bni: A PDDL to C compiler with integrated REPL for interactive testing

Bruno Campos Ribeiro , Igor e Silva Penha , Bruno Cesar Ribas

[1]University of Brasília, Brasília, DF – Brazil

bbrunoo@icloud.com, igorpenha.it@gmail.com, bruno.ribas@unb.br

Abstract. *This paper presents the implementation of a modular parser and an interactive REPL PDDL, entirely developed in the C programming language. The proposed solution enables the translation of PDDL domain and problem specifications into efficient, manipulable C data structures, fostering transparency, performance, and portability. Complementing the parser, the integrated REPL facilitates incremental testing and interactive debugging of planning domains and problem instances, supporting a cycle of real-time action validation and state exploration. The system also incorporates an integrated validation functionality. This feature enables both incremental and* post hoc *verification of action sequences and complete plans, ensuring that they adhere to the defined domain and problem specifications. The validate capability of the tool allows users to detect logical inconsistencies and confirm goal achievement efficiently, even in large-scale instances. Compared to established tools such as VAL, the proposed solution offers improved flexibility and performance for plan validation, particularly in scenarios involving extensive and complex plans. This work represents a novel contribution to the field of automated planning, bridging the gap between high-level modelling languages and low-level systems programming.*

Keywords *Artificial Intelligence, Automated Planning, PDDL.*

1. Introduction

Automated planning is a well-established branch of Artificial Intelligence, focused on devising solutions to complex problems through autonomous decision making. Among the most prominent tools for modelling such problems is the Planning Domain Definition Language (PDDL) [Aeronautiques et al. 1998], whose high-level abstraction allows a concise representation of fundamental domain components. Its adoption was largely consolidated through the International Planning Competition [IPC 2023], held within the International Conference on Automated Planning and Scheduling [ICAPS 2025], which established PDDL as a standard for evaluating planning systems.

This work presents the implementation of a compiler that translates PDDL into manipulable structures in the C programming language [Kernighan and Ritchie 1988], thus promoting greater transparency and control over domain and problem elements. The choice of C is motivated by its efficiency and predictable performance.

Complementing the parser, we developed an interactive Read-Eval-Print Loop (REPL) environment that allows users to load domain and problem files, observe internal structures, test actions, and get an effect of the action in real time. This approach

facilitates incremental verification of domain logic and supports iterative debugging, significantly accelerating the development cycle.

The designed solution includes the parsing and structuring of several key elements of the PDDL language: objects, constants, predicates and actions, as well as the interpretation of operators and universal quantifiers (i.e., and, or, not, forall). The system also supports the loading and execution of problem instances, enabling complete interactive simulations.

To demonstrate the practical applicability of the developed tool, we present a simplified PDDL domain *bus-line*, at listing 1 and 2 featuring typical object, predicate, and action definitions drawn from classical planning scenarios. This example is used throughout the following sections to illustrate the operation of the parser, the internal structure of the extracted elements, and the interactive capabilities provided by the REPL environment.

The selected domain models a simplified transportation scenario, where in an agent navigates between stops and interacts with a specific location the ICE-CREAM-PARLOUR to acquire ice cream. Defined actions include movement between connected stops and purchasing available items at appropriate locations, which are marked by specific preconditions and effects that alter the state of the system.

The goal of this study is therefore the construction of a modular PDDL parser in C, accompanied by a functional REPL interface integrated with functional readline library tools [Project 2024]. This solution aims to bridge the gap between high-level academic tools and low-level implementations capable of offering transparency, performance, and portability with additional potential for educational applications.

The structure of this paper is as follows. Section 2 begins by discussing topics related to the scope of the research. In Section 3 it presents the technical context of the parser implementation, addressing elements such as the definition of objects, constants, predicates, actions, initial conditions and goal, as well as the handling of universal quantifiers. Section 4 describes the architecture of the developed REPL interface, including its main auxiliary functions. Finaly, shows its performance in comparison to the Validate tool VAL [Howey et al. 2004].

Listing 1. *bus-line* domain in PDDL

```
(define (domain bus-line)
  (:requirements :strips :typing )
  (:types stop ice-cream)
  (:constants ICE-CREAM-PARLOUR - stop
              vanilla chocolate strawberry - ice-cream)
  (:predicates (connected ?s1 ?s2 - stop)
               (i-am-at ?s - stop)
               (order-ice-cream ?i - ice-cream)
               (has-ice-cream ?i - ice-cream)
               (stop-is ?s1 ?s2 - stop))
  (:action TRAVEL
    :parameters (?s1 ?s2 - stop)
    :precondition (and (i-am-at ?s1) (or (connected ?s1 ?s2) (connected ?s2 ?s1)))
    :effect (and (not (stop-is ?s1 ?s1))
                 (not (i-am-at ?s1)) (stop-is ?s2 ?s2) (i-am-at ?s2)))
  (:action BUY-ICE-CREAM
    :parameters (?s - stop ?i - ice-cream)
    :precondition (and (stop-is ?s ICE-CREAM-PARLOUR) (has-ice-cream ?i))
    :effect (and (order-ice-cream ?i) (not (has-ice-cream ?i)))))
```

Listing 2. *bus-line* problem in PDDL

```
(define (problem bus-line-ice-cream)
  (:domain bus-line)
  ;; Problem with first, second and third stops.
  (:objects f-stop s-stop t-stop - stop)
  (:init (connected f-stop s-stop)
         (connected s-stop t-stop)
         (connected t-stop ICE-CREAM-PARLOUR)
         (i-am-at f-stop)
         (has-ice-cream vanilla)
         (has-ice-cream chocolate)
         (has-ice-cream strawberry))
  (:goal (forall (?i - ice-cream) (not (has-ice-cream ?i)))))
```

2. Related Work

Among the most prominent efforts, the Fast Downward planning system [Helmert 2006] provides robust parsing and heuristic search capabilities for parsing and manipulating PDDL files, albeit primarily geared towards static analysis rather than dynamic, interactive use. However, it is primarily implemented in C++ and Python, focusing on efficient plan generation rather than facilitating interactive exploration or low-level manipulation of PDDL structures.

Recent advancements have explored the integration of planning tools into high-level programming environments. For instance, the PDDL.jl library [Zhi-Xuan 2022] offers parsing and modeling capabilities within the Julia programming ecosystem, leveraging its strengths in scientific computing but abstracting away from low-level control and efficiency concerns typical of system programming languages like C. Likewise, AI planners such as SATPlan [Kautz et al. 2006], LPG [Gerevini and Serina 2002] and Madagascar [Rintanen 2014] focus on translating PDDL models into SAT encodings or heuristic search problems, optimizing for plan synthesis performance rather than interactive model exploration.

The integration of PDDL parsing and REPL environments remains relatively underexplored. Some environments, like WebPlanner [Magnaguagno et al. 2020] and PDDL Editor [Muise 2015], provide graphical interfaces for plan visualization but do not offer interactive manipulation of PDDL model elements or real-time simulation capabilities.

In contrast, interactive theorem provers such as Coq [Yves Bertot 2015] and Lean [de Moura et al. 2015] provide sophisticated REPL-based environments that enable formal verification of complex mathematical properties through stepwise proof construction. Analogously, in the domain of automated planning, the VAL tool [Howey et al. 2004] has been developed as a widely adopted plan validator for PDDL-based models. VAL performs the formal verification of generated plans by ensuring that they satisfy the preconditions and effects specified in the domain and problem definitions. However, while VAL focuses on the *post hoc* validation of entire plans, it does not support interactive, incremental validation or direct exploration of partial plans and state transitions during the modeling process.

The present work differentiates itself by proposing a modular parser and REPL entirely implemented in C, aiming to bridge the gap between the high-level abstraction of PDDL and the transparency, efficiency, and control afforded by low-level programming.

Beyond serving as a parser and interactive simulator, *bni* (source code is available at our repository[1]) also functions as a plan validator, enabling users to experiment with and validate action sequences incrementally, or in *post hoc* validation of entire plans. To the best of our knowledge, this is the first PDDL parser designed explicitly for direct manipulation within a REPL environment in C, allowing not only structural inspection of PDDL elements but also dynamic testing of actions, state transitions, and interactive debugging within a single integrated system.

3. Crafting the PDDL Parser

The PDDL was introduced in 1998 as a standard formalism for representing planning domains and problems in Automated Planning and Scheduling systems. Its first version, PDDL 1.0 [Aeronautiques et al. 1998], established the foundational constructs for modeling: *types*, *objects*, *predicates*, *actions*, *preconditions*, and *effects*, using a Lisp-like syntax.

Subsequent versions progressively extended PDDL's expressive power. PDDL 2.1 [Fox and Long 2003] introduced support for *temporal planning* and *numeric fluents*, enabling the representation of actions with durations and continuous effects. Later versions, such as PDDL 2.2 [Edelkamp 2004], incorporated *derived predicates* and *timed initial literals*, while PDDL 3.0 [Gerevini and Long 2005] extended the language with *preferences* and *soft constraints*, allowing more nuanced problem specifications.

Despite these advancements, the core elements of PDDL — namely *objects*, *constants*, *predicates*, and *actions* with *preconditions* and *effects* — remain consistent across versions and constitute the fundamental layer upon which most planners operate. These basic constructs are sufficient to model a vast range of classical planning problems, which are the primary focus of this work.

This paper concentrates on parsing and processing a substantial subset of PDDL 1.0, covering its core syntactic and semantic features. Specifically, the developed parser supports the definition and management of *types*, *constants*, and *objects*, ensuring correct categorization and association, process of *predicates*, translation of *action schemas*, capturing *preconditions* and *effects*, and handling logical connectors such as `and`, `or`, `not`, and the *universal quantifier* `forall`, initialization of problem instances through parsing of the *init* section and the verification of *goal* conditions based on structured logical expressions.

By focusing on PDDL 1.0 constructs and core logical operators, we prioritize a modular and transparent parser design, facilitating structural inspection, incremental validation, and interactive manipulation within the developed REPL environment. This deliberate scope ensures that the tool remains accessible, while also serving as a foundational platform for future extensions toward more advanced PDDL features such as *numeric fluents*, *durative actions* and *conditional-effects*.

Thus, the *bni* parser serves both as a practical implementation of PDDL model processing and as a pedagogical tool, making the internal structures of planning domains and problems explicit and manipulable at runtime. In the following subsections, we detail

[1]https://github.com/UnB-SAT/bni

the design and implementation strategies for handling each PDDL construct within this scope.

3.1. Objects and Constants

The sections `:objects` and `:constants` of PDDL are converted into specific C structures to facilitate their use in the problem domain. During this conversion, objects and constants are transformed into enumerations and mapping structures.

Each object or constant from PDDL is translated into a typed enumeration in C, ensuring that each entity can be uniquely identified in the generated code. Additionally, a mapping structure is created to associate the original string representations of the objects/constants with their respective enumerations.

The following examples illustrate how data structures representing objects of type *stop* from the *bus line* domain are initialised in the C code. This representation is applicable to both objects and constants. These examples ensure that the logic and accessibility needed during programme execution are maintained, as illustrated in Listings 3 and 4.

Listing 3. Constants and objects representation in .h file

```c
enum t_stop {
    f_stop,
    s_stop,
    t_stop,
    LENGTH_stop
};
typedef struct stopMap {
    const char *str;
    enum t_stop value;
}stopMap;
```

Listing 4. Constants and objects representation in .c file

```c
stopMap stop_map[LENGTH_stop] = {
    {"f_stop", f_stop},
    {"s_stop", s_stop},
    {"t_stop", t_stop},
};
const char* get_stop_names(enum t_stop e) {
    if (e >= 0 && e < LENGTH_stop)
        return stop_map[e].str;
    return NULL;
}
enum t_stop get_stop_enum(const char *s) {
    if (s == NULL) return LENGTH_stop;
    for (int i = 0; i != LENGTH_stop; i++)
        if (strcmp(s, stop_map[i].str) == 0)
            return stop_map[i].value;
    return LENGTH_stop;
}
```

3.2. Predicates

In the `:predicates` section of PDDL, predicates are converted into C data structures to support their functionalities in the problem domain. During this conversion, predicates are represented as boolean arrays that record the truth state of each predicate.

Each predicate in PDDL is mapped to a boolean array in C, allowing the representation of specific combinations of entities and establishing the basis for logical operations. For example, predicates with a single parameter become one-dimensional arrays, whilst

predicates that relate two parameters are translated into two-dimensional arrays, and so forth.

Below, in the Listing 5, the translation of predicates into the C code is presented, showing the resulting structure and how each predicate and its parameters are stored for efficient manipulation within the programme.

Listing 5. Translation :predicates to C

```
bool connected[4][4];
bool i_am_at[4];
bool passed_through[4];
bool order_ice_cream[7];
bool has_ice_cream[7];
bool stop_is[4][4];
bool checktrue_connected(int s1, int s2) {
    return connected[s1][s2];
}
bool checktrue_i_am_at(int s) {
    return i_am_at[s];
}
bool checktrue_order_ice_cream(int i) {
    return order_ice_cream[i];
}
bool checktrue_has_ice_cream(int i) {
    return has_ice_cream[i];
}
bool checktrue_stop_is(int s1, int s2) {
    return stop_is[s1][s2];
}
```

3.3. Actions

The `:action` section in PDDL is converted into C code by interpreting the parameters, preconditions, and effects of each action, ensuring that the planning logic is preserved and executable. Below is a detailed explanation of the translation processes for each subsection.

3.3.1. Parameters

In PDDL, `:parameters` for actions are declared with specific types and variables. These are transformed into C `structs`, where each struct member represents a parameter. The `struct` defines the action's parameters, using `enums` to restrict values to predefined types, providing a structured way to store and manipulate action parameters.

Listing 6 presents the conversion of the `:parameters` section into C code, illustrating how parameters are structured and stored. This organisation offers the flexibility for manipulation during programme execution while maintaining the integrity of the planning logic.

Listing 6. Translation :parameters to C

```
struct travel {
    enum stop s2;
    enum stop s1;
};
struct buy_ice_cream {
    enum stop s;
    enum ice_cream i;
};
```

3.3.2. Preconditions

The `:precondition` section in each `:action` specifies the logical conditions that must be satisfied for the action to be executed. These conditions are translated into boolean functions in C, named `check_true_<action>`, which verify whether the preconditions hold in the current state of the world. These functions ensure adherence to the model's rules by confirming that all prerequisites for an action are met.

Listing 7 demonstrates the translation of the `:precondition` section for actions such as `TRAVEL` and `BUY-ICE-CREAM` into C code. This demonstrates how preconditions are structured and indexed, ready for verification before action execution, efficiently meeting the model's requirements.

Listing 7. Translation :precondition to C

```
bool checktrue_travel(struct travel s) {
    return (checktrue_i_am_at(s.s1) and (checktrue_connected(s.s1, s.s2) or
        checktrue_connected(s.s2, s.s1)));
}
bool checktrue_buy_ice_cream(struct buy_ice_cream s) {
    return (checktrue_stop_is(s.s, ice_cream_parlour) and checktrue_has_ice_cream(s.i
        ));
}
```

3.3.3. Effects

The `:effect` section outlines the consequences of executing an action, involving updates to the model's state. In C, these effects are translated into functions named `apply_<action>`, which directly modify the current state to reflect the outcome of the action. These functions correctly apply and manage the changes to the state resulting from an action's execution.

Listing 8 shows the translation of the `:effect` section into C code. This demonstrates how each effect is stored for manipulation during programme execution, ensuring that the action's behaviour is accurately applied to the model's state.

Listing 8. Translation :effect to C

```
void apply_travel(struct travel s) {
    i_am_at[s.s2] = 1;
    i_am_at[s.s1] = 0;
    stop_is[s.s1][s.s1] = 0
    stop_is[s.s2][s.s2] = 1
}
void apply_buy_ice_cream(struct buy_ice_cream s) {
    order_ice_cream[s.i] = 1;
    has_ice_cream[s.i] = 0;
}
```

3.4. Init

The `:init` section in PDDL defines the initial state of the problem instance by listing the facts that are true from the beginning. The *bni* parser translates this into C code, ensuring accurate initialisation within the programme's runtime environment.

During this process, each true fact is converted into corresponding entries in C array structures representing predicates. These predicates, previously stored as boolean

arrays, are initialised to `true` for the specific facts, as demonstrated in earlier sections. This method preserves the state representation established previously.

The parser generates an `initialize()` function in C, which is tasked with setting up the initial state. Declared in both the implementation and header files, this function configures the initial state by setting specific elements of the predicate arrays to `true`, thereby reflecting the initial truth of each fact. This is exemplified in Listing 9, showing how these mapped arrays bring the PDDL's initial conditions to life in C code.

Listing 9. Translation :init to C

```
void initialize(void) {
    connected[f_stop][s_stop] = true;
    connected[s_stop][t_stop] = true;
    connected[t_stop][ice_cream_parlour] = true;
    i_am_at[f_stop] = true;
    has_ice_cream[vanilla] = true;
    has_ice_cream[chocolate] = true;
    has_ice_cream[strawberry] = true;
}
```

3.5. Goal

The `:goal` section in PDDL specifies the desired end state or conditions that the problem-solving process aims to achieve. In C, this section is translated into a Boolean function that checks whether these goal conditions are met within the programme's current state.

The *bni* parser converts the `:goal` block into a `checktrue_goal()` function in C. This function evaluates the conditions specified in the goal using previously defined arrays and helper functions that represent the state of predicates.

Each goal condition is expressed as a logical expression, where the predicates are checked using the corresponding functions to determine if they satisfy the desired outcome. The function returns `true` if all conditions are met, indicating that the goal state has been achieved. This translation ensures that the problem's objectives are clearly defined and verifiable at different stages of programme execution, as shown in Listing 10.

3.6. Universal Quantifiers

The forall quantifier in PDDL, used to express that a condition must hold for all objects of a given type, has been systematically translated into equivalent constructs in the C language. This translation enables the expression of universal quantification within action descriptions, specifically in the `:precondition`, `:goal`, and `:effect` sections.

With the intention of convert PDDL's declarative `forall` expressions into imperative C structures that iterate over all relevant objects. The implementation ensures semantic fidelity by preserving logical structure, managing variable scopes, and supporting nested quantifiers. The resulting C code in Listing 10 mirrors the original intent of the PDDL specification while enabling direct execution in a compiled environment.

Listing 10. Translation :goal to C

```
bool checktrue_goal(void) {
  bool forall1 = true;
  for (int i0 = 0; i0 < LENGTH_ice_cream; i0++) {
```

```
4       if (!!(checktrue_has_ice_cream(i0))) { forall1 = false; break; }
5     }
6     return (forall1);
7   }
```

4. REPL

The complexity of PDDL poses a significant challenge for beginners in automated planning. In this context, the adoption of a REPL-based interface emerges as a promising solution to simplify interactions with planning models by enabling iterative, responsive, and incremental testing and validation of actions.

The REPL developed in this project integrates with the *bni* parser, allowing users to execute plans, observe state transitions of the system, and iterate on decisions based on immediate feedback. This approach reduces development time, facilitates debugging, and promotes active learning [van Binsbergen et al. 2020].

Beyond plan execution, the REPL empowers users to interactively manipulate the problem world by invoking any available action, provided its preconditions are satisfied in the current state. This enables direct and flexible exploration of the model, helping users better understand domain dynamics and verify the expected effects of each action.

This interactivity transforms the REPL into more than just a validation tool, it becomes a powerful environment for controlled experimentation. Users can simulate partial plans, test corner cases, and analyze specific state transitions without executing full solutions. Such capabilities are especially valuable during early domain modeling, where exploring the consequences of isolated actions is essential for refining both syntax and behavior.

The implemented REPL provides command history support, allowing users to quickly reuse and revise previously entered commands. Additionally, it features an autocomplete mechanism that suggests available actions based on the current planning state, enhancing efficiency, and reducing input errors.

4.1. Supportive Functions

The interactive capability of the REPL is supported by a set of auxiliary functions that enable communication between the plannning model translated into C and the current state of the simulated environment. The functions `check_show_actions()` and `apply_action()` has a important role in identifying valid and conditionally executing them based on the domain's logic.

The (`check_show_action()`) function iterates over all possible actions defined in the domain and, for each admissible parameter combination, verifies whether the action is executable in the current state. This verification is delegated to action-specific predicate-checking functions, which are automatically generated during the PDDL-to-C translation.

Complementarily, `apply_actions()` interprets the user input from the REPL prompt and if the action is valid triggers its execution through the respect `apply_action()` function. If the action is invalid an error code is returned, notifying the REPL of the failed execution attempt.

5. Empirical Evaluation

The *bni* system incorporates built-in support for plan validation, which validates whether a given sequence of actions correctly transitions the system from the initial state to a goal state while respecting all specified preconditions and effects.

Two experiments were conducted to assess performance, robustness, and correctness. The first increased plan complexity using large-scale *bus-line* instances, while the second tested compatibility with IPC-1998 and IPC-2000 domains. Metrics collected were CPU time and memory usage. All experiments ran on a high-performance server with two Intel Xeon E5-2680 v3 CPUs (12 cores/24 threads each) and 768 GB RAM, compiling all code with `gcc -O3`.

In *bni*, each validation generates a domain and instance specific binary from translated C code. The plan validation was compared with VAL [Howey et al. 2004], the standard IPC validator. While VAL is domain-independent and compiled only once, *bni* requires compilation per domain. However, once compiled, *bni* validates plans faster. Although VAL offers broad support for PDDL, it failed to process plans exceeding 10,000 steps, as observed in several large-scale instances. In such cases, validation was completed exclusively with *bni*, as summarized in Table 1.

The first test stresses the validator's scalability by gradually increasing the number of objects in the *bus-line* domain, resulting in plans with thousands of steps. This setting is particularly useful for revealing bottlenecks related to compilation, memory, or validation time.

The second experiment evaluated compatibility with diverse PDDL constructs using benchmark domains from IPC-1998 and IPC-2000. Plans were generated by the *fast-downward-fdss-2023* planner [Büchner et al. 2023], configured with the `lama-first` alias, a well-established setup for generating fast plans.

In this benchmark setting, both *bni* and VAL were used to validate the same set of plans. As shown in Table 2, the tools achieved comparable CPU times, demonstrating that *bni* performs on par with VAL, once it is already compiled, while maintaining full compatibility with standard PDDL formats. The reported times represent cumulative validation time across all instances. Furthermore, these plans are significantly smaller than those in the previous experiment (Table 1), with the largest comprising only 827 steps.

Table 1. CPU and memory comparison between *bni* and VAL

Number of Steps	bni			VAL	
	Compile (s)	CPU (s)	RAM (MB)	CPU (s)	RAM (MB)
100	0.301	0.00	1.536	0.01	6.144
1,000	0.934	0.01	3.456	0.02	8.064
5,000	4.129	0.15	41.472	0.11	16.512
7,000	5.581	0.27	57.984	0.16	20.352
10,000	7.885	0.49	82.176	-	-
30,000	23.099	3.46	243.072	-	-
40,000	30.631	5.75	323.712	-	-

Table 2. Performance of *bni* and VAL on IPC 1998/2000 domains

Domain	Instances	bni			VAL	
		Compile (s)	CPU (s)	RAM (MB)	CPU (s)	RAM (MB)
grid-round-2-strips	5	2.48	0.00	7.680	0.05	32.000
gripper-round-1-adl	20	6.53	0.00	29.952	0.18	124.544
gripper-round-1-strips	20	7.15	0.01	30.336	0.16	123.136
logistics-round-1-strips	35	18.35	0.02	52.224	0.38	222.464
logistics-round-2-strips	5	2.15	0.00	7.680	0.05	30.720
mystery-round-1-strips	19	8.13	0.01	29.184	0.19	116.224
blocks-strips-typed	92	32.51	0.02	140.160	0.88	580.480
blocks-strips-untyped	92	32.43	0.02	277.632	0.89	579.840

6. Conclusion

This work presents the implementation of a PDDL language parser in C, interrelated with an interactive REPL-based interface. The proposed solution aims to provide an efficient and extensible alternative for handling and experimenting with planning domains and problems, adhering to the principles of reproducibility and modularity that support the development of tools in the field of Intelligent Systems.

The parser offers comprehensive support for the core constructs of the PDDL language. The REPL interface improves the system by enabling interactive exploration of planning domains, which has valuable for educational purposes, debugging, and rapid prototyping of planning problems.

The primary contributions of this work include demonstrating the viability of using the C programming language as a foundation for PDDL manipulation, and introducing a functional REPL shaped for planning tasks. This constitutes a practical innovation with potential impact on the adoption and experimentation of PDDL domains, particularly in constrained or resource-limited contexts.

The main contributions of this research include: (i) demonstrating the feasibility of using the C programming language as a foundation for manipulating PDDL models, combining performance and portability; (ii) introducing a functional REPL that enables incremental testing and interactive debugging of actions and states; and (iii) the empirical validation of the system, evidencing its efficiency in comparison with established tools, particularly in scenarios involving large-scale instances.

As future work, we intend to extend the tool's support to more advanced versions of PDDL, incorporating features such as *conditional-effects*, *durative-actions*, as well as enhancing the interactive interface with new visualisation resources and automated testing capabilities.

References

Aeronautiques, C., Howe, A., Knoblock, C., McDermott, I. D., Ram, A., Veloso, M., Weld, D., Sri, D. W., Barrett, A., Christianson, D., et al. (1998). Pddl — the planning domain definition language. *Technical Report, Tech. Rep.*

Büchner, C., Christen, R., Corrêa, A. B., Eriksson, S., Ferber, P., Seipp, J., and Sievers, S. (2023). Fast downward stone soup 2023. Prague, Czech Republic.

de Moura, L., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. (2015). The lean theorem prover (system description). In *2015 Conference on Automated Deduction*, pages 378–388. Springer, Cham.

Edelkamp, S. (2004). Pddl2. 2: The language for the classical part of the 4th international planning competition.

Fox, M. and Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. 20:61–124.

Gerevini, A. and Long, D. (2005). Plan constraints and preferences in pddl3 the language of the fifth international planning competition.

Gerevini, A. and Serina, I. (2002). Lpg: A planner based on local search for planning graphs with action costs. *AIPS: 6th International Conference on Artificial Intelligence Planning Systems*.

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*. Available at https://www.jair.org/index.php/jair/article/view/10457.

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*, pages 294–301.

ICAPS (2025). International conference on automated planning and scheduling. Available at https://icaps25.icaps-conference.org/.

IPC (2023). International planning competition. Available at https://ipc2023-classical.github.io/.

Kautz, H., Selman, B., and Hoffmann, J. (2006). Satplan: Planning as satisfiability. *5th International Planning Competition, Cumbria, UK*.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. 2nd edition.

Magnaguagno, M. C., Pereira, R. F., Móre, M. D., and Meneguzzi, F. (2020). *Web Planner: A Tool to Develop, Visualize, and Test Classical Planning Domains*, pages 209–227. Springer International Publishing, Cham.

Muise, C. (2015). Pddl editor. https://editor.planning.domains/.

Project, G. (2024). *GNU Readline Library*. Version used: 8.2. Available at https://tiswww.case.edu/php/chet/readline/rltop.html.

Rintanen, J. (2014). Madagascar: Scalable planning with sat. In *Proceedings of the International Planning Competition (IPC) 2014*.

van Binsbergen, L. T., Verano Merino, M., Jeanjean, P., van der Storm, T., Combemale, B., and Barais, O. (2020). A principled approach to repl interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, page 84–100, New York, NY, USA. Association for Computing Machinery.

Yves Bertot, P. C. (2015). *Le Coq' Art*. Available at https://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf.

Zhi-Xuan, T. (2022). Pddl.jl: An extensible interpreter and compiler interface for fast and flexible ai planning. Ms thesis, Massachusetts Institute of Technology.