Formally Verified SAT-Based AI Planning

Mohammad Abdulaziz[1,2], **Friedrich Kurz**[2]

[1] King's College London
[2] Techniche Universität München

Abstract

We present an executable formally verified SAT encoding of ground classical AI planning problems. We use the theorem prover Isabelle/HOL to perform the verification. We experimentally test the verified encoding and show that it can be used for reasonably sized standard planning benchmarks. We also use it as a reference to test a state-of-the-art SAT-based planner, showing that it sometimes falsely claims that problems have no solutions of certain lengths.

Introduction

Planning systems are becoming more and more scalable and efficient, as shown by different planning competitions (Long et al. 2000; Coles et al. 2012; Vallati et al. 2015), making them suited for realistic applications. Since many applications of planning are safety-critical, increasing the trustworthiness of planning algorithms and systems is instrumental to their widespread adoption. Consequently there currently are substantial efforts to improve the trustworthiness of planning systems (Howey, Long, and Fox 2004; Eriksson, Röger, and Helmert 2017; Abdulaziz, Norrish, and Gretton 2018; Abdulaziz and Lammich 2018).

Increasing trustworthiness of software is a well-studied problem. Three approaches have been tried in the literature (Abdulaziz, Mehlhorn, and Nipkow 2019). Firstly, a system's trustworthiness can be increased by applying software engineering techniques, e.g. programming at the right level of abstraction, code reviewing, and testing. Although these practices are relatively easy to implement, they are incomplete.

Secondly, there is certified computation, where the given program computes, along with its output, a certificate showing why this output is correct. This relegates the burden of trustworthiness to the certificate checker, which should be much simpler than the system whose output is to be certified, and thus is less error prone. Certified computation was pioneered by Mehlhorn and Näher 1998 who used it for their LEDA library. In the realm of planning, this approach was pioneered by Howey, Long, and Fox who developed the plan validator VAL (Howey, Long, and Fox 2004). Also, certifying unsolvability for planning was pioneered by Eriksson, Röger, and Helmert 2017. A fundamental problem with certified computation for classical planning, as well as other PSPACE-complete problems, is that no succinct certificates exist unless $\mathrm{NP} = \mathrm{PSPACE}$ (Arora and Barak 2009, Chapter 4).

Thirdly, there is formal verification, where system properties are verified by means of a mechanically checkable formal mathematical proof. This gives the highest possible trust in a program. Unlike certified computation, formal verification guarantees program completeness, in addition to its output correctness. Nonetheless, formal verification needs intensive effort, and it is usually much harder for a formally verified program to perform as efficiently as an unverified program since verifying all performance optimisations is usually infeasible. Nonetheless, formal verification has seen recent wide-spread success. Notable applications include a verified OS kernel (Klein et al. 2009), a verified SAT-solver (Blanchette et al. 2018), a verified model checker (Esparza et al. 2013), a verified conference system (Kanav, Lammich, and Popescu 2014), a verified optimised C compiler (Leroy 2009), and, in the context of planning, verified validators (Abdulaziz and Lammich 2018; Abdulaziz and Koller 2022), and algorithms for bounding plan lengths (Abdulaziz, Norrish, and Gretton 2018).

Encoding planning problems as logical formulae has a long history (McCarthy and Hayes 1981). SAT-based planning (Kautz and Selman 1992) is the most successful such approach, where the question of whether a planning problem has a solution of bounded length is encoded into propositional (SAT) formulae. For a sound and complete encoding, a formula for a length bound (aka horizon) h is satisfiable iff there is a solution with length bounded by h. A series of such formulae for increasing horizons are passed to a SAT solver until a plan is found. The encoding of Kautz and Selman was significantly improved by exploiting problem symmetries (Rintanen 2003), state invariants, and by parallelising the encodings (Rintanen, Heljanko, and Niemelä 2006). In this paper we present a formally verified $\forall$-step parallel encoding of classical planning (Kautz and Selman 1996; Rintanen, Heljanko, and Niemelä 2006). Using it as a reference, we discover bugs in the state-of-the-art SAT-based planner Madagascar (Rintanen 2012). We also show that our encoder is an order of magnitude slower than Madagascar, which is reasonable for formally verified software.

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Motivation Before we delve into details, we find it sensible to answer the question of why to verify a SAT encoding of planning? The first motivation is to demonstrate the process of formally verifying a state-of-the-art planning algorithm. Secondly, a verified planner can be used as a reference implementation, against which unverified planners can be tested. Thirdly, SAT encodings of planning are particularly suited for formal verification since the process of encoding the problem into a SAT formula is not the computational bottleneck. This means that not much effort needs to be put into verifying different implementation optimisations. Also, importantly, verified SAT solving technology is continuously advancing, which means that the performance of verified planning using the verified encoding will continuously improve. E.g. the verified SAT solver by Blanchette et al. is one to two orders of magnitude slower than Minisat (Eén and Sörensson 2003), and the verified unsatisfiability certificate checker GRAT (Lammich 2017) is faster than DRAT-trim (Wetzler, Heule, and Jr 2014). Lastly, the problem of increasing trustworthiness of 'traditional' SAT-based planning was most recently raised by Eriksson and Helmert (2020). They stated, as an open problem, certifying that an unsatisfiable SAT formula encoding a planning problem indeed shows a lower bound on the solution length. We address this open problem and show it is solvable with formal verification.

Approach Overview Our approach has two focuses, which we try to highlight in this paper. The first is *minimising trusted code*. To reduce the trusted code base we make sure the encoder and the decoder take (produce) as input (output) abstract syntax trees (AST) of the formats used in the input (output) files. This way we only trust parsing and pretty printing components. The encoder takes an AST of Fast-Downward's translator format (Helmert 2006) as input (hereafter, FD-AST) and produces an AST of the standard DIMACS-CNF format. This DIMACS-CNF is passed to an unverified SAT solver. If the SAT solver finds a model for the encoded formula, our verified decoder takes the AST of the DIMACS model and the FD-AST and produces a plan only if the model entails the CNF encoding of the FD-AST. If the SAT solver finds the formula unsatisfiable, we take the unsatisfiability certificate and pass it, together with the DIMACS-CNF encoding, to the formally verified unsatisfiability certificate checker of Lammich 2017. Accordingly, all outputs of our system have formal guarantees, whether the output is a plan or a conclusion that none exists.

The second is *engineering tradeoffs for feasible verification*. Formally verifying a reasonably big piece of software is a daunting task. E.g. it has been reported that the size of the proof scripts for verifying the OS kernel *seL4* is quadratically as big as the C implementation of the kernel (Klein et al. 2009). Thus, a major verification effort like verifying a SAT-based planner needs careful engineering and modularisation. For complicated algorithms, the most successful approach is stepwise refinement, where one starts with an abstract version of the algorithm and verifies it. Then one devises more optimised versions of the algorithm, and only proves the optimisations correct. This approach was used in most successful algorithm verification efforts (Klein et al. 2009; Blanchette et al. 2018; Esparza et al. 2013; Kanav, Lammich, and Popescu 2014).

```
1                    begin_state   1                p cnf 6 15
begin_variable       0             begin_operator   3 0
var1                 end_state     move r0 r1       -5 0
-1                   begin_goal    0                -4 0
2                    1             1                ...
Atom at-robby(r0)    0 1           0 0 0 1          5 -6 1 0
Atom at-robby(r1)    end_goal      0                -3 4 1 0
end_variable                       end_operator     -5 6 0
                                                    -1 1 0
```

Figure 1: The first three listings are the concrete syntax of a planning problem in Fast Downward's translator format. The fourth listing is a CNF-DIMACS formula.

For compilers, a different approach is used: one splits the compiler into smaller translation steps, and verifies each one of those steps separately. In the end, the composition of those verified transformations is the verified compiler. This approach was used in all notable verified compilers (Leroy 2009; Kumar et al. 2014). We follow this methodology.

Isabelle/HOL We perform the verification using the interactive theorem prover Isabelle/HOL (Nipkow, Paulson, and Wenzel 2002), which is a theorem prover for Higher-Order Logic. Roughly speaking, Higher-Order Logic can be seen as a combination of functional programming with logic. Isabelle is designed for trustworthiness: following the Logic for Computable Functions approach (LCF) (Milner 1972), a small kernel implements the inference rules of the logic. 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.

Availability All theorems in this paper were formally proved in Isabelle/HOL. The full formal proofs can be found at https://www.isa-afp.org/entries/Verified_SAT_Based_AI_Planning.html and the verified planner based on these proofs can be found at https://github.com/mabdula/Verified-SAT-Based-Planning.

Background

In this paper, lists (sets) of objects are written between square brackets (curly braces). E.g. $[a, b, c]$ ($\{a, b, c\}$) denotes the list (set) of objects a, b and c. We also make use of the choice function ch, which, for a non-empty set s, denotes an arbitrary element of s. If $s = \{\}$, ch is undefined. A mapping $f : V \to A$ is a set of maplets, s.t., for every $v \mapsto a \in f$, $v \in V$ and $a \in A$, and, if $v \mapsto a_1 \in f$ and $v \mapsto a_2 \in f$, then $a_1 = a_2$. For a mapping f, we define $f(v)$ to be a if $v \mapsto a \in f$, otherwise it is undefined. Also, for a mapping $f : V \to A$, $\mathcal{D}(f)$ denotes $\{v \mid v \mapsto a \in f\}$. We call f a complete mapping iff $\mathcal{D}(f) = V$. Otherwise, we call it a partial mapping.

There are multiple formalisms to represent planning and logical formulae. Some of those formalisms are geared towards being file formats and others are abstract formalisms that are geared towards performing abstract and pen-and-paper reasoning. Since the main goal of our development is to minimise trusted (i.e. unverified) code, we aim at having our verified program (i) take as input as close a representation as possible to the actual input format (i.e. Fast-Downward's translator format[1]) and (ii) produce as output a format that is as close to the output file format as possible (i.e. the DIMACS-CNF format). However, although we want our verified program to operate on inputs (outputs) that are as close as possible to the input (output) files, it cannot directly operate on the input (output) strings. In particular, the input strings have to be parsed into ASTs, and the program has to produce an output that is also an AST, which is to be pretty printed into an output file.

Definition 1 (FD-AST)**.** *An FD-AST is a tuple (V, I, G, O), where $V : \mathbb{N} \to \mathbb{N}$ is the variables section, $I : \mathbb{N} \to \mathbb{N}$ is the initial state, $G : \mathbb{N} \to \mathbb{N}$ is the goal, and O is the set of operators. For each $v \mapsto D_v \in V$, v stands for a variable and D_v is the number of assignments this variables can take. An operator is a tuple $(name, ps, es)$, where name is the operator's name, $ps : \mathbb{N} \times \mathbb{N}$ is a partial mapping called the prevail preconditions, and es is a set of effects. An effect is a tuple $(epre, v, l, m)$, where $epre : \mathbb{N} \to \mathbb{N}$ is a partial mapping called the effect's precondition and $v, l, m \in \mathbb{N}$. If epre is non-empty the effect is called conditional.*

Note: in parsing, we ignore parts of the format that are irrelevant to our encoding, e.g. metric, mutex, and axiom sections.

Definition 2 (Well-Formedness)**.** *An FD-AST (V, I, G, O) is well-formed iff (i) I is a well-formed state w.r.t. the problem, (ii) G is a well-formed partial state w.r.t. the problem, (iii) all operators have distinct names, and (iv) every operator is well-formed w.r.t. the problem. A (partial) state s is well-formed w.r.t. a problem (V, I, G, O) iff it is a (partial) mapping s.t. $\mathcal{D}(s) \subseteq \mathcal{D}(V)$, and for any $v \in \mathcal{D}(s)$, $s(v) < V(v)$. An operator $(name, ps, es)$ is well-formed iff (i) ps is a well-formed partial state, (ii) $v_1 \neq v_2$, for every $(epre_1, v_1, l_1, m_1), (epre_2, v_2, l_2, m_2) \in es$ and (iii) for every $(epre, v, l, m) \in es$: (iii)(1) epre is a well-formed partial state, and (iii)(2) $l, m < V(v)$.*

Definition 3 (Execution)**.** *An operator $(name, ps, es)$ is executable in a state s iff $ps \subseteq s$ and, for each effect $(epre, v, l, m) \in es$, $v \mapsto l \in s$, if $0 \leq l$. The state resulting from executing the operator is*

$$(s \setminus \{v \mapsto l \mid \exists (epre, v, l, m) \in es \wedge epre \subseteq s\}) \cup \{v \mapsto m \mid \exists (epre, v, l, m) \in es \wedge epre \subseteq s\},$$

i.e. it is the same as s, except that it assigns variables according to the effects whose preconditions are satisfied.

A solution to a problem P is a list of names $[name_1, name_2, \ldots name_n]$ s.t. there is a mapping $sol : \{name_1 \ldots name_n\} \to O$ s.t. $sol(name_i) \in O$, for all $1 \leq i \leq n$, and the sequence of operators $[sol(name_1), sol(name_2), \ldots sol(name_n)]$ are executable in-order starting at I, and the goal is a subset of the state resulting from executing all operators.

Example 1. *Figure 1 shows a planning problem in Fast Downward's translator syntax that models a robot that is in one room and whose goal is to move to another room. The abstract syntax tree of that problem is $([(v_0, 2)], \{v_0 \mapsto 0\}, \{v_0 \mapsto 1\}, (move, 0, \{\}, \{([], v_0, 0, 1)\}))$. A solution to this problem is $[move]$.*

[1] http://www.fast-downward.org/TranslatorOutputFormat

We limit ourselves to problems without conditional effects and with only consistent preconditions, defined as follows.

Definition 4 (Valid FD-AST Problem)**.** *We call a problem (V, I, G, O) to be valid iff (i) it is well-formed, (ii) $epre = \emptyset$, for any $(name, ps, es) \in O$ and $(epre, v, l, m) \in es$, i.e. it has no conditional effects, and (iii) for any variable $v \in \mathcal{D}(p_1) \cap \mathcal{D}(p_2)$ where $p_1, p_2 \in \{ps\}$, where $(name, ps, es) \in O$, we have $p_1(v) = p_2(v)$, i.e. all preconditions are consistent.*

To formally verify our encoding, we need to formalise the definitions in Isabelle/HOL's logic. As input, we use the formalisation of FD-AST developed by Abdulaziz and Lammich 2018.

Note that this representation of planning problems, as well as the final SAT formula, have features which make it is too cumbersome for abstract pen-and-paper reasoning. However, we need to start from representations which are as close as possible to the input/output file to reduce trusted code. In particular, if we prove our encoding to be correct on other more abstract representations, we will have to use trusted (i.e. unverified) pre-processing programs to convert between the files and the more abstract representation. We also note that this software is only a simple parser if one considers the input to be in FD's translator's format, which is our claimed input. The trusted code base is significantly larger if one considers the input to be a PDDL domain and instance, as the conversion then includes grounding and the computation of invariants, etc.

The rest of the paper is structured s.t. there is a section describing every intermediate representation and translation step, with the associated correctness theorem.

Translating FD-AST to FDR

Although we can define our encoding directly on FD-AST, we opted to firstly translate the FD-AST to the Finite Domain Representation (FDR), which is another representation of planning problems with multi-valued state variables. FDR is more abstract than FD-AST. This facilitates smoother formal reasoning, e.g. it is more suitable for stating algorithms and theorem statements, while FD-AST is a file format. In most expositions in planning literature, there is not a distinction between file formats and the abstract formalisms on which algorithms and theorems are stated. The actual implementations, however, start from a file format, like Fast Downward's translator format, which is simplified to a more abstract formalism. In our case, nonetheless, we make the distinction between the two formalisms since our main goal is a formal correctness guarantee on an implementation of a planner, including the translation of FD-AST to FDR.

Definition 5 (Finite Domain Representation)**.** *An FDR planning problem Ψ is a tuple $(V, \mathcal{R}, I, O, G)$, where V is the set of state variables, $\mathcal{R} : V \to A_v$ is mapping from variables to sets of assignments, O is a set of operators, I is the initial state, and G is the goal. Each variable (operator) has a unique natural number index, where, for a variable $v \in V$*

(operator $op \in O$), its index is $0 \leq v_i < |V|$ ($0 \leq op_i < |O|$). For FDR, a (partial) state $s : V \to \bigcup\{\mathcal{R}(v) \mid v \in V\}$ is a mapping, and $\mathcal{D}(s) = \{v \mid v \mapsto a \in s\}$. A (partial) state s is valid iff for any $v \mapsto a \in s$ we have that $a \in \mathcal{R}(v)$. A valid operator is a pair of valid partial states $op = (p, e)$, where p is the precondition, denoted by $p(op)$, and e is the effect, denoted by $e(op)$. We denote the the execution of a sequence of operators ops at a state s by ops(s), and it is defined as follows: if ops is not empty and if for the first operator (p, e) in ops we have $p \subseteq s$ then ops(s) = ops$'(op(s))$, where ops$'$ is the tail (i.e. every element but the first) of ops and $op(s)$ is defined to be $\{v \mapsto a \mid \text{if } v \in \mathcal{D}(e) \text{ then } a = e(v) \text{ else } s(v)\}$. Otherwise, ops$(s) = s$. Ψ is a valid FDR problem iff I is a valid state, G is a valid partial state, and O is a set of valid operators. A solution for Ψ is an operator sequence ops where all operator in ops come from O and $G \subseteq$ ops(I).

Note: in contrast to FD-AST plan execution semantics, FDR execution semantics are defined as a total function, where plan execution always returns the last state reached before the first operator whose preconditions are not satisfied. A total execution function makes many of the formal proofs easier.

Translating FD-ASTs into FDR problems is done using the following encoding.

Encoding 1. *For an FD-AST problem $P = (V, I, G, O)$, let $FDR_V = \{v \mid \exists n.\ (v, n) \in V\}$ and $FDR_{\mathcal{R}} = \{v \mapsto \{0 \ldots n-1\} \mid (v, n) \in V\}$. For an FD-AST effect $e = (epre, v, l, m)$, let e_{old} denote $v \mapsto l$ and e_{new} denote $v \mapsto m$. For an operator $op = (name, ps, es)$, let $FDR_O(op) = (ps \cup \{e_{old} \mid e \in es\}, \{e_{new} \mid e \in es\})$. For the FD-AST problem P, its encoding as an FDR problem, FDR(P), is $(FDR_V, FDR_{\mathcal{R}}, I, \{FDR_O(op) \mid op \in O\}, G)$.*

For the other direction of this encoding, we devise a decoding function AST that, given a solution for the FDR problem *FDR*(P), decodes it into a solution for P.

Decoding 1. *First, for a set s let $\mathsf{ch}(s)$ denote an arbitrary element of s if s is not empty, and undefined otherwise. For an FDR operator op, let $AST(op) = \mathsf{ch}\{name \mid FDR_O((name, ps, es)) = op\}$, where O are the operators in P.*

Example 2. *The compiled FDR equivalent to the FD-AST problem P in Example 1, FDR(P), is*

$$\begin{pmatrix} FDR_V = \{v_0\}, FDR_{\mathcal{R}} = \{v_0 \mapsto \{0,1\}\}, \\ I = \{v_0 \mapsto 0\}, FDR_O = \{op_0\}, FDR_G = \{v_0 \mapsto 1\} \end{pmatrix},$$

where $op_0 = (\{v_0 \mapsto 0\}, \{v_0 \mapsto 1\})$.

The following theorem represents the soundness and completeness of this compilation step.

Theorem 1. *Let P be a valid FD-AST problem. We have that: (i) if $[name_1, name_2, \ldots]$ is a plan for P then $[FDR_O(\mathsf{ch}(O_{name_1}), FDR_O(O_{name_2}), \ldots]$ is a plan for the FDR task FDR(P), where, for $name_i$, $O_{name_i} = \{op \mid \exists\, ps'\ es'.op \in O \wedge op = (name_i, ps', es')\}$, and (ii) if $[op_1, op_2, \ldots]$ is a plan for the FDR task FDR(P), then $[AST(op'_1), AST(op'_2), \ldots]$ is a plan for P, where $[op'_1, op'_2, \ldots]$ are the operators from the given FDR plan whose preconditions are satisfied.*

Proof sketch. Both statements are proved by induction on the length of the given plan, while generalising over the initial state of P, and then careful unfolding of Definitions 1, 2, 3, 4, 5, Encoding 1 and Decoding 1. □

Note: since FDR execution function is total, while that of FD-AST is not, operators whose preconditions are not satisfied have to be removed when decoding the FDR plan.

Before we close this section we note a few points regarding the formal proof of the above theorem. The proof of this theorem does not have complicated mathematical ideas or constructions. However, the main difficulty is correctly formulating the definitions of well-formed and valid FD-ASTs, valid FDRs and the encoding and the decoding. Due to the many conjuncts and components of these definitions, their interactions make formally stating these definitions and proving the theorem a very error-prone and cumbersome process. E.g. a detail in the formal proof, which would be glossed over in a pen-and-paper treatment, is to show that encoding a well-formed valid FD-AST results in a valid FDR, which is necessary for using the different theorems about FDR problems. Proving that depends on the assumption that the AST operators have no conditional effects, a fact which we only understood during our development of the formal proof. To overcome these difficulties, we employed deliberate engineering efforts to make the formal proof more modular. E.g. we split the encoding of FD-AST operator to FDR operators into multiple stages. First, the effect preconditions are removed and added into the operator's preconditions, resulting in a simpler FD-AST, with no effect preconditions. Then, we define a function to encode these simpler FD-ASTs into FDRs.

Translating FDR to STRIPS

Since a SAT encoding only has propositional variables, we need to compile the multi-valued state variables of FDR to propositional values. Instead of performing the compilation of the multi-valued variables together with the compilation of the transition relation in one step, we opted to do them in separate steps, where we first compile FDR problems to STRIPS problems and then compile STRIPS problems into SAT formulae. This decision is not of much theoretical importance, but more geared towards making the verification more modular and thus more manageable.

Definition 6 (STRIPS Problem)**.** *An FDR problem $\Pi = (V, \mathcal{R}, I, O, G)$ is a STRIPS problem iff $\mathcal{R}(v) = \{\bot, \top\}$, for all $v \in V$. When constructing STRIPS mappings, we denote $v \mapsto \top$ with v and $v \mapsto \bot$ with $\overline{v}$.*

Encoding 2. *Consider an FDR problem $\Psi = (V, \mathcal{R}, I, O, G)$. Our FDR problem compiled to STRIPS is*

$$\varphi(\Psi) = \begin{pmatrix} \varphi_V = \{(v, a) \mid a \in \mathcal{R}(v) \wedge v \in V\}, \\ \varphi_{\mathcal{R}} = \{(v, a) \mapsto \{\top, \bot\} \mid (v, a) \in \varphi_V\}, \\ \varphi_S(I), \varphi_S(G), \\ \varphi_O = \{(\varphi_S(p), \varphi_S(e)) \mid (p, e) \in O\}. \end{pmatrix},$$

where, for a (partial) state s, $\varphi_S(s)$ is defined as

$$\{(v, a) \mid s(v) = a \wedge v \in \mathcal{D}(s)\} \cup$$
$$\{\overline{(v, a)} \mid s(v) \neq a \wedge v \in \mathcal{D}(s) \cap V \wedge a \in \mathcal{R}(v)\}$$

Example 3. *Consider the FDR problem FDR(P) in example 2. For that FDR problem, we have*

$$\begin{pmatrix} \varphi_V = \{(v_0,0),(v_0,1)\}, \\ \varphi_{\mathcal{R}} = \{(v_0,0) \mapsto \{\bot,\top\},(v_0,1) \mapsto \{\bot,\top\}\}, \\ \varphi_S(I) = \{(v_0,0),\overline{(v_0,1)}\},\varphi_S(G) = \{\overline{(v_0,0)},(v_0,1)\}, \\ \varphi_O = \{(\{(v_0,0),\overline{(v_0,1)}\},\{\overline{(v_0,0)},(v_0,1)\})\} \end{pmatrix}$$

Decoding 2. *For an FDR problem $\Psi = (V,\mathcal{R},I,O,G)$ and an operator $(p,e) \in \varphi_O$, let*

$$\varphi_O^{-1}(p,e) = (\{v \mapsto a \mid (v,a) \in p\},\{v \mapsto a \mid (v,a) \in e\})$$

Note: we ignore negative effects when decoding STRIPS operators since they only ensure operator effect consistency.

The soundness and completeness theorems of this encoding of FDR problems follow.

Theorem 2. *For a valid FDR problem Ψ (i) if $[op_0,\ldots,op_k]$ solves the STRIPS problem $\Pi = \varphi(\Psi)$, then $\psi = [\varphi_{\mathrm{O}}^{-1}(op_0),\ldots,\varphi_{\mathrm{O}}^{-1}(op_k)]$ is a solution for Ψ. (ii) if $[op_0,\ldots,op_k]$ solves Ψ, then $\pi = [\varphi_{\mathrm{O}}(op_0),\ldots,\varphi_{\mathrm{O}}(op_k)]$ solves the STRIPS problem $\varphi(\Pi)$.*

Proof sketch. (i) We show that $G \subseteq \psi(I)$ and moreover $\varphi_O^{-1}(op_0),\ldots,\varphi_O^{-1}(op_k) \in \mathcal{O}$ where G and $\mathcal{O}$ are the goal state respectively operator set of Ψ. The first part of the proof is by induction over π with arbitrary initial state.
(ii)We show that $G \subseteq \pi(I)$ and moreover $\varphi_O(op_0),\ldots,\varphi_O(op_k) \in \mathcal{O}$ where G and $\mathcal{O}$ are the goal state respectively operator set of Π. The first part of the proof is by induction over ψ with arbitrary initial state. □

Encoding STRIPS Problems as SAT

In this step we encode the question of whether a STRIPS problem has a plan of length at most h into a propositional satisfiability formula. In our formalisation, we use Michaelis and Nipkow's formalisation of propositional logic. The specific encoding we use is similar to the parallel ∀-step encoding used by Rintanen, Heljanko, and Niemelä. We limit operators to ones without conditional effects, require a total initial state, and constrain preconditions and the goal to conjunctions of literals. These restrictions are always satisfied by problems produced by Encoding 2. Informally, such a parallel encoding constitutes an unrolling of the transition relation underlying the STRIPS problem, which allows more than one operator to execute in one time step, as long as those operators are *non-interfering*. This allows for the encoding to be significantly more compact in practice, compared to only allowing one operator per step.

Definition 7 (Interference)**.** *Two STRIPS operators $o_1 = (p_1,e_1)$ and $o_2 = (p_2,e_2)$ are interfering iff $\{v \mid v \mapsto \top \in p_i\} \cap \{v \mid v \mapsto \bot \in e_j\} \neq \emptyset$, for all $i \neq j$, and $i,j \in \{1,2\}$. For a set of STRIPS operators O, we denote the set of pairs of interfering operators in O by intrfr(O).*

Encoding 3. *Consider a given natural number h and a STRIPS problem $\Pi = (V,R,I,O,G)$. For a STRIPS state s, let s^t denote the propositional formula*

$$(\bigwedge v \in \{v \mid v \mapsto \top \in s\}.\ v^t) \wedge (\bigwedge v \in \{v \mid \overline{v} \in s\}.\ \neg v^t)$$

Also, for a variable $v \in V$, let $add(v) = \{(p,e) \mid v \in e\} \cap O$ and $del(v) = \{(p,e) \mid \overline{v} \in e\} \cap O$. The encoding $\Phi(\Pi,h)$ is the conjunction of the following propositional formulae:

$$I^0 \quad \text{(i)}$$

$$G^h \quad \text{(ii)}$$

$$\bigwedge t \in \{0..h\}.\ \bigwedge op \in O.\ op^t \longrightarrow p(op)^t \wedge op^t \longrightarrow e(op)^{t+1} \quad \text{(iii)}$$

$$\bigwedge t \in \{1..h\}.\ \bigwedge v \in V.\ \neg v^{t-1} \wedge v^t \longrightarrow \bigvee op \in add(v).\ op^t \quad \text{(iv)}$$

$$\bigwedge t \in \{1..h\}.\ \bigwedge v \in V.\ v^{t-1} \wedge \neg v^t \longrightarrow \bigvee op \in del(v).\ op^t \quad \text{(v)}$$

$$\bigwedge t \in \{1..h\}.\ \bigwedge (op,op') \in intrfr(O).\ op^t \vee \neg op'^t \quad \text{(vi)}$$

This encoding is defined over the atoms

$$\{v^t \mid v \in V \wedge 0 \leq t \leq h\} \cup \{op^t \mid op \in O \wedge 0 \leq t < h\}$$

In the encoding above the first conjunct stands for the initial state, the second for the goal, the third for the transition relation, the fourth and fifth are the frame axioms, and the last is a constraint ensuring that if more than one operator execute in the same step, they are not interfering operators. Also note that the actual encoding we verified only computes the formula in CNF form, but we use syntactic sugar in our definition and examples to improve readability, e.g. $x_1 \wedge x_2 \longrightarrow \bigvee y \in \{y_1,y_2,..\}.y$ is syntactic sugar for $\neg x_1 \vee \neg x_2 \vee y_1 \vee y_2..$, and $x \longrightarrow \bigwedge y \in \{y_1,y_2,..\}.y$ is syntactic sugar for $(\neg x \vee y_1) \wedge (\neg x \vee y_2) \wedge ...$

Example 4. *Consider the STRIPS problem φ(FDR(P)) from Example 3. Let $h = 1$ be the horizon. The encoding is the conjunction of*

$$(v_0,0)^0 \wedge \neg(v_0,1)^0 \quad \text{(i)}$$

$$\neg(v_0,0)^1 \wedge (v_0,1)^1 \quad \text{(ii)}$$

$$(op_0^0 \longrightarrow (v_0,0)^0 \wedge \neg(v_0,1)^0) \wedge (op_0^0 \longrightarrow \neg(v_0,0)^1 \wedge (v_0,1)^1) \quad \text{(iii)}$$

$$(\neg(v_0,0)^0 \wedge (v_0,0)^1 \longrightarrow \bot) \wedge (\neg(v_0,1)^0 \wedge (v_0,1)^1 \longrightarrow op_0^0) \quad \text{(iv)}$$

$$((v_0,0)^0 \wedge \neg(v_0,0)^1 \longrightarrow op_0^0) \wedge ((v_0,1)^0 \wedge \neg(v_0,1)^1 \longrightarrow \bot) \quad \text{(v)}$$

$$\top \quad \text{(vi)}$$

Decoding 3. *Consider a horizon h, a STRIPS problem $\Pi = (V,R,I,O,G)$ and a model $\mathcal{M} \vDash \Phi(\Pi,h)$. Let, for a set s, list$(s)$ denote an arbitrary list which contains all the elements of s, s.t. $|s| = |list(s)|$. Let for a list of lists $ls = [l_0,l_1,..]$, flat(ls) denote the list $l_0 \frown l_1,..$, where $\frown$ is the list append function. The decoding function is defined $\Phi^{-1}(\Pi,h,\mathcal{M}) = flat([list(\{op \mid op^0 \in \mathcal{M}\}),list(\{op \mid op^1 \in \mathcal{M}\}),..,list(\{op \mid op^{h-1} \in \mathcal{M}\})])$.*

Example 5. *For the propositional formula in Example 4, a model is $\{(v_0,0)^0,\overline{(v_0,1)^0},\overline{(v_0,0)^1},(v_0,1)^1,op_0^0\}$. The decoded plan is $[op_0]$.*

This translation to SAT is sound and complete.

Theorem 3. *For a valid STRIPS problem Π and a horizon h:(i) if $\mathcal{M}$ is a model for $\Phi(\Pi, h)$, then $\Phi^{-1}(\Pi, h, \mathcal{M})$ is a solution for Π and (ii) if ops is a solution for Π and $|ops| \leq h$, then there is a model for $\Phi(\Pi, h)$.*

Proof sketch. (i) This proof is by induction on the horizon, with generalising the initial state. It depends, crucially, on the fact that non-interference implies that any order of operators coming from a single step is executable.
(ii) Let the operators in *ops* be $[op_0, op_1, .., op_{|ops|-1}]$. We construct a model for $\Phi(\Pi, h)$ by considering the sequence of states traversed by executing the plan π at I, which can be recursively specified as $s_0 = I$, and $s_{t+1} = op_t(s_t)$ for $0 < t \leq h$. The model we construct is

$$\{op_t^t \mid 0 \leq t < |ops|\} \cup \{\overline{op_t^{t'}} \mid 0 \leq t \neq t' < |ops|\} \cup \{s_t^t \mid 0 \leq t \leq |ops|\}$$

□

Before we conclude, we note that other encoding methods from FDR to SAT have also been proposed, e.g. Balyo 2013.

Abstract SAT Formulae to DIMACS

The SAT formulae produced by Encoding 3 are structured in the following way: (i) the formulae use the connectives $\wedge, \vee, \rightarrow$ and $\neg$ and (ii) atoms representing state variables and operators are indexed by the time step. In a pen-and-paper exposition this would be enough. However, this is not enough in our case because there is still an encoding step to simplify these structured formulae to DIMACS-CNF, which is the representation of SAT formulae used in practice, and we do not want to trust that step. Thus, as a last step, we present the following encoding of structured formulae to DIMACS-CNF ASTs. That encoding has to simplify the connectives as well as replace the structured variables with integers.

Definition 8 (DIMACS-CNF)**.** *A DIMACS-CNF AST is a list of lists of non-zero integers. A list of non-zero integers $[l_1, l_2, .., l_m]$ is a model for a DIMACS-CNF AST $[c_1 = [l_{11}, l_{12}..], c_2 = [l_{21}, l_{22}..], .., c_n = [l_{n1}, l_{n2}..]]$ iff for each c_i, for $1 \leq i \leq n$, there is $l_j \in c_i$, where $1 \leq j \leq m$, and, for each $1 \leq j, k \leq m$, $l_j \neq -l_k$.*

Encoding 4. *For a STRIPS problems (V, R, I, O, G), fix an arbitrary ordering $\mathbf{V}$ ($\mathbf{O}$) of V (O) s.t. for a state variable (operator) v (op), v_i (op_i) is the index of v (op) in $\mathbf{V}$ ($\mathbf{O}$). For an atom a and a horizon h, let int(a) be defined as: (i) $1 + t + op_i(h + 1)$, if $\exists op^t.\ a = op^t$ and (ii) $1 + |O|(h + 1) + t + v_i(h + 1)$ if $\exists v^t.\ a = v^t$. For a propositional formula ϕ that has no conjunctions, let simpORs(ϕ) be defined recursively as follows:(i) simpORs$(\phi_1)\frown$simpORs(ϕ_2), if $\exists \phi_1, \phi_2.\ \phi = \phi_1 \vee \phi_2$, (ii) $[]$, if $\phi = \bot$, (iii) $[-1, 1]$, if $\phi = \top$, (iv) $[-$simp$(\phi')]$, if $\exists \phi'.\ \phi = \neg\phi'$, and (v) $[$int$(\phi)]$, if ϕ is an atom. Let simp(ϕ) be defined recursively as follows:(i) simp$(\phi_1)\frown$simp(ϕ_2), if $\exists \phi_1, \phi_2.\ \phi = \phi_1 \wedge \phi_2$ and (ii) $[$simpORs$(\phi)]$ otherwise.*

Note: *int* and *var* are adapted from Knuth 1998, Section 4.4 on encoding numbers with arbitrary radixes. Also, note that *simp* is only well-defined if the given formula is a CNF, which is not problematic since the formulae produced by Φ are CNF for a valid STRIPS problem.

Example 6. *Consider the CNF formula from Example 4. First consider the following two orderings for the variables encoding operators and state variables: $[op_0]$ and $[(v_0, 0), (v_0, 1)]$. The mapping of the time indexed variables to natural numbers is: int$(op_0^0) = 1$, int$(op_0^1) = 2$, int$((v_0, 0)^0) = 3$, int$((v_0, 0)^1) = 4$, int$((v_0, 1)^0) = 5$, and int$((v_0, 1)^1) = 6$. Applying simp to that formula results in $[[3], [-5], [-4], [6], [-1, 3], [-1, -5], [-1, -4], [-1, 6], [3, -4], [5, -6, 1], [-3, 4, 1], [-5, 6], [-1, 1]]$. This AST is then pretty printed as DIMACS-CNF concrete syntax, like the one in Fig. 1, which only shows 7 clauses.*

Decoding 4. *Consider a DIMACS-CNF AST encoding a STRIPS problem (V, R, I, O, G) and a horizon h. Let, for an integer n, the function var(n) be defined as: (i) $(\mathbf{O}(|n|-1 \mod h))^{(|n|-1)\div h}$, if $|n| < 1+h|O|$ (ii) $(\mathbf{V}(k \mod h))^{k\div h}$, where $k = |n| - h|O| - 1$, otherwise. Now, let lit(n) be the literal var(n), if $0 < n$, and $\neg$var(n) otherwise.*

The correctness theorem for this step is as follows.

Theorem 4. *For a valid STRIPS problems Π and a horizon h: (i) if $\{l_1, l_2, ..\}$ is a model for $\Phi(\Pi, h)$, then $\{$simp(l_1), simp$(l_2), ..\}$ is a model for simp$(\Phi(\Pi, h))$, and (ii) if $[n_1, n_2, ..]$ is a model for simp$(\Phi(\Pi, h))$, then $\{$lit(n_1), lit$(n_2), ..\}$ is a model for $\Phi(\Pi, h)$.*

Proof sketch. The proof of both statements is by structural induction on the formula $\Phi(\Pi, h)$. □

.

Theorem 5. *For a valid FD-AST P and a horizon h: (i) if $[n_1, n_2, ..]$ is a model for simp$(\Phi(\varphi($FDR$(P)), h))$ then a plan for P is $[AST(\varphi_O^{-1}(op_1)), AST(\varphi_O^{-1}(op_2)), ..]$, where $[op_1, op_2, ..] = \Phi^{-1}(\Pi, h, \{lit(n_1)$, lit$(n_2), ..\})$, and (ii) if $[name_1, name_2, .., name_h]$ is a plan for P, then there is a model for simp$(\Phi(\varphi($FDR$(P)), h))$.*

Proof sketch. From Theorems 1, 2, 3 and 4. □

Experimental Evaluation

We use Isabelle/HOL's code generator to generate a Standard ML implementation of our correct encoding. Readers interested in implementation details can inspect the attachement. We evaluate the performance of our encoding compared to the $\forall$-step encoding of Rintanen, Heljanko, and Niemelä 2006 as computed by Madagascar when invariant generation is disabled. Although this setup has a weakness, namely, that Madagascar takes the PDDL domain as input while our system takes the grounded output of Fast Downward's translator, it should indicate the scalability of the verified encoding and can be used to test Madagascar's completeness. We compute the encodings of different planning domains from previous competitions and then feed them

	SAT		UNSAT	
	Mad.	Ver.	Mad.	Ver.
newopen	3128	2897	2214	2205
logistics	986	358	1031	452
elevators	75	14	61	44
rover	270	233	172	121
storage	66	35	41	33
pipesworld	46	9	74	7
nomystery	83	13	197	15
zeno	180	54	70	28
hiking	45	6	100	11
TPP	106	46	71	30
Transport	78	2	134	18
GED	68	19	105	19
woodworking	99	51	17	20
visitall	74	53	279	131
openstacks	62	—	316	62
satellite	46	34	20	20
scanalyzer	88	4	88	9
tidybot	24	—	12	—
trucks	45	8	117	29
parcprinter	113	63	128	111
maintenance	35	34	—	—
pegsol	114	3	200	237
blocksworld	31	25	30	30
floortile	221	54	361	281
barman	20	—	208	34
Thoughtful	—	—	—	5

Table 1: Number of solved satisfiable and unsatisfiable formulae solved by Kissat for our encoding (labelled Ver.) and the encoding generated by Madagascar (labelled Mad.).

to the SAT solver Kissat, opting for a 30 minutes timeout and a 8GB memory limit for encoding and solving. We generate the encodings for horizons 2, 5, 10, 20, 50, 100, and the bounds generated by the algorithm of Abdulaziz 2019. We record a few findings. Firstly, we found a bug in Madagascar: it produces unsatisfiable formulae for instances and horizons, despite those instances having ∀-step plans bounded by the horizon. This happens in at least 24 instances of different variants of the Rovers and PARCPrinter domains. This is because Madagascar adds incorrect action mutex constraints which rule out valid ∀-step plans, thus causing Madagascar to not be complete. The fact that such a well-established planning system has such bugs demonstrate that it is imperative we verify planning systems, especially that there no generally succinct unsolvablity certificates do not exist for AI planning algorithms.

Secondly, we compare the performances of our encoding and Madagascar. (i) For most instances, our encoding is solved by Kissat in significantly shorter time than Madagascar. Kissat fails to terminate on Madagascar encodings of some Rovers instances, while it succeeds for our encodings. We note that Madagascar's ∀-step encoding is linear in size due to the use of auxiliary variables to represent the operator interference clauses, while ours is quadratic. A hypothesis is that auxiliary variables interfere with Kissat's deduction mechanisms, as has been reported about compact encodings in other contexts (Knuth 2015)[Section 7.2.2.2]. However, this needs further study. (ii) Since our verified implementation is purely functional in Standard ML, computing our encoding takes longer time than Madagascar's, e.g. we use balanced trees instead of arrays, causing every access/update to be worst-case logarithmic instead of constant time. This leads to our encoding to have a worse total (i.e. grounding, encoding and solving) running time, despite the fact that our encoding is usually solvable in shorter time. A bigger problem is that, as Standard ML does not support lazy evaluation and has poor memory management in general, our encoding frequently runs out of memory as it computes the entire encoding in memory before producing any output. This leads to less of our encodings being solved by Kissat compared to the ones produced by Madagascar (see Table 1).

Discussion

We presented an executable formally verified SAT encoding of AI planning. We showed details of the verification process, and experimentally tested our encoding. Experiments show that, although our verified encoder is primarily hindered by its memory consumption, it can handle planning problems of reasonable sizes, where it can solve, or show bounded length plan non-existence. By testing Madagascar's encoding against our verified encoding, we discovered that Madagascar sometimes mistakenly claims that problems have no solutions of a certain length. Also, compared to Madagascar, our encoding can be more efficiently processed by the SAT solver Kissat. The size of the verified Standard ML program is around 1.2K lines of code, and the size of the formal proof is around 17.5K lines of proof scripts.

One goal of our work here is to showcase theorem proving and its application to verification as a methodology to increase trustworthiness of planning software and, more generally, AI software. Although there are other approaches to increase reliability of AI systems, most notably certification for planning or SMT-based methods for verifying properties of neural networks, we believe that correct-by-construction algorithms have their niche. For instance, this is the case when there are not general certification methods or when desired formal properties are too complex for automated methods to practically handle.

As future work, the most interesting direction is to verify the encoding of costs by Abdulaziz 2021, yielding formally verified certificates of cost optimality. Another direction is optimising the memory consumption of our implementation, e.g. via lazy evaluation (Lochbihler and Stoop 2018), or by using a low-level target language instead of Standard ML, like LLVM (Lammich 2022).

References

Abdulaziz, M. 2019. Plan-Length Bounds: Beyond 1-Way Dependency. In *The 33rd AAAI Conference on Artificial Intelligence (AAAI)*.

Abdulaziz, M. 2021. Cost Optimal Planning as Satisfiability. In *The Workshop on Heuristics and Search for Domain-Independent Planning (HSDIP)*.

Abdulaziz, M.; and Koller, L. 2022. Formal Semantics and Formally Verified Validation for Temporal Planning. In *The 36th AAAI Conference on Artificial Intelligence (AAAI)*.

Abdulaziz, M.; and Lammich, P. 2018. A Formally Verified Validator for Classical Planning Problems and Solutions. In *The 30th International Conference on Tools with Artificial Intelligence (ICTAI)*.

Abdulaziz, M.; Mehlhorn, K.; and Nipkow, T. 2019. Trustworthy Graph Algorithms (Invited Paper). In *The 44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*.

Abdulaziz, M.; Norrish, M.; and Gretton, C. 2018. Formally Verified Algorithms for Upper-Bounding State Space Diameters. *J. Autom. Reason.*

Arora, S.; and Barak, B. 2009. *Computational Complexity - A Modern Approach*.

Balyo, T. 2013. Relaxing the Relaxed Exist-Step Parallel Planning Semantics. In *25th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*.

Blanchette, J. C.; Fleury, M.; Lammich, P.; and Weidenbach, C. 2018. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. *J. Autom. Reason.*

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.*

Eén, N.; and Sörensson, N. 2003. An Extensible SAT-solver. In *Internation Conference on Theory and Applications of Satisfiability Testing (SAT)*.

Eriksson, S.; and Helmert, M. 2020. Certified Unsolvability for SAT Planning with Property Directed Reachability. In *The 30th International Conference on Automated Planning and Scheduling (ICAPS)*.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability Certificates for Classical Planning. In *The 27th International Conference on Automated Planning and Scheduling (ICAPS)*.

Esparza, J.; Lammich, P.; Neumann, R.; Nipkow, T.; Schimpf, A.; and Smaus, J.-G. 2013. A Fully Verified Executable LTL Model Checker. In *25th International Conference on Computer Aided Verification (CAV)*.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*

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)*.

Kanav, S.; Lammich, P.; and Popescu, A. 2014. A Conference Management System with Verified Document Confidentiality. In *The 26th International Conference on Computer Aided Verification (CAV)*.

Kautz, H. A.; and Selman, B. 1992. Planning as Satisfiability. In *10th European Conference on Artificial Intelligence (ECAI)*.

Kautz, H. A.; and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. In *13th National Conference on Artificial Intelligence (NCAI)*.

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S. 2009. seL4: Formal Verification of an OS Kernel. In *22nd ACM Symposium on Operating Systems Principles 2009 (SOSP)*.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition*.

Knuth, D. E. 2015. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*.

Kumar, R.; Myreen, M. O.; Norrish, M.; and Owens, S. 2014. CakeML: A Verified Implementation of ML. In *The 41st Symposium on Principles of Programming Languages (POPL)*.

Lammich, P. 2017. The GRAT Tool Chain - Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees. In *SAT*.

Lammich, P. 2022. Refinement of Parallel Algorithms down to LLVM. In *13th International Conference on Interactive Theorem Proving (ITP)*.

Leroy, X. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM*.

Lochbihler, A.; and Stoop, P. 2018. Lazy Algebraic Types in Isabelle/HOL. In *Isabelle Workshop*.

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.*

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

Mehlhorn, K.; and Näher, S. 1998. From Algorithms to Working Programs: On the Use of Program Checking in LEDA. In *23rd International Symposium Mathematical Foundations of Computer Science (MFCS)*.

Michaelis, J.; and Nipkow, T. 2018. Formalized Proof Systems for Propositional Logic. In *The 23rd International Conference on Types for Proofs and Programs (TYPES)*.

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*.

Rintanen, J. 2003. Symmetry Reduction for SAT Representations of Transition Systems. In *The 13th International Conference on Automated Planning and Scheduling (ICAPS)*.

Rintanen, J. 2012. Planning as Satisfiability: Heuristics. *Artif. Intell.*

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

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.*

Wetzler, N.; Heule, M.; and Jr, W. A. H. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*.