Verification and Validation of Planning Domain Models

By

ANAS SHRINAH

Department of Computer Science
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Engineering.

JUNE 2023

ABSTRACT

The verification and validation of planning domain models is one of the biggest challenges to deploying planning-based automated systems in the real world.

The state-of-the-art verification methods of planning domain models are vulnerable to false positives, i.e. counterexamples that are unreachable by sound planners when using the domain under verification during planning tasks. False positives mislead designers into believing correct models are faulty. Consequently, designers needlessly debug correct models to remove these false positives. This process might unnecessarily constrain planning domain models, which can eradicate valid and sometimes required behaviours. Moreover, catching and debugging errors without knowing they are false positives can give verification engineers a false sense of achievement, which might cause them to overlook valid errors.

To address this shortfall, the first part of this thesis introduces *goal-constrained planning domain model verification*, a novel approach that constrains the verification of planning domain models with planning goals to reduce the number of unreachable planning counterexamples. This thesis formally proves the correctness of this method and demonstrates the application of this approach using the model checker Spin and the planner MIPS-XXL. Furthermore, it reports the empirical experiments that validate the feasibility and investigates the performance of the goal-constrained verification approach. The experiments show that not only the goal-constrained verification method is robust against false positive errors, but it also outperforms under-constrained verification tasks in terms of time and memory in some cases.

The second part of this thesis investigates the problem of validating the functional equivalence of planning domain models. The need for techniques to validate the functional equivalence of planning domain models has been highlighted in previous research and has applications in model learning, development and extension. Despite the need and importance of proving the functional equivalence of planning domain models, this problem attracted limited research interest.

This thesis builds on and extends previous research by proposing a novel approach to validate the functional equivalence of planning domain models. First, this approach employs a planner to remove redundant operators from the given domain models; then, it uses a Satisfiability Modulo Theories (SMT) solver to check if a predicate mapping exists between the two domain models that makes them functionally equivalent. The soundness and completeness of this functional equivalence validation method are formally proven in this thesis.

Furthermore, this thesis introduces D-VAL, the first planning domain model automatic validation tool. D-VAL uses the FF planner and the Z3 SMT solver to prove the functional equivalence of planning domain models. Moreover, this thesis demonstrates the feasibility and evaluates the performance of D-VAL against thirteen planning domain models from the International Planning Competition (IPC). Empirical evaluation shows that D-VAL validates the functional equivalence of the most challenging task in less than 43 seconds. These experiments and their results provide a benchmark to evaluate the feasibility and performance of future related work.

ACKNOWLEDGEMENTS

All praise be to Allah for giving me the ability to complete my PhD thesis. I thank Allah for blessing me with amazing people who helped me and kept me going throughout this journey. Prophet Muhammad, peace be upon him, said, “He who does not thank people does not thank Allah.”

First and foremost, I would like to thank my supervisors, Prof. Kerstin Eder and Prof. Derek Long, for their support during my PhD study. I am truly grateful to Kerstin for her exceptional guidance and unlimited support. I am particularly thankful for Kerstin's unwavering belief in my abilities and continuous encouragement. The invaluable opportunities she offered me over the past few years have played a pivotal role in my growth as a researcher and lecturer. Thank you, Kerstin, for being an extraordinary supervisor and mentor.

I am immensely grateful to Derek for his invaluable guidance, insightful comments, and constructive suggestions. His vast knowledge and extensive experience have been instrumental in shaping the development of my research. Derek's generous support and patience have helped me navigate through various challenges and refine the proofs of the theorems presented in this thesis. I am truly indebted to him for his invaluable contributions to my academic journey.

I would like to thank the members of Trustworthy Systems Laboratory for their useful discussions and joyful moments at the lab, and the staff at the University of Bristol for their prompt support, especially Callum Wright for his help in setting up my experiments on the high performance computing system BlueCrystal. I also would like to thank Dr. Djamel Rezgui for his friendly advice and encouragement.

This thesis was sponsored by Engineering and Physical Sciences Research Council (EPSRC) grant EP/P510427/1 in collaboration with Schlumberger. Many thanks to EPSRC and Schlumberger for generously funding this PhD. I am also grateful to Dr. Inês Cecílio, who was the Science and Technology Manager: Automation and Planning at Schlumberger Cambridge Research Center. Inês arranged my visits to the centre and advised me on the first part of my research.

I am deeply grateful to my family for their endless love, support, and encouragement. In particular, I want to express my heartfelt gratitude to my parents, Ahmad and Maha; my parents-in-law, Fayez and Malak; my brothers, Abdulrahman, Ali, and Ehsan; and my sister, Safi. Your constant care and love have been a source of strength and motivation throughout my journey. I am forever indebted to you all for the countless ways you have shown your belief in me. From the bottom of my heart, thank you for everything!

I am grateful to my dear friends Obada, Khaled, Mousab, Omran, Yasin, Alaa, Kitty, Jacquie, and Sue for their support and friendship. Your presence and encouragement have made the challenges more bearable and the successes more meaningful. Thank you for always being there for me!

Lastly, I am profoundly grateful to my beloved wife and soulmate, Lujain Altaiyan, for her unconditional love, care, and patience. Her constant support and understanding have been the pillars that have kept me strong and motivated. I would also like to extend my heartfelt appreciation to my sons, Hamzah and Kareem, whose presence and laughter have brought joy and inspiration to my life. I am blessed to have such a loving family by my side. Thank you all for being my greatest source of happiness.

AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

ANAS SHRINAH

16 JUNE 2023

BRISTOL

TABLE OF CONTENTS

List of Tables

LIST OF FIGURES

LISTINGS

CHAPTER 1

INTRODUCTION

Planning and task scheduling techniques are increasingly applied to real-world problems like activity sequencing and resource management. These processes are implemented in planning-based automated systems which are already used in space missions [3–5], search and rescue [6], logistics [7] and many other domains.

A planning system consists of a planner, a planning domain model, and a planning problem. The planner takes as an input the domain model, which describes application-specific states and actions, and the planning problem that specifies the goal condition and the initial state. From these inputs, the planner returns a plan, which is a sequence of actions that can achieve the goal starting from the initial state.

Planning domain models provide the foundation for planning; they provide abstract models of real-world physical actions. These models formalise available actions along with their preconditions and effects. Due to modelling errors, a domain model can be inconsistent, incomplete, or inaccurate with respect to its specification. Erroneous planning domain models could cause planners to fail to find a plan, generate unrealistic plans that will fail to execute, or produce unsafe plans. Therefore, planning domain model verification and validation methods that are robust, trustworthy and systematic are crucial to gain high confidence in the safety, integrity and correctness of the performance of planning-based automated systems.

This thesis is concerned with the research area of classical planning, where the effects of actions are deterministic and fully observable, and the environment is discrete. Specifically, this thesis focuses on the verification and validation of classical planning domain models. The definitions of the terms "verification" and "validation" in the literature are often inconsistent [8] and, in many cases, are used interchangeably or joined together and abbreviated as "V&V" to refer to the efforts that aim to prove the correctness of a system with regard to specified criteria [9]. In the context of this thesis, the verification of a planning domain model is described as verifying that any plan produced using the planning domain model satisfies a set of properties. On the other hand, the validation task checks whether the given

planning domain model correctly captures the knowledge of the planning domain.

This thesis aims to provide automated support to verify and validate planning domain models. First, I will discuss the verification research problem and summarise the contributions related to this topic. Second, I will explain the validation research problem and list the contributions related to this area of research.

1.1 Planning Domain Model Verification

Planning domain model verification is accomplished using test-based techniques or formal methods such as model checking. The former approach tests if a planning domain model can be used by a planner to produce plans that can achieve a given set of planning goals starting from a set of initial states. Moreover, test-based verification methods check if produced plans satisfy some required properties, like the execution of plans does not cause the system to reach an unsafe state. Since such approaches depend on the soundness of the used planner and also need to check the validity of the produced plans, it might be challenging to verify specific parts of the planning system independently using test-based verification methods. In fact, test-based verification of planning domain models assumes a sound planner is used in the testing to restrict the testing outcomes to the verification of planning domain models [10, 11].

On the other hand, a model checker takes as input a representation of the planning domain model, a set of objects, an initial state, and a set of formal properties which are required to hold for any plan generated using the planning domain model. First, the model checker produces a finite state machine from the planning domain model and the set of objects. Then, the model checker traverses this finite state machine to search for a state or a sequence of states that violates the given property. The trace of transitions that leads to the violation of the property is returned as a counterexample. If no counterexample is found, then the model is deemed correct with respect to the property, i.e. the model does not allow the construction of plans that could falsify the given property [12, 13].

1.1.1 Research Problem: Planning Domain Model Verification

Model checkers are designed to verify properties of finite state systems like communication protocols, hardware designs, and concurrent programs [14]. A model checker traverses the state space of a system with the only objective of finding a sequence of transitions that falsifies a given property. Model checkers operate under the assumption that the system under verification can execute any valid transition in any state. Therefore, model checkers can select any valid transition in any state while searching for a sequence of transitions that violates the given property.

When a model checker is used to verify if a planning domain model satisfies some property with regard to a set of objects, the ground actions of the planning domain model are interleaved to produce the state space of the planning domain model. Then, the produced state space is searched by the model checker to find a violation of the given property. This direct application of model checkers to the verification of planning domain models overlooks a subtle difference between how a finite state system

executes valid transitions and how a planner schedules actions from the state space of a planning domain model when searching for plans The important distinction lies in the order of the execution of valid transitions. While the execution of the valid transitions of finite state systems is non-deterministic and is not guided by a central search algorithm, the execution of planning actions is scheduled by planners, automated reasoners, which aim to achieve certain goals.

A planner searching for a plan does not schedule any valid action in a given state. In fact, some valid actions might even be pruned by the planner. However, model checkers are agnostic to planning goals when verifying the state space of a planning domain model; thus, they can still consider valid but unrealistic transitions which might never be considered by a planner while searching for a plan to achieve a given planning goal.

By failing to consider the effects of planners and planning goals on the behaviour of planning domain models, the state-of-the-art verification methods for planning domain models that directly employ model checkers are susceptible to false positive counterexamples, i.e. counterexamples that are unreachable by sound planners when the domain under verification is used for a planning task.

Designers can be misguided to needlessly constrain domain models due to such unrealisable counterexamples, thereby potentially inhibiting necessary behaviours. In addition, false positive counterexamples can lead verification engineers to overlook counterexamples that are reachable by planners. According to the Electronic Engineering Times, a leading technological news website in the electronics industry:

> "When a design is under-constrained, illegal inputs can lead to the formal tool exploring illegal design states. The tool may report false bugs, resulting in the verification team spending time pursuing 'wild-goose chases'. Under-constrained designs can also lead to a false sense of achieving the desired coverage." [15].

To address this shortfall, we propose to use planning goals as constraints during verification, a concept transferred from verification and validation research [16]. Thus, we introduce goal-constrained planning domain model verification, a novel approach that reduces the number of invalid planning counterexamples in the verification of planning domain models.

1.1.2 Thesis Contributions to the Area of the Verification of Planning Domain Models

In the first part of this thesis, I answer the following research questions: What are the shortcomings of the under-constrained application of the state-of-the-art verification methods to the verification of planning domain models? How to perform planning domain model verification of safety properties in a way that increase the robustness of verification methods against false positives?

In answering these questions, this thesis provides five research contributions to the area of the verification of planning domain models. First, we explain the downside of under-constrained planning domain model verification methods and introduce the notion of valid planning counterexamples. Second, we explain how to reduce the numbers of false positive counterexamples from the verification of planning domain models by using planning goals to constrain the verification task. Third, we formally prove

that goal-constrained planning domain model verification of safety properties is guaranteed to return only valid planning counterexamples, if and only if any exist as per the definition of valid planning counterexamples introduced in this thesis. Fourth, we demonstrate how model checkers, as well as state trajectory constraints planning techniques, should be used to verify planning domain models so that the number of invalid planning counterexamples is reduced. Fifth, we perform empirical experiments to demonstrate the feasibility and investigate the behaviour of our approach using the Spin model checker [17] and the MIPS-XXL planner [18].

An initial version of the research reported in this part is presented in the ICAPS-2019 workshop on Knowledge Engineering for Planning and Scheduling (KEPS) [19]. Subsequently, a more advanced version, incorporating empirical experiments, is published in the proceedings of the 9th European Starting AI Researchers' Symposium (STAIRS-2020) [20].

The following section introduces the second research problem investigated in this thesis.

1.2 Planning Domain Model Validation

Despite the recent steady development in Knowledge Engineering in Planning and Scheduling (KEPS) research, the formal validation of planning domain models - not to be confused with planning domain model verification - has received very little attention. Nevertheless, planning domain model validation is one of the key aspects of KEPS. Among other tasks, this activity is concerned with checking the correctness of a planning domain model with respect to a set of requirements. If the requirements are described informally, then the process of validating the domain model is also informal [21]. On the other hand, when the requirements are described formally, it is feasible to perform formal validation.

An example of the applications where a planning domain model and its reference are both given in the same formal language is the evaluation of the quality of automated planning model learning algorithms [22, 23]. For this application, a hand-crafted model is used to generate a number of plans that are fed to the model learning method to produce the learnt planning domain model. Then, if the original and learnt models are functionally equivalent, the learning algorithm can be deemed to be of good quality.

So if a planning domain model and its reference are given in the same formal language, validating the correctness of the planning domain model according to its reference becomes a matter of validating the functional equivalence between the model and its reference. In the thesis, I focus on investigating the functional equivalence between planning domain models.

1.2.1 Research Problem: Functional Equivalence of Planning Domain Models

Two planning domain models are functionally equivalent if both can be used to solve the same set of problems, not necessarily with the same plans. The functionality of a planning domain model is expressed with regard to a set of objects as the reach set of this domain model for this set of objects. The reach set of a planning domain model for a set of objects is the set of all transitions that can be produced from interleaving the actions produced by grounding the operator schemata of the domain model with the

members of the set of objects. As such, two planning domain models are functionally equivalent for a set of objects if they have equal reach sets for this set of objects under a bijective predicate mapping.

Since the reach set of two planning domain models for any set of objects can be infinite, it is impossible to validate the functional equivalence of such domains by comparing the members of their reach sets. Therefore, rather than comparing individual transitions, we propose to compare the structures of the operator schemata of planning domain models.

Comparing the structure of operator schemata can be challenging, especially when the two domain models contain different numbers of operators, and their operator schemata and predicates have different names. To overcome this challenge, we define a special predicate mapping that if exists between the predicates of the two domains, then the two domains are guaranteed to be functionally equivalent.

1.2.2 Thesis Contributions to the Area of the Validation of Planning Domain Models

In the second part of this thesis, I answer the following research questions: How to define functional equivalence between planning domain models independently from planning problems? How to automatically and efficiently check if two planning domain models are functionally equivalent? How does the concept of functional equivalence between planning domain models benefit the KEPS area?

In answering these questions, this thesis provides five research contributions to the area of the validation of planning domain models. First, we formally define the notation of the functional equivalence of two planning domain models. Second, we propose a novel method, called D-VAL, that uses a planner and an SMT solver to check the functional equivalence of planning domain models. Third, we formally prove the soundness and completeness of our method. Fourth, to test and evaluate D-VAL, we develop a random test generation method that modifies given planning domain models to generate functional equivalence validation tasks. Fifth, we introduce a test benchmark that covers a range of published planning domain models and we perform empirical experiments on this benchmark to demonstrate the feasibility, and scalability of our method.

The research presented in this part of the thesis was submitted to the Journal of Artificial Intelligence Research on the 3rd of May 2023. A preprint of this paper is accessible from arXiv [24].

1.3 Thesis Outline

The rest of this thesis is organised as follows. Chapter 2 sets the context of this thesis by first introducing the area of automated planning and explaining the classical planning problem in Section 2.1. After that, an overview of the process of modelling planning domain models and an explanation of the planning modelling language used in this thesis are provided in Section 2.2. Then, I introduce main planning approaches used by classical planners in Section 2.3. After that, I explain the model checking problem in Section 2.4. Following that, Section 2.5 introduces the field of Satisfiability Modulo Theories and explains SMT problems which is used in Chapter 4 in validating the functional equivalence of planning domain models. The verification and validation of planning systems is reviewed in Section 2.6.

Chapter 3 describes invalid counterexamples in planning domain models. After that, this chapter explains the novel method of goal-constrained verification of planning domain models and demonstrates the application of this new method using model checking and planning techniques. Afterwards, this chapter describes the empirical experiments that evaluate the overhead cost of our constrained verification method. Next, a recommendation for designing inherently safe planning domain models is provided. Then, the contribution of this chapter is contrasted with related work.

Chapter 4 defines planning domain models functional equivalence and contrasts the problem of validating functional equivalence of planning domain models with related work. Then, this chapter differentiates between simple and complex planning domain models and explains the validation method for each type. After that, this chapter proposes a new planning domain models functional equivalence validation method, called D-VAL, that uses a planning-based approach to remove redundant operators and an SMT solver to prove the functional equivalence of planning domain models. Afterwards, this chapter describes the empirical experiments that showcase the performance of D-VAL through 75 validation tasks.

Chapter 5 summarises the contributions of this thesis to the verification and validation of planning domain models and discusses directions for future work.

Appendix A contains the proofs of the theorems presented in Chapter 4. Finally, Appendix B describes the modifications applied to the selected planning domain models to produce the validation task in Section 4.14.

Since this research is focused on “planning domain models”, this term will be repeated many times throughout this thesis. Therefore, to improve readability, this term is sometimes shortened to “domain model” or “domain”. Especially in Chapter 3 and Chapter 4. This abuse of the notation dose not cause any confusion between domain models and real-world application domains as this thesis mostly uses the word “domain” to refer to a planning domain model. In the few cases where the word “domain” is used to refer to a real-world application domain, the context clarifies this reference. Otherwise, I use the term “application domain” to unambiguously refer to real-world application domains.

CHAPTER 2

BACKGROUND

This chapter aims to set the context for this thesis and provides essential background knowledge on the verification and validation of planning domain models. It begins with an introduction to automated planning, accompanied by an explanation of a mathematical representation of classical planning problems. The chapter proceeds with an overview of Knowledge Engineering in Planning and Scheduling (KEPS) research, emphasising the core focus of this thesis: the verification and validation of planning domain models. This overview gives particular attention to the Planning Domain Definition Language (PDDL), the modelling language targeted by the methods developed in this thesis.

This thesis employs planning, model checking, and SMT solving methods to verify and validate planning domain models. Consequently, this chapter presents brief overviews of these methods. Firstly, it presents the main planning approaches utilised in classical planning. Following that, this chapter introduces model checking and provides an example of applying a model checker to verify a finite state system against a set of safety properties. Furthermore, it introduces the research field of SMT, highlighting how constraint problems can be formulated as SMT problems.

Lastly, this chapter offers a brief survey of the verification and validation of the main parts of planning systems. Comprehensive reviews of relevant literature pertaining to the specific topics covered in this thesis are further provided in the respective chapters.

2.1 Introduction to Automated Planning

Automated planning is the branch of Artificial Intelligence that studies the process of searching for actions that, when applied in a specific order, can transform a system from a given state to a desired state [25]. This process of searching is called planning. Planning is performed by a planner, which is an intelligent search algorithm that takes as inputs the available actions, the initial state, and the goal state. In the realm of domain-independent planning, the input to planners is a planning task.

A planning task consists of an initial state, a goal condition and a planning domain model. Planning domain models capture the knowledge about a domain in terms of abstract models of the available actions. An action model specifies the conditions that must be met for an action to be applicable and the effects of the action on the states of the world. The objective of the planner is to find a sequence of actions that transforms the world from its initial state to a state that satisfies the given goal condition.

Consider the Blocks World planning domain model. This domain describes a world that consists of a finite number of blocks. These blocks are either stacked into towers or placed directly on a table. A solution to a planning problem in this world changes the arrangement of these blocks from a given configuration, called the initial state, to a desired one, called a goal state.

For example, an initial state can be Block C on Block B, which is on Block A, and Block A on the table. A goal state might require the blocks to be staked in a reversed order. Planning goals do not have to specify unique word states. Starting from the previous initial state, we may want Block B to be placed directly on the table with no other blocks on top of it. This goal disallows the other blocks to be on top of Block B, but, otherwise, it ignores the positions of the other blocks. In this case, the planning goal can be described as the condition: a state is a goal state if it has Block B on the surface of the table with no other blocks on top of it. This planning condition can be satisfied by many world states rather than just one state.

For an agent to achieve planning goals, it must be provided with a model of the actions that change the state of the world. In this example, the actions available to the agent are Pick-up, Put-down, Stack and Unstack. The action Pick-up causes the agent to get hold of block X if the agent is not holding other blocks, and block X is on the table, and it is clear, i.e. it has no blocks on top of it. On the other hand, the action Put-down enables the agent to put a block on the table if the agent is holding it. The Stack action allows the agent to put block X on top of block Y if the agent holds block X and block Y is clear. Conversely, the action Unstack causes the agent to take block X from the top of block Y if the agent is not holding any blocks and block X is clear.

Each one of these actions has preconditions and effects that specify how the action interacts with the world. For example, an action's precondition can be “Block X is clear” , and an action's effect is “put block X on top of block Y”. These actions are specified in a planning domain model using special modelling languages, explained in Section 2.2.2.

As mentioned in Chapter 1, the scope of this thesis is the verification and validation of planning domain models in classical planning. The area of classical planning assumes that actions have deterministic outcomes, meaning that the effects of an action are predictable and do not depend on uncertain factors or external influences. It also assumes that the state of the world is fully observable, meaning that the planner has complete knowledge of the current state at any given time. Additionally, the environment is discrete, meaning that there are a finite number of possible states and actions. To facilitate the discussions in the main chapters, first, I explain the mathematical representation of classical planning problems used in this thesis.

2.1.1 Classical Representation of Planning problems

The formal representation of classical planning used in this thesis follows the classical representation presented by Ghallab et al. [25]. In classical planning, world objects are represented by unique constants which are the elements of a finite set *Obj*. The properties of the objects are described using atoms. Atoms consist of predicate letters that are applied to a set of variables. A ground atom has its variables assigned to object constants from *Obj*. Let P be a finite set of the predicates that appears in the atoms that describe the status of all the objects of a world.

A state of the world, s, is defined by the status of all objects in this world. The status of an object is defined by the truth evaluation of its atoms. The state space of a world is the set S of all states, where $|S| = 2^{\{\text{number of all ground atoms}\}}$. False atoms are not included in the state definition and a closed-world assumption is made.

A planning domain model is a tuple $D = (P, O)$, where P is a set of predicates and O is a set of planning operators. A planning operator, o, is a tuple $o = (name(o), Pre(o), Add(o), Del(o))$ where $name(o) = op\ name(x_1, \ldots, x_k)$, "*op name*" is a unique operator name and $x_1, \ldots, x_k$ are variable symbols (arguments or parameters) appearing in the operator's atoms, and $Pre(o), Add(o)$ and $Del(o)$ are sets of ungrounded atoms, specifying preconditions, add effects and delete effects of the operator o, respectively. The intersection of the precondition and add effects of an operator must be empty. The intersection of the add effects and delete effects of an operator must be empty. The delete effects of an operator must be a subset of the preconditions of that operator. The atoms in operators consist of predicate letters from P that are applied to variables taken only from $x_1, \ldots, x_k$.

An action is a ground instance of an operator. Actions are instantiated from operators by grounding the atoms of the operators, i.e. by substituting their variables with objects. Let A be a set of all actions instantiated from the set of operators O using the set of objects *Obj*. An action name is the same as the name of the operator from which the action has been instantiated. An action's preconditions, add effects and delete effects are sets of ground atoms. Action a is applicable in state s if and only if $Pre(a) \subseteq s$. The application of a in s results in the successor state $\gamma(s, a) = (s \setminus Del(a)) \cup Add(a)$ provided that a is applicable in s.

A planning problem is a tuple $\mathcal{P} = (D, Obj, s_0, g)$, where $s_0 \in S$ is the initial state of the world, g is the planning goal expressed as a set of ground atoms. Let S_g be the set of all states that evaluates the set of ground atoms in g to true. In other words, S_g is the set of all states that satisfies the planning goal g. A sequence of actions $\pi = \langle a_0, a_1, ..., a_n \rangle$, where $a_i \in A$, is a solution to the planning problem $\mathcal{P}$ if $s_n \in S_g$, where $s_n = \gamma(s_{n-1}, a_n)$. The sequence π is called a plan that solve the planning problem $\mathcal{P}$. Furthermore, the goal is only satisfied after applying the last action in the plan. Formally, $\forall a_i \in \pi \setminus \{a_n\} \implies \gamma(s_{i-1}, a_i) \notin S_g$. We also define the state-transition system of the planning problem, $\mathcal{P}$, as $\Sigma = (S, A, \gamma)$.

In this section, I have explained the mathematical representation of the classical planning problem used in this thesis. The following section provides an overview of the research area of KEPS, which aims to develop effective ways to describe planning domain models and problems.

2.2 Knowledge Engineering in Planning and Scheduling (KEPS)

Knowledge engineering for automated planning is the process of producing and maintaining planning domain models. The former task is concerned with acquiring planning knowledge and formulating it in a planning domain model. This task also incorporates the activity of validating the produced planning domain models to ensure their correctness. The latter task aims to maintain planning domain models' accuracy [1]. This process is depicted in Figure 2.1. Moreover, knowledge engineering for

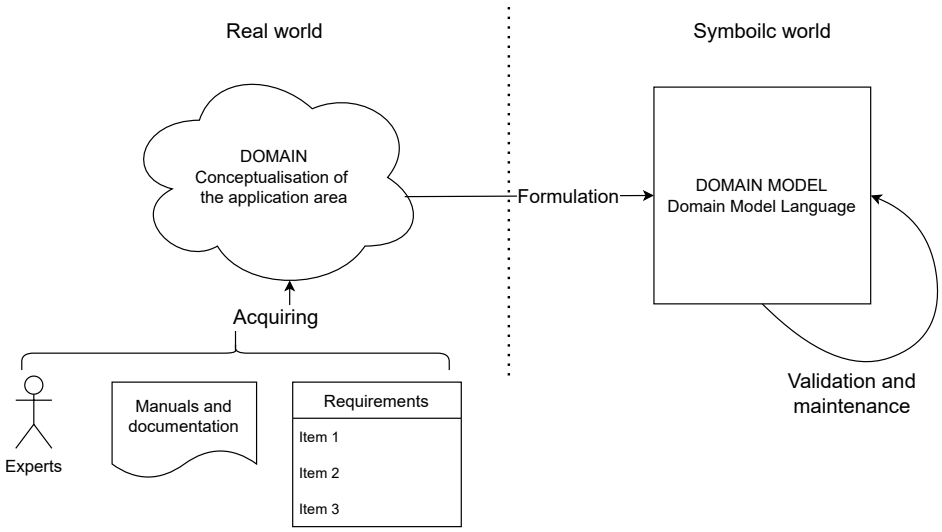


Figure 2.1: The knowledge acquisition, formulation, validation and maintenance parts of the KEPS as defined by McCluskey et al. [1]. (This pictorial representational is inspired from [2].)

automated planning includes selecting and optimising the tools that support the other planning knowledge engineering tasks.

The importance and need to develop tools for KEPS are reflected in the ICAPS workshops on Knowledge Engineering for Planning and Scheduling, which have been organised every year since 2017 until the date of writing this thesis (2023) in addition to 2008, 2010-2011, and 2013-2014. Among others, this specialised workshop covers topics like methods and tools for the acquisition of domain knowledge, formulation of domains and problem descriptions, formal languages for domain description, and domain model, problem and plan validation. The initial version of the research reported in Chapter 3 about goal-constrained planning domain models verification for safety properties was presented in ICAPS KEPS 2019 workshop [19].

To promote domain modelling, ICAPS also hosts the International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS). These competitions aim to encourage the development of knowledge engineering tools to facilitate designing reliable and efficient planning domain models. The ICKEPS was organised in 2005, 2007,2009, 2012 and 2016. The participants competed in designing planning knowledge engineering tools, as well as off-site and on-site modelling and demonstration. The reader is referred to the review by Vallati and Chrpa [26], where the authors

explained the format of the ICKEPS and their evaluation criteria, analysed the strengths and weaknesses of ICKEPS formats, and suggested alternative formats for future competitions.

The validation of planning domain models is one of the main categories of the tools design competitions in the ICKEPS and a primary topic of the KEPS workshops. Due to the significance of the research area of planning domain model validation, three specialised workshops on the Verification and Validation of Planning Systems (VVPS) were organised in 2005, 2009, and 2011. The objective of the VVPS workshop is to foster continuous interaction between the Planning and Scheduling (P&S) and Verification and Validation (V&V) communities. It aims to promote the exploration of innovative tools and methodologies, as well as the identification of open issues and real challenges. Some of this workshop's topics of interest are V&V of domain models using model checking and application of P&S techniques to V&V. Some aspects of these topics are investigated in Chapter 3.

The next section explains the process of modelling planning domains.

2.2.1 Planning Domain Modelling Process

As described in the previous section, the planning domain modelling process consists of knowledge acquisition, formulation, validation and maintenance. These topics are discussed in the following sections.

2.2.1.1 Knowledge Acquisition

Knowledge acquisition is the first step in modelling planning domain models. In this task, designers use requirement engineering techniques such as document analysis and structured interviews with stakeholders like users and experts to build a conceptualisation of the application area. Based on this requirement analysis, the designers can choose the best planning domain modelling language that can express the application domain and the suitable planning algorithms that can solve the planning problems of this application. The acquired knowledge describes the domain structure, which identifies the relevant object types in the domain and their properties and relations. Moreover, the domain structure lists the actions available to the agent and describes how these actions change the status of the application's objects.

2.2.1.2 Knowledge Formulation

In the context of automated planning, knowledge formulation is the process of describing the knowledge about the domain of the targeted application in a well formatted model. Broadly speaking, a domain model is a declarative description of the types of objects in the world of the application, the properties of and the relation between these objects, and the actions of an agent that can act in the world. Domain models are described by standard planning domain modelling languages. Section 2.2.2 provides an overview of some of these modelling languages. These languages differ in their expressiveness. Thus, designers must formulate domains into models with a proper level of expressiveness. More expressive languages can be used to design models that capture domains more accurately, but planners deliberating

with such models require more computation. On the other hand, planners need less computation to reason about models developed with less expressive languages, but they lose on accuracy [27].

Before a domain model is ready, it must be validated to be correct with respect to its specification. The following section discusses the importance of the validation of domain models.

2.2.1.3 Model Validation

The validation of domain models increases confidence in the correctness of the models with respect to their specifications. McCluskey et al. [28] define three attributes of domain models: accuracy, consistency, and completeness. The authors' intention for these attributes is to contribute to the overall validation of planning systems. They define model accuracy based on actions' assertions. A domain model implies a set of assertions, an assertion for each action. The assertion of an action asserts that if the preconditions of this action are satisfied, then executing it causes its effects to be realised. A domain model accurately represents its requirements, if its assertions are mapped to assertions in these requirements.

The second attribute is consistency. A domain model is consistent if none of its assertions violates domain invariants. Note that the consistency attribute of domain models is only defined if the application requirements specify domain invariants. An example of simple invariant is that object's properties and relations, and their complements cannot be true in same states. Note that the consistency attribute of planning domain models is a special case of the accuracy attribute because if the requirements of a domain are consistent, then the assertions of the planning domain model that accurately captures these requirements cannot violate the domain invariants.

The third attribute is completeness. A domain model is complete if any plan formed from the ground operators of the model is an acceptable plan in the requirement, and the converse is true. In other words, if a plan is valid in the requirement of a planning domain, then it is also valid in the planning domain model.

In addition to the three aforementioned attributes, we define the concept of functional equivalence of planning domain models. Functional equivalence is a relation between two planning domain models or a domain model and its specification. Our functional equivalence relation relaxes the completeness attribute defined by McCluskey et al. [28] by requesting same problems must be solvable by the domain models and their specifications, not necessarily by the same plans as in complete domains. Chapter 4 is dedicated to explanning our approach to validating the functional equivalence of planning domain models.

2.2.1.4 Models Maintenance

The problem of planning domain model maintenance assumes application domains change over time and seeks to update planning domain models to bridge the drift between them and their application domains [29]. An important part of model maintenance is to revalidate that the new model still correctly models the application domain [30]. The need for domain model validation in the post-design phase of the planning domain model's life cycle further emphasises the importance of planning domain model verification and validation.

2.2.2 Planning Domain Modelling Languages

This section provides an overview of the descriptive languages used to model planning domain models and planning problems in this research.

2.2.2.1 Stanford Research Institute Problem Solver (STRIPS) Formalism

The Stanford Research Institute Problem Solver (STRIPS) [31] was introduced in 1971 as a problem-solving program. In STRIPS, world states are represented by a set of well-formed formulas (wffs) of the predicate logic. STRIPS makes the closed-world assumption; hence, any wff not in a state is considered false. For any world model, STRIPS assumes that there exists a set of actions that can affect the objects of the world. The available actions are lifted and grouped into operator schemata. Each operator schema is defined by the name of the operator, a set of parameters, the add list and delete list of its effects, and the preconditions under which the operator is applicable. The effects and preconditions are sets of wffs. An action is instantiated from an operator schema by substituting its parameters with object constants. An action is applicable in a state if its preconditions are a subset of the wffs in that state. The applied action adds the wffs in its add list to the old state and deletes the wffs in its delete list from the old state to produce a new state. A solution to a STRIPS problem is a sequence of ground operators that transform an initial state into a state that satisfies a goal condition.

2.2.2.2 Action Description Language (ADL)

Action Description Language (ADL) [32] introduced in 1989 extends the expressiveness of STRIPS representation by augmenting its operator schemata with disjunctive, quantified and negative preconditions, and conditional effects. The extended syntax in ADL language facilitates designing planning domain models that are more compact than the models produced using STRIPS for the same planning domains.

Nevertheless, ADL domain models can be compiled into STRIPS models. The fourth International Planning Competition provides a tool called “adl2strips”[1] that translates ADL planning domain models into STRIPES planning domain models [33]. This tool is based on the FF planner's preprocessor developed by Hoffmann and Nebel [34] and uses either of the methods proposed by Gazen and Knoblock [35] or Nebel [36] to compile conditional effects away.

2.2.2.3 Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language (PDDL) [37] was introduced by the Planning Competition Committee of the International Conference on Artificial Intelligence Planning Systems (AIPS) in 1998. It emerged from efforts to create a common language for specifying planning problems and facilitating the exchange of planning benchmarks to support meaningful comparisons of the performance of planners.

[1] adl2strips can be downloaded from the IPC-4 web page at `https://ipc04.icaps-conference.org/deterministic/`

Since then, PDDL has become the de-facto standard language for representing classical planning tasks. We also use PDDL for modelling planning domains and problems to ensure compatibility and seamless integration with existing planning tools.

Over time, PDDL has been extended to incorporate more expressive features. The first significant extension is PDDL 2.1 [38]. This version includes numeric fluents that model numeric resources, durative actions to model temporal properties of planning domains, and plan metrics to specify plan quality measures. From PDDL 2.1, the language was extended to add support for derived predicates and timed initial literals in PDDL 2.2 [39]. To increase the expressiveness of PDDL about plan quality specification, PDDL was extended to express strong and soft constraints on plan trajectories in PDDL3.0 [40]. We use the feature of plans' strong constraints to formulate planning domain model verification as a planning problem in Chapter 3. After that, version PDDL3.1 introduced the feature of object fluents, which allows the definition of functions that ranges not only on numerical values, but also on object types [41].

Planners do not have to support all PDDL extensions and features. Therefore, PDDL allows designers to specify the set of required features using the flag *:requirements*. For instance, a planning domain model with durative actions should specify the requirement *:durative-actions*. The complete set of the allowed requirement flags, along with the syntax of PDDL3.0, is provided in [42].

To define a planning problem in PDDL, we have to define the following descriptors of this problem:

1. The set of objects used in the planning problem. These are the objects in the world of concern to the planning problem.

2. The set of predicates that describe properties or characteristics of individual objects or combinations of objects. Predicates are expressed by the name of the attribute of the predicates and the arity of this attribute.

3. Predicates are the templates for formulating atoms which have sets of lifted variables as per the arity of their predicates.

4. Operators represent the rules that change the state of the world. They are defined over a set of parameters. The preconditions of an operator are made from a set of atoms. The effects of an operator are divided into add and delete effects. These effects are also represented by sets of atoms.

5. The initial state of the system. A state of the world is represented by the truth evaluation of the ground atoms produced from all lifted atoms and all possible objects. True ground atoms mean the attribute expressed by the predicate of the atom applies to the object listed in the ground atom. On the contrary, a false ground atom means the attribute described by the predicate of the atom does not apply to the objects in the ground atom. To avoid stating all false ground atoms, PDDL follows the closed-world assumption, i.e. if any ground atom is not mentioned in an initial state, this atom is considered to be false.

6. Planning goal condition is expressed as a partial state. A partial state only specifies the ground atoms that must be true in any goal state. Ground atoms not listed in the planning goal can be either true or false in goal states.

A PDDL planning task consists of two parts: a planning domain model and a planning problem description. The planning domain model specifies the set of predicates and operators that define a planning domain independently from individual planning problems. On the other hand, the planning problem lists the objects, initial state and goal condition that define a specific planning problem instance. For example, we will consider the Blocksworld domain introduced in Section 2.1. The PDDL model of this planning domain is shown in Listings 2.1 to 2.5, and the PDDL description of an example of a planning problem for this domain is shown in Listing 2.6.

Note that as part of this planning domain model, Listing 2.1 defines the name of the domain as "Blocksworld" and lists the features of PDDL used in defining this domain as":strips" and ":typing", the types of the domain's objects as "block", and the predicates that describe the status of these objects as "on", "ontable", "clear", "holding" and "handempty". The definition of predicates defines the arity of the predicates, and because we have specified the PDDL feature :typed, the predicate definition also declares the types of the parameters of the atoms that will be created from these predicates.

```
1 (define (domain Blocksworld)
2   (:requirements :strips :typing)
3   (:types block)
4   (:predicates (on ?x - block ?y - block) (ontable ?x - block) (clear ?x - block)
      ↪ (handempty) (holding ?x - block))
```

Listing 2.1: The PDDL definition of Blocksworld domain requirements,types, and predicates.

The parameters of the action "stack", its preconditions, add effects and delete effects are defined in Listing 2.2. The delete effects are the atoms passed as parameters to the predicates "not" in the effects of the operator.

```
1 (:action stack
2  :parameters (?x - block ?y - block)
3  :precondition (and (holding ?x) (clear ?y))
4  :effect (and (not (holding ?x)) (not (clear ?y)) (clear ?x) (handempty)
5               (on ?x ?y)))
```

Listing 2.2: The declaration of the operator stack in PDDL.

```
1 (:action unstack
2  :parameters (?x - block ?y - block)
3  :precondition (and (on ?x ?y) (clear ?x) (handempty))
4  :effect (and (holding ?x) (clear ?y) (not (clear ?x)) (not (handempty))
5               (not (on ?x ?y)))))
```

Listing 2.3: The declaration of the operator unstack in PDDL.

```
(:action pick-up
 :parameters (?x - block)
 :precondition (and (clear ?x) (ontable ?x) (handempty))
 :effect (and (not (ontable ?x)) (not (clear ?x)) (not (handempty)) (holding ?x)))
```

Listing 2.4: The declaration of the operator pick-up in PDDL.

```
(:action put-down
 :parameters (?x - block)
 :precondition (holding ?x)
 :effect (and (not (holding ?x)) (clear ?x) (handempty) (ontable ?x)))
```

Listing 2.5: The declaration of the operator put-down in PDDL.

The Blocksworld problem introduced in Section 2.1 described an initial state where Block A on the table, Block B is on Block A, and Block C is on Block B. As a goal state, we want the blocks to be staked in a reversed order. The PDDL specification of this problem is shown in Listing 2.6.

```
(define (problem Blocksworld_problem)
  (:domain Blocksworld)
  (:objects A, B, C - block)
  (:init (on C B) (on B A) (ontable A) (clear C) (handempty))
  (:goal (and (on A B) (on B C))))
```

Listing 2.6: The PDDL definition of a Blocksworld problem.

The PDDL description of this problem specifies the name of this planning problem instance, the planning domain model for which this planning instance is defined, the objects of interest in this planning problem and their types, the initial state of these objects and the planning goal. PDDL adopts the closed-world assumption; therefore, all ground atoms not mentioned in the initial state are assumed to be false. The planning goal condition specifies the ground atoms that must be true in any goal state. Ground atoms not listed in the goal condition can be either true or false in goal states.

To get a solution to this planning problem, we run the FF planner (explained in the next section) with the Blocksworld planning domain model and this planning problem as its inputs. The planner found the pan shown in Listing 2.7.

```
UNSTACK C B
PUT-DOWN C
UNSTACK B A
STACK B C
PICK-UP A
STACK A B
```

Listing 2.7: The PDDL definition of a Blocksworld problem.

Finding the plan presented in the previous example was relatively straightforward due to the simplicity of the planning domain model, and the small number of objects involved in the planning problem.

However, as the complexity of the domain increases and problems become larger in scale, planning becomes increasingly challenging. To tackle these challenges, planners employ sophisticated algorithms designed to efficiently search for solutions to complex and significant planning problems. The following section briefly reviews some prominent planning techniques that address these complexities and enable effective solution discovery.

2.3 Planning Techniques

Domain-independent classical planners employ various planning techniques to search for plans in different search spaces, such as state space [34, 43], plan space [44], planning graphs [45], and Hierarchical Task Network (HTN) planning [46].

First, this section explains the state-space search, the most straightforward planning algorithm. Then, it describes the plan-space planning and planning graphs algorithms. After that, it briefly introduces the HTN planning framework. The planner used in this thesis performs state-space search planning. The other planning techniques are explained to give a broader overview of the main classical planning algorithms.

In state-space search, nodes represent states of the world's objects, and edges represent the application of planning actions. The main state-space search algorithm is the forward search algorithm, which searches for a sequence of transitions from an initial state to a goal state in the planning problem's state-transition space. During the state space exploration, the algorithm terminates if the current state is a goal state; otherwise, it determines which actions are applicable to this state. Then, the algorithm chooses an action and computes the successor state that results from applying this action to the current state. Examples of forward state-space planners is the FF planner [34] and Fast Downward [43].

Alternatively, planning can be approached through a backward search methodology. In this approach, the process begins from the goal condition and finds the set of actions that can achieve part of this condition without falsifying its other parts. Then, the algorithm chooses one of these relevant actions and updates the goal condition by removing the conditions satisfied by the effect of the selected action and adding the preconditions of this action as literals to the goal. The algorithm repeats the previous steps until the produced goal condition is satisfied by the initial state. For instance, the planner Hybrid STAN [47] selects between forward and backward search based on analysing the planning domain model.

This thesis does not only support the adoption of automated planning techniques by proposing methods to verify and validate planning domain models, but also employs planners to solve some of the challenges faced in this quest. Chapter 3 explains how to formulate the goal-constrained planning domain model verification task of safety properties as a planning problem. We use the planner MIPS-XXL [18] to solve this planning problem. MIPS-XXL translates the verification-as-planning problem into a standard planning problem and invokes the planner FF to solve it. We chose MIPS-XXL because it supports the strong constraints in PDDL3.0. In Chapter 4, we use the FF planner to check if a planning operator is redundant in its planning domain model. This is an essential step in validating the functional equivalence of planning domain models. The planner FF support negative preconditions and conditional effects;

these features are used to formulate the planning problem of checking the redundancy of the operators of planning domain models.

The FF (Fast-Forward) planner employs a forward search approach in the planning state space, guided by a heuristic function derived from the planning domain model of the given problem. This heuristic function takes the current state as input and computes a heuristic value, which serves as an estimate of how close the current state is to a goal state. The heuristic value guides the search algorithm to prioritise the exploration of states that are more likely to lead to a solution.

The heuristic function used in FF is based on a relaxed search algorithm, which disregards the delete effects of actions. By considering only the positive effects of actions, FF derives a relaxed plan from the current state to the goal state. The length of this relaxed plan represents the distance from the current state to the goal state. The calculated heuristic is then used by the enforced hill-climbing search method to select which actions to schedule from the current state. This method searches for states with lower heuristic values among the successor states and their subsequent successors. In other words, it explores the states estimated to be closer to the goal state.

Furthermore, FF restricts the choice of actions to the set of helpful actions in each state. These helpful actions are applicable in the given state and contribute to the generation of the relaxed plan. By focusing on actions deemed beneficial in the current state, FF narrows down the search space and explores action sequences that have the potential to lead to a solution. By combining the relaxed planning approach, enforced hill-climbing search, and the selection of helpful actions, FF effectively navigates the planning state space, incrementally moving closer to the goal state. This heuristic-guided search strategy allows FF to find solutions efficiently in a variety of planning domain models.

2.3.1 Other Planning Techniques

This section provides an overview of other planning techniques, such as plan space, plan graphs and HTN planning. In plan-space search, planners search through a graph of partial plans, where nodes represent partially specified plans and arcs represent plan refinement operations. An arc in this graph transforms the current plan into a successor plan as an effect of the applied refinement operation. Refinement operations consist of adding an action to the current partial plan, adding an ordering constraint to restrict one action to be scheduled before another, and adding a causal link to relate a precondition of an action to an add effect of another action.

Planning within a plan space involves exploring the partial plans graph to find a path from a node representing an initial partial plan to a node representing a solution plan. In each step of this process, the planner selects and applies a refinement operation on the current partial plan to produce another plan that is more capable of achieving the planning goal. In plan-space planning, solutions are partial-order plans. To satisfy a planning goal, a partial-order plan must ensure that all the preconditions of the actions are met before they are executed and that the effects of the actions collectively achieve the desired goal conditions. However, a partial-order plan allows for different valid orderings of some actions. This means the plan can be executed successfully regardless of the specific order in which these unordered actions are performed.

The plan-space planning paradigm is closely related to planning graphs. A planning graph is a directed layered graph that consists of alternating layers of action and proposition levels. The first layer is the layer of the propositions of the initial state, and the second layer is the layer of the actions applicable in the initial state. The third layer, which is a proposition layer, contains the union of the positive effects of the actions from the previous layer and the propositions of the previous proposition layer. There are arcs to connect the preconditions of the actions in an action layer to the propositions that support these preconditions from the previous proposition layer. Also, there are arcs that connect the add effects of each action from an action layer to the propositions supported by these add effects in the following proposition layer. Similarly, any proposition falsified by an action in the previous action layer is connected to the delete effects of this action.

The Graphplan algorithm, introduced by Blum and Furst [45], iteratively expands the planning graph until reaching a proposition layer that contains all the propositions of the planning goal, and and no pair of these propositions is a mutex. Two propositions are mutex if the action that satisfies one proposition causes the other to be falsified. After that, it searches backwards from the last level of this graph for a plan. Like in the backward search algorithm, Graphplan first selects an unmutex action that satisfies the preconditions of the planning goal. Then, it searches in the previous action layer for actions that support the preconditions of the action selected in the previous step. This process continues until the algorithm reaches the initial layer. The actions selected during the backward search represent the plan found by Graphplan.

HTN planning is similar to planning graphs and plan-space planning in terms of searching in the space of plan-refinements rather than explicitly searching in the state space of the world's states. However, HTN planning searches for how to achieve a high-level task by decomposing it into subtasks and decomposing these subtasks until all tasks are decomposed into primitive tasks. Primitive tasks are atomic actions that can be directly executed by an agent in the world without further deliberation. Though HTN performs planning in classical settings, this algorithm attempts planning in a very different way from the traditional classical planning methods [48].

This section outlined some of the main planning techniques in the literature. In the following section, I introduce model checking, one of the most important formal verification methods, which it is used in the third chapter of this thesis to perform robust verification of planning domain models.

2.4 Model Checking

Model checking is a formal verification technique that checks the correctness of models of finite-state systems. It is an automated process that exhaustively explores a system model's states to verify if certain temporal properties hold true. By exhaustively examining the state space, model checkers can identify potential errors or violations in the system design. Temporal properties express properties that depend on the order, sequence, or timing of events or states in the system's execution. Temporal properties can represent either safety or liveness properties. A safety property specifies that something bad must

never happen in a system. On the other hand, liveness properties mandate that something good must eventually happen at some point during the system's execution.

Some of the most important model checkers include Spin [17], NuSMV [49], UPPAAL [50], and PRISM [51]. Spin and NuSMV use different verification techniques to verify finite state systems against temporal logic properties. UPPAAL verifies systems that can be modelled as timed automatons. PRISM is designed to analyse probabilistic systems with regard to properties expressed in probabilistic temporal logic. This thesis explains how to use Spin to perform goal-constrained planning domain model verification mainly because the under-constrained verification method that motivated our research also used Spin. Spin (Simple Promela Interpreter) is a widely used model checker that accepts models specified using Promela. Promela, which stands for "Process Meta Language", is specifically designed for modelling concurrent systems by specifying the behaviour of systems using concurrent processes and other concurrent constructs like communication channels.

Model checkers takes as inputs a model of the system under analysis and the properties that need to be verified. Then they search the state space produced from interleaving the processes defined in the model to find a violation of the given property. If such a violation is found, the model checker returns the sequence of transitions that caused the violation as a counterexample to the correctness of the property. The properties are expressed using formal logic such as Linear Temporal Logic (LTL) [52]. LTL provides several temporal operators to specify the ordering and relationships between events. The most commonly used operators in LTL are:

1. Always ($\Box$ or **G**): This operator states that a property should hold at all future states. For example, the formula ($\Box\, p$) holds in a state if the property p holds in this state and all following states.

2. Eventually ($\Diamond$ or **F**): This operator states that a property should hold at some point in the future. For example, the formula ($\Diamond\, p$) holds in a state if the property p holds at some future state.

3. Next ($\bigcirc$ or **X**): This operator states that a property should hold in the next state. For example, the formula ($\bigcirc\, p$) holds in a state if the property p holds in the next state.

4. Until ($\mathcal{U}$ or **U**): This operator states that a property p must continuously hold at least until another property q becomes true. For example, the formula ($p\ \mathcal{U}\ q$) holds if the property q holds at some future state and that property p holds from the current state at least until the property q becomes true.

Consider the FSM of the simple traffic light system depicted in Figure 2.2. The Promela model of this FSM is shown in Listing 2.8. The initial state of this FSM is specified in line number 1. The transitions t_0, t_1, t_2, and t_3 are described in lines four to seven, respectively. An example of safety properties is "once red, the light cannot become green immediately", in LTL: $\Box(Red \implies \neg \bigcirc Green)$. We can add more details to this safety property by specifying that the traffic light must transit from red to amber to green. The refined property would be "once red, the light always becomes green eventually after being amber for some time", which is expressed in LTL as $\Box(Red \implies (\Diamond Green \wedge ((\neg Green)\mathcal{U} Amber)))$.

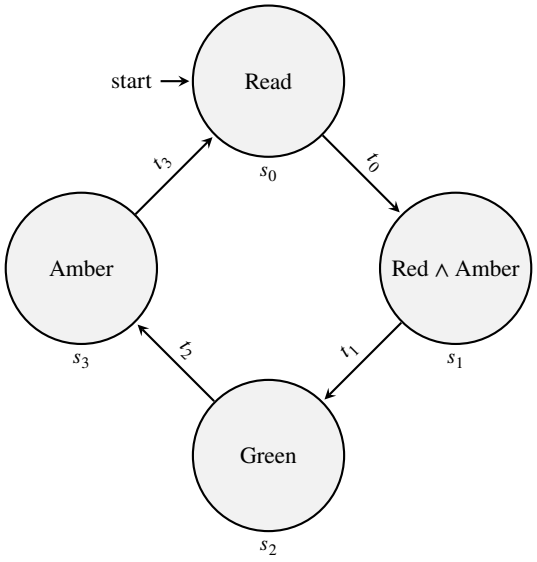


Figure 2.2: Traffic Light FSM

Another important safety property is "the light cannot be red and green at the same time", which is expressed in LTL as $\Box(\neg(Red \wedge Green))$.

```
1  bool  Red = 1,Amber = 0,Green = 0;
2  proctype TrafficLight() {
3   do
4   :: atomic {(Red == 1)-> Amber = 1}
5   :: atomic {((Red == 1) && (Amber == 1))->  Green = 1; Red = 0; Amber = 0}
6   :: atomic {(Green == 1)-> Amber = 1; Green = 0}
7   :: atomic {(Amber == 1)-> Red = 1; Amber = 0}
8   ::  else -> skip
9   od;
10 }
11 init {
12  atomic{run TrafficLight()}}
```

Listing 2.8: Promela model of the traffic light FSM.

```
1  ltl p0 {[](Red -> ! X(Green))}
2  ltl p1 {[](Red -> (<>Green && ((! Green) U Amber)))}
3  ltl p2 {[] !(Red && Green)}
```

Listing 2.9: Promela deceleration of the traffic light properties.

These properties are declared in Promela syntax in Listing 2.9, where [] represents the temporal operator always, <> represents the temporal operator eventually, X represents the temporal operator next, and U represents the temporal operator until.

SPIN interleaves the atomic transactions specified in the given Promela model to produce the automaton that describes the the state space of the traffic light system. To verify a given property,

SPIN converts the negation of the property's LTL formula into a Büchi automaton, and computes the synchronous product of the automata of the model and the negation of the property. If the language accepted by the produced automaton is empty, i.e, there is no sequence of transitions can falsifies the given property. On the other hand, if the language is not empty, there is a sequence of transitions that falsifies the property.

Running Spin to verify these properties confirms that p_1 and p_2 are satisfied by the provided Promela model, but not p_0. Intuitively, the property p_0 is meant to capture the requirement that there is no direct transition from state s_0 to state s_2 in Figure 2.2, i.e. the traffic light does not change from red to green directly before becoming amber first. However, p_0 actually says there is no transition from a state where *Red* is true to a state where *Green* is true. As a counterexample to this property, Spin produced this trace $\langle t_0, t_1 \rangle$, which causes the FSM of the traffic light to move from s_1, where *Red* is true to s_2, where *Green* is true. Thus, the model does not satisfy property p_0.

Note that property p_0 is not satisfied by the model, not because there is a design error in the model, but because the property is not well specified. Therefore, we must correct this property by adding the condition $\neg Amber$ to the antecedent of its implication. Thus, the corrected property becomes $\square((Red \wedge \neg Amber) \implies \neg \bigcirc Green)$. Listing 2.10 shows the corrected property in Promela.

```
1 ltl p0 {[]((Red && !Amber) -> ! X(Green))}
```

Listing 2.10: Promela deceleration of the correct p_0 of the traffic light FSM.

Now all properties are satisfied by the traffic light FSM; hence, Spin does not return any counterexample for these properties.

This section briefly introduced model checking, LTL logic and Promela modelling language. These techniques are used to describe our goal-constrained planning domain model verification method in the third chapter of this thesis. The following section describes satisfiability modulo theories solvers, which are used in the process of validating the functional equivalence of planning domain models in the fourth chapter of this thesis.

2.5 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the research field concerned with finding assignments to the variables of first-order formulas such that these assignments cause these formulas to be evaluated to be true. The symbols that appear in these formulas are interpreted with respect to some background theories. For instance, in the Integer theory, the add sign is interpreted over integer numbers in the usual way.

A good example of an SMT problem is the rectangle packing problem. Given a big rectangle and a number of smaller rectangles, how to fit the smaller rectangles in the big one such that no two small rectangles overlap? This problem is known as two-dimensional bin packing [53]. This is a challenge faced by poster printing companies [54].

The constraints that capture the requirements of this problem can be modelled using a modelling language called SMT-LIB. This is a specification language for SMT solvers that defines a syntax for

specifying background theories and expressing logical formulas, constraints, and commands to interact with solvers. The purpose of SMT-LIB is to simplify the comparison of SMT solvers. It achieves this by offering standardised modelling language for SMT solvers making available to the research community a vast collection of benchmark problems for SMT solvers [55].

To model a rectangle packing problem, we have to express the coordinates of the Lower Left Corner (LLC) of a rectangle r_i as two constants x_i and y_i, and its width and height as the constants w_i and h_i. The width and the height of a rectangle are the displacements from its LLC along the X-axis and the Y-axis, respectively.

Now we will provide the model of this problem in SMT-LIB. For each rectangle, we have to define four integer constants. For a rectangle r_1, these constants are declared in SMT-LIB as follows.

```
1 (declare-const x1 Int)
2 (declare-const y1 Int)
3 (declare-const w1 Int)
4 (declare-const h1 Int)
```

Listing 2.11: The declaration of the four constants that describe a small rectangle in the rectangle packing SMT problem.

Using these constants, we can specify the formulas that constrain the orientation and position of the small rectangles in the bigger one (the printing canvas). To simplify this example, we limit the orientation of the small rectangles to either portrait or landscape modes. Assume the dimensions of r_1 are 75×110, the two potential orientations of this rectangle are expressed by the following constraint.

```
1 (assert (or (and (= w1 75) (= h1 110)) (and (= w1 110) (= h1 75))))
```

Listing 2.12: Orientation constraints of a small rectangle in the rectangle packing SMT problem.

Next, we have to restrict the small rectangles to be entirely contained within the borders of the canvas. For that, we need an inequality to restrict the x_i coordinate of the LLC of r_i to be situated to the right of the origin of the canvas ($x_i \geq 0$) and another inequality to restrict the right edge of r_i to be situated to the left of the canvas's right edge ($x_i + w_i \leq$ the width of the canvas). We also need similar rules to constrain the position of the small rectangles to be placed between the bottom and top edges of the canvas. Assuming the width of the canvas is 280 and its height is 190, these inequalities are defined for r_1 as follows.

```
1 (assert (and (>= x1 0) (<= ( + x1 w1) 280)))
2 (assert (and (>= y1 0) (<= ( + y1 h1) 190)))
```

Listing 2.13: Position constraints of a small rectangle in the rectangle packing SMT problem.

The last set of constraints is required to ensure no overlapping occurs between the small rectangles. For instance, to ensure the right edge of r_i is not situated to the right of the left edge of r_j, the x coordinate of the lower right corner of r_i ($x_i + w_i$) must be less the x coordinate of the LLC of r_j (x_j), This constraint is expressed as ($x_i + w_i \leq x_j$). Similarly, we need other inequalities to enforce the clearance between the right edge of r_j and the left edge of r_i, and between the upper and lower edges of these two rectangles. For r_1 and r_2, these constraints are expressed in SMT-LIB as follows.

```
1 (assert (or (<= (+ x1 w1) x2) (<= (+ x2 w2) x1) (<= (+ y1 h1) y2) (<= (+ y2 h2) y1)))
```

Listing 2.14: Overlapping constraints of a small rectangle in the rectangle packing SMT problem.

For a rectangle fitting problem with five small rectangles, we need 20 constant decelerations for the coordinates and dimensions of each rectangle and 25 assertions to express the constraints of this problem. Rather than manually drafting these constraints, one can use a script to produce the required declarations and assertions. After getting the problem modelled, we use the SMT-LIB commands *(check-sat)* and *(get-model)*, which tells the SMT solver to check the satisfiability of the problem and return a model that satisfies these constraints. Such a model is an assignment to the constants that represent the position and orientation of each of the five rectangles.

An SMT solver is a software tool that implements algorithms and techniques to solve SMT problems. These solvers take a logical formula as input, typically written in a constraint modelling language, and attempt to find a satisfying assignment to the formula or prove its satisfiability. SMT solvers have a wide range of applications, including software verification [56], model checking [57], automated test generation [58], and automated planning [59]. Some of the most widely used SMT solvers include Z3 [60], Yice [61], CVC [62], MathSAT [63] and Boolector [64].

In this thesis, we formulate the problem of finding a mapping between the variables of the atoms of the operators of two planning domain models as an SMT problem. This variable mapping must satisfy a set of constraints that are designed to ensure the two planning domain models are functionally equivalent. The tool proposed by this thesis in Chapter 4 constructs the constraints that model this problem and invoke the SMT solver Z3 to find a mapping that satisfies these constraints. We chose to use Z3 in this study. However, any other SMT solver that supports quantifier-free linear integer arithmetic logic can also be used.

This section provided a high-level overview of the SMT problem and provided an example of how SMT problems can be specified. We formulate part of the planning domain model functional equivalence validation problem as an SMT problem in the fourth chapter of this thesis. The next section provides a brief review of the verification of plans, planning tasks and planning domain models.

2.6 Verification and Validation of Planning Systems

The systematic verification and validation of planning systems is increasingly becoming more important. Planning systems could produce wrong plans or improperly execute correct ones. This could be due to design errors in planning tasks or planning domain models. The following sections describe some verification and validation methods of plans, planning tasks and planning domain models.

2.6.1 Plan Validation

As planning systems grow in sophistication and capabilities, plans become complicated, and the need for plan validation becomes immense. Plan validation checks whether a given plan is a correct solution to a given planning problem. To do so, plan validators check if the plan's sequence of actions can transform

the world from the initial state into a goal state as described in the planning problem, and that the plan's actions respect all their preconditions and constraints as described in the planning domain model. Additionally, plan validation techniques should match the targeted planning approach. For example, the plan validator VAL, proposed by Howey et al. [65], is designed to verify classical plans, whereas Behnke et al. [66] presented a hierarchical plan validator.

VAL is a classical plan validator that supports PDDL2.1 features, such as continuous effects, exogenous events and processes, which enable validating plans produced using accurate models of real-world applications. Events and processes (actions with continuous effects) can be used to model the background behaviour of the environment where the planning agent is situated. In addition to plan validation, VAL supports mixed-initiative planning. In this concept, a human planner generates an initial plan and refines it with the help of the plan validation report produced by VAL. This report identifies the reasons behind the plan failure with respect to the given planning domain model and provides suggestions to fix it. These attractive features enable automated plan checking and faulty plan debugging, which are essential parts of planning verification systems.

The validation of hierarchical plans, such as HTN plans, needs to check the validity of primitive task networks, the solutions to HTN planning problems. Then, hierarchical plan validators check if primitive task networks are legal decompositions of initial task networks. The HTN plan validator proposed by Behnke et al. [66] first uses VAL to validate the executability of the plan. Secondly, it expresses the problem of checking that primitive task networks correspond to initial task networks as Boolean satisfiability problems (SAT). Finally, a SAT solver is used to prove the existence of a legal decomposition of the initial task network into the primitive task network.

Plan validators are important tools and have been used in the international planning competitions to validate the soundness of the plans produced by participating planners.

2.6.2 Planning Task Repair

Given a planning domain model and a planning problem, planning task repair methods check if the provided planning problem is solvable using the planning domain model otherwise propose repairs that make the planning problem solvable. A planning task might be unsolvable because its specification is incomplete or incorrect. In such cases, planning task repair systems can suggest ways to change the planning task to make it solvable. These changes include weakening the goal condition, adding more atoms to the initial state, adding new objects, relaxing some preconditions of the actions or even adding new actions. Interestingly, Herzig et al. [67] showed that changes to planning actions could be reduced to initial state updates.

A method that makes a planning task solvable by finding corrections to the initial state and set of objects was proposed by Göbelbecker et al. [68]. For this problem to be useful and not trivial, changes to the initial state that directly contribute to the satisfaction of the planning goal are not permitted. The authors define such a change to the initial state as an “excuse” because this change explains why the planning task was unsolvable. They compile the problem of finding excuses for an unsolvable planning

task as an extended planning problem. This planning problem is extended with change operators that change the initial state. These operators are constrained to be scheduled before the actual planning actions. Thus, the planner can use them to change the initial state of the unsolvable planning task to a new initial state from which the original actions can solve the problem. To find a minimal set of corrections, the change operators are associated with costs specified to discourage plans that rely on change operators rather than the actual planning operators to get closer to a goal state.

On the other hand, Gragera et al. [69] proposed an approach to repair planning domain models where a planning task is unsolvable because the effects of some actions are incomplete. They have extended the unsolvable planning task with special operators to repair the incomplete domain actions. The actions in the extended planning problem are the original, fix, add-fix, and close actions. Fix actions select an atom, and assign it as a new effect to any action. Add-fix actions instantiate the atom's variables with the objects used in the action under repair. Finally, the close action denotes the conclusion of repapering the current action. These additional actions can add any atom as an add effect to any action. To prevent the solution to the reparation planning problem from adding all predicates as add effects to all actions, the authors classified the repairing actions into four categories based on the type of the added effect and associated different costs with these sets of repairing actions. Invoking a planner with the metric of minimising the reparation cost causes the planner to find a set of corrections with a minimal number of repairing actions and from the least penalised type of corrections.

2.6.3 Planning Domain Model Validation

Planning domain model validation focuses on evaluating the accuracy, consistency, and completeness of planning domain models [28]. It aims to assess whether the model accurately represents the problem domain and meets the requirements of the planning system. The validation process typically involves checking various aspects of the model, such as the definition of states, actions, preconditions, and effects, to ensure their consistency and accuracy. It may also involve verifying that the model captures all relevant aspects of the planning problem.

Long et al. [70] analysed simple planning models and showed examples of domain model problems. For example, examining the Settlers domain from IPC3 revealed a logical error that was not discovered earlier. Settlers is a strategic war game of infrastructure development and equipment production. By rational evaluation of the model, the authors found that the action of building a ship which produces a wharf and a ship is missing a location constraint to restrict the shipbuilding, including the wharf construction to coastal areas only.

Consider the completeness attribute of planning domain models as defined in [28]. A planning domain model is complete, with respect to its specification, if any valid plan according to the specification is also valid plan according to the planning domain model. Given a plan that is supposed to be valid as per a domain's specification, but it cannot be generated using the developed planning domain model, then this plan represents an indication of a flawed planning domain model. Lin et al. [71] proposed a method to find a minimal set of repairs that corrects the flawed planning domain model. These repairs

are limited to removing actions' preconditions and delete effects, and adding actions' add effects.

A way to enhance the accuracy and consistency of planning domain models is to learn domain models from natural language specifications [72–75] or by observing how domain experts interact with the world's objects [23, 76–80]. Comprehensive reviews of learning methods that learn by observing traces of actions are provided in [81, 82].

Planning actions learning methods are classified based on many characteristics; targeted planning domain modelling languages, inputs to the system, and learning assumptions. Learning methods start from partial planning domain model description, or they learn planning models entirely from scratch. Some of these methods require ideal action traces, whereas others can tolerate noisy execution traces. Action learning algorithms either assume full state observability, i.e. states are fully observable after executing actions or partial state observability, where only some parts of the states are observable. Furthermore, some methods do not require any kind of state observability.

In addition to the efforts of developing planning domain model learning methods, researchers develop knowledge engineering tools to assist modellers in designing planning domain models. In addition to making the design of planning domain models easier, these tools also help designers to build better planning domain models by subjecting the modelling process to systematic requirement elicitation and analysing approaches. A good example of such tools is GIPO [83], a Graphical Interface for Planning with Objects. GIPO follows an object-centric approach in formulation planning domain models. GIPO offers a visual interface that enables users to define world's objects along with their relations and states. Furthermore, using GIPO's invariant editor, users can explicitly define any assumptions about the domain's dynamics. Moreover, to model domain operators, designers can use the transition editor to define operators' preconditions and effects. GIPO also employs Opmaker [79], a planning operator learning tool, that interacts with users to learn operator schema from provided planning tasks. The structured development method applied by GIPO and the constraints imposed by its editors, along with the invariant checks, reduce the number of errors in planning domain models.

The platform itSIMPLE [84] (Integrated Tools Software Interface for Modelling PLanning Environments) enables knowledge engineers to use Unified Modelling Language (UML) [85] to model planning domain models. In the requirements elicitation and modelling phase, object types and their relationships, in addition to operators and their parameters, are expressed using class diagrams. Moreover, the state of objects and how operators affect these states are described with state machine diagrams. The operators' preconditions and actions are defined using the Object Constraint Language from UML [86]. Planning problem instances are modelled using object diagrams; when creating these diagrams, itSIMPLE verifies if objects' relations are consistent with class diagrams. Additionally, itSIMPLE enables users to simulate and analyse the developed model by translating it into a Petri Net graph [87].

These knowledge acquisition and modelling tools for planning are further analysed in [88, 89]. Vaquero et al. [88] reviewed how itSIMPLE and GIPO approach the design phases of modelling planning domains. On the other hand, Shah et al. [89] analysed the usability of these tools with respect to collaborative modelling, model maintenance, required experience, efficiency in producing models, model

debugging, and user support. Other related research areas such as model reconciliation, maintenance, and recognition are reviewed in Section 4.2.

2.6.4 Planning Domain Model Verification

Complex domains have a large number of constraints and are prone to modelling errors. Errors in planning domain models directly affect the validity of plans. Flawed planning domain models can cause planners to fail in generating plans or produce incorrect ones. To address this, verification techniques verify planning domain models to confirm if plans generated using these models satisfy specified properties. Planning domain model verification is performed using either model checking or test-based methods. These techniques are explained in the following sections.

2.6.4.1 Test-based Verification of Planning Domain Models

Test-based verification methods of planning domain models rely on tasking a sound planner to produce plans for a large number of planning problems that resemble a variety of initial states and planning goals. Then, these verification methods check if produced plans meet some expectations.

Raimondi et al. [10] proposed a test-based verification method for planning domain models and implemented this method in a tool called PDVer. This tool generates planning goals that stress given requirements. These goals represent positive and negative test cases. The former mean a sound and complete planner should find a valid solution to the given planning goal. On the other hand, negative test cases require the planner to fail to find a plan. Requirements are first expressed in the form of LTL formulas. Then, positive and test cases are generated and translated into state trajectory constraints in PDDL. After that, the MISP-XXL planner is used to search for plans that respect the specified constraints. The results of the planning are judged based on the type of the test cases. If the planning domain model allows the production of plans for all positive test cases and not for any negative test case, then the planning domain model is considered correct with respect to the requirements.

Goldberg et al. [11] propose a runtime verification method to verify planning domain models. The test inputs for this method compromise planning goals and set of properties that are required to hold in any produced plan using the planning domain model under testing. The used planner is instrumented to emit the values of all variables in every state during planning to a log. The properties are translated into temporal logic formulas in Eagle (rule-based framework for runtime verification) environment [90]. Eagle then produces oracles that can automatically monitor and check whether the state of the produced plans satisfies the given properties. The satisfaction of these properties increases the confidence in the correctness of the planning domain model with respect to the provided properties.

2.6.4.2 Model Checking Verification of Planning Domain Models

Using a model checker to verify a planning domain model with regard to given requirements requires the planning domain model to be translated into the modelling language of the model checker. After

that, the requirements are expressed in terms of formal properties. For example, safety properties could be that plans should never enter a forbidden state. The model checker will either produce a trace of actions that leads to that forbidden state in the form of a counterexample or fail to find any trace of actions that can reach that state. In the latter case, the model is confirmed to have no provision for producing faulty plans with regard to the given property.

Smith et al. [12] proposed an approach to verify that planning domain models do not permit planners to produce undesired plans. The planning domain model is translated into Promela, the modelling language of the Spin model checker, and then the requirement "undesired plans could never be produced from the planning domain model under verification" is expressed as a formal safety property. Then, Spin searches the model's state space and produces a counterexample to falsify the safety property if one exists. The counterexample is an undesired plan which is permitted by the planning model and that arose due to possibly missing constraints in that model. Counterexamples are then used to debug the planning domain model and add the required constraints as preconditions.

Cesta et al. [13] used model checkers (NuSMV and UPPAAL) to verify the planning domain model of the timeline-based MrSPOCK planner. First, the planning domain model is automatically translated into the model checkers' modelling languages. Then the model is verified against properties derived from the requirements. Whenever a property does not hold, the model checker produces a counterexample which can be used to identify the planning domain model inconsistency.

Penix et al. [91] used model checking to verify a planning domain model for the HSTS planner (Heuristic Scheduling Tested System) [92]. They translated a model of an autonomous robot from the HSTS Domain Description Language (DDL) into SMV, Spin and Murphi model checkers' input languages. Then, they compared the performance of the examined model checkers in verifying the given model.

Khatib et al. [93] proposed to use the model checker UPPAAL to verify that an HSTS planning model can produce valid plans. They presented an algorithm called ddl2uppaal to translate HSTS planning domain models into UPPAAL modelling language. A given planning goal is then expressed as an UPPAAL property. Then UPPAAL is used to confirm if the given planning goal is reachable given the HSTS planning domain model.

Havelund et al. [94] used the model checker Spin to verify that an AML model of a drilling rover can produce sound plans. They expressed planning goals as LTL properties and used their tool to convert models from AML to Promela. Using Spin's message sequence diagram, they found that the model under verification enables producing a sequence of actions that leads to transmitting analysis data before collecting samples. To prevent this unwanted behaviour, they constrained the data transmitting activity with the condition that requires the collection of samples to be scheduled before the data transmitting activity. Chapter 3 argues that the verification task described in this study is under-constrained because planners employed in real planning tasks, where the goal is to transmit analysis data of some rock sample, do not schedule the transmission activity before sampling the targeted rock.

Section 3.9 contrasts the approaches of these methods in applying model checking to verify planning domain models with our goal-constrained verification method presented in Chapter 3.

2.7 Summary

This chapter presented essential background knowledge on the verification and validation of planning domain models. It covered key topics such as automated planning, classical planning problem representation, Knowledge Engineering in Planning and Scheduling (KEPS), and planning domain definition language PDDL. This chapter also introduced the methods employed in thesis, including planning techniques, model checking, and SMT solving. Along with introducing these methods, this chapter provided illustrative examples demonstrating the formulation of planning problems in PDDL, the application of model checkers for verifying finite state systems, and the expression of constraint problems as SMT problems. Additionally, this chapter offered a brief review of the verification and validation of plans, planning tasks and planning domain models. Subsequent chapters will provide more detailed reviews of the relevant literature in the verification and validation of planning domain models.

CHAPTER 3

GOAL-CONSTRAINED PLANNING DOMAIN MODEL VERIFICATION OF SAFETY PROPERTIES

3.1 Introduction

The verification of planning domain models is crucial to ensure the safety, integrity and correctness of planning-based automated systems.

Planning domain model verification aims to verify that any plan produced by a sound planner using the planning domain model satisfies a given property. This aim is a very ambitious objective because such a verification task does not constrain the verification to a specific set of planning problems. Rather than attempting this unconstrained verification task, the state-of-the-art planning domain model verification methods aims to verify any plan produced, with regard to an initial state and a set of objects, by a sound planner using the planning domain model under verification satisfies a given property. Thus, the current verification approaches perform the verification of planning domain models with regard to an initial state and a set of objects. However, such verification methods of planning domain models are still under-constrained because they do not specify the planning goal in the verification problem.

This chapter addresses the fact that under-constrained verification methods for planning models are vulnerable to false positive counterexamples. In particular, these methods might return counterexamples that are unreachable by planners when using the planning domain model under verification (DUV for short) during a planning task. Such counterexamples can mislead designers to unnecessarily restrict domain models, thereby potentially blocking valid and possibly necessary behaviours.

To address this shortfall, we propose to use planning goals as constraints during verification, a concept transferred from verification and validation research [16]. Thus, we introduce *goal-constrained planning domain model verification*, a novel approach that reduces the number of invalid planning counterexamples. This planning domain model verification approach is independent form the modelling

language used to model planning domain models. We formally prove that goal-constrained planning domain model verification of safety properties is guaranteed to return only planning counterexamples that falsify safety properties before achieving planning goals, if and only if any exists. Additionally, we perform empirical experiments to demonstrate the feasibility and investigate the behaviour of our approach using the Spin model checker [17] and the MIPS-XXL planner [18].

The rest of this chapter is organised as follows. Section 3.2 informally discusses the problem of invalid planning counterexamples in the verification of planning domain models. The goal-constrained planning domain model verification concept is presented in Section 3.3. To facilitate explaining our approach, Section 3.4 introduces a running example. The description of how planning goals should be used to constrain planning domain model verification tasks when employing model checking is provided in Section 3.5, and using planning techniques in Section 3.6. Next, Section 3.7 reports and discusses the results of the empirical experiments conducted to analyse the feasibility and the behaviour of our approach. Section 3.8 suggests a good practice for modelling planning domains that are inherently safe. After that, Section 3.9 contrasts the goal-constrained verification method presented in this chapter with existing planning domain model verification methods. Finally, Section 3.10 summarises the research presented in this chapter.

3.2 Invalid Counterexamples in Planning Domain Model Verification

Planning domain model verification aims to demonstrate that produced plans for any planning goal satisfy a set of properties. To achieve this, formal planning domain model verification methods leave the planning goal open. We call such methods *under-constrained* verification of planning domain models, i.e. the verification is expected to hold for any potential goal. Under-constrained verification task searches the state space produced using the planning domain model and the set of objects for a counterexample, a sequence of actions that can falsify the given property, regardless of any other conditions. In particular, whether or not a planner would consider this sequence to be a plan, is not taken into account. This is a critical shortfall, because, when the domain model is used to solve a specific planning problem, the sequence of actions that constitutes such a counterexample might, in fact, be "pruned away" by the planner, if it does not satisfy the planning goal. Hence, we consider them to be invalid planning counterexamples.

To illustrate the concept of invalid planning counterexamples, we use a modified version of the microwave oven example, introduced in [14]. The Finite State Machine (FSM) of this example is illustrated in Figure 3.1.

The FSM consists of four states, described with four propositions. The Close proposition represents that the oven's door is closed, while Start indicates that the start push button is pressed. The Error proposition denotes the oven is in an erroneous state, and Heat means that the oven is heating. The available actions for this FSM are Close Door and Start Oven.

The goal of a planner reasoning about the planning problem represented by this FSM is to find a plan to change the state of the oven from the initial state to a state where the heating is on, i.e. s_3. Moreover,

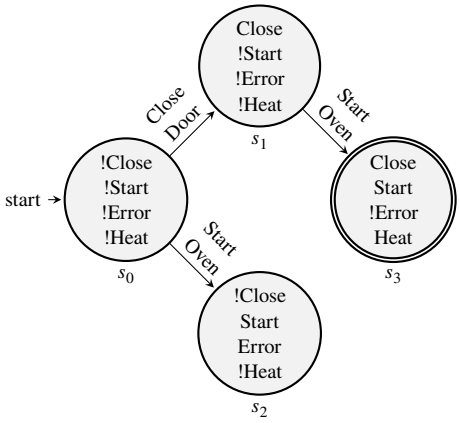


Figure 3.1: Microwave oven FSM.

this FSM models the requirement that if the oven is started with the door open, then the oven must raise an error. This requirement can be captured by a safety property. This property mandates that any plan produced from this FSM does not cause the oven to reach a state where the proposition Error is true. In LTL, this safety property is expressed as $p_0 = G(\neg Error)$, where G is the LTL operator *always*.

Under-constrained verification of this FSM and the safety property p_0 will return ⟨Start Oven⟩ as a counterexample to this property because when this sequence is applied to s_0 will produce s_2, which is an erroneous state. However, when this model is used to find a plan that achieves the goal ($g = Heat$), this sequence will not be considered by the planner as it does not lead to a state that achieves the goal. Furthermore, this sequence is also not a part of a valid plan that can achieve the goal. Therefore, this model does not have any valid planning counterexample.

3.2.1 Invalid Planning Counterexamples in the Literature

Counterexamples that are unreachable by a planner exist in the literature. For example, Smith et al. [12] used the Spin model checker to verify whether a planning domain model would permit an automated planning system to produce plans that would waste resources and, therefore, not meet the mission's science goals. To express this requirement, they used *"five data-producing activities must be completed by any valid schedule"* as a property for model checking. The automated system has two data-producing activities, take a picture and collect a soil sample, two data-consuming activities, data compress and data uplink, and a buffer that can hold four data blocks. The goal of the planner is to schedule five data-producing activities, taking three samples and two images.

The counterexample returned by the model checker is a sequence of actions that have two data-consuming activities followed by four data-producing activities. This sequence did not contain a fifth data-producing activity because the data buffer was full after the four data-producing activities. Note that the data-consuming activities that would have cleared the buffer were already scheduled at the beginning of the plan when the buffer was empty. Though the model checker found a counterexample to falsify the property, we argue that any sound planner would not generate such a plan, because it does not

achieve the planning goal. As such, this counterexample would have been pruned during the planner's goal search, and consequently, it would never have been returned as a plan, i.e. it is unreachable for the planner, yet reachable by a goal-ignorant model checker.

The problem with unreachable counterexamples is that they mislead the designer to unnecessarily restrict the domain model in the process of removing them. Consequently, debugging is made harder and genuine counterexamples could potentially be introduced in the process.

To overcome this downside of using goal-ignorant model checkers to verify planning domain models, we observe that planning is performed for a specific goal. To exploit this observation for planning domain model verification, we propose to use the goal given to the planner as a constraint to ensure that the counterexamples returned by a model checker, or other tools used in this context, falsify the given safety property before also achieving the planning goal. Thus, instead of performing under-constrained planning domain model verification, we introduce goal-constrained verification of planning domain models. The details of this novel method are further explained in the next section.

3.3 Goal-constrained Verification of Planning Domain Models

Goal-constrained verification can be performed using advanced search algorithms, such as model checkers or classical planners, to find a valid counterexample for a given safety property, if one exists.

We define a valid planning counterexample to be a sequence of actions that can falsify the given safety property before it can achieve the planning goal from the given initial state.

As introduced in Section 2.1.1 in Chapter 2, the plan π, defined as a sequence of actions, $\pi = \langle a_0, a_1, ..., a_n \rangle$, is a solution to the planning problem $\mathcal{P}$. When π is applied to the initial state s_0, it yields a sequence of states $S(\pi)$, $S(\pi) = \langle s_0, s_1, ..., s_n \rangle$ where only the last state s_n satisfies the planning goal g, $s_n \vDash g$. Furthermore, we say a plan π satisfies a property p, $\pi \vDash p$, if the sequence of states $S(\pi)$, generated by the plan π, satisfies the property p, $S(\pi) \vDash p$.

Definition 3.1. A **valid planning counterexample** for a safety property, p, of a planning problem $\mathcal{P} = (D, Obj, s_0, g)$, is a plan, π, that falsifies the safety property, $\pi \nvDash p$.

Note that this definition excludes counterexamples that do not achieve planning goals and those that falsify the safety property after achieving the goal. The latter condition is implied because valid counterexamples are defined as plans, and plans are defined to satisfy the planning goal at the last state of the execution. Note that a planning goal might be a conjunction of multiple conditions, and some of these conditions might be satisfied during the execution. However the conjunction itself must not be satisfied before the execution reaches the last state.

This definition of valid counterexamples does not exclude plans that are enriched with action sequences which are unnecessary to achieve the planning goal but required to falsify the given safety property. It can be argued that such plans also represent invalid planning counterexamples as a planner tasked with solving the planning problem $\mathcal{P}$ might never return such plans because they are modified to falsify the safety property. Determining if planning counterexamples are augmented with additional

actions solely for the purpose of falsifying given properties is left for future work. Therefore, the verification approach presented in this chapter might still return counterexamples that can be seen as invalid planning counterexamples. As such, we claim to only reduce the number of invalid planning counterexamples during planning domain model verification. Hereafter, a counterexample is considered a valid planning counterexample if it meets the condition specified in Definition 3.1.

3.4 Running Example

In the following sections, we explain how goal-constrained verification of planning domain models can be realised using model checking and planning techniques. To further clarify these methods, we use the planning problem presented in Listing 3.1 and Listing 3.2 as a running example. This planning problem extends the FSM of the microwave oven example described in Figure 3.1.

```
(define (domain microwave_domain)
  (:requirements :typing :adl)
  (:types microwave)
  (:predicates
    (start ?m - microwave) (close ?m - microwave)
    (error ?m - microwave) (heat ?m - microwave))
  (:action start-oven
    :parameters (?m - microwave )
    :effect (and
      (when  (and (not (close ?m)))
                  (and (error ?m) (start ?m)))
             (when  (and (close ?m)  )
                  (and (heat ?m) (start ?m)))))
  (:action close-door
    :parameters (?m - microwave )
    :effect (and (when  (start ?m)
                  (and (heat ?m) (close ?m) (not (error ?m))))
             (when  (not (start ?m)) (close ?m))))
  (:action open-door
    :parameters (?m - microwave )
    :effect (and (when  (and (heat ?m) (close ?m))
                  (and (error ?m) (not (close ?m))
                  (not (start ?m)) (not (heat ?m))))
             (when  (and (not (heat ?m)) (close ?m))
                  (and (not (close ?m)) (not (start ?m)))))))
```

Listing 3.1: Microwave oven planning domain model.

An FSM that captures some of the interesting transitions that can be produced from this planning problem is depicted in Figure 3.2. This extended FSM models the possibility to recover from the erroneous state s_2 by closing the door. So in this FSM, applying the action Close Door to state s_2 resets the error and turns on the heating, i.e. transfers the FSM to state s_3. Another refinement is modelling the possibility of

opening the microwave door while the heating is on in state s_3. Applying the action Open Door to state s_3 causes the FSM to transit to state s_4, where the heating is turned off and an error is raised.

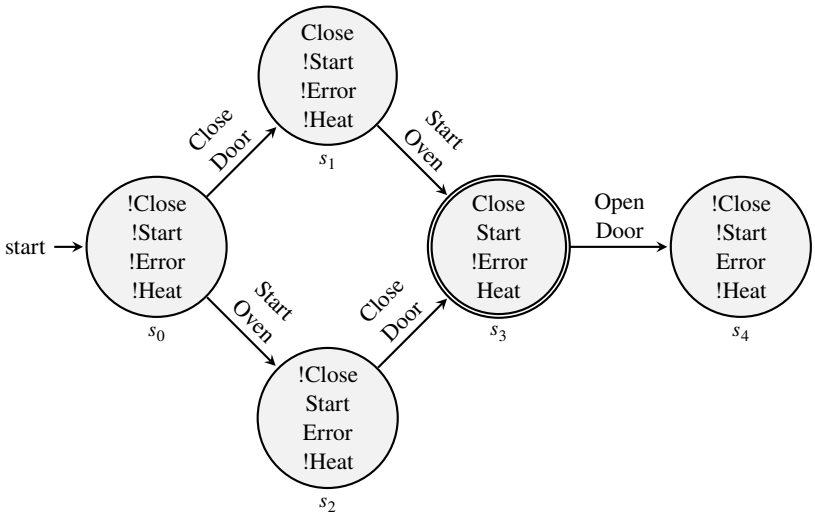


Figure 3.2: Microwave oven extended FSM

The verification problem of the planning domain model of the microwave oven presented in this example is expressed as $V = (\mathcal{P}, p)$, where $\mathcal{P} = (D, Obj, s_0, g)$. In this verification problem:

- D is the planning domain model described in Listing 3.1.
- $Obj = \{m\}$, where m is of type microwave.
- $s_0 =$ {(not (close m)),(not (start m)),(not (error m))(not (heat m))}
- $g =$ (heat)
- $p = G(\neg Error)$

Obj, s_0, and g are specified in Listing 3.2.

```
1 (define (problem microwave_problem)
2   (:domain microwave_domain)
3   (:objects m - microwave)
4   (:init   )
5   (:goal (and (heat m))))
```

Listing 3.2: Microwave oven planning problem.

3.5 Goal-constrained Verification Using Model Checkers

Model checkers verify safety properties by searching for counterexamples that falsify those properties. In the case of planning applications, any sequence of actions that does not achieve the given goal, will

not be considered as a plan by any sound planner. Therefore, in planning domain model verification, any counterexample that does not achieve the given planning goal should be eliminated on the basis that this counterexample is unreachable by the planner.

To force model checkers to only return valid planning counterexamples, the safety property is first negated and then joined with the planning goal in a conjunction. This conjunction is then negated and supplied to the model checker as an input property. The final property requires the model checker to find a counterexample that both, falsifies the safety property and satisfies the planning goal. Note that this modified property permits sequences that falsify the property after satisfying the goal. To eliminate such sequences, model transitions should be augmented with an additional guard, representing the negation of the goal, to restrict all transitions once the goal is achieved. With this modification, the model checker is forced to return counterexamples that falsify the safety property before achieving the goal.

For a verification problem, $V = (\mathcal{P}, p)$, we first check whether the planning goal is achievable. Then, we translate the state-transition system, Σ, that is produced from the planning domain model D, the set of objects Obj, and the initial state s_0 into the model checker's input language, obtaining the model M. Then, the guards of the transitions of the model M are augmented with the negation of the goal condition to produce the modified model M'. This model satisfies the following two properties.

Proposition 3.1. *All plans generated from M are also plans that can be generated from M'.*

Proof. Any sequence of transitions from M that ends with a transition that achieves the goal is also a sequence of transitions from M'. These sequences represent valid plans as they terminate with a transition that achieves the goal. Therefore, all plans generated from M are also plans in M'. ■

Proposition 3.2. *Any valid counterexample for M' is also a valid counterexample for M*

Proof. As M' is a more constrained version of M, the set of all legal transition of M', $\Pi(M')$, is contained in the set of all legal transitions of M, $\Pi(M)$, i.e. $\Pi(M) \supseteq \Pi(M')$. It follows that any valid counterexample in $\Pi(M')$ is also in $\Pi(M)$. ■

Using the modified model M', a model checker is then can be applied to the verification problem $V'' = (M', p')$, where p' is defined using the LTL operator *eventually*, F, as follows:

$$p' = \neg\big(F(\neg p) \wedge F(g)\big) \tag{3.1}$$

There are two possible outcomes of applying a model checker to the verification problem V''. First, if the model checker returns a counterexample π:

$$\exists \pi \in \Pi(M').\pi \nvDash p' \equiv \exists \pi \in \Pi(M').\pi \vDash (F(\neg p) \wedge F(g)) \tag{3.2}$$

From the definition of the operator *eventually*:

$$\exists i \geq 0,\ s_i \in S(\pi), s_i \vDash \neg p \tag{3.3}$$

$$\exists j \geq 0,\ s_j \in S(\pi), s_j \vDash g \tag{3.4}$$

It follows that there is at least one sequence $S(\pi)$ that falsifies the property p, and there is a state s_j in that sequence which satisfies the goal g, according to (3.3) and (3.4). In addition to that, in the sequence $S(\pi)$, p is guaranteed to be falsified before g is satisfied. This is because $\pi \in \Pi(M')$ and M' is constrained to not produce any transitions after achieving the goal. Thus, the counterexample π is a valid planning counterexample in M' for the original safety property p as per Definition 3.1. Furthermore, according to Proposition 3.2, π is also a counterexample in M. This proves that the DUV does not satisfy the safety property p with respect to the goal g.

The other potential outcome is that the model checker fails to find a counterexample, then $\forall \pi \in \Pi(M')$:

$$\pi \vDash p' \equiv \pi \vDash \neg(F(g) \wedge F(\neg p)) \equiv \pi \vDash (\neg F(g) \vee \neg F(\neg p)) \tag{3.5}$$

$$\equiv \pi \vDash \neg F(g) \vee \pi \vDash \neg F(\neg p) \equiv \pi \nvDash F(g) \vee \pi \vDash G(p) \tag{3.6}$$

$$\equiv \pi \vDash F(g) \Rightarrow \pi \vDash G(p) \tag{3.7}$$

where G is the LTL operator *always*. This means p is always true for any sequence of actions in M' that achieves the goal, i.e. for all possible plans. Therefore, we can deduce all plans in M' are safe. Furthermore since Proposition 3.1 states all plans generated from M are also plans in M', we can conclude that all plans in M are safe. This proves that the DUV satisfies the original property with respect to the goal g.

3.5.1 Example

Consider the example presented in Section 3.4 which assumes the initial state is s_0, the goal ($g = Heat$) and the safety property $p_0 = G(\neg Error)$. A goal-constrained verification task would be:

$$V = \Big(\mathcal{P} = (D, Obj, s_0, g = Heat), p_0 = G(\neg Error)\Big)$$

First, we find the negation of the safety property:

$$\begin{aligned} & q = \neg p \\ \equiv\ & q = \neg\big(G(\neg Error)\big) \\ \equiv\ & q = F(Error) \end{aligned}$$

Using Equation (3.1), the model checker input property is formulated as:

$$\begin{aligned} & p' = \neg\Big(F\big(F(Error)\big) \wedge F(Heat)\Big) \\ \equiv\ & p' = \neg\big(F(Error) \wedge F(Heat)\big) \end{aligned}$$

The modified model M' is produced from the original FSM, as explained in Section 3.5, by augmenting the guards of all transitions with the negation of the goal condition. This modified FSM is shown in Figure 3.3. The Promela model of the modified model M' is presented in Listing 3.3. Line number 1 in this listing represents the model input property in LTL. The initial state is specified in line number 2.

The transitions of the operators Start Oven, Close Door, and Open Door are described in lines 5 to 16. Note that the Open Door action has two transitions. The one described in line 14 has a contradiction in its guard. This contradiction is caused because we added the negation of the planning goal ($\neg\, Heat$) to the guard of each transition in the model, and the guard of the transition in line 14 has ($Heat$) as one of its conditions. Thus, this transition will never be executed. With this modification, the model checker cannot falsify the safety property after achieving the planning goal. The second transition of the Open Door action is described in line 15. The guard of this transition has the condition ($\neg\, Heat$) repeated twice. This is not a typo; this transition has this condition originally in its guard, and it was added again because ($\neg\, Heat$) is the negation of the planning goal. The guard of this transition was not simplified to emphasise that ($\neg\, Heat$) is an original condition of the action Open Door. Note that the FSM depicted in Figure 3.3 shows only the interesting transitions from the model described in Listing 3.3.

```
1  ltl p0 {!(<> Error && <> Heat)}
2  bool  Close = 0,Start = 0,Error = 0,Heat = 0;
3  proctype Microwave() {
4   do
5   //StartOven when the door is not closed
6   :: atomic {((Heat == 0) && (Close == 0))-> Error = 1; Start = 1}
7   //StartOven when the door is closed
8   :: atomic {((Heat == 0) && (Close == 1))-> Heat = 1; Start = 1}
9   //CloseDoor when the start button is pressed
10  :: atomic {((Heat == 0) && (Start == 1))-> Heat = 1; Close = 1; Error = 0}
11  //CloseDoor when the start button is not pressed
12  :: atomic {((Heat == 0) && (Start == 0))-> Close = 1}
13  //OpenDoor when the door is closed and the heat is on
14  :: atomic {((Heat == 0) && (Heat == 1) && (Close == 1))->  Error = 1; Close = 0;
        ↪ Start = 0; Heat = 0}
15  //OpenDoor when the door is closed and the heat is not on
16  :: atomic {((Heat == 0) && (Heat == 0) && (Close == 1))->  Close = 0; Start = 0}
17  od;
18 }
19 init {
20  atomic{run Microwave()}}
```

Listing 3.3: The promela model of the modified microwave oven FSM M'.

Applying a model checker to the verification problem $V'' = (M', p')$ produces a valid counterexample ⟨Start Oven, Close Door⟩ which satisfies the goal and falsifies the safety property. This indicates that the model is not safe with respect to the given goal and safety property. Hence, the domain model must be fixed to eliminate this valid counterexample. This is accomplished by adding the condition that the door must be closed to the preconditions of the action Start Oven. To add this condition to the Promela model, line number 6 must be deleted in Listing 3.3. With this additional condition, the transition between s_0 and s_2 is blocked. Thus, state s_2 becomes unreachable. Consequently, the valid counterexample ⟨Start Oven, Close Door⟩ is removed from the model.

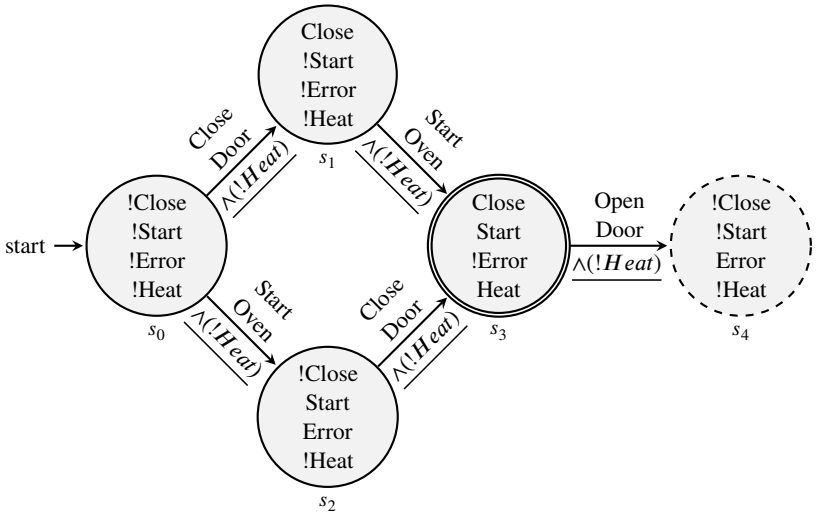


Figure 3.3: Microwave oven extended FSM with the proposed model modification (the conjunction of the negation of the planning goal)

Another sequence worth discussing is ⟨Close Door, Start Oven, Open Door⟩. This sequence achieves the goal at s_3 and falsifies the safety property at s_4. However, this sequence does not meet the order requirement in Definition 3.1; therefore, it is not a valid counterexample. Intuitively, this sequence is not a valid planning counterexample because any sound planner would terminate once the goal is achieved at s_3 and would not schedule any actions beyond that state. Unlike the traditional application of model checkers in planning domain model verification, our method does not return such an unreachable sequence. On the first state, where the goal condition is satisfied, all transitions are blocked thanks to the modification introduced into the model, where the guards of all transitions are augmented with the negation of the goal condition. Hence, the transition from s_3 to s_4 would not be allowed in the modified FSM, and subsequently, the sequence ⟨Close Door, Start Oven, Open Door⟩ is not returned by the goal-constrained model checkers as a valid counterexample. With no further counterexamples returned by the model checkers, the planning domain model represented by this FSM is deemed safe with respect to the given goal and safety property.

3.6 Goal-constrained Verification Using Planners

Planning domain models can be verified to only permits safe plans, in terms of satisfying a given safety property, for a specific goal using planners. This is achieved by consulting a planner over the DUV to produce a plan that can satisfy the goal and the negation of the property. If the planning domain model allows producing plans that, along with achieving the goal, contradict the safety property, then an unsafe plan can be found. Thus, the returned plan is a counterexample. This section describes how planning with state trajectory constraints can be used to perform goal-constrained verification of planning domain models.

State trajectory constraints, introduced in PDDL3, are defined as assertions that must be satisfied by the sequence of states produced by the execution of a plan. They are expressed through temporal modal operators [95]. Among others, these operators include *always, sometime,* and *sometime-before*.

The operators *always and sometime* are similar to the LTL operators *always* and *eventually*. The operator *(sometime-before A B)* means if A is ever true during the execution of a plan, then B must also have been true before that point. These temporal modal operators can be used to express soft and strong constraints. The former are preferences that describe good plans. They are called soft because a plan is still considered correct even if it does not satisfy them. On the other hand, strong constraints are conditions that must be met by a plan to be valid.

Planners that support the strong constraints feature of PDDL3 can be used to perform planning domain model verification of safety properties. To employ such a planner to find a valid planning counterexample for a safety property, we have to provide the planner with a strong constraint that requires the violation of the safety property to happen before achieving the goal. This constraint is specified as *(sometime-before (goal) (negation of the safety property))*. This constraint enforces the order requirement of falsifying the safety property and aching the planning goal as per Definition 3.1.

For a verification problem, $\mathcal{V} = (\mathcal{P} = (D, Obj, s_0, g), p)$, the safety property, p, is negated and joined with the goal as the strong constraint:

$$c = (\textit{sometime-before}\ (g)\ (\neg p)) \tag{3.8}$$

Safety properties states nothing bad can happen. Thus, they are expressed as $\textit{always}(\neg \mathit{Error})$, where Error refers to any safety violation. When such a safety property is negated, during verification, it becomes $\textit{sometime}(\mathit{Error})$. PDDL3 does not allow nested temporal operators; therefore, we cannot just substitute $\neg p$ with $\textit{sometime}(\mathit{Error})$ because this will produce the illegal constraint:

$$c = (\textit{sometime-before}\ (g)\ (\textit{sometime}(\mathit{Error})) \tag{3.9}$$

This restriction limits the level of complexity in the description of safety properties. However, we note that we do not need to nest modal operators to express the strong constraint of our valid planning counterexample if the safety violation Error can be expressed without using modal operators. In the case of Equation (3.9), we can simply drop the operator *sometime* without compromising the behaviour that needs to be captured by the constraint. Thus, this new constraint becomes

$$c = (\textit{sometime-before}\ (g)\ (\mathit{Error})) \tag{3.10}$$

This constraint is only satisfied if the falsification of the safety property happens before achieving the original planning goal.

This constraint is then added to the verification problem $\mathcal{V}$ to produce a verification-as-planning problem. Using the algorithm proposed by Edelkamp et al. [18], this verification-as-planning problem is transformed into a PDDL2 planning problem. This is performed by first translating the state trajectory constraint into a non-deterministic finite state automaton (NFA). This NFA, which can capture property

violations, is then embodied in the original planning task. This automaton is encoded by special predicates that represent the states of the automaton. In addition, the transitions of this automaton are described by special operators. Moreover, the synchronisation of this automaton with the exploration of the planning state space is achieved by using synchronisation interlocks, which ensure the planner advances the simulation of the automaton after each time a planning operator is scheduled. These special predicates and operators are added to the planning domain model. Furthermore, a conjunction of a predicate that represents the accepting state of the automaton and the original planning goal is used as the new planning goal of the modified planning problem. These additions observe the behaviour of the automaton that represents the constraint and stop goal satisfaction unless the constraint is satisfied too. The compilation of the state trajectory constraints to a PDDL2 planning task is implemented as part of the MIPS-XXL planner developed by Edelkamp et al. [96].

This compilation process yields a new planning problem, $\mathcal{P}' = (D', Obj, s'_0, g')$, where D', s'_0, g' are modified instances of D, s_0, g that are supplemented with the automaton-related predicates and operators. Note that $\mathcal{P}'$ is the verification-as-planning problem for the planning problem $\mathcal{P}$ and the safety property p because the solutions to $\mathcal{P}'$ are counterexamples to the property p when verified in the context of $\mathcal{P}$. Let the set of legal sequences of actions that can be generated from D be $\Pi(D)$ and from D' be $\Pi(D')$. Note that D' is an augmented version of D, and the modifications in D' do not affect the number of or the causal-relations between the original operators. Hence, $\Pi(D) = \Pi(D')$.

After that, a planner is applied to $\mathcal{P}'$ with two possible outcomes. First, if the planner finds a plan π then: $\exists \pi \in \Pi(D').\ \pi \vDash g'$. Since the satisfaction of g' implies both, the satisfaction of the original goal g at the last state of the sequence $S(\pi)$, and the satisfaction of the state trajectory constraint (the negation of the safety property) by the sequence $S(\pi)$: $\exists \pi \in \Pi(D').\ (\pi \vDash g \wedge \pi \vDash \neg p)$. Furthermore, since $\Pi(D) = \Pi(D')$:

$$\exists \pi \in \Pi(D).\ (\pi \vDash g \wedge \pi \vDash \neg p) \tag{3.11}$$

Furthermore, from (3.11) it follows that $\pi \nvDash p$, confirming that there is at least one sequence of actions from D that achieves the goal while not respecting the safety property. Therefore, this sequence is a valid planning counterexample for that property as per Definition 3.1. Hence, the DUV does not satisfy the property with respect to the planning goal. Alternatively, if the planner fails to find a plan, then, as opposed to (3.11), we have:

$$\nexists \pi \in \Pi(D).\ (\pi \vDash g \wedge \pi \vDash \neg p) \equiv \forall \pi \in \Pi(D).\ (\pi \nvDash g \vee \pi \nvDash \neg p) \tag{3.12}$$

$$\equiv \forall \pi \in \Pi(D).\ (\pi \vDash g \Rightarrow \pi \vDash p) \tag{3.13}$$

Hence, any sequence of actions from D that achieves the goal also satisfies the safety property. Therefore, the property holds for the planning domain model with respect to the given goal.

An advantage of goal-constrained planning domain verification is, where a deterministic planner is used to perform the verification task, there is no need to for this planner to be complete, as long as the planner used for the verification is also the planner that will be used during the planning task. This is because any counterexample not found by that planner during verification, will then also not be reached by the same planner during the planning task.

3.6.1 Example

Consider the example presented in Section 3.4, the goal-constrained verification-as-planning task would be:

$$V = \Big(\mathcal{P} = (D, Obj, s_0, g = Heat), p = always(\neg Error)\Big)$$

The safety property, p, is negated and joined with the goal Heat as per Equation (3.10) to produce the strong constraint:

$$c = (sometime\text{-}before\ (Heat)\ (Error))$$

This constraint is inserted in the planning problem specified in Listing 3.2 to produce the constrained planning problem presented in Listing 3.4.

```
1 (define (problem microwave_problem)
2   (:domain microwave_domain)
3   (:objects m - microwave)
4   (:init   )
5   (:goal
6     (and (heat m)))
7   (:constraints (and (sometime-before (heat m) (error m)))))
```

Listing 3.4: Microwave planning problem with the safety property constraint.

Applying a PDDL3-enabled planner, like MIPS-XXL, to the verification-as-planning problem that consists of the planning domain file presented in Listing 3.1 and the planning problem file shown in Listing 3.4 translates the strong constraint into the FNA depicted in Figure 3.4.

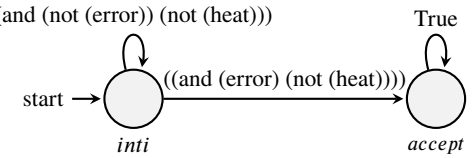


Figure 3.4: The NFA of the strong constraint in the microwave oven verification-as-planning problem.

Note that all transitions of the initial state have *(not (heat))* as a condition. Therefore, if *(heat)*, which is the original planning goal, becomes true while the automaton is still in the initial state, then, the accepting state becomes unreachable. Therefore, the modified planning task, which requires the planning goal to be achieved and the automaton's accepting state to be reached, becomes unsolvable. Moreover, if the proposition Error becomes true in the initial state, the automaton reaches the accepting state and waits for the original planning goal to be achieved. Thus, this automaton enforces the planner to only produce valid planning counterexamples as per Definition 3.1. The transitions of this automaton are listed in Listing 3.5. The *(sync-automaton-a-0)* and *(sync-ordinary)* are the synchronisation predicates. They appear as preconditions and effects of the original and synchronisation operators. All original operators are augmented with these synchronisation predicates.

```
(:action sync-trans-a-0-init-a-0-init
  :parameters()
  :precondition (and (at-a-0-init) (sync-automaton-a-0)
                     (not (error)) (not (heat)))
  :effect       (and (not (sync-automaton-a-0)) (sync-ordinary)))
(:action sync-trans-a-0-init-a-0-accept-0
  :parameters()
  :precondition (and (at-a-0-init) (sync-automaton-a-0) (error) (not (heat)))
  :effect       (and (accepting-a-0) (not (at-a-0-init)) (at-a-0-accept-0)
                     (not (sync-automaton-a-0)) (sync-ordinary)))
(:action sync-trans-a-0-accept-0-a-0-accept-0
  :parameters()
  :precondition (and (at-a-0-accept-0) (sync-automaton-a-0))
  :effect (and  (not (sync-automaton-a-0)) (sync-ordinary)))
```

Listing 3.5: The synchronisation operators of the strong constraint NFA.

An example of the modified original operators is shown in Listing 3.6. The predicate *(sync-automaton-a-0)* is a precondition of each synchronisation operator and an add effect for the original operators. Thus, the execution of the synchronised operators must follow the execution of original operators. Furthermore, this predicate is added as a delete effect for synchronisation operators to prevent the execution of two consecutive operators of this type. Similarly, the predicate *(sync-ordinary)* organises the execution of the ordinary operators.

```
(:action START-OVEN-M
  :parameters()
  :precondition (and (sync-ordinary))
  :effect       (and (not (sync-ordinary)) (sync-automaton-a-0)
                     (when (and (close))
                           (and (start) (heat)))
                     (when (and (not (close)))
                           (and (start) (error))))))
```

Listing 3.6: The Start Oven operator with the synchronisation predicates.

Listing 3.7 lists the initial state and the goal conditions of the produced verification-as-planning task. Note that the initial state specifies That the FNA must be in its initial state, and the synchronisation predicate *(sync-ordinary)* must be true to enable the plan to start with one of the original operators. Moreover, the goal condition mandates that the original planning goal must be satisfied. Additionally, the FNA must be in an accepting state to ensure the safety property is falsified before achieving the original goal. Furthermore, the goal condition also requires the synchronisation predicate *(sync-ordinary)* to be true. This is needed to ensure that the effects of the last original operator in the plan are monitored by the NFA. In our example, due to the structure of the NFA of the used strong constraint, this condition is not required because once the acceptance state is reached, it is impossible to leave this state, check Figure 3.4.

```
(:init
 (at-a-0-init) (sync-ordinary))
(:goal
  (and (sync-ordinary) (accepting-a-0) (heat))))
```

Listing 3.7: The initial state and the goal conditions of the Microwave oven verification-as-planning task.

Solving the verification-as-planning problem with a planner produces the plan presented in Listing 3.8. This plan represents a valid planning counterexample.

```
START-OVEN
SYNC-TRANS-A-0-INIT-A-0-ACCEPT-0
CLOSE-DOOR
SYNC-TRANS-A-0-ACCEPT-0-A-0-ACCEPT-0
```

Listing 3.8: A valid planning counterexample.

This is the same verification result we obtained using the model checker Spin in Section 3.5.1. To remove this counterexample, we modify the operator Start Oven by adding the condition that the door must be closed before this operator can be scheduled. Adding the precondition *(close)* to this operator means the conditional effect shown in Listing 3.9, which is part of the original operator Start Oven, becomes unachievable; hence, we remove it.

```
(when (and (not (close)))
      (and (start) (error)))
```

Listing 3.9: Start Oven operator.

Re-invoking the planner to solve the new verification-as-planning problem, after removing this condition effect from Start Oven, causes the planner to confirm this planning problem is unsolvable; hence, the new planning domain model satisfies the safety property that states no plan should permit the microwave oven to reach an erroneous state.

3.7 Experiments

In this section, we describe two experiments. The first one is conducted as a proof-of-concept experiment, which aims to showcase the robustness of the goal-constructed planning domain model verification method against invalid planning counterexamples. In this experiment, Spin and MIPS-XXL are used to perform four verification tasks on a published planning domain model. The second experiment investigates the behaviour of our approach by studying the overhead cost incurred by our method when compared with under-constrained verification methods.

3.7.1 Proof of Concept Experiment

This experiment shows how goal-constrained planning domain verification can verify safety properties using both the Spin model checker and the MIPS-XXL planner. We perform constrained and under-

constrained verification tasks to show how unlike the latter task our method does not return unreachable counterexamples. As an example, we consider the classical cave diving planning domain taken from the Satisficing Track of the IPC-2014 [97]. This planning domain model is chosen for this experiment because it has an important safety property which requires divers not to drown during diving. The problem consists of an underwater cave system with a finite number of partially interconnected locations. Divers can enter the cave from a specific location, entrance, and swim from one location to an adjacent connected one. They can hold up to four oxygen tanks and they consume one for every swim and take-photo action. Only one diver can be in the cave at a time. Finally, divers have to perform a decompression manoeuvre to go to the surface and this can be done only at the entrance. Additionally, divers can drop tanks in or take tanks from any location if they hold at least one tank or there is one tank available at the location, respectively.

The planning goals of this domain, as provided in the problem files in the IPC-2014, consist of two parts. The first part dictates the required underwater location of which a photo is to be taken (we call it mission goal) and the second part which mandates divers should return to the surface after completing the mission (we call it safety goal).

A critical safety property, p, is that divers should not drown, i.e. they should not be in an underwater location, other than the entrance, where neither the divers nor the location has one full oxygen tank at least.

To enable the planner and the model checkers to explore the entire state space, we considered only one diver and we modified some actions to enable the diver to go back into the water after a dive. These modifications are further explained in the commented simplified planning domain model PDDL file which is provided along with the tasks problem PDDL and Promela files online [1].

First, the planning domain model is translated from PDDL to Promela. Thus, the verification results using the translated model only hold provided that the translation is valid. The verification of the translation is outside the scope and focus of this thesis and left for future work.

In this example, the chosen planning goal is to have a photo of the first location, L_1, and to get the diver outside the water. The verification tasks are:

1 - Under-constrained verification with only the safety property p: Both Spin and MIPS-XXL found the counterexample $\langle$*prepare-a-tank, enter-water, swim*$(L_0, L_1)\rangle$. Indeed, this counterexample leads the diver to a drowning state. At the end of this sequence, the diver will have consumed their oxygen tank and will be in underwater location L_1. This is not the entrance, so they can not surface and they do not have an oxygen tank to swim back to the entrance. However, this is not a plan because it does not achieve any useful goal. Therefore, it will not be produced by any sound planner when it is used in a practical scenario (taking a photo of any location).

2- Verification with safety property and incomplete goal (mission goal only): Both Spin and MIPS-XXL returned $\langle$*prepare-tank, prepare-tank, enter-water, swim*(L_0, L_1)*, take-photo*$\rangle$. This counterexample achieves the goal and violates the property. However, without the safety part of the goal, it would be possible to generate plans that imply divers should swim to an underwater location and take a photo of it without requiring the divers to return to the surface. These kind of plans are illegal as they do

[1] `https://github.com/Anas-Shrinah/Goal-constrained-planning-domain-model-verification-repository`

not respect the safety part of the goal. Therefore, such sequences are unreachable counterexamples, i.e. will never be produced by any sound planner while planning for a legal goal.

3- Verification using Spin with both safety property and proper goal but without the augmented model M': Spin found a counterexample $\langle$*prepare-tank, prepare-tank, prepare-tank, prepare-tank, enter-water, swim*(L_0, L_1)*, take-photo, swim*(L_1, L_0)*, decompress, enter water, swim*$(L_0, L_1)\rangle$. This counterexample achieves the goal and violates the safety property but only after the goal is achieved. Therefore, this is also an unreachable counterexample because a sound planner will terminate after achieving the goal and any counterexample that violates the property after achieving the goal will not be returned. Hence, it is invalid.

4- Goal-constrained planning domain verification, as presented in this chapter. The result was: No plan is returned by the planner MIPS-XXL with complete exploration and no counterexample is returned by Spin with exhaustive verification mode. This means the planning domain model has no provision of producing a plan that can violate the safety property before achieving the goal, i.e. this model is safe with respect to the given property and goal pair.

Though the counterexamples returned by the incomplete verification tasks number one, two and three are obviously unreachable and should not misguide the designers to overcomplicate the model, in a real world sized application such invalid planning counterexamples can be critical and much more difficult to recognise and avoid. We expect that our proposed concept can save practitioners a huge amount of person-hours trying to alter planning domain models for behaviours that their planners will never experience in practice.

3.7.2 Performance Experiment

To evaluate the feasibility and the behaviour of our approach, we designed two experiments to investigate how constraining the verification with the planning goal impacts the verification cost. This cost is measured by the number of states evaluated by the verification tools to confirm whether or not a counterexample exists.The scripts to repeat the experiments along with the data are available online.[2]

The first experiment focuses on comparing the cost of both under-constrained and goal-constrained verification tasks while varying the safety property violation depth in order to explore situations with and without a valid planning counterexample. We synthesised a fully reachable model that consists of one critical and three independent variables, each with a range from 0 to 31. Each variable has two actions, one to increase and one to decrease its value by one. The goal is achieved when the critical variable value reaches 14. The error condition (the violation of the safety property) is varied from the value of the critical variable being 1 to 31. The value of the critical variable that represents a safety property violation is hereafter termed “error depth”. For instance, in an experiment run when the safety violation happens when the value of the critical variable is equal to five, we say the error depth in this run is equal to five. In this case, the error is shallower than the goal because any trace that achieves the goal must first falsifies the safety property. On the other hand, in an experiment run when the the error depth is

[2]Repository like is provided in the footnote in Section 3.7.1

equal to 20 (the safety violation happens when the value of the critical variable is equal to 20), the error is deeper than the goal because any trace that falsifies the safety property must first achieves the goal.

The range of the variables is chosen as 31 to expose any possible trends. Consequently, the number of variables is set to four to allow the model to be explored within a memory limit of 10 GB.

The second experiment investigates the effect of the early termination of the verification process, after achieving the goal, on the cost of verification tasks while increasing the depth of the planning goal. The model used in this experiment has one critical and four independent variables, each with a range from 0 to 15. Variables have two actions as in the previous model. This time, there is no error in the model and the goal condition is varied from critical value 1 to 14. The variables' range is reduced to 15 to permit increasing the number of variables to five while keeping the required memory within the 10 GB constraint. Both experiments are performed using the Spin model checker and the MIPS-XXL planner with breadth first search option.

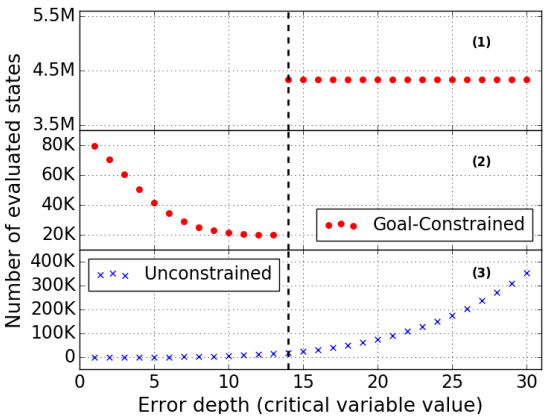
3.7.3 Results and Discussion

In the first experiment, our approach showed broadly similar behaviour when it was applied using Spin and MIPS-XXL in Figure 3.5a and 3.5c. Note that the aim of these experiments is to showcase the feasibility of using our approach and to explore its behaviour, rather than comparing the performance of the verification tools. We believe such comparison depends heavily on the model under verification, for more insights the reader is referred to [98–100]. Ergo, we focus our discussion on the results obtained from model checking.

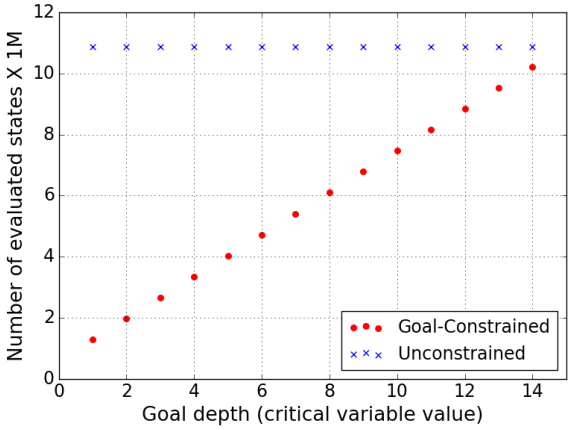
The vertical line in Figure 3.5a marks the goal level (critical value of 14) and splits the graph into two areas. On the right-hand side, the errors are deeper than the goal, i.e. the errors can only be reached after the goal is achieved. Thus, these errors are regarded as invalid planning counterexamples by our method as per Definition 3.1. Therefore, unlike under-constrained verification approaches, our method continues its exhaustive search to confirm the non-existence of any valid planning counterexamples. Thus, our method evaluates the maximum number of states for these verification tasks as shown in Figure 3.5a-(1).

On the left-hand side, the errors are shallower than the goal, i.e. the errors are reachable before achieving the goal. Hence, these errors are considered as valid planning counterexamples according to Definition 3.1. For the same verification task, Figure 3.5a-(2) shows that our method assesses more states than the under-constrained approaches as depicted in Figure 3.5a-(3). This is due to the fact that after finding an error, a safety property violation, our method keeps exploring and searching for a path to the planning goal while traditional methods terminate as soon as an error is found. However, the short counterexamples returned by these methods may or may not be valid planning counterexamples, whereas our method is guaranteed to return valid planning counterexamples only. The extra states visited by our approach are the cost associated with this guarantee.

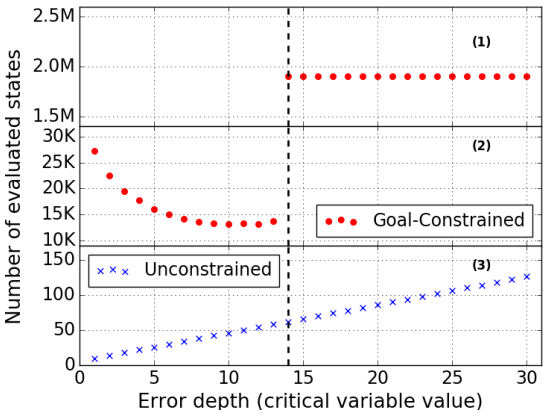
In Figure 3.5a-(2) (and in Figure 3.5c-(2), respectively), we notice a drop in the number of evaluated states by our method as the error depth gets closer to the goal depth. This is attributed to the fact that the safety property (state trajectory constraint) in the model checker (planner) is translated into an automaton. This automaton influences the state space exploration during the verification process. The



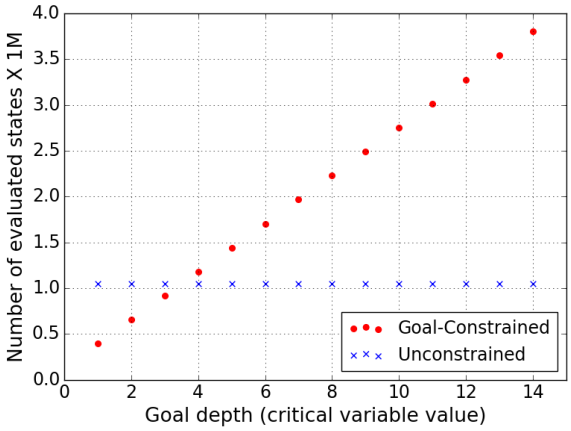
(a) Evaluated states vs error depth using Spin



(b) Evaluated states vs goal depth using Spin



(c) Evaluated states vs error depth using MIPS-XXL



(d) Evaluated states vs goal depth using MIPS-XXL

Figure 3.5: The behaviour of Goal-constrained verification method with different verification tasks using Spin and MIPS-XXL.

automaton has a transition that is activated when an error is reached. Therefore, if an error is reached in an early stage in the verification, the error transition is triggered and the verification tool is forced to explore more states than if the error transition was triggered closer to the goal. Once both the error and the goal transitions are triggered, then the automaton reaches an acceptance state. Thus, the search terminates with a valid planning counterexample.

In the second experiment, Figure 3.5b shows that when using Spin with a planing domain model with no counterexample, our approach explores fewer states than under-constrained verification methods. This reduction in the verification cost is realised by the early termination of the verification search once the goal is achieved and no error could be found at shallower depths. This advantage of the goal-constrained verification approach comes at the cost of limiting the verification results to a single planning goal. Additionally, it is observed that the number of evaluated states by the goal-constrained verification method rises as an effect of the increasing goal depth. This is caused by the expansion of the part of the model that needs to be checked as the goal depth increases. On the other hand, the under-constrained

methods visit a constant number of states as they are independent from the goal depth by definition.

Another interesting observation when using the planner in Figure 3.5d is that our method explores more states than the under-constrained verification approaches when the planning goal depth is more than three. This behaviour is caused by the interaction of two factors. In our approach, MIPS-XXL translates the state trajectory constraint to an automaton which is then incorporated in the planning domain model. Thus, the model used in our method is more complicated than the model used by the under-constrained approaches were state trajectory constraints are not used. After a certain depth of the planning goal, the extra states evaluated as a result of the additional state trajectory constraint in our method outweigh the saving from the early termination of the verification process.

Our method's behaviour is independent from the verified planning domain model. Our approach will always evaluates less states than unconstrained methods when the domain models do not have provisions to falsify the given safety properties. On the other hand, our approach will evaluates more states than unconstrained methods when the the error depth is shallower than goal depth. In such cases unconstrained verification approaches terminate as soon as a violation of the safety property is found, while our approach continues the search process to confirm the trace that violates the safety property will also achieve the planning goal.

3.8 Inherently Safe Planning Domain Models

The ultimate objective of planning domain model verification is to ensure that plans produced by the verified domain models satisfy a given specification. An alternative and more efficient way of achieving this goal is to extract plan constraints from the specification and then include them in the domain model. A sound planner using this constrained domain model cannot produce any plan that could violate these constraints. This idea was first noticed in 2005 by Smith et al. [12] but was dismissed as it was not possible to describe overall plan constraints using PDDL 2.2. However, in 2006, Gerevini and Long [101] proposed an extension to the PDDL 2.2 language that allows the expression of plan state trajectory constraints. The extended language called PDDL3.0 was proposed for the fifth international planning competition (IPC-5).

Smith et al. [12] provided an example of a system consisting of a camera, a solid-state recorder and a radio, and a requirement that for all plans, if an image is taken and stored, then it is eventually uplinked. With the hard state trajectory constraints, this property can be expressed as *sometime-after((image is taken and image is stored) image is uplinked)*. With this constraint, any sequence of actions that does not respect this property would not be returned as a plan.

The idea of using plan constraints to design inherently safe planning domain models was stated by Gerevini et al. [95] in 2009. Though including specification properties in the domain model as strong constraints is enough to guarantee that sound planners using the constrained planning domain models will produce plans that meet the specification, this method cannot find any errors in the planning domain model. Instead, it will just ensure these errors, if any, are masked and prevented from affecting any plans that could possibly be generated using the modified domain model. As such, this method can be

seen as a safety defence layer, a firewall, that prevents any potential property violation. Nevertheless, note that undetected bugs in a domain model could cause what would have been valid plans to be masked, thus unnecessarily restricting the planner. Therefore, further verification efforts are needed to reveal and rectify any underlying errors.

We consider including plan constraints in planning domain models to be good practice for designing inherently safe domain models. The effort of extracting formal properties from specifications and inserting them as constraints in planning domain models is a small investment in return for the huge benefit of guaranteed safe plans.

3.9 Related Work

Closely related, but different, is the work by Albarghouthi et al. [99]. Their main objective is to treat verification as a planning task, whereas our aim is to demonstrate how model checkers and planners can be used for domain model verification. They proposed to perform system model verification using classical planners. To do this, they first translated the model of the system under verification into a planning domain model. Then, the negation of the safety property to be established is used as the goal for the planner, which is then consulted to find a plan that acts as counterexample for the given property. In our study, because our aim is to verify planning domain models against a given safety property with respect to a specific goal, we use state trajectory constraints to restrict counterexamples to identify plans that can achieve the planning goal while falsifying the safety property. In their work the negation of the safety property is used as the goal, whereas, in our method, the negation of the safety property is represented as a state trajectory constraint and the goal is the given planning goal.

Raimondi et al. [10] also applied verification-as-planning to verify planning domain models, starting from LTL specifications. This work fundamentally differs from our work. They tested the impact of individual atomic propositions on the validity of the overall verified property by translating the specification properties into trap formulas. However, their method does not consider the interaction between property testing and the original planning goal. Note that finding a planning constraint to exercise a specific atomic proposition is not enough to ensure the constraint itself would be exercised during the planning process. For example, a planning goal might be achieved through a state trajectory that does not exercise the hard constraint used to represent the tested property. Our research is mainly focused on investigating this interaction. Therefore, we use state trajectory constraints to guarantee the property is tested while achieving the planning goal.

Goldman et al. [102] also used classical planners for planning systems verification, but they examined verifying plans rather than domain models. They proposed an approach that uses classical planners to find counterexamples for a given planning problem and plan instance. Their work and ours are related in that both suggest performing planning verification for a specific planning problem rather than attempting under-constrained verification of a planning system. However, their work is limited to the verification of single plan instances, whereas our method verifies all potential plans that can be

spun from a domain model for a specific goal.

Among others, the researchers in Smith et al. [12], Cesta et al. [13], Penix et al. [91], Khatib et al. [93], Havelund et al. [94] used model checkers to verify planning domain models. They translated the respective domain models into the input language of the selected model checker. The model checker is then applied to verify the domain model with respect to a given specification property. Similarly, we also propose a method to verify domain models using model checkers. However, our method differs from the others in two aspects. First, in the way we define the planning domain model verification problem, and, second, in the way we use model checkers to perform verification. As explained in Section 3.3, we constrain the verification of planning domain models with a specific goal. In contrast, previous studies perform under-constrained verification of domain models, i.e. they leave the goal open. As discussed in Section 3.2, the under-constrained goal may cause the model checker to return counterexamples that are unreachable when a planner uses the DUV. On the other hand, when the goal is constrained for verification, then we show that the returned counterexamples, if any, are guaranteed to be reachable by any sound planner. The second difference is that, after the planning domain model is translated to the model checker's input language, we augment the model transitions, introducing the negation of the goal as a new constraint, thereby forcing the model checker to terminate once the goal is reached. This modification prevents the model checker from returning counterexamples that falsify the given property after satisfying the goal.

3.10 Summary

Verifying planning domain models is essential to guarantee the safety of planning-based automated systems. Invalid planning counterexamples returned by under-constrained planning domain model verification techniques undermine the verification results. They can mislead system designers to perform unnecessary remediations that can be prone to errors.

In this chapter, we introduced goal-constrained verification, a new concept to address this problem, which restricts the verification task to a specific goal. This limits counterexamples to those practically reachable by a planner that is tasked with achieving the goal. We have demonstrated how model checkers and planning techniques can be used to perform goal-constrained planning domain model verification. Our experimental evaluation confirmed the feasibility of our method and presented its benefits and limitations compared to under-constrained verification methods.

CHAPTER 4

FUNCTIONAL EQUIVALENCE VALIDATION OF PLANNING DOMAIN MODELS

4.1 Introduction

The need for a technique to analyse planning domain models for functional equivalence has been highlighted in the literature [21, 103, 104]. One example application is the evaluation of the quality of planning model learning algorithms [22, 23]. This evaluation process can be achieved by using hand-crafted models to generate a number of plans which are then fed to the model learning method to produce the learnt planning domain models. The functional equivalence of the original and learnt models is then checked to evaluate the quality of the learning algorithm.

Another application is to validate the modifications performed by applying optimisation methods to planning domain models do not alter the functionality of the original models. This includes reformulation, re-representation [105] and tuning [106] of domain models in order to increase the efficiency of the planning process. Examples of domain reformulation include macro-learning [107–109], action schema splitting [110] and entanglements [111–113].

The idea of proving functional equivalence between two artefacts is not new; this concept is used to check program equivalence in software development [114–116] and verify circuit equivalence in hardware design [117–119].

To prove the functional equivalence of two planning domain models, we introduce a novel method that uses a planner to find and remove any redundant operator from the given domains. From the perspective of the functionality of planning domain models, a redundant operator is any operator that can be removed without changing the functionality of its domain. After that, the method employs an SMT solver to find a special mapping between the predicates of the two domains. This predicate mapping is defined such that it only exists between the predicates of the two domains if they are are guaranteed to

be functionally equivalent. Thus, it (constructively) proves the functional equivalence of two planning domain models. We refer to our algorithm as D-VAL.

In this chapter, we first formally define the functional equivalence of two planning domain models. Then, we classify domains into two types: simple and complex. Complex domains have at least one operator that has the same delete effects as another operator or a sequence of operators from the same domain. On the contrary, any domain that does not match the definition of complex domains is simple domain. We then prove that if two simple domains are functionally equivalent, our method can find a bijective predicate mapping that makes the two domains functionally redundant. Hence, our method is complete with regard to simple domains. On the other hand, our method cannot disprove the functional equivalence for complex planning domain models.

Furthermore, we introduce a test benchmark consisting of 74 functional equivalence validation tasks produced from 13 planning domain models from the International Planning Competition (IPC) [97] and perform empirical experiments on this benchmark to demonstrate the feasibility of our method. This chapter considers typed planning domain models, which have an equal number of predicates and are described using STRIPS subsets of the Planning Domain Definition Language (PDDL) without equality predicates.

The scope of our method is limited to planning domain models with an equal number of atoms. This restriction causes the following limitations. Reformulations that produce domains with supplementary atoms such as action schema splitting [110] and entanglements [111–113] are currently not covered in the scope of this research. Similarly, functionally equivalent domains with atoms in one domain derived from two or more atoms in the other domain are also out of the scope of this research. Despite these limitations, this research opens new research avenues by formalising and solving the problem of validating the functional equivalence of planning domain models.

Furthermore, erroneous domains that contain invalid operators or redundant atoms require some forms of preprocessing, which is not in the scope of this research. Moreover, symmetry reduction methods, like bagged representation [120], produce planning domain models that are functionally equivalent to the original domains only with respect to specific planning problems. Thus, such methods are not considered to produce planning domain models that are functionally equivalent independently from planning problems.

4.1.1 Chapter overview

This chapter is organised as follows. Firstly, Section 4.2 contrasts our method with related work. Then, Section 4.3 presents the planning theory concepts used in this chapter. Section 4.4 defines the concept of planning domain models' functional equivalence and its supporting concepts: the reach set of operators, the reach set of sequences of operators, and the reach set of planning domain models. After that, Section 4.6 introduces the tasks that need to be performed to remove redundant operators. Moving on, Section 4.7 explains the difference between simple and complex domains. Section 4.8 explains our method to validating the functional equivalence of simple planning domain models. In addition, this

section also introduces the theorems that form the theoretical foundation of our method and points towards the proofs of these theorems in the appendices. Section 4.9 explains our approach to validating the functional equivalence of complex planning domain models. The core of our method, the variable mapping SMT problem, is introduced in Section 4.10 and then detailed in Section 4.11. To provide readers with a concrete example of our approach, Section 4.12 illustrates the outputs of our method for validating the functional equivalence between the Mystery domain [121] and its undisguised version. The flowchart of the D-VAL algorithm is explained in Section 4.5. The method of randomly generating valid macros for the experiments is descried in Section 4.13. The results of the empirical experiments, along with the subsequent discussions, are reported in Section 4.14. Finally, Section 4.15 summarise the contributions of this chapter.

4.2 Related Work

Our definition of planning domain model functional equivalence is a relaxed version of the model weak equivalence defined by Shoeeb and McCluskey [103]. In this study, the authors define two flavours of domain equivalences. They first define the planning domain model strong equivalence as a one-to-one mapping that maps the names of predicates, variables, operator schema and types from one domain to another. Additionally, they define the planning domain model weak equivalence as a relation between two domains when both can represent the same set of planning problems, and every valid plan that can be produced from the first domain is also a valid plan in the second domain under a certain bijective mapping of the components of the two domains.

We relax their definition in two ways. Firstly, we require a bijective mapping between the predicates, variables and primitive operators, but not necessarily between every operator schema of functionally equivalent domains. Secondly, we consider planning domain models functionally equivalent if they can solve the same set of planning problems, even if the solutions differ.

In fact, the weak and strong equivalence relations presented in the authors' research are identical. For two domains to produce the same set of plans for the same set of problems, the directed graphs representing the reachable state space of both domains must be isomorphic. Isomorphism is a strong bi-simulation relation. For the directed graphs of the reachable state space of two domains to be isomorphic, both domains should be identical apart from the names of the domains' components. Hence, the requirement to generate the same plans for every planning problem can be achieved only through strong equivalence, not by a weak equivalence, as proposed in [103].

The logic behind our definition is that if two methods are always proven to find valid answers for any set of problems, not necessarily the same answers, then these two methods can do the same job regardless of the steps taken by each method. Thus, we consider them functionally equivalent. With this mindset, we propose our planning domain model functional equivalence definition. If two domains can be used to solve the same set of problems (under a certain mapping) regardless of the actions taken, then we call them functionally equivalent domains. This research extends the work presented by Shoeeb and

McCluskey [103] by formally defining planning domain model functional equivalence and proposing a method to prove this equivalence. Moreover, we support our method with a sound set of theorems along with their proofs, which form the theoretical basis of our method.

McCluskey et al. [21] argue that the correctness of domain models is an essential factor in the overall quality of the planning function. They consider a knowledge model to consist of a planning domain model and a planning problem instance and suggest translating the components of the knowledge model into assertions. A knowledge model is then said to accurately capture its requirements if the interpretation given by the requirements satisfies the assertions in the knowledge model. Our notion of "functional equivalence" is closely related to their notion of "accuracy". If two planning domain models are functionally equivalent, both domain models satisfy the same functional requirements and represent these requirements to the same degree of accuracy.

Furthermore, McCluskey et al. [21] outlined an approach to check the accuracy of operators from a planning domain model with the help of a single planning instance. Their approach is limited to individual planning instances, whereas our proposed method is independent of the planning problems. Besides that, this research describes a working implementation of our method based on proven theoretical bases; moreover, our research reports on the feasibility of our method through empirical evaluation.

Cresswell et al. [76] define the equivalence of two domains with regard to a planning instance using graph isomorphism. According to their definition, two domains are equivalent if the directed graphs representing the reachable state space of both domains are isomorphic. This criterion proves the equivalence between the two domains for individual planning instances. On the contrary, our method proves the functional equivalence for any set of problems. Another subtle difference is that our method is invariant to the domains' state space paths because it defines the functional equivalence as a weak equivalence relation, whereas their approach checks strong equivalence. Hence, their approach is sensitive to any variation in the possible state space paths between the two domains. The path sensitivity in their method is due to the fact that their definition is based on graph isomorphism. Furthermore, proving isomorphism between two graphs is computationally dependent on the size of the planning problems, whereas the computational cost of our method is dependent only on the size of the given domains.

The planning domain models functional equivalence problem is also related to model reconciliation and maintenance research. The following paragraphs contrast the models' functional equivalence problem and our validation approach to some of the prominent research in these areas.

The model reconciliation problem introduced by Chakraborti et al. [122] is concerned with changing one model to make it closer to another model with respect to the cost of a given plan. For example, given two different planning domain models M_1 and M_2, the model reconciliation task aims to change M_1 to $\widehat{M_1}$ such that an optimal plan produced using M_2 is also optimal when interpreted using the modified model $\widehat{M_1}$. Note that two reconciled planning domain models are not necessarily functionally equivalent as the reconciliation process is performed with respect to individual planning instances. Moreover, unlike our method, the model reconciliation approach proposed in [122] assumes that both models M_1 and M_2 have the same set of predicates and operators. Our approach relaxes this condition and aims to find a

mapping between the predicates of the two domains that makes the two domains functionally equivalent.

Sreedharan et al. [123] generalise the work proposed by Chakraborti et al. [122] by learning a model approximation of the mental model of the human user through interacting with the user. First, the user is asked whether some transitions from the Markov Decision Process (MDP) that represents the robot model are explainable. Then, after the human model is learnt, this method reconciles the user-approximated model with the robot model with respect to some execution trace. This research discusses the difference between model reconciliation processes that intended to explain an optimal policy or an execution trace of an MDP. However, the reconciliation method proposed in this work is limited to explaining individual execution traces of an MDP; thus, this method also cannot be used to prove the functional equivalence of two models. Moreover, the transitions of the MDP model, including the states and the action labels, are a subset of the transitions of the learned model which is then reconciled with the MDP model. In contrast, as explained earlier, our method is able to prove the functional equivalence of models with different states and transition labels.

The model maintenance problem addresses the challenge of updating a planning domain model M_1 to match the always-evolving mental model of the user M_2. In dynamic environments, a user must update their understanding to reflect changes in the environment. As a result of these changes, the two models, M_1 and M_2, drift apart. A system called Marshal that solves this problem is proposed by Bryce et al. [29]. Marshal interacts with human users through queries to provide observations to a particle filter which anticipates the model that most closely resembles the mental model of the user. This method uses a stochastic process to learn a model through interaction with a user; thus, it is not automated and cannot be guaranteed to learn the actual mental model of the user.

Macro generation or macro-learning is a technique that helps increase the speed of the planning process [107–109, 124, 125]. Macro-learning methods augment planning domain models with macro operators. These macro operators create shortcuts in the state spaces of planning problems, hence improving the search for goal states. The ultimate goal of macro-learning methods is to find the most effective macros in the least amount of time so that these macros can be added to a given planning domain model. While macro-learning methods add shortcuts to the state spaces of planning problems, our approach removes such shortcuts during the process of validating the functional equivalence of planning domain models. Our method tests if any of the operators of a given domain is a macro operator. An operator is considered a macro operator if a consolidated sequence of operators from the same domain has the same add and delete effects as the tested operator. After macros are identified, we remove them from the given domain. Removing macro operators from a planning domain model does not affect the reachability of its planning problem [108], but it is an essential part of the process of proving functional equivalence between planning domain models. Another distinctive difference is that, unlike the macro-learning problem, our validation approach must find all possible macros in the given planning domain models. On the other hand, macro-learning methods do not have to be exhaustive.

Domains functional equivalence and model recognition solve different problems. Nevertheless, the model recognition method proposed by Aineto et al. [126] and our approach both compile a

search problem into a classical planning problem. The model recognition problem is concerned with identifying the model that better explains a partially observed plan execution. Given a set of models $\mathbb{M} = \{M_1, \ldots, M_n\}$ and a partially observed plan execution $\mathcal{O}$, the method proposed in [126] recognises the model M_i that best explains the given observation $\mathcal{O}$. This method compares the models in $\mathbb{M}$ based on the number of atoms insertions and deletions that need to be applied on the operators of each model so that the modified models satisfy the observation $\mathcal{O}$. This number of modified atoms is called the edit distance between a model M and its modified model M'. The model that requires the minimum edit distance to produce a modified model that satisfies $\mathcal{O}$ is more likely to be the model that best explains the observation $\mathcal{O}$. The authors compile the problem of calculating the edit distance needed to modify a model to satisfy a given observation $\mathcal{O}$ as a meta-planning problem. The solution to this problem is a plan that consists of meta-actions. Each meta-action inserts or deletes an atom to one of the action schemata of the model M or validates the application of a modified action schema to the observation $\mathcal{O}$. The initial state of this meta-planning problem is a model M and observation $\mathcal{O}$. The goal is met when the planner finds the set of meta-actions that transform M into M' which satisfies $\mathcal{O}$.

Our method also compiles a search problem into a classical planning problem. The approach proposed in this chapter formulates the problem of finding a macro operator with the same add and delete effects as a given operator from the same domain as a planning problem. This task is needed to check whether an operator is primitive as part of validating the functional equivalence of planning domain models.

4.3 Preliminaries

This chapter follows the classical planning representation introduced in Section 2.1.1 in Chapter 2. In addition to the notations presented in the Background chapter, this section defines the concept of comparing the preconditions and effects of operators that belong to the same planning domain model. When we compare two atoms from different operators in one domain, we compare their predicates and the order of the types of their parameters. So, we say two add effects are the same if the atoms that represent them are similar in every aspect up, but not necessarily, to the variable names. The same concept applies to the comparison of delete effects and preconditions.

Definition 4.1. Consider two operators, o_1 and o_2, from a planning domain model D. We say the two operators have the ***same add (delete) effects*** if and only if for each add effect t_1 in o_1 there is an add (delete) effect t_2 in o_2 such that the atoms that represent t_1 and t_2 are similar in every aspect up but not necessarily to the variable names.

If o_1 and o_2 have the same add effects, we write $\text{Add}(o_1) = \text{Add}(o_2)$, and if they have the same delete effects, we write $\text{Del}(o_1) = \text{Del}(o_2)$. If $\text{Add}(o_1) = \text{Add}(o_2) \wedge \text{Del}(o_1) = \text{Del}(o_2)$, then o_1 and o_2 have the ***same effects*** .

This definition also applies to the comparison of the preconditions of operators. We also need to define the relation of containment between the sets of the preconditions and the effects of operators.

Definition 4.2. Consider two operators, o_1 and o_2, from a planning domain model D. We say the set of the preconditions of o_1 is a subset of the set of the preconditions of o_2 if and only if for each precondition $t_1$in o_1 there is a precondition t_2 in o_2 such that the two preconditions are the same, then we write $\text{Pre}(o_1) \subseteq \text{Pre}(o_2)$

This definition also applies to the comparison of the add and delete effects of operators.

4.3.1 Function Definitions

In this section, we will define the functions used in the definitions and theorems proposed in this chapter. We will use the operator Navigate from the Rover domain to provide examples of the application of these functions.

```
1 (:action navigate
2  :parameters (?x - rover ?y - waypoint ?z - waypoint)
3  :precondition (and (can_traverse ?x ?y ?z) (available ?x)
4                (at ?x ?y) (visible ?y ?z))
5 :effect (and (not (at ?x ?y)) (at ?x ?z)))
```

Listing 4.1: The operator "Navigate" from the Rover domain.

- The function Pred returns the predicate name of a given atom. Pred: atom → predicate name. For instance, $\text{Pred}((at\ ?x\ ?y)) = at$.
- The function Arity returns the arity of a given atom. Arity: atom → $\mathbb{N}$. For instance, $\text{Arity}((at\ ?x\ ?y)) = 2$.
- The function Var returns the set of variables of a given atom. Var: atom → set of variables. For instance, $\text{Var}((at\ ?x\ ?y)) = \{?x, ?y\}$.
- The function Variables returns the set of variables of a given operator. Variables: operator → set of variables. For instance, $\text{Variables}(navigate) = \{?x, ?y, ?z\}$.
- The function Position returns the position of a given variable in a given atom. Position: (variable,atom) → $\mathbb{N}$. For instance, $\text{Position}(?x, (at\ ?x\ ?y)) = 1$.
- The function Predicates returns the set of predicates of the atoms that appear in a given operator. Predicates: operator → set of predicates. For instance, Predicates(navigate) = $\{\text{can\_traverse}^3, \text{available}^1, \text{at}^2, \text{visible}^2\}$. The superscript of the predicate name is its arity.
- The function Atoms returns the set of atoms of a given operator. Atoms: operator → set of atoms. For instance, Atoms(navigate) = $\{(can\_traverse\ ?x\ ?y\ ?z), (available\ ?x), (at\ ?x\ ?y), (visible\ ?y\ ?z), (at\ ?x\ ?z)\}$.
- The functions Domain(f) and Range(f) return the domain and range of the function f, respectively.

4.3.2 Consolidating Sequence of Operators

In the description of our approach, we refer to the terms sequence of operators, configured sequence of operators and consolidated sequence of operators. This section explains these terms and the process of consolidating a sequence of operators.

Consolidating the operators of a sequence of operators is an iterative process. The procedure starts by consolidating the first two operators; then, it consolidates the next operator with the outcome of the previous consolidation step. The consolidation process continues until all operators are considered. The result of each consolidation process is a single operator. The produced operator is a macro operator built from the consolidated operators. If a sequence of operators consists of a single operator o, then the product of consolidating this sequence of operators is the operator o.

The consolidation procedure unifies or unites the parameters of the two given operators based on the types of their parameters. Suppose the type of the considered parameter is in the types of the parameters of the other operator. In that case, the consolidation procedure can either unify or unite this parameter with some of the parameters of the other operator. The unification process unifies a parameter from one operator with a parameter from the other operator if both parameters are of the same type. On the other hand, the unionisation process adds the considered parameter to the parameters from the other operator with the same type. However, suppose the type of the considered parameter is not in the types of the other operator. In that case, the consolidation procedure must perform a unionisation process to add the considered parameter and its type to the parameters and types of the produced macro.

For a given unification and unionisation configuration of the parameters of the two operators o_1 and o_2, the consolidation process produces a macro m if none of the atoms in the preconditions of o_2 are in the delete effects of o_1. Otherwise, consolidating the two operators o_1 and o_2 is an invalid process for the given unification and unionisation configuration. If the consolidation process is valid, the preconditions, add effects and delete effects of m are defined as follows. The preconditions of m are the preconditions of o_1 and the preconditions of o_2 that are not supported by add effects from o_1. The add effects of m are the add effects of o_2 and the add effects of o_1 that are not in the delete effects of o_2. The delete effects of m are the delete effects of o_2 and the delete effects of o_1 that are not in the add effects of o_2.

So, a macro is made of a sequence of operators with a specific unification and unionisation configuration of the parameters of the operators such that the consolidation of this sequence of operators is a valid process. We call such a sequence of operators a “configured sequence of operators”. Thus, we say a macro is made from a configured sequence of operators. For brevity, from here onwards, we will refer to “configured sequence of operators” just as “sequence of operators”. This does not create any ambiguity because we do not use the concept of unconfigured sequence of operators.

In the next section, we define the reach sets of operators, sequence of operators and planning domain models for any set of objects. In addition to that, we define the functional equivalence of planning domain models.

4.4 The Definition of the Functional Equivalence of Planning Domain Models

The functionality of a planning domain model is characterised by the set of planning problems that can be solved using this domain. For a domain to support solving a planning problem, a planner has to be able to use this domain to produce the required transitions in the state space from the initial state to one of the goal states. Thus, for a given set of objects, we call the set of all tuples of the start and the end states of all transitions that can be produced using the operators of a planning domain model as the reach set of this domain. Note that these transitions can result from applying any legal sequence of actions to any state that satisfies the preconditions of the first action in the sequence of actions. A legal sequence of actions means that all preconditions of each action are satisfied in the state that results from applying the previous action in the sequence. Therefore, two planning domain models with equal reach sets are considered functionally equivalent as they can solve the same set of problems. Defining the reach set of a planning domain model requires the following definitions.

Definition 4.3. *The reach set of an operator o* over a set of objects *Obj* is defined using the set of actions $A_{\{o,Obj\}}$ which is instantiated from the operator o with the set of objects *Obj* as $\Gamma(o, Obj) = \{(s, \gamma(s,a)) \mid s \in S,\ a \in A_{\{o,Obj\}}$ and a applicable in $s\}$.

Definition 4.4. *The reach set of a sequence of operators seq* over a set of objects *Obj* is defined using the set of action sequences $\Pi_{\{seq,Obj\}}$ which are instantiated from the sequence of operators *seq* with the set of objects *Obj* as $\Gamma(seq, Obj) = \{(s, \gamma(s,\pi)) \mid s \in S,\ \pi \in \Pi_{\{seq,Obj\}}$ and π applicable in $s\}$. Where $\gamma(s,\pi)$ is the successor state of a state s when the sequence π of actions is applied. A sequence of operators could consist of a single or many operators where some items might be repeated.

Definition 4.5. The *reach set of a planning domain model D* over a set of objects *Obj* is then defined as $\Gamma(D, Obj) = \bigcup_{seq \in \mathrm{SEQ}} \Gamma(seq, Obj)$, where SEQ is the set of all possible sequences of operators generated from D.

For two planning domain models to have equal reach sets, both domains must have the same predicates. We can relax this condition by requesting only a bijective mapping between the predicates of equal arity in the two domains. Under such mapping, two domains with different predicates but with an equal number of predicates with equal arity can be used to represent the same state space. Thus if these two domains are functionally equivalent, they can have the same reach sets under this mapping for any set of objects.

Now we will informally define the planning domain model functional equivalence. Two planning domains, D_1 and D_2, are ***functionally equivalent*** if and only if there is a bijective mapping F_p from the predicates of D_1 to the predicates of D_2 with equal arity such that when the predicates of D_1 are substituted with the predicates from D_2 using the predicate mapping F_p, the reach set of the produced domain $F_p(D_1)$ is equal to the reach set of D_2 for any set of objects.

Definition 4.6. Two planning domain models $D_1 = (P_1, O_1)$ and $D_2 = (P_2, O_2)$ are ***functionally equivalent***, $D_1 \equiv_{Func} D_2$, **iff**:

1. $\exists F_p : P_1 \twoheadrightarrow P_2$ where $\text{graph}(F_p) = \{(p, F_p(p)) \in P_1 \times P_2 : p \in P_1 \text{ and } arity(p) = arity(F_p(p))\}$
2. $F_p(D_1)$ is the image of D_1 using F_p to substitute its predicates with those of D_2.
3. $\forall Obj : \Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj)$.

Note that the functional equivalence of planning domain models is independent of the planning tasks for which these domains might be used. This relation depends only on the set of predicates and operator schemata in the given domains. Therefore, this relation needs to be proven with respect to the planning domain models regardless of any set of objects, initial states or goal states.

The functional equivalence of planning domain models is a reflexive, symmetric and transitive relation; thus, the functional equivalence of planning domain models is an equivalence relation. According to this definition, the functional equivalence between two planning domain models can be proven by showing that, for any set of objects, the reach set of one domain is equal to the reach set of an image of the other domain under a certain predicate mapping.

Throughout this chapter, we will compare the reach sets of operators from two different domains for any set of objects. For brevity, we will not repeat the phrase "for any set of objects" unless we need to re-emphasise the scope of our method. Furthermore, since the operators of two different domains have different predicates, their reach sets will always be different unless they are compared through a predicate mapping. Thus, whenever we say the reach set of an operator o from a domain D_1 is equal to the reach set of an operator o' from another domain D_2, we mean the image of the reach set of o under a predicate mapping is equal to the reach set of the operator o'. In addition, when we say the reach set of o is not equal to the reach set of o', we mean the image of the reach set of o under any predicate mapping is not equal to the reach set of o'. Nevertheless, we will clarify what predicate mapping is used in some comparisons when omitting its mention can cause ambiguity. Moreover, the predicate mappings to which we refer implicitly or explicitly when we compare the reach sets of planning domain models and operators are always bijective mapping between predicates of equal arities.

The reachability of planning domain models depends on the reachability of some indispensable operators; we call such operators primitive operators. Removing one of these operators from a domain changes the set of problems that can be solved using this domain, i.e. changes the domain reach set.

Definition 4.7. An operator o is a ***primitive operator*** in a domain D if the reach set of o is not a subset of the union of the reach sets of any sequences of operators in D excluding the operator o.

$$o \in Primitive(O) \rightarrow \forall Seq \subseteq SEQ(O \setminus o)\ (\Gamma(o) \nsubseteq \bigcup_{seq \in Seq} (\Gamma(seq))$$

where $SEQ(O \setminus o)$ is the set of all sequences of operators in D excluding the operator o.

In the special case when the set Seq has just one sequence, and this sequence has just one primitive operator, we can infer that the reach set of a primitive operator is not a subset of the reach set of any other primitive operator in its domain. Thus, primitive operators are not redundant by definition. The primitive operators of a domain decide which end states are reachable from which initial states. Therefore, primitive operators are the only source of functionality for their domains.

4.5 D-VAL Algorithm

D-VAL validates the functional equivalence of two planning domain models with an equal number of predicates with equal arities. The flowchart of the D-VAL algorithm is illustrated in Figure 4.1. The first step is to remove any operator with a reach set that is a subset of the reach set of another operator or a sequence of operators. Such operator are redundant and they do not change the reach sets of their domains; this step is explained in Section 4.6.1. This step must be applied to both domains. For two domains, simple or complex, to be functionally equivalent, they must have equal number of operators after removing macro operators. If the two domains have an equal number of operators, then D-VAL proceeds to find atom mappings according to Theorem 3 and checks that the individual mappings are consistent as explained in Section 4.10. If consistent atom mappings are found then the two domains are functionally equivalent. On the contrary, If consistent atom mappings are not found or the domains do not have an equal number of operators, then if both domains are simple, they are confirmed to be not functionally equivalent. On the other hand, if one or both of the domains are complex, then D-VAL cannot produce conclusive verdict on the functional equivalence of the given domains. The method to check the type of the provided domains is explained in Section 4.7.

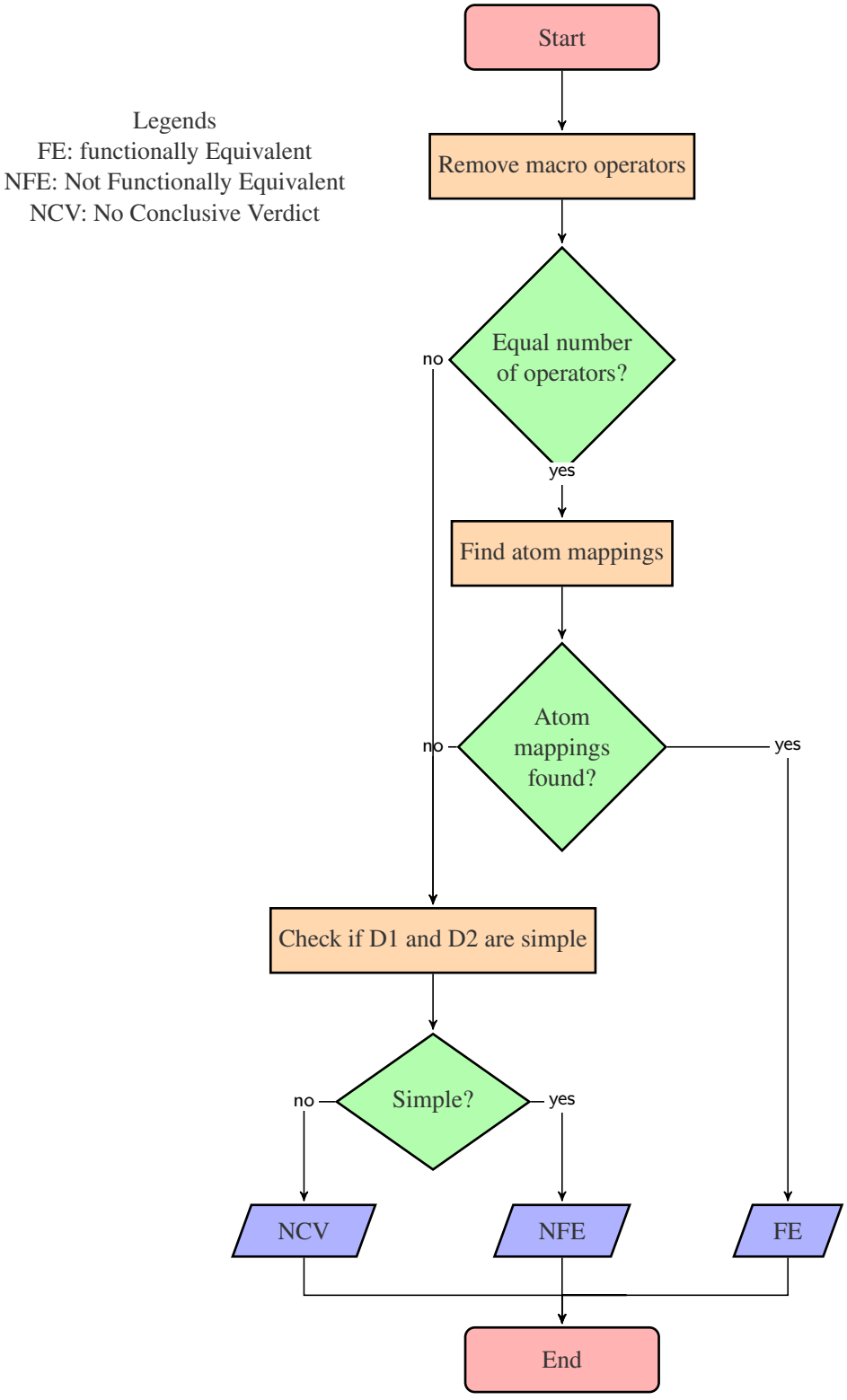
The theoretical foundation of validating the functional equivalence of simple planning domain models is presented in Section 4.8 for complex domain models in Section 4.9. D-VAL is able to provide conclusive verdict on the functional equivalence of simple domains because the theorems listed in Section 4.8 are proved as equivalence relations, whereas D-VAL cannot disprove the non functional equivalence of complex domains because the theorems introduced in Section 4.8 are proved as implications.

Next section explains how to find and remove non-primitive operators.

4.6 Identifying and Removing Non-primitive Operators

To prove the functional equivalence of two simple planning domain models, we have to find and remove non-primitive operators. Definition 4.7 states that an operator o from a domain D is primitive if the reach set of o is not a subset of the union of the reach sets of any subset of the set of all sequences of operators from D that do not include the operator o. So, if the reach set of an operator o is a subset of the reach set of another operator, the reach set of a sequence of operators or the union of the reach sets of multiple sequences of operators, then the operator o is a non-primitive operator. Section 4.6.1 explains the method we use to check if the reach set of a given operator is a subset of the reach set of a sequence of operators. This method also checks if the reach set of the given operator is a subset of the reach set of other operators.

Figure 4.1: D-VAL algorithm flowchart.



Though removing non-primitive operators is not a condition for proving the functional equivalence of complex domains, this step is performed by D-VAL as it removes some of the redundant operators in such domains. This task enables D-VAL to confirm the functional equivalence of functionally equivalent complex domains when the domains has such redundant operators.

The next step is to to verify whether the reach set of any remaining operator is a subset of the union of the reach sets of multiple sequences of operators. Section 4.6.2 presents our approach to deal with this check.

4.6.1 Checking if the Reach Set of an Operator is a Subset of the Reach Set of a Sequence of Operators (macro operator)

To check if the reach set of an operator is a subset of the reach set of a sequence of operators, we compare the preconditions and effects of the operator and the consolidated sequence of operators. The relation between comparing the reach sets of an operator o and a sequence of operators seq and comparing the effects and preconditions of o and seq is established in Theorem 1. This theorem discusses the case of two operators, which is a general case of the relation between the reach sets of an operator and a consolidated sequence of operators.

Theorem 1. *Consider a set of objects Obj, two operators o_1 and o_2 from the same domain where the two operators have the same parameters. The operators o_1 and o_2 have the same effects, and the preconditions of o_2 are a subset (proper subset) of the preconditions of o_1 **iff** the reach set of o_1 is a subset (proper subset) of the reach set of o_2.*

$$(\mathrm{Add}(o_1) = \mathrm{Add}(o_2)) \wedge (\mathrm{Del}(o_1) = \mathrm{Del}(o_2)) \wedge (\mathrm{Pre}(o_2) \subseteq \mathrm{Pre}(o_1)) \textit{ iff } \Gamma(o_1, Obj) \subseteq \Gamma(o_2, Obj) \tag{4.1}$$

$$(\mathrm{Add}(o_1) = \mathrm{Add}(o_2)) \wedge (\mathrm{Del}(o_1) = \mathrm{Del}(o_2)) \wedge (\mathrm{Pre}(o_2) \subset \mathrm{Pre}(o_1)) \textit{ iff } \Gamma(o_1, Obj) \subset \Gamma(o_2, Obj) \tag{4.2}$$

The prove of this theorem is provided in Appendix A.1. According to this theorem, to find if the reach set of an operator o from a domain D is a subset of the reach set of a consolidated sequence of operators from D that does not include o, we have to find a sequence of operators such that its consolidation has the same effects as o and the preconditions of the consolidation of this sequence is a subset of the preconditions of o. We formalise this search problem as a meta-planning problem.

Starting from the domain of the operator o, we create a modified domain D_m that does not have the operator o. Then, we define a dummy object for each variable in the parameters of the operator o. After that, we use the FF planner [34] to find a plan starting from a specific initial state s_{init} to a goal state from a set of states S_g that fulfils certain conditions. The specific initial state s_{init} is a state where only the truth evaluations of the atoms of the preconditions of o are true, and the truth evaluation of any other atom is false. For a state s_g to be in the set S_g, the truth evaluations of the atoms of the add effects of o in s_g must be true, and the truth evaluations of the atoms of the delete effects of o in s_g must be false. Moreover, the truth evaluation of any atom in the preconditions of o that is not in its delete effects must

be true in any goal state in S_g. This requirement excludes any sequence of operators with more delete effects than o. Furthermore, the truth evaluation of any atom other than the atoms in the add effects of o or the preconditions of o that are not in its delete effects must be false in any goal state in S_g. This condition prevents any sequence of operators with more add effects than o.

To improve the efficiency of this meta-planning problem, we reduce the size of the domain D_m by removing any operator which cannot be part of any sequence of operators that matches the requirement of this search problem. There are three rules an operator o_o must satisfy to be part of the sequence of operators *seq* such that the consolidation of *seq* has the same effects as o and the preconditions of *seq* are contained in the preconditions of o. These rules are derived from the description of the consolidation of sequences of operators in Section 4.3.2.

First rule *The types of the parameters of o_o must be in the types of the operator o.*

If an operator o_o has a parameter type that is not in the types of the operator o, then during the consolidation of any sequence of operators *seq* that contains o_o, the parameter with the extra type of o_o will be unionised with the other parameters in *seq* to produce a consolidated sequence of operators with more types than the operator o. If o has fewer types than the consolidation of *seq*, the reach set of o cannot be a subset of the reach set of *seq*.

Second rule *The number of parameters of any type in o_o must be equal to or less than the number of parameters of the same type in o.*

Suppose an operator o_o has more parameters of a type t than the number of parameters of the same type in the operator o. Then, even if all the parameters of the operator o_o are unionised during the consolidation of any sequence of operators *seq* that contains o_o, the sequence *seq* will have more parameters of the type t than the number of parameters in o of the same type. If o has less number of parameters from one type than the consolidation of *seq*, the reach set of o cannot be a subset of the reach set of *seq*.

Third rule *The predicates in the preconditions, add effects and delete effects of o_o must be in the union of the predicates of the preconditions, add effects and delete effects of o.*

If an operator o_o has an atom with a predicate that does not appear in the atoms of o, then no matter how we consolidate any sequence of operators *seq* that contains o_o, the consolidation of *seq* will have more preconditions, add effects or delete effects than o. Hence, the reach set of o cannot be a subset of the reach set of *seq*.

So, when we make the modified domain D_m, we can safely remove any operator that does not satisfy any of these rules.

Note that a plan is a sequence of actions which can be lifted to form a sequence of operators. Thus, if a plan is found for this meta-planning problem, we can conclude that the reach set of the operator o is a subset of the reach set of the sequence of operators that can be lifted from the plan. Therefore, the operator o is a non-primitive operator and must be removed. On the other hand, if no plan is found, then because Theorem 1 is proved as an equivalence, we can conclude that the reach set of o is not a subset of

the reach set of any sequence of operators from its domain. However, we cannot prove that the reach set of o is not a subset of the union of the reach sets of multiple sequences of operators. Hence, in this case, we cannot judge whether the operator o is primitive or not. Section 4.6.2 explains how to confirm if the reach set of any remaining operator is a subset of the union of the reach sets of multiple sequences of operators.

4.6.2 Checking if the Reach Set of an Operator is a Subset of the Union of the Reach Sets of Multiple Sequences of Operators

In the previous section, we have described how to check and remove any operator o if its reach set is a subset of the reach set of any sequence of operators that do not include o. Nevertheless, the reach set of o can still be a subset of the union of reach sets of multiple sequences of operators that do not include o; in this case, o will be a non-primitive operator according to the definition of primitive operators, Definition 4.7.

We define the transitions produced by applying the operators o to states where the preconditions of o are the only true atoms as the core transitions of o. Note that if the reach set of a sequence of operators is a superset of the reach set of the core transitions of o, this sequence of operators can produce any transition that can be produced by o. Hence, the reach set of this sequence of operators is a superset of the reach set of o. Actually, the reach set of a sequence of operators, seq, is a superset of the reach set of an operator o if and only if the reach set of seq is a superset of the reach set of the core transitions of o.

Since the reach set of any reaming operator o, after we apply the method in Section 4.6.1, cannot be a subset of the reach set of a single sequence of operators that do not include o, the reach set of o cannot be a subset of the union of the reach sets of a set of sequences of operators if this set has a single sequence of operators. Thus, for the reach set of o to be a subset of the union of the reach sets of a set of sequences of operators, at least two sequences must exist in this set such that the union of the reach sets of these sequences is a superset of the core transitions of o. Furthermore, the reach set of any sequence of operators cannot be equal to the set of the core transitions of o. Thus, the reach set of any of these sequences of operators must be a proper subset of the set of the core transitions of o. Therefore, for the reach set of an operator o to be a subset of the union of the reach sets of multiple sequences of operators, there must be at least two sequences such that the reach set of each of these sequences is a proper subset of the reach set of o.

Definition 4.8. We call a **S**equence of **O**perators with a reach **S**et that is a proper subset of the reach set of an **O**perator o an **SOSO** for the operator o.

If we find two or more SOSOs of an operator o, then it is possible for the reach set of o to be a subset of the union of the reach sets of these SOSOs. In this case, o can be a non-primitive operator. On the other hand, if we prove the non-existence of two or more SOSOs of an operator o, then it is not possible for the reach set of o to be a subset of the union of the reach sets of any set of sequences of operators. Hence, o will be guaranteed to be a primitive operator. So, to check if an operator o is primitive or not, we have to prove the existence or non-existence of two or more SOSOs for the operator o. Note that,

according to Theorem 1, the consolidation of an SOSO of an operator o has the same effects as o and has a set of preconditions that is a proper superset of the preconditions of o [1]. So, to find the SOSOs of an operator o, we have to find sequences of operators that, when consolidated, will have the same effects as o, and their sets of preconditions are proper supersets of the preconditions of o.

Rather than trying to find SOSOs for the operators of a given domain, we test whether the domain is simple or complex. If the domain is simple, then any operator o from this domain does not have the same delete effects as any sequence from this domain. Hence, the operator o does not have any SOSOs. Thus, The operator o is guaranteed to be primitive operator and its reach set is guaranteed to be not subset of the union of the reach sets of any multiple sequences of operators from the same domain. Therefore, if a domain D is simple, then the reach sets of all of its operators are guaranteed to be not subsets of the union of the reach sets of any multiple sequences of operators from D. Hence, the remaining operators in a simple domain after applying the method described in Section 4.6.1 are all primitive. Our approach to validate the functional equivalence of simple planning domain models with only primitive operators is described in Section 4.8.

On the other hand, if the domain is complex, then we do not have to remove all non-primitive operators as our method to validate the functional equivalence of complex domains, explained in Section 4.9, does not depend on the condition that the operators of the two domains being primitive.

Next section explains how to check whether a given domain is simple or complex.

4.7 Types of Planning Domain Models

According to Definition 4.6, functionally equivalent planning domain models must have equal reach sets for any set of objects. Hence, we can prove or disprove the functional equivalence of two planning domain models by comparing their reach sets. We differentiate between two types of planning domain models; complex and simple. We say a domain is complex if it has at least one operator that has the same delete effects as another operator or a sequence of operators from the same domain. On the other hand, a domain is said to be simple if none of its operators have the same delete effects as other operators or sequences of operators in the same domain.

According to our classification of simple and complex domains, complex domains are more common than simple ones in the literature. Out of the 13 IPC domains used in the empirical evaluation of our method in Section 4.14, we found that only one domain is simple, the gripper domains, while all other domains are complex. For instance the Elevator domain is complex because it has two or more operators that share the same delete effects. The operator “up” presented in Listing 4.2 and the operator “down” depicted in Listing 4.3 from the Elevator domain both have (not (lift-at ?f1)) as their delete effect.

[1] This statement is supported by Theorem 1 if we substitute the operator o by the placeholder o_2 and the SOSOs of o by the placeholder o_1 in the theorem.

```
1 (:action up
2  :parameters (?f1 - floor ?f2 - floor)
3  :precondition (and (lift-at ?f1) (above ?f1 ?f2))
4  :effect (and (lift-at ?f2) (not (lift-at ?f1))))
```

Listing 4.2: The operator "up" from the Elevator domain.

```
1 (:action down
2  :parameters (?f1 - floor ?f2 - floor)
3  :precondition (and (lift-at ?f1) (above ?f2 ?f1))
4  :effect (and (lift-at ?f2) (not (lift-at ?f1))))
```

Listing 4.3: The operator "down" from the Elevator domain.

Note that the condition of classifying simple and complex domains is introduced to facilitate the prove of Lemma 4.1, which is necessary to prove Theorem 2. This theorem is the base of our method to validate the function equivalence of simple planning domain models as explained in Section 4.8. We note that currently most of the examined domains are classified as complex domains because our domain classification condition is over conservative. For example, relaxing the definition of complex domains to be only domains with two or more operators with similar add and delete effects makes all investigated domains simple apart from Elevator, Floor-tile, Child-snack and Pipesworld. However, such relaxation of this condition breaks the proof of Lemma 4.1. As a future work, we need to find a new proof of Lemma 4.1 that depends on a weaker condition that classifies most of the domains as simple domains.

4.7.1 Check Planning Domain Models Type

Each set of operators that have the same delete effects forms an equivalence class. We call such equivalence classes the sets of operators with the Same Delete Effects (*SDE*). We consider the delete effects similarity relation to be not reflexive; hence, an *SDE* set cannot have a single operator. Therefore, an *SDE* set exists in a domain if this domain has at least two operators with the same delete effects. So, if a domain has at least one *SDE*, then this domain is complex.

To simplify the implementation of D-VAL, we assume domains do not have predicates that share the same name but have different arities or different types.

To test if a domain is simple or complex, D-VAL creates a unique identifier for each operator from the given domain. The unique identifier of an operator o is the set of the predicates of the predicates of the delete effects of the operator o. After that, D-VAL groups operators of similar identifiers, if they are two or more, into an *SDE* set. If a domain has at least one *SDE* set, then it is complex because two or more operators have the same delete effects. For instance, the identifier of the operator Up is (lift-at), and for the operator Down is also (lift-at). Therefore, these two operators are grouped in one *SDE* set. Hence, the Elevator domain is considered complex planning domain model.

On the other hand, if the domain does not have any *SDE* set, then we still cannot judge whether the domain is simple or complex; we have to check if any operator has the same delete effects as a sequence of operators to confirm the type of the domain.

To check if a given operator has the same delete effects as a sequence of operators from the same domain, we modify the method of checking if the reach set of an operator is a subset of the reach set of a sequence of operators, which is explained in Section 4.6.1. In Section 4.6.1, we had to find a sequence of operators such that its consolidation has the same effects as o and the preconditions of the consolidation of this sequence is a subset of the preconditions of o. But, to check if an operator o has the same delete effects a sequence of operators, we only have to search for a sequence of operators such that its consolidation has the same delete effects as o. This search problem is also formalises as a meta-planning problem.

The specific initial state s_{init}, in this search problem, is a state where the truth evaluations of all of the atoms of all operators in D_m are true. For a state s_g to be in the set S_g of this search problem, the truth evaluations of the atoms of the delete effects of o must be false in s_g. This requirement excludes any sequence of operators with different delete effects than o. Unlike, the meta-planning problem introduced in Section 4.6.1, all the operators from the domain D can be part of a sequence of operators that has the same delete effect like the operator o. Thus, D_m must have all the operators of the domain D apart from o in this meta-planning problem. To enable this search, enough number of dummy objects must be instantiated to avoid excluding any operator. Thus, this meta planning problem must have a number of dummy objects form a type t that is equal to the maximum number of parameters of this type in all operators.

If a plan is found for this meta-planning problem, we can conclude that the operator o has the same delete effect as the sequence of operators that can be lifted from the plan. Therefore, the domain that contains the operator o is complex. On the other hand, if no plan is found, we can conclude that the operator o does not have the same delete effects as any sequence of operators from the same domain. Once all operators of a given domain, that does not have any *SDE* , are confirmed to not have the same delete effects as any sequence of operators, then the domain is proved to be simple.

When testing if a domain D is simple or complex, we first check if the domain D has any *SDE* . If the domain D has an *SDE* , then D is confirmed to be complex and we do not have to continue the more computationally costly process of checking if any operator has the same delete effects as a sequence of operators.

4.8 Roadmap for Validating the Functional Equivalence of Simple Planning Domain Models

This section presents the logical steps required to prove and disprove the functional equivalence of simple planning domain models with only primitive operators. This explanation introduces our approach and the theorems that support the soundness and completeness of our method.

Assume two functionally equivalent planning domain models D_1 and D_2, and a bijective function $F_p : P_1 \rightarrowtail\!\!\!\!\!\rightarrow P_2$. Let $F_p(D_1)$ be the image of D_1 using f to substitute its predicates with those of D_2 with

equal arity. To prove that D_1 and D_2 are functionally equivalent, we have to prove that the reach sets of $F_p(D_1)$ and D_2 are equal for any set of objects, as per Definition 4.6. Of course, we could also prove the functional equivalence between D_1 and D_2 by proving that the reach sets of D_1 and $F_p^{-1}(D_2)$ are equal for any set of objects because the functional equivalence relation is an equivalence relation as per Definition 4.6. However, in this chapter, we will focus on proving that the reach sets of $F_p(D_1)$ and D_2 are equal for any set of objects.

So, the question is how to prove $\Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj)$. To address this question, we have taken a divide-and-conquer approach. We have reduced the relation between the reach sets of two domains to relations between their operators. This simplification is captured in Theorem 2 in Section 4.8.1 and proven in Appendix A.2. This theorem is based on the observation that the reach set of any simple domain is produced exclusively by its primitive operators. Thus, for two planning domain models D_1 and D_2 to have equal reach sets for any set of objects, this theorem requests the image of the reach set of each primitive operator o in D_1 to be equal to the reach set of a primitive operator o' from D_2 under a bijective predicate mapping f_p from the predicates of o to the predicates of o' for any set of objects.

By definition, primitive operators in one domain have different reach sets. Therefore, under a predicate mapping $F_p : P_1 \twoheadrightarrow P_2$, the images of the reach sets of any two primitive operators in D_1 cannot be equal to the reach set of a single primitive operator in D_2. It follows that if D_1 and D_2 have an unequal number of primitive operators, then the image of the reach set of each primitive operator in one domain cannot be equal to the reach sets of a primitive operator in the other domain. Hence, for D_1 and D_2 to have an equal reach set, Theorem 2 also mandates that the two domains must have an equal number of primitive operators.

Note that the scope of the mapping f_p that makes the reach set of a primitive operator from D_1 equal to the reach set of a primitive operator from D_2 is limited to the predicates of these two operators. However, for D_1 and D_2 to be functionally equivalent, Definition 4.6 requires the existence of an overall predicate mapping F_p from the predicates of D_1 to the predicates of D_2 such that the reach set of $F_p(D_1)$ is equal to the reach set of D_2. Therefore, Theorem 2 dictates that the union of the individual predicate mappings f_p between the predicates of the primitive operators of the two domains must be a bijective function.

According to Theorem 2, to prove $\Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj)$, we have to prove the following three conditions.

First condition *D_1 and D_2 must have an equal number of primitive operators.*

To prove this condition, we must compute the set of primitive operators in each domain and show that these two sets have equal cardinality. This task is explained in the previous section (Section 4.6).

Second condition *The reach set of an image of each primitive operator o in D_1 must be equal to the reach set of a primitive operator o' in D_2.*

The image of o is produced using a predicate mapping f_p that substitutes the predicates of o with predicates from o' with equal arity. To prove that the reach set of an image of one operator is equal to the reach set of a primitive operator from another domain without reasoning about individual elements of the

reach sets, we have to compare the structures of operators. Theorem 3, formalised in Section 4.8.1 and proven in Appendix A.3, relates the equality of the reach sets of two operators o and o' to the existence of a mapping between the atoms of the two operators, such that this mapping satisfies a set of constraints that guarantees the two operators have the same structure.

To satisfy the second condition of Theorem 2, we have to find one atom mapping from the atoms of each primitive operator in D_1 to the atoms of a primitive operator in D_2, such that each of these atom mappings satisfies the constraints of Theorem 3. The existence of such atom mapping from the atoms of an operator o from D_1 to the atoms of an operator o' from D_2 proves the existence of a bijective predicate mapping f_p from the predicates of o to the predicates of o' such that the reach set of $f_p(o)$ is equal to the reach set of o'. The task of finding such mappings is explained in Section 4.10.1 and Section 4.10.2.

Third condition *The individual mappings f_p from the predicates of each primitive operator in D_1 to the predicates of each primitive operator in D_2 must be consistent.*

Individual mappings are considered consistent if the union of these mappings is a bijective function. This condition can be satisfied by imposing a predicate consistency constraint that ensures the atoms from D_1 with the same predicate p are mapped to atoms in D_2 with one predicate p'. This constraint differs from the constraints of Theorem 3 by the scope. The scope of the constraints of Theorem 3 is limited to individual operators. On the other hand, the scope of this constraint includes all atoms in all operators of both domains. This consistency constraint is explained in Section 4.11.2 and formalised in Equation (4.28).

We have shown that proving the functional equivalence of functionally equivalent domains depends on satisfying the premises of Theorem 3 and Theorem 2. Since these theorems are proven as logical equivalences, we can safely infer that two planning domain models are not functionally equivalent if these domains do not satisfy any of the antecedents of these theorems.

A flowchart that illustrates the proposed logical steps to prove the functional equivalence of two simple planning domain models is depicted in Figure 4.2.

This section explained the required logical steps to prove or disprove the functional equivalence of simple planning domain models. The following section lists the theorems that form the theoretical foundation of our method.

4.8.1 Simple Domains Theorems

4.8.1.1 Simple Domains Reachability Theorem

The Domain Reachability Theorem reduces the relation between the reach sets of two simple domains, D_1 and D_2, to relations between their primitive operators. To prove that the reach set of an image of D_1 is equal to the reach set of D_2, we have to prove that the reach set of an image of each primitive operator o in D_1 is equal to the reach set of a primitive operator o' in D_2, under a bijective mapping from the predicates of o to the predicates of o'; both domains have an equal number of primitive operators, and all the mappings from the predicates of each primitive operator in D_1 to the predicates of each

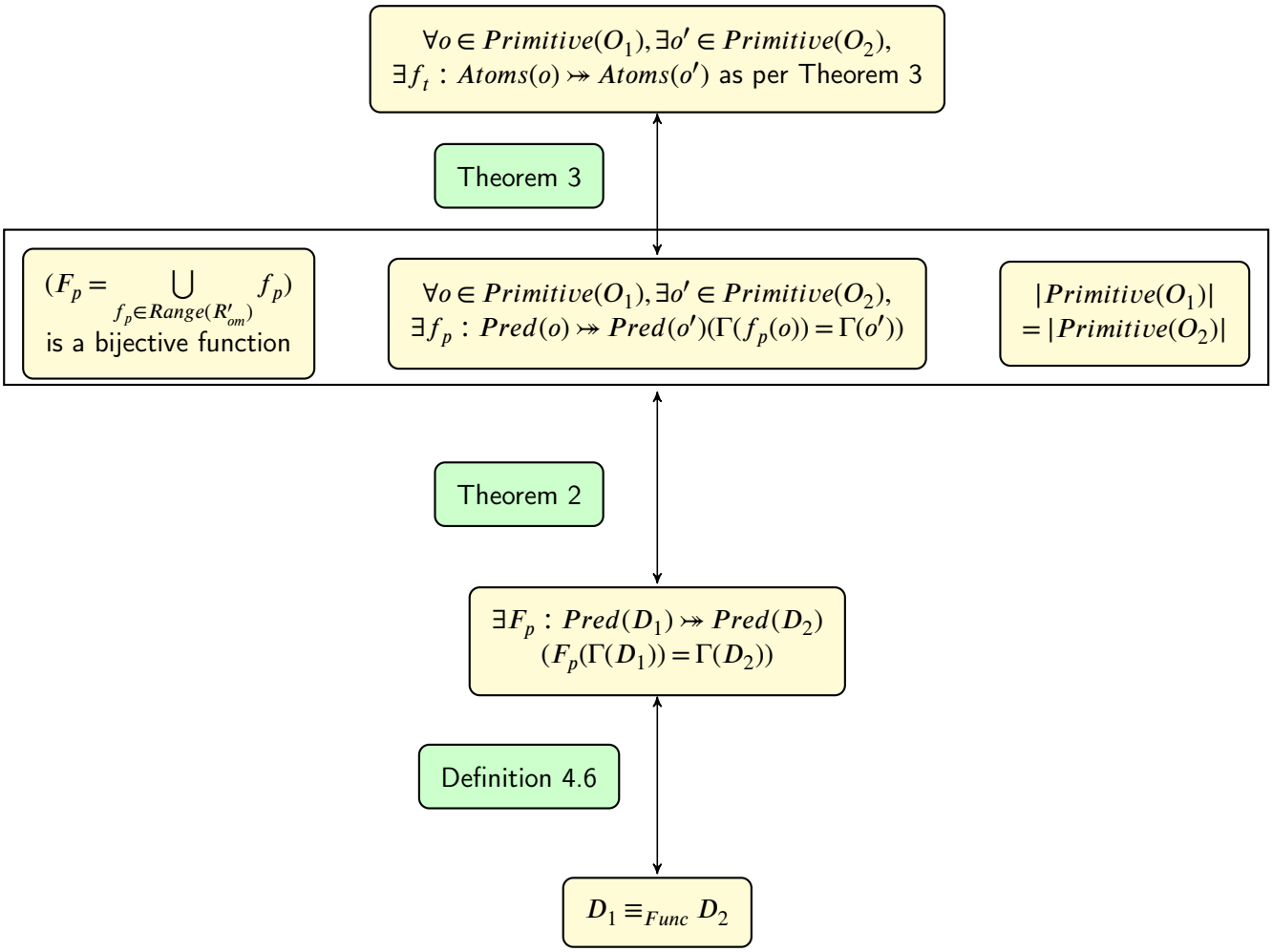


Figure 4.2: The road map to prove the functional equivalence of two simple planning domain models.

primitive operator in D_2 are consistent. We define the following two additional concepts to simplify the explanation of this theorem.

Let $\mathbb{F}$ be the set of all bijective mappings between predicates of equal arity from the predicates of every primitive operator in D_1 to the predicates of every primitive operator in D_2.

$$\mathbb{F} = \{f_p | f_p : Predicates(o) \twoheadrightarrow Predicates(o') \text{ where } o \in Primitive(O_1), o' \in Primitive(O_2), \text{ and if } f_p(p) = p' \text{ then } Arity(p) = Arity(p')\}$$

Let R_{OM} be a relation between primitive operators from D_1 and predicate mappings from $\mathbb{F}$. A primitive operator o from D_1 is related to a mapping f_p from $\mathbb{F}$ by R_{OM} if there exists a primitive operator o' from D_2 such that the reach set of $f_p(o)$ is equal to the reach set of o'.

$$R_{OM} = \{(o, f_p) \in Primitive(O_1) \times \mathbb{F} \mid \exists o' \in Primitive(O_2),\ \Gamma(f_p(o), Obj) = \Gamma(o', Obj)\}$$

The reachability theorem of simple domains is formalised as follows.

Theorem 2 (Simple Domains Reachability Theorem)**.** *Consider a set of objects Obj, two simple planning domain models, D_1 and D_2, a bijective function F_p from the predicates of D_1 to the predicates of D_2 with equal arities, and the relation R_{OM} that relates each primitive operator o in O_1 to a bijective predicate mapping f_p that makes the reach set of $f_p(o)$ equals to the reach set of a primitive operator from O_2. We have:*

$$\Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj) \textit{ iff } \exists R'_{om} \subseteq R_{OM}(\text{Domain}(R'_{om}) = primitive(O_1) \wedge (F = \bigcup_{f_p \in Range(R'_{om})} f_p) \textit{ is a bijective function} \wedge |Primitive(O_1)| = |Primitive(O_2)|) \tag{4.3}$$

To simplify the proof of this theorem, we formalise the following lemma, which states that the reach set of D_1 is equal to the reach set of D_2 if and only if the reach set of every primitive operator in D_1 is equal to the reach set of a primitive operator in D_2 and if the number of primitive operators in D_1 is equal to the number of primitive operators in D_2.

Lemma 4.1 (Simple Domain Reachability Lemma)**.** *Consider a set of objects Obj, two simple planning domain models, D_1 and D_2, and a bijective function $F_p : P_1 \twoheadrightarrow P_2$. We have*

$$\forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(\Gamma(F_p(o), Obj) = \Gamma(o', Obj) \wedge |Primitive(O_1)| = |Primitive(O_2)|) \iff \Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj) \tag{4.4}$$

Theorem 2 is proven in Appendix A.2 with the help of Lemma 4.1. The proofs of this lemma is provided in Appendix A.2.1.

4.8.1.2 Operators Structure Reach Set Theorem

Operators Structure Reach Set Theorem relates the equality of the reach sets of two operators to the condition that both should have the same structure. This condition is represented by the existence of a special mapping between the atoms of the two operators. This theorem is fundamental as it enables us to evaluate the equality of the reach sets of two operators without enumerating the elements of these two sets. Note that it is impossible to enumerate the reach sets of two operators for any set of objects, as there are infinitely many sets of objects. This theorem is formalised as follows.

Theorem 3 (Operators Structure Reach Set Theorem)**.** *Consider a set of objects Obj, two operators o and o', and a bijective function $f_p : Predicates(o) \twoheadrightarrow Predicates(o')$ such that f_p maps the predicates of o to those of o' with equal arities. Let $f_p(o)$ be the image of o using f_p to substitute the predicates of o with those of o' with equal arities. For any set of objects, the reach set of $f_p(o)$ is equal to the reach set of o' iff there exists a bijective mapping f_t from the atoms of o to the atoms of o' such that f_t and f_t^{-1} satisfy the following conditions:*

1. *Atoms in the preconditions, delete effects and add effects of one operator must be mapped to atoms in the preconditions, delete effects and add effects of the other operator, respectively;*

2. *Atoms in one operator must be mapped to atoms in the other operator with equal arity;*

3. *Atoms with the same predicate p in one operator must be mapped to atoms with some predicate p' in the other operator; and*

4. *Atoms with a shared variable v in one operator must be mapped to atoms with some shared variable v' in the other operator such that the positions of v and v' in the parameters of the mapped atoms are equal.*

Here we provide the formal form of the theorem. The part related to f_t^{-1} has been omitted to avoid repetition.

$$\forall Obj, \exists f_p : \text{Predicates}(o) \twoheadrightarrow \text{Predicates}(o') \text{ where if } f_p(p) = p' \text{ then } \text{Arity}(p) = \text{Arity}(p') \,($$

$$\Gamma(f_p(o), Obj) = \Gamma(o', Obj) \iff$$

$$\exists f_t : Atoms(o) \twoheadrightarrow Atoms(o') \,($$

$$\forall t \in \text{Pre}(o) \,(\exists t' \in \text{Pre}(o') : f_t(t) = t') \wedge$$

$$\forall t \in \text{Del}(o) \,(\exists t' \in \text{Del}(o') : f_t(t) = t') \wedge$$

$$\forall t \in \text{Add}(o) \,(\exists t' \in \text{Add}(o') : f_t(t) = t') \wedge$$

$$\forall t \in Atoms(o) \,(\exists t' \in Atoms(o') : f_t(t) = t' \wedge Arity(t) = Arity(t')) \wedge$$

$$\forall t_1, t_2 \in Atoms(o) \,(Pred(t_1) = Pred(t_2) \rightarrow \exists t'_1, t'_2 \in Atoms(o') \,(Pred(t'_1) = Pred(t'_2)$$

$$\wedge (f_t(t_1) = t'_1 \wedge f_t(t_2) = t'_2) \vee (f_t(t_2) = t'_1 \wedge f_t(t_1) = t'_2))) \wedge$$

$$\forall t_1, t_2 \in Atoms(o), \forall v_1 \in Var(t_1), \forall v_2 \in Var(t_2) \,(v_1 = v_2 \rightarrow$$

$$\exists t'_1, t'_2 \in Atoms(o'), \exists v'_1 \in Var(t'_1), \exists v'_2 \in Var(t'_2) \,(v'_1 = v'_2$$

$$\wedge ((f_t(t_1) = t'_1 \wedge f_t(t_2) = t'_2 \wedge Position(v_1, t_1) = Position(v'_1, t'_1) \wedge Position(v_2, t_2) = Position(v'_2, t'_2))$$

$$\vee (f_t(t_2) = t'_1 \wedge f_t(t_1) = t'_2 \wedge Position(v_2, t_2) = Position(v'_1, t'_1) \wedge Position(v_1, t_1) = Position(v'_2, t'_2))))))$$

This theorem is proven in Appendix A.3.

We have introduced our approach in the introduction of Section 4.8. Furthermore, we have formalised and proved all required theorems and lemmas in this subsection. In the following subsection, we will provide an overview of our method to validate the functional equivalence of simple domains with only primitive operators.

4.8.2 Validating the Functional Equivalence of Simple Planning Domain Models

To prove the functional equivalence of two simple planning domain models with only primitive operators, we have to show that the reach set of one domain is equal to the reach set of the other domain under a bijective predicate mapping, as per Definition 4.6. The logical steps required to prove that the reach set of an image of a domain D_1 is equal to the reach set of a domain D_2 under a bijective predicate mapping are depicted in Figure 4.2. These logical steps motivate developing this search task:

Search task 4.1. *Consider two simple planning domain models, D_1 and D_2. If D_1 and D_2 have an equal number of primitive operators, find a bijective mapping from the atoms of each primitive operator o in D_1 to the atoms of a primitive operator o' in D_2, such that these atom mappings are consistent with regard to the mapping of their predicates and respect the conditions specified in Theorem 3.*

The first step to proving the functional equivalence of two simple planning domain models, D_1 and D_2, is to check if the two domains have an equal number of primitive operators according to the requirement of Theorem 2. We can check the two domains have an equal number of primitive operators by finding and removing all non-primitive operators from D_1 and D_2. This step is explained in Section 4.6. After removing non-primitive operators from D_1 and D_2, we can check the number of primitive operators in these two domains.

If the two domains do not have an equal number of primitive operators, then, according to Theorem 2, the reach sets of the two domains are not equal under any predicate mapping. Hence, D_1 and D_2 are not functionally equivalent. On the other hand, if the two domains have an equal number of primitive operators, then we can validate the functional equivalence of D_1 and D_2 according to Theorem 2 by finding a mapping F_p from the predicates of D_1 to the predicates of D_2 such that this mapping can make the reach set of each primitive operator from D_1 equals to the reach set of a primitive operator in D_2. The method to find such predicate mapping is explained in Section 4.10.

If the two domains are checked to have an equal number of primitive operators, then the first condition of Theorem 2 (Condition 4.8) is satisfied. Therefore, if we find a mapping F_P from the predicates of D_1 to the predicates of D_2 such that F_p satisfies the second and third conditions of Theorem 2 (Condition 4.8 and Condition 4.8), we prove D_1 and D_2 are functionally equivalent.

This section explained our method for validating the functional equivalence of simple planning domain models with only primitive operators. The following section describes our approach to validate the functional equivalence of complex planning domain models.

4.9 Roadmap for Validating the Functional Equivalence of Complex Planning Domain Models

In this section, we will explain why we have to differentiate between the validation of the functional equivalence of simple and complex domains. Then, we will introduce the required theorems that prove the correctness of our approach for validating the functional equivalence of complex domains. The

method proposed in this section depends on Theorem 4, which is a relaxed version of Theorem 2. Theorem 4, which we will shortly introduce, removes the condition that states the two domains must have an equal number of primitive operators.

Remember, as explained in Section 4.7, complex domains are defined as those domains that have at least two operators with the same effects. Assume a complex domain D_1 which has two operators, o_1 and o_2, and another domain D_2 which has an operator o' such that the reach set of o' is equal to the union of the reach sets of o_1 and o_2 under some predicate mapping F_p and not equal to the reach set of either of these two operators under any predicate mapping. Note that any transition that can be produced by D_1 can also be produced by D_2 under F_p and vice versa. Hence, D_1 and D_2 are functionally equivalent because their reach sets are equal under F_p.

Since D_1 and D_2 have a different number of primitive operators, and the reach set of neither of the primitive operators of any domain is equal to the reach set of a primitive operator from the other domain, these two domains would have been labelled as not functionally equivalent according to the Simple Domains Reachability Theorem (Theorem 2). Note that D_1 is complex, because o_1 and o_2 share the same delete effects as the union of the images of their reach sets is equal to the reach set of o'. Theorem 2 would have failed to produce the correct verdict about the functional equivalence of these two domains because one of them is not simple.

Therefore, we need new theorems and a new method to check the functional equivalence of complex planning domain models. We defer the full investigation of this problem to future research. Nevertheless, we note that we can still leverage our method of validating the functional equivalence of simple domains to validate the functional equivalence of complex domains in some cases. For instance, assume a complex domain, D_1, with three operators, o_1, o_2 and o_3, that share the delete effects, and the reach set of o_3 is equal to the union of the reach sets of o_1 and o_2. Also, suppose another complex domain, D_2, with also three operators such that the reach set of each of the operators of D_2 is equal to the reach set of an operator from D_1. In this case, the reach set of D_1 is equal to the reach set of D_2. Hence the two domains are functionally equivalent.

To address such cases, we propose Theorem 4, which informally says that if the reach set of each operator from a domain D_1 is equal to the reach set of an operator from D_2, then the reach set of D_1 is a subset of the reach set of D_2. This theorem is formalised in Section 4.9.1.1 and proven in Appendix A.4.

Note that the consequence of the implication in Theorem 4 implies only a containment relation rather than an equality relation between the reach sets of planning domain models. Therefore to prove the existence of $F_p : P_1 \rightarrow P_2$ such that $\Gamma(F_p(D_1)) = \Gamma(D_2)$, we need to prove $\Gamma(F_p(D_1)) \subseteq \Gamma(D_2)$ and $\Gamma(D_2) \subseteq \Gamma(F_p(D_1))$. Proving the existence of a bijective mapping $G_p : P_2 \rightarrow P_1$ such that $\Gamma(G_p(D_2)) \subseteq \Gamma(D_1)$ is easier than proving $\Gamma(D_2) \subseteq \Gamma(F_p(D_1))$ because proving $\Gamma(G_p(D_2)) \subseteq \Gamma(D_1)$ is symmetric to proving $\Gamma(F_p(D_1)) \subseteq \Gamma(D_2)$. Furthermore, we found that starting from $\Gamma(F_p(D_1)) \subseteq \Gamma(D_2)$ and $\Gamma(G_p(D_2)) \subseteq \Gamma(D_1)$ we can deduce $\Gamma(D_2) \subseteq \Gamma(F_p(D_1))$. This claim is captured in Theorem 5, which is formalised in Section 4.9.1.2 and proven in Appendix A.5.

We can further simplify the proof of the functional equivalence of complex planning domain models

with the help of Corollary 4.1, which is deduced from Theorem 5. Corollary 4.1 states that if D_1 and D_2 have an equal number of operators and the reach set of either of the domains is a subset of the reach set of the other domain under a predicate mapping, then the reach sets of both domains are equal under that mapping and its inverse mapping as well. Corollary 4.1 dramatically reduces the time required to prove the functional equivalence of two domains by enabling us to search for one mapping rather than two and still reach the same conclusion. This Corollary is formalised in Section 4.9.1.2 and proven in Appendix A.6. The following section lists the theorems used in our approach to validate the functional equivalence of complex domains and points to their proofs.

4.9.1 Complex Domains Theorems

4.9.1.1 Complex Domains Reachability Theorem

Let $\mathbb{F}$ be the set of all bijective mappings from predicates of equal arity from every operator in D_1 to every operator in D_2.

$$\mathbb{F} = \{f_p | f_p : Predicates(o) \twoheadrightarrow Predicates(o') \text{ where } o \in O_1, o' \in O_2 \\ \text{and if } f_p(p) = p' \text{ then } Arity(p) = Arity(p')\}$$

Let R_{OM} be a relation between operators from D_1 and predicate mappings from $\mathbb{F}$. An operator o from D_1 is related to a mapping f_p from $\mathbb{F}$ by R_{OM} if there exists an operator o' from D_2 such that the reach set of $f_p(o)$ is equal to the reach set of o'.

$$R_{OM} = \{(o, f_p) \in O_1 \times \mathbb{F} \mid \exists o' \in O_2,\ \Gamma(f_p(o), Obj) = \Gamma(o', Obj)\}$$

Theorem 4 (Complex Domains Reachability Theorem)**.** *Consider a set of objects Obj, two planning domain models, D_1 and D_2, a bijective function F_p from the predicates of D_1 to the predicates of D_2 with equal arity, and the relation R_{OM} that relates each operator o in O_1 to a bijective predicate mapping f_p that makes the reach set of $f_p(o)$ equals to the reach set of an operator from O_2. We have:*

$$\exists R'_{om} \subseteq R_{OM}(Domain(R'_{om}) = O_1 \wedge \\ (F = \bigcup_{f_p \in Range(R'_{om})} f_p) \text{ is a bijective function}) \implies \Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj)$$

Lemma 4.2 (Complex Domain Reachability Lemma)**.** *Consider a set of objects, Obj, two planning domain models D_1 and D_2, and a bijective function $F_p : P_1 \twoheadrightarrow P_2$. We have*

$$\forall o \in O_1, \exists o' \in O_2(\Gamma(F_p(o), Obj) = \Gamma(o', Obj)) \implies \Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj) \tag{4.5}$$

Theorem 4 is proven in Appendix A.4 with the help of Lemma 4.2. The proof of this lemma is provided in Appendix A.4.1.

4.9.1.2 Reach Sets Containment Theorem and Domain Reach Sets Equality Corollary

In order to prove D_1 and D_2 are functionally equivalent according to Definition 4.6, we have to prove $\Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj)$. The Reach Sets Containment Theorem shows that for two planning domain models, D_1 and D_2, we have $\Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj)$ if and only if $\Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj)$ and $\Gamma(G_p(D_2), Obj) \subseteq \Gamma(D_1, Obj)$. This theorem is formalised as follows.

Theorem 5 (Reach Sets Containment Theorem)**.** *For two planning domain models, D_1 and D_2, and two bijective functions $F_p : P_1 \twoheadrightarrow P_2$ and $G_p : P_2 \twoheadrightarrow P_1$, let $F_p(D_1)$ be the image of D_1 using F_p to substitute its predicates with those of D_2, and $G_p(D_2)$ be the image of D_2 under G_p. Then we have for a set of objects Obj:*

1. $\Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj)$ *and* $\Gamma(G_p(D_2), Obj) \subseteq \Gamma(D_1, Obj) \iff \Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj)$ *and* $F_p = G_p^{-1}$.
2. $\Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj)$ *and* $\Gamma(G_p(D_2), Obj) \subseteq \Gamma(D_1, Obj) \iff \Gamma(G_p(D_2), Obj) = \Gamma(D_1, Obj)$ *and* $G_p = F_p^{-1}$.

This theorem is proven in Appendix A.5

We can use this theorem to prove a useful corollary, The Domain Reach Sets Equality Corollary, which states that for any set of objects, the reach set of a planning domain model D_1 is equal to the reach set of a planning domain model D_2 under a bijective predicate mapping F_p if and only if the reach set of the image of D_1 under F_p is a subset of the reach set of D_2, and the two domains D_1 and D_2 have an equal number of operators. This corollary is formalised as follows:

Corollary 4.1 (Domain Reach Sets Equality corollary)**.** *Consider a set of objects, Obj, two planning domain models, D_1 and D_2, a bijective function F_p from the predicates of D_1 to the predicates of D_2 with equal arity. We have:*

$$\forall o \in O_1, \exists o' \in O_2(\Gamma(F_p(o)) = \Gamma(o')) \wedge |O_1| = |O_2| \implies \Gamma(F_p(D_1) = \Gamma(D_2)$$

This corollary is proven in Appendix A.6.

4.9.2 Validating the Functional Equivalence of Complex Planning Domain Models

Just like the method of proving the functional equivalence of simple domain models with only primitive operators, the first step to validate the functional equivalence of complex planning domain models is to remove non-primitive operators because such operators do not alter the reach set of their domains. Removing non-primitive operators is explained in Section 4.6. After that, we follow the steps in Section 4.10 to find a mapping f_p from the predicates of each operator o from D_1 to the predicates of an operator o' from D_2 such that $\Gamma(f_p(o)) = \Gamma(o')$ and the union of all individual mappings from the operators of D_1 to operators in D_2 is a bijective function F_p. Then, according to Theorem 4, the reach set of D_1 is a subset of the reach set of D_2 under F_p. Then if $|O_1| = |O_2|$, we can conclude that

the reach set of D_1 is equal to the reach set of D_2 under F_p according to Corollary 4.1. Thus, as per Definition 4.6, D_1 and D_2 are functionally equivalent. On the other hand, if $|O_1| \neq |O_2|$ or there is no predicate mapping F_p that makes the reach set of each operator in D_1 equal to the reach set of an operator in D_2, we cannot disprove the functional equivalence of two complex planning domain models because Corollary 4.1 is proven as implication, not as an equivalence. Thus, our method cannot give a conclusive verdict about the functional equivalence of these domains.

This section and Section 4.8 provided an overview of our approaches to validate the functional equivalence of simple and complex domains. In addition, the following sections provide detailed explanations of the methods used to find consistent predicate mappings between the operators of the given domains.

4.10 Finding a Predicate Mapping for Equating the Reach Sets of Two Planning Domain Models

To find a predicate mapping F_p that makes the reach sets of two domains equal to each other, we have to find bijective mappings from the atoms of each operator o in D_1 to the atoms of an operator o' in D_2, such that these atom mappings are consistent with regard to their predicates and satisfy the conditions of Theorem 3. To find such atom mappings, we start our search from the set of atom mappings that satisfy two constraints: (1) map operators with equal numbers of preconditions, add effects and delete effects, i.e. respect the first condition of Theorem 3. (2) map operators with an equal number of variables, i.e. potentially satisfy the fourth condition of this theorem. Finding every operator o' in D_2 with atoms that can be mapped to the atoms of an operator o from D_1 as per the first condition of Theorem 3 can be performed by directly comparing the numbers of atoms in the preconditions, add effects and delete effects of the operators o and o'; basically, o and o' should have equal numbers of preconditions, add effects and delete effects. In addition, o and o' must have the same number of parameters; if these two operators have different numbers of variables, then no mapping between the atoms of o and o' will satisfy the fourth condition of Theorem 3. We chose to start the search with these criteria, which are further explained in Section 4.10.1, because checking if an atom mapping satisfies them is a simple numerical comparison.

To find mappings that satisfy all the conditions of Theorem 3, we narrow down the mappings found in the previous step by enforcing the remaining constraints of this theorem. This search problem is presented in Section 4.10.2 and detailed in Section 4.11. This approach necessitates that for every operator o in D_1, we find every operator o' from D_2 that supports the existence of mappings from the atoms of o to the atoms of o' that satisfy the first condition and part of the requirement of the fourth condition of Theorem 3. The need to find all mappings from the atoms of each operator in D_1 to the atoms of every operator in D_2 is essential, so we do not exclude any potential mappings which can possibly prove the functional equivalence between the two domains.

To ensure the atom mappings found by our method can produce a bijective predicate mapping, we further constrain our search problem to find only atom mappings that are consistent with regard to the

mapping of atoms with shared predicates. This consistency constraint is explained in Section 4.11.2 and formalised in Equation (4.28).

4.10.1 Finding Potentially Functionally Equivalent Operators (PFEOs)

The first condition of Theorem 3 states that *atoms in the preconditions, delete effects and add effects of an operator o from domain D_1 must be mapped to atoms in the preconditions, delete effects and add effects of the operator o' from domain D_2 respectively.* Hence, for a mapping from the atoms of o to the atoms of o' to satisfy the first condition of Theorem 3, the operator o must have numbers of atoms in its precondition, add effects and delete effects that are equal to numbers of atoms in the precondition, add effects and delete effects of the operator o'. These conditions are captured in the following equations.

$$|\text{Pre}(o)| = |\text{Pre}(o')| \tag{4.6}$$

$$|\text{Add}(o)| = |\text{Add}(o')| \tag{4.7}$$

$$|\text{Del}(o)| = |\text{Del}(o')| \tag{4.8}$$

Furthermore, since we are searching for bijective mappings from the atoms of the operator o to the atoms of each operator o' in D_2, o and o' must have an equal number of atoms. Note that the add effects of an operator cannot appear as delete effects or preconditions in any valid operator. Therefore, there is no intersection between the add and delete effects and between the add effects and the preconditions of any operator. Moreover, the delete effects of an operator must be a subset of the preconditions of that operator. Therefore, the set of atoms of an operator is the union of the atoms of its preconditions and the atoms of its add effects. Hence, Equation (4.6) and Equation (4.7) guarantee the existence of a bijective mapping from the atoms of an operator o to the atoms of an operator o'.

A necessary but not sufficient condition for an atom mapping to satisfy the fourth constraint of Theorem 3 is for o and o' to have an equal number of variables.

$$|\text{Variables}(o)| = |\text{Variables}(o')| \tag{4.9}$$

We group the quantities referenced in Equations (4.6) to (4.9) in a tuple which we call the operator signature.

Definition 4.9. The ***signature*** of an operator o, OpSig(o), is defined based on the number of atoms in its different parts.

$$\text{OpSig}(o) = (|\text{Pre}(o)|, |\text{Add}(o)|, |\text{Del}(o)|, |\text{Variables}(o)|) \tag{4.10}$$

The second stage of our method is to find all operators from D_2 that have the same signature as each operator in D_1.

Definition 4.10. We call an operator o in a planning domain model D_2 a ***Potentially Functionally Equivalent Operator (PFEO)*** of an operator o from a planning domain model D_1 if o and o' have the same signature.

As such, the set of PFEOs from a domain D_2 of an operator o is defined as follows.

$$\text{PFEOs(o)} = \{o' : o' \in O_2, \text{OpSig}(o') = \text{OpSig}(o)\} \tag{4.11}$$

Finding the PFEOs from D_2 for each operator o in D_1 provides us with sets of bijective mappings from the atoms o to the atoms of each of its PFEOs from D_2. These mappings are bijective and are guaranteed to satisfy the first condition and part of the requirement of the fourth condition of Theorem 3. Starting from these sets, we can check the other conditions in the second stage.

4.10.2 Finding Consistent Atom Mappings for Satisfying the Conditions of Theorem 3

This section discusses how to find a consistent atom mapping that satisfies the conditions of Theorem 3. In the previous section, we have seen how restricting the mappings of the atoms of the operators of D_1 to atoms of their PFEOs guarantees these mappings are bijective and satisfy the first condition of Theorem 3. So, we need to find a consistent atom mapping that respects the restriction imposed by the PFEOs and satisfies the remaining conditions of Theorem 3. We encode the problem of finding such an atom mapping as an SMT problem. An SMT problem consists of a set of decision variables and a set of constraints. A solution to an SMT problem is an assignment of the decision variables that satisfies the constraints of this SMT problem. The following sections explain the encoding of our SMT problem and its solution.

4.10.2.1 Decision Variables of the Variable Mapping SMT Problem

The first step in encoding the problem of finding a consistent atom mapping that satisfies the conditions of Theorem 3 as an SMT problem is to define the decision variables of this problem. We must analyse our problem carefully to choose appropriate decision variables for our encoding. Note that the fourth condition of Theorem 3 constrains the mapping of atoms based on their variables. Therefore, to capture this condition, we need decision variables that can express constraints at the level of the variables of the atoms of operators. The solution to our SMT problem is a variable mapping. This mapping maps the variables of the atoms of the operators in D_1 to the variables of the atoms of some of the PFEOs from D_2. To guarantee the variable mappings produce consistent atom mappings, we have to add a set of constraints to ensure the variables of one atom in an operator in a domain are mapped to the variables of some atom in an operator in the other domain. With these additional constraints, the variable mappings will be merely another representation of atom mappings.

To map variables of atoms of operators, we have to first agree on how to define unique variables. We define the canonical signature of variables by concatenating the variable name with some features of its atom and its operator as follows.

Definition 4.11. The *variable signature* of a variable consists of a concatenation of the following items:

1. the name of the given variable,
2. the arity of the atom of the given variable,

3. the name of the predicate of the atom of the given variable,
4. the name of the operator of the atom of the given variable,
5. the concatenation of all variables of the atom of the given variable, and
6. the position of the variable in the parameters of the atom of the given variable.

We refer to the signature of a variable v by $\text{VarSig}(v)$, and to reference a part of the signature of the variable v, we subscript $\text{VarSig}(v)$ with the number of the part from the list above list. For example, the signature of the variable ?r in the atom (at ?r ?x) in the operator Move is (?r 2 at move ?r?x 1); the arity of the atom of the variable ?r is $\text{VarSig}_2(at\ ?r\ ?x)$ which equals 2. Two variables are different in a domain if their signatures differ by at least one part. For instance, the four variables of the atoms (at ?r ?x) and (at ?r ?y) in the same operator, Move, are different. The variables with the signatures (?x 2 at move ?r?x 2) and (?y 2 at move ?r?y 2) are clearly different as they differ by the variable symbol. Moreover, the two variables with the signatures (?r 2 at move **?r?x** 1) and (?r 2 at move **?r?y** 1) are also different. They differ by VarSig_5, i.e. item number five of the signature. These two variables are considered different because they are parts of two different atoms. The proposed variable signature provides us with enough information about the variables of the atoms of the operators in order to implement the conditions of Theorem 3 in addition to the condition of consistent mappings. Since our problem is encoded at the level of variables, we call it a variable mapping SMT problem. The following subsection describes the solution to the variable mapping SMT problem.

4.10.2.2 Variable Mapping SMT Problem Solutions

The solution to our variable mapping SMT problem is a bijective function $d : V_1 \rightarrow N$. This function relates the variables of the atoms of D_1 to a set of natural numbers ($N \subset \mathbb{N}$) such that the elements of the set N represent the variables of the atoms of D_2. The natural numbers in this set are linked to the variables of D_2 using the inverse of the bijective function $b : V_2 \rightarrow N$. The composition of functions d and b^{-1} produces a mapping function that maps the variables of the atoms of D_1 to the variables of the atoms of D_2.

$$\forall v \in V_1, \exists v' \in V_2\ (v' = b^{-1} \circ d(v)) \tag{4.12}$$

The function b can be defined arbitrarily. For example, sorting the variables of the atoms of D_2 according to some criteria on the signatures of these variables defines a function b. Actually, we are not sorting the variables of the atoms of D_2, but we are sorting the list of their signatures. In this case, b maps every variable or every variable signature of the atoms of D_2 to the index of this variable signature in the sorted list of the variable signatures. So, function b is the function that returns the index of a variable signature in a sorted list of variable signatures. After we have defined function b, we need to find function d in order to define the mapping from the variables of the atoms of D_1 to the variables of the atoms of D_2. The function d represents a relation between the variables of the atoms of D_1 and the natural numbers in the domain of function b. This relation can be described with the help of the decision variables that are

related to the variables of the atoms of D_1. The value of a decision variable is a member of the set of the natural numbers that forms the range of d, which is also the domain of b. As such, every assignment to a decision variable forms a map from a variable of an atom of D_1 to a variable of an atom of D_2.

In this section, we have introduced the variable mapping SMT problem. The following section further explains this problem and lists the SMT constraints that capture the conditions of Theorem 3 in addition to the condition of consistent predicate mapping.

4.11 Variable Mapping SMT Problem

The variable mapping SMT problem consists of a set of decision variables and a set of constraints on these decision variables. Each decision variable is associated with a variable from the atoms of the operators of D_1. A solution to this problem is an assignment of the decision variables that satisfies the problem's constraints. To solve this problem, we sort the variables of the atoms of the operators of D_2. Then, we define the function b to capture the order of the sorted variables. After that, we compile the set of constraints that captures the atom mapping consistency criterion and the conditions of Theorem 3. The compilation of these constraints is explained in the following sections. Then we task the Z3 SMT solver [60] to solve our variable mapping SMT problem. The solution found by the SMT solver assigns a natural number to each decision variable $d(v)$. Thus, it defines the function d. Then a mapping from the variables of D_1 to the variables of D_2 is extracted from the functions b and d with the help of Equation (4.12). For each variable v in D_1, $d(v)$ points through b^{-1} to a variable v' in the atoms of D_2. In other words, the function d maps v to v'. For example, suppose $b(v_5^{'}) = 5$ and the value of $d(v_1)$ is found to be equal to 5, then the variable v_1 in D_1 is mapped to the variable $v_5^{'}$ in D_2.

To formalise the constraints implied by the third and fourth conditions of Theorem 3, we need the variables of D_2 to be sorted in two specific orders, one order for each condition. On the other hand, the way we order the variables of D_2 is irrelevant for compiling the first two conditions of Theorem 3 as long as the variables are sorted in a known way. Therefore, we will discuss the SMT constraints of the first two conditions before we explain the idea of the ordered variables when we introduce the constraints of the third and fourth conditions.

The following three subsections explain the SMT constraints that capture the conditions of Theorem 3.

4.11.1 SMT Constraints of the First, and Second Conditions and the Second Part of the Fourth Condition of Theorem 3

The first condition of Theorem 3 requests preconditions, add effects and delete effects of o to be mapped to their counterparts in o'. This condition is captured by contrasting the decision variables of the variables of the preconditions of o to the values of the variables of the preconditions of o'. The same constraints are defined for the add and delete effects.

The second condition of Theorem 3 mandates that only atoms of the same arity are mapped to each other. This condition is satisfied by requesting the second parts of the signatures (VarSig_2) of mapped

variables to be equal to each other, i.e. any mapped variables must belong to atoms of equal arity. For instance, for v to be mapped to v', then $\text{VarSig}_2(v)$ must be equal to $\text{VarSig}_2(v')$.

The fourth condition of Theorem 3 necessitates the positions of mapped variables in the parameters of their respective atoms to be equal. We can impose this requirement of the fourth condition by requesting the sixth part of the signatures (VarSig_6) of the mapped variables to be equal to each other, i.e. any mapped variables must be in the same positions in the parameters of their respective atoms. For example, for v to be mapped to v', then $\text{VarSig}_6(v)$ must be equal to $\text{VarSig}_6(v')$. The second condition and the second part of the fourth condition are simple. We can use explicit equality constraints on the arity of the atoms of the variables and the position of the variables in the parameters of their atoms because these properties are natural numbers.

Our constraints are defined using nested loops as follows. For each operator o in D_1, for each operator $o' \in \text{PFEO}(o)$, for each variable in the atoms of the preconditions, add effects and delete effects of o, for each variable in the atoms of the preconditions, add effects and delete effects of o', we will add a constraint. These constraints are explained in the following subsections.

4.11.1.1 Operator-Level Grouping of Decision Variable Constraints

We need one set of constraints to map the variables of the atoms in each part of the operators: preconditions, add effects and delete effects. These sets are called PreConst, AddConst, and DelConst, respectively. PreConst is a set of constraints that constrain the assignment of the decision variable of a given variable v from an atom t in the preconditions of an operator o to the indices of the variables in the preconditions of an operator o'. AddConst and DelConst are defined similarly. We represent these sets with $\text{PreConst}(v, o')$, $\text{AddConst}(v, o')$, and $\text{DelConst}(v, o')$. These sets are formally defined as follows.

$$\begin{aligned}(4.13)\qquad \text{PreConst}(v,o') = \{(d(v) == b(v')) \mid \forall p' \in \text{Pre}(o'),\\ \forall v' \in \text{Var}(p')(\text{VarSig}_2(v) = \text{VarSig}_2(v') \wedge \text{VarSig}_6(v) = \text{VarSig}_6(v'))\}\end{aligned}$$

$$\begin{aligned}(4.14)\qquad \text{AddConst}(v,o') = \{(d(v) == b(v')) \mid \forall p' \in \text{Add}(o'),\\ \forall v' \in \text{Var}(p')(\text{VarSig}_2(v) = \text{VarSig}_2(v') \wedge \text{VarSig}_6(v) = \text{VarSig}_6(v'))\}\end{aligned}$$

$$\begin{aligned}(4.15)\qquad \text{DelConst}(v,o') = \{(d(v) == b(v')) \mid \forall p' \in \text{Del}(o'),\\ \forall v' \in \text{Var}(p')(\text{VarSig}_2(v) = \text{VarSig}_2(v') \wedge \text{VarSig}_6(v) = \text{VarSig}_6(v'))\}\end{aligned}$$

Note that the scope of v and o' in the definition of the above sets are not defined yet. These scopes will be defined as we explain the top levels of our constraints. The main idea is for $\text{PreConst}(v, o')$, v will be quantified over all variables of all preconditions of an operator o, and o' is quantified over all PFEOs of the operator o. In its turn, the operator o will be quantified over all the operators of the domain D_1. Another important note is that the constraints that will be made from $\text{PreConst}(v, o')$ do not just constrain the mapping of a variable v to the variables of the preconditions of o' but also impose

the second condition and the second part of the fourth condition of Theorem 3. Of course, the above notes apply to $\text{AddConst}(v, o')$ and $\text{DelConst}(v, o')$ as well.

If the reach set of o' is equal to the reach set of the operator o that has a variable v in one of its preconditions, then one member of the $\text{PreConst}(v, o')$ must be true. To achieve this requirement, we need a disjunction of the members of $\text{PreConst}(v, o')$ as represented in the following equation; let us call this disjunction $\text{PreConst}_{\text{or}}(v, o')$.

$$\text{PreConst}_{\text{or}}(v, o') = \bigvee_{\forall \text{const} \in \text{PreConst}(v, o')} \text{const} \tag{4.16}$$

Similarly, we call the disjunctions of the add effects $\text{AddConst}_{\text{or}}(v, o')$ and delete effects $\text{DelConst}_{\text{or}}(v, o')$. These disjunctions are expressed in Equation (4.17) and Equation (4.18).

$$\text{AddConst}_{\text{or}}(v, o') = \bigvee_{\forall \text{const} \in \text{AddConst}(v, o')} \text{const} \tag{4.17}$$

$$\text{DelConst}_{\text{or}}(v, o') = \bigvee_{\forall \text{const} \in \text{DelConst}(v, o')} \text{const} \tag{4.18}$$

We know the disjunction $\text{PreConst}_{\text{or}}(v, o')$ must be used to constrain the mapping of variables from the preconditions of an operator o. As such, the variable v in $\text{PreConst}_{\text{or}}(v, o')$ must be quantified over all variables of the preconditions of an operator o. This will produce a disjunction $\text{PreConst}_{\text{or}}(v, o')$ for every variable v in the preconditions of an operator o. These disjunctions must be combined in a conjunction to enforce the decision variables of every variable in the preconditions of o to be assigned to a value. We call this conjunction $\text{PreConst}_{\text{and}}(o, o')$.

$$\text{PreConst}_{\text{and}}(o, o') = \bigwedge_{\forall p \in \text{Pre}(o), \forall v \in \text{Var}(p)} \text{PreConst}_{\text{or}}(v, o') \tag{4.19}$$

Similar to the conjunction $\text{PreConst}_{\text{and}}(o, o')$, we call the conjunction of the disjunctions of the constraints of the add effects $\text{AddConst}_{\text{and}}(o, o')$ and the conjunction of the disjunctions of the delete effects $\text{DelConst}_{\text{and}}(o, o')$. These conjunctions are expressed in Equation (4.20) and Equation (4.21).

$$\text{AddConst}_{\text{and}}(o, o') = \bigwedge_{\forall p \in \text{Pre}(o), \forall v \in \text{Var}(p)} \text{AddConst}_{\text{or}}(v, o') \tag{4.20}$$

$$\text{DelConst}_{\text{and}}(o, o') = \bigwedge_{\forall p \in \text{Pre}(o), \forall v \in \text{Var}(p)} \text{DelConst}_{\text{or}}(v, o') \tag{4.21}$$

Note that the scope of v is now defined. Also, note that the scopes of the operators o and o' are still not defined yet. These scopes will be defined shortly as we progress in explaining our constraints.

4.11.1.2 Bijective Mapping Constraints

We are looking for a bijective mapping between the atoms of the two domains. Hence, each variable must be mapped to just one variable from the other domain. However, our constraints, which are conjunctions of disjunctions of the assignments of decision variables to natural numbers, can be satisfied by assigning two design variables with the same value. For example, assume an operator o that has a precondition with three variables. Let the decision variables $d(v_1), d(v_2),$ and $d(v_3)$ to be associated with the variables of the precondition of o. Suppose an operator o' is one of the PFEOs of o. Let o' has a precondition with three variables that are associated with the numerical values (1, 2, and 3). Then we have:

$$\begin{aligned}\text{PreConst}_{\text{and}}(o, o') = \{&((d(v_1) == 1) \vee (d(v_1) == 2) \vee (d(v_1) == 3)) \wedge \\ &((d(v_2) == 1) \vee (d(v_2) == 2) \vee (d(v_2) == 3)) \wedge \\ &((d(v_3) == 1) \vee (d(v_3) == 2) \vee (d(v_3) == 3))\}\end{aligned}$$

Then, the assignments $d(v_1) == 1$, $d(v_2) == 1$, and $d(v_3) == 1$ satisfy the conjunction $\text{PreConst}_{\text{and}}(o, o')$. Such cases should be avoided to ensure the produced mapping is bijective. Thus, we need to add more constraints to guarantee that the values of the decision variables are unique. This requirement is also applicable to the decision variables for the variables in the add and delete effects. As such, the uniqueness constraints are defined for the decision variables of all variables as follows.

$$\text{UniqeVariables}(O_1) = \{\text{Distinct}(d(v)) : \forall v \in \text{Variables}(O_1)\} \tag{4.22}$$

Where the constraint *Distinct* is defined as follows.

$$\forall v, v' \in V \ (\text{Distinct}(d(v)) \rightarrow d(v) \neq d(v')) \tag{4.23}$$

4.11.1.3 PFEOs-Level Grouping of Decision Variable Constraints

Each of the previous conjunctions $\text{PreConst}_{\text{and}}(o, o')$, $\text{AddConst}_{\text{and}}(o, o')$, and $\text{DelConst}_{\text{and}}(o, o')$ imposes partial mapping from the variables of an operator o to the variables of its PFEO o'. The conjunction of these conjunctions demands all decision variables of the variables in the preconditions, add effects and delete effects of o to be assigned to the indices of variables in the preconditions, add effects and delete effects of o'. Let us call the conjunction of these conjunctions $\text{OMConst}(o, o')$.

$$\text{OMConst}(o, o') = \text{PreConst}_{\text{and}}(o, o') \wedge \text{AddConst}_{\text{and}}(o, o') \wedge \text{DelConst}_{\text{and}}(o, o') \tag{4.24}$$

We know the variables of the operator o must be mapped to the variables of one of its PFEOs. This requirement is expressed by the disjunction of $\text{OMConst}(o, o')$ for every PFEO of o . We call this disjunction $\text{OConst}(o)$.

$$\text{OConst}(o) = \bigvee_{\forall o' \in \text{PFEO}(o)} \text{OMConst}(o, o') \tag{4.25}$$

4.11.1.4 Domain-Level Grouping of Decision Variable Constraints

Now that we have defined the scope of o', we are left with defining the scope of o. For the reach set of D_1 to be a subset of the reach set of D_2 under a predicate bijective mapping, we need to find a bijective mapping from the atoms of each operator in D_1 to the atoms of an operator in D_2 as per the logical steps depicted in Figure 4.2. Therefore, the variables of every operator o from the operators of the domain D_1 must be mapped to the variables of one of its PFEOs. This demand is expressed by the conjunction of $\text{OConst}(o)$ for every o of O_1. We call this conjunction $\text{DConst}(O_1)$.

$$\text{DConst}(O_1) = \bigwedge_{\forall o \in O_1} \text{OConst}(o) \tag{4.26}$$

4.11.1.5 Range Constraints for Decision Variables

The domain of the function d is $N \subset \mathbb{N}$ where $N = |\,\text{variables}(O_1)|$. Therefore, to ensure the values of the decision variables of D_1 are within the range of the function d, we need the following set of constraints.

$$\text{VRange}(O_1) = \{(1 \leq d(v) \leq |\,\text{Variables}(O_1))| : \forall v \in \text{Variables}(O_1)\} \tag{4.27}$$

4.11.2 SMT Constraints of the Third Condition of Theorem 3

The third condition of Theorem 3 states that the atoms that share the same predicate in one domain must be mapped to atoms with one predicate in the other domain. In our variable mapping SMT problem, we need constraints to guarantee that the variables that belong to atoms with a shared predicate are mapped to variables of atoms that share one predicate in the other domain. This type of constraint relates the assignments of two variables to each other. To compile such constraints, we have to sort the list of variable signatures of the variables of D_2 according to the predicate name part of the variable signature (Sig_3). Then, we will have the variables of D_2 that belong to the atoms that share the same predicate in adjacent places in the sorted list of variable signatures. We have to ensure that the decision variables of the variables of the atoms of D_1 that have the same predicate are assigned to the indices of variables of atoms of D_2, which share one predicate. Therefore, we have to constrain the assignments of these decision variables such that the difference between the values of these decision variables is less than a specific value. We will call this specific value the spacing value (Sv). In other words, we have to ensure that the decision variables of the variables of D_1 that belong to atoms that share the same predicate are assigned to adjacent indices in the list of sorted variable signatures of the atoms of D_2.

So far, we have arranged the variable signatures so that the signature of the variables of the atoms that share the same predicates are adjacent. However, we also have to ensure the signatures of the variables that belong to atoms with different predicates are not adjacent, i.e. separated. Therefore, we have to create buffers between the groups of the signatures of the variables that belong to atoms with different predicates. The size of these buffers is the spacing value Sv. We propose the size of these buffers to be greater than the cardinality of the largest group of the variables of D_2 that belong to atoms with one predicate.

We refer to the spacing value of a list of signatures of variables that are sorted and spaced according to the predicate name of the atoms of these variables as Sv_p. To ensure the decision variables associated with the variables of atoms with the same predicate in D_1 are mapped to variables in D_2 that belong to atoms with one predicate from the predicates of D_2, the values of the decision variables of two variables in D_1 that belong to atoms with the same predicate must not differ by more than the spacing value Sv_p. We call the set of constraints that captures these requirements the Same Predicate Constraints, $\mathrm{SPC}(O_1)$, which is formalised as follows.

$$\mathrm{SPC}(O_1) = \{(|d(v_1) - d(v_2)| \leq \mathrm{Sv}_p) : \forall v_1, v_2 \in \mathrm{Variables}(O_1)(\mathrm{VarSig}_3(v_1) = \mathrm{VarSig}_3(v_2))\} \tag{4.28}$$

4.11.3 SMT Constraints of the First part of the Fourth Condition of Theorem 3

The fourth condition of Theorem 3 states that the atoms that share the same variable v in one domain must be mapped to atoms with one variable v', and these mapped atoms must have v and v' in the same position in the parameters of their atoms. The constraints that imply the second part of this condition are compiled along with the constraints of the first and second conditions in Section 4.11.1. In this section, we will explain the compilation of the first part of the fourth condition into SMT constraints as per our SMT problem representation.

Similar to the constraints of the third condition, the constraints of the fourth condition also relate the assignments of two variables to each other. To compile such constraints, we also have to sort the list of the variable signatures of the variables of D_2. However, this time we will sort this list differently. The fourth condition is concerned with the variables of the atoms of the two domains. Therefore, the constraints that will capture this condition will have to deal with the variable names. Thus, we sort the list of the signature of the variables of D_2 according to the variable name part of the variable signature (Sig_1). Then we will have the variables of D_2 that belong to the atoms that share the same variable in adjacent places in the sorted list of the variable signatures. We have to ensure the decision variables of the variables of the atoms of D_1 that have the same variables are assigned to the indices of variables of atoms of D_2 that one variable. Therefore, we have to constrain the assignment of these decision variables such that the difference between the values of these decision variables is less than a specific value. Similar to the third condition, we will call this specific value the spacing value Sv. In other words, we have to ensure that the decision variables of the variables of D_1 that belong to atoms that share the same variable are assigned to adjacent indices in the list of sorted variable signatures of the atoms of D_2.

So far, we have arranged the variable signatures so that the signatures of the variables of the atoms that share the same variables are adjacent. Moreover, we also have to ensure the signatures of the variables that belong to atoms with different variables are not adjacent, i.e. separated. Therefore, we have to create buffers between the groups of the signatures of the variables that belong to atoms with different variables. The size of these buffers is the spacing value Sv. We propose the size of these buffers to be greater than the cardinality of the largest group of the variables of D_2 that belong to atoms with one variable. We refer to the spacing value of a list of signatures of variables that are sorted and spaced according to the variable name of the atoms of these variables as Sv_v. To ensure the decision variables associated with

the variables of atoms with the same variable in D_1 are mapped to variables in D_2 that belong to atoms with one variable in D_2, the values of the decision variables of two variables in D_1 that belong to atoms with the same variable must not differ by more than the spacing value Sv_V. We call the set of constraints that capture these requirements the Same Variable Constraints, $\mathrm{SVC}(O_1)$, which is formalised as follows.

$$\mathrm{SVC}(O_1) = \{(|d(v_1) - d(v_2)| \leq \mathrm{Sv}_V) : \forall v_1, v_2 \in \mathrm{Variables}(O_1)(\mathrm{VarSig}_1(v_1) = \mathrm{VarSig}_1(v_2))\} \tag{4.29}$$

In this section, we have explained the variable mapping SMT problem and listed the SMT constraints that capture the conditions of Theorem 3 in addition to the condition of consistent domains predicate mapping. The following section describes the meta-planning task of finding the SOSOs of the operators of the given domains. This process is an essential part of validating the functional equivalence of simple planning domain models. The concept of the SOSOs of operators was introduced in Section 4.6.2.

4.12 Worked Example

There is no better example to explain the outputs of our method than validating the functional equivalence between the Mystery domain [121] and the original logistic domain from which the Mystery domain was devised. The committee of the 1998 Planning Competition wanted to conceal the underlying structure of a logistic planning domain model to make it harder for planners to get any clues from the semantics of the names of the domain's operators and predicates. The logistic domain describes vehicles moving cargoes between the nodes of a network of routes. To disguise the identity of the logistic domain, they labelled nodes as foods, vehicles as pleasures, and cargo objects as emotions. Though this disguise changed the names of the operators and predicates of the logistic domain, these changes did not affect its reach set. Hence, both the logistic and Mystery domains must be functionally equivalent under the predicate mapping used in producing the Mystery domain.

Before we can test the functional equivalence of these two domains, we have to modify them to meet our tool's requirements. In the Mystery domain, we have to transform the static atoms that implicitly specify the types of the parameters of the operators into explicit types. For instance, the operator "overcome" has the untyped parameters (?c ?v ?n ?s1 ?s2). The types of these variables are defined by the static atoms (pain ?c), (pleasure ?v), (food ?n), and (planet ?s1). Thus, we have to remove these static atoms and constrain the parameters of this operator with the types derived from their static predicates. Hence, the untyped parameters of the operator "overcome" becomes (?c - pain ?v - pleasure ?n - food ?s1 - planet ?s2 - planet). Furthermore, in the logistic domain, we had to split the untyped PDDL predicate "(at ?v ?n)" into two typed predicates, "(at_v ?v - vehicle ?n - node)" and "(at_c ?c - cargo ?n - node)". Similarly, in the Mystery domain, we had to split the untyped PDDL predicate "(craves ?v ?n)" into two typed predicates, "(craves_v ?v - pleasure ?n - food)" and "(craves_c ?c - pain ?n - food)".

Running our tool on these modified domains resulted in the verdict that the two domains are functionally equivalent. Moreover, our tool provided the variable mappings listed in Table 4.1. These variable mappings are the solution to the SMT problem explained in Section 4.11; they also represent atom mappings as described in Section 4.10.2.1.

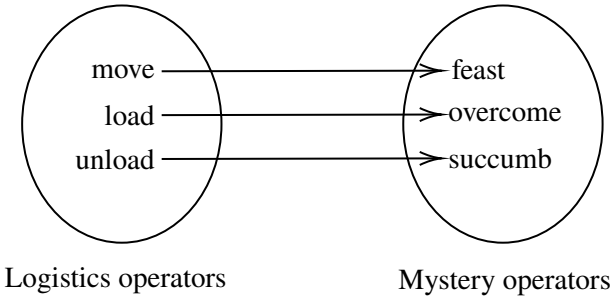


Figure 4.3: The mapping from the operators the Logistics domain to the operators of the Mystery domain.

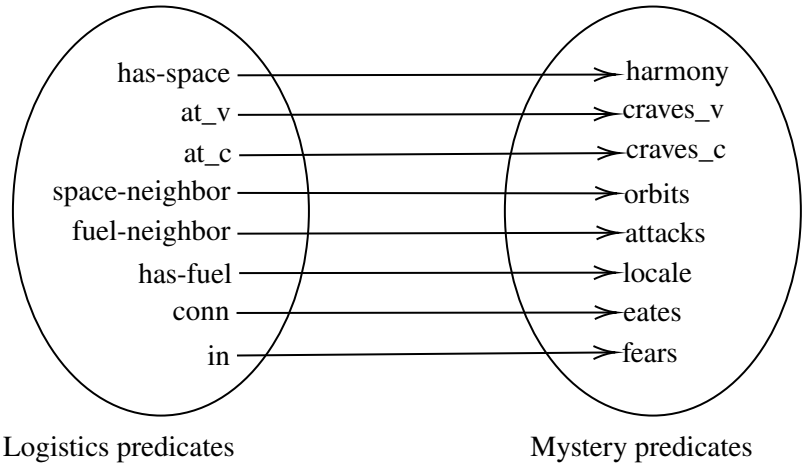


Figure 4.4: The mapping from the predicates of the Logistics domain to the predicates of the Mystery domain that makes the reach set of the former equal to the reach set of the latter.

These atom mappings respect the conditions of Theorem 3 because they are generated from the solution of our SMT problem, which is designed to enforce the constraints of Theorem 3. Note that these atom mappings are not shown explicitly; they are impeded in the variable mappings in Table 4.1. According to Theorem 3, for each atom mapping produced from the variable mappings in Table 4.1, there is a predicate mapping that makes the reach set of each operator from the Logistic domain equal to the reach set of an operator from the Mystery domain. The predicate mappings extracted from the atom mappings, which in their turn are produced from the variable mappings, are shown in Figure 4.4.

Moreover, the atom mappings are consistent with regard to the mapping of their predicates because they are produced from the solution of our SMT problem, which has the constraint that enforces this predicate consistency. Therefore according to Theorem 2, under the predicate mapping, which is made from the union of the predicate mappings in Figure 4.4, the reach set of the Logistic domain is equal to the reach set of the Mystery domain. Hence, as per Definition 4.6, the Logistic domain is functionally equivalent to the Mystery domain. Furthermore, our tool extracted the operator mapping depicted in Figure 4.3 from the found variable mappings in Table 4.1.

Variable signatures of the Logistics domain	**Variable signatures of the Mystery domain**
?v 2 has-space load ?v?s1 1	?v 2 harmony overcome ?v?s2 1
?v 2 has-space unload ?v?s2 1	?v 2 harmony succumb ?v?s2 1
?v 2 at_v move ?v?l1 1	?v 2 craves_v feast ?v?n1 1
?v 2 has-space unload ?v?s1 1	?v 2 harmony succumb ?v?s1 1
?v 2 in load ?c?v 2	?v 2 fears overcome ?c?v 2
?v 2 in unload ?c?v 2	?v 2 fears succumb ?c?v 2
?v 2 at_v unload ?v?l 1	?v 2 craves_v succumb ?v?n 1
?v 2 has-space load ?v?s2 1	?v 2 harmony overcome ?v?s1 1
?v 2 at_v load ?v?l 1	?v 2 craves_v overcome ?v?n 1
?v 2 at_v move ?v?l2 1	?v 2 craves_v feast ?v?n2 1
?c 2 in load ?c?v 1	?c 2 fears overcome ?c?v 1
?c 2 in unload ?c?v 1	?c 2 fears succumb ?c?v 1
?c 2 at_c load ?c?l 1	?c 2 craves_c overcome ?c?n 1
?c 2 at_c unload ?c?l 1	?c 2 craves_c succumb ?c?n 1
?l 2 at_c unload ?c?l 2	?n 2 craves_c succumb ?c?n 2
?l 2 at_v unload ?v?l 2	?n 2 craves_v succumb ?v?n 2
?l 2 at_c load ?c?l 2	?n 2 craves_c overcome ?c?n 2
?l 2 at_v load ?v?l 2	?n 2 craves_v overcome ?v?n 2
?l1 2 conn move ?l1?l2 1	?n1 2 eats feast ?n1?n2 1
?l1 2 at_v move ?v?l1 2	?n1 2 craves_v feast ?v?n1 2
?l1 2 has-fuel move ?l1?f1 1	?n1 2 locale feast ?n1?l2 1
?l1 2 has-fuel move ?l1?f2 1	?n1 2 locale feast ?n1?l1 1
?s1 2 space-neighbor load ?s2?s1 2	?s2 2 orbits overcome ?s1?s2 2
?s1 2 space-neighbor unload ?s1?s2 1	?s1 2 orbits succumb ?s1?s2 1
?s1 2 has-space unload ?v?s1 2	?s1 2 harmony succumb ?v?s1 2
?s1 2 has-space load ?v?s1 2	?s2 2 harmony overcome ?v?s2 2
?s2 2 space-neighbor load ?s2?s1 1	?s1 2 orbits overcome ?s1?s2 1
?s2 2 has-space load ?v?s2 2	?s1 2 harmony overcome ?v?s1 2
?s2 2 has-space unload ?v?s2 2	?s2 2 harmony succumb ?v?s2 2
?s2 2 space-neighbor unload ?s1?s2 2	?s2 2 orbits succumb ?s1?s2 2
?f1 2 has-fuel move ?l1?f1 2	?l2 2 locale feast ?n1?l2 2
?f1 2 fuel-neighbor move ?f2?f1 2	?l2 2 attacks feast ?l1?l2 2
?f2 2 fuel-neighbor move ?f2?f1 1	?l1 2 attacks feast ?l1?l2 1
?f2 2 has-fuel move ?l1?f2 2	?l1 2 locale feast ?n1?l1 2
?l2 2 conn move ?l1?l2 2	?n2 2 eats feast ?n1?n2 2
?l2 2 at_v move ?v?l2 2	?n2 2 craves_v feast ?v?n2 2

Table 4.1: The mapping from the signatures of the variables of the Logistics domain to the signatures of the variables of the Mystery domain.

4.13 Create random valid macros for testing

As part of our numerical experiment, we randomly create valid macros and augment the given planning domain models with these valid macros to generate planning domain model validation tasks.

The macro-building task is treated as a meta-planning task. The solution to this meta-planning problem is a plan that specifies which operators from the given domain, D, have to be added to an empty sequence of operators and in what order. This planning task consists of a meta-planning domain model, which is explained in Section 4.13.2 and a meta-planning problem, that is explained in Section 4.13.3.

The meta-planning domain model, which is a macro-building planning domain model provides a planner with the tools to gradually add operators from the domain D to a macro template, which we call the macro under construction. The meta-planning problem specifies the initial state of the macro under construction, which should be empty, i.e. the macro should not have any operators. Additionally, the meta-planning problem sets the required length of the macro under construction in the meta-goal states. Each state in the search space of this meta-planning problem represents a sequence of operators; the initial state represents an empty sequence; the goal state is any sequence of operators with the required number of operators. Each transition in this search space represents adding an operator from the domain D to the existing sequence of operators (the macro under construction).

Once the meta-planning domain and problem are produced for the given domain, we use the FF planner to find a solution to the meta-planning problem. The following sections further explain the meta-planning task for randomly building valid macros.

4.13.1 Building macros in the space of lifted operators

Building macros in the space of lifted operators is the process of finding a sequence of lifted operators from D and a unification and unionisation configuration of the parameters of consecutive operators, such that the macro produced from consolidating this sequence of operators has the required length.

Instead of searching for a sequence of operators in the space of lifted operators of D and then searching for a proper unification and unionisation configuration of the parameters of consecutive operators, we propose to do the search in the space of ground operators of D. The advantage of searching in the space of ground operators of D is that we do not have to worry about finding the proper unification and unionisation configuration of the parameters of consecutive operators. This advantage is realised because the search in the space of ground operators deals with objects rather than parameters. On the other hand, a unification and unionisation configuration of the parameters of consecutive operators is required when we reason about lifted operators because the scope of parameters is limited to their operators. When searching in the space of ground operators, we can readily consolidate a sequence of ground operators because the scope of objects, which depends on the scope of types, is general to all operators of D. After finding a ground macro with the required length, we have to lift the objects of this ground macro so we can add it to the domain d. Lifting the objects of a macro is a straightforward process because we know the types of the objects.

To work with ground operators, we have to ensure all operators can be instantiated by the planner, we define constants from each type. The number of constants of a given type is equal to the maximum number of the parameters of that type in all operators in the domain.

4.13.2 Meta-planning Domain Model

Like other regular planning domain models, our meta-planning domain model has types, atoms, and operators. We suffix these components with the word "meta" to indicate that these parts belong to the meta-planning domain model. The meta parts of our meta-planning domain model D_M are derived from the respective parts of the domain D.

Meta-atoms The atoms in the meta domain D_M are derived from the atoms of the domain D. For every atom that appears as a precondition in the operators of D, a meta-atom is added to the meta domain D_M. The name of this added atom is the name of the original atom suffixed with "pre-om". For example, for the atom (at−soil−sample ?p), which is a precondition in the operator "sample−soil" in D, we add the meta-atom (at−soil−sample−pre−om ?p) to the atoms of the meta domain D_M. Similarly, new atoms are added to the meta domain with suffixes "add-om" and "del-om" for the atoms in add and delete effects in the operators of the domain D. The meta-atoms will track the preconditions, add effects and delete effects of the macro under construction. For instance, when a meta-operator adds "(at−soil−sample ?p)" as a precondition to the macro under construction m, this meta-operator has to set to true the meta-atom "(at-soil-sample-pre-om ?p)" to indicate "(at-soil-sample ?p)" is now a precondition of m.

To ensure the macro-building task returns macros with distinct operators, we supplemented the meta-operators with meta-atoms to work as interlocks to prevent an operator from being scheduled twice.

The meta-numerical fluent "operators-count" is also add to D_M to track the number of operators added to the macro under construction. Each meta-operator is augmented with an add effect to increase the fluent "operators-count". Thus, whenever the planner schedules a meta-operator to add its normal operator to the macro under construction, the fluent "operators-count" value gets increased by one.

Note that the difference between the atoms of D and the meta-atoms of the meta domain of D_M is just the name of the atoms. Therefore, the types of the meta-atom variables are the same as those of the atoms in the domain D.

Meta-operators The meta-operators of the meta-planning domain D_M add the operators from the domain D to the macro m. For each operator o in D, there is a meta-operator that adds the operator o to the end of the macro m. **The name of the meta-operator** that adds an operator o is the name of the operator o prefixed with the word"add". For example, the meta-operator that adds the operator "sample−soil" from D to the end of a macro m is "add−sample−soil". Besides the name, meta-operators also have parameters, preconditions, add effects, conditional add effects, and conditional delete effects, but no delete effects.

The preconditions and effects of the meta-operators have meta-atoms. The meta-atoms are derived from the atoms of the normal operators for which the meta-operators are defined. Since the meta-atoms

of the meta-operators are derived from the atoms of the normal operators by only suffixing the name of the atoms, both normal and meta-atoms have the same parameters. Furthermore, since the parameters of the operators are made from the union of the parameters of their atoms, **the parameters of the meta-operators** are the same as the parameters of the normal operators.

The preconditions of the meta-operators ensure only valid macros are built by guaranteeing any operator o_o from D is added by its meta-operator to the macro under construction only if none of its preconditions appears as a delete effect in the macro under construction. So, the meta-operator that adds the operator o_o must have a precondition to check that the preconditions of o_o have not been added as delete effects to the macro under construction. To do so, the atoms in the preconditions of the operator o_o are suffixed with "del-om" and added as negative preconditions to the meta-operator. For example, assume the operator "soil−sample" have the precondition "(at−soil−sample ?p)", then its meta-operator "add−soil−sample" will have as the meta preconditions "(not (at−soil−sample−del−om ?p))". This negative precondition means the meta-operator "add-soil-sample" cannot be added to the macro under construction if the meta-atom "(at-soil-sample-del-om ?p)" is true. This meta-atom is true if only there is a proceeding operator in the macro under construction that has "(at-soil-sample ?p)" as a delete effect, and none of the subsequent operators in the macro under construction has it as an add effect.

Meta add effects add **Meta delete effects** of the operator represented by the meta-operator to the macro under construction. The atoms in the delete effects of the normal operator are suffixed with "del-om" and added as meta add effects to the meta-operator. For example, suppose the operator "soil−sample" have the atom "(at−soil−sample ?p)" as a delete effect, then its meta-operator "add−soil−sample" will have the meta-atom "(at−soil−sample−del−om ?p)" as a meta add effect. This meta add effect means when the meta-operator "add-soil-sample" is added to the macro under construction, then the meta-atom "(at-soil-sample-del-om ?p)" will become true to indicate the macro under construction has "(at-soil-sample ?p)" as a delete effect.

Meta-conditional effects are used by the meta-operators to update the preconditions, add effects, and delete effects of the macro under construction according to the preconditions, add effects, and delete effects of their related normal operators and according to the status of the macro under construction. The meta-conditional effects are generated using four rules. These rules are explained in the following subsection.

4.13.2.1 The Rules of the Meta-conditional Effects

Let us consider an operator o_o and a macro under construction m. When appending the macro m with the operator o_o, the first rule dedicates each atom t in the preconditions of o_o must be added as a precondition to m unless t has already been added to the preconditions of m in a previous step, or the precondition t has a supporter in m, i.e. t is already supported by an add effect in m; hence, t does not have to be added as a precondition to m.

1. For all t in Pre(o_o), t is added to Pre(m) if t is neither in Pre(m) nor in Add(m)

The second rule states that when adding an operator o_o with an atom t as an add effect to the macro m which has the same atom t as a delete effect, the delete effect of t in the macro m has to be removed. It is essential to remove any violators for t in m when appending o_o with the add effect t to m so the macro m remains consistent.

2. For all t in Add(o_o), t is removed from Del(m) if t is in Del(m)

The third rule dedicates any add effect t of o_o have to be added as an add effect to m unless t has already been added to the add effects of m in a previous step, or the atom t does not have to be added as an add effect to m because it is already a precondition for m. Note that the macro m cannot have t as a delete effect after completing the process of adding this atom as an add effect to m because the second rule requests this atom to be removed from the delete effect of m when it is added as an add effect to m.

3. For all t in Add(o_o), t is added to Add(m) if t is neither in Add(m) nor in Pre(m)

The fourth rule implies that the add effects of m that match the delete effects of o_o have to be removed as add effects of m, so we do not produce a macro m with a delete effect and an add effect with the same atom.

4. For all t in Del(o_o), t is removed from Add(m) if t is in Add(m)

This section explained our meta-planning domain model, including the meta-atoms and meta-operators. The following section describes the meta-planning problem, which includes our meta-planning task's initial state and goal condition.

4.13.3 Meta-planning Problem

The solution to this planning problem is a sequence of meta-actions that gradually build a macro from the operators of D with the required length. The components of our meta-planning problem are discussed in the following paragraphs.

Meta-objects In the previous section, we have explained that the meta-planning domain model has the same types as the domain D. Moreover, in Section 4.13.1, we have explained the number of required objects for each type of D.

Meta-initial state The Meta-initial state of this problem has the value of the numerical fluent "operators-count" as zero to indicate that the initial macro under construction is empty. Moreover, all meta atoms are false in the Meta-initial state. This configuration means the macro under construction is clean, i.e, it has no preconditions, add effects or delete effects.

Meta-goal The goal condition of this modified macro building problem is the number of the required operators in the targeted macro. The goal condition requests only the numerical fluent "operators-count" to be equal to the required length of the produced macro. If "operators-count" equals the specified value in a meta-state, then the transitions from the empty meta-initial state to this meta-state define a sequence of ground operators such that its consolidation is a macro of the required length.

In this section, we have explained how to randomly build valid macros with a specific lengths. These macros are used in empirical experiments that evaluates D-VAL in the the following section.

4.14 Experiment

To demonstrate the feasibility of our method, we implemented it and tested it with 74 functional equivalence validation tasks. We used 13 planning domain models from the International Planning Competition (IPC) that meet the scope of D-VAL. From nine domains, we created six modified versions, and from four domains, we created five modified versions. Then, we validated the functional equivalence between the original domains and their modified versions. The six proposed modifications are intended to test the response of our method to the change in the number of operators, the change of atoms in the preconditions and add effects, and the change in the parameters of the atoms.

Three modifications increase the number of operators by augmenting the modified versions with a hand-crafted valid macro operator, a randomly created valid macro, and a randomly created invalid macro. One modification decreases the number of operators by deleting a randomly selected operator from the original domain to create a modified version. The modification that changes the atoms of operators swaps one atom from the add effect of an operator with a precondition from the same operator to create a modified version. To make a change in the parameters of an atom, the last modification swaps two variables of the same type in the parameters of an atom in a random part of an operator.

We have developed a random test generation method to apply these modifications to the original domains and implemented as a tool. This tool randomly selects the targeted operator, atom and variable for each modification. To randomly create valid macros, we use the method explained in Section 4.13. In this experiment, we choose to build the randomly created valid macros such that each macro has three unique operators.

Randomly created invalid macros are produced from the randomly created valid macros by randomly selecting a precondition and an add effect from the valid macro and swapping them.

Table 4.2 lists the description of the modifications applied to the Elevator domain to produce its modified versions. The table also states if the modified version is expected to be functionally equivalent to the original domain. In theory, adding hand-crafted and randomly created valid macros should not change the reach sets of the modified version from the reach set of its original domain. Hence the modified version in these cases and their original domains must be functionally equivalent. On the contrary, the other modifications are expected to produce versions with reach sets different from the reach sets of their original domains. The modifications to the other domains are detailed in Appendix B.1.

Note that the test generation tool did not generate versions of the Gripper, Child-snack and Logistics domains with swapped variables because these domains do not have any predicate with two variables of the same type. The variables of an atom must have the same types to be swapped by our tool because an atom with swapped variables of different types will not match its predicate definition. A domain with an atom that does not match a defined predicate is invalid. Thus, to swap variables of different types in an atom, we would have to add a new predicate to match the atom with the swapped variables. If we add a new predicate to the modified version of the original domain, then the two domains, the original and its modified version, will have a different number of predicates. Hence, our tool will judge the two domains as not functionally equivalent without any proper investigation, as the scope of our method is limited to domains with an equal number of predicates. Therefore, the condition that an atom's variables must be of the same type to be swapped is essential to ensure the produced validation test is meaningful.

Moreover, the test generation tool did not generate a version of the Blocksworld domain with swapped atoms because, in each operator from this domain, the set of the preconditions is equal to the set of delete effects. Therefore, any swap between a precondition and an add effect in any operator will cause the produced modified domain to be inconsistent because there will be an operator with the same atom that appears as an add effect and a delete effect.

The results in Section 4.14.1 report the verdict of our tool on the functional equivalence of each domain and its modified versions. In addition, the results provide the CPU time spent by the planner and the SMT solver for each validation task. The experiments were run on a computer node from BlueCrystal Phase 4 in the Advanced Computing Research Centre at the University of Bristol[2]. The used computer node has two 14-core 2.4 GHz Intel E5-2680 v4 (Broadwell) CPUs and 128 GiB of RAM. Each validation test was limited to one CPU with 20 GiB of RAM.

[2] `https://www.bristol.ac.uk/acrc/`

Domain version	Expected impact	Modification description
Elevator with crafted valid macro	Yes	The valid macro operator "board-up-depart" is handcrafted and added to this modified version.
Elevator with random valid macro	Yes	The valid macro operator "board-depart-up" is randomly created from the operators of the original domain and added to this modified version.
Elevator with random invalid macro	No	The invalid macro operator "board-depart-up" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (served ?x3) with the precondition (origin ?x3 ?x1) in the original valid macro.
Elevator with swapped variables	No	The variable ?f1 in the parameters of the add effect (above ?f1 ?f2) in the operator "down" in the original domain is swapped with the variable ?f2 from the parameters of the same add effect of the same operator in this modified version.
Elevator with swapped atoms	No	The atom (lift-at ?f2) in the add effects of the operator "down" in the original domain is changed to a precondition in this modified version, and the atom (above ?f2 ?f1) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Elevator with deleted operator	No	The operator "up" is removed from this modified domain, this operator exists in the original domain.

Table 4.2: The description of the modifications applied to the Elevator domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Elevator domain is expected to produce a version that is functionally equivalent to the original domain.

The domains used in these experiments, alongside their modified versions, the outputs obtained from using D-VAL for validating their functional equivalence, and instructions on how to use D-VAL are all available online.[3]

4.14.1 Results and Discussion

Tables 4.4 to 4.16 provide the results of our experiments. These experiments show that our method confirmed the functional equivalence of all functionally equivalent domains and their versions and the non-functional equivalence of the non-functionally equivalent simple domains in less than 45 seconds. Furthermore, our tool took just a few seconds, and sometimes fraction of seconds, to terminate with no conclusive verdict for validating the functional equivalence between all complex domains (apart from the Parking and Freecell domains) and their versions that are supposed to be not functionally equivalent to their original domains.

D-VAL took around 914.27 seconds to produce no conclusive verdict for validating the functional equivalence between the Parking original domain and its modified versions with random invalid macro,

[3] `https://github.com/Anas-Shrinah/DVAL-Validation-tool-for-planning-domain-models-functional-equivalence`

swapped variables, and deleted operator. Additionally, D-VAL needed 615.46 seconds to terminate with no conclusive verdict for validating the functional equivalence between the Parking original domain and its modified version with swapped atoms. On the other hand, D-VAL spent around 1803 seconds to complete the validation with no conclusive verdict for validating the functional equivalence between the original Freecell domain and its modified versions (with random invalid macro, with swapped variables, with swapped atoms, with deleted operator).

Most of the computation time in this cases was used by the planner as it was working to check whether the provided original domains and their modified versions were simple or complex using the method described in Section 4.7.1. As discussed in that section, the method of checking the type of the provided domains performs two tests. The first one is concerned with checking if either of the two domains has any *SDE* . This test performs simple and direct checks; thus it does not take a long time. On the other hand, if neither of the domains have an *SDE* , then D-VAL checks if any operator from each domain has the same delete effects as a sequence of operators from the same domain. This test uses a planner to prove a meta-planning problem is not solvable. Thus, this test can take substantial time depends on the size of this meta-planning problem. In the case of the Parking and Freecell planning domain models, D-VAL did not find these domains and their modified versions to have any *SDE* . Therefore, D-VAL invoked the planner to prove the meta-planning problem produced by the method described in Section 4.7.1 were unsolvable.

Note, the number of parameters of the same type and the number of the groups of parameters with different types in the operators of a domain affect the size of the search space produced by the meta-planning of checking the type of this domain. Moreover planners, like other combinatorial search algorithms, suffer from the state-space explosion problem. Therefore, in comparison with the Cave diving domain, where the maximum number of parameters of the same type in any operator in this domain was just two, the planner took long time to decide on the complexity of the Parking domain, which has an operator with three parameters of the same type. Furthermore, the planner timed out, reached the time limit of 1800 seconds, before it could decide on the type of the Freecell domain because this domain has an operator with four parameters of one type.

Our tool produced correct verdicts on the functional equivalence of all 74 validation tasks. These verdicts matched our expectations apart from three tests: Scanalyzer with swapped variables, Hiking with swapped variables and Hiking with deleted operator. The version of the Hiking domain with swapped variables is produced from the original Hiking domain by swapping the positions of the variable ?x3 with the variable ?x5 in the parameters of the precondition “(partners ?x6 ?x3 ?x5)” in the operator “walk_together_M”. This operator is depicted in Listing 4.4.

To clarify why our method found the Hiking domain functionally equivalent to its version with swapped variables, we have to compare this validation task to a similar one where our tool produced a non-functional equivalence verdict. For this purpose, we choose the validation task of testing the functional equivalence of the Blocksworld domain and its version with swapped variables.

In the version of the Hiking domain with swapped variables, the variables ?x3 and ?x5 are symmetric in the atoms of the operator “walk_together_M”. Two variables are symmetric if they appear in the

same positions of the same atoms with the same other variables. In the operator "walk_together_M", the variable ?x3 appears as the first variable in the precondition (at_person ?x3 ?x4) and the variable ?x5 appears as the first variable in the delete effect (at_person ?x5 ?x4). Note that in these two atoms, the second variable is the same, ?x4. Moreover, both variables, ?x3 and ?x5, are the first variables in the add effects (at_person ?x3 ?x2) and (at_person ?x5 ?x2), respectively, while the second variable in both add effects is the same, ?x2.

```
(:action walk_together_M
 :parameters (?x1 - tent ?x2 ?x4 - place ?x3 ?x5 - person ?x6 - couple)
 :precondition (and (at_tent ?x1 ?x2) (up ?x1) (at_person ?x3 ?x4)
   (next ?x4 ?x2) (at_person ?x5 ?x4)
   (partners ?x6 ?x5 ?x3)  \\ the precondition with swapped variables.
   (walked ?x6 ?x4))
 :effect (and (at_person ?x3 ?x2) (at_person ?x5 ?x2) (walked ?x6 ?x2)
  (not (at_person ?x3 ?x4)) (not (at_person ?x5 ?x4))
  (not (walked ?x6 ?x4))))
```

Listing 4.4: The operator "walk_together" from the version of the Hiking domain with swapped variables.

The impact of swapping ?x3 and ?x5 in the operator "walk_together_M" can be observed when both domains, the original one and the modified version, are used in a planning task where the operators "walk_together" from the original domain and "walk_together_M" from the modified domain are instantiated with the same objects and scheduled to produce the same transition. For example, let Anas and Lujain be the persons who make Couple1. Assume this couple has Tent1 and is at Snuff Mills, and they want to walk together to Oldbury Court. From this initial state, a planner using the original domain will produce this plan:

```
walk_together(Tent1,Snuff Mills,Oldbury Court,Anas,Lujain,Couple1)
```

On the other hand, from the same initial state, the same planner using the modified domain will produce this plan:

```
walk_together(Tent1,Snuff Mills,Oldbury Court,Lujain,Anas,Couple1)
```

The only difference will be in the order of the objects in the parameters of the two operators. Nevertheless, both domains can be used to reach the same end state for this initial state. Thus, the two domains are functionally equivalent.

The aim of this example is not to prove that swapping the positions of symmetric variables in the parameters of an operator does not change the reach set of its domain. This property is valid according to Theorem 3. This example is meant to provide a concrete example of this case and show the impact of this modification.

On the other hand, swapping two variables in an operator in the original Blocksword domain caused the original domain and its modified version to become non-functionally equivalent. In our experiment, the version of the Blocksword domain with swapped variables is produced from the original Blocksword

domain by swapping the positions of the variable ?x with the variable ?y in the parameters of the add effect "(on ?y ?x)" in the operator "stack_M". This operator is depicted in Listing 4.5.

```
1 (:action stack_M
2  :parameters (?x ?y - block )
3  :precondition (and (holding ?x) (clear ?y))
4  :effect (and (clear ?x) (handempty)
5   (on ?y ?x) \\ The add effect with the swapped variables.
6   (not (holding ?x)) (not (clear?y))))
```

Listing 4.5: The operator "stack" from the version of the Blocksworld domain with swapped variables.

Note that the variables ?x and ?y are not symmetric in the atoms of the operator "stack_M". For instance, ?x is the first and only variable in the precondition (holding ?x), and ?y is the first and only variable in the precondition (clear ?y).

The impact of swapping ?x and ?y in the operator "stack_M" can be observed when both the original domain and the modified version are used in a planning task where the operators "stack" from the original domain and "stack_M" from the modified domain are instantiated with the same objects and scheduled to produce the same transition. For example, assume we want to stack a block B_1 on top of a block B_2. According to the original domain, starting from an initial state where a robot holds B_1, and B_2 is be clear, applying the action "stack B_1 B_2 " causes B_1 to be on B_2. However, according to the modified domain, starting from the same initial state, applying the action "stack_M B_1 B_2 " causes B_2 to be on B_1.

Note that "stack_M B_2 B_1 " is not applicable from the assumed initial state because this action requires the robot to hold B_2, and B_1 to be clear, which is the opposite of the initial state in this example. Thus, both domains cannot be used to reach the same end state for this initial state. Hence, the original Blocksworld domain and its version with swapped variables are not functionally equivalent.

The above discussion about the functional equivalence of the Hiking domain and its version of Hiking with swapped variables is also applicable for the validation task of the functional equivalence between the original Scanalyzer domain and its version with swapped variables.

The last case where our tool gave an unexpected outcome is when validating the functional equivalence between the original Hiking domain and its version with a deleted operator. The version of the Hiking domain with deleted operator is produced from the original Hiking domain by deleting the operator "drive_tent_passenger". Intuitively, deleting an operator from a domain is expected to produce a domain that is not functionally equivalent to its original domain. However, this was a different case. It turned out that the operator "drive_tent_passenger" is a macro in the original domain. Any transition produced by this operator can be matched by the sequence ⟨"drive_tent", "drive", "drive_passenger"⟩. In other words, any state reachable by the operator "drive_tent_passenger" is also reachable by this sequence. Thus, removing this operator would not render any reachable state unreachable. Hence, this operator is redundant from the perspective of domain reach sets. Therefore, the original Hiking domain and its version with the deleted operator are functionally equivalent.

It is worth noting that the planner was able to complete its part in a fraction of a second apart from checking the type of the Parking and Freecell domains. The first task of the planner is to check if the

reach set of each operator in the two domains is a subset of the reach set of a sequence of operators in the same domain, as explained in Section 4.6.1. This planning task aims to find all sequences of operators that satisfy certain conditions. In a sense, this planning task aim to prove their planning problems are unsolvable. It is well known that proving unsolvability is much harder than finding solutions to planning problems in general. However, each of these planning problems was solved in a fraction of second. This impressive performance of the planner is attributed to the reduction of unrelated operators before commencing solving the meta-planning problems in Section 4.6.1. This reduction in the number of operators in the meta-planning problems makes proving their unsolvability easy.

The other general observation is that the time required for the Z3 solver to solve our variable mapping SMT problem increases as the complexity of the planning domain models involved in the validation task increases. This behaviour is anticipated as the number of SMT constraints increases as the number of operators, their atoms, and the number of the variables of the atoms increases.

Table 4.3 explains the justifications behind the verdicts returned by D-VAL in the results of the experiments depicted in Tables 4.4 to 4.16. Row number one, from Table 4.3, explains why D-VAL found two planning domain models (simple or complex) to be functionally equivalent. For instance, D-VAL provided this justification to explain why the validation tasks “Blocksworld with crafted valid macro” and “Blocksworld with random valid macro” were decided to be functionally equivalent in Table 4.6. Rows 2, 4, 6, 8, and 10 detail why D-VAL found two simple planning domain models not functionally equivalent. For example, the justifications in rows 2 and 8 of Table 4.3 are provided by D-VAL to clarify why the validation tasks Gripper with swapped atoms and Gripper with deleted operator in Table 4.5 were found to be not functionally equivalent.

Moving on, rows 3, 5, 7, 9, and 11 detail why D-VAL could not provide a conclusive verdict about the functional equivalence of two complex planning domain models. For example, the justifications in rows 3, 9, and 11 of Table 4.3 are provided by D-VAL for not producing conclusive verdicts about the other validation tasks in Table 4.14. Finally, row 12 describes the case where the two given domains have different numbers of predicates of equal arity.

Reason Number	Justification
1	The two domains D_1 and D_2 have the same number of operators and a predicate consistent mapping from the atoms of the operators of D_1 to the atoms of the operators of D_2 that makes the reach set of D_1 equal to the reach set of D_2 has been found.
2	There is no predicate consistent mapping from the atoms of the primitive operators of D_1 to the atoms of the primitive operators of D_2 that makes the reach set of D_1 equal to the reach set of D_2.
3	There is no predicate consistent mapping from the atoms of the operators of D_1 to the atoms of the operators of D_2 that makes the reach set of D_1 equal to the reach set of D_2 .
4	At least one of D_1 primitive operators does not have any PFEO from D_2.
5	At least one of D_1 operators does not have any PFEO from D_2.
6	At least one of D_2 primitive operators does not have any PFEO from D_1.
7	At least one of D_2 operators does not have any PFEO from D_1.
8	D_1 has more primitive operators than D_2.
9	D_1 has more operators than D_2.
10	D_2 has more primitive operators than D_1.
11	D_2 has more operators than D_1.
12	The two domains have different number of predicates. Hence they are considered not functionally equivalent by our method.

Table 4.3: The description of the reasons behind the decisions returnd by D-VAL.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Elevator with crafted valid macro	Yes	FE	1	NT	0.27	0.18	0.09
Elevator with random valid macro	Yes	FE	1	NT	0.26	0.18	0.08
Elevator with random invalid macro	No	NCV	11	No	0.1	0	0.1
Elevator with swapped variables	No	NCV	3	No	0.23	0.15	0.08
Elevator with swapped atoms	No	NCV	3	No	0.23	0.15	0.08
Elevator with deleted operator	No	NCV	9	No	0.08	0	0.08

Table 4.4: The results of validating the functional equivalence between the Elevator domain and its modified versions. Expected impact: “yes” means the introduced modification on the original Elevator domain is expected to produce a version that is functionally equivalent to the original domain. These modifications are detailed in Table 4.2. The FE column reports the verdict of our tool on the functional equivalence of each validation task: FE: functionally equivalent; NFE: not functionally equivalent; and NCV: no conclusive verdict. The reason column provides the justification of the decision of our tool. The reported numbers can be decoded with the help of Table 4.3. Simple domains: “Yes” means both the original domain and its modified version are simple domains, “No” means both domains are not simple domains, i.e. either one or both are complex domains, “NT” means D-VAL did not test the type of the provided domains in this validation task, and “TO” means the planner timed out before it could decide on the type of the provide domains. Total time is the time taken by our tool to complete the validation task; it equals the sum of SMT and Planning times. SMT time is the time taken by the Z3 solver to check the existence of a suitable mapping between the predicates of the original domain and its modified version. Planning time is the time taken by the FF planner to find macro operators in the original domain and its modified version and to check operators are primitive in the case of simple domains.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Gripper with crafted valid macro	Yes	FE	1	NT	0.2	0.12	0.08
Gripper with random valid macro	Yes	FE	1	NT	0.15	0.11	0.04
Gripper with random invalid macro	No	NCV	11	No	0.09	0	0.09
Gripper with swapped atoms	No	NFE	2	Yes	0.33	0.18	0.15
Gripper with deleted operator	No	NFE	8	Yes	0.06	0	0.06

Table 4.5: The results of validating the functional equivalence between the Gripper domain and its modified versions. These modifications are detailed in Table B.1. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Blocksworld with crafted valid macro	Yes	FE	1	NT	0.25	0.16	0.09
Blocksworld with random valid macro	Yes	FE	1	NT	0.25	0.16	0.09
Blocksworld with random invalid macro	No	NCV	11	No	0.11	0	0.11
Blocksworld with swapped variables	No	NCV	3	No	0.22	0.13	0.09
Blocksworld with deleted operator	No	NCV	9	No	0.1	0	0.09

Table 4.6: The results of validating the functional equivalence between the Blocksworld domain and its modified versions. These modifications are detailed in Table B.2. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Parking with crafted valid macro	Yes	FE	1	Yes	0.82	0.69	0.12
Parking with random valid macro	Yes	FE	1	Yes	0.8	0.67	0.13
Parking with random invalid macro	No	NCV	11	No	914.69	0	914.69
Parking with swapped variables	No	NCV	3	No	913.33	0.64	912.69
Parking with swapped atoms	No	NCV	3	No	615.46	0.39	615.07
Parking with deleted operator	No	NCV	9	No	914.8	0	914.8

Table 4.7: The results of validating the functional equivalence between the Parking domain and its modified versions. These modifications are detailed in Table B.3. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Hiking with crafted valid macro	Yes	FE	1	NT	3.32	3.15	0.17
Hiking with random valid macro	Yes	FE	1	NT	3.12	2.94	0.17
Hiking with random invalid macro	No	NCV	11	No	0.21	0	0.21
Hiking with swapped variables	No	FE	1	NT	3.04	2.9	0.15
Hiking with swapped atoms	No	NCV	3	No	1.64	1.45	0.19
Hiking with deleted operator	No	FE	1	NT	3.08	2.95	0.13

Table 4.8: The results of validating the functional equivalence between the Hiking domain and its modified versions. These modifications are detailed in Table B.4. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Floor-tile with crafted valid macro	Yes	FE	1	NT	2.06	1.9	0.16
Floor-tile with random valid macro	Yes	FE	1	NT	2.06	1.89	0.17
Floor-tile with random invalid macro	No	NCV	11	No	0.16	0	0.16
Floor-tile with swapped variables	No	NCV	3	No	2.01	1.83	0.18
Floor-tile with swapped atoms	No	NCV	3	No	1.85	1.72	0.13
Floor-tile with deleted operator	No	NCV	9	No	0.13	0	0.13

Table 4.9: The results of validating the functional equivalence between the Floor-tile domain and its modified versions. These modifications are detailed in Table B.5. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Child-snack with crafted valid macro	Yes	FE	1	NT	0.88	0.74	0.14
Child-snack with random valid macro	Yes	FE	1	NT	0.84	0.71	0.13
Child-snack with random invalid macro	No	NCV	11	No	0.13	0	0.13
Child-snack with swapped atoms	No	NCV	3	No	1.05	0.85	0.2
Child-snack with deleted operator	No	NCV	9	No	0.12	0	0.12

Table 4.10: The results of validating the functional equivalence between the Child-snack domain and its modified versions. These modifications are detailed in Table B.6. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Logistics with crafted valid macro	Yes	FE	1	NT	3.68	3.43	0.24
Logistics with random valid macro	Yes	FE	1	NT	3.89	3.64	0.25
Logistics with random invalid macro	No	NCV	11	No	0.23	0	0.23
Logistics with swapped atoms	No	NCV	3	No	3.16	2.86	0.3
Logistics with deleted operator	No	NCV	9	No	0.23	0	0.23

Table 4.11: The results of validating the functional equivalence between the Logistics domain and its modified versions. These modifications are detailed in Table B.7. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Cave-diving with crafted valid macro	Yes	FE	1	NT	2.04	1.9	0.14
Cave-diving with random valid macro	Yes	FE	1	NT	2.12	1.96	0.16
Cave-diving with random invalid macro	No	NCV	11	No	0.24	0	0.23
Cave-diving with swapped variables	No	NCV	3	No	2.03	1.8	0.23
Cave-diving with swapped atoms	No	NCV	3	No	1.52	1.31	0.21
Cave-diving with deleted operator	No	NCV	9	No	0.18	0	0.18

Table 4.12: The results of validating the functional equivalence between the Cave-diving domain and its modified versions. These modifications are detailed in Table B.8. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Rover with crafted valid macro	Yes	FE	1	NT	5.11	4.95	0.15
Rover with random valid macro	Yes	FE	1	NT	5.3	5.09	0.21
Rover with random invalid macro	No	NCV	11	No	0.15	0	0.15
Rover with swapped variables	No	NCV	3	No	3.16	3.01	0.13
Rover with swapped atoms	No	NCV	3	No	3.19	3.0	0.19
Rover with deleted operator	No	NCV	9	No	0.13	0	0.13

Table 4.13: The results of validating the functional equivalence between the Rover domain and its modified versions. These modifications are detailed in Table B.9. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Pipesworld with crafted valid macro	Yes	FE	1	NT	15.49	15.34	0.15
Pipesworld with random valid macro	Yes	FE	1	NT	13.55	13.42	0.13
Pipesworld with random invalid macro	No	NCV	11	No	0.15	0	0.15
Pipesworld with swapped variables	No	NCV	3	No	14.64	14.47	0.15
Pipesworld with swapped atoms	No	NCV	3	No	4.07	3.94	0.13
Pipesworld with deleted operator	No	NCV	9	No	0.12	0	0.12

Table 4.14: The results of validating the functional equivalence between the Pipesworld domain and its modified versions. These modifications are detailed in Table B.10. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Scanalyzer with crafted valid macro	Yes	FE	1	NT	25.49	25.37	0.12
Scanalyzer with random valid macro	Yes	FE	1	NT	23.24	23.12	0.11
Scanalyzer with random invalid macro	No	NCV	11	No	0.12	0	0.12
Scanalyzer with swapped variables	No	FE	1	NT	24.75	24.64	0.1
Scanalyzer with swapped atoms	No	NCV	3	No	1.56	1.49	0.07
Scanalyzer with deleted operator	No	NCV	9	No	0.09	0	0.09

Table 4.15: The results of validating the functional equivalence between the Scanalyzer domain and its modified versions. These modifications are detailed in Table B.11. The description of the reported values in this table is available in the caption of Table 4.4.

Domain version	Expected impact	FE	Reason	Simple domains	Total time (s)	SMT time (s)	Planning time (s)
Freecell with crafted valid macro	Yes	FE	1	Yes	44.78	44.47	0.31
Freecell with random valid macro	Yes	FE	1	Yes	44.86	44.52	0.34
Freecell with random invalid macro	TO	NCV	11	No	1800.33	0	1800.3
Freecell with swapped variables	TO	NCV	3	No	1807.53	7.18	1800.35
Freecell with swapped atoms	TO	NCV	3	No	1807.55	7.14	1800.41
Freecell with deleted operator	TO	NCV	9	No	1800.34	0	1800.33

Table 4.16: The results of validating the functional equivalence between the Freecell domain and its modified versions. These modifications are detailed in Table B.12. The description of the reported values in this table is available in the caption of Table 4.4.

4.15 Summary

Validating the functional equivalence of planning domain models has many applications in KEPS, yet this topic has not received much attention from researchers.

In this chapter, we formally defined the concept of functional equivalence between planning domain models. Moreover, we presented a novel method that uses a planner and an SMT solver to validate the functional equivalence of planning domain models. Our approach differentiates between two types of

planning domain models, simple and complex. We have proved that our method is sound and complete when validating simple planning domain models and sound when validating complex planning domain models.

Additionally, we implemented our approach in a tool called D-VAL. To evaluate the performance of D-VAL, we developed a random test generation method that produced various types of validation tasks. Our experimental evaluation demonstrated the feasibility and effectiveness of our method. We evaluated 74 validation tasks from 13 published domains. D-VAL produced unexpected but correct verdicts for three of these validation tasks. These interesting test cases further increased the confidence in the correctness of D-VAL, which is based on a method that is theoretically proven to be sound. Furthermore, the results of the experiments showed that D-VAL is able to validate the functional equivalence of even the most challenging task in less than 45 seconds. These domains include complex domains such as Scanlayser and Free-cell domains.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

This thesis studied the verification and validation of planning domain models. Specifically, this thesis is focused on the verification of planning domain models with regard to safety properties and the validation of planning domain models' functional equivalence. The main objectives of this thesis were to design a novel verification approach that reduces the number of false positive counterexamples in the verification of planning domain models, as well as to develop a method to validate the functional equivalence of planning domain models independently from planning problems. This chapter reiterates the main contributions of this thesis and suggests interesting directions for future research.

5.1 Planning Domain Model Verification of Safety Properties

The existing state-of-the-art planning domain model verification methods are prone to false positive counterexamples due to their under-constrained nature. Consequently, they mislead designers into unnecessarily constraining planning domain models in the process of debugging these false counterexamples. This is a time-consuming process and can lead designers to have a false sense of achievement, which might cause them to overlook genuine violations of required properties [15]. Under-constrained verification is a well-studied problem in the V&V community [127]. However, the literature, e.g. Smith et al. [12], suggests that this aspect has been overlooked in the context of planning domain model verification.

To address this shortfall, we proposed to employ planning goals as constraints during verification. Thus, we introduced *goal-constrained planning domain model verification*, a concept borrowed from V&V research, which reduces the number of invalid planning counterexamples. Furthermore, we explained how model checking and planning techniques can be used to perform goal-constrained planning domain model verification of safety properties. Additionally, we formally proved that this novel approach is guaranteed to produce only true positive violations of safety properties, if and only if any exist as per

the definition of valid planning counterexamples introduced in this thesis. The proposed verification approach is simple, which makes it readily usable in practice. It is also effective since it reduces false positives, as formally proven in this thesis.

The empirical experiments showed that the goal-constrained verification method explores more states than under-constrained methods when verifying planning domain models that violate safety properties. The extra states explored by our method represent the overhead cost associated with guaranteeing that only valid planning counterexamples are returned. In contrast, when using model checking, our method explores fewer states than under-constrained methods in verifying planning domain models that do not violate safety properties. This advantage of our method makes it appealing for the final verification stages, where all counterexamples have been considered, and the aim is to conclude that the planning domain model is safe. Moreover, under-constrained verification might be intractable when verifying models with large state spaces. However, since our method explores fewer states when verifying models that do not have counterexamples, it might be possible to use our method to prove that such models are safe for specific planning goals.

Goal-constrained and under-constrained verification methods verify planning domain models with some constraints. The former verifies planning domain models with respect to single planning problems, whereas the latter verifies planning domain models with respect to individual pairs of a set of objects and an initial state. Consider the planning domain models used in the International Planning Competition. Each of these models is published with a set of relevant planning problems. These planning problems vary in terms of the number of objects and the initial states. Therefore, to verify if one of the IPC planning domain models satisfies some property, both constrained and under-constrained verification methods must be repeatedly applied to verify the given model against each of its planning problems.

An interesting direction for future work is investigating how verification reusability [128] and generalised planning [129, 130] can be used to increase the efficiency of verifying planning domain models for a set of planning problems. This could be possible by reusing results from verifying planning domain models for individual planning problems to establish the correctness of the same models against other planning problems.

Additionally, though our method reduces the number of invalid planning counterexamples as it removes counterexamples that do not falsify the safety property before achieving the planning goal, it does not exclude plans that are enriched with action sequences which are unnecessary to achieve the planning goal but required to falsify the given safety property. Therefore, our approach is partially robust against false positive counterexamples.

This limitation motivates future work to study how we can decide whether an action is added to a plan just for the purpose of falsifying the safety property or is it really needed to achieve the planning goal. With inside information about the search algorithm of a planner, it could be possible to check whether a sequence of actions can be potentially produced by the planner as a plan for a specific planning problem. Suppose we can guarantee a sequence of actions, which falsifies the safety property before achieving the planning goal, can never be returned by a planner. In that case, we can consider this sequence of

actions as an invalid planning counterexample for the specific planner. Therefore, we can strengthen the robustness of planning domain model verification by further reducing false positives. The potential benefit of this additional step in vetting planning counterexamples comes with the cost of further limiting the verification of planning domain models to a specific planner. This additional constraint poses no significant issue, as, in practice, organisations using planning techniques can commit to specific planners.

Moreover, we have explained how goal-constrained planning domain model verification can be performed using model checkers and planners that support state-trajectory-constrains. The latter method uses the modal operators defined in PDDL3 to specify the strong planning constraint that captures the safety property in the verification process. However, PDDL3 does not permit the use of nested modal operators. This restriction limits the level of expressiveness available to specify safety properties.

As future work, we suggest investigating the use of Temporally Extended Goals (TEGs) [131] to perform goal-constrained verification of planning domain models with nested LTL formulas. TEGs refer to goals that must not only be satisfied by the plan's final state, but also by the state trajectory overall.

5.2 Functional Equivalence Validation of Planning Domain Models

The recent boom in planning domain models' optimisation, reconciliation, learning and maintenance research areas motivates the need for methods to confirm that two versions of a planning domain model are functionally equivalent. Validating the functional equivalence of planning domain models is the problem of formally confirming that two planning domain models can be used to solve the same set of planning problems.

The need for techniques to validate the functional equivalence of planning domain models has been highlighted in the literature [21, 103, 104]. In this thesis, we have built on and extended the previous research by Shoeeb and McCluskey [103] and McCluskey et al. [21] by formally defining the notation of the functional equivalence of two planning domain models and proposing a novel approach that uses a planner and an SMT solver to validate the functional equivalence of planning domain models independently from planning problem instances.

Moreover, we have formally proven the soundness and completeness of our approach, and we have implemented it in a validation tool called D-VAL. To evaluate the performance of D-VAL, we have developed a random test generation method to generate functional equivalence validation tasks for planning domain models. This method takes as an input a planning domain model and produces six modified versions of this planning model. These modifications consist of randomly creating valid and invalid macros, deleting an operator, swapping two variables of the same type from an atom, and swapping two atoms between the preconditions and add effects of an operator. This method is implemented as a tool and used to generate 61 validation tasks from 13 published IPC planning domain models that meet the scope of D-VAL. We also augmented the randomly generated validation tasks with another 13 hand-crafted validation tasks to design a test suite for evaluating the performance of this method and future work.

The empirical experiments showed that, as anticipated, D-VAL produced correct verdicts on the

functional equivalence of 71 validation tasks. However, the experiments also yielded unexpected results for three validation tasks. Specifically, when validating three planning domain models and their modified versions, which were intended to be not functionally equivalent, D-VAL surprisingly ruled that these domains and their modified versions were functionally equivalent.

At first sight, these unexpected results suggest that our approach or its implementation might be flawed. However, after careful consideration, it was revealed that each planning domain model and its modified version in these three validation tasks, for which D-VAL gave unexpected verdicts, were indeed functionally equivalent. In one validation task, a modified planning domain model was produced by our random test generation tool by randomly deleting one operator from the Hiking planning domain model introduced by Simpson and McCluskey [132]. Initially, this validation task was expected to show that the domain model and its modified version were not functionally equivalent because the modified version misses one original operator. However, D-VAL determined that these two domain models were functionally equivalent. The investigation of the result of this validation task found that the deleted operator was actually a macro in the Hiking planning domain model. Therefore, deleting this operator from the Hiking planning domain model did not change its functionality. Ultimately, the correctness of D-VAL's verdicts was confirmed in all test cases.

Furthermore, the empirical experiments showed that D-VAL is very efficient as it validated the functional equivalence of even the most challenging task in less than 45 seconds. These domains include complicated domains such as Scanlayser and Free-cell domains.

The scope of D-VAL is limited to planning domain models with an equal number of predicates of equal arities. This restriction limits the applications of D-VAL. Therefore, an important area for future work is extending the scope of D-VAL to cover planning domain models with unequal numbers of predicates.

D-VAL gives conclusive verdicts when validating the functional equivalence of simple planning domain models. On the contrary, D-VAL cannot always return a decisive judgement when validating the functional equivalence of complex planning domain models. This limitation restricts the cases when our approach can give conclusive verdicts. Therefore, extending D-VAL to provide conclusive verdicts for complex domains is an important area for future work.

Moreover, the conclusive verdicts returned by D-VAL when validating the functional equivalence of two planning domain models are either Functionally Equivalent or Not Functionally Equivalent. This binary output might be of limited benefit for evaluating the quality of planning domain models engineering tools. As future work, we suggest extending D-VAL to return the distance between the given planning domain models if they are found to be not functionally equivalent. For example, when the variable mapping SMT problem of a functional equivalence validation task is found to be unsatisfiable, the maximum satisfiability core for this problem can be used to calculate a distance between the two planning domain models. Another ambitious research direction related to the idea of domains' functional distance is to investigate how to extend D-VAL to recommend corrections to the given planning domain models that make them functionally equivalent.

To deploy D-VAL, its implementation must be enhanced to perform preprocessing checks to test the

validity and suitability of the given domains to its scope before testing their functional equivalence.

Due to these limitations, our approach might not be readily helpful to some planning domain model engineering applications. However, this work certainly opens new research avenues by formalising and solving the problem of validating the functional equivalence of planning domain models.

5.3 Closing Remarks

A benefit of revisiting the area of planning domain model verification is the clarification of the miss conception of the power of under-constrained verification methods.

Under-constrained verification methods do not verify planning domain models for any planning problem, but they verify these models against partially-specified individual planning problems that leave planning goals open. Not specifying planning goals causes these verification methods to be susceptible to false positives. To address this issue, we proposed to perform planning domain model verification against fully specified planning problems. The novel approach presented in this thesis reduces the number of false positive counterexamples, but does not eliminate them. Nevertheless, this research forms the first step towards developing a planning domain model verification method that is robust against false positive counterexamples.

By presenting D-VAL, the first algorithm that can validate the functional equivalence of planning domain models independently from planning problems, this thesis achieves a significant milestone in the development of the area of Knowledge Engineering of Planning and Scheduling. D-VAL is proved to be sound when validating the functional equivalence of complex planning domain models and sound and complete when validating the functional equivalence of simple planning domain models.

This thesis has identified significant research gaps within the area of verification and validation of planning domain models. Moreover, this thesis has advanced this research area by proposing novel verification and validation methods that address specific aspects of these gaps. It is our aspiration that this work serves as an inspiration to drive the research towards closing these gaps and advancing the field of knowledge engineering for planning and scheduling.

APPENDIX A

PROOFS OF THEOREMS

A.1 Proof of Theorem 1 (See page 65)

Theorem 1. *Consider a set of objects Obj, two operators o_1 and o_2 from the same domain where the two operators have the same parameters. The operators o_1 and o_2 have the same effects, and the preconditions of o_2 are a subset (proper subset) of the preconditions of o_1 **iff** the reach set of o_1 is a subset (proper subset) of the reach set of o_2.*

$$(4.1)\quad (\text{Add}(o_1) = \text{Add}(o_2)) \wedge (\text{Del}(o_1) = \text{Del}(o_2)) \wedge (\text{Pre}(o_2) \subseteq \text{Pre}(o_1)) \textbf{\textit{ iff }} \Gamma(o_1, Obj) \subseteq \Gamma(o_2, Obj)$$

$$(4.2)\quad (\text{Add}(o_1) = \text{Add}(o_2)) \wedge (\text{Del}(o_1) = \text{Del}(o_2)) \wedge (\text{Pre}(o_2) \subset \text{Pre}(o_1)) \textbf{\textit{ iff }} \Gamma(o_1, Obj) \subset \Gamma(o_2, Obj)$$

Proof. First we prove the forward implication:

If o_2 has the same effects as the operator o_1, then o_2 and o_1 must reach the same end states from the same initial states. Moreover, if the set of the preconditions of o_2 is a subset (proper subset) of the set of the preconditions of o_1, then the set of the initial states that satisfies the preconditions of o_2 is a superset (proper superset) of the set of the initial states that satisfies the preconditions of o_1. As such, the set of the transitions that can be produced by applying o_1 is a subset (proper subset) of the set of the transitions that can be produced by applying o_2. Thus, we can infer that the reach set of o_1 is a subset (proper subset) of the reach set of o_2 for any set of objects.

Now we prove the backward implication. If the reach set of o_1 is a subset of the reach set of o_2, then any transition that can be produced by o_1 can also be produced by o_2. Thus, the reach set of o_2 contains the transitions that start from the states where the atoms of the preconditions of o_1 are the only true atoms; we call these transitions the core transitions of o_1. In such an initial state, an atom t can be either true or false. If t is true, then it must be in the preconditions of o_1. If applying o_1 to this state produces an

end state where t is also true, then t must not be in the delete effects of o_1. On the other hand, if applying o_1 produces a state where t is false, then t must be a delete effect of o_1. Since o_2 shares these transitions, if t is in the delete effects of o_1, it also must be in the delete effects of o_2, and if t is not in the delete effects of o_1, it also must not be in the delete effects of o_2. Hence, o_1 and o_2 have the same delete effects.

If t is false, then it must not be in the preconditions of o_1. If applying o_1 to this state produces an end state where t is also false, then t must not be in the add effects of o_1. On the other hand, if applying o_1 produces a state where t is true, then t must be an add effect of o_1. Since o_2 shares these transitions, if t is in the add effects of o_1, it also must be in the add effects of o_2 and if t is not in the add effects of o_1, it also must not be in the add effects of o_2. Thus, o_1 and o_2 have the same add effects.

Notice that o_2 cannot have any precondition that is not a precondition of o_1; otherwise, the reach set of o_2 would not contain the core transitions of o_1. Consequently, the reach set of o_1 would not be a subset of the reach set of o_2. Thus, the preconditions of o_2 must be either the same as or a subset of the preconditions of o_1. If the preconditions of o_2 are the same as the preconditions of o_1, then both o_2 and o_1 will have the same reach sets. On the other hand, if the set of the preconditions of o_2 is a subset (proper subset) of the set of the preconditions of o_1, then, according to the proved forward implication of this theorem, the reach set of o_2 is a subset (proper subset) of the reach set of o_1.

■

A.2 Proof of Theorem 2 (See page 74)

To prove this theorem, we first prove Lemma 4.1 in Appendix A.2.1 and then the main theorem is proven in Appendix A.2.2.

A.2.1 Proof of the Lemmas of Theorem 2

Lemma 4.1 (Simple Domain Reachability Lemma)**.** *Consider a set of objects Obj, two simple planning domain models, D_1 and D_2, and a bijective function $F_p : P_1 \twoheadrightarrow P_2$. We have*

$$\begin{aligned}(4.4)\quad \forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(\Gamma(F_p(o), Obj) = \Gamma(o', Obj) \wedge \\ |Primitive(O_1)| = |Primitive(O_2)|) \iff \Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj)\end{aligned}$$

To prove this lemma, the biconditional statement is broken into forward and backward implications which will be proved separately. For brevity, the set *Obj* is dropped from the reach set symbols in the proofs as it is common to all reach sets.

Proof. First, we prove the backward implication:

$$\begin{aligned}(A.1)\quad \Gamma(F_p(D_1)) = \Gamma(D_2) \implies \forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(\\ \Gamma(F_p(o)) = \Gamma(o') \wedge |Primitive(O_1)| = |Primitive(O_2)|)\end{aligned}$$

This implication will be spilt into two implications and each will be proved separately.

$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(\Gamma(F_p(o)) = \Gamma(o')) \tag{A.2}$$

(A.3)

$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(|Primitive(O_1)| = |Primitive(O_2)|)$$

First,we prove Equation (A.2). Since o is an operator in D_1, we have

$$\forall o \in Primitive(O_1)(\Gamma(o) \subseteq \Gamma(D_1)) \tag{A.4}$$

However, $\Gamma(F_p(D_1)) \subseteq \Gamma(D_2)$ because the antecedent of this implication states $\Gamma(F_p(D_1)) = \Gamma(D_2)$. Therefore,

$$\forall o \in Primitive(O_1)(\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \Gamma(F_p(o)) \subseteq \Gamma(D_2)) \tag{A.5}$$

Using the definition of the reach set of planning domain models, we infer

$$\forall o \in Primitive(O_1), \exists Seq' \subseteq SEQ_2(\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \Gamma(F_p(o)) \subseteq \bigcup_{seq' \in Seq'} (\Gamma(seq'))) \tag{A.6}$$

This can be rewritten as

$$\forall o \in Primitive(O_1), \exists Seq' \subseteq SEQ_2(\Gamma(F_p(D_1)) = \Gamma(D_2) \implies (\Gamma(F_p(o)) \subset \bigcup_{seq' \in Seq'} (\Gamma(seq'))) \tag{A.7}$$

$$\vee$$

$$(\Gamma(F_p(D_1)) = \Gamma(D_2) \implies (\Gamma(F_p(o)) = \bigcup_{seq' \in Seq'} (\Gamma(seq'))))$$

First, we will prove the left-hand side term of the disjunction in Equation (A.7) is only true if the reach set of $F_p(o)$ is equal to the reach set of a primitive operator from the set Seq'. Then, we will prove the right-hand term of the the disjunction in Equation (A.7) can only be true if Seq' has only one sequence and this sequence has only one operator, i.e., the reach set of any primitive operator from D_1 must be equal to the reach set of a primitive operator from D_2. This will prove Equation (A.2).

According to the left-hand side term of the disjunction in Equation (A.7), $\Gamma(F_p(o))$ is a proper subset of $\bigcup_{seq' \in Seq'}(\Gamma(seq'))$. Therefore, some elements from the set $\bigcup_{seq' \in Seq'}(\Gamma(seq'))$ are not in the reach set of the operator $F_p(o)$. However, for $\Gamma(F_p(D_1)) = \Gamma(D_2)$, any element from $\Gamma(D_2)$, including the elements in $\bigcup_{seq' \in Seq'}(\Gamma(seq')) \setminus \Gamma(F_p(o))$, must be in $\Gamma(F_p(D_1))$. Thus,

$$\exists Seq \subseteq SEQ(\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{seq' \in Seq'} (\Gamma(seq')) \setminus \Gamma(F_p(o)) \subseteq \bigcup_{seq \in Seq} (\Gamma(F_p(seq)))) \tag{A.8}$$

Following a similar discussion, we can also infer that if $\bigcup_{seq' \in Seq'}(\Gamma(seq')) \setminus \Gamma(F_p(o)) \neq \bigcup_{seq \in Seq}(\Gamma(F_p(seq)))$, then $\bigcup_{seq \in Seq}(\Gamma(F_p(seq))) \setminus \left(\bigcup_{seq' \in Seq'}(\Gamma(seq')) \setminus \Gamma(F_p(o))\right)$ must be a subset of $\bigcup_{seq'' \in Seq''}(\Gamma(seq''))$, where $Seq'' \in SEQ_2$ and $Seq'' \neq Seq'$.

(A.9)

$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{seq \in Seq} (\Gamma(F_p(seq))) \setminus \left(\bigcup_{seq' \in Seq'} (\Gamma(seq')) \setminus \Gamma(F_p(o))\right) \subseteq \bigcup_{seq'' \in Seq''} (\Gamma(seq''))$$

If the left-hand side of the containment relation in the consequent of Equation (A.9) is not equal to the right-hand side, then for the D_1 and D_2 to be functionally equivalent, the right-hand side minus the left-hand side must be also a subset of the image of the reach set of the domain that contains the reach set of the left-hand side set.

This series of containment relations will continue until eventually we reach a containment relation where the left-hand side is equal to the right-hand side. Otherwise, the reach set of D_1 cannot be equal to the reach set of D_2. Therefore, Equation (A.9) can be rewritten as:

(A.10)
$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{seq \in Seq} (\Gamma(F_p(seq))) \setminus \Big(\bigcup_{seq' \in Seq'} (\Gamma(seq')) \setminus \Gamma(F_p(o)) \Big) = \bigcup_{seq'' \in Seq''} (\Gamma(seq''))$$

The reach set of $\bigcup_{seq \in Seq}(\Gamma(F_p(seq)))$ can be expressed as

(A.11)
$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{seq \in Seq} (\Gamma(F_p(seq))) = \Big(\bigcup_{seq \in Seq} (\Gamma(F_p(seq))) \setminus \Big(\bigcup_{seq' \in Seq'} (\Gamma(seq')) \setminus \Gamma(F_p(o)) \Big) \Big) \cup \Big(\bigcup_{seq' \in Seq'} (\Gamma(seq')) \setminus \Gamma(F_p(o)) \Big)$$

Use Equation (A.10) to substitute $\big(\bigcup_{seq \in Seq}(\Gamma(F_p(seq))) \setminus \big(\bigcup_{seq' \in Seq'}(\Gamma(seq')) \setminus \Gamma(F_p(o))\big)\big)$ by $\bigcup_{seq'' \in Seq''}(\Gamma(seq''))$ in Equation (A.11), we get

(A.12)
$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{seq \in Seq} (\Gamma(seq)) = \Big(\bigcup_{seq' \in Seq'} (\Gamma(seq')) \setminus \Gamma(F_p(o)) \cup \bigcup_{seq'' \in Seq''} (\Gamma(seq'')) \Big)$$

From Equation (A.12), we have

(A.13)
$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \Big(\bigcup_{seq' \in Seq'} (\Gamma(seq')) \setminus \Gamma(F_p(o)) \Big) = \bigcup_{seq \in Seq} (\Gamma(seq)) \setminus \bigcup_{seq'' \in Seq''} (\Gamma(seq''))$$

The reach set of $\bigcup_{seq' \in Seq'}(\Gamma(seq'))$ can be expressed as

(A.14)
$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{seq' \in Seq'} (\Gamma(seq')) = \Big(\bigcup_{seq' \in Seq'} (\Gamma(seq')) \setminus \Gamma(F_p(o)) \Big) \cup \Gamma(F_p(o))$$

Use Equation (A.13) to substitute $\big(\bigcup_{seq' \in Seq'}(\Gamma(seq')) \setminus \Gamma(F_p(o))\big)$ with $\big(\bigcup_{seq \in Seq}(\Gamma(F_p(seq))) \setminus \bigcup_{seq'' \in Seq''}(\Gamma(seq''))\big)$, we get

(A.15)
$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{seq' \in Seq'} (\Gamma(seq')) = \Big(\bigcup_{seq \in Seq} (\Gamma(F_p(seq))) \setminus \bigcup_{seq'' \in Seq''} (\Gamma(seq'')) \Big) \cup \Gamma(F_p(o))$$

Depending on the cardinality of the set Seq', ($|Seq'|$), and the number of operators in each sequence seq', ($|seq'|$), $\bigcup_{seq' \in Seq'}(\Gamma(seq'))$ takes different shapes. These configurations are captured in the following list. In each state, Equation (A.15) can be rewritten to one of the following equations, Equations (A.16) to (A.18).

1. $|Seq'| = n \geq 1$ and $|seq'| = 1$: the set Seq' has one or more sequences and each sequence has one operator, o'_i, where $1 \leq i \leq n$.

   (A.16)
   $$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{i=1}^{n}(\Gamma(o'_i)) = \Big(\bigcup_{seq \in Seq} (\Gamma(F_p(seq))) \setminus \bigcup_{seq'' \in Seq''} (\Gamma(seq''))\Big) \cup \Gamma(F_p(o))$$

2. $|Seq'| \geq 1$ and $|seq'| > 1$: the set Seq' has two or more sequences and each sequence has two or more operators.

   (A.17)
   $$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{seq' \in Seq'} (\Gamma(seq')) = \Big(\bigcup_{seq \in Seq} (\Gamma(F_p(seq))) \setminus \bigcup_{seq'' \in Seq''} (\Gamma(seq''))\Big) \cup \Gamma(F_p(o))$$

3. $|Seq'| > 1$ and $|seq'| \geq 1$: the set Seq' has two or more sequences and each sequence has one or more operators, suppose the number of sequences with just one operator is n.

   (A.18)
   $$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \bigcup_{i=1}^{n}(\Gamma(o'_i)) \cup \bigcup_{seq' \in Seq' \setminus \{o'_i : 1 \leq i \leq n\}} (\Gamma(seq')) = \Big(\bigcup_{seq \in Seq} (\Gamma(F_p(seq))) \setminus \bigcup_{seq'' \in Seq''} (\Gamma(seq''))\Big) \cup \Gamma(F_p(o))$$

In each of these states, Seq and Seq'' can also have one of the three different configurations explained in the previous list.

In Equation (A.16), if $|Seq| \geq 1$ and $|seq_i| = 1$, then Seq has one or more operators. In this state, this equation means the image of the operator o share at least one transition with an operator o'_i and either the image of at least one operator in Seq share at least another transition with the operator o'_i or none of the images of any operator in Seq share any transition with the operator o'_i. Moreover, because if two operators share one or more transitions, the two operators must share at least one add effect and the same delete effects, the image of the operator o and the operator o'_i must have some shared add effects and the same delete effects. Furthermore, if the image of at least one operator in Seq share at least another transition with the operator o'_i, the image of these operators from Seq and the operator o'_i must have some shared add effects and the same delete effects. Thus, the operator o and these operators from Seq must have the same delete effects. However, the operator o and the operators of Seq are operators of the simple domain D_1, therefore, they must not have similar delete effects. This contradiction proves Equation (A.16) is false in this case. Note that the operator o and these operators from Seq do not have

to share any add effects because shared add effects between o and o'_i, and between some of the operators in Seq and o'_i are not necessary the same add effects.

On the other hand, if none of the images of any operator in Seq share any transition with the operator o'_i, then all transitions of o'_i must be in the image of the reach set of the operator o. Thus, the image of the reach set of o is superset of the reach set of o'_i. If the image of the reach set of o has some other transitions from other operators from the set $\{o'_i : 1 \leq i \leq n\}$ like o'_j where $1 \leq j \leq n$ and $j \neq i$, then o'_i and o'_j must have similar delete effects because both share some transitions with he image of the reach set of o. But o'_i and o'_j are primitive operators from D_2. Thus, they must not have similar delete effects. This contradictions falsifies this case.

The only remaining option is the image of the reach set of o is equal to the reach set of o'_i. This argument agrees with Equation (A.2).

In Equation (A.16), if $|Seq| \geq 1$ and $|seq_i| > 1$, then Seq has one or more sequence of operators and each has two or more operators. In this state, this equation means the operator o'_i shares at least one transition with the image of one sequence of operators from Seq. Any transition from such sequence must be consolidated from some transitions of primitive operators in D_1. Moreover, each of the images of the transitions of these primitive operators must be in the reach set of D_2. Thus, the consolidation of the images of these transitions in D_2 is also in the the reach set of o'_i. Therefore, there intersection between the reach set of o'_i and some of the sequences in D_2 is not empty. However, D_2 is simple domain, hence, the intersection between o'_i and any sequence from D_2 must be empty. This contradiction falsifies this case.

In Equation (A.16), when $|Seq| > 1$ and $|seq_i| \geq 1$, then Seq has two or more sequences of operators and each has two or more operators and some sequences with just one operator. All cases in this configuration are discussed in the proves of the previous states of this equation.

The discussion of Equation (A.17) follows the arguments provided for In Equation (A.16) when $|Seq| \geq 1$ and $|seq_i| > 1$. In the case of Equation (A.17), we have to compare the image of the reach set of the operator o with the reach set of the sequences in Seq'. Similarly, the discussion of Equation (A.18) is covered by the previous cases.

This discussions proved that if the two simple domains D_1 and D_2 are functionally equivalent and the image of the reach set of a primitive operator ,o, from D_1 is subset of the union of the reach set of the members of a set of sequences form D_2, then the image of the reach set of o must be equal to the reach set of a primitive operator from D_2. Thus, we have

$$\begin{aligned}(\text{A.19}) \quad \forall o \in Primitive(O_1), \exists Seq' \subseteq SEQ_2, \exists o' \in Seq' (\\ \Gamma(F_p(D_1)) = \Gamma(D_2) \wedge \Gamma(F_p(o)) \subset \bigcup_{seq' \in Seq'} (\Gamma(seq')) \implies \Gamma(F_p(o)) = \Gamma(o'))\end{aligned}$$

Next, we have to prove that if the two simple domains D_1 and D_2 are functionally equivalent and the image of the reach set of a primitive operator ,o, from D_1 is equal to the union of the reach set of the members of a set of sequences form D_2 ,Seq', then Seq' must have just one sequence of operators and this sequence must have just one operators.

According to the right-hand term of the the disjunction in Equation (A.7), we have

(A.20)
$$\forall o \in Primitive(O_1), \exists Seq' \subseteq SEQ_2(\Gamma(F_p(D_1)) = \Gamma(D_2) \implies (\Gamma(F_p(o)) = \bigcup_{seq' \in Seq'} (\Gamma(seq'))))$$

Similar to the previous discussion, we examine the different configurations of $\bigcup_{seq' \in Seq'}(\Gamma(seq'))$. However, in this discussion we split the case with $|Seq'| = n \geq 1$ and $|seq'| = 1$ into two different cases: $|Seq'| = 1$ and $|seq'| = 1$ and $|Seq'| = n > 1$ and $|seq'| = 1$. In each state, Equation (A.20) can be rewritten to one of the following equations, Equations (A.21) to (A.24).

1. $|Seq'| = 1$ and $|seq'| = 1$: the set Seq' has one sequence and this sequence has one operator, o'.

   (A.21) $$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \Gamma(F_p(o)) = \Gamma(o')$$

2. $|Seq'| = n > 1$ and $|seq'| = 1$: the set Seq' has two or more sequences and each sequence has one operator, o'_i, where $1 \leq i \leq n$.

   (A.22) $$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \Gamma(F_p(o)) = \bigcup_{i=1}^{n} (\Gamma(o'_i))$$

3. $|Seq'| \geq 1$ and $|seq'| > 1$: the set Seq' has two or more sequences and each sequence has two or more operators.

   (A.23) $$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \Gamma(F_p(o)) = \bigcup_{seq' \in Seq'} (\Gamma(seq'))$$

4. $|Seq'| > 1$ and $|seq'| \geq 1$: the set Seq' has two or more sequences and each sequence has one or more operators, suppose the number of sequences with just one operator is n.

   (A.24) $$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \Gamma(F_p(o)) = \bigcup_{i=1}^{n} (\Gamma(o'_i)) \cup \bigcup_{seq' \in Seq' \setminus \{o'_i : 1 \leq i \leq n\}} (\Gamma(seq'))$$

According Equation (A.22), the image of the reach set of the operator o share some transitions with a primitive operator o'_i and other transitions with another primitive operator, o'_j, from D_2. Therefore, o'_i and o'_j must have similar delete effects because both share some transitions with the image of the reach set of o (check the arguments in the prove of Equation (A.16) when $|Seq| \geq 1$ and $|seq_i| = 1$,). But o'_i and o'_j are primitive operators from D_2. Thus, they must not have similar delete effects. This contradictions falsifies this case.

Following the same discussion of Equation (A.16) when $|Seq| \geq 1$ and $|seq_i| > 1$ we can conclude Equations (A.23) and (A.24) are false. Since Equations (A.22) to (A.24) are false, then if the right-hand term of the the disjunction in Equation (A.7) is true, then Equation (A.21) must be true. Thus, we have

$$\text{(A.25)}\quad \forall o \in Primitive(O_1), \exists Seq' \subseteq SEQ_2, \exists o' \in Seq'(\Gamma(F_p(D_1)) = \Gamma(D_2) \wedge \Gamma(F_p(o)) = \bigcup_{seq' \in Seq'} (\Gamma(seq')) \implies \Gamma(F_p(o)) = \Gamma(o'))$$

Using Equation (A.7) with Equation (A.19) and Equation (A.25) we can conclude the prove of Equation (A.2)

To prove Equation (A.3), we start from the contrapositive of this equation:

$$\text{(A.26)}\quad \forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(|Primitive(O_1)| \neq |Primitive(O_2)| \implies \Gamma(F_p(D_1)) \neq \Gamma(D_2))$$

If two domains have a different number of primitive operators, then one domain will have at least one primitive operator more than the other domain. From the definition of primitive operators, we know the reach set of a primitive operator is not equal to the reach set of any other primitive operators in the same domain. Moreover, from the contrapositive form of the proved Equation (A.2), we know If the reach set of a primitive operator in one domain is not equal to the reach set of a primitive operator in the other domain, then the two domains do not have equal reach sets. Thus, if the reach set of each primitive operator from the domain with the fewer primitive operators is equal to the reach set of a primitive operator from the other domain, then the reach set of one primitive operator from the domain with the greater number of primitive operators cannot be equal to the reach set of any primitive operator from the other domain. Therefore, two domains with an unequal number of primitive operators are guaranteed to have unequal reach sets. This concludes the proof of Equation (A.3). Thus, we prove the backward implication of this lemma.

Second, we prove the forward implication:

$$\text{(A.27)}\quad \forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(\Gamma(F_p(o)) = \Gamma(o')) \wedge |Primitive(O_1)| = |Primitive(O_2)| \implies \Gamma(F_p(D_1)) = \Gamma(D_2)$$

Let $D_3 = (P_2, O_3)$ be a planning domain model with the same set of predicates as D_2, and the set of its primitive operators, $Primitive(O_3)$, satisfies the following conditions:

$$\text{(A.28)}\quad Primitive(O_3) \subseteq Primitive(O_2) \wedge \forall o \in Primitive(O_1), \exists o' \in Primitive(O_3)(\Gamma(F_p(o)) = \Gamma(o'))$$

From the definition of O_3, we have $Primitive(O_3) \subseteq Primitive(O_2)$; thus, any sequence of operators that can be made by the primitive operators of O_3 can also be made by the primitive operators of D_2. Since the reach set of a domain is equal to the union of the reach sets of its sequence of operators, we have $\Gamma(D_3) \subseteq \Gamma(D_2)$.

Furthermore, the definition of O_3 implies the image under F_p of the reach set of each primitive operator in D_1 is equal to the reach set of a primitive operator from D_3. This means the reach set of any sequence of operators made by the primitive operators of D_1 is equal to the reach set of a sequence of operators which can be made from the primitive operators of D_3. Thus we have $\Gamma(F_p(D_1)) \subseteq \Gamma(D_3)$. Therefore, $\Gamma(F_p(D_1)) \subseteq \Gamma(D_2)$.

From the antecedent of Equation (A.27), since the reach set of each primitive operator in O_1 is equal to the reach set of a primitive operator in O_2; because O_1 and O_2 have an equal number of primitive operators; and as the reach set of a primitive operator is not equal to the reach set of any other primitive operators in the same domain, we have the reach set of each primitive operator in O_2 is equal to the reach set of a primitive operator in O_1.

$$\forall o' \in Primitive(O_2), \exists o \in Primitive(O_1)(\Gamma(o') = F_p(o)) \tag{A.29}$$

Let $D_4 = (P_1, O_4)$ be a planning domain model with the same set of predicates as D_1, and the set of its primitive operators $Primitive(O_4)$ satisfies the following conditions:

$$\begin{gathered} Primitive(O_4) \subseteq Primitive(O_1) \\ \wedge \\ \forall o' \in Primitive(O_2), \exists o \in Primitive(O_4)(\Gamma(o') = \Gamma(F_P(o)) \end{gathered} \tag{A.30}$$

With the help of D_4 we can prove $\Gamma(D_2) \subseteq \Gamma(F_p(D_1))$ in a similar way as we proved $\Gamma(F_p(D_1)) \subseteq \Gamma(D_2)$. Therefore, we prove $\Gamma(F_p(D_1)) = \Gamma(D_2)$. Proving the forward implication concludes the proof of this lemma. ■

A.2.2 Proof of Theorem 2 using Lemma 4.1 (See page 74)

Theorem 2 (Simple Domains Reachability Theorem)**.** *Consider a set of objects Obj, two simple planning domain models, D_1 and D_2, a bijective function F_p from the predicates of D_1 to the predicates of D_2 with equal arities, and the relation R_{OM} that relates each primitive operator o in O_1 to a bijective predicate mapping f_p that makes the reach set of $f_p(o)$ equals to the reach set of a primitive operator from O_2. We have:*

$$\begin{aligned} (4.3)\quad \Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj) \textit{ iff } \exists R'_{om} \subseteq R_{OM}(\text{Domain}(R'_{om}) = primitive(O_1) \wedge \\ (F = \bigcup_{f_p \in Range(R'_{om})} f_p) \textit{ is a bijective function} \wedge |Primitive(O_1)| = |Primitive(O_2)|) \end{aligned}$$

Let $\mathbb{F}$ be the set of all bijective mappings between predicates of equal arity from the predicates of every primitive operator in D_1 to the predicates of every primitive operator in D_2.

$$\begin{aligned} \mathbb{F} = \{ f_p | f_p : Predicates(o) \nrightarrow Predicates(o') \text{ where } o \in Primitive(O_1), o' \in Primitive(O_2) \\ \text{and if } f_p(p) = p' \text{ then } Arity(p) = Arity(p') \} \end{aligned}$$

Let R_{OM} be a relation between primitive operators from D_1 and predicate mappings from $\mathbb{F}$. A primitive operator o from D_1 is related to a mapping f_p from $\mathbb{F}$ by R_{OM} means there exists a primitive operator o' from D_2 such that the reach set of o is equal to the reach set of o' under the mapping f_p.

$$R_{OM} = \{(o, f_p) \in Primitive(O_1) \times \mathbb{F} \mid \exists o' \in Primitive(O_2),\ \Gamma(f_p(o), Obj) = \Gamma(o', Obj)\} \tag{A.31}$$

Proof. To prove this theorem, the biconditional statement is broken into forward and backward implications which will be proved separately. First, we prove the forward implication:

$$\Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj) \implies \exists R'_{om} \subseteq R_{OM} (Domain(R'_{om}) = Primitive(O_1) \\ \wedge (F = \bigcup_{f_p \in Range(R'_{om})} f_p) \text{ is a bijective function}) \wedge |Primitive(O_1)| = |Primitive(O_2)|$$

For brevity, the set *Obj* is dropped from the reach set symbols in the proofs as it is common to all reach sets. From the antecedent of this implication and from Lemma 4.1, we have:

$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(\Gamma(F_p(o)) = \Gamma(o')) \\ \wedge |Primitive(O_1)| = |Primitive(O_2)| \tag{A.32}$$

This proves if the reach set of $F_p(D_1)$ is equal to the reach set of D_2, then both D_1 and D_2 have an equal number of primitive operators.

From the definition of the relation R_{OM} (Equation (A.31)) and Equation (A.32), we can infer

$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \forall o \in Primitive(O_1),\ (o, F_p) \in R_{OM} \tag{A.33}$$

Let the relation R'_{om} be defined as follows.

$$R'_{om} = \{(o, F_p) | \forall o \in Primitive(O_1)\} \tag{A.34}$$

From Equation (A.33), we deduce that R'_{om} is a subset of R_{OM}. Furthermore, from the definition of R'_{om} in Equation (A.34), we have Domain$(R'_{om}) = Primitive(O_1)$. Thus, we conclude

$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \exists R'_{om} \subseteq R_{OM} \wedge Domain(R'_{om}) = Primitive(O_1) \tag{A.35}$$

From Equation (A.34), we have

$$Range(R'_{om}) = F_p \tag{A.36}$$

So,

$$F = \bigcup_{f_p \in Range(R'_{om})} f_p = F_p \tag{A.37}$$

From the antecedent of the forward implication, we know F_p is a bijective function, therefore, based on Equation (A.37), we conclude $F = \bigcup_{f_p \in Range(R'_{om})} f_p$ is also a bijective function.

This conclusion and Equation (A.35) proves

$$\Gamma(F_p(D_1)) = \Gamma(D_2) \implies \exists R'_{om} \subseteq R_{OM}(Domain(R'_{om}) = Primitive(O_1) \wedge (F = \bigcup_{f_p \in Range(R'_{om})} f_p) \text{ is a bijective function}) \tag{A.38}$$

The proofs of Equation (A.38) and Equation (A.32) complete the proof of the forward implication of this theorem.

Second, we prove the backward implication:

$$\exists R'_{om} \subseteq R_{OM}(Domain(R'_{om}) = Primitive(O_1) \wedge (F = \bigcup_{f_p \in Range(R'_{om})} f_p) \text{ is a bijective function}) \wedge |Primitive(O_1)| = |Primitive(O_1)| \implies \Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj) \tag{A.39}$$

The antecedent $\exists R'_{om} \subseteq R_{OM}(Domain(R'_{om}) = Primitive(O_1))$ implies

$$\forall o \in Primitive(O_1), \exists f \in \mathbb{F}, (o, f) \in R'_{om} \tag{A.40}$$

From the definition of R_{OM} in Equation (A.31) and Equation (A.40), we have

$$\forall o \in Primitive(O_1), \exists f \in Range(R'_{om}), \exists o' \in Primitive(O_2)(\Gamma(f(o)) = \Gamma(o')) \tag{A.41}$$

The antecedent of the backward implication, Equation (A.39), defines F as the union of all mappings in Range(R'_{om}) ($F = \bigcup_{f_p \in Range(R'_{om})} f_p$). Hence, we have

$$\forall f \in Range(R'_{om})(f \subseteq F) \tag{A.42}$$

Moreover, because F is a bijective function according to the antecedent of this backward implication, we can replace f with F in Equation (A.41). Then, this equation can be written as

$$\forall o \in Primitive(O_1), \exists o' \in Primitive(O_2)(\Gamma(F(o)) = \Gamma(o')) \tag{A.43}$$

The domain of every mapping in Range(R'_{om}) is the set of the predicates of one operator from $Primitive(O_1)$.

$$\forall f \in Range(R'_{om})(Domain(f) = \{p | p \in Predicates(o) \text{ where } (o, f) \in R'_{om}\}) \tag{A.44}$$

Since F is the union of all mappings in Range(R'_{om}), the domain of F is equal to the union of the domains of all mappings in Range(R'_{om}).

$$Domain(F) = \bigcup_{f_p \in Range(R'_{om})} Domain(f_p) \tag{A.45}$$

From Equation (A.40), Equation (A.44), and Equation (A.45), we conclude

$$Domain(F) = Predicates(Primitive(O_1)) \tag{A.46}$$

However, since primitive operators are the only source of the functionality of their domains, the predicates of a domain are the union of the predicates of its primitive operators. Thus, we have Predicates($Primitive(O_1)$) = P_1. Hence,

$$Domain(F) = P_1 \tag{A.47}$$

The range of every mapping in Range(R'_{om}) is the set of the predicates of an operator o' from $Primitive(O_2)$ such that the reach set of an operator from $Primitive(O_1)$ under the given mapping is equal to the reach set of the primitive operator o'. The following equation formally defines the range of a mapping from Range(R'_{om}).

$$\begin{aligned} &\forall f \in Range(R'_{om})(Range(f) = \{p | p \in Predicates(o') \\ &\text{where } o' \in Primitive(O_2) \text{ if } \exists o \in Primitive(O_1)(\Gamma(f(o)) \subseteq \Gamma(o')\}). \end{aligned} \tag{A.48}$$

Since F is the union of all mappings in Range(R'_{om}) and F is a bijective function, the range of F is equal to the union of the ranges of all mappings in Range(R'_{om}).

$$Range(F) = \bigcup_{f_p \in Range(R'_{om})} Range(f_p) \tag{A.49}$$

From Equation (A.48) and Equation (A.49), we conclude

$$Range(F) \subseteq Predicates(Primitive(O_2)) \tag{A.50}$$

However, since primitive operators are the only source of the functionality of their domains, the predicates of a domain are the union of the predicates of its primitive operators. Thus, we have Predicates($Primitive(O_2)$) = P_2. Hence,

$$Range(F) \subseteq P_2 \tag{A.51}$$

The function F is a bijective function because it is the union of a set of bijective functions. Hence, $|Domain(F)| = |Range(F)|$. From Equation (A.47), we have $|Domain(F)| = |P_1|$. So, $|Range(F)| = |P_1|$. We know $|P_1| = |P_2|$ from the assumption of this theorem, therefore $|Range(F)| = |P_2|$. Thus, according to Equation (A.51) we infer

$$Range(F) = P_2 \tag{A.52}$$

We have F is a bijective function from P_2 to P_1 as per Equation (A.47) and Equation (A.52). Furthermore, we know from the antecedent of the backward implication that $|Primitive(O_1)| = |Primitive(O_2)|$. In addition, Equation (A.43) proves the image of the reach set of each primitive operator in D_1 under F is equal to the reach set of a primitive operator in D_2. Therefore, with the help of Lemma 4.1, we can deduce $\Gamma(F_p(D_1)) = \Gamma(D_2)$. With this statement we concludes the proof of the backward implication of this theorem. Thus, it completes the proof of the theorem. ■

A.3 Proof of Theorem 3 (See page 74)

Theorem 3 (Operators Structure Reach Set Theorem)**.** *Consider a set of objects Obj, two operators o and o', and a bijective function $f_p : Predicates(o) \twoheadrightarrow Predicates(o')$ such that f_p maps the predicates of o to those of o' with equal arities. Let $f_p(o)$ be the image of o using f_p to substitute the predicates of o with those of o' with equal arities. For any set of objects, the reach set of $f_p(o)$ is equal to the reach set of o' **iff** there exists a bijective mapping f_t from the atoms of o to the atoms of o' such that f_t and f_t^{-1} satisfy the following conditions:*

1. *Atoms in the preconditions, delete effects and add effects of one operator must be mapped to atoms in the preconditions, delete effects and add effects of the other operator, respectively;*
2. *Atoms in one operator must be mapped to atoms in the other operator with equal arity;*
3. *Atoms with the same predicate p in one operator must be mapped to atoms with some predicate p' in the other operator; and*
4. *Atoms with a shared variable v in one operator must be mapped to atoms with some shared variable v' in the other operator such that the positions of v and v' in the parameters of the mapped atoms are equal.*

Here we provide the formal form of the theorem. The part related to f_t^{-1} has been omitted to avoid repetition.

$$\forall Obj, \exists f_p : \text{Predicates}(o) \twoheadrightarrow \text{Predicates}(o') \text{ where if } f_p(p) = p' \text{ then } \text{Arity}(p) = \text{Arity}(p') \,($$

$$\Gamma(f_p(o), Obj) = \Gamma(o', Obj) \iff$$

$$\exists f_t : Atoms(o) \twoheadrightarrow Atoms(o') \,($$

$$\forall t \in \text{Pre}(o)\ (\exists t' \in \text{Pre}(o') : f_t(t) = t') \wedge$$

$$\forall t \in \text{Del}(o)\ (\exists t' \in \text{Del}(o') : f_t(t) = t') \wedge$$

$$\forall t \in \text{Add}(o)\ (\exists t' \in \text{Add}(o') : f_t(t) = t') \wedge$$

$$\forall t \in Atoms(o)\ (\exists t' \in Atoms(o') : f_t(t) = t' \wedge Arity(t) = Arity(t')) \wedge$$

$$\forall t_1, t_2 \in Atoms(o)\ (Pred(t_1) = Pred(t_2) \rightarrow \exists t'_1, t'_2 \in Atoms(o')\ (Pred(t'_1) = Pred(t'_2)$$

$$\wedge (f_t(t_1) = t'_1 \wedge f_t(t_2) = t'_2) \vee (f_t(t_2) = t'_1 \wedge f_t(t_1) = t'_2))) \wedge$$

$$\forall t_1, t_2 \in Atoms(o), \forall v_1 \in Var(t_1), \forall v_2 \in Var(t_2)\ (v_1 = v_2 \rightarrow$$

$$\exists t'_1, t'_2 \in Atoms(o'), \exists v'_1 \in Var(t'_1), \exists v'_2 \in Var(t'_2)\ (v'_1 = v'_2$$

$$\wedge ((f_t(t_1) = t'_1 \wedge f_t(t_2) = t'_2 \wedge Position(v_1, t_1) = Position(v'_1, t'_1) \wedge Position(v_2, t_2) = Position(v'_2, t'_2))$$

$$\vee (f_t(t_2) = t'_1 \wedge f_t(t_1) = t'_2 \wedge Position(v_2, t_2) = Position(v'_1, t'_1) \wedge Position(v_1, t_1) = Position(v'_2, t'_2))))))$$

A.3.1 Definitions of Important Relations Between Atom Mappings

In the proof of this theorem, we will construct a mapping between the atoms of two operators from the mappings between their preconditions, delete effects and add effects. The mappings relations defined in this section facilitate describing the consistency conditions for unifying the mappings of the preconditions, delete effects and add effects of the two operators.

The first mapping relation states that two atom mappings have predicate-consistent relation, or they are predicate-consistent mappings if they map the set of atoms that share a common predicate p in one operator to the set of atoms that share some predicate p' in the second operator.

For example, Let t_1 and t_2 be two atoms in the preconditions and delete effects of an operator m respectively, and let t_1 and t_2 to have the same predicate p. Let f_{t-pre} be a mapping that maps the preconditions of an operator m to the preconditions of an operator o and f_{t-del} a mapping which maps the delete effect of m to the delete effects of o. We say f_{t-pre} and f_{t-del} are predicate-consistent mappings if f_{t-pre} maps t_1 to the atom t'_1 in the preconditions of o and f_{t-del} maps t_2 to the atom t'_2 in the delete effects of o such that t'_1 and t'_2 share a common predicate p'.

Definition A.1. Two bijective atom mappings f and g are ***predicate-consistent mappings***, $f \approx_{pred} g$, **iff** they map atoms with a shared predicate p in their domains to atoms with some predicate p' in their ranges.

$$f \approx_{pred} g \iff \forall t_1 \in Domain(f), \forall t_2 \in Domain(g)\ (Pred(t_1) = Pred(t_2) \rightarrow Pred(f(t_1)) = Pred(g(t_2)))$$

Another useful relation is the variable-order-consistent relation. This relation requests two different mappings to map atoms with a shared variable v in one operator to atoms that also share some variable v' in the other operator. Furthermore, this relation has an additional condition that necessitates the positions of the variable v in the parameters of the atoms of the first operator to be equal to the positions of the variable v' in the parameters of the atoms of the second operator. The function of this relation is to guarantee that when two different mappings are unified, the resulting mapping is consistent with regard to the shared variables and their orders.

For example, Let t_1 and t_2 be two atoms in the preconditions and delete effects of an operator m respectively, and let v be the second variable in the parameters of t_1 and the first variable in the parameters of t_2. We say f_{t-pre} and f_{t-del} are variable-order-consistent mappings if f_{t-pre} maps t_1 to the atom t'_1 in the preconditions of o and f_{t-del} maps t_2 to the atom t'_2 in the delete effects of o such that v' is the second variable in the parameters of t'_1 and v' is also the first variable in the parameters of t'_2.

Definition A.2. Two bijective atom mappings f and g are ***variable-order-consistent mappings***, $f \approx_{var} g$, **iff** they map atoms with a shared variable in their domains to atoms with some shared variable in their ranges such that the positions of the variable in the parameters of the atoms in the domains of the mappings are equal to the positions of the variable in the parameters of the atoms in the ranges of the mappings.

$$f \approx_{var} g \iff \forall t_1 \in Domain(f), \forall t_2 \in Domain(g)\ ($$

$$\forall v_1 \in Var(t_1), \forall v_2 \in Var(t_2)\ (v_1 = v_2 \rightarrow$$
$$\exists v'_1 \in Var(f(t_1)), \exists v'_2 \in Var(g(t_2))\ (v'_1 = v'_2$$
$$\wedge Position(v_1, t_1) = Position(v'_1, f(t_1)) \wedge Position(v_2, t_2) = Position(v'_2, g(t_2))\ )))$$

To simplify the process of referring to the two previous atom mapping relations, we define the atom-consistent relation which encompasses them both.

Definition A.3. Two bijective atom mappings f and g are ***atom-consistent mappings*** **iff** they are predicate and variable-order consistent mappings.

A.3.1.1 Properties of Atom Mapping Relations

Predicate-consistent, variable-order-consistent and atom-consistent mappings symmetric and transitive relations. Therefore proving the following formula:

$$\exists f_{t-pre}, \exists f_{t-add}, \exists f_{t-del}((f_{t-pre} \approx_{pred} f_{t-del}) \wedge (f_{t-del} \approx_{pred} f_{t-add}))$$

Implies:

$$\exists f_{t-pre}, \exists f_{t-add}, \exists f_{t-del}((f_{t-pre} \approx_{pred} f_{t-add}) \wedge (f_{t-add} \approx_{pred} f_{t-pre})$$
$$\wedge$$
$$(f_{t-del} \approx_{pred} f_{t-pre}) \wedge (f_{t-add} \approx_{pred} f_{t-del}))$$

A.3.2 Operators Structure Reach Set Theorem

Theorem 3 (Operators Structure Reach Set Theorem)**.** *Consider a set of objects Obj, two operators o and o', and a bijective function $f_p : Predicates(o) \twoheadrightarrow Predicates(o')$ such that f_p maps the predicates of o to those of o' with equal arities. Let $f_p(o)$ be the image of o using f_p to substitute the predicates of o with those of o' with equal arities. For any set of objects, the reach set of $f_p(o)$ is equal to the reach set of o' **iff** there exists a bijective mapping f_t from the atoms of o to the atoms of o' such that f_t and f_t^{-1} satisfy the following conditions:*

1. *Atoms in the preconditions, delete effects and add effects of one operator must be mapped to atoms in the preconditions, delete effects and add effects of the other operator, respectively;*
2. *Atoms in one operator must be mapped to atoms in the other operator with equal arity;*
3. *Atoms with the same predicate p in one operator must be mapped to atoms with some predicate p' in the other operator; and*

4. *Atoms with a shared variable v in one operator must be mapped to atoms with some shared variable v' in the other operator such that the positions of v and v' in the parameters of the mapped atoms are equal.*

Here we provide the formal form of the theorem. The part related to f_t^{-1} has been omitted to avoid repetition.

$$\forall Obj, \exists f_p : \text{Predicates}(o) \twoheadrightarrow \text{Predicates}(o') \text{ where if } f_p(p) = p' \text{ then } \text{Arity}(p) = \text{Arity}(p') \text{ (}$$

$$\Gamma(f_p(o), Obj) = \Gamma(o', Obj) \iff$$

$$\exists f_t : Atoms(o) \twoheadrightarrow Atoms(o') \text{ (}$$

$$\forall t \in \text{Pre}(o) \ (\exists t' \in \text{Pre}(o') : f_t(t) = t') \wedge$$

$$\forall t \in \text{Del}(o) \ (\exists t' \in \text{Del}(o') : f_t(t) = t') \wedge$$

$$\forall t \in \text{Add}(o) \ (\exists t' \in \text{Add}(o') : f_t(t) = t') \wedge$$

$$\forall t \in Atoms(o) \ (\exists t' \in Atoms(o') : f_t(t) = t' \wedge Arity(t) = Arity(t')) \wedge$$

$$\forall t_1, t_2 \in Atoms(o) \ (Pred(t_1) = Pred(t_2) \rightarrow \exists t'_1, t'_2 \in Atoms(o') \ (Pred(t'_1) = Pred(t'_2)$$

$$\wedge \ (f_t(t_1) = t'_1 \wedge f_t(t_2) = t'_2) \vee (f_t(t_2) = t'_1 \wedge f_t(t_1) = t'_2))) \wedge$$

$$\forall t_1, t_2 \in Atoms(o), \forall v_1 \in Var(t_1), \forall v_2 \in Var(t_2) \ (v_1 = v_2 \rightarrow$$

$$\exists t'_1, t'_2 \in Atoms(o'), \exists v'_1 \in Var(t'_1), \exists v'_2 \in Var(t'_2) \ (v'_1 = v'_2$$

$$\wedge \ ((f_t(t_1) = t'_1 \wedge f_t(t_2) = t'_2 \wedge Position(v_1, t_1) = Position(v'_1, t'_1) \wedge Position(v_2, t_2) = Position(v'_2, t'_2))$$

$$\vee \ (f_t(t_2) = t'_1 \wedge f_t(t_1) = t'_2 \wedge Position(v_2, t_2) = Position(v'_1, t'_1) \wedge Position(v_1, t_1) = Position(v'_2, t'_2))))))$$

Proof. To prove the bidirectional implication in this theorem, we prove its forward and backward implications. We will provide the proofs of the forward and backward implications with regard to f_t. In the same way, we can prove the forward and backward implications with respect to f_t^{-1}, but this proof is not provided to avoid repetition. In this proof, we refer to the second condition of this theorem as the "arity condition", the third condition as the "predicate condition", and the last condition as the "variable-order condition".

A.3.3 The proof of the Forward Implication

A.3.3.1 Proof Sketch

To prove the forward implication, we will show that if m and o have equal reach sets under a bijective mapping f_p between the predicates of o with those of m with equal arity, then there will exist an atom mapping f_t that respects the conditions of this theorem.

Starting from the antecedent of the forward implication, we will prove the existence of three bijective mappings, f_{t-pre},f_{t-del} and f_{t-add}, from the atoms of the preconditions, delete effects and add effects of m to the atoms of the preconditions, delete effects and add effects of o respectively, such that these mappings respect the arity, predicate, and variable-order conditions of this theorem. Then, we will prove that these three mappings are atom-consistent mappings with respect to each other. Finally, we will show because f_{t-pre},f_{t-del} and f_{t-add} respect the arity, predicate, and variable-order conditions and they are atom-consistent mappings, they can be used to define an atom mapping f'_t that satisfies the conditions of f_t. Hence proving the existence of f_t starting from the antecedent of the forward implication.

A.3.3.2 The Proof of the Existence of Preconditions, Delete Effects and Add Effects Mappings that Satisfy the Arity, Predicate, and Variable-order Conditions

The antecedent of the forward implication, $\Gamma(f_p(m)) = \Gamma(o)$, can be rewritten as follows.

$$\forall(s_i, s_j) \in \Gamma(o) \rightarrow (s_i, s_j) \in \Gamma(f_p(m)) \ \wedge \forall(s_i, s_j) \in \Gamma(f_p(m)) \rightarrow (s_i, s_j) \in \Gamma(o) \tag{A.53}$$

$(s_i, s_j) \in \Gamma(o)$ means there is an action instantiated from o which is applicable in s_i and can reach s_j. Similarly, $(s_i, s_j) \in \Gamma(f_p(m))$ means there exists an action instantiated from $f_p(m)$ which is also applicable in s_i and can reach s_j. So, both $f_p(m)$ and o can produce actions applicable in the same states. For two operators to be able to instantiate actions applicable in the same states, the two operators must have the same preconditions up, but not necessarily, to the variables' names.

This implies that for every atom in the preconditions of $f_p(m)$, there is an atom in the preconditions of o such that the two atoms have the same predicate and the same order of variables with respect to the other atoms in their perspective preconditions.

This means the number of atoms of each arity and the number of atoms of each predicate are equal in the preconditions of the two operators. Furthermore, for every two atoms in the preconditions of $f_p(m)$ that have a shared variable v, there are two atoms in the preconditions of o that share some variable v' and the positions of v in the parameters of its atoms are equal to the positions of v' in the parameters of its atoms.

The operator $f_p(m)$ is produced by renaming the predicates of m using the predicate mapping f_p which does not change the number of predicates, atoms or the variables of m. So, $f_p(m)$ and m differ by only the name of the predicates. Therefore, the number of atoms of each arity is equal in the preconditions of $f_p(m)$ and m. Furthermore, the number of atoms that share some predicate is also equal in the preconditions of $f_p(m)$ and m. Moreover, $f_p(m)$ and m have the same variables.

The conclusion is that the number of atoms of each arity and the number of atoms of each predicate are equal in the preconditions of m and o. Furthermore, for every two atoms in o that have a shared variable v, there are two atoms in the preconditions of m that share some variable v' and the positions of v in the parameters of its atoms are equal to the positions of v' in the parameters of its atoms. This proves the existence of a bijective mapping from the atoms of the preconditions of m to the atoms of the preconditions of o that satisfies the following conditions:

1. Atoms from the preconditions of m are mapped to atoms in the preconditions of o of equal arity;
2. Atoms in the preconditions of m that share a predicate p must be mapped to atoms in the preconditions of o that share some predicate p'; and
3. Atoms in the preconditions of m that share a variable v must be mapped to atoms in the preconditions of o that share some variable v' and the position of v in the parameters of its atoms is equal to the position of v' in the parameters of its atoms.

Formally,

$$\exists f_{t-pre} : \text{Pre}(m) \twoheadrightarrow \text{Pre}(o)) \; ($$

$$\forall t \in \text{Pre}(m) \; (\exists t' \in \text{Pre}(o) : f_{t-pre}(t) = t' \wedge Arity(t) = Arity(t'))$$

$$\wedge$$

$$\forall t_1, t_2 \in \text{Pre}(m) \; (Pred(t_1) = Pred(t_2) \rightarrow \exists t'_1, t'_2 \in \text{Pre}(o) \; (Pred(t'_1) = Pred(t'_2)$$

$$\wedge (f_{t-pre}(t_1) = t'_1 \wedge f_{t-pre}(t_2) = t'_2) \vee (f_{t-pre}(t_2) = t'_1 \wedge f_{t-pre}(t_1) = t'_2)))$$

$$\wedge$$

$$\forall t_1, t_2 \in Atoms(m), \forall v_1 \in Var(t_1), \forall v_2 \in Var(t_2) \; (v_1 = v_2 \rightarrow$$

$$\exists t'_1, t'_2 \in Atoms(o), \exists v'_1 \in Var(t'_1), \exists v'_2 \in Var(t'_2) \; (v'_1 = v'_2$$

$$\wedge((f_{t-pre}(t_1) = t'_1 \wedge f_{t-pre}(t_2) = t'_2 \wedge Position(v_1, t_1) = Position(v'_1, t'_1) \wedge Position(v_2, t_2) = Position(v'_2, t'_2))$$

$$\vee(f_{t-pre}(t_2) = t'_1 \wedge f_{t-pre}(t_1) = t'_2 \wedge Position(v_2, t_2) = Position(v'_1, t'_1) \wedge Position(v_1, t_1) = Position(v'_2, t'_2))))))$$

We have concluded from the proposition (A.53) that both $f_p(m)$ and o can produce actions that are applicable in the same states. Furthermore, from this proposition, we can also conclude that $f_p(m)$ and o can produce actions that can reach the same end states from equal states. For two operators to be able to instantiate actions that can reach the same states from equal states, the two operators must have the same preconditions, add effects and delete effects up, but not necessarily, to the variables name. With a similar discussion as the one used to prove the existence of f_{t-per}, we can prove the existence of:

1. A bijective mapping between the atoms of the add effects of m and o, $f_{t-add} : \text{Add}(m) \twoheadrightarrow \text{Add}(o))$, and

2. A bijective mapping between the atoms of the delete effects of m and o ,f_{t-del} : Del(m) $\twoheadrightarrow$ Del(o)).

Such that these mappings satisfy the following conditions:

1. Atoms in Add(m) (Del(m)) are mapped to atoms in Add(o) (Del(o)) of equal arity;
2. Atoms in Add(m) (Del(m)) that share a predicate p must be mapped to atoms in Add(o) (Del(o)) that share some predicate p'; and
3. Atoms in Add(m) (Del(m)) that share a variable v must be mapped to atoms in Add(o) (Del(o)) that share some variable v' and the positions of v in the parameters of its atoms is equal to the positions of v' in the parameters of its atoms.

A.3.3.3 Proving the Preconditions, Delete effects and Add effects Mappings are Atom-consistent Mappings

Let's call the set of mappings that satisfy the conditions of f_{t-pre} as Sf_{t-pre}. This set is not empty as we have proved. Similarly, we define the non-empty sets of mappings Sf_{t-add} and Sf_{t-del} which satisfy the conditions of f_{t-add} and f_{t-del} respectively. Now, we have to prove that f_{t-per}, f_{t-add} and f_{t-del} are atom-consistent mappings with respect to each other where $f_{t-pre} \in Sf_{t-pre}$, $f_{t-add} \in Sf_{t-add}$, and $f_{t-del} \in Sf_{t-del}$. Formally, we have to prove this formula:

$$\exists f_{t-pre} \in Sf_{t-pre}, \exists f_{t-add} \in Sf_{t-add}, \exists f_{t-del} \in Sf_{t-del} \,($$

$$(f_{t-pre} \approx_{pred} f_{t-del}) \wedge (f_{t-del} \approx_{pred} f_{t-add})$$

$$\wedge$$

$$(f_{t-pre} \approx_{var} f_{t-del}) \wedge (f_{t-del} \approx_{var} f_{t-add})$$

Note, since $\approx_{pred}$ and $\approx_{var}$ are symmetric and transitive relations, we do not need to have all possible combinations of f_{t-pre}, f_{t-del}, and f_{t-add} with $\approx_{pred}$ and $\approx_{var}$ to express f_{t-pre}, f_{t-del}, and f_{t-add} are atom-consistent mappings with respect to each other.

We will prove that if m and o have equal reach sets under a bijective mapping between the predicates of o and those of m with equal arity, then f_{t-per}, f_{t-add} and f_{t-del} are atom-consistent mappings. For this purpose, we will prove the contrapositive form of this implication. If f_{t-per}, f_{t-add} and f_{t-del} are not atom-consistent mappings then m and o do not have equal reach sets under any bijective between the predicates of o and those of m with equal arity. The contrapositive form is formally defined as follows:

$$\forall f_p : Predicates(m) \twoheadrightarrow Predicates(o) \text{ where if } f_p(p) = p' \text{ then } Arity(p) = Arity(p')($$

$$\forall f_{t-pre} \in Sf_{t-pre}, \forall f_{t-add} \in Sf_{t-add}, \forall f_{t-del} \in Sf_{t-del} \,($$

$$(f_{t-pre} \not\approx_{pred} f_{t-del}) \vee (f_{t-del} \not\approx_{pred} f_{t-add})$$

$$\vee$$

$$(f_{t-pre} \not\approx_{var} f_{t-del}) \vee (f_{t-del} \not\approx_{var} f_{t-add}))$$

$$\rightarrow$$

$$\Gamma(f_p(m)) \neq \Gamma(o))$$

To prove this implication, we have to prove that every element in the antecedent implies the consequent. First, we prove $\forall f_{t-pre} \in Sf_{t-pre}, \forall f_{t-del} \in Sf_{t-del}\ (f_{t-pre} \not\approx_{pred} f_{t-del} \rightarrow \Gamma(f_p(m)) \neq \Gamma(o))$. If there are no precondition mappings and delete effect mappings from the atoms of m to the atoms of o such that these mappings are predicate-consistent, then there will be two atoms that share the same predicate p, one in the preconditions and the other in the delete effects of m, that cannot be mapped to atoms that share some predicate p' in the preconditions and delete effects of o respectively. This means that under any bijective predicate mapping f_p, $f_p(m)$ will be different from o by either a precondition or a delete effect. If $f_p(m)$ differ from o by a precondition, then the actions produced from $f_p(m)$ and o will be applicable in different set of states. On the other hand, if $f_p(m)$ differs from o by a delete effect, then the actions produced from $f_p(m)$ and o will not reach the same states. Therefore, $f_p(m)$ and o will have different reach sets under any bijective mapping between the predicates of o and those of m with equal arity. This concludes the proof of this sub-formula. Similarly, we prove the correctness of the following formulas:

$$\forall f_{t-del} \in Sf_{t-del}, \forall f_{t-add} \in Sf_{t-add}\ (f_{t-del} \not\approx_{pred} f_{t-add} \rightarrow \Gamma(f_p(m)) \neq \Gamma(o))$$

Now we have to prove $\forall f_{t-pre} \in Sf_{t-pre}, \forall f_{t-del} \in Sf_{t-del}\ (f_{t-pre} \not\approx_{var} f_{t-del} \rightarrow \Gamma(f_p(m)) \neq \Gamma(o)$. There are two possible reasons for preconditions mapping and delete effects mapping to be not variable-order-consistent mappings. Either the cardinality of the sets of atoms that share some variable v in the preconditions and delete effects of m is greater than the cardinality of the sets of atoms that share some variable v' in the preconditions and delete effects of o. In this case, there will be two atoms t_1 and t_2 that share the same variable v, one in the preconditions and the other in the delete effects of m, but o does not have two atoms that share some variable v' in the preconditions and delete effects of o and can be mapped to the atoms t_1 and t_2. Consequently, under any bijective predicate mapping f_p, $f_p(m)$ and o will produce different actions for any set of objects. Thus, the actions produced from $f_p(m)$ will have a precondition and a delete effect that have the same object as a parameter, whereas the same precondition and delete effect in the actions produced from o will not have the same object as well. Thus $f_p(m)$ differs from o by either a precondition or a delete effect.

The other reason for not having precondition and delete effect mappings that are variable-order-consistent is as follows. If the number of atoms with a shared variable v in each of the preconditions and delete effects of m is equal to the number of atoms with a shared variable v' in each of the preconditions and delete effects of o respectively, then the order of the variables v and v' in the atoms of m and o must be not the same. Consequently, under any bijective predicate mapping f_p, $f_p(m)$ and o will produce different actions for any set of objects. In the actions produced from $f_p(m)$, there will be a precondition and a delete effect that have the same object as a parameter in specific positions, whereas the same

precondition and delete effect in the actions produced from o will have the same object as well but not in the same positions as the actions produced from $f_p(m)$. Thus $f_p(m)$ differs from o by either a precondition or a delete effect.

In either case, if $f_p(m)$ differ from o by a precondition, then the actions produced from $f_p(m)$ and o will be applicable in different set of states. On the other hand, if $f_p(m)$ differs from o by a delete effect, then the actions produced from $f_p(m)$ and o will not reach the same states. Therefore, $f_p(m)$ and o will have different reach sets under any bijective predicate mapping. This concludes the proof of this sub-formula. Similarly, we prove the correctness of the following formulas:

$$\forall f_{t-del} \in Sf_{t-del}, \forall f_{t-add} \in Sf_{t-add}\ (f_{t-del} \not\approx_{var} f_{t-add} \rightarrow \Gamma(f_p(m)) \neq \Gamma(o)).$$

A.3.3.4 Proving the Unification of the Preconditions, Delete effects and Add effects Mappings is a Mapping that Satisfy the Conditions of f_t

So far, we have proven that three atom-consistent bijective mappings exist from the preconditions, add effects and delete effects of the operator m to the preconditions, add effects and delete effects of the operator the o respectively. We also proved that these mappings satisfy the arity, predicate, and variable-order conditions. Now, we will show that the unification of these mappings produces an atom mapping f'_t that satisfies the conditions of f_t.

$$\text{Let } f'_t(t) = \begin{cases} f_{t-pre}(t) & \text{if } t \in \text{Pre}(m) \\ f_{t-del}(t) & \text{if } t \in \text{Del}(m) \setminus \text{Pre}(m) \\ f_{t-add}(t) & \text{if } t \in \text{Add}(m) \end{cases}$$

We know f_t is defined over Atoms(m) which is equal to $\text{Pre}(m) \cup (\text{Del}(m) \setminus \text{Pre}(m)) \cup \text{Add}(m)$. The range of f_t is Atoms(o) which is equal to $\text{Pre}(o) \cup \text{Del}(o) \cup \text{Add}(o)$. Therefore, f'_t and f_t have equal domains and ranges. Moreover, since f'_t consists of the bijective atom-consistent mappings f_{t-pre}, f_{t-add}, and f_{t-del} which satisfy the arity, predicate, and variable-order conditions, the relation f'_t is a bijective mapping and has the following properties.

1. Atoms in Pre(m), Del(m), and Add(m) are mapped to atoms in Pre(o), Del(o), and Add(o) respectively;
2. Atoms in m are mapped to atoms in o with equal arity;
3. Atoms in m with the same predicate p are mapped to atoms with some predicate p' in o; and
4. Atoms in m with a shared variable v are mapped to atoms in o with some shared variable v' such that the positions of v and v' in the mapped atoms are equal.

Thus f'_t satisfies the conditions of f_t as stated in this theorem. The existence of f'_t is a prove of the existence of f_t. Therefore the proof of the existence of f'_t concludes the proof of the forward implication.

A.3.4 The Proof of the Backward Implication Using a Constructive Approach

The backward implication states that the existence of a bijective mapping f_t from the atoms of m to the atoms o, which respects the conditions in this theorem, implies there exists a bijective mapping f_p between the predicates of m and those of o of equal arity, and that the reach set of m under f_p is equal to the reach set of o.

A.3.4.1 Proof Sketch

The backward implication is proven constructively starting from the existence of a bijective mapping f_t that satisfies the conditions stated in this theorem. From this antecedent, we will prove the existence of a bijective mapping f_p between the predicates of m and those of o of equal arity. Then, we will demonstrate that $\Gamma(f_p(m)) = \Gamma(o)$.

A.3.4.2 Proving the Existence of a Bijective Mapping f_p Between the Predicates of m and those of o of Equal Arity

The function f_t maps atoms with the same predicate p in m to atoms with some predicate p' in o. This implies that both operators m and o have an equal number of predicates. Furthermore, f_t maps atoms in m to atoms in o with equal arity. Therefore, any atom in m with a predicate p must be mapped by f_t to an atom in o with a predicate p' such that p and p' have equal arity. Thus, there excites a bijective mapping f_p between the predicates of m and those of o of equal arity.

A.3.4.3 Proving $\Gamma(f_p(m)) = \Gamma(o)$

We will prove that $f_p(m)$ and o are identical apart from the variable names. This means we will prove that one operator can be produced from the other by renaming its variables. This guarantees that $f_p(m)$ and o have the same reach set.

First, the existence of the mapping f_t between the atoms of m and o implies the number of atoms of each arity and the number of atoms of each predicate are equal in the preconditions of m and o. Furthermore, for every two atoms in o that have a shared variable v, there are two atoms in the preconditions of m that share some variable v' and the positions of v in the parameters of its atoms are equal to the positions of v' in the parameters of its atoms. This conclusion is supported by the properties of the mapping f_t.

Second, the operator $f_p(m)$ is produced by renaming the predicates of m using the predicate mapping f_p which does not change the number of predicates, atoms or the variables of m. So, $f_p(m)$ and m differ by only the name of the predicates. Therefore, the number of atoms of each arity is equal in the preconditions of $f_p(m)$ and m. Furthermore, the number of atoms that share some predicate is also equal in the preconditions of $f_p(m)$ and m. Moreover, $f_p(m)$ and m have the same variables.

From the previous two points, we can conclude that the number of atoms of each arity and the number of atoms of each predicate are equal in the preconditions of $f_p(m)$ and o. Furthermore, for every two

atoms in the preconditions of $f_p(m)$ that have a shared variable v, there are two atoms in the preconditions of o that share some variable v' and the positions of v in the parameters of its atoms are equal to the positions of v' in the parameters of its atoms. Therefore, $f_p(m)$ and o must have the same preconditions apart from the variable names.

With a similar discussion to the one used to prove both $f_p(m)$ and o have the same preconditions, we can prove the two operators $f_p(m)$ and o have the same delete and add effects apart from the variable names as well.

This concludes that both $f_p(m)$ and o are identical apart from the variable names. In other words, we can say that one operator is produced from the other by renaming its variables. As such, the two operators have equal reach sets. This proves the backward implication of this theorem. Hence, we conclude the proof of this theorem. ■

A.4 Proof of Theorem 4 (See page 78)

To prove this theorem, we first prove Lemma 4.2 in Appendix A.4.1 and then the main theorem is proven in Appendix A.4.2.

A.4.1 Proof of the Lemma of Theorem 4

Lemma 4.2 (Complex Domain Reachability Lemma)**.** *Consider a set of objects, Obj, two planning domain models D_1 and D_2, and a bijective function $F_p : P_1 \twoheadrightarrow P_2$. We have*

$$\forall o \in O_1, \exists o' \in O_2 (\Gamma(F_p(o), Obj) = \Gamma(o', Obj)) \implies \Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj) \tag{4.5}$$

Proof. Let $D_3 = (P_2, O_3)$ be a planning domain model with the same set of predicates as D_2, and the set of its operators O_3 satisfies the following conditions:

$$\begin{gathered} O_3 \subseteq O_2 \\ \wedge \\ \forall o \in O_1, \exists o' \in O_3 (\Gamma(F_p(o)) = \Gamma(o')) \end{gathered} \tag{A.54}$$

From the definition of O_3, we have $O_3 \subseteq O_2$; thus, any sequence of operators that can be made by the operators of O_3 can also be made by the operators of D_2. Since the reach set of a domain is equal to the union of the reach sets of its sequence of operators, we have $\Gamma(D_3) \subseteq \Gamma(D_2)$.

Furthermore, the definition of O_3 implies the image under F_p of the reach set of each operator in D_1 is equal to the reach set of an operator from D_3. This means the reach set of any sequence of operators made by the operators of D_1 is equal to the reach set of a sequence of operators which can be made from the operators of D_3. Thus we have $\Gamma(F_p(D_1)) \subseteq \Gamma(D_3)$. Therefore, $\Gamma(F_p(D_1)) \subseteq \Gamma(D_2)$. This argument concludes the proof of this lemma. ■

A.4.2 Proof of Theorem 4 using Lemma 4.2 (See page 78)

Theorem 4 (Complex Domains Reachability Theorem)**.** *Consider a set of objects Obj, two planning domain models, D_1 and D_2, a bijective function F_p from the predicates of D_1 to the predicates of D_2 with equal arity, and the relation R_{OM} that relates each operator o in O_1 to a bijective predicate mapping f_p that makes the reach set of $f_p(o)$ equals to the reach set of an operator from O_2. We have:*

$$\exists R'_{om} \subseteq R_{OM}(Domain(R'_{om}) = O_1 \wedge (F = \bigcup_{f_p \in Range(R'_{om})} f_p) \text{ is a bijective function}) \implies \Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj)$$

Let $\mathbb{F}$ be the set of all bijective mappings from predicates of equal arity from every operator in D_1 to every operator in D_2.

$$\mathbb{F} = \{f_p | f_p : Predicates(o) \twoheadrightarrow Predicates(o') \text{ where } o \in O_1, o' \in O_2 \text{ and if } f_p(p) = p' \text{ then } Arity(p) = Arity(p')\}$$

Let R_{OM} be a relation between operators from D_1 and predicate mappings from $\mathbb{F}$. An operator o from D_1 is related to a mapping f_p from $\mathbb{F}$ by R_{OM} means there exists an operator o' from D_2 such that the reach set of $f_p(o)$ is equal to the reach set of o'.

$$R_{OM} = \{(o, f_p) \in O_1 \times \mathbb{F} \mid \exists o' \in O_2,\ \Gamma(f_p(o), Obj) = \Gamma(o', Obj)\} \tag{A.55}$$

Proof. The antecedent $\exists R'_{om} \subseteq R_{OM}(Domain(R'_{om}) = O_1)$ implies

$$\forall o \in O_1, \exists f \in \mathbb{F}, (o, f) \in R'_{om} \tag{A.56}$$

From the definition of R_{OM} in Equation (A.55) and Equation (A.56), we have

$$\forall o \in O_1, \exists f \in Range(R'_{om}), \exists o' \in O_2 (\Gamma(f(o)) = \Gamma(o')) \tag{A.57}$$

The antecedent defines F as the union of all mappings in Range(R'_{om}) ($F = \bigcup_{f_p \in Range(R'_{om})} f_p$). Hence, we have

$$\forall f \in Range(R'_{om})(f \subseteq F) \tag{A.58}$$

Moreover, because F is a bijective function according to the antecedent of this backward implication, we can replace f with F in Equation (A.57). Then, this equation can be written as

$$\forall o \in O_1, \exists o' \in O_2 (\Gamma(F(o)) = \Gamma(o')) \tag{A.59}$$

The domain of every mapping in Range(R'_{om}) is the set of predicates of one operator from O_1.

$$\forall f \in Range(R'_{om})(Domain(f) = \{p | p \in Predicates(o) \text{ where } (o, f) \in R'_{om}\}) \tag{A.60}$$

Since F is the union of all mappings in Range(R'_{om}), the domain of F is equal to the union of the domains of all mappings in Range(R'_{om}).

$$Domain(F) = \bigcup_{f_p \in Range(R'_{om})} Domain(f_p) \tag{A.61}$$

From Equation (A.56), Equation (A.60), and Equation (A.61), we conclude

$$Domain(F) = Predicates(O_1) \tag{A.62}$$

However, Predicates(O_1) = P_1. Hence,

$$Domain(F) = P_1 \tag{A.63}$$

The range of every mapping in Range(R'_{om}) is the set of predicates of an operator o' from O_2 such that the reach set of an operator from O_1 under the given mapping is equal to the reach set of the operator o'. The following equation formally defines the range of a mapping from Range(R'_{om}).

$$\forall f \in Range(R'_{om})(Range(f) = \{p | p \in Predicates(o') \text{ where } o' \in O_2 \text{ if } \exists o \in O_1(\Gamma(f(o)) \subseteq \Gamma(o'))\}) \tag{A.64}$$

Since F is the union of all mappings in Range(R'_{om}) and F is a bijective function, the range of F is equal to the union of the ranges of all mappings in Range(R'_{om}).

$$Range(F) = \bigcup_{f_p \in Range(R'_{om})} Range(f_p) \tag{A.65}$$

From Equation (A.64) and Equation (A.65), we conclude

$$Range(F) \subseteq Predicates(O_2) \tag{A.66}$$

However, Predicates(O_2) = P_2. Hence,

$$Range(F) \subseteq P_2 \tag{A.67}$$

The function F is a bijective function because it is the union of a set of bijective functions. Hence, $|Domain(F)| = |Range(F)|$. From Equation (A.63), we have $|Domain(F)| = |P_1|$. So, $|Range(F)| = |P_1|$. We know $|P_1| = |P_2|$ from the assumption of this theorem, therefore $|Range(F)| = |P_2|$. Thus, according to Equation (A.67) we infer

$$Range(F) = P_2 \tag{A.68}$$

We have F is a bijective function from P_2 to P_1 as per Equation (A.63) and Equation (A.68). In addition, Equation (A.43) proves the image of the reach set of each operator in D_1 under F is equal to the reach set of a operator in D_2. Therefore, with the help of Lemma 4.2, we can deduce $\Gamma(F_p(D_1)) \subseteq \Gamma(D_2)$. These arguments conclude the proof of this theorem. ■

A.5 Proof of Theorem 5 (See page 79)

Theorem 5 (Reach Sets Containment Theorem)**.** *For two planning domain models, D_1 and D_2, and two bijective functions $F_p : P_1 \twoheadrightarrow P_2$ and $G_p : P_2 \twoheadrightarrow P_1$, let $F_p(D_1)$ be the image of D_1 using F_p to substitute its predicates with those of D_2, and $G_p(D_2)$ be the image of D_2 under G_p. Then we have for a set of objects Obj:*

1. $\Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj)$ *and* $\Gamma(G_p(D_2), Obj) \subseteq \Gamma(D_1, Obj) \iff \Gamma(F_p(D_1), Obj) = \Gamma(D_2, Obj)$ *and* $F_p = G_p^{-1}$.
2. $\Gamma(F_p(D_1), Obj) \subseteq \Gamma(D_2, Obj)$ *and* $\Gamma(G_p(D_2), Obj) \subseteq \Gamma(D_1, Obj) \iff \Gamma(G_p(D_2), Obj) = \Gamma(D_1, Obj)$ *and* $G_p = F_p^{-1}$.

Proof. For brevity, the set *Obj* is dropped from the reach set symbols in the proofs as it is common to all reach sets. Items one and two of this theorem will be proved together as follows. From the antecedent of the forward implication, we have

$$\Gamma(F_p(D_1)) \subseteq \Gamma(D_2) \tag{A.69}$$

$$\Gamma(G_p(D_2)) \subseteq \Gamma(D_1) \tag{A.70}$$

$F_p(D_1)$ and (D_2) have the same predicates and the reach set of $F_p(D_1)$ is a subset of the reach set of D_2. So, if we rename the predicates of $F_p(D_1)$ and D_2 using F_p^{-1}, then both $F_p^{-1}(F_p(D_1))$ and $F_p^{-1}(D_2)$ will have the same predicates and the reach set of $F_p^{-1}(F_p(D_1))$ will be a subset of the reach set of $F_p^{-1}(D_2)$. Let us also rename the predicates of $G_p(D_2)$ and D_1 using G_p^{-1}. Then we have

$$\Gamma(D_1) \subseteq \Gamma(F_P^{-1}(D_2)) \tag{A.71}$$

$$\Gamma(D_2) \subseteq \Gamma(G_P^{-1}(D_1)) \tag{A.72}$$

Since $\Gamma(G_p(D_2)) \subseteq \Gamma(D_1)$ and from (A.71) we have

$$|\Gamma(D_1)| \geq |\Gamma(G_p(D_2))| \tag{A.73}$$

$$|\Gamma(D_1)| \leq |\Gamma(F_p^{-1}(D_2))| \tag{A.74}$$

Since f and g are bijective functions, they do not change the size of the reach sets on which they are applied. Thus we have

$$|\Gamma(D_2)| = |\Gamma(G_p(D_2))| \tag{A.75}$$

$$|\Gamma(D_2)| = |\Gamma(F_p^{-1}(D_2))| \tag{A.76}$$

Therefore,

$$|\Gamma(F_p(D_2))| = |\Gamma(g^{-1}(D_2))| \tag{A.77}$$

From (A.73), (A.74) and (A.77) we have $|\Gamma(D_1)| = \Gamma(G_p(D_2))| = |\Gamma(F_p^{-1}(D_2))|$.
$\Gamma(G_p(D_2))$ is a subset of $\Gamma(D_1)$ as per (A.70) and both have equal cardinality, hence:

$$\Gamma(G_p(D_2)) = \Gamma(D_1) \tag{A.78}$$

$\Gamma(D_1)$ is a subset of $\Gamma(F_p^{-1}(D_2))$ as per (A.71) and both have equal cardinality, hence:

$$\Gamma(D_1) = \Gamma(F_p^{-1}(D_2)) \tag{A.79}$$

Rename the predicates of D_1 and $F_p^{-1}(D_2)$ using F_p, then we have

$$\Gamma(F_p(D_1)) = \Gamma(D_2) \tag{A.80}$$

From (A.78) and (A.79), we infer that $\Gamma(G_p(D_2)) = \Gamma(F_p^{-1}(D_2)) =$. Hence, $G_p = F_p^{-1}$. Furthermore, $G_p^{-1} = F_p^{-1^{-1}}$. Thus, $F_p = G_p^{-1}$ These arguments proves the forward implication. The proof of the backward implication is trivial because if two sets are equal then the subset relation follows directly. ■

A.6 Proof of Corollary 4.1 (See page 79)

Corollary 4.1 (Domain Reach Sets Equality corollary)**.** *Consider a set of objects, Obj, two planning domain models, D_1 and D_2, a bijective function F_p from the predicates of D_1 to the predicates of D_2 with equal arity. We have:*

$$\forall o \in O_1, \exists o' \in O_2(\Gamma(F_p(o)) = \Gamma(o')) \wedge |O_1| = |O_2| \implies \Gamma(F_p(D_1) = \Gamma(D_2)$$

Proof. From the antecedent of the forward implication and according to Lemma 4.2, we have

$$\Gamma(F_p(D_1)) \subseteq \Gamma(D_2) \tag{A.81}$$

As we assume the two domains D_1 and D_2 do not have duplicated operators and according to Theorem 3, we deduce

$$\forall o_1, o_2 \in O_1\ (\Gamma(o_1) \neq \Gamma(o_2)) \tag{A.82}$$

From Equation (A.81) and Equation (A.82), and because F_p is a bijective function, we infer

$$\forall o' \in O_2, \exists o \in O_1(\Gamma(F_p^{-1}(o')) = \Gamma(o)) \implies \Gamma(F_p^{-1}(D_2)) \subseteq \Gamma(D_1) \tag{A.83}$$

From Equation (A.81) and Equation (A.83), and according to Theorem 5, we conclude

$$\Gamma(F_p(D_1) = \Gamma(D_2) \tag{A.84}$$

Hence, we complete the prove of the this corollary. ■

APPENDIX B

DESCRIPTION OF FUNCTIONAL EQUIVALENCE VALIDATION TASKS

B.1 Description of the Modifications Applied to the Planning Domain Models in the Experiments

Domain version	Expected impact	Modification description
Gripper with crafted valid macro	Yes	The valid macro operator "pick-move-drop" is handcrafted and added to this modified version.
Gripper with random valid macro	Yes	The valid macro operator "move-pick-drop" is randomly created from the operators of the original domain and added to this modified version.
Gripper with random invalid macro	No	The invalid macro operator "move-pick-drop" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (at-robby ?x2) with the precondition (at ?x3 ?x2) in the original valid macro.
Gripper with swapped atoms	No	The atom (at-robby ?to) in the add effects of the operator "move" in the original domain is changed to a precondition in this modified version, and the atom (at-robby ?from) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Gripper with deleted operator	No	The operator "drop" is removed from this modified domain, this operator exists in the original domain.

Table B.1: The description of the modifications applied to the Gripper domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Gripper domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Blocksworld with crafted valid macro	Yes	The valid macro operator "pick-up-stack" is handcrafted and added to this modified version.
Blocksworld with random valid macro	Yes	The valid macro operator "pick-up-put-down-stack" is randomly created from the operators of the original domain and added to this modified version.
Blocksworld with random invalid macro	No	The invalid macro operator "pick-up-put-down-stack" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (clear ?x2) with the precondition (ontable-M ?x1) in the original valid macro.
Blocksworld with swapped variables	No	The variable ?x in the parameters of the add effect (on ?y ?x) in the operator "stack" in the original domain is swapped with the variable ?y from the parameters of the same add effect of the same operator in this modified version.
Blocksworld with swapped atoms	No	The atom (clear ?y) in the add effects of the operator "unstack" in the original domain is changed to a precondition in this modified version, and the atom (clear ?x) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Blocksworld with deleted operator	No	The operator "unstack" is removed from this modified domain, this operator exists in the original domain.

Table B.2: The description of the modifications applied to the Blocksworld domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Blocksworld domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Parking with crafted valid macro	Yes	The valid macro operator "move-car-to-curb-move-car-to-car" is handcrafted and added to this modified version.
Parking with random valid macro	Yes	The valid macro operator "move-curb-to-curb-move-curb-to-car-move-car-to-curb" is randomly created from the operators of the original domain and added to this modified version.
Parking with random invalid macro	No	The invalid macro operator "move-curb-to-curb-move-curb-to-car-move-car-to-curb" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (at-curb ?x1) with the precondition (at-curb-num ?x1 ?x4) in the original valid macro.
Parking with swapped variables	No	The variable ?car in the parameters of the add effect (behind-car ?cardest ?car) in the operator "move-curb-to-car" in the original domain is swapped with the variable ?cardest from the parameters of the same add effect of the same operator in this modified version.
Parking with swapped atoms	No	The atom (at-curb ?car) in the add effects of the operator "move-car-to-curb" in the original domain is changed to a precondition in this modified version, and the atom (behind-car ?car ?carsrc) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Parking with deleted operator	No	The operator "move-curb-to-car" is removed from this modified domain, this operator exists in the original domain.

Table B.3: The description of the modifications applied to the Parking domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Parking domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Hiking with crafted valid macro	Yes	The valid macro operator "drive-tent-put-up" is handcrafted and added to this modified version.
Hiking with random valid macro	Yes	The valid macro operator "put-down-put-up-drive-passenger" is randomly created from the operators of the original domain and added to this modified version.
Hiking with random invalid macro	No	The invalid macro operator "put-down-put-up-drive-passenger" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (at-person ?x1 ?x4) with the precondition (at-tent ?x5 ?x3) in the original valid macro.
Hiking with swapped variables	No	The variable ?x3 in the parameters of the precondition (partners-M ?x6 ?x3 ?x5) in the operator "walk-together" in the original domain is swapped with the variable ?x5 from the parameters of the same precondition of the same operator in this modified version.
Hiking with swapped atoms	No	The atom (up ?x3) in the add effects of the operator "put-up" in the original domain is changed to a precondition in this modified version, and the atom (at-tent ?x3 ?x2) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Hiking with deleted operator	No	The operator "drive-tent-passenger" is removed from this modified domain, this operator exists in the original domain.

Table B.4: The description of the modifications applied to the Hiking domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Hiking domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Floor-tile with crafted valid macro	Yes	The valid macro operator “up-right” is handcrafted and added to this modified version.
Floor-tile with random valid macro	Yes	The valid macro operator “change-color-paint-up-paint-down” is randomly created from the operators of the original domain and added to this modified version.
Floor-tile with random invalid macro	No	The invalid macro operator “change-color-paint-up-paint-down” is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (robot-has ?x1 ?x3) with the precondition (available-color ?x3) in the original valid macro.
Floor-tile with swapped variables	No	The variable ?x in the parameters of the precondition (right-M ?y ?x) in the operator “right” in the original domain is swapped with the variable ?y from the parameters of the same precondition of the same operator in this modified version.
Floor-tile with swapped atoms	No	The atom (clear ?x) in the add effects of the operator “down” in the original domain is changed to a precondition in this modified version, and the atom (down ?y ?x) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Floor-tile with deleted operator	No	The operator “right” is removed from this modified domain, this operator exists in the original domain.

Table B.5: The description of the modifications applied to the Floor-tile domain to produce its modified versions. Expected impact: “yes” means the introduced modification to the original Floor-tile domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	**Expected impact**	**Modification description**
Child-snack with crafted valid macro	Yes	The valid macro operator "make-sandwich-put-on-tray" is handcrafted and added to this modified version.
Child-snack with random valid macro	Yes	The valid macro operator "make-sandwich-no-gluten-put-on-tray-serve-sandwich-no-gluten" is randomly created from the operators of the original domain and added to this modified version.
Child-snack with random invalid macro	No	The invalid macro operator "make-sandwich-no-gluten-put-on-tray-serve-sandwich-no-gluten" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (no-gluten-sandwich ?x3) with the precondition (allergic-gluten-M ?x7) in the original valid macro.
Child-snack with swapped atoms	No	The atom (at ?t ?p2) in the add effects of the operator "move-tray" in the original domain is changed to a precondition in this modified version, and the atom (at ?t ?p1) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Child-snack with deleted operator	No	The operator "put-on-tray" is removed from this modified domain, this operator exists in the original domain.

Table B.6: The description of the modifications applied to the Child-snack domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Child-snack domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	**Expected impact**	**Modification description**
Logistics with crafted valid macro	Yes	The valid macro operator "load-fly-unload-airplane" is handcrafted and added to this modified version.
Logistics with random valid macro	Yes	The valid macro operator "load-truck-location-load-truck-airport-load-airplane" is randomly created from the operators of the original domain and added to this modified version.
Logistics with random invalid macro	No	The invalid macro operator "load-truck-location-load-truck-airport-load-airplane" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (in-truck ?x1 ?x4) with the precondition (at-package-location ?x1 ?x2) in the original valid macro.
Logistics with swapped atoms	No	The atom (at-truck-location ?truck ?loc-to) in the add effects of the operator "drive-truck-airport-location" in the original domain is changed to a precondition in this modified version, and the atom (airport-in-city ?loc-from ?city) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Logistics with deleted operator	No	The operator "fly-airplane" is removed from this modified domain, this operator exists in the original domain.

Table B.7: The description of the modifications applied to the Logistics domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Logistics domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Cave-diving with crafted valid macro	Yes	The valid macro operator "swim-photograph" is handcrafted and added to this modified version.
Cave-diving with random valid macro	Yes	The valid macro operator "prepare-tank-enter-water-pickup-tank" is randomly created from the operators of the original domain and added to this modified version.
Cave-diving with random invalid macro	No	The invalid macro operator "prepare-tank-enter-water-pickup-tank" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (holding ?x1 ?x2) with the precondition (capacity ?x1 ?x5) in the original valid macro.
Cave-diving with swapped variables	No	The variable ?q1 in the parameters of the precondition (next-quantity-M ?q1 ?q2) in the operator "drop-tank" in the original domain is swapped with the variable ?q2 from the parameters of the same precondition of the same operator in this modified version.
Cave-diving with swapped atoms	No	The atom (at-diver ?d ?l) in the add effects of the operator "enter-water" in the original domain is changed to a precondition in this modified version, and the atom (cave-entrance ?l) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Cave-diving with deleted operator	No	The operator "prepare-tank" is removed from this modified domain, this operator exists in the original domain.

Table B.8: The description of the modifications applied to the Cave-diving domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Cave-diving domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Rover with crafted valid macro	Yes	The valid macro operator "calibrate-take-image" is handcrafted and added to this modified version.
Rover with random valid macro	Yes	The valid macro operator "navigate-sample-soil-drop" is randomly created from the operators of the original domain and added to this modified version.
Rover with random invalid macro	No	The invalid macro operator "navigate-sample-soil-drop" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (have-soil-analysis ?x1 ?x3) with the precondition (available ?x1) in the original valid macro.
Rover with swapped variables	No	The variable ?y in the parameters of the precondition (can-traverse ?x ?y ?z) in the operator "navigate" in the original domain is swapped with the variable ?z from the parameters of the same precondition of the same operator in this modified version.
Rover with swapped atoms	No	The atom (calibrated ?i ?r) in the add effects of the operator "calibrate" in the original domain is changed to a precondition in this modified version, and the atom (at ?r ?w) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Rover with deleted operator	No	The operator "communicate-rock-data" is removed from this modified domain, this operator exists in the original domain.

Table B.9: The description of the modifications applied to the Rover domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Rover domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Pipesworld with crafted valid macro	Yes	The valid macro operator "pop-start" is handcrafted and added to this modified version.
Pipesworld with random valid macro	Yes	The valid macro operator "push-start-push-end-pop-start" is randomly created from the operators of the original domain and added to this modified version.
Pipesworld with random invalid macro	No	The invalid macro operator "push-start-push-end-pop-start" is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (pop-updating ?x1) with the precondition (last ?x2 ?x1) in the original valid macro.
Pipesworld with swapped variables	No	The variable ?last-batch-atom in the parameters of the delete effect (follow-M ?next-last-batch-atom ?last-batch-atom) in the operator "push-end" in the original domain is swapped with the variable ?next-last-batch-atom from the parameters of the same delete effect of the same operator in this modified version.
Pipesworld with swapped atoms	No	The atom (last ?batch-atom-in ?pipe) in the add effects of the operator "pop-unitarypipe" in the original domain is changed to a precondition in this modified version, and the atom (connect ?from-area ?to-area ?pipe) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Pipesworld with deleted operator	No	The operator "push-unitarypipe" is removed from this modified domain, this operator exists in the original domain.

Table B.10: The description of the modifications applied to the Pipesworld domain to produce its modified versions. Expected impact: "yes" means the introduced modification to the original Pipesworld domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Scanalyzer with crafted valid macro	Yes	The valid macro operator “analyze-2-rotate-2” is handcrafted and added to this modified version.
Scanalyzer with random valid macro	Yes	The valid macro operator “analyze-2-analyze-4-rotate-2” is randomly created from the operators of the original domain and added to this modified version.
Scanalyzer with random invalid macro	No	The invalid macro operator “analyze-2-analyze-4-rotate-2” is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (analyzed ?x5) with the precondition (on ?x6 ?x2) in the original valid macro.
Scanalyzer with swapped variables	No	The variable ?s1 in the parameters of the precondition (cycle-2 ?s1 ?s2) in the operator “rotate-2” in the original domain is swapped with the variable ?s2 from the parameters of the same precondition of the same operator in this modified version.
Scanalyzer with swapped atoms	No	The atom (on ?c4 ?s3) in the add effects of the operator “analyze-4” in the original domain is changed to a precondition in this modified version, and the atom (on ?c2 ?s2) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Scanalyzer with deleted operator	No	The operator “analyze-2” is removed from this modified domain, this operator exists in the original domain.

Table B.11: The description of the modifications applied to the Scanalyzer domain to produce its modified versions. Expected impact: “yes” means the introduced modification to the original Scanalyzer domain is expected to produce a version that is functionally equivalent to the original domain.

Domain version	Expected impact	Modification description
Freecell with crafted valid macro	Yes	The valid macro operator “move-sendtohome” is handcrafted and added to this modified version.
Freecell with random valid macro	Yes	The valid macro operator “move-move-b-sendtofree” is randomly created from the operators of the original domain and added to this modified version.
Freecell with random invalid macro	No	The invalid macro operator “move-move-b-sendtofree” is added to this modified version. This invalid macro is produced from the valid macro that is explained in the previous entry of this table by swapping the add effect (cellspace ?x5) with the precondition (canstack ?x3 ?x2) in the original valid macro.
Freecell with swapped variables	No	The variable ?cols in the parameters of the precondition (successor ?cols ?ncols) in the operator “sendtohome-b” in the original domain is swapped with the variable ?ncols from the parameters of the same precondition of the same operator in this modified version.
Freecell with swapped atoms	No	The atom (cellspace ?ncells) in the add effects of the operator “colfromfreecell” in the original domain is changed to a precondition in this modified version, and the atom (successor ?ncells ?cells) from the preconditions of the same operator in the original domain is changed to an add effect in this modified version.
Freecell with deleted operator	No	The operator “sendtofree” is removed from this modified domain, this operator exists in the original domain.

Table B.12: The description of the modifications applied to the Freecell domain to produce its modified versions. Expected impact: “yes” means the introduced modification to the original Freecell domain is expected to produce a version that is functionally equivalent to the original domain.

BIBLIOGRAPHY

[1] T. McCluskey, R. Aler, P. Borrajo, D.and Haslum, P. Jarvis, I. Refanidis, and U Scholz. Knowledge engineering for planning roadmap. `http://planet.hud.ac.uk/road1.pdf`, 2003. [Online; accessed 16-June-2023].

[2] Lukáš Chrpa and Mauro Vallati. Knowledge engineering in planning - representation matters. ICAPS2017, 2017. URL `https://icaps17.icaps-conference.org/tutorials/T4-Knowledge-Engineering-in-Planning.pdf`.

[3] Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1):5 – 47, 1998. ISSN 0004-3702. doi: 10.1016/S0004-3702(98/00068-X. Artificial Intelligence 40 years later.

[4] S. Chien, R. Sherwood, D. Tran, B. Cichy, G. Rabideau, R. Castano, A. Davies, R. Lee, D. Mandl, S. Frye, B. Trout, J. Hengemihle, J. D'Agostino, S. Shulman, S. Ungar, T. Brakke, D. Boyer, J. Van Gaasbeck, R. Greeley, T. Doggett, V. Baker, J. Dohm, and F. Ip. The EO-1 autonomous science agent. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004.*, pages 420–427, July 2004.

[5] M. Ai-Chang, J. Bresina, L. Charest, A. Chase, J. C. . Hsu, A. Jonsson, B. Kanefsky, P. Morris, Kanna Rajan, J. Yglesias, B. G. Chafin, W. C. Dias, and P. F. Maldague. Mapgen: mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1): 8–12, Jan 2004. ISSN 1541-1672. doi: 10.1109/MIS.2004.1265878.

[6] Version Hugh, Hugh Cottam, Nigel Shadbolt, and John Kingston. Knowledge level planning in the search and rescue domain. In *In Research and Development in Expert Systems XII, proceedings of BCS Expert Systems'95*, pages 309–326. SGES Publications, 1995.

[7] Austin Tate, Brian Drabble, and Jeff Dalton. *O-Plan: a knowledge-based planner and its application to logistics*. University of Edinburgh, Artificial Intelligence Applications Institute, 1996.

[8] Robert G. Sargent and Osman Balci. History of verification and validation of simulation models. In *2017 Winter Simulation Conference (WSC)*, pages 292–307, 2017. doi: 10.1109/WSC.2017.8247794.

[9] Saddek Bensalem, Klaus Havelund, and Andrea Orlandini. Verification and validation meet planning and scheduling. *International Journal on Software Tools for Technology Transfer*, 16(1): 1–12, Feb 2014. ISSN 1433-2787. doi: 10.1007/s10009-013-0294-x.

[10] Franco Raimondi, Charles Pecheur, and Guillaume Brat. PDVer, a tool to verify PDDL planning domains. 2009.

[11] Allen Goldberg, Klaus Havelund, and Conor McGann. Runtime verification for autonomous spacecraft software. In *Aerospace Conference, 2005 IEEE*, pages 507–516. IEEE, 2005.

[12] Margaret H Smith, Gerard J Holzmann, Gordon C Cucullu, and BD Smith. Model checking autonomous planners: Even the best laid plans must be verified. In *Aerospace Conference, 2005 IEEE*, pages 1–11. IEEE, 2005.

[13] Amedeo Cesta, Alberto Finzi, Simone Fratini, Andrea Orlandini, and Enrico Tronci. Validation and verification issues in a timeline-based planning system. *The Knowledge Engineering Review*, 25(3):299–318, 2010.

[14] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[15] EE Times. Formal verification with constraints — it doesn't have to be like tightrope walking. `https://www.eetimes.com/document.asp?doc_id=1277001`, 2010. [Online; accessed 16-June-2023].

[16] Rolf Drechsler, editor. *Advanced formal verification*, volume 122. Springer, 2004.

[17] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23 (5):279–295, 1997.

[18] Stefan Edelkamp, Shahid Jabbar, and Mohammed Nazih. Costoptimal planning with constraints and preferences in large state spaces. In *International Conference on Automated Planning and Scheduling (ICAPS) Workshop on Preferences and Soft Constraints in Planning*, pages 38–45, 2006.

[19] Anas Shrinah and Kerstin Eder. Goal-constrained planning domain model verification of safety properties. In *Proceedings of the ICAPS Workshop on Knowledge Engineering for Planning and Scheduling KEPS2019*, 2019.

[20] Anas Shrinah and Kerstin I Eder. Goal-constrained planning domain model verification of safety properties. In Sebastian Rudolph and Goreti Marreiros, editors, *Goal-constrained planning domain model verification of safety properties*, volume 2655 of *European Starting AI Researchers' Symposium 2020*. CEUR Workshop Proceedings, August 2020. European Starting AI Researchers' Symposium 2020, STAIRS 2020 ; Conference date: 29-08-2020 Through 30-08-2020.

[21] Thomas L McCluskey, Tiago S Vaquero, and Mauro Vallati. Engineering knowledge for automated planning: Towards a notion of quality. In *Proceedings of the Knowledge Capture Conference*, pages 1–8, 2017.

[22] Diego Aineto, Sergio Jiménez, and Eva Onaindia. Learning strips action models with classical planning. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*, 2018.

[23] Hankz Hankui Zhuo and Subbarao Kambhampati. Action-model acquisition from noisy plan traces. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[24] Anas Shrinah, Derek Long, and Kerstin Eder. D-VAL: An automatic functional equivalence validation tool for planning domain models, 2023.

[25] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[26] Mauro Vallati and Lukáš Chrpa. The international competition on knowledge engineering for planning and scheduling: Food for thoughts (and call to action). In *Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, United States, May 2020. OpenReview. URL `https://icaps19.icaps-conference.org/`. 29th International Conference on Automated Planning and Scheduling, ICAPS 2019 ; Conference date: 11-07-2019 Through 15-07-2019.

[27] Leah A. Chrestien and Lukáš Chrpa. Appropriate expressiveness of planning domain models: An urban traffic control case study. In *Proceedings of the 10th International Conference on Knowledge Capture*, K-CAP '19, page 247–250, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370080. doi: 10.1145/3360901.3364437.

[28] T.L. McCluskey, Tiago Vaquero, and Mauro Vallati. Issues in planning domain model engineering. In *PlanSIG 2015/16*, February 2016. URL `http://eprints.hud.ac.uk/id/eprint/27290/`.

[29] Dan Bryce, J Benton, and Michael W Boldt. Maintaining evolving domain models. In *Proceedings of the twenty-fifth international joint conference on artificial intelligence*, pages 3053–3059, 2016.

[30] Donald J Reifer. *Software Maintenance Success Recipes*. CRC Press, 2016.

[31] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971. ISSN 0004-3702. doi: 10.1016/0004-3702(71/90010-5.

[32] Edwin P. D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings*

of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89). Toronto, Canada, May 15-18 1989, pages 324–332. Morgan Kaufmann, 1989.

[33] Jörg Hoffmann, Stefan Edelkamp, Sylvie Thiébaux, Roman Englert, Frederico Liporace, and Sebastian Trüg. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research*, 26:453–541, 2006.

[34] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[35] B. Cenk Gazen and Craig A. Knoblock. Combining the expressivity of UCPOP with the efficiency of Graphplan. In Sam Steel and Rachid Alami, editors, *Recent Advances in AI Planning*, pages 221–233, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69665-0.

[36] Bernhard Nebel. On the compilability and expressive power of propositional planning formalisms. *J. Artif. Int. Res.*, 12(1):271–315, may 2000. ISSN 1076-9757.

[37] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL-the planning domain definition language. Technical report, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, CT, 1998.

[38] Maria Fox and Derek Long. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[39] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical report, Technical Report 195, University of Freiburg, 2004.

[40] Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL 3 the language of the fifth international planning competition. Technical report, Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy, 2005.

[41] Malte Helmert. Changes in PDDL3.1. `https://ipc08.icaps-conference.org/deterministic/PddlExtension.html`, 2008. [Online; accessed 16-June-2023].

[42] Alfonso Gerevini and Derek Long. Bnf description of PDDL3.0. *Unpublished manuscript from the IPC-5 website*, 2005.

[43] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[44] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation*

and Reasoning, KR'92, page 103–114, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1558602623.

[45] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):281–300, 1997. ISSN 0004-3702. doi: 10.1016/S0004-3702(96/00047-1.

[46] Kutluhan Erol, James Hendler, and Dana S Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996.

[47] Maria Fox and Derek Long. Hybrid STAN: Identifying and managing combinatorial optimisation sub-problems in planning. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 445–452, 2001.

[48] Ilche Georgievski and Marco Aiello. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156, 2015. ISSN 0004-3702. doi: 10.1016/j.artint.2015.02.002.

[49] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[50] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on UPPAAL. *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*, pages 200–236, 2004.

[51] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation: Modelling Techniques and Tools: 12th International Conference, TOOLS 2002 London, UK, April 14–17, 2002 Proceedings 12*, pages 200–204. Springer, 2002.

[52] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. ieee, 1977.

[53] Andrea Lodi, Silvano Martello, and Daniele Vigo. Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal on Computing*, 11(4): 345–357, 1999.

[54] Petar Borisov Stoykov. Rectangle packing in practice. Master's thesis, TU Eindhoven, 2017.

[55] Clark Barrett, A. Stump, and Cesare Tinelli. The SMT-LIB standard - version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories, Edinburgh, Scotland,(SMT '10)*, 2010.

[56] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 387–398, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2491978.

[57] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2011.

[58] Dirk Beyer, Adam J Chlipala, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings. 26th International Conference on Software Engineering*, pages 326–335. IEEE, 2004.

[59] Michael Cashmore, Maria Fox, Derek Long, and Daniele Magazzeni. A compilation of the full PDDL+ language into SMT. In *Proceedings of the international conference on automated planning and scheduling*, volume 26, pages 79–87, 2016.

[60] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[61] Bruno Dutertre and Leonardo De Moura. The yices SMT solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[62] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99524-9.

[63] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*, pages 93–107. Springer, 2013.

[64] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*, pages 174–177. Springer, 2009.

[65] R. Howey, D. Long, and M. Fox. 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, Nov 2004. doi: 10.1109/ICTAI.2004.120.

[66] Gregor Behnke, Daniel Höller, and Susanne Biundo. This is a solution!(... but is it though?) verifying solutions of hierarchical planning problems. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017*, 2017.

[67] Andreas Herzig, Maria Viviane de Menezes, Leliane Nunes De Barros, and Renata Wassermann. On the revision of planning tasks. In *ECAI*, pages 435–440, 2014.

[68] Moritz Göbelbecker, Thomas Keller, Patrick Eyerich, Michael Brenner, and Bernhard Nebel. Coming up with good excuses: What to do when no plan can be found. In *Proceedings of the international conference on automated planning and scheduling*, volume 20, pages 81–88, 2010.

[69] Alba Gragera, Raquel Fuentetaja, Ángel García-Olaya, and Fernando Fernández. Repair suggestions for planning domains with missing actions effects. In *ICAPS 2022 Workshop on Explainable AI Planning*, 2022. URL `https://openreview.net/forum?id=KrHA1GyDF4n`.

[70] Derek Long, Maria Fox, and Richard Howey. Planning domains and plans: validation, verification and analysis. In *Proc. Workshop on V&V of Planning and Scheduling Systems*, 2009.

[71] Songtuan Lin, Alban Grastien, and Pascal Bercher. Towards automated modeling assistance: An efficient approach for repairing flawed planning domains. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI*, 2023.

[72] Alan Lindsay, Jonathon Read, João Ferreira, Thomas Hayton, Julie Porteous, and Peter Gregory. Framer: Planning models from natural language action descriptions. *Proceedings of the International Conference on Automated Planning and Scheduling*, 27:434–442, 2017.

[73] Maurício Steinert and Felipe Rech Meneguzzi. Planning domain generation from natural language step-by-step instructions. In *2020 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS@ ICAPS), 2020, França.*, 2020.

[74] SRK Branavan, Nate Kushman, Tao Lei, and Regina Barzilay. Learning high-level planning from text. The Association for Computational Linguistics, 2012.

[75] Avirup Sil and Alexander Yates. Extracting strips representations of actions and events. In *RANLP*, pages 1–8. Citeseer, 2011.

[76] Stephen N Cresswell, Thomas Leo McCluskey, and Margaret M West. Acquiring planning domain models using LOCM. *Knowledge Engineering Review*, 28(2):195–213, 2013.

[77] Stephen Cresswell, Thomas McCluskey, and Margaret West. Acquisition of object-centred domain models from planning examples. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 19, pages 338–341, 2009.

[78] Stephen Cresswell and Peter Gregory. Generalised domain model acquisition from action traces. In *Proceedings of the international conference on automated planning and scheduling*, volume 21, pages 42–49, 2011.

[79] T.L. McCluskey, S.N. Cresswell, Nona Elizabeth Richardson, and Margaret M. West. Opmaker2: efficient action schema acquisition. December 2007. URL `http://eprints.hud.ac.uk/id/eprint/3215/`.

[80] Thomas Leo McCluskey, N Elisabeth Richardson, and Ron M Simpson. An interactive method for inducing operator descriptions. In *AIPS*, pages 121–130, 2002.

[81] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty. A review of learning planning action models. *The Knowledge Engineering Review*, 33:e20, 2018.

[82] Rabia Jilani. *Automated Domain Model Learning Tools for Planning*, pages 21–46. Springer International Publishing, Cham, 2020. ISBN 978-3-030-38561-3. doi: 10.1007/978-3-030-38561-3_2.

[83] Ron M Simpson, Diane E Kitchin, and Thomas Leo McCluskey. Planning domain definition using GIPO. *The Knowledge Engineering Review*, 22(2):117–134, 2007.

[84] Tiago Vaquero, Rosimarci Tonaco, Gustavo Costa, Flavio Tonidandel, José Reinaldo Silva, and J Christopher Beck. itSIMPLE4.0: Enhancing the modeling experience of planning problems. In *System Demonstration–Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12)*, pages 11–14, 2012.

[85] OMG. Unified modeling language specification - version 2.0. `https://www.omg.org/spec/UML/2.0`, 2005. [Online; accessed 16-June-2023].

[86] *Object Constraint Language*. Object Management Group (OMG), 2003. Rev. 2.0.

[87] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4): 541–580, 1989.

[88] Tiago Stegun Vaquero, José Reinaldo Silva, and J Christopher Beck. A brief review of tools and methods for knowledge engineering for planning & scheduling. *KEPS 2011*, 7, 2011.

[89] M Shah, Lukáš Chrpa, Falilat Jimoh, D Kitchin, T McCluskey, Simon Parkinson, and Mauro Vallati. Knowledge engineering tools in planning: State-of-the-art and future challenges. *Knowledge engineering for planning and scheduling*, 53:53, 2013.

[90] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program monitoring with LTL in EAGLE. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 264. IEEE, 2004.

[91] John Penix, Charles Pecheur, and Klaus Havelund. Using model checking to validate AI planner domain models. In *Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard*, 1998.

[92] Nicola Muscettola. HSTS: Integrating planning and scheduling. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST, 1993.

[93] Lina Khatib, Nicola Muscettola, and Klaus Havelund. Verification of plan models using UPPAAL. In *International Workshop on Formal Approaches to Agent-Based Systems*, pages 114–122. Springer, 2000.

[94] Klaus Havelund, Alex Groce, Gerard Holzmann, Rajeev Joshi, and Margaret Smith. Automated testing of planning models. In *International Workshop on Model Checking and Artificial Intelligence*, pages 90–105. Springer, 2008.

[95] Alfonso E Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.

[96] Stefan Edelkamp, Shahid Jabbar, and Mohammed Nazih. Large-scale optimal PDDL3 planning with MIPS-XXL. *5th International Planning Competition Booklet (IPC-2006)*, pages 28–30, 2006.

[97] IPC. International planning competition 2014 web site. `https://helios.hud.ac.uk/scommv/IPC-14/domains.html`, 2014. [Online; accessed 16-June-2023].

[98] Stefan Edelkamp. Limits and possibilities of PDDL for model checking software. *Edelkamp, & Hoffmann (Edelkamp & Hoffmann, 2003)*, 2003.

[99] Aws Albarghouthi, Jorge A. Baier, and Sheila A. McIlraith. On the use of planning technology for verification. In *In VVPS'09. Proceedings of the ICAPS Workshop on Verification & Validation of Planning & Scheduling Systems*, 2009.

[100] Yi Li, Jing Sun, Jin Song Dong, Yang Liu, and Jun Sun. Planning as model checking tasks. In *2012 35th Annual IEEE Software Engineering Workshop*, pages 177–186. IEEE, 2012.

[101] Alfonso Gerevini and Derek Long. Preferences and soft constraints in PDDL3. In *ICAPS workshop on planning with preferences and soft constraints*, pages 46–53, 2006.

[102] Robert P Goldman, Ugur Kuter, and A Schneider. Using classical planners for plan verification and counterexample generation. In *Proceedings of AAAI Workshop on Problem Solving Using Classical Planning. To appear*, 2012.

[103] S Shoeeb and TL McCluskey. On comparing planning domain models. In *Proceedings of the Annual PLANSIG Conference*, pages 92–94. UK PLANNING AND SCHEDULING Special Interest Group, 2011.

[104] M. Vallati, L. Chrpa, and F. Cerutti. Towards automated planning domain models generation. In *The 5th Italian Workshop on Planning and Scheduling, 4th December 2013, Turin, Italy.*, 2013.

[105] Mauro Vallati, Frank Hutter, Lukáš Chrpa, and Thomas Leo McCluskey. On the effective configuration of planning domain models. In *International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI press, 2015.

[106] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[107] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[108] Muhammad Abdul Hakim Newton, John Levine, Maria Fox, and Derek Long. Learning macro-actions for arbitrary planners and domains. In *ICAPS*, volume 2007, pages 256–263, 2007.

[109] Lukas Chrpa and Mauro Vallati. Planning with critical section macros: Theory and practice. *Journal of Artificial Intelligence Research*, 74:691–732, 2022.

[110] Carlos Areces, Facundo Bustos, Martín Ariel Domínguez, and Jörg Hoffmann. Optimizing planning domains by automatic action schema splitting. In *ICAPS*. Citeseer, 2014.

[111] Lukáš Chrpa and Thomas Leo McCluskey. On exploiting structures of classical planning problems: Generalizing entanglements. In *ECAI*, pages 240–245, 2012.

[112] Lukáš Chrpa, Mauro Vallati, and Thomas Leo McCluskey. Outer entanglements: a general heuristic technique for improving the efficiency of planning algorithms. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(6):831–856, 2018.

[113] Lukáš Chrpa, Mauro Vallati, and Thomas Leo McCluskey. Inner entanglements: Narrowing the search in classical planning by problem reformulation. *Computational Intelligence*, 35(2): 395–429, 2019. doi: 10.1111/coin.12203.

[114] Benny Godlin and Ofer Strichman. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*, pages 466–471, 2009.

[115] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[116] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 349–360, 2014.

[117] CAJ Van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):814–819, 2000.

[118] Andreas Kuehlmann and Cornelis AJ van Eijk. Combinational and sequential equivalence checking. *Logic synthesis and Verification*, pages 343–372, 2002.

[119] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 836–843, 2006.

[120] Pat Riddle, Jordan Douglas, Mike Barley, and Santiago Franco. Improving performance by reformulating PDDL into a bagged representation. In *Proceedings of the 8th Workshop on Heuristic Search for Domain-independent Planning (HSDIP@ ICAPS)*, pages 28–36, 2016.

[121] Drew M. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35, Jun. 2000. doi: 10.1609/aimag.v21i2.1506.

[122] Tathagata Chakraborti, Sarath Sreedharan, Yu Zhang, and Subbarao Kambhampati. Plan explanations as model reconciliation: Moving beyond explanation as soliloquy. *arXiv preprint arXiv:1701.08317*, 2017.

[123] Sarath Sreedharan, Alberto Olmo Hernandez, Aditya Prasad Mishra, and Subbarao Kambhampati. Model-free model reconciliation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 587–594. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/83.

[124] Andrew I Coles and Amanda J Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, 2007.

[125] Lukáš Chrpa, Mauro Vallati, and Thomas Leo McCluskey. On the online generation of effective macro-operators. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[126] Diego Aineto, Sergio Jiménez, Eva Onaindia, and Miquel Ramírez. Model recognition as planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 13–21, 2019.

[127] Minh D Nguyen, Max Thalmaier, Markus Wedler, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27 (11):2068–2082, 2008.

[128] Gerd Behrmann, Kim G. Larsen, Henrik R. Andersen, Henrik Hulgaard, and Jørn Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 163–177, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-49059-3.

[129] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A review of generalized planning. *The Knowledge Engineering Review*, 34:e5, 2019.

[130] León Illanes and Sheila A McIlraith. Generalized planning via abstraction: arbitrary numbers of objects. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7610–7618, 2019.

[131] Jorge Torres and Jorge A Baier. Polynomial-time reformulations of LTL temporally extended goals into final-state goals. In *IJCAI*, pages 1696–1703, 2015.

[132] Ron M Simpson and Thomas Leo McCluskey. A tool supported structured method for planning domain acquisition. In *Foundations of Intelligent Systems: 13th International Symposium, ISMIS 2002 Lyon, France, June 27–29, 2002 Proceedings 13*, pages 544–552. Springer, 2002.