The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Engineering, Computing and Cybernetics (CECC)

Domain Modelling Assistance in AI Planning

— 4 pt research project (S1 2023)

A report submitted for the course
ENGN2706, Engineering Research Methods

By:
Kayleigh Sleath
u7284920

Supervisor:
Dr. Pascal Bercher

May 2023

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

May, Kayleigh Sleath

Acknowledgements

Over a very short time, a lot of people have helped me with this project.

First and foremost, I would like to thank Dr. Pascal Bercher for taking on the role of supervisor in this project. He has given an incredible amount of feedback, patience and good humour over the course of this semester, and I've learnt a huge amount working with him. Thank you for your high expectations and eye for detail!

Additional thanks to Dr. Bernd Schattenberg for spending many fascinating hours chatting with me about AI planning. Your time is very much appreciated.

Thank you to Thomas Willingham and Gregor Behnke for almost immediately fixing the compiler errors that I battled for weeks. Thank you to Yikai Ge and Yuchen Fang for being so happy to share their experiences with domain modelling.

Finally, thank you to Prof. Dan MacDonald for reading the weekly reflections which became my personal form of therapy, and to the RnD fam for being such a supportive group of peers. Particular thanks to everyone who took the time to provide me with peer feedback.

Abstract

Breakthroughs in domain-independent planners have made AI planning more efficient and effective. This leaves domain creation as one of the hardest and most error-prone aspects of the AI planning process. Unfortunately, domain modelling assistance software is relatively new and therefore has underdeveloped error detection and diagnosis mechanisms. These need to be more robust before domain-independent planners are likely to cross into mainstream use. This paper details a list of potential domain errors and corresponding repository of flawed domains, which are intended to form the basis for a standardised set of tests for domain modelling assistance software. Additionally, it applies these test cases in an evaluation of currently notable domain modelling assistance software and identifies areas requiring improvement.

Table of Contents

Chapter 1

Introduction

AI planning is a branch of artificial intelligence particularly suited to applications requiring dependable decision-making capabilities. Unlike other AI approaches such as machine learning or language models, AI planning employs algorithmic processes to generate a collection of solutions for a given problem. This makes it transparent and deterministic, and hence a more reliable choice of AI than techniques which rely on probability or data-driven training. It has a variety of applications including urban planning (Vallati et al., 2019), robotics (Beetz et al., 2012; Stock et al., 2015; González et al., 2017), scheduling (Barták and Vlk, 2016), and web service composition (Georgievski and Aiello, 2015; Sirin et al., 2004; Sohrabi et al., 2009). It is particularly of interest for manufacturing and features in discussions of cyber-physical systems and Industry 4.0 (Kattepur and Mohalik, 2021; Köcher et al., 2022; Parimi et al., 2022).

In the past, AI planning techniques have been chiefly tailored for specific use cases. Implementing planners in this manner requires individuals equipped with the expertise to develop custom planning algorithms for a given situation. This makes AI planners costly, inefficient, and inaccessible due to the limited availability of people with such specialized knowledge.

In the past decade or so, notable advancements have been achieved in the realm of *domain-independent planners*. These planners employ a standardized set of algorithms, written and optimised by researchers with extensive expertise in the field, to solve any planning problem. Using a domain-independent planner only requires learning a simple syntax to encode a situation in a manner understandable to the planner. This is accomplished through a *domain* file, outlining the governing principles of the world in which the situation exists and the feasible steps for a solution, along with a *problem* file, specifying the initial state of the world and the desired outcome of a solution plan. The relative simplicity of using a domain-independent planner significantly enhances the efficiency and accessibility of AI planning for users beyond the academic sphere.

Now that domain-independent planners have become a reality, domain creation is one of the most challenging aspects of the planning process. Writing domain and problem files is the sole phase of the planning process which demands human input, making it susceptible to human error. Since AI planning is an emerging field, the available domain modelling assistance software are relatively young and their error diagnosis capabilities are underdeveloped. As a result, modellers can spend extensive amounts of time sifting through a broken domain only to discover a trivial mistake. Even more concerning, such errors may go undetected by the planner, leading to incorrect solutions that could have severe consequences if implemented in the real world. Consequently, adequate error detection and diagnosis mechanisms for domain and problem files are imperative for the widespread utilization of AI planning.

This report makes three distinct contributions in pursuit of this goal:

1. It provides a comprehensive compilation of potential domain errors, encompassing both syntax errors (issues with the way the model is written, such as an inconsistent number of parameters or misspelled variable name) and semantic warnings (indicators of potential issues with the meaning of the model, such as unused variable definitions).
2. It supplies a public repository of examples of flawed domains containing each of these errors. These can be used as test cases to ensure that domain modelling assistance software responds to each potential error in an accurate and helpful manner. Ideally, this list and repository can be built upon to establish a standardized benchmark for evaluating and enhancing domain modelling assistance software.
3. It offers an evaluation of currently available domain modelling assistance software based on extensive testing with the repository of flawed domains. This aims to enable developers of domain modelling assistance software to improve the reliability and efficiency of creating AI planning domains.

Chapter 2

Background

This section introduces the fundamentals of AI planning as related to the modelling languages used for testing in this report, PDDL (McDermott et al., 1998; Fox and Long, 2003) and HDDL (Höller et al., 2020) (see Section 2.3 for details). It begins with a general overview of AI planning concepts, then provides a lifted formalisation of Hierarchical Task Network planning (the formalisation expressed by HDDL). Finally, it narrows in to explain the syntax of PDDL and HDDL domains.

2.1 Introduction to AI Planning

The AI planning process uses algorithms to compose a sequence of actions which achieve a certain set of goals (Weld, 1999). Any situation with an initial state that we want to change into some other state can be considered a *planning problem*. AI planners generate all possible solutions to a planning problem using a description of the world in which the problem exists, provided by a domain and problem file.

AI planning problems are initially modelled with a lifted formalism which compactly represents the physics of a real-world situation by classing objects into types and describing how each type of object can interact with the world. The problem can then be *grounded*, which instantiates the formalisation with specific objects rather than abstract type variables. Grounding is required for a planner to solve the problem but is a much lengthier way of describing the domain. Consequently, domains are usually written with a syntax that corresponds to the lifted form, and then automatically grounded in the process of planning. Therefore the following paragraphs describe a lifted formalisation with examples of how it is expressed in HDDL (a domain modelling language discussed in Section 2.3). Note that there are other syntaxes which can be used to express the same formalisation; these examples are merely included to assist with clarity on the theoretical concepts.

The lifted formalisation of AI planning is based on a logic $\mathcal{L} = (P, T, V, C)$ (Höller et al., 2020). T is a set of type symbols which embody the types of objects involved in the problem (such as *container* or *ingredient*). The logic supports a type hierarchy; the notation A < B, where A and B are type symbols, will be used to show that type A is a sub-type of type B. In other words, every object of type A is also of type B (for example, *egg* < *ingredient*). V is a set of typed variable symbols which are used as parameters in the lifted formalisation (for example *?x - container*, which describes a variable named x with type *container*). C is a set of typed constants representing specific objects, which will take the place of variable symbols when the problem is grounded (for example, *cup1 - container*). P is a set of predicate symbols representing functions which return a truth value about the properties of constants in the system (for example *full ?x - container*, which returns true if the container given as a parameter is full and false otherwise). These predicates define the state of the real-world system at any one time (Höller et al., 2020).

The planner can change the state by executing tasks (Höller et al., 2020). A *task* (also referred to as a *task name*) is represented by a name and series of parameters (for example, *Fill : parameters (?x_0 - container ?x_1 - beverage)*). An *action* is a tuple *(name, pre, eff)* which associates the task name *name* with its precondition *pre* and effects *eff*. *pre* is a collection of predicates which must be true for the task to be carried out, and *eff* is a collection of predicates which will be true (eff^+) or false (eff^-) after the task has been carried out. Note that the variables in *pre* and *eff* must be included in the *name* parameters (Höller et al., 2020).

This report is concerned with two genres of AI planning: classical planning and hierarchical planning. Classical planning aims to generate an executable series of actions which, through their effects, will change the initial state into a state satisfying the goal description (McDermott et al., 1998). Hierarchical planning considers actions as previously defined for classical planning (now also called *primitive tasks*) and additionally involves *compound tasks* (Bercher et al., 2019). Compound tasks are tasks which cannot be executed atomically, but can be split into a series of smaller tasks (for example, baking a cake can't be done in one go, but requires a series of smaller steps to complete). Hierarchical planning no longer aims to fulfil certain goal conditions, but instead to decompose all compound tasks into a series of primitive tasks. These primitive tasks can then be executed to achieve the initial compound task/s specified (for example, cracking an egg could be a primitive task in the task decomposition of baking a cake) (Bercher et al., 2019).

2.2 Lifted Formalisation of HTN Planning

This report explores modelling assistance for domains written in PDDL (a classical planning language) and HDDL (a hierarchical planning language) (see Section 2.3 for details). As HDDL subsumes PDDL, the following section will focus on the lifted hierarchical planning formalisation expressed by HDDL, called *Hierarchical Task Network*

(HTN) planning. In HTN planning, a collection of tasks is called a *task network* (Höller et al., 2020).

Definition 1 (Task Network). *A task network tn over a task name set X is a tuple* $(I, \prec, \alpha, VC)$*, where:*

- I *is a set of task identifiers (arbitrary symbols used to uniquely represent tasks, as the same task may appear multiple times in a network).*
- $\prec$ *is a strict partial ordering of* I *(which specifies whether certain tasks need to be completed before other tasks, or vice versa).*
- $\alpha : I \to X$ *associates each task identifier with its corresponding task name.*
- VC *is a set of variable constraints which can enforce conditions on task parameters.*

Decomposition methods define how compound tasks can be broken into other task networks. They are expressed as a tuple (c, tn, VC) of a compound task name c, task network tn, and set of constraints VC on the parameters of c and tn. A set of decomposition methods is defined in the *planning domain.*

Definition 2 (Planning Domain). *A planning domain D is described by a tuple* $(\mathcal{L}, T_P, T_C, M)$*, where:*

- $\mathcal{L}$ *is the predicate logic.*
- T_P *and* T_C *are (respectively) a set of primitive and compound tasks.*
- M *is the set of decomposition methods for compound tasks in* T_C*.*

An HTN planning problem is represented as an initial task network where each compound task should be decomposed by the planner into a series of primitive tasks (Bercher et al., 2019).

Definition 3 (HTN Problem). *An HTN problem is represented by a tuple* (D, s_I, tn_I, g, c)*, where:*

- D *is a planning domain.*
- s_I *is the initial state, defined by the truth value of the predicates for each constant* ($s_I \in 2^F$*, where* F *is the set of grounded predicates).*
- tn_I *is the initial task network.*
- g *is the goal description* ($g \subseteq F$*) (this is unnecessary for hierarchical planning, but is included to increase similarity to the PDDL formalism)*
- C *is a set of typed constants representing the objects involved in the problem*

Any constraints on the variable parameters of a problem are taken into account in the process of grounding. Therefore, once a problem's lifted representation has been instantiated with constants we no longer need to consider variable semantics. This means we can move into a separate formalisation based on propositional logic which is equivalent to a grounded version of the lifted formalisation. This propositional formalisation is often colloquially referred to as a *ground formalisation.* The domains examined in this

report are written using a lifted formalisation, so the grounding process will not be thoroughly explained. An example of a grounded formalisation can be found in the paper by Bercher et al. (2019).

Once grounded, a solution to an HTN planning problem is defined as follows:

Definition 4 (Solution to an HTN Problem). *A solution to an HTN problem P is a task network tn_S that:*

- *can be reached by applying a series of decomposition methods from M to the initial task network tn_I,*
- *is composed of only grounded primitive tasks, and*
- *is executable; that is, there exists a sequence of its tasks which can be validly applied from state s_I (according to their required preconditions and effects).*

2.3 PDDL and HDDL

In the course of AI planning research many researchers have designed their own syntax, tailored to their own planners, to express the lifted formalisation of specific planning problems (Höller et al., 2020). However as AI planning techniques improve, domain-independent planners have become increasingly relevant and a need has arisen for standardised languages to express classical and hierarchical planning problems. Without universally accepted planning languages, researchers cannot directly compare the performance of planners without going through a long process of translating a planning domain written for one planner into a syntax that is recognised by another (Höller et al., 2020). This report utilizes two languages which are a potential solution to this problem: the *Planning Domain Definition Language* (PDDL) and the *Hierarchical Domain Definition Language* (HDDL).

PDDL is a classical planning language which was first proposed by McDermott et al. (1998), and later extended by Fox and Long (2003). It is widely accepted as the standard by the classical planning community, and has enabled significant research development by allowing researchers to more easily share planning resources.

HDDL has been recently proposed by Höller et al. (2020) to address the need for a standardised language for hierarchical planning problems. It extends PDDL to allow for hierarchical planning and has garnered interest as the official language used in the *International Planning Competition* (IPC) (Höller et al., 2021).

The following is a brief overview by example of the syntax of PDDL and HDDL, which will be necessary to understand the domain flaws tested later in this report. The predicate and type definition is the same in both languages:

```
1. (define (domain bake)
2.     (:types
```

```
3.         container ingredient - anything
4.         egg flour sugar - ingredient
5.     )
6.    (:predicates
7.          (empty ?x - container)
8.          (mixed ?x - container)
9.          (contains ?x0 - container ?x1 - ingredient)
10.     ...)
```

A primitive task or action is defined as follows:

```
11. (action mix-contents
12.    :parameters (?x - container)
13.    :precondition (and
14.        (not (empty ?x))
15.    )
16.    (:effect
17.          (mixed ?x - container)
18.     )
19. )
```

In HDDL, a compound task is defined as follows:

```
20. (:task MakeDough :parameters (?c - container ?egg1 - egg ?egg2 - egg ?f
- flour))
```

A method decomposition in HDDL is defined as:

```
21. (:method MixEggsAndFlour
22.    :parameters (?c - container ?egg1 - egg ?egg2 - egg ?f - flour)
23.    :task (MakeDough ?c - container ?egg1 - egg ?egg2 - egg ?f - flour)
24.    :precondition (and
25.          (empty ?c)
26.          (not (= ?egg1 ?egg2))
27.    )
28.    :ordered-subtasks (and
29.          (AchieveEggsAndFlourInBowl ?egg1 ?egg2 ?flour ?c)
30.          (mix-contents ?c)
31.     )
32. )
```

There is an alternative syntax for expressing partial orderings:

```
33. (:method MixEggsAndFlour
34.    :parameters (...)
```

```
35.     :task (...)
36.     :precondition (...)
37.     :subtasks (and
38.         (t1 (AchieveEggsAndFlourInBowl ?egg1 ?egg2 ?flour ?c))
39.         (t2 (mix-contents ?c))
40.     )
41.     :ordering (and
42.         (< t1 t2)
43.     )
44. )
```

This specifies that task t1 must be completed before task t2, but does not dictate the order of the tasks outside of this constraint. In method decompositions with more subtasks than the above example, this can be used to define a partial order.

Chapter 3

Related Work

This section explores the current landscape of domain modelling assistance in AI planning. It analyses current domain modelling assistance techniques, including approaches which diagnose errors for the modeller to fix and emerging approaches to automatically repair flawed domains. It justifies this report's focus on diagnosing errors for the modeller to fix, then details notable modelling assistance software in this field. It concludes by identifying the need for a comprehensive set of tests to ensure that modelling assistance software adequately identifies flawed domains and assists the modeller in remedying domain issues.

3.1 Modelling Assistance Techniques

Due to breakthroughs in domain-independent AI planning algorithms, one of the hardest parts of solving a planning problem is now defining a correct domain (Lin et al., 2023). The human input required to create a domain makes it much more inefficient and error-prone than the planning process itself. An incorrect domain may cause a planner to provide incorrect solutions or falsely conclude that no solution exists at all.

Consequently, domain modelling assistance is vital to make AI planning effective and accessible, but is often neglected in research in favor of improving planning techniques. Approaches to modelling assistance mainly fall into two camps: building smart auto-complete interfaces and parsers which pinpoint domain errors for the modeller to fix, or creating automated domain repair or creation tools which attempt to remove human error entirely from the domain modelling process.

Diagnosing Errors for the Modeller to Fix

Triep (2017) proposes in a thesis that domains should be checked based on two categories: *adequacy* and *consistency*. Adequacy requires that a domain matches the real-world

situation it is intended to describe, and that any solution found by the planner can be executed directly in the application environment. Consistency requires that the domain itself is free of contradictions, taking both syntax and semantics into account. Consistency can be validated based solely on the domain description and is the category most commonly addressed by existing modelling assistance software. Adequacy is much harder to validate, as checking a domain's compatibility with the intended real-world scenario requires more information than just the domain itself. Triep created a catalog of criteria for domain correctness and a tool to check these criteria. Triep's report isn't publicly available and is written in German, which makes it difficult to access for many researchers. This report will convert Triep's requirements for consistency into a list of errors which could contravene these requirements (although it will not address domain adequacy), extend the list by synthesising the work of other researchers, and make some of Triep's knowledge available to English-speaking researchers in a summarised form.

Zakhary (2018) provides the most comprehensive account of PDDL and HDDL modelling mistakes thus far. His bachelor's thesis considers flaws in both syntax and semantics, and provides extensive examples of each mistake and an explanation of why it may cause errors. While helpful, this list may be considered too longwinded for a reader who already understands domain modelling concepts and can make the link themselves as to why certain examples may cause errors. This report will summarise Zakhary's points more succinctly, allowing the reader to find extensive examples of each error in a repository of flawed domains should they require it. Zakhary also created a modelling assistance tool which diagnosed a range of syntactic and semantic domain flaws and provided constructive error messages to assist the modeller in repairing these issues. Unfortunately it is no longer compatible with the most recent updates to AI planning tools.

Automated Domain Repair/Creation

Lin et al. (2023) take a different approach to the issue of flawed modelling domains. Their paper acknowledges that for domain-independent AI planning to be widely used outside research contexts, people can't be relied upon to accurately write domains. Even with modelling assistance tools the process of correctly configuring a domain is too intensive to be used in industry or by anyone who hasn't spent a significant amount of time studying the topic. Lin et al. advocate for the use of automated domain repair tools and develop an approach which can repair a flawed classical domain in one second.

The tool takes a domain description and a possible solution (provided by the modeller) that should be found by the planner but isn't due to errors in the domain. It then determines ways of changing the domain so that the planner obtains the desired solution. The results of the paper are impressive, and the technique would greatly improve AI planning for some applications. However, it doesn't make the domain modelling process ubiquitously trivial. As well as the initial domain needing to be created, it adds the requirement of communicating a valid solution for the repair system to conform to. This solution would have to be written by the modeller, and in some scenarios a solution

plan could be quite long. It also reintroduces the potential for human error in writing the desired solution, particularly if the solution is complex when broken into primitive tasks (although the solution can likely be conclusively checked for correctness in the real world application, so this poses less risk than error in the domain). This automatic repair approach could potentially be combined with machine learning. Machine learning techniques could be used to write a valid solution to feed into the automated repair system and produce a correct domain. Once the domain has been created correctly, the AI planning algorithm will be able to find all other solutions more reliably than machine learning could. Machine learning has been used in domain repair by (Lindsay et al., 2020), who used it to refine domains to more accurately represent their real-world problem.

These automated repair processes straddle the gap between error diagnosis and automated domain creation. Vaquero et al. (2013) note that the domain creation process cannot be fully automated as it requires some level of informal knowledge input from the people posing the problem. They develop itSIMPLE, software dedicated to minimising the distance between the requirements of a real life problem and its formalisation. itSIMPLE facilitates the collation of informal requirements and assists with translating them into PDDL to be fed into an AI planner. If effective enough, this type of approach would eliminate or at least minimise the need for domain repair by stepping the modeller through the modelling process in a way that reduces the opportunity for human error.

This report will focus on diagnosing errors for the modeller to fix. While the possibility of automated domain repair is exciting and would make domain creation extremely effective, the above techniques are still in their early stages and are unlikely to be helpful until much more research has been done in the area. On the other hand, error diagnosis can be improved with existing techniques and will have more immediate benefits for the accessibility of AI planning.

3.2 Modelling Assistance in Commonly Used AI Planning Platforms

There are various systems which currently provide some form of domain error checking. This section gives a brief overview of notable platforms.

Note that except for the VSCode plugin, these platforms are all planners. This means their main purpose is to provide solutions to a domain and problem file, not to provide modelling assistance. However, as they are some of the most prominent software in AI planning, we are testing their parsers to see how well they implement error checking and diagnosis. If these planners are going to be used on a widespread basis, it's important that they are able to provide robust error checking to ensure that the solutions from dangerously flawed domains aren't implemented in real-world applications.

PDDL

Planning.Domains (Muise, 2016)
Planning.Domains is one of the most widely used platforms in AI planning, with over 178,000 usages since it was launched in January 2015 (Muise, 2023). It provides an API to various resources, primarily: a repository of existing planning problems, a cloud-based planner, and a PDDL editor. It is especially valuable in educational settings due to its online accessibility (no software download required), and the site includes a page of educational resources such as lecture slides and example assignments.

Visual Studio PDDL Plugin (Dolejsi, 2017)
The Visual Studio plugin provides syntax support for writing PDDL in VSCode. It does not include a planner itself, merely helps with the process of creating domains and is integrated with the Planning.Domains planner.

Fast Downward (Helmert, 2006)
Fast Downward is a widely used planning system based on heuristic search. In 2004 it won the classical planning section of the 4th International Planning Competition (IPC), and since then its system has formed the foundation of various other planners.

HDDL

Planning and Acting in a Network Decomposition Architecture (PANDA) (Höller et al., 2021)
PANDA is a state-of-the-art hierarchical planning system. It integrates planners based on a range of algorithms with a variety of related functionalities such as plan and goal recognition, plan repair, and plan verification.

HyperTensioN (Magnaguagno et al., 2021)
HyperTensioN was the winner of the total order HTN planning section of the 2020 IPC. Its compiler has a flexible three-stage design which enables easy incorporation of extensions and support for different languages.

LiloTane (Schreiber, 2021)
LiloTane was the runner up of the total order HTN planning section of the 2020 IPC. It accelerates the planning process by grounding domains via a lazy decision making strategy, delaying grounding statements until absolutely necessary.

3.3 Addressing the Need for Comprehensive Modelling Software Tests

While there has been some comprehensive work done exploring potential domain flaws, to the best of our knowledge there are no accessible papers yet wholly dedicated to providing a clear and succinct list of these potential errors. In addition, the existing research on modelling assistance has focused on developing tools for validating and repairing domains without a formal evaluation of whether current AI planning software is effectively implementing these techniques.

To fill these gaps, this report will:

1. Provide a clear and succinct synthesis of potential errors in classical (PDDL) and hierarchical (HDDL) planning domains in Sections 4.1 and 4.2.
2. Create a benchmark repository of flawed domains detailed in Section 4.3 and use them to evaluate the error diagnosis capabilities of the software described in Section 3.2.
3. Make this repository publicly available to assist developers in testing and improving the error diagnosis capabilities of current and future domain modelling assistance software.

Chapter 4

Potential Errors in Planning Domains

This chapter details a list of errors or potential errors that may be encountered when modelling planning domains in PDDL/HDDL. It is separated into syntax errors and warning signs of potential semantics errors, with subsections to distinguish errors related to classical planning (modelled in PDDL) and hierarchical planning (modelled in HDDL) (Höller et al., 2020). All classical planning errors also apply to hierarchical planning, as hierarchical planning subsumes classical planning (and thus HDDL builds on PDDL).

The list has been translated into a repository of flawed domains which can be used to test modelling assistance software or parsers used in planning systems and ensure that they provide helpful error messages in each situation. Tables 4.1 and 4.2 detail these domains, which can also be used as examples to help understand each potential error. The syntax errors would be suited to an error message from the software, while the potential semantics errors may produce a warning and explanation of why the modeller should check their code.

4.1 Syntax Errors

These errors are flaws in the created model which should prevent planning software from producing an output.

Classical Planning

- **Inconsistent Parameter Use.** The modeller attempts to use a predicate or task with either
  - a different number of parameters than it was defined with, or
  - a parameter of an incompatible type (Zakhary, 2018).

  A type error may be due to parameters being written in the wrong order, or

mistyping a variable name.

- **Undefined Entities.** The modeller attempts to use an undefined predicate, type, or task. This may be due to a typo in the entity name.

- **General Syntax Errors.** The modeller forgets to include a key piece of syntax or makes a typo - for instance, forgets to write ":parameters" in a task definition, adds an extra parentheses (Zakhary, 2018), forgets a dash when defining a variable, or forgets to write a ? in front of a variable label to differentiate it from a constant. It is expected that these types of errors would be captured by any parser.

- **Duplicated Definitions.** (Zakhary, 2018) The modeller attempts to write multiple definitions with the same name for a task, decomposition method, or predicate. Alternatively, the modeller writes duplicate entries in a task definition – for example, includes multiple ":parameters" entries.

- **Cyclic Type Declaration.** (Zakhary, 2018) When two types are directly or indirectly declared to be subtypes of each other. For example:

  - For two types A and B: A $<$ B and B $<$ A.
  - For three types A, B, and C: A $<$ B, B $<$ C and C $<$ A.

- **Undeclared Parameters.** The modeller tries to use a variable in the definition of a task or decomposition method that wasn't declared as a parameter of that task/method.

Hierarchical Planning

- **Missing Decomposition Methods.** (Bercher et al., 2011) The modeller defines an abstract task without a corresponding decomposition method, meaning no solution to the planning problem can involve this task (as it can't be decomposed into a series of actions).

- **Cyclic Ordering Constraints.** (Zakhary, 2018) A cyclic ordering defined for the subtasks in a decomposition method. For example, the ordering task1 $<$ task2, task2 $<$ task3, and task3 $<$ task1 is not a valid partial ordering, and hence the decomposition method does not define a task network.

4.2 Semantics Errors

These are errors in the domain logic which compile, but have a potentially flawed meaning which may be ineffective or produce an incorrect output.

Classical Planning

- **Complementary Effects.** (Triep, 2017) There is an intersection between the negated and positive effects of a task. One of these effects will be given precedence over the other, which may result in unintended side effects.

  This may occur in a valid domain when a predicate is added and removed from the state with two different inputs of the same type. A grounded plan may input the same constant into each predicate, resulting in the above situation despite an error-free domain. Modellers should be made aware via a warning if this case is a possibility in their domain. The software could suggest they consider adding a precondition to ensure the two constants are not the same.

- **Impossible Preconditions.** (Triep, 2017) The domain contains an unreachable task whose preconditions can never be fulfilled. This may be due to complementary preconditions, when the task requires the same variable predicate to be both true and false to execute. Since this error doesn't result in any side effects it is likely just a mistake, and has less of an effect on the meaning of the plan than complementary effects.

- **Unused Elements.** The modeller defines a type or predicate that isn't used in the domain (Triep, 2017), or declares a parameter in a task or decomposition method that isn't used in its definition.

- **Redundancy in Preconditions and Effects.** (Triep, 2017) The same literal is contained in the pre-conditions and effects of a task. Since this means the task hasn't changed the state in regard to that literal, it's inclusion is redundant.

- **Redundant Effects.** (Bercher, 2023) The modeller includes task effects which are already indirectly inherent in the precondition (similar to the above, but the redundancy is implicit).

- **Immutable Predicate.** (Schattenberg, 2023) A predicate is defined which never occurs in task effects (either as a negative or a positive effect). This means the state of that predicate is constant. The modeller should consider whether this is the intended behaviour.

Hierarchical Planning

- **Duplicate Orderings.** (Zakhary, 2018) The modeller defines ordered subtasks and partial orderings for a method decomposition. The partial ordering is redundant, since an ordered-subtask definition defines a total order.

- **Abstract Tasks Without a Primitive Refinement.** (Zakhary, 2018) The modeller defines a decomposition method which can never lead to a primitive termination. For example, a decomposition method's subtasks recursively reference an abstract task and the model does not include an alternate task decomposition method which can reach a complete decomposition into actions.

4.3 Corresponding Test Domains

Each potential error listed above has been written into a domain and compiled in a repository of flawed domains at https://gitlab.cecs.anu.edu.au/u7284920/ai-planning-domain-modelling-assistance-tests. These domains are intended to be used as tests for domain modelling assistance software, to see how well the software recognises and communicates each potential error. They can also serve as example cases should there be any confusion regarding the above explanations of potential errors. Table 4.1 and Table 4.2 list each potential error with their corresponding test domain/s. Flaws that are specific to hierarchical planning only have domains written in HDDL, and are marked with an asterisk. All other domains have both a HDDL and PDDL version. All HDDL domains run on the same problem, HDDL-base-problem.hddl, and all PDDL domains run on PDDL-base-problem.pddl.

Table 4.1: This table lists each domain in the provided repository with flawed syntax under its error category. Domains that are only relevant to hierarchical planning (HDDL) are marked with an asterisk *. All others have both a PDDL and HDDL test domain provided.

Error Category	**Test Domain**
	SYNTAX
Undefined Entities	undefined-type undefined-predicate undefined-task*
Inconsistent Parameter Use	inconsistent-num-parameters-predicate inconsistent-type-parameters-predicate inconsistent-num-parameters-task* inconsistent-type-parameters-task*
General Syntax Errors	extra-parentheses forgotten-dash forgotten-question-mark forgotten-entries
Duplicated Definitions	duplicate-action duplicate-predicate duplicate-parameters duplicate-compound-task* duplicate-decomposition-method*
Cyclic Type Declaration	directly-cyclic-subtypes indirectly-cyclic-subtypes
Undeclared Parameters	undeclared-task-parameter undeclared-method-parameter*
Missing Decomposition Methods	abstract-task-without-decomposition*
Cyclic Ordering Constraints	cyclic-ordering-for-subtasks*

Table 4.2: This table lists each domain in the provided repository with indications of flawed semantics under its potential error category. Domains that are only relevant to hierarchical planning (HDDL) are marked with an asterisk * . All others have both a PDDL and HDDL test domain provided.

Error Category	**Test Domain**
SEMANTICS	
Complementary Effects	complementary-effects possible-complementary-effects
Impossible Preconditions	complementary-preconditions
Unused Elements	unused-type unused-predicate unused-parameter
Redundancy In Preconditions and Effects	redundant-precondition-and-effect
Redundant Effects	implied-task-effects
Immutable Predicate	immutable-predicate
Duplicate Orderings	duplicate-orderings*
Abstract Tasks Without a Primitive Refinement	abstract-task-without-refinement*

Chapter 5

Evaluation

The test domains discussed in Section 4.3 were run on the software detailed in Section 3.2 to evaluate the proficiency of prevalent modelling assistance tools at diagnosing potential errors. The following sections provide the overall results followed by a short discussion of notable points for each software.

5.1 Overall Test Results

Software was evaluated for each flawed domain based on three categories:

- **Error Detection.** Whether the software recognises the error and stops the parsing process, crashes without catching the error, or provides a solution despite the model being wrong (or claims the problem is unsolvable). These cases are recorded as 'yes' (green), 'crashes' (yellow), and 'no' (red), respectively.
- **Location Guidance.** Whether the software pinpoints the correct line number of the error, points toward the correct area of code (usually by naming the task which contains the error), or provides an inaccurate or no indication of the location of the error. These cases are recorded as 'yes' (green), 'close' (yellow), and 'no' (red), respectively.
- **Error Description.** Whether the software provides a clear and helpful description of the error, a correct description which is unclear or confusing, or no correct error description. These cases are recorded as 'yes' (green), 'close' (yellow), and 'no' (red), respectively. Images of each error message can be found in the repository of flawed domains 'softwareTesting → results' folder.

The results of each software in these three categories for each test domain are presented in Figure 5.2. For an easier comparison, the overall performance of each software is given in Figure 5.1.

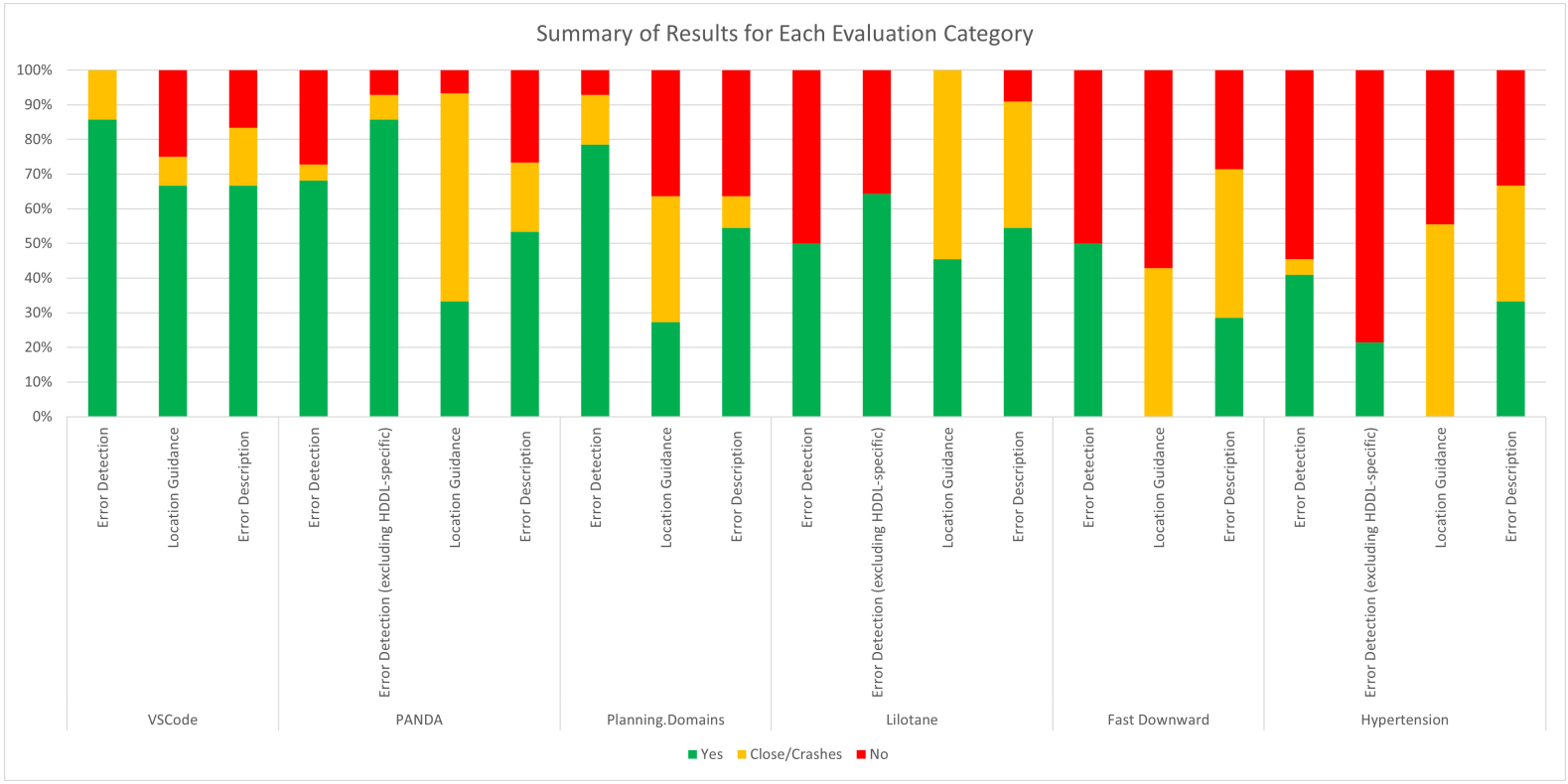

Figure 5.1: The overall success rates of the evaluated software at error detection, error location guidance, and clear and descriptive error messages. Note that the location guidance and error description rates are percentages of the number of errros caught by the parser, not the total number of errors tested. For example if a parser catches 6 of 10 errors, and provides a helpful error message for 3 of them, its success rate for Error Description would be 50% in this chart.

For hierarchical planners, performance on only the domains which were tested on both classical and hierarchical planning systems is included (called 'excluding HDDL-specific') to allow for fair comparison between hierarchical and non-hierarchical software.

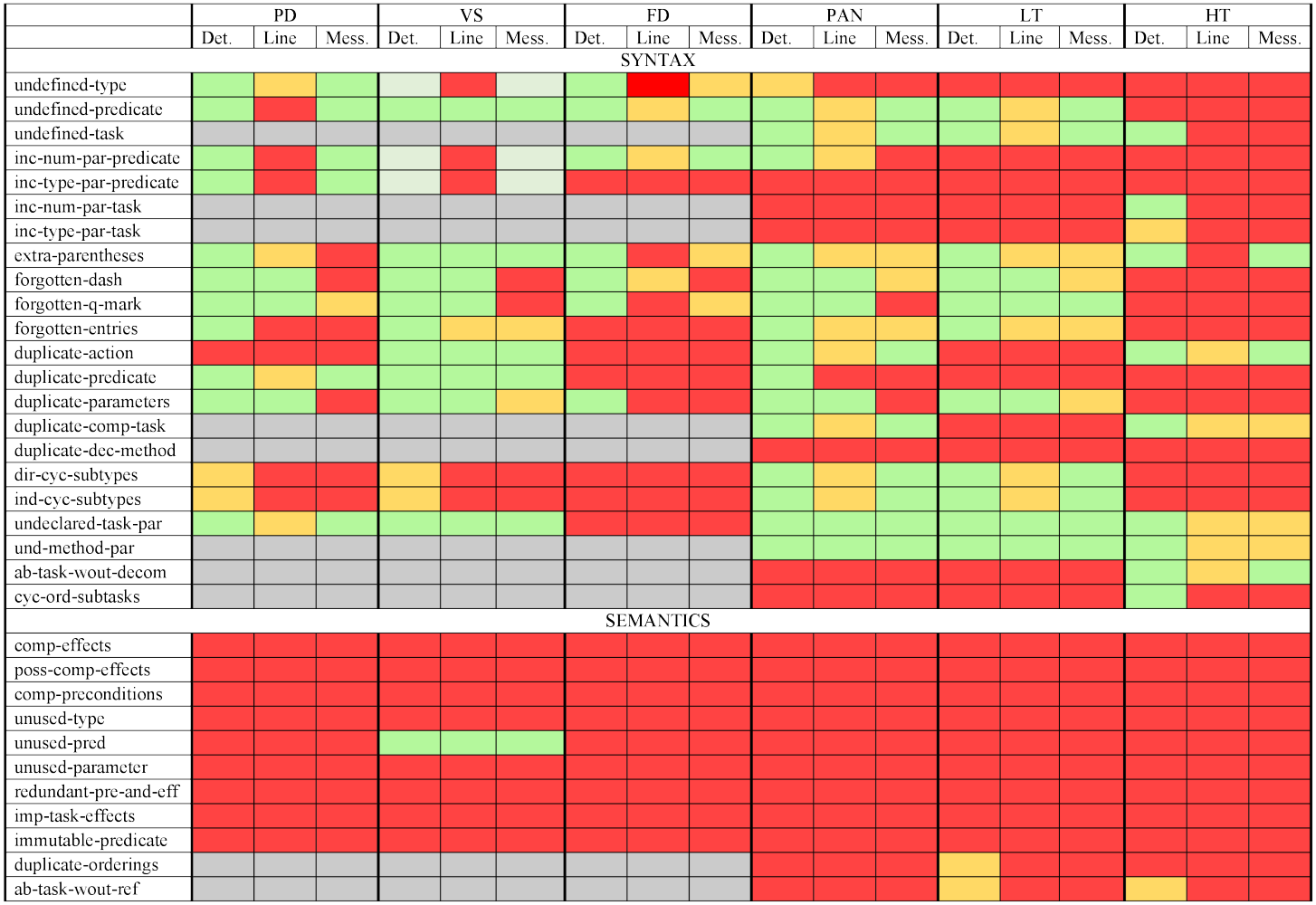

Figure 5.2: Results of each software (PD - Planning.Domains, VS - VSCode plugin, FD - Fast Downward, PAN - PANDA, LT - Lilotane, HT - Hypertension) being tested on each flawed domain. These are evaluation under categories of error detection (Det.), line pinpointing (Line), and error message quality (Mess.). Note that for the VSCode plugin, the lighter shade of green corresponds to errors which were caught by the Planning.Domains backend software (see Section 5.3.

5.2 Planning.Domains Evaluation

The online modelling functionality of Planning.Domains made it by far the most accessible software in this report. Its error diagnosis interface was similarly user-friendly, with an error file and short error message clearly generated when problems are encountered. However, Planning.Domains does not have the ability to give potential error warnings before running the planner. This means, like most of the software, it did not diagnose any potential semantics errors. It did have a fairly good detection rate for syntax errors, successfully detecting 79% of syntax errors as in Figure 5.3. It was less successful at guiding the modeller to fix those errors, only pointing toward the location of the error 64% of the time and giving a clear error description 55% of the time.

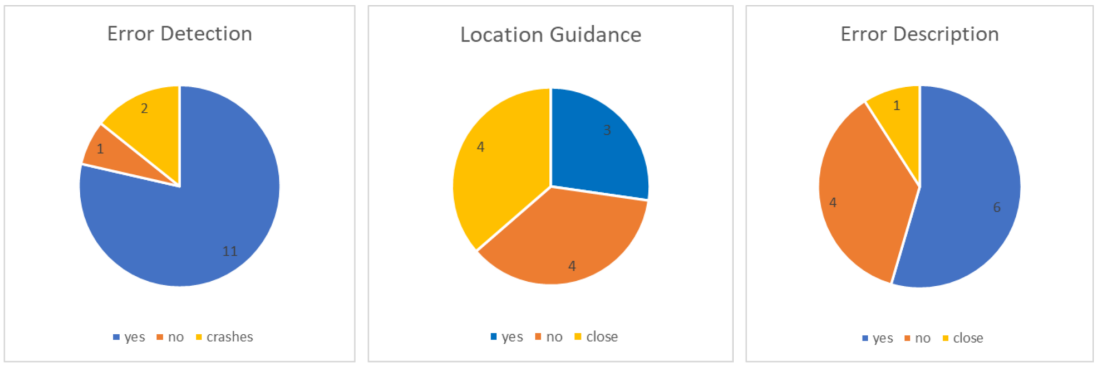

Figure 5.3: A breakdown of Planning.Domain's responses to the flawed syntax test domains.

5.3 VSCode Plugin Evaluation

In terms of modelling assistance, the VSCode plugin proved to be the most effective software among those evaluated. Given the critical importance of reliability and accuracy in real-world AI planning scenarios, it is concerning if a planner can execute a flawed domain without recognizing the error. Unlike other software, the plugin never generated a plan for a domain with syntax errors, as shown in Figure 5.4. This is partly because the plugin runs the Planning.Domains planner, so any errors that were not directly detected by the plugin were caught by Planning.Domains during the planning process (Dolejsi, 2017). In such cases, the plugin outputs the error message from Planning.Domains to the terminal.

The plugin's error pinpointing method is also particularly effective, directly highlighting 67% of the errors it identifies in red. The error messages often offer automatic fixes, although it is important to verify their correctness before implementing them. The only notable drawback of the plugin is that when Planning.Domains software catches er-

rors, the error message provided by the plugin often includes unnecessary or extraneous information. This makes the message more confusing than when viewed on the Planning.Domains site, which only includes the most relevant section of the error message.

The plugin was also the only software to recognise any potential semantics errors; it was surprisingly proficient at warning the modeller of unused predicates. It even found an unused predicate, is-start-runway, in the AIPS2000 Planning Competition Airplane domain (Trueg, 2000).

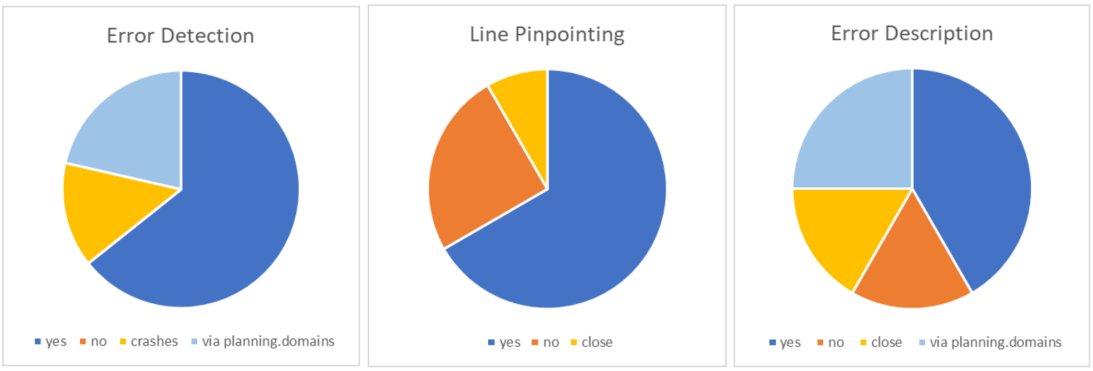

Figure 5.4: A breakdown of the VSCode plugin's responses to the flawed syntax test domains.

5.4 Fast Downward Evaluation

Given that it is fairly well established in the AI planning space, Fast Downward performed surprisingly badly at error detection and diagnosis. It detected 50% of syntax errors, provided some form of location guidance 27% of the time, and gave a clear and helpful error message only 18% of the time (see Figure 5.5).

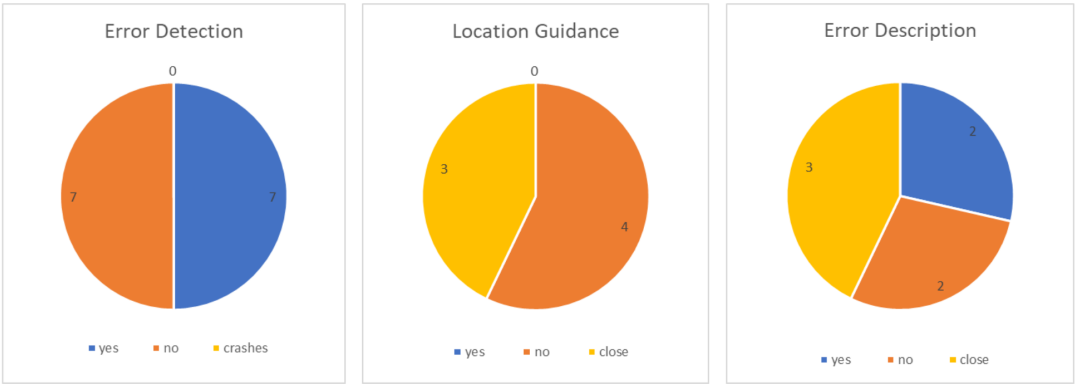

Figure 5.5: A breakdown of Fast Downward's responses to the flawed syntax test domains.

5.5 PANDA Evaluation

PANDA had the best error diagnosis capabilities of the hierarchical planners. This is unsurprising given that Lilotane and Hypertension are the newest of the evaluated software. PANDA picked up 68% of errors overall as demonstrated by Figure 5.6, and 86% of the errors related to both PDDL and HDDL (i.e. errors which had been tested on both PDDL and HDDL software). This makes it the second best software at parsing domains after VSCode.

While PANDA performed well, there is one concerning issue that merits attention. Specifically, PANDA appeared to solve the inconsistent-num-parameters-task domain by introducing an arbitrary constant (seg_pp_0_60) into the solution, even though it was not specified in the domain. This is troubling, as there could be serious consequences if an AI planner interacting with the real world introduces a random constant into its plan. It is recommended that someone with a strong understanding of PANDA's underlying algorithms investigate this issue further to determine how the planning process produces this solution, and whether it points toward an issue which should be addressed.

Figure 5.6: A breakdown of PANDA's responses to the flawed syntax test domains.

5.6 Lilotane Evaluation

Lilotane leverages *pandaPIparser* (Behnke et al., 2020) to parse domains (Schreiber, 2021), which results in many of its error messages being identical to PANDA's. Lilotane detected 50% of syntax errors (see Figure 5.7), and proved to be excellent at locating errors by providing some indication of location for every error caught. It supplied helpful error messages in 55% of cases.

Like PANDA, Lilotane's handling of the inconsistent-num-parameters-task domain is troubling. Lilotane, too, provided a solution that included the segment seg_pp_0_60 without the domain specifying it. It is noteworthy that both PANDA and Lilotane used the same object in this case, suggesting the issue may arise in a shared underlying logic or library. Unlike PANDA, Lilotane solved the inconsistent-type-parameters-task domain as well, which should not be possible for a reliable planner. This indicates that Lilotane's type checking is not robust enough to prevent false solutions. Given these concerns, further investigation is recommended to verify and remedy these issues.

Figure 5.7: A breakdown of Lilotane's responses to the flawed syntax test domains.

5.7 Hypertension Evaluation

Despite being the winner of the IPC 2020 total order track, Hypertension had the weakest error detection capabilities among the planners examined, as shown in Figure 5.8. The planner frequently ran on flawed domains, only catching 41% of syntax errors. It did not provide any line numbers for the errors it did catch, although it identified the correct area of code 56% of the time. It gave an adequate error description only 33% of the time.

Interestingly, Hypertension was surprisingly effective at handling the extra-parentheses and abstract-task-without-decomposition domains, outperforming all other hierarchical planners for these errors.

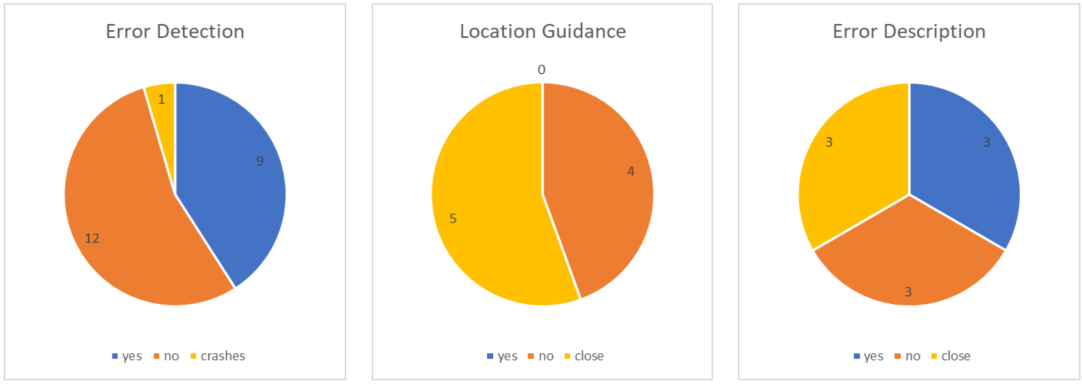

Figure 5.8: A breakdown of Hypertension's responses to the flawed syntax test domains.

5.8 ChatGPT Initial Testing

ChatGPT 3.5 and 4.0 were also briefly tested for error diagnosis capabilities, with the prompt "Hey ChatGPT! Can you please take a look at this AI planning domain and tell me if it contains any errors?". The results were initially promising, with GPT seemingly able to determine the correct cause of error. However, it would give a 'solved' version of the code which seemed unrelated to the domain it was given. Eventually it was realised that the domains included a comment which identified the error (as in the repository). Rather than actually parsing the domain, ChatGPT was simply reading the error directly from the comment. Once this comment was removed, ChatGPT began guessing arbitrary AI planning domain errors. It was not able to accurately diagnose a domain error on its own.

While this is the case, ChatGPT could quickly write an adequate domain (with some minor errors) for a real-world situation described to it. If it can do this more reliably, it may be possible to largely automate the domain creation process. This was very recently implemented with another language model by Liu et al. (2023), which may lead to some exciting breakthroughs.

Chapter 6

Concluding Remarks

The following chapter summarises the work done in this report and provides suggestions for future avenues of research.

6.1 Conclusion

This report gives an exhaustive list of potential domain modelling errors for classical and hierarchical AI planning. It provides example domains containing each of these errors, proposed to form the foundation of a set of standardized tests for domain modelling assistance software. Finally, it details an evaluation of current notable PDDL and HDDL planners and modelling assistance software (Planning.Domains, the VSCode PDDL Plugin, Fast Downward, PANDA, Lilotane, and Hypertension), reporting how they responded to each flawed domain in terms of error detection and diagnosis. Given that the average syntax error detection rate was 62%, the average location guidance rate was 69%, the average helpful error message rate was 43%, and almost none of the software picked up any semantics errors, it is clear that domain modelling assistance software requires more robust error checking features before domain-independent planners can become an effective and widespread AI technique.

6.2 Future Work

This report points toward a range of opportunities for future work in AI planning. First and most obviously, it confirms that further development is required in the error detection and diagnosis capabilities of domain modelling assistance software. Future work should include improving the parsers of these software to catch a wider range of potential errors (hopefully all of the errors listed in this report), and to provide more precise location guidance and clearer error messages for these errors. Particularly, new techniques may need to be considered to detect potential semantics errors.

The repository of flawed domains could be improved to make the domains generic and more concise. Additionally, it should be extended to include testing of errors related to problem files as well as domain files. Problem files were not included in the scope of this report due to time constraints, but have their own set of potential errors which should be investigated as thoroughly as potential domain errors, such as a predicate not being given an initial state or a typo in the name of the corresponding domain at the top of the problem file.

It is highly recommended that the strange solving behaviour of Lilotane and PANDA, described in Sections 5.5 and 5.6, be further investigated. Ideally, developers of these software who have a deep understanding of the algorithms used in each planner should verify this issue, determine the root cause, and fix it if necessary.

As discussed in Section 5.2, Planning.Domains is highly accessible due to being a cloud-based planner. In contrast, the hierarchical planners all require an error-prone software installation process. It is recommended that a hierarchical planner be made cloud-based to increase accessibility for users without experience in command-line package installation.

Further research could be done into the ability of ChatGPT to independently create domains and problem files based on a short description of a real-world situation, as described in Section 5.8.

Finally, techniques could be developed to diagnose errors with the adequacy of a domain as defined in Section 3.1. This was outside the scope of this project, as it could not be done solely with a domain file and would require some form of additional information regarding the intended semantics of the domain. Perhaps a survey of intended mutexes or short description of the intended behaviour of the domain parsed by a language model could be investigated with this purpose in mind.

Bibliography

Bartàk, R. and Vlk, M., 2016. Hierarchical task model for resource failure recovery in production scheduling. In *Advances in Computational Intelligence - 15th Mexican International Conference on Artificial Intelligence, MICAI 2016, Cancún, Mexico, October 23-28, 2016, Proceedings, Part I*, vol. 10061 of *Lecture Notes in Computer Science*, 362–378. Springer. https://doi.org/10.1007/978-3-319-62434-1_30. [Cited on page 1.]

Beetz, M.; Jain, D.; Mösenlechner, L.; Tenorth, M.; Kunze, L.; Blodow, N.; and Pangercic, D., 2012. Cognition-enabled autonomous robot control for the realization of home chore task intelligence. *Proc. IEEE*, 100, 8 (2012), 2454–2471. https://doi.org/10.1109/JPROC.2012.2200552. [Cited on page 1.]

Behnke, G.; Höller, D.; Schmid, A.; Bercher, P.; and Biundo, S., 2020. On succinct groundings of HTN planning problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, 9775–9784. AAAI Press. https://ojs.aaai.org/index.php/AAAI/article/view/6529. [Cited on page 27.]

Bercher, P., 2023. Emails regarding redundant effects. Private Communication. [Cited on page 17.]

Bercher, P.; Alford, R.; and Höller, D., 2019. A survey on hierarchical planning - one abstract idea, many concrete realizations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 6267–6275. ijcai.org. https://doi.org/10.24963/ijcai.2019/875. [Cited on pages 4, 5, and 6.]

Bercher, P.; Biundo, S.; Geier, T.; Müller, F.; and Schattenberg, B., 2011. Advanced user assistance based on AI planning. *Cogn. Syst. Res.*, 12, 3-4 (2011), 219–236. https://doi.org/10.1016/j.cogsys.2010.12.005. [Cited on page 16.]

Dolejsi, J., 2017. PDDL. https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl. [Cited on pages 12 and 24.]

Fox, M. and Long, D., 2003. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20 (2003), 61–124. https://doi.org/10.1613/jair.1129. [Cited on pages 3 and 6.]

Georgievski, I. and Aiello, M., 2015. HTN planning: Overview, comparison, and beyond. *Artif. Intell.*, 222 (2015), 124–156. https://doi.org/10.1016/j.artint.2015.02.002. [Cited on page 1.]

González, J. C.; Pulido, J. C.; and Fernández, F., 2017. A three-layer planning architecture for the autonomous control of rehabilitation therapies based on social robots. *Cogn. Syst. Res.*, 43 (2017), 232–249. https://doi.org/10.1016/j.cogsys.2016.09.003. [Cited on page 1.]

Helmert, M., 2006. The fast downward planning system. *J. Artif. Intell. Res.*, 26 (2006), 191–246. https://doi.org/10.1613/jair.1705. [Cited on page 12.]

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S., 2021. The PANDA framework for hierarchical planning. *Künstliche Intell.*, 35, 3 (2021), 391–396. https://doi.org/10.1007/s13218-020-00699-y. [Cited on pages 6 and 12.]

Höller, D.; Behnke, G.; Bercher, P.; Biundo, S.; Fiorino, H.; Pellier, D.; and Alford, R., 2020. HDDL: an extension to PDDL for expressing hierarchical planning problems. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, 9883–9891. AAAI Press. https://ojs.aaai.org/index.php/AAAI/article/view/6542. [Cited on pages 3, 4, 5, 6, and 15.]

Kattepur, A. and Mohalik, S., 2021. AI planning for tele-operated robotic network slice reconfiguration. In *4th IEEE 5G World Forum, 5GWF 2021, Montreal, QC, Canada, October 13-15, 2021*, 447–452. IEEE. https://doi.org/10.1109/5GWF52925.2021.00085. [Cited on page 1.]

Köcher, A.; Heesch, R.; Widulle, N.; Nordhausen, A.; Putzke, J.; Windmann, A.; and Niggemann, O., 2022. A research agenda for AI planning in the field of flexible production systems. In *5th IEEE International Conference on Industrial Cyber-Physical Systems, ICPS 2022, Coventry, United Kingdom, May 24-26, 2022*, 1–8. IEEE. https://doi.org/10.1109/ICPS51978.2022.9816866. [Cited on page 1.]

Lin, S.; Grastien, A.; and Bercher, P., 2023. Towards automated modeling assistance: An efficient approach for repairing flawed planning domains. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI 2023)*. AAAI Press. [Cited on pages 9 and 10.]

Lindsay, A.; Franco, S.; Reba, R.; and McCluskey, T. L., 2020. Refining process descriptions from execution data in hybrid planning domain models. In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, 469–477. AAAI Press. https://ojs.aaai.org/index.php/ICAPS/article/view/6742. [Cited on page 11.]

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P., 2023. LLM+P: empowering large language models with optimal planning proficiency. *CoRR*, abs/2304.11477 (2023). doi:10.48550/arXiv.2304.11477. https://doi.org/10.48550/arXiv.2304.11477. [Cited on page 28.]

Magnaguagno, M. C.; Meneguzzi, F.; and Silva, L. d., 2021. HyperTensioN: A three-stage compiler for planning. In *Proceedings of the 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, 5–8. [Cited on page 12.]

McDermott, D.; Ghallab, M.; Howe, A. E.; Knoblock, C. A.; Ram, A.; Veloso, M. M.; Weld, D. S.; and Wilkins, D. E., 1998. PDDL - the planning domain definition language. Yale Center for Computational Vision and Control. [Cited on pages 3, 4, and 6.]

Muise, C., 2016. Planning.Domains. ICAPS. http://www.haz.ca/papers/planning-domains-icaps16.pdf. [Cited on page 12.]

Muise, C., 2023. Private communication between Pascal Bercher and Christian Muise. Private Communication. [Cited on page 12.]

Parimi, V.; Rubinstein, Z.; and Smith, S., 2022. T-htn: Timeline based htn planning for multiple robots. In *5th Workshop on Hierarchical Planning, ICAPS 2022, Singapore, June 13-24, 2022*, 73–77. ICAPS. https://icaps22.icaps-conference.org/workshops/HPlan/papers/HPlanProceedings-2022.pdf. [Cited on page 1.]

Schattenberg, B., 2023. Conversation regarding graphical user interfaces for domain modelling. Private Communication. [Cited on page 17.]

Schreiber, D., 2021. Lifted logic for task networks: TOHTN planner Lilotane in the IPC 2020. In *Proceedings of the 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*, 9–12. [Cited on pages 12 and 27.]

Sirin, E.; Parsia, B.; Wu, D.; Hendler, J. A.; and Nau, D. S., 2004. HTN planning for web service composition using SHOP2. *J. Web Semant.*, 1, 4 (2004), 377–396. https://doi.org/10.1016/j.websem.2004.06.005. [Cited on page 1.]

Sohrabi, S.; Baier, J. A.; and McIlraith, S. A., 2009. HTN planning with preferences. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 1790–1797. http://ijcai.org/Proceedings/09/Papers/298.pdf. [Cited on page 1.]

Stock, S.; Mansouri, M.; Pecora, F.; and Hertzberg, J., 2015. Online task merging with a hierarchical hybrid task planner for mobile service robots. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2015, Hamburg, Germany, September 28 - October 2, 2015*, 6459–6464. IEEE. https://doi.org/10.1109/IROS.2015.7354300. [Cited on page 1.]

Triep, J., 2017. Untersuchung von konsistenz und adäquatheit von planungsmodellen. [Cited on pages 9, 10, and 17.]

Trueg, S., 2000. PDDL file for the AIPS2000 Planning Competition. https://github.com/AI-Planning/classical-domains/tree/main/classical/airport. [Cited on page 25.]

Vallati, M.; Chrpa, L.; and Kitchin, D. E., 2019. How to plan roadworks in urban regions? A principled approach based on AI planning. In *Computational Science - ICCS 2019 - 19th International Conference, Faro, Portugal, June 12-14, 2019, Proceedings, Part V*, vol. 11540 of *Lecture Notes in Computer Science*, 453–460. Springer. https://doi.org/10.1007/978-3-030-22750-0_37. [Cited on page 1.]

Vaquero, T. S.; Silva, J. R.; Tonidandel, F.; and Beck, J. C., 2013. itSIMPLE: towards an integrated design system for real planning applications. *Knowl. Eng. Rev.*, 28, 2 (2013), 215–230. https://doi.org/10.1017/S0269888912000434. [Cited on page 11.]

Weld, D. S., 1999. Recent advances in AI planning. *AI Mag.*, 20, 2 (1999), 93–123. https://doi.org/10.1609/aimag.v20i2.1459. [Cited on page 3.]

Zakhary, K., 2018. A generic approach for the provision of advice in modeling hierarchical planning domains. [Cited on pages 10, 15, 16, 17, and 18.]