 atoms 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 $\perp$ (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.

```
datatype 't effect = Effect
  (adds: "'t atom formula list")
  (dels: "'t atom formula list")
```

The restriction on the add and delete formulae to predicate atoms is modelled by a well-formedness condition. A ground action (a.k.a. operator) consists of a precondition and an effect.

```
datatype ground_action = Ground_Action
  (precondition: "object atom formula")
  (effect: "object ast_effect")
```

B. The Semantics of STRIPS + Negation and Equality

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

```
type_synonym world_model = "object atom formula set"
```

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 =
    M ∪ {¬(Atom (predAtm p as)) | p as. Atom (predAtm p as) ∉ M}
    ∪ {Atom (Eq a a) | a. True} ∪ {¬(Atom (Eq a b)) | a b. a≠b}"
```

We write ⟨M $^c\models_=$ φ⟩ for ⟨close_world M $\models$ φ⟩, where $\models$ is propositional logic entailment from Michael and Nipkow [19].

An effect is applied to a basic world model by first removing the delete-predicates, and then adding the add-predicates.

```
fun apply_effect :: "object ast_effect ⇒ world_model ⇒ world_model"
  where "apply_effect (Effect a d) s = (s - set d) ∪ (set a)"
```

Note that **fun** in Isabelle/HOL allows pattern matching and (terminating) recursive definitions.

A valid ground action sequence $\alpha_1 \ldots \alpha_n$ connecting an initial world model $M = M_1$ and a world model $M_{n+1} = M'$ is a sequence of actions such that there are intermediate models M_i, each M_i entails the precondition of α_i (the action is enabled), and M_{i+1} is obtained from M_i by applying the effect of α_i. In Isabelle/HOL, this is most elegantly modelled as a recursive function, leaving implicit the intermediate models:

```
fun ground_action_path
    :: "world_model ⇒ ground_action list ⇒ world_model ⇒ bool"
  where
    "ground_action_path M [] M' ⟷ (M = M')"
  | "ground_action_path M (α#αs) M' ⟷ M ᶜ⊫₌ precondition α
    ∧ (ground_action_path (apply_effect (effect α) M) αs M')"
```

III. Formalised Semantics of a PDDL Fragment

We now specify semantics of (a fragment of) PDDL, by adding action schemata and types to the semantics of ground actions. An action schema (sometimes simply called action) is parameterised over object-valued variables. Substituting these variables for actual objects yields a ground action. Moreover, a hierarchical type system restricts the applicability of predicates to objects. We first formalise the abstract syntax of PDDL following Kovacs' grammar, and then define its semantics.

A. Abstract Syntax

A term is an object or a variable.

```
datatype "term" = VAR variable | CONST object
```

A type is a list of type names (Either-type).

```
datatype type = Either (primitives: "name list")
```

An action schema has a name and a list of typed parameters, as well as a precondition and an effect, which are defined over terms (instead of objects for ground actions).

```
datatype ast_action_schema = Action_Schema
    (name: name)
    (parameters: "(variable × type) list")
    (precondition: "term atom formula")
    (effect: "term ast_effect")
```

Finally, an action of a plan is a name (referencing an action schema) and a list of (argument) objects:

```
datatype plan_action = PAction (name: name) (args: "object list")
```

Similarly, we model the remaining abstract syntax, based on Kovacs' grammar[3]. We only display the top-level entities for a PDDL domain and instance here:

```
datatype ast_domain = Domain               datatype ast_problem = Problem
  (types: "(name × name) list")              (domain: ast_domain)
  (predicates: "predicate_decl list")        (objects: "(object × type) list")
  (consts: "(object × type) list")           (init: "object atom formula list")
  (actions: "ast_action_schema list")        (goal: "object atom formula")
```

[3]We sometimes use slightly different names than Kovacs.

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[4].

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 formulae 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 oT) ⊆ subtype_rel* `` set (primitives T)"
```

Here, ⟨*⟩ denotes reflexive transitive closure, and ⟨``⟩ 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 M πs M' =
      ((∀ π ∈ set πs. wf_plan_action π)
    ∧ ground_action_path M (map resolve_instantiate 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'. plan_action_path I πs M' ∧ M' ᶜ⊫₌ (goal P)"
```

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

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 semantics 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 ⟨s $\models \varphi$⟩ to denote that a valuation ⟨s⟩ entails a formula ⟨φ⟩. However, we extend the entailment operator of their logic to include equalities and denote this new entailment by ⟨s $\models_= \varphi$⟩. Lifschitz, and we, define abstract actions as partial functions from abstract states to abstract states. However, we model partiality of actions using the ⟨option⟩ type in Isabelle.

To demonstrate the simulation, we define mappings from world-models to abstract states and from ground actions to abstract actions. A world-model ⟨M⟩ is mapped to an abstract state ⟨s⟩ if $\forall \varphi \in$close_world M. s $\models_= \varphi$. Let α be a ground action with precondition ⟨pre⟩, and effects ⟨add⟩ and ⟨del⟩. α is mapped to an abstract action f if for any abstract state s, ⟨s $\models_=$ 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 ⟨atm⟩ (i) if ⟨atm $\notin$ del⟩ and ⟨s $\models_=$ atm⟩, then ⟨s' $\models_=$ atm⟩, and (ii) if ⟨atm $\notin$ add⟩ and ⟨s $\models_= \neg$ atm⟩, then ⟨s' $\models_= \neg$ atm⟩, and (iii) if ⟨atm $\in$ add⟩, then ⟨s' $\models_=$ atm⟩w, and (iv) if ⟨atm $\in$ del⟩ and ⟨atm $\notin$ add⟩, then ⟨s' $\models_= \neg$ 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 ⟨add⟩ 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 ⟨πs⟩, the validator checks that the problem is well-formed and the plan is valid:

validator P πs $\equiv$ wf_problem P $\wedge$ valid_plan P πs

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 mathematical 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

definition of_type_impl G oT T

$\equiv$ ($\forall$ pt$\in$set (primitives oT). dfs_reachable G (op=pt) (primitives T))

Here, the ⟨G⟩ parameter will be instantiated by the tabulated successor function of the subtype graph, which is only computed once and passed as extra parameter to all subsequent functions (λ-lifting). We prove

lemma of_type_impl_correct: of_type_impl STG oT T $\longleftrightarrow$ of_type oT T

Above, ⟨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:

```
definition en_exE2 where
  en_exE2 G mp ≡ λ(PAction n args) ⇒ λM. 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:

```
lemma (in wf_ast_problem) en_exE2_return_iff:
  assumes wm_basic M
  shows en_exE2 STG mp_objT a M = Inr M'
    ⟷ plan_action_enabled a M ∧ 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.

```
lemma check_plan_return_iff:
   check_plan P πs = Inr () ⟷ wf_problem P ∧ valid_plan P πs
```

For a problem P and an action sequence πs, 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 OA OB)› will be satisfied by an atom ‹(P O AOB)› 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.

	VAL (Min/Max)	INVAL (Min/Max)	* (Min/Max)
logistics	0/.048(406)	.080/.360(406)	0/.024(406)
elevators	.004/.036(20)	.104/.268(20)	.004/.048(20)
rover	0/.088(70)	.092/.856(70)	0/.116(70)
nomystery	.004/.628(37)	.120/49.916(50)	.004/4.992(50)
zeno	0/.012(40)	.084/.136(40)	0/.004(40)
hiking	0/.004(38)	.096/.564(38)	0/.012(38)
TPP	0/.020(30)	.088/.176(30)	0/.016(30)
Transport	.008/.016(11)	.088/.152(11)	.008/.020(11)
GED	.004/.020(40)	.100/.328(40)	0/.020(40)
woodworking	.004/.016(20)	.096/.164(20)	.008/.020(19)
visitall	0/.528(70)	.084/50.992(70)	0/1.304(70)
openstacks	0/.188(60)	.084/1.832(80)	0/.344(80)
satellite	0/0(10)	.112/.136(10)	0/0(10)
scanalyzer	0/.016(40)	.096/.144(60)	0/.012(60)
gripper	0/.004(22)	.088/.152(22)	0/.004(22)
tidybot	.004/.012(47)	.092/.148(47)	0/.040(47)
storage	0/.008(19)	.096/.136(19)	0/.004(19)
trucks	0/.004(2)	.112/.116(2)	0/0(2)
parcprinter	.004/.012(20)	.088/.132(34)	0/.028(34)
pipesworld	.004/.024(43)	.084/.144(43)	0/.024(43)
pegsol	.004/.008(40)	.108/.200(60)	0/.012(60)
Parking	0/.008(80)	.088/.500(100)	0/.008(100)
blocksworld	0/.004(10)	.120/.140(10)	0/0(10)
floortile	.004/.004(20)	.080/.120(24)	0/.012(24)
barman	.004/.012(42)	.100/.168(42)	0/.016(42)
Thoughtful	.004/.020(16)	.136/.532(16)	.004/.016(16)
childsnack	0/.004(10)	.112/.144(10)	0/.012(10)

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 topic of this work 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 a planning problem or domain, etc. Also, unsolvable planning problems can be tackled by implementing a verified checker for unsolvability certificates like the ones in [31]. A more comprehensive approach to unsolvable planning would be to implement a planner whose soundness and completeness is formally verified. This could be done by verifying SAT encodings for planning problems like Rintanen's [4] and upper bounds on plan lengths like the ones by Abdulaziz et al. [32], [33]. This could then be combined with a verified SAT solver [28] or an existing verified SAT certificate checker [29].

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

References

[1] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[2] C. Green, "Application of theorem proving to problem solving," SRI, Tech. Rep., 1969.

[3] H. A. Kautz and B. Selman, "Planning as satisfiability," in *ECAI*, 1992, pp. 359–363.

[4] J. Rintanen, "Planning as satisfiability: Heuristics," *Artificial Intelligence*, vol. 193, pp. 45–86, 2012.

[5] R. Howey, D. Long, and M. Fox, "VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL," in *Tools with Artificial Intelligence, 2004*, 2004.

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

[7] M. Fox, R. Howey, and D. Long, "Validating plans in the context of processes and exogenous events," in *AAAI*, vol. 5, 2005, pp. 1151–1156.

[8] S. Thiébaux, J. Hoffmann, and B. Nebel, "In defense of PDDL axioms," *Artificial Intelligence*, vol. 168, no. 1-2, pp. 38–69, 2005.

[9] K. Slind and M. Norrish, "A brief overview of HOL4," in *TPHOLs*, 2008.

[10] L. C. Paulson, *Isabelle: A generic theorem prover*. Springer, 1994, vol. 828.

[11] T. Coquand and G. Huet, "The calculus of constructions," *Information and computation*, vol. 76, no. 2-3, pp. 95–120, 1988.

[12] U. Norell, *Towards a practical programming language based on dependent type theory*. Chalmers University of Technology, 2007, vol. 32.

[13] V. Lifschitz, "On the semantics of strips," in *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, 1987, pp. 1–9.

[14] F. Haftmann and T. Nipkow, "A code generator framework for Isabelle/HOL," *Theorem Proving in Higher Order Logics (TPHOLs 2007). Lecture Notes in Computer Science*, vol. 4732, pp. 128–143, 2007.

[15] A. Lochbihler, "Light-weight containers for isabelle: efficient, extensible, nestable," in *ITP*, 2013.

[16] R. Milner, *The definition of standard ML: revised*. MIT press, 1997.

[17] ——, "Logic for computable functions description of a machine implementation," Stanford University, Tech. Rep., 1972.

[18] D. L. Kovacs, "Bnf definition of pddl 3.1," *IPC-2011*, 2011.

[19] J. Michaelis and T. Nipkow, "Formalized proof systems for propositional logic," in *23rd Int. Conf. Types for Proofs and Programs (TYPES 2017)*, ser. LIPIcs, A. Abel, F. N. Forsberg, and A. Kaposi, Eds., vol. 104. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 6:1–6:16.

[20] M. Norrish, "C formalised in hol," University of Cambridge, Computer Laboratory, Tech. Rep., 1998.

[21] M. Helmert, "The Fast Downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[22] J. Brunner and P. Lammich, "Formal verification of an executable ltl model checker with partial order reduction," in *NASA Formal Methods Symposium*. Springer, 2016, pp. 307–321.

[23] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[24] J. McCarthy, "Formalization of strips in situation calculus," Citeseer, Tech. Rep., 1985.

[25] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[26] G. Klein *et al.*, "seL4: Formal verification of an OS kernel," in *SOSP*. ACM, 2009, pp. 207–220.

[27] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens, "Hol with definitions: Semantics, soundness, and a verified implementation," in *ITP*, 2014.

[28] J. C. Blanchette, M. Fleury, and C. Weidenbach, "A verified sat solver framework with learn, forget, restart, and incrementality," in *International Joint Conference on Automated Reasoning*. Springer, 2016, pp. 25–44.

[29] P. Lammich, "Efficient verified (UN)SAT certificate checking," in *CADE*, 2017. [Online]. Available: https://doi.org/10.1007/978-3-319-63046-5_15

[30] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus, "A fully verified executable LTL model checker," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 463–478.

[31] S. Eriksson, G. Röger, and M. Helmert, "Unsolvability certificates for classical planning," 2017.

[32] M. Abdulaziz, C. Gretton, and M. Norrish, "A State Space Acyclicity Property for Exponentially Tighter Plan Length Bounds," in *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2017.

[33] M. Abdulaziz, M. Norrish, and C. Gretton, "Formally verified algorithms for upper-bounding state space diameters," *J. Autom. Reasoning*, vol. 61, no. 1-4, pp. 485–520, 2018.