LEARNING STATIC KNOWLEDGE FOR AI PLANNING DOMAIN MODELS VIA PLAN TRACES

RABIA JILANI

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the degree of Doctor of Philosophy

The University of Huddersfield

August 2017

COPYRIGHT STATEMENT

ABSTRACT

Learning is fundamental to autonomous behaviour and from the point of view of Machine Learning, it is the ability of computers to learn without being programmed explicitly. Attaining such capability for learning domain models for Automated Planning (AP) engines is what triggered research into developing automated domain-learning systems. These systems can learn from training data. Until recent research it was believed that working in dynamically changing and unpredictable environments, it was not possible to construct action models a priori. After the research in the last decade, many systems have proved effective in engineering domain models by learning from plan traces. However, these systems require additional planner oriented information such as a partial domain model, initial, goal and/or intermediate states. Hence, a question arises - whether or not we can learn a dynamic domain model, which covers all domain behaviours from real-time action sequence traces only.

The research in this thesis extends an area of the most promising line of work that is connected to work presented in an REF Journal paper. This research aims to enhance the LOCM system and to extend the method of Learning Domain Models for AI Planning Engines via Plan Traces. This method was first published in ICAPS 2009 by Cresswell, McCluskey, and West (Cresswell, 2009). LOCM is unique in that it requires no prior knowledge of the target domain; however, it can produce a dynamic part of a domain model from training. Its main drawback is that it does not produce static knowledge of the domain, and its model lacks certain expressive features. A key aspect of research presented in this thesis is to enhance the technique with the capacity to generate static knowledge. A test and focus for this PhD is to make LOCM able to learn static relationships in a fully automatic way in addition to the dynamic relationships, which LOCM can already learn, using plan traces as input.

We present a novel system - **The ASCoL** (**A**utomatic **S**tatic **Co**nstraints **L**earner) which provides a graphical interface for visual representation and exploits directed graph discovery and analysis technique. It has been designed to discover domain-specific static relations/constraints automatically in order to enhance planning domain models. The ASCoL method has wider applications. Combined with LOCM, ASCoL can be a useful tool to produce benchmark domains for automated planning engines. It is also useful as a debugging tool for improving existing domain models. We have evaluated ASCoL on fifteen different IPC domains and on different types of goal-oriented and random-walk plans as input training data and it has been shown to be effective.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Dedicated to my Parents, Mr and Mrs Prof Ghulam Jilani,
My Sister, Aisha Jilani and my Brother, Shakeel Ahmed

The people who taught me to fly…

ACKNOWLEDGEMENTS

I would like to take the opportunity to convey my gratitude to those whom I owe great respect and appreciation. I see my PhD as a long journey of not only achieving a degree but grooming myself overall as a person. I have to acknowledge many people who have contributed to my success one way or the other; starting from my full PhD fee waiver grant up till the completion of my Thesis.

I applied for this research position as a first class fresh graduate of an IT degree with no previous job or research experience and with lots of unsaid fears about upcoming career and life in mind. I owe huge thanks to The University of Huddersfield who made my dream come true. They took away my financial worries by providing me with a 100% fee waiver opportunity. I owe special thanks to Prof Lee McCluskey for helping me explore and identify my abilities unknown to me previously. Not only did he trust my skills but also supported me throughout my research work. A very considerate person Dr Diane Kitchin supervised me. She always gave me right advice for solving all kind of issue either academic or non-academic. She made me realise that a PhD is not about earning a degree; it is about learning to remain determined and be productive despite how low-flight the journey is or how high your research target is. An intellectually generous personality and my second supervisor, Dr Andrew Crampton, gave me more support and hopes. Here, another sympathetic and brainy personality, Dr Mauro Vallati allowed me access to his expertise. He encouraged me to identify and fit in the missing pieces of the jigsaw. Together, my research team gave me the directions which led me to a point where I was free to think and pave my way to the end.

I would also like to thank my department Director Dr Julie Wilkinson for her moral and personal support in all general matters. I must also mention Ms Gwen Wood and Chris Sentence who have taken care of all admin issues that have arisen in this time. Thanks a lot, SCEN tech team for helping with technical issues I had every now and then. I would like to say special thanks to Dr Karpenko-Seccombe for improving my academic English writing skills through her workshops and one-to-one sessions.

Overall, my PhD journey was tough but all harsh situations turned into ease as my compassionate sister Aisha and brother Ahmed Bhai were always by my side from start to end with their on-going support that I never felt lost or away from home. Thanks a bundle T and Ahmed Bhai for giving me wings. I must have never achieved this without your support. I owe you much in my life ahead.

Now to mention contributions of my parents and family... Nothing in this world can be achieved unless you have your parents behind your back to trust on, to lean on and to reflect on. They did everything and anything to give me the best opportunity to gain this success. I would like to express my deep gratitude to my parents, my father Prof. Ghulam Jilani and mother Shahida Jilani. Ammi and Puppa I am ME because of YOU. The more I am learning, the more I understand your unbeatable value in my life. Finally yet importantly, thanks a lot to my dear nephew Shaheer Ahmed and niece Izzah for lending me cheers in my life in the UK with your presence around. This provided me with a safe zone and brought me out of stress at times.

Finally, I thank Allah SWT for enabling me through this research work. May He bless me more success. Ameen*!!!*

LIST OF ABBREVIATIONS

Abbreviation	Explanation
AI	Artificial Intelligence
ASCoL	Automatic Static Constraints Learner
LOCM	Learning Object-Centred Models
LOP	LOCM with Optimised Plans
AP	Automated Planning
KE	Knowledge Engineering
KEPS	Knowledge Engineering for Planning & Scheduling
ML	Machine Learning
IPC	International Planning Competitions
ICAPS	International Conference on Automated Planning and Scheduling
PDDL	Planning Domain Definition Language
KR	Knowledge Representation
GO	Goal-Oriented
RW	Random Walks
FSM	Finite State Machines
MSR	Main Static Relation
ESR	Extended Static Relation
DSR	Dual-typed Static Relation
O_{SM}	Static Modifier Operator
TPP	Travelling Purchase Problem

Chapter 1 - Introduction

This thesis work concerns the area of Learning Domain Models for Artificial Intelligence Planning Engines via Plan Traces. This chapter covers the overview, motivation, novel contribution of the thesis and a reader's guide to the structure of the thesis.

Overview

Artificial Intelligence can be classed as a dimension of Computer Science that is related to the automation of intelligent behaviour. Intelligent behaviour is when an agent can perceive its environment and can take actions which maximise its chances of success (Russell et al., 2003). The main aim of building intelligent systems is to aid humans in various activities without undermining their sustainability, accuracy, and certainty and to extend their capacities in various fields of life. Automation in intelligent systems is the implementation of a process to execute it according to pre-set instructions with little or no intervention by humans.

The automated processes only substitute usual manual processing of activity with software/hardware processes that follow a systematic order that may still require human involvement. By defining the specific rules for possible scenarios, automation can be pre-set (e.g. an automatic washing machine) or it can be variable (a computer operating system), where different scenarios are tackled by different rules. In both the cases, the rules are pre-defined and fixed before achieving a pre-set outcome under all expected inputs (Skibniewski and Golparvar-Fard, 2016). Intelligent agents can learn, plan, understand, recognise, sense and solve problems in different real life situations. The system can essentially be observed and tested against different types of parameters and input conditions to assure the anticipated output.

There is an advanced category of automation in intelligent systems called autonomous/autonomic systems. These are systems with more ambitious goals of working independently of human intervention and take decisions on their own to carry out the process for achieving given goals to increase productivity. The autonomic systems embody high-level autonomy property which is defined is the following section.

Difference between Autonomy and Automaticity

Before moving into details, it is important to give clear definitions of terms used and their context to aid understanding of the discussion ahead. The difference between the terms

Automatic and Autonomic is a grey area that is not readily conforming to a set of rules. We use the terms based on their most common definitions in the literature.

The ***Autonomy*** property of any computer based system infers 'self-governance' and 'self-direction' characteristics in the system to fulfil the goals it is assigned. The term Autonomic is derived from the noun autonomy, and an autonomic system refers to a system/process that acts/occurs involuntarily, without any conscious control i.e. self-management. The best example of an autonomic system is a human autonomic nervous system (ANS). ANS is a division of the peripheral nervous system that influences the function of internal organs.

In terms of computer-based system design paradigm, a good example of autonomy in autonomic systems quoted by (Truszkowski et al., 2009) is the operation of Flight Software Program (FWP) in order to detect faults in flights. FWP works all independently starting from monitoring of spacecraft health & safety (H&S) condition, it identifies when the spacecraft goes beyond safe H&S conditions and then takes healing steps in order to maintain the safe limit without any intervention from ground at any point of time. Briefly, it is the assignment of responsibility to the system to meet the defined goals of the system (comprising of some decision making for the success of tasks).

Automaticity is a system's self-management property (comprising of automation in decision making for the successful operation of the overall system) (Sterritt et al., 2005). Based on the definition of automaticity, it may be argued that in order to meet overall system goals, automaticity is required in addition to autonomy (self-governance) and the success of autonomy requires successful automaticity (Truszkowski et al., 2009). Eventually, to make sure that the tasks run successfully requires that the system be reliable (Sterritt and Bustard, 2003).

A good example to enhance the understanding of the role of autonomy and automaticity in the operation of an autonomic system is a system that has to find a particular phenomenon using its onboard science instrument. For example, in order to decide between certain parameters, it requires system autonomy. The goal of the system to be fault tolerant, for instance, does not come under this specific delegated task of the system (its autonomy). Eventually, the task can fail if the system cannot tolerate faults. From this viewpoint, the autonomic and self-management characteristics can be considered specialised forms of autonomy (self-governance, self-direction) where it is specifically managing the system (to heal, protect, configure, optimise, and so on).

Since the autonomic computing research became more common, the "self-x" list has expanded, giving rise to the general term *selfware* or *self-**. Autonomic computing includes following key properties that cover the general goal of self-management (self*) property:

- self-configuration
- self-optimization
- self-awareness
- self-healing
- self-protection
- self-adaptation

Examples include autonomous ground transport, autonomous flight system, disaster response and recovery, autonomous space exploration, agricultural applications, Law enforcement technology and In-home services.

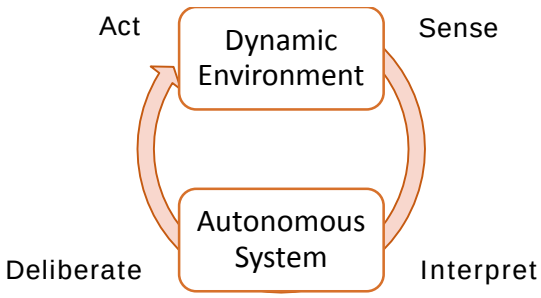


Figure 1.1: Autonomic Process

The autonomic decisions of the autonomic systems are based on different factors including the surrounding environments. Most of the time, humans provide goals to the autonomous systems. The process generally works through four steps to achieve the desired goals i.e. it senses from a dynamic environment using controllers; controllers interpret the data, take decisions and act in real time to change the environment to achieve some goals. Figure 1.1 illustrates the four steps and Figure 1.2 shows the real-time interaction of a high-level autonomous system with the dynamically changing environment. Autonomic systems require extensive domain model description that is constructed by using Knowledge Engineering (KE). These descriptions include the operator schema that can be executed in the environment, the states, constraints and rules of the objects in the environment and the goals to accomplish.

The background notions and theories for developing autonomic system include control theory, learning and adaptation algorithms, software agents, robots, fault tolerance computing, automated planning and problem-solving, machine learning, knowledge engineering, knowledge representation and much more. These skills have been applied in industries to substitute or support humans in activities that fall under the “4Ds”: **Dangerous, Difficult, Dirty, and Dull** (RailResearchAssosiation, 2012). Besides a number of reservations concerning legal and reliability issues, social factors, policy, ethics

and other non-technical issues, they still possess the potential to uplift the industry by decreasing maintenance time and advancing safety levels.

The requirement of such systems is that they can learn/plan and act effectively after such deliberation so that behaviourally they appear self-aware, self-adaptive and self-optimized. Thus, narrowing the gap between the weak AI (agents can be made to act as if they are intelligent) and strong AI (agents can be made to think so they can represent human minds) (Armstrong, 2016). There are several architectures and approaches to embodying systems with intelligent Autonomous behaviour.

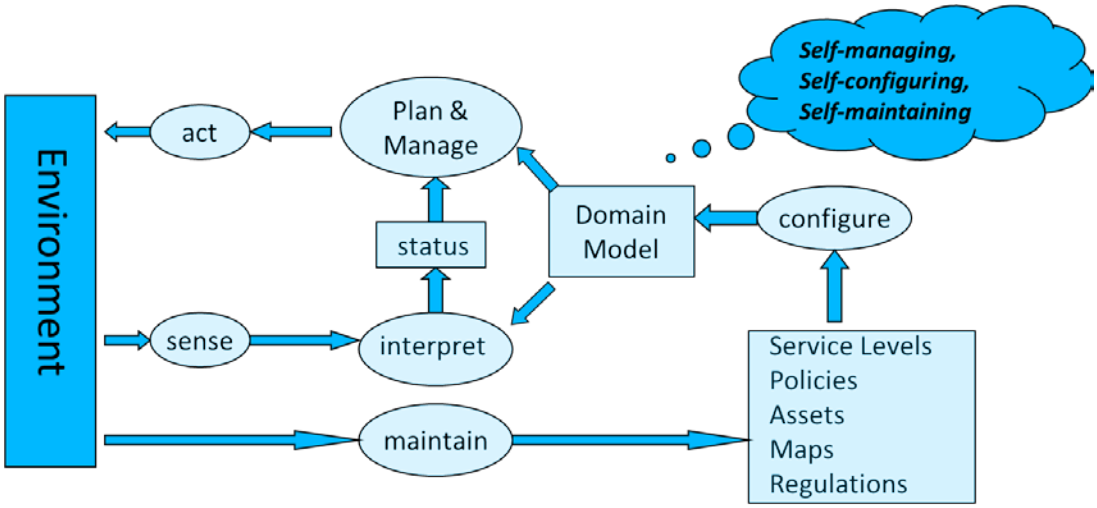


Figure 1.2: Autonomic Architecture (McCluskey 2015)

AI Planning or Automated Planning (AP) - a form of general problem solving, is a pivotal task that has to be performed by every autonomous system. The AP community has demonstrated a need to uplift planning systems from small problems to capture more complex domains that closely reflect real life applications (e.g. planning space missions, fire extinction management, and operation of underwater vehicles) - a way to satisfy the aims of Autonomic systems. Generally, AP techniques require an extensive domain model description, which is constructed by using Knowledge Engineering (KE).

An action model for a particular domain encodes the knowledge of the domains in terms of actions with pre and post conditions. Each action contains the relevant properties of the objects taking part in that action. In a centralised approach, this domain is represented as a knowledge base and automated logical reasoning could be used to determine acts. Specifying actions' description by hand for planning domain models is time-consuming, error-prone and still a challenge for the AP community.

Rationale of the Research

Learning is fundamental to autonomic behaviour and it can be defined in many ways. From the point of view of Machine Learning, it is defined as a change in behaviour through learning to allow improvement in performance. Attaining such capability for automatically learning domain models for Automated Planning (AP) engines is what triggered research into developing automated domain-learning systems from training data. This builds the requirement for knowledge engineers to integrate the system of automated learning of domain models, for AP, to attain the properties of Autonomic systems. In order to improve the performance, intelligent machines can learn for two different purposes:

i. Refine the knowledge they already have through automated skills acquisition
ii. Learn altogether new knowledge from the environment/training through automated knowledge acquisition.

This research is motivated by the second point mentioned above i.e. to automate knowledge acquisition from example training data to automatically learn domain models for Automated Planning. Some authors are of the view that to be able to plan in unforeseen, dynamically changing and unpredictable environments, it is not possible to create an action model a priori and agents must be able to incrementally adjust their existing model. Some other systems even use less ambitious motivation that such automated learning systems can only support human-driven knowledge acquisition (Cresswell et al., 2013). In recent research, a few systems have proved effective in developing domain models by learning from training data, but they require additional planner oriented information such as a partial domain model, initial, goal and/or intermediate states (discussed in the literature review). Hence, a question arises whether we can learn a dynamic domain model from real-time action sequence traces only.

The work presented in this thesis concerns the area of automated acquisition of a full or partial domain model from one or more examples of action sequences within the domain under study. This research project extends an area of the most promising line of work, which is connected to work published in an REF Journal paper and the awarded EPSRC project to Huddersfield in the national AIS robotics programme. This research aims to enhance the LOCM (Cresswell et al., 2009) system and to extend the method of Learning Domain Models for Automated Planning Engines via Plan Traces first published in 2009 by Cresswell, McCluskey, and West. The LOCM method is unique in that it requires no prior knowledge; however, it can produce a domain model from training data in the form of sequences of actions. Its main drawback is that it does not produce static domain knowledge. The underlying static structure or knowledge of the domain cannot be

dynamically changed, but it affects the way in which actions can be performed e.g. predicates representing the connections of roads (map) or the fixed stacking relationships between specific cards in card games. Complete details of the LOCM system have been provided later in section 3.2.

A key aspect of the research presented in this thesis is to enhance the LOCM technique with the capacity to generate static knowledge. A test and focus for this PhD is to make the LOCM able to learn static relationships in a fully automatic way in addition to the dynamic relationships that the LOCM already learns. To the best of our knowledge, no domain learning system has previously been developed with the aim to learn merely from a set of action traces.

The intention is to provide an implementation of a software control architecture which is competent in working towards the production of the complete domain model, functionally rich in learning maximum static knowledge in less runtime, behaviourally diverse to handle different types of static knowledge, and which encourages and readily facilitates extensive experimental evaluation.

Motivation

In 1959, Arthur Samuel defined Machine Learning (ML) informally as the process that gives computers the capability to learn knowledge without being explicitly programmed (Samuel, 2000). It is hardly surprising that ML has often been used to make changes in systems that perform tasks associated with AI, such as perception, robotics or planning. Since the beginning of research in AP, ML has been a useful area to overcome knowledge acquisition problems, as knowledge acquisition and representation is important for the success of an application. A comprehensive survey of AP systems up to 2003 that benefit from ML can be found at (Zimmerman and Kambhampati, 2003), while Jimenez et al. (Jiménez et al., 2012) provides an extensive ML review for AP up to 2012. The focus of such reviews and surveys is to specify the updated research in the area, mention the gaps that researchers should concentrate to fill and to improve the overall performance of ML.

Among several other ML tools and methods, LOCM is a generic tool based on a machine learning induction algorithm and it can be used in a range of application areas. There are many existing systems that share the same motivation and these are discussed in chapter 2. The research in this dissertation provides a methodology for learning static knowledge for AP domains from training data. The aim is to provide an implementation of a method which is competent, boosts and facilitates an easy and automated availability of AI Planning domain models. In addition to overcoming the knowledge acquisition bottleneck (discussed in chapter 2), this desire is motivated by several factors mentioned below:

1. To help planning systems become more autonomous and make them able to adapt and plan in unforeseen domains they must be able to learn a new domain model with the least available input resources.
2. To make intelligent agents able to debug/refine already generated domain models and teach themselves to grow and change. It is difficult to generate a correct domain model at the first attempt so agents must be able to incrementally adjust their existing model to deal with unexpected changes.
3. To support the domain modelling process. Knowledge engineering of complex domains by hand into planning languages such as PDDL (Malik Ghallab, 1998) is an inefficient, tedious and error-prone task. This needs a human expert in domain knowledge to type in action schema which can take hours and could also cause unintentional errors. So human-driven knowledge acquisition and maintenance tasks require automated support if not fully automated.
4. To support domain-independent planners by producing the domain specific/dependent knowledge.

This research work is aimed at automating and enhancing the domain model learning process and learning domain-specific knowledge through plan traces (training data). The dissertation searches for an answer to the question – Are there any inductive generalisation techniques to hypothesise the static part of domain model given enough examples in input in the form of training plans? In an attempt to answer this question, this thesis presents a new method - The ASCoL (**A**utomatic **S**tatic **Co**nstraints **L**earner), a generic method which demonstrates the feasibility of automatically identifying domain-specific static relations/constraints between object states. As Input, ASCoL considers application knowledge in the form of training plans in a range of application areas.

ASCoL exploits graph analysis to learn static relations automatically in order to enhance planning domain models and to strengthen the performance of domain-independent planners. Here 'graph' is the classical mathematical notion in graph theory i.e. the structure represented by vertices that are connected by the set of edges representing connections between the vertices. The static information is not explicitly present in the training data (plan traces in our case) and it is a challenging task to learn such static constraints from them. ASCoL has been tested against fifteen different International Planning Competitions (IPC) domains (ICAPS) and on different types of goal oriented and random walk plans as input training data and has been shown to be effective. In most of the domains, ASCoL is able to produce all the static facts that are present in the benchmark domains.

This thesis can be viewed as a step towards an increase in the autonomy of domain construction which would allow AP to be embedded into autonomous agents so that they can plan more autonomously.

Novel Contributions of Thesis

To the best of our knowledge, prior to this work, no fully-fledged computational PhD programme has been conducted for Learning Static Knowledge for AP Domain Models. The novelty of this research lies in attempting to learn only from the set of plan traces by applying graph theory coupled with Knowledge Engineering through generalisation concept. The summary of the key thesis contributions to the subject area is as follows:

- ASCoL (**A**utomatic **S**tatic **Co**nstraints **L**earner) - A state of the art approach to effectively learn static relations as a useable knowledge for a wide range of problems, by exploiting knowledge from plan traces only; using a two-staged domain enhancement process that first learns missing static facts for an action model and then embeds those facts in the partial domain model to get a working PDDL domain model
- A useful debugging tool for improving existing models, which can indicate hidden static knowledge helpful for pruning the search space
- Combined with LOCM, ASCoL is a useful tool to produce benchmark domains
- Identification of basic categories of static knowledge and their impact on fact learning systems
- Model sets of domains action sequence – ASCoL Model Dataset
- The introduction of eight criteria identified for the analysis and comparison of the KE tools for inducing domain models. The purpose is to objectify their functionality from different perspectives.

The work presented in this thesis is inspired by on-going recent research in this area that has been mentioned in the rationale. It is expected that contributions of this novel research will attract more researchers towards the subject. We intend to further extend this work and make the products of this research freely available for reuse under General Public Licence (GPL).
Following are the international conference and workshop papers where part of the contents and results of this thesis have been published:

Contributions in Conferences:

Authors: Rabia Jilani, Andrew Crampton, Diane Kitchin, Mauro Vallati

Title: ASCoL: a tool for improving automatic planning domain model acquisition

Conference: AI* IA 2015, Advances in Artificial Intelligence

Book title: In Proceedings of the XIVth International Conference of the Italian Association for Artificial Intelligence

Location: Ferrara, Italy

Date: September 2015

Contributions in Workshops:

Authors: Rabia Jilani, Andrew Crampton, Diane Kitchin, Mauro Vallati

Title: Automated Knowledge Engineering Tools in Planning: State-of-the-art and Future Challenges

Workshop: The Knowledge Engineering for Planning and Scheduling Workshop (KEPS)

Book title: In Proceedings of the KEPS, ICAPS

Location: Portsmouth, New Hampshire, USA

Date: June 2014

Authors: Rabia Jilani, Andrew Crampton, Diane Kitchin, Mauro Vallati

Title: Automated Acquisition of Domain Specific Static Constraints from Plan Traces

Workshop: 32nd UK Planning and Scheduling Special Interest Group (UK PlanSIG)

Location: School of Computing, Teesside University, UK

Date: December 2014

Authors: Rabia Jilani, Andrew Crampton, Diane Kitchin, Mauro Vallati

Title: Have a Little Patience: Let Planners Play Cards

Workshop: 34th UK Planning and Scheduling Special Interest Group (UK PlanSIG)

Location: School of Computing, University of Huddersfield, UK

Date: December 2016

Thesis Structure and Highlights

This section gives details of how the remainder of the thesis is structured. *Chapter 2, Background Studies and Literature Review,* presents the theoretical underpinning of the general subject areas around this thesis including Automated Planning, Knowledge Engineering (KE) Graph Theory concepts in KE, and some more important definitions and reviews of previous work on learning domain models. KE section also introduces the general architecture of Domain Model Learning (DML) systems, their approaches, the bottleneck and challenges that automated knowledge engineering has and would be expected to deal with.

Section 2.4 and 2.5 reflects on the Scope of our Research and reflects on how it is challenging to learn from plan traces and what problems a knowledge engineer has to deal with using this input, respectively. The discussion also includes the types of plan traces we used based on their production method. Moving on, section 2.7 introduces the problem domain that this research exploits for a running example to explain the methodology in later chapters. Towards the end, it includes the related work in the area. The background contents explained in this chapter are useful in order to aid understanding of the methodology and evaluation of the results described in Chapters 5 and 6.

Chapter 3 discusses approaches to learning in autonomous systems and describes in more detail the choice of approach that has been adopted in ASCoL for learning static knowledge for planning domain models. It also introduces the LOCM family of algorithms that provides the baseline for this thesis along with the additional experimental work we performed with LOCM in order to investigate the behaviour of LOCM.

Chapter 4 presents a brief overview of the automated tools that can be exploited to induce or assist in inducing planning domain models. After reviewing the literature on the existing tools for Knowledge Engineering, it also includes a comparative analysis of these systems. The analysis is based on a set of criteria. The aim of the analysis is to give insights into the strengths and weaknesses of the considered systems and to provide input for new, forthcoming research on KE tools in order to address future challenges in the automated KE area. Towards the end, a section discusses the general structure, guidelines, and recommendations that can be derived based on the review and assessment of the seven different state-of-the-art automated KE tools

Chapter 5, this chapter introduces ASCoL – the outcome of this research and core methodology, in detail supported by diagrams and other visual aids to support readers'

understanding. A graphic walkthrough of the developed methodology has been provided at the start of the chapter to give an overview. The theoretical foundation that provides the base of the proposed system is given and the multistage methodology has been explained. The same chapter informs on the implementation details of the developed system as evidence of the concept of this research and includes the software engineering procedures used to design the software system. All these concepts have been described along with running examples for enhancing understanding.

This chapter includes a section which demonstrates the combined usage of both ASCoL with Wickler's system (Wickler, 2013) for Analysis of Domain Models in Terms of Static Graphs. Another section also covers the Argument for Extracting Same-Typed Static Relations. A testing section has been provided towards the end which includes the investigation we conducted to verify the quality and performance of the piece of software developed.

Chapter 6, Evaluation, as the name indicates, provides a thorough description of the empirical evaluation system. It describes the experimental setup, evaluation metrics and some interesting dataset domains. It also includes the corresponding plan traces used for experimentation and the overall evaluation layout, i.e. empirical evaluation measures adopted and the technical details used for collection of data.

This chapter also discusses the learning behaviour of ASCoL for particular types of static facts. It also explains the combined contribution of ASCoL and LOCM for producing useful benchmark domains for planners. Experimental results along with visual aids used to explain the results have been stated towards the end of the chapter. A separate section is dedicated to the explanation on how different types of plan traces can affect the ability of both LOCM and ASCoL in generating domain models and static knowledge, respectively. Towards the end, to emphasise the performance of the developed technique, analysis of some of the significant results has been presented using multiple examples from experimentations. Specialised cases have been discussed in much detail. Experiments have been presented in a dry-run manner where graphical divisions like tables, screen captures and variation graphs are used to assist understanding. Analysis and discussion on each of the result subsets have been presented along with an example. Investigations have been classified on certain factors for useful comparisons. Certain trends have been found for each class of results that are described using graphical aids for improved understanding.

Chapter 7 - *Extended Uses of ASCoL* combines the ASCoL technique and its application to other systems. The first section presents the analysis we carried out for domain models by combining the ASCoL system with the system of Wickler. Section 7.2 describes the combined use of ASCoL with the LOCM system in order to generate domain models corresponding to a number of solitaires and exploit the solitaire card games as a pool of interesting benchmarks.

The closing *Chapter 8* – *Conclusion & Future Work* of the thesis describes the conclusion of this research. Some future work directions have also been discussed. This contains our remarks on how the ASCoL System can assist autonomic domain model learners from further research in specific directions and what other factors can be worked on to extend the same research technique. It also discusses the potential applications of the research on the subject area and how some contributions of the thesis can be reused for similar domains.

Summary

This chapter gives a brief overview of this research, its rationale and the motivation behind the work presented in this thesis. It also informs the reader about the novel contributions of this thesis. Finally, it provides an outline of the thesis structure giving highlights for each chapter.

Chapter 2 - Background and Literature Review

2.1 Theoretical Underpinning

The key attribute of theories of autonomous systems relies on the fact that for an agent's internal representation there must be an implicit domain model to follow. The essential part of a domain model is the representation of the set of actions that a planner can reason about and the elements that support the specification of actions.

In the centralised approach, the domain model can be represented as a knowledge base and automated logical reasoning could be used to determine the right sequence of actions for undertaking any activity. Synthesising domain models and domain-specific constraints by hand for Artificial Intelligence (AI) planning domain models is time intense, error-prone and challenging. It can lead to errors in the domain model which can cause errors in plan generation. Therefore, Knowledge Engineering (KE) for planning domain models is a topic that is receiving active research attention in the Automated Planning (AP) community. Building such models from scratch is an exceedingly difficult task that is usually done manually. To the best of our knowledge, all the domain models used in the past International Planning Competitions (IPCs) have been hand-crafted, and also those which are used in major applications e.g. in remote agent applications and automated manufacturing.

To alleviate the problem of encoding domain models by hand, a number of techniques are currently available for automatic domain model acquisition; many of them rely on example plans for deriving domain models. Significant differences can be found in terms of the quantity and quality of the required inputs. Amongst others, the LOCM and LOCM2 (Cresswell and Gregory, 2011) systems require as input some plan traces only, and are effectively able to automatically encode the dynamic part of the domain knowledge. However, the static part of the domain is usually missed, since it can hardly be derived by observing transitions only. The static part of the domain is the current underlying structure of the domain and is not normally changed by the planner, but it affects the way in which actions can be performed e.g. predicates representing the connections of roads; the level of floors in the Miconic domain, or the fixed stacking relationships between specific cards in card games.

Based on these premises and using the principles of graph theory, the research in this dissertation solves a learning problem with the provision of a methodology for learning static knowledge for AP domains from training data (Chapter 5). Complete details and exceptions about the concept of static facts in regards to the research in this thesis is discussed in Section 2.2 – The Scope of this Research.

This chapter presents the theoretical underpinnings that lays the basis of this research. It also introduces the main research areas with the emphasis on techniques of KE for AI planning involved in the development of the proposed system. The aim of this chapter is to provide a reader with the background within which the contributions of this research lie. It also reflects on how this work links up to recent and past research in the closely related areas. It is not intended as a comprehensive analysis of KE and AP, but to provide a context to easily understand the contribution of the thesis while providing the background concepts in the areas. Specific references have been provided to link each idea to relevant literature.

This chapter presents a step-by-step explanation, supported by diagrams and other visual aids to support readers' understanding. We define some of the terms commonly used in the planning literature and describe some of the survey results in the area. We begin with the sections about Automated Planning and a brief outline of its techniques followed by a general overview of Knowledge Representation (KR) and Knowledge Engineering, more specifically from the point of view of AI Planning. Later on, the concepts of domain knowledge classification and graph theory, exploited in the proposed system, have been introduced. Further ahead, the chapter formally describes the learning problem this thesis addresses while reflecting on how it is challenging to learn from plan traces and what problems knowledge engineer has to deal with when using this input. Section 2.5 introduces the problem domain we exploited as our running example to explain the methodology in upcoming chapters. Towards the end, it includes the related work in the area.

I - Automated Planning (AP)

This section discusses AP concepts that are related to our research including assumptions in planning, classical planning, different classes of planning, STRIPS and other planning techniques, an overview of different kinds of planners and general planning problem.

The early phase of AP can be tracked back to a General Problem Solver system (GPS) developed in 1963 by Allen Newell, J.C.Shaw and Herbert A.Simon (Newell and Simon, 1963). It divides a problem by designing and achieving sub-goals, before achieving the final goal to minimise the differences between the current state and the goal state. In AI, the problem solving relates to synthesis. It can be defined as systematic search through a range of actions to reach some desired goals.

Automated Planning – a form of general problem solving - has been a major research discipline in Artificial Intelligence for more than five decades. AI Planners take action models and problem specifications as input to generate solution plans. It can be defined formally as:

Definition 2.1 A deliberation process that reasons with knowledge (representation) of actions from a knowledge base and arranges them in order by predicting their outcomes to display some rational behaviour in an explicit way.

It produces a full plan for doing or accomplishing some task starting from some initial state of the world before the task to achieving desired state of the world also known as goal states i.e. legal moves or action selection that transform states.

Definition 2.2 A solution to a planning problem, which is a sequence of legal actions that maps the initial world state into a goal world state is known as a Plan. A Plan can be imagined as a state transition graph with actions as nodes. Ideally, this sequence of actions is then given to an agent, or some other form of effector, which can execute the plan and generate the anticipated output.

In Scheduling nearly all cases have to deal with concrete temporal assignments of activities to resources, whereas Planning mainly deals with the order in which the activities have to be performed, i.e. planning focuses on ”what has to be done” whereas scheduling focuses on ”when this has to be done” (Sauer, 2003). The two are closely related but conceptually different problems. Scheduling is considered as part of planning as it provides the allocation of resources taking various constraints into account. For example, brewery production line scheduling, aircraft crew scheduling, shipyard scheduling for ship building etc. Both Planning and scheduling are important components of rational intelligent behaviour, thus, the methodologies have been applied in a variety of application areas. Application areas include robotics and autonomous vehicles (Menze and Geiger, 2015), automatic control of industrial processes (Engineering, 2015), logistics (Davidsson et al., 2005), spacecraft (Chien et al., 1999), underwater robots (Petres et al., 2007), crisis management (Nunamaker Jr et al., 1989), user support for project planning (Levitt et al., 1988), planning in forest fire (Avesani et al., 2000), generation of control programmes like chemical plant control (Engineering, 2015) , web services composition (Bertoli et al., 2010), simulation of game software (Naveed et al., 2012), business processes modelling (González-Ferrer et al., 2009), autonomous agents, spacecraft mission control like NASA’s Mars Rover (Feng et al., 2004), multiple battery usages (Fox et al., 2011) and many more. Languages used to describe planning and scheduling are often called action languages.

To generate plans, planning engines search the knowledge base (that contains application knowledge) to find potential actions to achieve the goals. Application knowledge is crucial for the efficiency of planning systems, and for the correctness of resulting plans. It requires substantial effort as well as skilful experts to encode and maintain an error free, correct planning domain model and action constraints. This is where

Knowledge Engineering and Automated Planning meet. A huge reservoir of knowledge about the required domain is provided by Knowledge Engineering tools and it is known as domain/application knowledge. The successfully generated plans can be used as solutions to the desired problem in self-learning systems to enable autonomic properties in the system.

Assumptions in Automated Planning

This sub-section discusses the common assumptions that AP has regarding the separation of its important components to deal with representational, algorithmic and maintenance concerns.

Planners can be classified into two main types: domain-dependent and domain-independent planners. As the name indicates, domain-dependent planners are designed to work in specific domains only. When such planners are needed to work with a different domain, special modifications are needed to be made in the planning system. Examples include MARS Rover, JUBBAH and AsbruHPD. Keeping in mind the shortcomings of domain-dependent planners, AP is interested in a domain-independent approach to planning. To solve a specific problem, these planners take problem specifications and domain knowledge as inputs. These planners also provide the means to make general use of constraints and resources to represent domains and actions. The domain-independent approach also supports the assumption that there should be a logical separation of the planning engine, heuristics and the fundamental domain model (or any other software that takes part in the planning application). This has been represented in Figure 2.1.

This assumption has several logical reasons in addition to avoiding a maintenance problem. One is to separate the development, testing, debugging and validation independent of both components, so that the concept of domain-independent planners could be strengthened. This also enables development of generic domain models which can be reused across related areas of Research and Development. Similarly, this assumption produces some concerns like domain model design and development is affected by what planners can deal with and that leads to inadequacy in planning systems. Due to IPC in 1998, much development has been occurring in AP to produce competent planning algorithms but tools and techniques to engineer action models suffered from somewhat less attention.

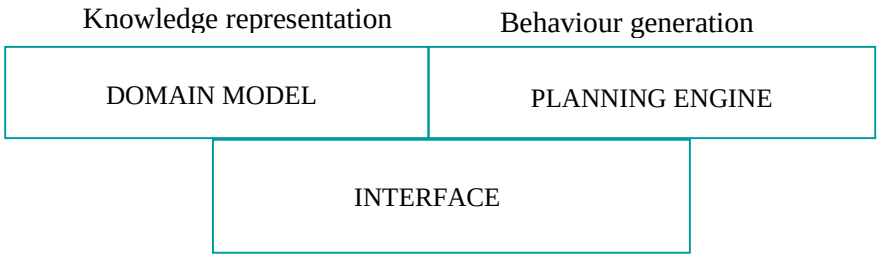


Figure 2.1: Logical Separation between Planning Engine and Domain Model

Classical planning

Numerous planning techniques have been introduced since the earliest days of AI planning. McCarthy and Hayes in 1969 proposed the classical approach to planning called situation calculus (McCarthy and Hayes, 1969). It uses first-order predicate logic (FOL) to reason about the actions and theorem proving to find plans. The benefit of the situation calculus is that it provides a theoretical framework to represent actions with a clear semantics by "reify" situations. To reify means to treat something non-concrete or abstract as an object. This introduced the concept of variables and constants in the logical language arranged over the given situation or state of the world by using them as the arguments of a predicate. It introduced the concepts of result function, effect axioms and frame axioms which enhanced the power of logical representation.

Definition 2.3 In order to produce a plan, classical planning deals with finding a partially or totally ordered sequence of applicable actions transforming the static, deterministic and fully observable environment from some initial state to the desired goal state.

Static, deterministic and observable are different characteristics of the environment that can be replaced with their opposites but this does not apply in classical planning definition. These environmental characteristics can be elaborated as:

- Static characteristic of the environment is when no external measure can interfere and change the state of the world
- The deterministic environment is when the effects of an agent's action are predictable.
- In the observable environment, the agent can observe and understand the state of the world.

In the classical representation, atoms are predicates used in operator definition. States are defined as sets of ground (positive) atoms.

A planning operator:
op = (name(o), pre(o), eff−(o), eff+(o))

is specified such that:

name(o) = op name(x1,...,xk)

(op name is a unique operator name and *x1,...xk* are variable symbols (arguments of certain types) appearing in the operator). *pre(o)* is a set of predicates representing the operator's preconditions which need to be true in order for action to execute. *eff−(o)* and *eff+(o)* are sets of predicates representing the operator's negative and positive effects (or the outcome effect of the action execution).

Actions are ground instances of planning operators. An action:

A = (pre(A), eff−(A), eff+(A))

is applicable in a state s if and only if pre(A) ⊆ s. Application of A in s (if possible) results in a state (s\eff−(A)) ∪ eff+(A).

A plan is a structured collection of actions, it should be applicable and correct. For applicability, all the preconditions for the execution of the first action are satisfied in the initial state, while the preconditions of following actions in the sequence should be satisfied by the corresponding intermediate state. Temporal projection analysis can establish if all the actions can be applied in the order stated by the plan. The first state considered in the projection is the problem's initial state. Action applications provide intermediate state descriptions. The correct plan is achieved when the effects of final action in the plan satisfies the goal conditions.

A plan can be called an optimal plan if no other shorter or more cost effective plan exists for the problem. Besides working on the optimisation of certain features, the planning problem can be enriched by including additional features, such as temporal or uncertainty factors. A plan is sometimes also called an **action sequence**, as a plan is actually a particular sequence of actions that are taken from Domain model. A plan is also referred as **training data**, **training sequence, action sequence** or **plan trace** in terms of self-learning systems and in this thesis as well.

The International Conference on Planning and Scheduling ICAPS is a platform for the advancement of the field of automated planning and scheduling. It also organises an International Planning Competition IPC held every two years since 1998 in order to expand

and bring the AI community's attention to improving automated planning algorithms. In the competition, automated planning planners compete against each other based on a number of pre-set criteria.

Different classes of Planning

This sub-section discusses different classes of planning which depend on the characteristics of the environment in the problem-solving task. The work in planning techniques has formed the majority of AI research in planning. In the simplest possible version of planning i.e. classical planning, problems always have the following world characteristics:

- Distinct known initial state
- Finite world (rather than infinite)
- Instantaneous (rather than durative) actions
- Completely observable (rather than partially observable)
- Static, closed world (rather than open, dynamic world)
- Deterministic actions (rather than non-deterministic) which can only be executed one at a time
- Use of single agent
- Infinite resources

The field of AI planning has grown beyond the classical planning state model that has complete knowledge of the world to include many decision theoretic ideas such as Markov Decision Process.

There are planning models with varying environmental characteristics and dynamics including imperfect successor state information e.g. deterministic dynamics require the current state and action to determine the unique successor state whereas in non-deterministic dynamics the combination of the current state and action may lead to several possible successor states. Similarly, in probabilistic dynamics, there is a probability distribution over the possible successor state. This is because planning under uncertainty has incomplete knowledge of the world in the form of the randomly determined stochastic outcome of the action and that outcome cannot be predicted precisely. Besides, state variables either can be discrete or continues. For discrete variables, the number of possible values can be either finite or infinite.

Similarly, rather than finite state space, the horizon of state space can be infinite and continues in terms of time and resources available. In Temporal and cost-based planning, time and cost are contributing factors to optimise time delays and minimise the cost involved, respectively, besides reaching the designated goal states.

Another class of problems is where the observability of states varies in a way that the current planning state can have ambiguity in its observability. Unlike a full observability environment, it can be partially observable or not at all. Planning algorithms are much simpler with the fully observable environment than partial or no observability at all.

An assumption that plays an important role in knowledge representation is the closed-world assumption (CWA), which states that a statement, which is true, is known to be true. Thus, what is not currently known to be true is false and this is known as open-world assumption (OWA). Choice between CWA and OWA defines the understanding of the actual semantics of an expression with the respective notations of concepts.

Another contributing factor is the number, nature and approach of planning inside the agents taking part in planning. There can be more than one agent taking part in the planning process, which can be either selfish or cooperative. Agents can have an independent planning process or may use centrally constructed plans.

Planning problem

In the context of this thesis, we use the closed world planning problems, which contain sets of objects that are mostly physical objects with the possibility of being in different states after undergoing different state transitions. Beside dynamic objects, there can be static objects. Static objects are used to model objects that never change their values, for instance, a location in an action remains static although dynamic objects can move between locations by action transitions in the application domain.

In early days of automated planning, STRIPS (Fikes and Nilsson, 1972) was the most popular representation language. Later PDDL (Malik Ghallab, 1998) was developed for the first IPC and since that date, it has become very popular in the planning community and, a majority of planning engines use it. More detail on PDDL and its variants is provided in the upcoming section as Domain Representation Languages. A PDDL planning problem consists of two parts:

1. Domain Model
2. Problem Model

1. **Domain Model:** A world state is a set of the states of each object in the problem. A planner requires a description of the world state space and actions (called events and action schema in the literature) that can be executed. Operator schemata describe actions in terms of preconditions (pre-transition states) and effects (post-transition states). In operator schema, a set of variables can be replaced by constant values to instantiate operators, which indicate actions in the

planning outcome. We use the term actions for ground instances of operators (schema or instances). An action that is directly executable by a planner is referred to as a primitive action or, simply, a primitive. We shall be considering primitive instantaneous actions in this thesis.

Descriptively, a complete PDDL domain model (hereinafter called domain description, action model, operator schema or action schema) is specified as domain name, domain requirements, object-type hierarchy, constant objects (if any), sets of predicates (logical facts and constraints), functions (for hierarchical domains) and a set of domain operators. Typed arguments of each operator are instantiated during operator execution with objects, preconditions, and effects (could be conditional). Preconditions and effects in an operator contain typed variables. On operator instantiation, specific objects of particular types replace the variables. The disjunction, conjunction, negation, or quantification of logical expressions constitute preconditions of an operator. Effects are the list of predicates to be added or deleted from the instantiated action.

A state of an object is the object name, followed by a set of predicates that are true of the object. The resulting description that contains all this required domain knowledge in the form of predicates and actions in known as a domain description and this theory is known as a domain theory in planning. The extent of knowledge related to a planning application is called the domain while the symbolic illustration of this knowledge is called a domain model (McCluskey, 2000).

In (McCluskey, 2000) a domain model is defined as a structure of knowledge that an intelligent agent can exploit to deduce rational actions about the domain area it represents. Figure 2.2 shows an example of the *Blocks* domain. The Blocks domain is a very common (IPC) hand-written benchmark domain in AI planning where four unique actions are used to arrange blocks in an order specified in a given problem. Benchmark domain models are constructed to evaluate the current state of the research, and are developed to match the solution methods dominant in a given community (e.g., IPC problem domains).

```
(define (domain blocksworld)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)(ontable ?x - block)
           (clear ?x - block)(handempty)(holding ?x - block))
  (: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)))
  (:action put-down
           :parameters (?x - block)
           :precondition (holding ?x)
           :effect(and (not (holding ?x))(clear ?x)(handempty)
                       (ontable ?x)))
  (:action stack
           :parameters (?x - block ?y - block)
           :precondition (and (holding ?x)(clear ?y))
           :effect(and (not (holding ?x))(not (clear ?y))
                  (clear ?x)(handempty)(on ?x ?y)))
  (:action unstack
           :parameters (?x - block ?y - block)
           :precondition (and (on ?x ?y)(clear ?x)(handempty))
           :effect(and (holding ?x)(clear ?y)(not (clear ?x))
                  (not (handempty))(not (on ?x ?y))))
)
```

Figure 2.2: The Blocks Domain

Other concepts in a planning domain model include Tasks, which are the sets of multiple actions to be executed consecutively, for instance, to make a cup of tea. Usually characterised by numeric values, resources are the available quantities used by the system.

2. **Problem Model:** If S is the set of states of the environment and A is the set of actions that shows the transition between these states then a problem file can be described as having the following three parts:

Objects: used in describing a problem to the planner.

Initial states: to indicate the set of facts s0 ϵ S that indicates the world state at the beginning of planning process.

Goal states: to indicate the set of facts G ϵ S to show the desired state in which a goal condition is fully or partially satisfied.

Following is an example of the simplest planning problem file for planning in the *Blocks* domain. In objects, it needs three blocks, a table, and a hand.

```
(define (problem ThreeBlocks)
    (:domain Blocks_Domain)
    (:objects
      A - Block
      B - Block
      C - Block
      H1 - Hand
      table1 - Table
    )
    (:init
      (ontable A table1)
      (ontable B table1)
      (on C A)
      (handempty H1)
    )
    (:goal
      (and
        (on A B)
        (on B C)
        (ontable C table1)
        (not (clear C))
        (not (clear B))
        (clear A)
        (handempty H1)
      )
    )
)
```

Figure 2.3: Blocks Problem

Figure 2.4 shows a pictorial form of blocks problem that is shown in figure 2.3 with the initial and the required goal states. Namely, A, B and C are block objects, H1 is hand object and table1 is a table object whereas unstack, stack, pickup, and putdown are action names in the input domain model.

The example plan (definition 2.2) generated by an AI planner for solving the mentioned problem is Unstack (c, a), putdown(c), pickup (b), stack (b, c), pickup (a), stack (a, b). This sequence of actions is called total-order/linear plan i.e. an exact ordering of actions (more details on types of planners/plans is covered in upcoming Planners subsection). A simple non-optimal plan is correct if all the preconditions of its actions are satisfied before the execution of that action. A correct plan achieves its goal mentioned through goal states.

Figure 2.4: Typical Blocks World problem

Common names for the plan generation process are plan synthesis and plan construction. Figure 2.5 shows a typical view of plan generation in AI planning. A planner is called domain independent in the sense that the plan representation language and plan generation algorithms are expected to work for a reasonably large variety of application

domain descriptions without requiring any changes in planner algorithm or representation language.

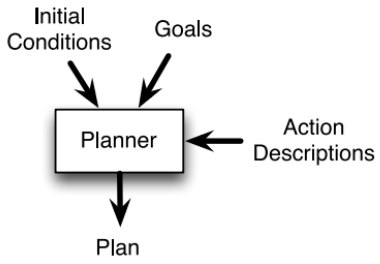


Figure 2.5: Planning as an independent component

Knowledge elements of Planning Domain Model

A planning domain model requires two major knowledge elements to be acquired or learnt:

1. **Dynamic Knowledge:** a set of parameterized action schema representing generic actions and their resulting effects in the domain under study.

2. **Static Knowledge:** relationships/constraints that are implicit in the set of operators and are not directly expressed in the plans. These can be seen as predicates that appear in the preconditions of operators only, and not in the effects. Therefore, static facts and objects never change in the world under closed world assumption, and that is why they do not appear in state transitions and cannot be relocated but are essential for modelling the correct action execution. In many domains, there is static background information that remains static even when the given (problem) task changes, for example, in transport domains the map of an area is the static fact whereas, the routes of a traveller might change depending on the purchases she needs to carry out each day.

The focus of this thesis is on learning static relationships for domain models, but there is no existing standard definition of static relationship in the planning literature. Different authors define static relationships differently using their own idea of domain structure. For instance, Wickler in (Wickler, 2011) defines static knowledge as:

Let $Op = op_1, op_2, \ldots, op_n$ be a set of operators and let $Pr = Pr_1, Pr_2, \ldots, Pr_n$ is a set of all the predicate symbols that occur in these operators. A predicate Pr_i in Pr is fluent *iff* there is an operator op_j in Op that has an effect that changes the value of the predicate Pr_i. Otherwise, the predicate is static. Gregory et al in (Gregory and Cresswell, 2015) define static relations as *restrictions on groundings of operators* in the domains. In other words, a static relation for each operator is a collection of all the valid groundings for that operator.

The static relation becomes true only with the allowed values of its parameters where the relationship between those parameters never change. For instance, consider a general transport operator *Drive* which has a static predicate *Connected (?location1 ?location2). Drive* can only be executed if the static precondition is true. This depends on whether the values of the two location parameters are connected in real life to drive from *location1* to *location2*.

Domain Representation Languages

The Knowledge Engineering process involves the use of a variety of representations. It has long been recognised that there can be a variety of encodings and languages for a single domain model. However, the open question is which of these is the best? The choice of encoding and language partially depends on the requirements of the planning application itself.

A domain modelling language should have some attributes. Tools to check its operation should support it. Moreover, it should be sufficiently expressive to capture complex scenario of the real world, customizable and structured to contain complex operator definitions. It should also have the network of operators and the inference mechanism should be strong logically. In addition, it should have clear syntax and semantics and should support operational aspects of the model. This section presents existing languages used for Modelling planning domains.

PDDL

Based on most of the research done on domain model generation, the Planning Domain Definition Language (PDDL) (Malik Ghallab, 1998) and its variants are the best-known languages which are used for Modelling domains and planning problems. In the early days of automated planning, STRIPS was the most popular representation language. Drew McDermott then introduced PDDL in the AIPS-98 competition with the aim to have a single common representation language for defining domain models. It is widely used by the automated planning community due to its compatibility with most of the planning systems. PDDL is built on an extension of First Order Logic (FOL) and is an action-centred language.

PDDL has become a standard language for the AI planning community. There are the following important milestones in the development of PDDL.

PDDL 1.2

This version of PDDL was introduced on the first IPC in 1998. It divides a planning problem into two components i.e. domain model description and planning task description. It is suitable for classical planning and incorporates some additional features such as object types or conditional effects.

PDDL 2.1

This variant (Fox and Long, 2003) was presented on the 3rd IPC in 2002 where several improvements were made in the previous version of PDDL. It extends the previous version by introducing numeric constraints to perform arithmetic operations, such as evaluating distances and time required for movements, used to express quantities and levels, for instance, fuel-level, energy, distance etc.). It also introduced temporal planning with the help of durative actions. PDDL 2.1 provided the base for all upcoming PDDL versions. PDDL 2.2 was also introduced in fourth IPC with the extension of derived predicates in PDDL 2.1.

PDDL 3.0

Introduced on the 5th IPC in 2006, PDDL 3.0 (Gerevini and Long, 2005) allows the users to include hard and soft constraints in the form of logical expressions for the structure of the desired plans. Hard constraints must be true during the execution of the plan while soft constraints (preferences), if true, increase the quality of the plan.

PDDL 3.1

This variant (Helmert, 2008) of PDDL is the latest and back in 2008 and 2011, it was made the official language of the deterministic track of the sixth and seventh IPC, respectively. It presents object fluents whose range not only include numerical but also could take any object type.

Probabilistic PDDL (PPDDL) (Younes and Littman, 2004) and PDDL+ (Fox and Long, 2006) are among two useful extensions of PDDL. Presented in the probabilistic track of the fourth IPC, PPDDL supports actions with probability distribution (non-deterministic effects). PPDDL does not allow partial observability and this is why it is not useful for conformant planning. PDDL+ is specifically designed for continuous planning tasks in order to model continuous processes and events.

Another representational language is NDDL (New Domain Definition Language) (Frank and Jónsson, 2003) which is mainly focused towards space applications, developed by NASA as a response to PDDL in 2002. The main difference between NDDL and PDDL is that NDDL uses object fluents instead of propositional representation as in PDDL. The expressiveness of NDDL is able to model continuous planning tasks where time is discretized; however, there is not a large number of planning systems which support NDDL. Only NASA's EUROPA2 (Bernardini and Smith, 2007) planner supports NDDL.

A number of planning domain languages have been developed beyond the PDDL family of languages e.g. STRIPS, ADL (Pednault, 1988), and OCL (McCluskey and Porteous, 1997)) to express a domain model correctly.

Some further Concepts

Before going into a deeper discussion on the learning problem that this research addresses, it is important to have the basic knowledge about some other definitions that are important in the context of this research.

Constraints

A constraint is an entity that restricts or limits the values of variables in domain modelling (Nareyek et al., 2005). These can also be viewed as static relationships between variables that help planning systems in the quick and efficient pruning of the search tree. More than one term is in use for domain constraints in the literature. Bresina et al (Bresina et al., 1993) use domain constraints and behavioural constraints to refer to constraints. Grant (Grant, 2010) uses constraints as a generic term, and invariants to mean constraints that must be true throughout the execution of a plan. Others may use static axioms in plan generation. In database terms (Halpin and Proper, 1995) invariants are known as static constraints and validation rules. In this thesis, we refer it as static facts, static constraints or static knowledge.

Restraint is another term, the use of which is rare in AI planning literature. In the Oxford dictionary definition, constraint refers to a limitation or a restriction while restraint refers to the action of holding/controlling something. However, there is a little difference between the two terms so these cannot be used as synonyms.

In military operations planning also, restraints and constraints indicate something different. By the Joint Operations Planning (joint publication 5-0 of USA Army and Navy) Restraints are limitations in terms of things that cannot be done, while constraints indicate tasks and things that must be done (Staff, 2011).

In AI planning and other wider areas of AI, constraints indicate restrictions on variables' value or conditions. There is no notion of actions, tasks or values that must be satisfied, and they are implicitly translated into constraints. Having said that, we conclude that in planning military terms, constraints are closer to the notion of goals, while restraints are what in AI is termed constraints. In the Ferry domain, for example, the overall task is to have all the vehicles in some destination locations (military constraints -- AI goals), but the ferry cannot carry more than one vehicle per time (military restraints -- AI constraints)

No principle definition of restraints in AI planning exists in the literature. We use the term Constraints or invariants throughout the thesis in order to indicate static restrictions/predicates. Planning constraints can be applied to domain actions as in (Chapman, 1987) or (Stefik, 1981), or it can be applied to states e.g. in (McCluskey and Porteous, 1997) or in (Rintanen, 2000), or to both actions and states as in (Blum and Furst, 1997).

Existing domain-independent planning systems require domain constraints or invariants explicitly as part of the input domain model. In the operator schema, static constraints are often hand-coded in the form of predicates, which restrict the values of variables in the preconditions of actions and define certain relationships that never change in the definition of a complete action model. Providing domain constraints to the planning engine may help the planning system in the quick and efficient pruning of the search tree, or to produce optimal plans (Refanidis and Sekallariou, 2009). In such planning systems, a generated plan is correct provided the constraints of the world in which the agent is operating are satisfied.

The question is how we can automatically identify the static relations/constraints during the process of domain model learning that operators' preconditions need. For this, we base our approach on the assumption that the input plan traces (action sequence) contain tacit knowledge about constraints validation/acquisition. Based on this assumption, we can draw correlations from the data by pattern discovery using graph theory on the training input only. It is important to note that, all types of planning constraints cannot be represented by graphical conceptual schemata, but may need to be depicted as tables, logical formulae, or graphs (Grant, 2010).

Closed World Models and Instantaneous Actions

In the closed world concept, the domain represents everything under some closed boundaries. The objects and states explicitly defined in the model take part in domain model operation. For instance, in Blocks Domain, the tabletop is the only operational area of the domain. Any blocks placed beyond that boundary are not part of the world and are not mentioned in the domain.

In our work, all the domains are closed world domains that contain sets of both dynamic and static objects, which are mostly physical objects. These objects have different features and can be transformed from one state to another state e.g. to pick a block from the table by the hand of a robot, the predicate (ontable ?block) has to undergo state change from true to false. After the transition to a new state, the state becomes true and the previous state may or may not become false in the form of preconditions and post-conditions in action definition.

In this work, all the actions are instantaneous instead of durative, as time is not a factor to provide correct action sequence.

Induction

Induction is a process of generalising from past experience/examples and background knowledge (Luger, 2005). Inductive Logic Programming (ILP) is defined as an intersection of inductive learning and logic programming (Muggleton, 1994). ILP uses computational

logic as a representation mechanism and exploits background knowledge and an example set to derive a hypothesised logic program, which involves all the positive examples.

In our system, the key idea is that by taking sets of actions (with parameter objects involved in the action) from some continuous series of events (i.e. plan traces), we can use induction process by exploiting the examples of the behaviour of a types of objects. We model static constraints for the domain in the form of partially and totally ordered graphs by generalising behaviours from the examples to a relationship. The static knowledge induced in this way shares the same facts as in the hand-written domains. In fact, by using this approach we have identified some additional static predicates in some IPC benchmark domains. Additionally, learnt knowledge does not reduce the solvability of the problem but helps in an efficient pruning of the search tree in some domains when the domain is used in the AI planning process.

The key idea within these approaches is that of inductive generalisation - using examples of behaviours of a class of objects and generalising these examples to a theory about the whole class of objects.

Planners

This subsection reviews the concepts related to the working of general domain-independent planners, which depends on general problem-solving techniques to obtain a satisficing plan of good quality. Domain-independent planners have been the focus of several IPCs over the years.

Planning engines determine what actions to take and in what order so as to progress from an initial world state to the desired goal state in order to generate a plan (Fox and Long, 2006). After going through the existing resource allocated for the task and the corresponding time constraint (if any), the planner produces the action sequence to achieve the goal. There are two main types of planners: the first, which guarantee to find the optimal solution (provided they are given enough time), are called exact planners; the second type produces the best possible solution plans within a given time with no optimality assurance.

In order to deal with real life domains, AI planning has moved ahead from restrictive classical domains. Planners now focus on several new areas involving numeric computations with limited resources, time and relationships that involve probabilities (Nau, 2007). The rest of this subsection describes some of the prominent planners in past IPCs. A complete list of planners is beyond the scope of this thesis.

Planners vary in how they define their search space and how they exploit it. The state space is a collective term for all the states to search. In state space, actions act as links between these states to connect them. In state space search, forward chaining planners work by applying an applicable action to some chosen state, proceeding to a

different state in the search space by state space traversal. A sequence of actions defines a solution to a problem starting from some initial state to some state defined as fulfilling the goal. It exploits the states of the world (the plan is to be executed in) and thus is called the state-space plan. In a graph like structure, all nodes correspond to the states of the world, all edges represent the state transition, and the final plan represents the path in the search space.

Most planners developed after 1975 advanced from this approach and produced partially ordered plans in contrast with total-order planning, which produces an exact ordering of actions. In Partially ordered plans a point in the search space can be transformed into any other applicable plan when required. It works on the Principle of Least Commitment which states that the decision of action ordering should be deferred as long as possible. This is to ensure a high probability of correctness when the actions are due to take place. In action choice decisions during plan elaboration, it expands the action in the plan into its detailed hierarchical actions' sequence containing a greater level of detail and the addition of ordering constraints to guide interaction between unordered actions. The final plan is considered to be applicable if it satisfies the goal, after fully detailed action transformations, such that all the goals are achieved using a set of primitive actions and not compound actions. One such system by Sacerdoti (Sacerdoti, 1974) introduced a hierarchical abstraction planning system ABSTRIPS on the base of STRIPS.

Unlike state-space plan, partial order plans define the relationships between actions instead of predicates and states. Typically, state-based plan structures require specification of intermediate states between actions in sequence, while action-ordering representation can order two actions without any knowledge about intermediate space in between.

The first hierarchical planner was developed in 1975 and was called NOAH (Sacerdoti, 1975) that produced plans to represent the partial order of actions. Later, the Graphplan system developed by Blum and Furst (Blum and Furst, 1997) and SAT-plan developed by Kautz and Selman (Kautz and Selman, 1996) provided perception into enhancing planning productivity and efficiency. FF (fast-forward) planner (Hoffmann and Nebel, 2001) is one of the successful state space planning algorithms. To estimate the goal, it uses forward chaining heuristics. It is considered as a descendant of HSP system (Bonet et al., 1997) with substantial differences.To tackle numeric constraints, FF was extended to Metric-FF (Hoffmann, 2003) in 2003. Using local search and planning graph, LPG (Gerevini et al., 2003) can work with the durative actions and numeric constraints. LPG won 3rd and 4th IPCs. LPG-td (Gerevini, 2004) is a newer version of LPG to deal with the features of PDDL 2.2. Both LPG-td and Metric-FF planners are used for the evaluation of research mentioned in the upcoming chapters. SATPlan (Kautz, 2006) and MAXPLAN

(Xing et al., 2006), the sat-based planners won an award for the best performance in optimal planning track of 5th IPC.

LAMA (Richter and Westphal, 2010) is based on the classical planning approach and exploits heuristic forward search. It has been the winner of the sub-optimal track for IPC6 and again for 2011 in IPC7 for the satisficing track. Dae_{yahsp} (Khouadjia et al., 2013), Yahsp2-MT (Vidal, 2011) and POPF2 (Coles et al., 2011) were among the winners of the temporal track in IPC-2011.

STRIPS and other Planning Techniques

This section discusses an important milestone in the development of planning systems. The STRIPS (the Stanford Research Institute Problem Solver) planning system, developed by Fikes and Nilsson in the late 1960s (Fikes and Nilsson, 1972) was first used to control the movement of a robot called Shakey, which pushed various boxes through several interconnecting rooms.

In the STRIPS-style planning, domain actions represent a set of three logical formulae: preconditions and effects, as add-list and a delete-list. The conjunction of ground preconditions and effects describes the given state of objects and a set of possible configurations of the world. Preconditions specify conditions that must hold true before the action can be applied. Successfully applying an action means acting on its add-list and delete-list to create a new state. The generation of a new state results from first deleting all formulas given in the delete-list and then including all facts in the add-list. Figure 2.6 shows a typical STRIPS operator. It is an operator from the well-known IPC Blocks world domain. Pick-up can be used to pick blocks from the table into the robot hand. For this, preconditions to pick a block are that the block must be clear to pick up, it must be on the table and the hand to pick the block must be empty. After pickup, the block is not on the table, the hand is not empty and the block is no longer clear (delete-list), the hand is now holding the block (add-list).

Although novel planning systems advanced from this technique, this Precondition-accomplishment planning system is the most common classical planning technology. This is because this representation puts no assumptions upon the structure of the domains to which it is valid and permits the complexity of planning to the domain-independent planning algorithm.

```
Pick-up (x)
        Precondition: clear (x) ^ ontable (x) ^ handempty
        Delete List: ontable (x) ^ clear (x) ^ handempty
        Add List: holding (x)
```

Figure 2.6: Typical STRIPS Operator

There is another planning technique called Hierarchical Task Network (HTN) planning that generates plans from non-flat domain models where actions are defined in the form of networks to show dependency among them. This can be traced down to real life tasks that have built in hierarchical structure, e.g. the process of building a house. In HTN style planning, the problem is defined as a set of tasks. A task can be primitive, compound or goal task. In HTN planning, both actions and states representation are like the STRIPS. The main difference between HTN and STRIPS techniques is the way they produce plans and for what purpose they plan.

HTN planners generate plans by searching over the task networks. All the collection of tasks could be primitive in the sense that they cause a simple state transition to the world, or they may contain compound tasks. Compound tasks can be split down during the planning of primitive tasks (Malik Ghallab, 2004). To produce a final plan, HTN planning uses partial-order planning techniques and task decomposition. Using the built-in hierarchy help prevent planning search space from an exponential explosion. Although this discussion of planning is limited, it serves for the purposes of this overview.

II - Knowledge Engineering (KE)

This section describes the KE concepts that are in-line with the topic of this thesis e.g. categories of KE, Bottlenecks of Knowledge Based Systems, Knowledge Representation for Planning and finally, KE for Planning.

Knowledge is theoretical or practical information of a subject or a domain. Knowledge can be in the form of theoretical facts, concepts, procedures, models, heuristics, and examples. Types of knowledge include specific or general knowledge, exact or fuzzy, procedural or declarative.

Definition 2.4 The process by which we gather, formulate, validate and input knowledge about the objects, their relationships, and constraints, in the form of logic and code (rather than factual theory), into the computer systems, is called Knowledge Engineering (Biundo et al., 2003b).

Designing an enormous knowledge base of a domain is a tedious task in real time applications. It includes the study of application requirements and creating a model that explains the domain and then testing its correctness. KE is a fast growing segment of Artificial Intelligence in that it is transforming the way in which computers network with the world. It focuses on how machines can show the intelligent behaviour of humans. It can further be divided into three categories of activities: Knowledge Representation, Inference Generation and Knowledge Acquisition (Redington, 1985).

- **Knowledge Representation (KR)** is dedicated to representing knowledge and data structures about the domain in the form that is in-line with machine representation formalism. Its purpose is to solve complex tasks such as to represent the scenario of patience card games in a form understandable by an intelligent game playing robot for winning any random arrangements of cards.
- **Inference Generation** is related to generating new knowledge after working on knowledge representation through an inference generator.
- **Knowledge Acquisition (KA)** is concerned with building techniques and methods to make the laborious process of collecting correct and consistent knowledge from various sources, as efficient and effective as possible. The focus of this thesis focuses on this activity i.e. to acquire knowledge for domain models from available input plan traces.

After discussing the main activities of KE briefly, we now focus on the knowledge acquisition activity of Knowledge Based Systems (KBS) as that forms the core of the research presented in this thesis. Next section discusses the main bottleneck of KBS.

Bottleneck of KBS: Knowledge Acquisition (KA)

With time, KA was recognised as the one major bottleneck in the development of KBS. Figure 2.7 shows the old idea of KBS development. In (Feigenbaum, 1984) Feigenbaum's KA bottleneck mentions the difficulty of accurately extracting experts' knowledge for a knowledge base:

"The problem of KA is the critical bottleneck problem in artificial intelligence." [8, p.93]

This is because obtaining adequate high-quality information to develop an efficient and robust system is a lengthy and expensive process. As the most important source of knowledge is the area experts' experience and skill set and mind, where they have lots of knowledge in both explicit and tacit form. Tacit knowledge is difficult to recognise and describe, and this is the main challenge that provides the foundation to this research: to elicit tacit knowledge from structured data.

It is hard to capture and validate all area experts' knowledge as each expert doesn't know every detail about everything. This is why KA has become one key research field in knowledge engineering.

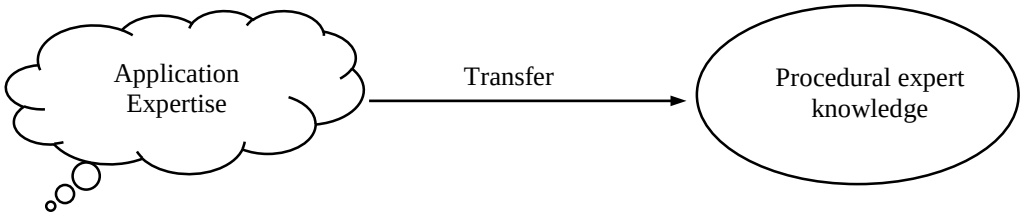


Figure 2.7: Old idea of KBS development

The purpose of KA is to build techniques and methods to make this laborious process of collecting correct and consistent knowledge as efficient and effective as possible. For the improvement of the process, various principles and techniques have been developed since the early 1980s that significantly enhance the process of KA.

Knowledge Acquisition in literature

One highly developed and structured way of building knowledge bases, which is also accepted as the European standard for knowledge-based systems, is Knowledge Acquisition and Documentation Structuring (KADS) (Waldner, 1992) developed by the University of Amsterdam in the early 1990s. KADS provided two major supporting contributions for KBS development in an industrial approach: firstly, a lifecycle to record a response to different constraints (technical and economic constraints), and secondly a set of models to structure system construction that involved analysing and transforming expert advice to machine usable form.

Later in 1994, KADS was extended to CommonKADS (Common Knowledge Acquisition and Design System) (Schreiber et al., 1994), a comprehensive operational framework that supports most aspects of KBS development. The perception behind CommonKADS is that the knowledge engineering process should be model-driven which means that in CommonKADS methodology, KBS development involves building a set of engineering models of problem-solving behaviour in the application context. In addition to including expert knowledge, this modelling also involves the characteristics of how that knowledge is exploited and embedded in an organisational KA tool, which is used in knowledge management is called SPEDE (Structured Process Elicitation Demonstrations Environment) (Shadbolt and Milton, 1999). The main aim of the SPEDE toolkit is to develop a set of commonly applicable techniques and software tools to assist the business process engineer in the task of business process improvement (BPI), to validate and interconnect the important knowledge for a variety of business process reengineering activities.

KE is a multistage process. After KA, it involves the development of an informal model that not only includes the expert's knowledge but the knowledge of the environment in which the engineered knowledge would be utilised. Generally, as a first stage of creating a knowledge base, the key process components of a system are identified. Each

component is then observed and analysed thoroughly. As a second step, data is extracted and collected around each component of the system, and finally the collected data is structured and organised to create a formal model understandable for a computer.

Knowledge Representation (KR) for Planning and Scheduling

KR focuses on the knowledge of the expert and it has long been recognised as a central issue in artificial intelligence (AI). Generally, it is related to the ways of encoding human knowledge in the form of symbols which can be processed by a computer to obtain intelligent behaviour. From the point of view of knowledge engineering for intelligent behaviour, the question to ask is, what domain KR formalism an agent needs to know to behave intelligently and the computational mechanisms for manipulation of its knowledge i.e. description of notions, facts, and rules of the world.

Automated planning reasons with the KR of actions from the knowledge base and arranges actions in order by predicting their outcomes to display some rational behaviour in an explicit way. A vital part of automated planning activity that requires substantial effort comprises of creating a formal domain model of the real life domain in such a language that the output domain is not application or case specific, and can be utilized by other planning systems and applications in its general form i.e. the model must have precise syntax and semantics. Based on the same idea, consistent and precise KR is important for the valid operation of domain independent planners where the plan representation language and plan generation algorithms are expected to work for a reasonably large variety of application domains.

Definition Knowledge Representation is dedicated to representing knowledge and data structures about the domain in the form that is in-line with machine representation formalism to solve complex tasks. For instance, to represent the scenario of patience card games in a form understandable by an intelligent game playing robot for winning any random arrangements of cards.

The well-modelled scenario should confirm to the real life domain area. To present the idea in a more understandable way, Tate et al in (Tate et al., 2012) consider two constants X and Y and assume X is a formal model of Y, where Y represents some real life domain. Then:

- X uses symbols in order to represent features of Y. A symbol can be anything that indicates the physical characteristic of domain Y, e.g. X uses symbol “above", to show spatial relation in the physical domain Y.

- Statements in X correspond to a true value in the real domain. For instance, if true in the real world domain, X contains “block a is above block b" statement to represent this situation.

- Operator definitions in X represent physical actions in Y. Action “put b on a" in X produces the true statement “above b on a" as the effect of the action.

Along with that, Shoeeb in (Shoeeb, 2012) presents a less formal definition of a domain model where:

A model **D** is always based on some reality **R. D** is a logical and mathematical abstraction of reality **R,** where an interpretation function **I: D←R** is the mapping of abstraction to reality.

According to the interpretation function, the object states set **S’** in **D** must be consistent with a reality domain states **S** (in **R)** by assertion **I (S’) = S**.

An important point and yet a challenge for knowledge engineering and representation is that this belief of consistency in an abstract and real domain cannot be proved formally, a reason being the real domain **R** itself is not a formal system in most of the cases.

With the increase in the complexity of domain models, its correctness is an essential factor in the overall quality of the planning process. There is an increase in the need to measure and evaluate domain models, in particular, to evaluate KR techniques. Indeed, Bensalem et al (Bensalem et al., 2014) state that domain models present the biggest validation and verification challenge to the P&S community. Several authors (Shoeeb and McCluskey, 2011, Ferrer, 2012) have proposed varying factors that identify the quality of a domain model from a KR and KE point of view. Among them, the most notable are the following:

Accuracy: Domain D is considered accurate, logical and a mathematical abstraction of the real and formal domain R if the knowledge that D contains is in-line with the requirements specifications of R, which are identified and specified by the real domain experts. Checking accuracy is an informal process if the requirements are also described informally. Checking that the model is accurate involves checking the accuracy of assertions in every state by comparison (McCluskey et al., 2016). For example, features represented in the model D are considered true if the preconditions and the effect of actions in the environment represent the real effects of actions faithfully.

More research is needed to discover sound ways of measuring differences when comparing two domain models for accuracy as well as completeness.

Adequacy: Domain D is adequate if it represents sufficient and equivalent details compared to the existing real domain R and does not contain redundancy. For the case when no domain exists in the real world, the domain must generate all the valid plans keeping in view the expected outcome of the knowledge engineering.

Completeness: Over the past several years, the problem of achieving the completeness for an abstract interpretation of a domain is recognised as difficult and has gained much attention in the literature. Some specific abstract analyses and interpretations problems have been successfully solved while the more general problem of making a generic abstract interpretation complete in the best possible way is still an open question.
McCluskey et al in (McCluskey et al., 2016) states that given a specific problem instance, a domain model, and **I** (their interpretation mapping to the requirements specification), then the domain model is complete if:

(i) For any solution plan S for problem P, I(S) is an acceptable solution for I(P) in the requirements; and

(ii) The converse is also true, i.e. for any acceptable solution S' to problem I(P) there exists a solution plan S derivable from the domain model for problem P such and I(S) = S'.

In addition to being in-line with the requirement and analysis phase outcome, we also consider the Domain D to be complete if the static and dynamic part of the domain is both adequate and accurate at the same time. The adequate model can lack accuracy by representing the task including complete details but some details are not presented accurately. Similarly, an accurate but inadequate model represents the features in-line with requirements but some requirements are totally missing. In addition to being adequate and accurate, a complete domain model must be sufficient and in-line with the requirement and analysis phase outcome.

In the AI Planning literature, the validation of a domain model often relies on a test of whether it will lead to acceptable behaviour in a P&S system (Shah et al., 2013b) i.e. if a satisfactory plan can be output. It requires a great deal of effort to correctly model the domains that can be used in the real life applications. The best example here is the complex domain models used by NASA, which are encoded by a team of highly specialised knowledge engineering experts to perform AI planning operations.

Knowledge Engineering for Planning and Scheduling (KEPS)

Humans are gifted with learning skills that help us plan when problems are selected from a stable population by the use of natural abilities of perception and self-awareness. Intelligent planning agents must have a model of the dynamics of the domain in which they act. Models can be encoded by human experts or, as required by autonomous systems, automatically acquired from observation.

KEPS is a support process of AI Planning and is solely dedicated to all the offline knowledge engineering aspects of automated planning and any online processes, which require changes in the domain model. KEPS was formally introduced in PLANET Roadmap 2003 (Biundo et al., 2003b). It is designed with special focus on improving domain-independent planning and it is taken as a special case of knowledge engineering. KEPS is a broad area, starting from acquisition to formalism, design, validation to the maintenance of domain models. The main activity involved in KEPS is the process of engineering domain models by integration of model generation techniques with AI planning algorithms. It has to do with off-line, knowledge-based properties of planning, based on the application under consideration. KEPS is also associated with the subject of describing the conceptual knowledge and producing domain models for Qualitative Reasoning in the general modelling and Simulation area (Bredeweg et al., 2008).

It is defined as the process that deals with acquiring the knowledge about the interaction among different objects involved in planning and the knowledge about actions that can be applied on those objects, where a major product is the domain model.
PLANET Roadmap (Biundo et al., 2003b), a technological Roadmap on AI Planning and Scheduling, organised by the European Network of Excellence in AI Planning, defines KEPS as follows:

“The processes to acquire, validate, verify and maintain the planning domain models and the selection and optimisation of appropriate planning engines to support knowledge engineering process.”

The research aims at improving methods to capture, learn and enhance knowledge to make planning engines more effective and independent. The importance of knowledge engineering methods can be clearly seen in the performance difference between domain-independent planners and domain-dependent planners where domain-independent planners have the broad focus in terms of exploiting different domains compared to domain-dependent or domain-configurable planners.

KEPS and KBS

KEPS inherits from Knowledge Based Systems. KEPS utilises simple knowledge bases with complex temporal reasoning where KBS traditionally uses relatively complex knowledge bases with simple reasoning, because, the vital use of KEPS output is to be a part of systems that construct plans while KBS solves diagnostics or classification problems.

Domain Modelling Process

The required efforts to formulate a domain model for AI planning engines has long been accepted as a challenge for the AI planning research community. The domain knowledge produced by KEPS is mainly knowledge about an action's preconditions and constraints. It also includes the knowledge about the objects and relationships between objects involved in an application domain. For the domain design process, among one of the challenges of KE is that there is no standard and dedicated domain design process for knowledge engineers to follow. In 2003 PLANET Roadmap (Biundo et al., 2003b), the authors generally describe the steps for domain model development that covers the complete definition of KEPS. Following are the general steps in the domain modelling process:

- Early Analysis
- Knowledge Acquisition
- Domain Design
- Verification/Verification Check
- Maintenance

- **Early Analysis** Firstly, the knowledge engineer must perform some early analysis to find the category of planning problems that need solving by the domain model. This includes specifying the varying environmental characteristics e.g. is the environment deterministic. Durative? Does it need to care about resources allocated? Is the environment observable? Types and optimisation measures of required plans etc. This also includes the identification of common and reusable capabilities within the application domain. Technical literature, existing applications, customer survey and advice, and the current/future requirements could be good sources of knowledge for the analysis phase. This information is also crucial for the selection of a planning algorithm. The output of this stage appears as the reuse standards, environmental characteristics, behavioural model and domain language.
- **Knowledge Acquisition** After early analysis, as step two, acquisition of domain knowledge is performed. A prolonged process includes analysis of the application area by knowledge engineers. It involves stakeholders (users, operators, beneficiaries, managers etc.) to conduct various KE processes including interviewing the experts of the area, to collect the required knowledge about the domain model and possibly

domain heuristics. The extracted theoretical knowledge needs to be converted to practical facts in either dynamic or static category of domain knowledge. The knowledge acquisition process can be classified into two categories:

- Handcrafted Methods
- Autonomic Method

The focus of the research in this thesis is based on automatically acquiring knowledge for domain models, which is discussed in more details in upcoming chapters.

- **Domain Design** After acquiring knowledge about the domain, the next step is the design or Modelling of the declarative description of the domain under study. It includes reasoning with and managing the knowledge of step two. The extracted theoretical knowledge needs to be converted to practical facts in either dynamic or static category of domain knowledge in the form of relevant objects and their types, predicates to express the relationships among objects and the constraints of the domain area.

  There is no standard design process to follow. (Vaquero et al., 2011) provides a review of tools and methods that address the challenges encountered in each phase of a design process and presents a semi-standard procedure for the domain design process. Their work covers all the steps of the design cycles and focuses on tools that can be used by human experts for encoding domain models.

  KEWI interface (Wickler et al., 2014) is another tool to encode the requirements into domain knowledge. It first synthesises the conceptual model where it encodes the requirements in a more application-oriented language AIS-DDL. It then maps into planners input language PDDL.

  There is another tool for domain analysis and design called TIM (Type Inference Module) (Fox and Long, 2011) which deduces object types and its invariants from domain definition and initial states. It constructs finite state machines to analyse and illustrate the transition pattern of actions involved in domain definition. This process can be implicit as it refers to unconscious self-reference or self-awareness of the knowledge engineer.
- **Verification / Validation Check** At the next step, to check logical consistency, identify bugs and to prove the accuracy of the domain model, its quality must be checked not only in terms of syntactic and semantics' accuracy but it also includes checking completeness of the generated domain through static and dynamic testing (Vaquero et al., 2011). As the sources of knowledge elicitation and model development are not mathematical procedures, therefore a domain model cannot be measured on a correctness scale.

  To verify the developed model, answer the questions 'Have we made it right?' Identify the required properties and demonstrate that the implementation will fulfil the properties. This also requires having the accurate and accepted description of implicit

parts of knowledge in the early analysis phase. Verification involves static tests with the developed model.

To check for validity, answer the question 'Have we made the right product?' to demonstrate that the built domain model fulfils the needs of the stakeholders. Validation involves dynamic tests with the developed model. Dynamic tests check the overall flow of the domain and compatibility of pre and post conditions of actions. Validation of the domain model supports validation of planning systems in the later stage. For the post design verification, VAL (Howey et al., 2004) is a plan validation tool that validates if the plan has reached the goals using the developed domain model.

- **Maintenance** Finally, after the developed knowledge base is refined to bring it to operational form, it must maintain it with every new originated/discovered feature in the domain area.

Knowledge Engineering for Planning is in its early stages so it has attracted much research. There have been many events of the subject in recent years including workshops that assist AI-planning, for instance, ICAPS and its elements competition series of ICKEPS (ICKEPS'05, ICKEPS'07, ICKEPS'09, ICKEPS'12 and ICKEPS'14) and workshops on KEPS (ICAPS-KEPS, 2014) and Verification and Validation of Planning Systems (ICAPS-VVPS, 2011). It has resulted in the creation of several powerful systems and techniques to support the design process of domain models and encouraged the use of planning engines for real world functions.

There is already a lot of published research aimed at automatically learning and maintaining a useable domain model for high-level reasoning, specifically to support AI planning. Building domain models for AI planning presents steep challenges for domain creators, no matter how efficient or powerful Planning and Scheduling engines are, they are only as good as the application knowledge that they use. If the Planning domain model is flawed, the resulting application will be flawed (Bartak and McCluskey, 2006).

Challenges for KEPS

After having discussed the domain Modelling process, and the types of tools that tend to support the KE process, we now present challenges that KE for AI planning still faces. There are multiple challenges for KEPS.

Although AI planning technology is mature, the research in KEPS mostly relies on experimental domains for benchmarking the KE experiments. This is also because KE has no standard evaluation methods yet and just like the requirements specification, it cannot be objectively assessed and proved correct.

The evaluation of KE tools and methodologies appear to be harder than other tools and methods, for instance, planning algorithms and planners. This problem of

methods/tools evaluation in KE inclines the research community to avoid paying much attention to knowledge engineering for AI planning and it becomes harder to foster the encoding of sophisticated domain models without evaluation. Moreover, knowledge engineers of domain models use planners to design, develop and debug the domain model where planners are not actually for this purpose. There is a need to build up more methods to match up engineered domain models with planners (McCluskey, 2000).

Moreover, planning research has a history of focus on experimental domains and problems where KE issues are not applicable. More focus has been on theoretical aspects of techniques rather than the domain engineering aspect. As a result, techniques do not serve for real-world planning problems.

Idealised planning KE environment: A technical report on the European Network of Excellence in AI Planning, discussed the issues and challenges of knowledge engineering to build platforms for efficient automated planning systems (Biundo et al., 2003b, McCluskey, 2000). The report depicts the essential environment from the user perspective (figure 2.8 taken from the report on the European Network of Excellence) which is an integration of all the tools of KE in one platform. Inspired by the PLANFORM project (McCluskey et al., 2002a) (by the UK Engineering and Physical Sciences Research Council), its aim is to enhance the research and development for the efficient construction of planner domain models, to abstract the descriptions of planning algorithms and the combination of both to produce plans for automated planning.

The platform, as shown in figure 2.8 a high-level architecture, uses some shared graphical metaphors. Hence, inside what they called “idealised planning KE environment”, a user can perform all the steps of domain modelling and design.

To avoid hand-coding knowledge, the platform is supposed to gather knowledge from many different sources e.g. by interviewing the users, engineers, managers and stakeholders of the system. These not only can add knowledge but also can apply model measurements, review the model, verify the properties and can test the model using simple planners. Already existing models and data can convert to the internal representation of the new domain model by using knowledge acquisition tools and methods. Even the previous SRS (software requirement specification) document and feasibility studies of the related domain may prove beneficial as the input knowledge. In the architecture, existing plans are a kind of plans that the planner is expected to produce eventually. These could be used to acquire knowledge like heuristics for the planning process and to validate the final plans produced by the planner. Hence, all the mentioned sources of information will influence the development of the planner. The interfacing tools to the process can be taken as interfacing all the parts of the system where the planning application is at the heart of the environment.

In PLANFORM project, Object-Centred Language (OCL) is used to represent domain models, to support validation and checking of tools and to support translation to other representation languages, for instance, PDDL. While in order to perform interactive construction and validation of a model, the GIPO toolset (Simpson et al., 2007) is exploited. GIPO is an experimental graphical user interface and tool for building planning domain models.

All three components i.e. basic domain knowledge, heuristics and planning algorithm are kept separate to avoid maintenance issues later. The only point where this information can combine is the compilation tool, which later produces the planning application. The compilation tool produces the planning application after deciding the appropriate planning technology based on domain features. Dynamic testing is applied after the planning application is produced.

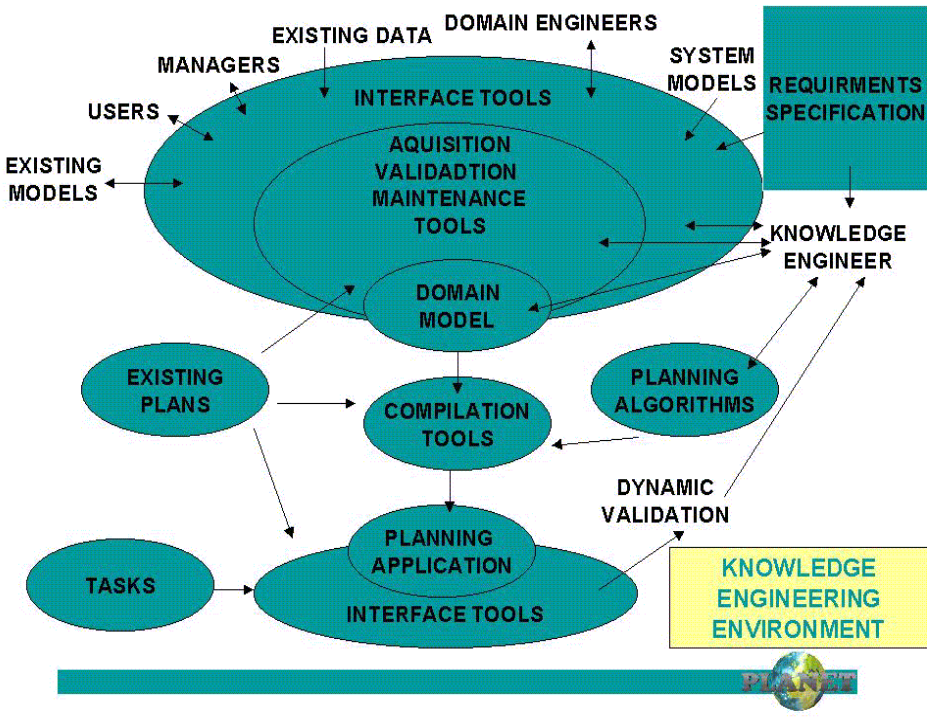


Figure 2.8: An Idealised Planning KE Environment (Biundo, Aylett et al. 2003)

III - Graph Theory

The study of graphs is known as graph theory and was first systematically investigated by D. König in the 1930s. C. Hoede and F.N. Stokman are among the few mathematicians who exploited the idea of Knowledge graph theory in the late 1980s (Stokman and de Vries, 1988). Initially, graphs - a discrete maths concept – were in use as a representation formalism of sociological and medical texts. The large sized graphs were used to represent the knowledge in such texts in a way that the graph acted as an expert system, to

automatically explore the causes of disease, to support decisions and to automatically calculate the consequences. This is discussed in more detail in (de Vries, 1989) with the focus on causal relationships.

With the increase in demand for reliable AI planning systems and domain models that can be applicable in real-life problems, graphs are appearing as a formalism with great potential for model generation, visualisation, checking and simulation in knowledge engineering, knowledge representation, and domain analysis for automated planning.

Definition: A graph is an abstract representation of data in the form of a set of vertices or nodes that are connected either by directed or undirected edges and is use to model multiple types of relationships between objects.

Graph: $G = (V, E, \mu, v)$
V: finite set of nodes.
$E \subseteq V \times V$ denotes a set of edges.
$\mu: V \rightarrow L_V$ denotes a node labelling function.
$v: E \rightarrow L_E$ denotes an edge labelling function.

The nodes of a graph represent constant symbols and edges between nodes represent static facts or properties/relationship between nodes. Nodes can be anything or any object and edges are the ordered pairs of nodes. The number of nodes and arcs in a graph can be infinite. Chein et al in (Chein et al., 2013) describe the benefits of using graphs for representing and engineering knowledge as the following:

- Firstly, being considered as among the simplest mathematical objects, graphs exploit simple set theory notions (e.g. nodes or elements, edges or relations etc.) which can be visualised due to its graphical representations.
- Secondly, a number of algorithms and methods have been proposed in the past for graph processing, and due to its increasing usage in operational research, graphs can be used as effective computational objects.
- Finally, graphs are used to represent logical semantics: the graph-based mechanisms are sound and complete in terms of deduction in the assigned logic.

Graphs are a popular way to formulate representations and provide a visual way for easily explaining to the user about varying type and complexity problems, including mathematical and computational problems and problems of reasoning about action. Due to the visual representations of complex interrelationships between entities graphs enable the human brain to hold facts and records for longer as compared to the logical

representation of the same data. For the same reason, *semantic networks* (cognitive models), have been helpful in knowledge representation since the early days of AI. The expression of the semantic network includes an entire family of graph-based visual illustrations (Chein et al., 2013). Moreover, the captured sequence of nodes in the graph may also support domain designing and modelling by depicting the relations or properties that are considered implicit in an engineering domain model and are usually ignored. It is a common way to construct representations for many problems, including the problems of reasoning about actions.

Graphs can also be used as the design of analytical formalisms in order to carry out dependency network analysis. These can be used to guide the effects of dependencies between entities and the path through which the dependencies affect many parts and paths in a graph. For example, to study the ripple effects of failure in one entity on other dependent entities across the system. Based on this capability of the graphs, Garvey et al in (Garvey and Pinto, 2009) presented a methodology called *Functional Dependency Network Analysis (FDNA)* for enterprise systems that anticipate the ripple effects early in the design and enables engineers to minimise dependency risks which can have negative effects on enterprise functioning. By exploiting graph analysis, an engineering system can be represented as a directed graph (capability portfolios) whose entities are nodes that characterise the systems and depict direction, strength, and criticality of dependency relationships.

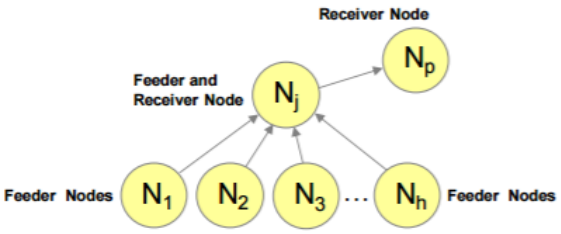


Figure 2.9: An FDNA Graph is a Topology of Receiver-Feeder Nodes.

An FDNA graph can be viewed as a topology of receiver-feeder node relationships (figure 2.9). A receiver node is one whose operability level relies, to some degree, on the operability level of at least one feeder node.

Graph embedding is also a familiar way for the representation of many problems in different fields ranging from data mining problems to reasoning and learning problems in AI planning. Examples include the discovery of diagnostic symptoms from patient history, searching records to support decisions and to calculate the consequences automatically. In addition, the usage of graph theory for mining data has its roots in Concept formation which is a sub-discipline of AI and is concerned with mining data from graph-structures.

Graphs are also commonly used as fundamental models in Knowledge Visualization, Knowledge Engineering, Software Engineering, Operating Systems, Graph Database,

Network Systems, Social Media Analysis and many more fields such as Bioinformatics, Image Processing, and Website Designing. The work in upcoming chapters of this thesis shows that usage of graph theory is effective, behaviourally diverse, and boosts and readily facilitates the easy and automated availability of AI planning domain knowledge.

Graphs in Knowledge Engineering for Automated Planning

Recently, a novel field of learning from a topological view of the data structure has arisen. Graphs are among few commonly used generic topological structures in mathematics as it is a very powerful way of capturing and representing logical semantics in visual and syntactic form. Structural Graph Theory deals with capturing results that describe various properties of graphs, and utilizes them in the design of efficient algorithms and other applications. Concepts included in this area are *treewidth*, *graph minors* and *modular decomposition*.

In the context of knowledge engineering in automated planning, there is much research done by using functions of graph theory, for instance, **itSIMPLE** (Vaquero et al., 2007) is an open source graphical system which supports the process of knowledge engineering for designing planning domain models. It exploits an object-oriented approach and uses Unified Modelling Language (UML) (Booch, 2005) to describe, visualise, modify, construct and document application models. The function of graphs comes in when it shows the domain model to the user in the form of Petri Net graph which is automatically generated from UML, and as a consequence, it can demonstrate and analyse the dynamic aspects of the domain model. Figure 2.10 represents the Planning domain design processes in itSIMPLE$_{2.0}$ (Vaquero et al., 2007).

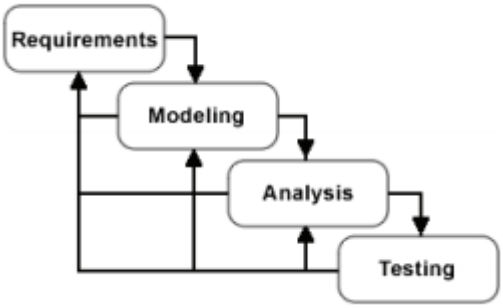


Figure 2.10: Planning domain design processes in itSIMPLE2.0

GIPO (Graphical Interface for Planning with Objects) (Simpson et al., 2007) is an experimental environment for simplifying the task of domain model formulation and validation. It exploits the functions of graphs for the development process by using graphical editors. It also contains animators to inspect the produced plans graphically. It

expresses the structure of domain in its own object-centred language OCL and OCL_h. The built-in graphical editor follows the OCL to demonstrate the domain structure graphically which effectively helps users visualise the developing domain features.

VIZ (Vodrázka and Chrpa, 2010) is another system that provides a complete graphical interface through visual representation and effectively supports knowledge engineers and even non-expert users (student level) in the design phase of planning domain models. It uses First Order Logic (FOL) and concepts of object-oriented programming (OOP) and provides a GUI for describing the planning domain and problem. The semantics of graphs are exploited to represent three levels of abstractions:

1. Classes and Predicates declaration
2. Using variables and predicates to define domain operators
3. Using predicates and objects to define problems

Rectangular nodes show types (classes in OOP language) and elliptical nodes indicate predicates. By connecting a rectangular node with an elliptical node by an undirected edge shows the argument type (rectangular node) of a predicate (elliptical node). The basic graph structure based on Blocks world domain is shown in figure 2.11 (Vodrázka and Chrpa, 2010).

In a graph for representing domain operators, rectangular nodes represent variables while three disjoint sets of elliptical nodes represent precondition, positive effects, and negative effects. Each type of elliptical node is connected to the required variable-node according to the logical occurrence (precondition/effect$^-$/effect$^+$) of predicates in the operator. It can export the developed graphs in PDDL form too.

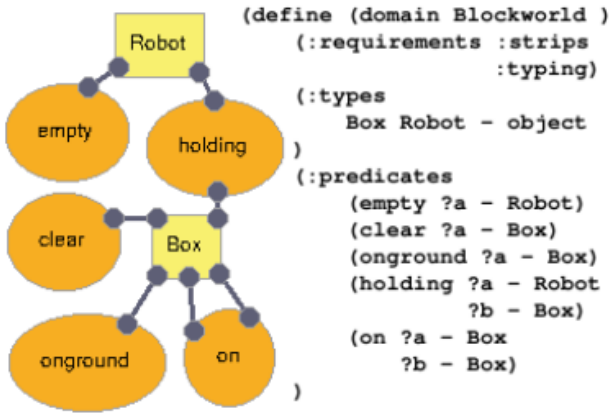


Figure 2.11: Declaration of language (Vodrázka and Chrpa 2010)

Francisco et al in (Palao et al., 2011) present a KE framework for HTN planning applications. It introduces a new graphical and intuitive notation for easily representing HTN domains, which intend to ease the understanding of domains for domain experts and engineers. Similarly, **Bernardi et al** in (Bernardi et al., 2013) have developed a plugin

that provides a graphical interface which visualises and analyzes the domain. The same graphical environment is used to define, add and edit new states of the domain.

JABBAH (González-Ferrer et al., 2009) is an integrated domain dependent tool that provides support for converting graphical notation of BPMN (Business Process Management Notation) to HTN-PDDL domain model.

A method presented in (Wickler, 2013) analyses the domain models for the purpose of domain validation and verification. Wickler exploits the functions of the graphs by identifying static relations in the form of a graphical structure from the operator schema of a domain model. The knowledge engineer explicitly specifies the static relation to be used in the drawing of a static graph.

The static knowledge of a domain model is usually fixed and cannot be changed by the action effects. Therefore, the graph formed by the validation system of Wickler can be analysed independently from the state information of the rest of the domain states. Once a static graph is identified (with variables acting as nodes and the relationships between the variables acting as links between the nodes), the method again exploits graph behaviour to identify and validate the node-fixed types i.e. the objects that cannot move between nodes in the static graph and have some fixed relationship with nodes in the static graph. Node-fixed types identify the functional properties of the defining static relation. Moreover, there are actions in the domain model (especially transportation domains) that shift objects from one node of the graph to the neighbour node. The method of Wickler also determines such shift operators with respect to node-fixed types. In short, the technique effectively supports knowledge engineers and builds on domain analysis based on the features described. Chapter 7 (section 7.1) thoroughly explains the method and explains the combined use of ASCoL with the system of Wickler in order to analyse planning domains using static analysis.

Bordes et al in (Bordes et al., 2011) presented a learning method based on a neural network architecture designed to embed knowledge from different structured Knowledge Bases (KB) into more flexible vector space to make it useful for other machine learning systems. They present a model that learns to represent elements of any KB into a low-dimensional vector space. It uses functions of graphs by considering Knowledge Bases as a graph model by defining it as a set of nodes and edges. They used WordNet and Freebase KBs to demonstrate their work.

Another knowledge discovery system based on graphs is **SUBDUE** system (Gonzalez et al., 2000). It works by finding the relational and structural patterns in data that represent entities (vertices) and relationships (edges) in the graph. It makes use of minimum description length (MDL) principle to discover patterns in the input graph. The system can perform supervised and unsupervised learning, graph grammar learning and

clustering. Authors have applied the system in a variety of areas that include web structure mining, aviation, bioinformatics, social network analysis, geology, and counter-terrorism.

Consequently, Graphs are a basic mathematical model for entities and relations and graph theory enables a visual representation of complex interrelationships between entities where the directed graph not only represents the network dependency but also depict the strength, direction and order of the system. The systems discussed above are the examples of how the graphs can be used not only in the design process of the domain models but also for development, verification, analysis and demonstration/representation of the domain models. The next section explains how graph theory relates to the thesis research and signifies the importance of graphs in identifying missing static preconditions from application knowledge.

2.2 The Scope of this Research

The aim of providing background in the previous sections of this chapter is to explain where the contributions of this research can fit in. It also shows how it links up to recent research in the area (related work is provided with each section), and to explain their relation and part in the topic of research. In broad terms, the topic of research is the 'Use of Graph Theory as a technique of Knowledge Engineering for Automated Planning Domain Models'.

The main aim of this research is to overcome the knowledge acquisition bottleneck in order to help planning agents become more autonomous, make them able to adapt and plan under unseen situations and to debug existing domain models.

The LOCM systems perform automated induction of the dynamic aspects of a planning domain model, i.e. changes in the state of the world occurring due to action execution, by considering only a set of plan traces as input. Section 3.2 covers LOCM in more detail.

The main assumption the LOCM relies on is that all objects in the domain go through transitions. This assumption is too strong for some scenarios especially when the domain contains static aspects too as they are not reflected in plan traces. Based on this, a drawback of the LOCM process is that it can only induce a partial domain model which represents the dynamic aspects of objects and not the static aspects. This is problematic since most domains require static predicates to both restrict the number of possible actions and correctly encode real-world constraints. This is the case in many well-known benchmark domains e.g. the static knowledge in the *Miconic* domain represents the level of floors, and in the *Freecell* domain, it represents the fixed stacking relationships between specific cards. In the *Driverlog*, the presence of static predicates represents the connections of roads.

In this thesis, our premise is that the static attributes considered for the purpose of research do not change, are current and correct. The learnt static facts also include exceptions when the training data is taken from the same exceptional event that happened in the past. To explain it better, there are certain domains where knowledge engineer has to update the static facts based on the exceptional scenarios. For instance, in the real world, when the static road system exceptionally faces land-sliding situation, the traffic should take the alternate route. Similarly, when a multi-storeyed building faces a fire situation, the lifts should not go higher than certain floor or should not work at all. In such situations, static facts do not remain current and need to be updated manually (by a domain expert) in order to handle the exception.

The LOCM is not able to induce static knowledge automatically, the domain engineer has to declare the following missing static relations information manually (written in Prolog) in the input training data to make it a mandatory part of the output domain model:

Static (connected (L1, L2), Drive (L1, L2)).

This appended static predicate needs to be true in the operators before the correct plans can be generated by using the learnt operator schema. The fact in the first argument of Static is added as a precondition of the action in the second operator argument of Static, where shared variable names provide the binding between the action and its precondition. The next subsection explains the learning problem that is addressed by the research, the importance of addressing this issue and the complete input/output structure of the developed methodology.

Objective and Learning Problem

This subsection first provides insight into the motivation for the research in this thesis and then explains the main aim and objective along with learning problem that this research addresses.

We present **the ASCoL algorithm**, a supervised learning method that exploits graph analysis for automatically identifying static facts from structured training data, in order to enhance the domain acquisition process. According to AI planning standard terminology, instantiated domain predicates are called facts. These are called ‘static facts’ if they are among the initial states of a problem and no operator can delete those facts. Facts always contain constant objects in pairwise linking and that pair along with other static objects appear as nodes in a static graph. All the resulting edges in static graphs are the static preconditions in planning domains.

Our **objective** is to learn a complete set of static preconditions. A static support for model generation and analysis is most often the visual support. ASCoL generates a directed graph representation of operator arguments' relations observed in the plan traces. ASCoL also embeds the learnt static knowledge into the correct operators in the LOCM-learnt output to enrich the domain with this required but missing knowledge. We formally define the correctness of the learnt static knowledge by comparison with benchmark domains.

The main difference to earlier work is that the technique performs pattern discovery in the plan traces given by the user as input, with no additional knowledge about initial, goal or intermediate states. There are other mechanisms for learning operator schema and static knowledge and those are covered in the next section of related work.

The input to the static constraints learning algorithm ASCoL is specified as a tuple (P, T), where

$P = P_1, P_2, \ldots, P_n$

P is a set of plan traces. In ASCoL's input, plan or action traces are represented in a text-based format (the same as supported by LOCM) such that each plan contains action sequence of N actions on numerous objects i.e. each $P_i \in P$ has the form:

$P_i (A_1, A_2, \ldots, A_N)$ for $i = 1, \ldots, N$

Where *A* is the action name. Each action A has a format which is made up of an identifier (the name of the action), and the names of objects that it affects, in order of occurrence, which all have the form:

$A_i (O_{i1}, \ldots, O_{ij})$ for $i = 1, \ldots, N$

Where *O* is the action's object name. Each action *A* in the plan is stated as a name and a list of arguments. In each action A_i, there are *j* arguments where each argument is object *O* of some type T.

T is a set of types of action arguments in P which ASCoL parses from the LOCM output such that:

$n(T) \leq n(O)$ types $T = t1, t2, \ldots tn$.

ASCoL does not require dynamic knowledge of the domain generated by the LOCM. It uses fully ordered plans so as to identify the maximum links between static nodes of the graph from the linear ordering of actions as action succession is important to determine the order of the occurrence and relations between objects. In contrast, a partial plan does not specify an exact order for the actions when the order does not matter. Figure 2.12 shows an example of a fully ordered plan trace for Blocks Domain.

```
sequence_task (30,
[unstack (b10, b1), stack (b10, b7),
unstack (b1, b6), stack (b1, b10),
unstack (b9, b5), stack (b9, b6),
unstack (b1, b10), stack (b1, b9),
unstack (b10, b7), stack (b10, b1),
unstack (b7, b2), stack (b7, b10),
unstack (b2, b4), stack (b2, b5),
unstack (b7, b10), stack (b7, b2),
unstack (b10, b1), stack (b10, b7),
unstack (b4, b3), stack (b4, b1),
unstack (b3, b8), stack (b3, b10),
unstack (b4, b1), stack (b4, b8)], _, _).
```

Figure 2.12: totally ordered plan from Blocks Domain

Precisely, each plan trace is a sequence of actions in the order of occurrence to satisfy some goal, where each action in the sequence contains the name of the action and objects that are affected by that action execution. Input plan traces do not include any initial, goal or intermediate states or constraints. Static constraints are to be learnt by the system.

ASCoL can also work with just the sequence of matching action instances in the sets of plan traces, therefore it does not necessarily require a formal sequences of plan traces in order to produce results. The only requirement on plans is that they have been generated by problems sharing the same objects; this follows the hypothesis of LOCM, in which different plan traces are generated by observing the behaviour of an agent in its environment.

The output of a learning problem is a constraint repository *R* in PDDL representation that stores all admissible constraints on the arguments of each action *A* in plan traces *P*. We assume that input plan traces are noise free while the input domain file at least

contains type information for all those operators that the algorithm aims to enhance. Figure 2.13 shows the general structure of ASCoL in terms of its inputs and outputs.

ASCoL works as a separate unit from LOCM in that LOCM first produces a domain model using a set of plan traces as input. The same LOCM generated domain model, along with the same set of input plan traces, are provided to ASCoL to first anticipate the required set of constraints, analyse plan traces and then learn constraints. Finally, it embeds these constraints into the correct operators in the LOCM-learnt output to enrich the domain.

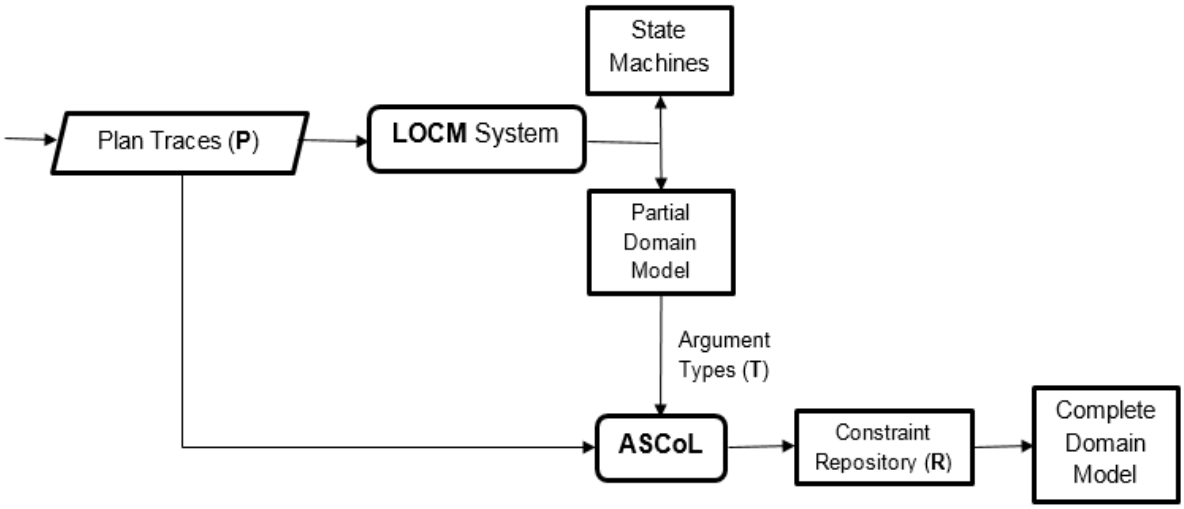


Figure 2.13: Input Output Structure of ASCoL

We aim to capture two major kinds of constraints: domain specific and domain independent constraints. By domain specific, we mean the knowledge that is strictly associated with a domain and is not found as a general example, e.g. the fixed relationships between specific cards in Freecell. Domain independent constraints describe the static knowledge that is generally associated with almost all domains in one way or another; non-equality constraints and link constraints for example.

2.3 Learning from Plan Traces

This section explains the rationale of using plan traces for learning, the limitations that learning systems face by using only plan traces as a source of learning and the types of plan traces used for learning based on their production method. Potential plan traces can be gathered from multiple sources and applications, for example, the sequence of workflow in some process execution, logs of commands for installing a piece of software or the moves or steps captured from game playing etc.

Rationale of using Training Plans as Input

Our rationale behind choosing training data as input for our approach is that domain engineers may not have access to thorough and adequate information about the domain's activities, i.e. Domain operators and its states.

Firstly, there may not be any experts in the area available to design the domain. If the experts are available and design the domain, yet there is a possibility that they may ignore the tacit/implicit knowledge in their mind about the domain unconsciously while they act on that knowledge out of their daily experience (but mistakenly ignoring it considering it as 'obvious' knowledge). Also, based on personal experience, experts' knowledge is most of the times situational (Klein, 1998).

Secondly, the domain may not yet exist in the real world or was not designed before. For example, designers of an intelligent shopping machine will want to make sure that it will do shopping effectively and optimally without violating budgeting constraints. Grant in (Grant, 2010) says that there can be many practical hitches in acquiring flawless information about a real-world domain and the factors can be time, space, resource, cultural, or other limitations and barriers.

Thirdly, experts may be able to describe true positives and false negatives among constraints of the domain while providing less insight into the true negatives and false positives, as experts more likely operate their domains well from invalid states and work inside safe boundary limits. This avoids coming across many possible invariants.

Owing to all the above-mentioned and more such issues, experts need automated assistance in modelling constraints fully.

Limitations faced by using Plan Traces as Input

In addition to the problems we faced by using plan traces as the only source of knowledge, Grant in (Grant, 2010) also discusses the limitations of using plan traces as the source of input information. Following are some limitations ASCoL faces, as the only input source to verify constraints are sequences of plans:

- For plan traces to exist there must be a comprehensive domain that should exist at least in experimental domain category, to evaluate the results of the system.
- Traces only provide examples of valid operating states, so these cannot be used to change the invalid constraint to a valid constraint.
- Traces may be inadequate to learn states and constraints fully.
- Multiple instances of plans may contain different object names if there is no consistency maintained in the source for synthesizing plan traces.
- If obtained from sensors, sometimes noise inevitably get introduced into plan traces when some sensors are occasionally damaged, with unintentional mistakes

in the recording of the action sequence, or may be due to the presence of other agents in the same environment.

We ran investigative experiments to determine which method of plan trace generation is better. After experimenting with both goal-oriented and randomly generated plans, we learned that both the methods have their own drawbacks and advantages.

Types of Input Training Plans

ASCoL has been evaluated on domain models generated by LOCM for international planning competition domains, using plans generated through goal-oriented (GO) solutions and random walks (RW).

Using RW, plan traces usually do not reach the goal required by the planning instance, but provide richer information in terms of the number of transitions for different types of static facts when compared to the goal-oriented plan sequences. GO solutions are generally expensive in that it requires a tool or a planner to generate a large enough number of correct plans, but they can also provide useful heuristic information. GO Plans are more purposeful while RW plans provide a good spread of ground actions for the knowledge acquisition systems.

For each domain, we used available random generators to create the training problems, which are subsequently solved in order to obtain the required plan traces. There is a variation in the number of objects considered in problems for generating plan traces for empirical experimentation. The same set of planning instances are used for generating GO plan traces. The required input type and amount vary widely depending upon the types of static facts exploited by a domain. Characteristic difference exists between goal-directed input plan traces and randomly generated ones in terms of learning.

Complete details and analysis of the number of plan traces used and about the use of both types of input – (required by LOCM and ASCoL for reaching convergence corresponding to each domain considered), are provided in the evaluation chapter.

As discussed earlier, using plan traces as a source of information has a number of limitations. In order to overcome some of the known issues, our experimental analysis considered both sets of traces --i.e., goal-oriented and random walks. This has allowed some light to be shed on the impact of differently generated plan traces on the learning abilities of LOCM and ASCoL, and to provide some guidelines for learning static facts' relations in commonly used domain model structures.

2.4 Illustration of the Controlled Search Problem

The search space graphs resulting from planning problems are so complex that explicitly specifying them is not feasible. Therefore, different techniques have been applied to address this problem. By exploiting the search control knowledge/constraints (that are not explicitly encoded in the initial PDDL formulation) to assist the solution search and improve the performance of the planner is one of the approaches to solving large search space planning problems quickly

For AI planning, the problem search space can be logically considered as a tree structure, where nodes of the tree represent states and the edges connecting different nodes represent actions that cause transitions from one state to another (figure 2.14).

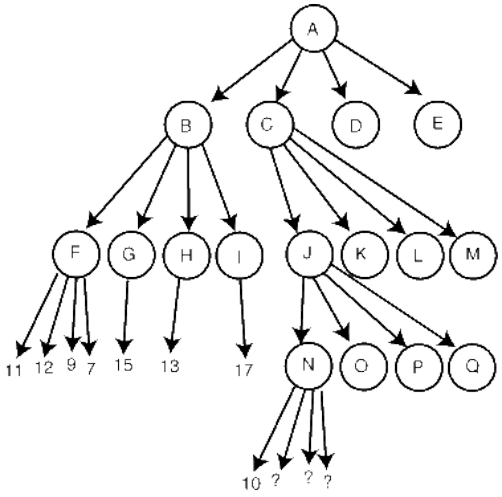


Figure 2.14: Planning as a Tree Search

A tree decreases the complexity of the problem at some cost. This induced cost is due to the repetition of some states (nodes) that were already linked several times in the form of the graph structure. In a tree, any two connected states use only one edge. Figure 2.15 illustrates this visually.

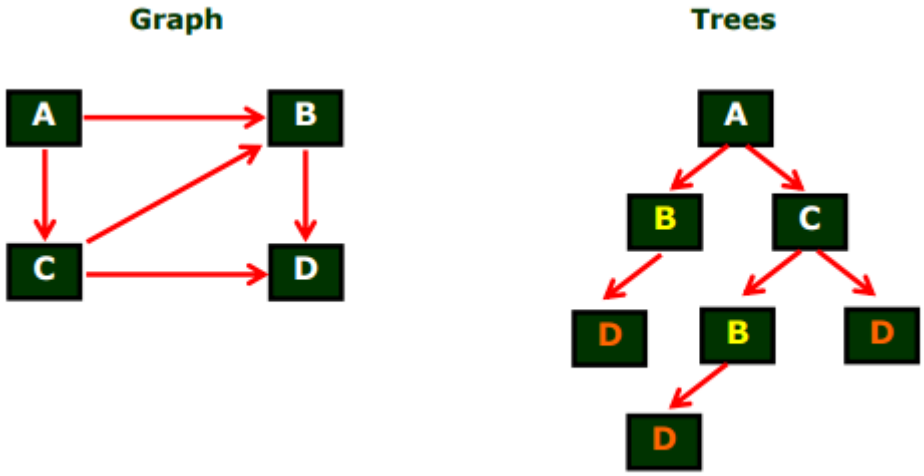


Figure 2.15: Graph (non-hierarchical) converted to Tree (hierarchical)

Using the tree graphs involves searching through states and looking for the next desired state, moving from the initial to final states that achieve the goals and proposing a plan as a solution to a problem. Searching a path for an optimal plan is even more complicated as it requires searching all the branches to find the shortest path for achieving the same goal. There are many different routinely applied algorithms in planning. For instance, breadth-first search is one of the simple exhaustive methods of searching; however, a planner has to search many states. The number of states increases exponentially with the increase in complexity of the problem. Without knowledge about directing the planning algorithm or which state or action to choose for pruning the search space next, the size of the planning search space gets intractable.

A constraint is an entity that restricts or limits the values of variables in domain modelling (Nareyek et al., 2005). It is a logical relation between variables and explicitly represents interdependencies between state variables to restrict certain paths in the search tree for the solver which otherwise would spend much effort to infer those interdependencies and even sometimes it becomes infeasible for solvers to solve large problems without constraints. Planning graph construction by Graphplan (Blum and Furst, 1997) was the first method that produced invariants for the efficient pruning of search space.

Controlled search in domains that have static aspects in them, require static rules/constraints to control or limit the search time while exploring the state space, by checking which branches require searching. Research efforts are made to extract such constraints and to exploit them for guiding and controlling search. The blue edges in Figure 2.16 show an example of this control search via using constraints in preconditions of actions.

Imagine the state space without the mentioning of Static rules for the domain, the search algorithm has to look through an extensive number of nodes to reach the goals, which may not be possible for complex problems due to time and space limits. With controlled search, the number of states to be examined using exhaustive search method can be lowered significantly, thus saving the planning time. Ideally, this should occur without interfering with the planner's ability to find an optimal plan.

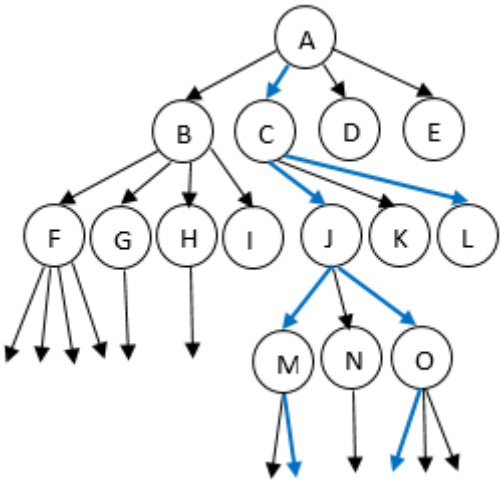


Figure 2.16: Control Searched Planning using Constraints in Preconditions

Example: TPP Domain

TPP domain is a generalisation of the Travelling Salesman Problem and we have used a propositional version of the original TPP domain. It contains a set of Markets, Depots and Goods. Trucks are available for transportation of goods from Markets to Depots. Each Market sells a limited quantity of each type of goods. Regardless of which Market is used to purchase the Goods, all Goods have the same price. A quantity of goods is represented as different discrete Levels. Trucks Load and Unload the Goods from Markets to Depots.

In the propositional TPP, four operators contain seven static relations that indicate the next level to load, unload or buy goods along with a map of indirectly connected Markets and Depots. Complete details of the domain are provided in evaluation chapter section 6.4.

In order to describe the importance of static facts in the domain model we selected a problem instance from the domain and search for a plan with and without exploiting static facts in the benchmark TPP domain.

```
(define (problem TPP)
(:domain TPP-Propositional)
(:objects
        goods1 goods2 goods3 - goods
        truck1 - truck
        market1 - market
        depot1 - depot
        level0 level1 - level)
(:init
        (next level1 level0)
```

```
        (ready-to-load goods1 market1 level0)
        (ready-to-load goods2 market1 level0)
        (ready-to-load goods3 market1 level0)
        (stored goods1 level0)
        (stored goods2 level0)
        (stored goods3 level0)
        (loaded goods1 truck1 level0)
        (loaded goods2 truck1 level0)
        (loaded goods3 truck1 level0)
        (connected depot1 market1)
        (connected market1 depot1)
        (on-sale goods1 market1 level1)
        (on-sale goods2 market1 level1)
        (on-sale goods3 market1 level1)
        (at truck1 depot1))
(:goal (and
        (stored goods1 level1)
        (stored goods2 level1)
        (stored goods3 level1)))
```

There are three types of goods that need to be transported and stored using a truck from a market to a depot. Following are the results produced by two different planners: FF and LPG-generated plans with static facts (right column) and without static facts (left column) in the benchmark TPP domain

Metric-FF

Without Static Facts	**With Static Facts**
0:UNLOAD GOODS1 TRUCK1 DEPOT1 LEVEL1 LEVEL0 LEVEL0 LEVEL1	0:DRIVE TRUCK1 DEPOT1 MARKET1
1:UNLOAD GOODS2 TRUCK1 DEPOT1 LEVEL1 LEVEL0 LEVEL0 LEVEL1	1:BUY TRUCK1 GOODS1 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1
2:UNLOAD GOODS3 TRUCK1 DEPOT1 LEVEL1 LEVEL0 LEVEL0 LEVEL1	2:BUY TRUCK1 GOODS2 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1
	3:BUY TRUCK1 GOODS3 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1
	4:LOAD GOODS1 TRUCK1 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1

	5:LOAD GOODS2 TRUCK1 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 6:LOAD GOODS3 TRUCK1 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 7:DRIVE TRUCK1 MARKET1 DEPOT1 8:UNLOAD GOODS1 TRUCK1 DEPOT1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 9:UNLOAD GOODS2 TRUCK1 DEPOT1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 10:UNLOAD GOODS3 TRUCK1 DEPOT1 LEVEL0 LEVEL1 LEVEL0 LEVEL1

LPG

Without Static Facts	**With Static Facts**
; Time 0.20 0:UNLOAD GOODS1 TRUCK1 DEPOT1 LEVEL1 LEVEL0 LEVEL0 LEVEL1 0: UNLOAD GOODS2 TRUCK1 DEPOT1 LEVEL1 LEVEL0 LEVEL0 LEVEL1 0:UNLOAD GOODS3 TRUCK1 DEPOT1 LEVEL0 LEVEL0 LEVEL0 LEVEL1	; Time 0.004 0:(DRIVE TRUCK1 DEPOT1 MARKET1 1:BUY TRUCK1 GOODS1 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 1:BUY TRUCK1 GOODS2 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 1:BUY TRUCK1 GOODS3 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 2:(LOAD GOODS1 TRUCK1 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 2:LOAD GOODS2 TRUCK1 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 2:LOAD GOODS3 TRUCK1 MARKET1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 3:DRIVE TRUCK1 MARKET1 DEPOT1 4:UNLOAD GOODS3 TRUCK1 DEPOT1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 5:DRIVE TRUCK1 DEPOT1 MARKET1 6:DRIVE TRUCK1 MARKET1 DEPOT1 7:UNLOAD GOODS2 TRUCK1 DEPOT1 LEVEL0 LEVEL1 LEVEL0 LEVEL1 7:UNLOAD GOODS1 TRUCK1 DEPOT1 LEVEL0 LEVEL1 LEVEL0 LEVEL1

The plans generated without static knowledge took more planning time with the wrong solution in the output while the solution with the static facts in the problem instance and domain took 0.004 seconds to produce the right plan.

2.5 The Problem Domains

To evaluate our developed method, we compare the performance of our method with fifteen existing standard IPC benchmarks as test cases. This is useful as these domains are already familiar to the planning and KEPS community. The next section only describes one domain that the methodology chapter mainly uses, as a running example; in order to better demonstrate the method's operation. Most of the plan libraries we use as input are among the collection of plans used in various planning competitions. The remaining input datasets for evaluation are produced by using Metric-FF and LPG planners. These two planners are selected due to their runtime performance (they can very efficiently solve all of the generated problems) and their ability to provide good quality plans. In this context, quality is measured in terms of the number of actions.

In almost all domains, ASCoL is able to produce a static graph that solves all the test problems. In next chapter on method evaluation, we describe all the significant datasets and complete experimental settings used to evaluate this thesis.

The Freecell Domain

All discrete puzzles and skill-based solitaire card games especially Freecell, provide fertile ground for research in AI. These have long been a subject of study in the literature and almost all work has been done on Freecell in the context of AI Planning. (Paul and Helmert, 2016, Elyasaf et al., 2011) (Sipper and Elyasaf, 2014, Fox and Long, 2001, Porteous and Sebastia, 2004, Russell et al., 2003).

As a running example in this section, we consider the Freecell domain that is one of the skill-based solitaire card games. The domain was introduced by Fahiem Bacchus in the AIPS-2000 planning competition and was later used by Long and Fox in the AIPS- 2002 (ICAPS, 2002) planning competition. It is a PDDL encoding of a card game which is similar to Solitaire and comes free with the Microsoft Windows (from Windows 95 onwards). The version of the game we chose as a demonstrative example is considered one of the most difficult domains in classical planning and the evidence is the poor performance of most of the general-purpose planners on the domain. The underlying ideas which the following chapters (especially Chapter 5) will discuss the application of the ASCoL algorithm to the Freecell domain, also apply not only to other variants of solitaires and domains evaluated in evaluation chapter but also to other classical planning domains that exhibit static aspects.

Figure 2.17: Microsoft Windows Freecell Game

A successful game involves a minimum of 52 moves and there are 52! (Factorial) different deals. Starting from an initial state, the Microsoft version of the game comprises a standard deck of 52 playing cards of four different suits (♠, ♣, ♥, ♦) and has a random configuration of cards across eight columns/cascades. To place a card in the column, it must be of different colour and one rank lower in value from the bottom card. To win the game, a user is required to move all the cards in an ascending order onto four home cells, following typical card stacking rules, and using four free cells as a resource. Figure 2.16 shows Freecell game configuration with bottom eight piles of cards called columns/cascades. Homecells/Foundations are four cells on the top right. Free cells are the four cells on the top left. Initially, columns are not arranged according to suits. The legal move in the current configuration includes, for example, moving 4♣ from the left most column on top of 5♥ in the second column in figure 2.17.

Although Freecell differs from other solitaire games, most of the concepts such as free cells, columns, and the rules of stacking are common to many solitaire games. The critical and the most complex situation while solving Freecell problems is the deadlock situation. It is a situation when one particular action, say A, cannot be executed because it requires action B as pre-requisite, which in turn requires action A to occur and the generalisation of such situations leads to a waiting condition. It has been recognised as a situation which prevents optimal solutions and makes domain theory hard. The deadlocks can be represented as cycles of the graph when the state space of the game is characterised by a directed graph (Paul and Helmert, 2016). A number of methodologies have been designed that exploit graph analysis in order to investigate the complexity of the problems (Gupta and Nau, 1992) (Helmert, 2003) and planning heuristics (Helmert, 2004).

The IPC PDDL encoding of the Freecell domain contains ten operators that implement the possible card moves. (Appendix A has a complete definition of the Benchmark Freecell PDDL Domain). The domain-specific static constraints/rules of the domain are the allowed sequential arrangement of cards in the free cells, the home cells and among the card columns, used within ten actions such as *colfromfreecell*, *sendtohomecell* and more.

In the encoding of the domain, each of the ten operators contains static knowledge, generally in the form of two static predicates, i.e., *canstack(?card1 ?card2 -card)* and *successor(?var1 ?var2 -num)*. The Freecell domain provides a suitable framework on which to demonstrate the effectiveness of our approach. We choose this domain as an illustrative example because all of its ten operators satisfy the condition (of having the same types) and it is rich in terms of static facts, e.g. *can-stack* and *successor* constraints.

Freecell plan traces only contain three types, i.e. *card*, *suit* and *num*. However, this domain is made more complex by the fact that one single type *num* is used to represent three different quantities (figure 2.18) and this, in turn, removes the boundary between three different behaviours. *num* is used to represent:

- the face value of cards (A,1,2, …, Q,K)
- to count free cells (0,1, …,4)
- to count free columns (0,1, …,8)

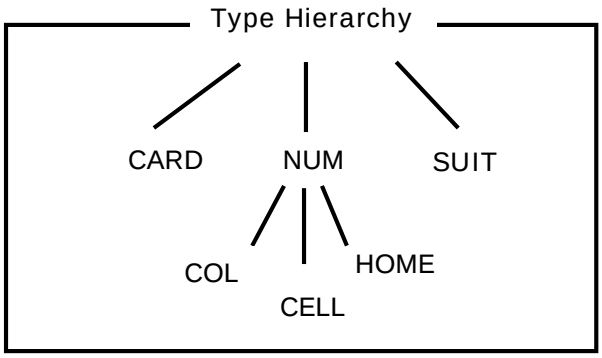


Figure 2.18: Type Hierarchy in Freecell Domain

Because of this variation of behaviour in type *num*, the *predicate successor(?var1 ?var2 -num)* is used in the following three variations:

- successor(?col1 ?col2 -num)
- successor(?cell1 ?cell2 -num)
- successor(?value1 ?value2 -num)

successor(?col1 ?col2 -num) keeps a record of the number of empty columns out of the total eight columns and *successor(?cell1 ?cell2 -num)* keeps a record of the number of

empty free cells out of the four available free cells. *successor(?value1 ?value2 -num)* allows the sequential arrangement of cards based on the face value of cards in the home cell.

Finally, the *can-stack(?card1 ?card2 -card)* allows the sequential arrangement of cards in columns based on the game rules, i.e., a card of suit Heart ♥ (red) can be stacked on top of a card with one higher face value but must be of suit Spade ♠ (black) or Club ♣ (black). Thus, there is a total of four unique static predicates in the domain which are repeated in ten unique operators. The complete Freecell domain uses these static facts sixteen times (in the ten operators).

It might seem that learning the same static fact for multiple operators in a domain may cause redundancy but, in fact, each operator uses the same static condition for different behaviour. For example operator *sendtohome* uses *successor(?value1 ?value2 -num)* for maintaining the allowed sequence of cards in the home cells while the operator *sendtofree-b* uses the same static fact *successor(?col1 ?col2 -num)* to keep the count of empty columns and *successor(?cell1 ?cell2 -num)* to keep the count of free cells.

To make it easy to understand and for a step-by-step explanation of the algorithm, we use the following PDDL encoding of an operator *homefromfreecell* from the benchmark Freecell domain (figure 2.18 Left). *homefromfreecell* is used to pick the card *?card* of value *?vcard* from a free cell space indicated by *?cells* and *?ncells* and stack it in a home cell over the home card *?homecard* of value *?vhomecard* of the same suit - *?suit*. *?cells* and *?ncells* variables keep a record of a number of available free cells. It should be noted here that four variables of the *homefromfreecell* operator are of type *num* and two are of type *card*. (successor ?vcard ?vhomecard) and (successor ?ncells ?cells) are two static facts exploited by the operator.

```
(: action homefromfreecell
        : parameters (?card - card ?suit - suit ?vcard - num ?homecard - card
                ?vhomecard - num ?cells ?ncells - num)
        : precondition (and (incell ?card)
                        (home ?homecard)
                        (suit ?card ?suit)
                        (suit ?homecard ?suit)
                        (value ?card ?vcard)
                        (value ?homecard ?vhomecard)
                        (successor ?vcard ?vhomecard)
                        (cellspace ?cells)
                        (successor ?ncells ?cells))
        : effect (and (home ?card)
                        (cellspace ?ncells)
                        (not (incell ?card))
                        (not (cellspace ?cells))
                        (not (home ?homecard)))
)
```

```
(: action homefromfreecell
        : parameters (?card - card ?suit - card ?vcard - num ?homecard -
    card ?vhomecard - num ?cells - num ?ncells - num)
        : precondition (and (zero_state0)
                        (card_state6 ? card)
                        (card_state0 ? suit)
                        (num_state0 ?vcard)
                        (card_state1 ?homecard ?suit ?vhomecard)
                        (num_state0 ?vhomecard)
                        (num_state0 ?cells)
                        (num_state0 ?ncells))
    : effect (and (card_state1 ?card ?suit ?vcard)
                (not (card_state6 ? card))
                (card_state7 ?homecard)
                (not (card_state1 ?homecard ?suit ?vhomecard)))
)
```

Figure 2.19: *homefromfreecell* - Freecell domain (Left) and LOCM (Right)

Figure 2.19 (Right) shows the same *homefromfreecell* operator induced by LOCM. This definition of an operator is informed by OLHE (Object Life History Editor) process from the parameterised state machine's transitions, induced as step 3 of the LOCM algorithm. Figure 2.20 shows the Induced FSMs corresponding to the action *homefromfreecell,* for the *card, num* and *suit* objects. LOCM creates one predicate corresponding to one state in each FSM. The predicates *card_state6*, *card_state1,* card_state0 and *card_state7* can be understood as *incell*, *home, suit* and *in_home_cell_and_covered* respectively. Appendix A has a complete definition of both Benchmarks: Freecell PDDL and LOCM: Freecell Domain in PDDL.

It should be noted that LOCM does not learn the background knowledge about objects i.e. the (*successor ?vcard ?vhomecard)* and (*successor ?ncells ?cells)* predicates, the adjacency between particular cards and the alternating sequence of black and red cards.

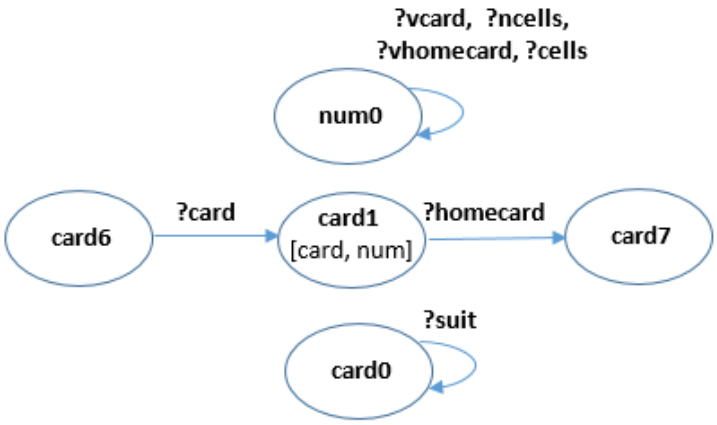


Figure 2.20: Induced FSMs for *card*, *num* and *suit* in action *homefromfreecell*

2.6 Related Work

Traditionally Domain Modelling for AI planners is done manually by hand in a text editor. To the best of our knowledge, all the domain models used in the past IPCs have been hand-crafted, and also those which are used in major applications e.g. in remote agent applications and automated manufacturing. Usually, simple syntax checkers are used to debug hand-coded domain models.

Research in Machine Learning has come a long way in acquisition/learning and improving action domains automatically from training and observations and still, there is considerable interest in learning for planning systems to attain self* properties in

autonomic agents. There are several knowledge engineering interfaces, translation and debugging tools with varying capabilities. These interfaces, translation and tools support AI planning, not only in the knowledge elicitation process but also for the design, validation and verification of developed models. They generate knowledge about actions and how objects are used by actions. This knowledge has to be satisfactory to allow efficient automated reasoning and construction of a plan as output. (Jiménez et al., 2012) reviews recent techniques in machine learning for the automatic definition of planning knowledge.

Two particular challenges for domain learning systems are:

(i) The amount of application knowledge required in the input to learn output domain model.

(ii) The extent of learning that takes place as the output of the system i.e. does the system produce a complete or near to complete domain model? Does the output entail both dynamic as well as static aspects of domain knowledge?

Using Machine Learning methods, several tools and techniques have been presented to facilitate the transformation process of real planning application requirements into a solver ready PDDL domain model that uses training or observation as inputs. These techniques use various types of knowledge besides plan traces, like general properties and constraints about domain actions, as well as partial knowledge about the kind of domain in which they are operating. To learn expressive domain models, systems tend to require more detailed inputs and substantial a priori knowledge which often include details about initial and goal state information. Some systems also require state information before and after action execution within each training plan. The main aim of this type of learning is to overcome the knowledge acquisition bottleneck, to help planning agents become more autonomous and make them able to adapt and plan for unseen situations and to debug existing domain models.

The domain model acquisition problem has mainly been tackled by exploiting two approaches. The first approach is to support human experts in modelling and debugging the knowledge. **O-plan** and **SIPE** are two of the pioneering projects that assisted KE using tool support (Tate, 1998, Tate et al., 1998) (Myers and Wilkins, 2001). Designed for the particular applications, O-Plan presented the Common Process Method while SIPE introduced the Act-Editor. To fulfil the need for tool support, several domain-independent and dependent tools came into being. A few particular examples are **itSIMPLE** (Vaquero et al., 2012), **GIPO** (Simpson et al., 2007), **JABBAH** (González-Ferrer et al., 2009), **EUROPA** (Barreiro, 2012), **MARIO** (Bouillet, 2009), **VIZ** (Vodrázka and Chrpa, 2010) and

PDDL Studio (Plch et al., 2012). A review of the state of the art is provided by Shah et al (Shah et al., 2013a)

On the other hand, a number of systems are currently available for automatic domain model acquisition; they rely mainly on example plans for deriving domain models through learning. Some of them include **ARMS** (Wu et al., 2005), **Opmaker** (Richardson, 2008), **Opmaker2** (McCluskey et al., 2008), **RIMS** (Zhuoa et al., 2013), the system of **Shahaf and Amir** (Shahaf and Amir, 2006), **LOCM** (Learning Object-Centred Models) (Cresswell, 2009, Cresswell et al., 2009) and the system of **Hanks** that uses crowdsourcing for model acquisition (Zhuo, 2015).

Significant differences can be found in terms of the quantity and quality of the required inputs for these systems. Chapter 4 provides an overview of the above mentioned automatic domain model learning techniques along with critical comparison among them. Shoeeb et al in (Shoeeb and McCluskey, 2011) indicate the challenges in evaluating domain model learning algorithms and present model classification for the comparisons of domain models.

To overcome the KE bottleneck, most of the learning systems (OpMaker, LAMP (Tsoumakas et al., 2004), LOCM) output the model in some variant of PDDL in their initial acquisition phase. For the sake of improvement in the initial version of the domain model, domain model adaptation systems e.g. RIMS (Zhuoa et al., 2013), can input an initially acquired domain model as well as training examples and output the updated and refined model. **EXPO** (Gil, 1992) is a learning-by-experimentation system for refining incomplete planning operators. It refines operators, which have missing preconditions and effects by failure-driven experimentation with the environment. Another example of refinement and incremental learning is **OBSERVER** (Wang, 1995) system. It learns an initial model and repairs it continuously during operation. It repairs plans from incorrect and partial domain knowledge. The framework learns planning operators by observing expert agents and subsequent knowledge refinement in a learning-by-doing paradigm. To refine the learnt operators, it solves practice problems with operators, analysing and learning from the execution traces of the resulting solutions. The method is implemented on **PRODIGY** (Veloso et al., 1995), an integrated architecture for planning and learning that include a general purpose planner and several learning modules that improve the planning domain knowledge and also the control knowledge to support the planning algorithm.

To provide support to the domain designer in encoding a correct domain model the **TIM** domain analysis tool (Fox and Long, 1998) learns state invariants using the automatically inferred type structure of the domain. This technique is used by STAN (Long and Fox, 1999), a Graphplan based planner to enhance overall planning performance.

LIVE (Shen 1989) is a system that learns rules by observing changes in the environment. It assimilates action exploration, experimentation, learning and problem-

solving. To create STRIPS-like rules, it requires actions and percept from the environment, a process to provide observation from the environment and the state description of the environment. (Benson, 1996) describes an inductive logic programming method for inducing a domain model using various positive and negative examples in addition to background knowledge.

(Grant, 2007) presents the idea of assimilating the domain knowledge from other agents and inducing a domain model from example domain states. He applied the induction algorithm successfully to eight different domains and assimilation is applied to the blocks world domain. The system presented in (Amir and Chang, 2008) generates a STRIPS-like domain model given as input the sequences of partial observations over time. It outputs a deterministic domain model that could have led to those observations. (Walsh and Littman, 2008) introduce a technique to acquire an action model for Web Service Composition.

(Zhuo et al., 2009b) provides two extensions of their work ARMS (Wu et al., 2005), in the form of a new system **LAMP** which learns domain models written in PDDL, or in other words in terms of quantifiers and logical implications. The LAMP system learns a domain model for an observable and deterministic environment from training plans with little or no pre-engineered domain knowledge including object types, predicate specifications, and action headers. It exploits a constraints solver to produce a domain model. The other extended system of ARMS learns domain models for hierarchical task networks (HTN), called **HTN-Learner** (Zhuo et al., 2009a). (Ersen and Sariel-Talay, 2012) also, presents a method that learns dynamic interaction among objects from a sequence of actions. The system uses Incredible Machine game for analysing interactions among objects.

Hoffmann et al in (Hoffmann et al., 2009) induce the business process models (BPM) using logs of actions recorded from real life business activity execution. The model then turns into the workflow. The main aim of the developed systems is to use the process mining technique to exploit the sequence of events. The process mining algorithm induces a model in the form of graphs such as Petri Nets. Process mining also deals with concepts of process synchronisation as it assumes that the concurrent process sequences cause events.

KEWI (Wickler et al., 2014) is another Knowledge Engineering technique that enables the technology of semantic web through Knowledge Engineering Web Interface (KEWI) for Modelling planning tasks in an object-centred way with automated plan generation tools. This is for situations where non-experts are required to encode knowledge. KEWI allows domain designers to encode their knowledge themselves, rather than knowledge engineers having to produce the knowledge before they formalise it into a representation. It produces the domain knowledge in an object-centred way which is

more expressive than PDDL. The conceptual model synthesised by KEWI consists of three layers: a rich ontology, a model of basic actions, and more complex methods. In KEWI, the requirements are first encoded in a more application-oriented language AIS-DDL and then mapped into planners' input language PDDL.

Recently, crowd-sourcing has also been exploited for acquiring planning domain models (Zhuo, 2015). It is worth noting that the problem of encoding domain models is being analysed not only from the point of view of generating models in a specific description language –such as PDDL but also for generating different sorts of automatically exploitable models. Konidaris et al in (Konidaris et al., 2014) proposed a method for constructing symbolic representations for high-level planning by establishing a close relationship between an agent's actions and the symbols required to plan to use them.

Learning from sources other than plans include learning a single state space using a system where an Oracle can validate plan hypotheses (Mehta et al., 2011). Another less common but useful category is Transfer Learning from other domains where operator schema are learnt using combined analysis of already existing domains and exploiting web queries in order to match operator names. (Taylor and Stone, 2009, Zhuo et al., 2008) are two techniques of this category.

Other than the planning community, there is much interest in automatic model acquisition in other fields. For instance, in robotics (Cakmak et al., 2010) (Bauer et al., 2008) e.g. RAX, in Image Processing (Clouard et al., 1999) (e.g. Vicar), Exploration Rovers (Bresina et al., 2005) (MER) and spacecraft assembly (Chien et al., 1999), constraint model acquisition (Bessiere et al., 2014), general game playing (Björnsson, 2012, Gregory et al.), software engineering (Reger et al., 2015) and computer security (Aarts et al., 2013).

Specialised Related Work

A first work in addressing the problem of inducing the static aspects missed when using LOCM is the **LOP** approach (Gregory and Cresswell, 2015). In order to extract static relations for extending LOCM-generated domain models, LOP exploits optimal goal-oriented plan traces. Specifically, LOP compares the optimal input plans with the optimal plans found by using the extended domain model. If the latter are shorter, then some static relations are deemed to be missing. This approach has drawbacks, as LOP strongly depends on the availability of optimal plans; they are usually hard to obtain for non-trivially solvable problems. It should be noted that trivial problems usually lead to extremely short plans, which tend not to be very informative for extracting knowledge. Moreover, LOP identifies the need for a static predicate between a set of action parameters but does not provide information about the type of relationship that connects the involved predicates.

Our system ASCoL (Jilani et al., 2015) is second in the series of systems which induce the static aspect of domain models in order to enhance the overall automated learning process for AI planning. In past research literature, we could not find sufficient number of findings to compare our results with and this is why we use standard IPC and FF domains as the benchmark and compare our outcomes based on isomorphic graphs. Explaination and complete details of the system are in Chapter 5 and 6.

Summary

This chapter presents the background studies that lay the basis of the research as well as a describes the main research areas with the emphasis on techniques of Knowledge Engineering for AI planning involved in the development of the proposed system. We also define some important terms that are commonly used in the planning literature and describe some of the survey results in the area. Our main aim is to link-up the recent and past research in the area. References are provided to link the text with the relevant literature.

The chapter also includes the learning problem this dissertation addresses and reflects on how it is challenging to learn from plan traces and what problems a knowledge engineer has to deal with using this input. It also describes the problem domain we use as our running example to explain the method in upcoming chapters. Towards the end, it includes the related work in the area.

Chapter 3 - Learning in Autonomous Systems

During the past two decades, the field of Machine Learning has emerged as an attractive field of research, investigating exciting research issues and developing challenging real-life applications. Machine learning is a broad area with a wide variety of sub-fields. It includes various methods starting from sub-symbolic methods like neural networks to high-level symbolic methods like Inductive Logic Programming. In our research, we are focused on the design of a learning mechanism in order to improve the learning capabilities of autonomous agents or systems.

Learning can be defined in many ways. The general definition of “Learning” is conceived as a change in behaviour but from the point of view of autonomous systems learning is anything that improves the overall performance of the system. There are classifications of learning and various ways that learning can be utilised by machines to underpin autonomic properties such as self-optimisation, adaptation, and configuration in connection with self* properties.

Benson in (Benson, 1996) indicates the factors on which the design of a learning mechanism depends:

- Firstly, it depends on what the system is trying to learn. Is it an achievement of certain goals in a particular domain? Does it need to design the complete domain model of a particular world? Is it trying to transfer knowledge from one domain to other?
- Secondly, what kind of input assistance is available to the learning system? Is any information about the surrounding world available? Is it supervised or unsurprised learning? Does a trainer indicate when something goes wrong in the case of supervised learning?
- Finally, the characteristics of the world to learn knowledge from also matters. For instance, is the world around discrete or continuous? Is it static of dynamic? Learning from discrete and static environmental characteristic offer lesser challenges than continuous and dynamic environmental features.

The remainder of this chapter will briefly discuss approaches for learning in autonomous systems and describe in more detail the choice of approach we adopted in ASCoL for learning static knowledge for planning domain models.

3.1 Approaches for Autonomous Learning

One form of learning from the environment is, for instance, the situation where the system has to build a complete domain model of the environment. When such a model is available, the agent can use various techniques like graph search to reason with knowledge of the actions available in the learnt domain model. Another simple approach can be to decide what actions to take in each particular situation. Following are four different categories of autonomous system learning and each has its own merits and demerits.

- Policy Learning
- Environmental Modelling
- Planning Domain Model Learning
- Specialised Knowledge Acquisition

3.1.1 Policy Learning

This is learning what to do in every possible situation. Based on the assumption that the goals of the learning system are static, and then instead of learning the domain model of the environment, the system can simply learn the reaction or response to common situations that may arise in the environment. As the system does not learn the domain model of the environment to take action rather it only learns the policies to respond to the exact situation this is why this approach is called policy learning.

Behavioural cloning (Bratko and Urbančič, 1997) also known as *learning by imitation* is an example of a policy learning approach. In behavioural cloning method, the learning system observes and reproduce the skills of the trainer agent (which is usually human) in carrying out the particular task. It then records the responses of the trainer in every situation along with the cause that gave rise to the response. A sequence of these responses is used as input to the learning system. When the learning system itself confronts some situation, it compares this situation with the learnt situation from the trainer and responds to act according to the learnt response of the trainer. It outputs a set of rules which reproduce the skilled behaviour. This technique has many advantages including the capability to quickly learn from a few training sessions and to act more reliably than a human trainer that it learnt from (Benson, 1995). This method is useful for building an automatic control system.

For complex domains, it appears to be a difficult exercise to train the system for all the possible values and situations that can occur inside the environment. However, in cloning the response to take according to a particular situation is usually same for a big set of possible state space, as the system can then generalise the response and develops

a policy to cover more inputs. ALVINN system (Pomerleau, 1991) is one of the best examples of behavioural cloning. Unlike supervised behavioural cloning, policy learning can also be done in an unsupervised way by Q-learning (Watkins, 1989) which is a method of reinforcement learning.

3.1.2 Environmental Modelling

Environmental Modelling (Shen, 1994) deals with the problem of learning a description about the environment. Developing a model using this approach offers most challenges but achieves good performance in autonomous systems. Theoretically, if the system contains a complete description of the world (which apparently seems impossible in dynamic and growing environments) it can design the optimal way to achieve the goals.
It requires the agent to create some assumptions about the nature of the respective environment first. (Benson, 1995) claims that based on the assumption that a respective environment can be modelled as a deterministic finite state automata, (where the agent's actions are the inputs and the agent's perceptions are output), then application of Rivest and Schapire's algorithm (Rivest and Schapire, 1993) for robots learning problem can develop an exact model of the environment with high probability. Vieira in his thesis on The Interplay between Networks and Robotics: Networked Robots and Robotic Routers (Vieira, 2010) investigates the amount of information that can be provided by the environment in which robots operate. In Modelling an environment as a topological map, the nodes on the map represent coarse-grained regions (a region with few and large discrete components) and the links connect the neighbouring regions.

Due to multiple reasons, e.g. noise in environment or presence of other agents in the same environment etc., many environments cannot be modelled correctly as deterministic finite automata. Therefore, it is important to choose the environment before attempting to learn it. The difficulty level of such a learning approach raises when the span of learning has to be increased from simple and toy environments to real-world complex environments.

3.1.3 Planning Domain Model Learning

The research in this thesis is generally concerned with planning domain model learning. For this reason, this approach is discussed in more detail.

Automated planning solvers can be found embedded in a wide variety of application areas. A planner requires a domain model that contains the action description, the objects involved in actions, a set of logical state space axioms along with the rules of inference and heuristics to accurately define the operators' description of some real world domain. It includes both a dynamic and a static object-type hierarchy, constant objects (if any)

and the declaration of predicates and functions (for hierarchical domains). In short, it is a declarative depiction of domain world functionalities.

Due to the increasing complexity of domain models required, the research focus is now shifting more towards the formulation, validation, and optimisation of planning domain models learning. The domain learning process has mainly been tackled by two approaches (Barták et al., 2013):

Handcrafted Methods: The manual method of writing a domain model from scratch in a text editor in the Modelling language supported by the solver, possibly assisted by the use of some automatic knowledge acquisition technique.
Automated Methods: A learning algorithm inputs the source to learn the domain model. The source can have variable forms; it can either be a set of plan scripts, a specification of a partial domain model or simple text. The output is a solver ready domain model.

In between the above two categories of domain Modelling process, there is a third less common but useful category of Transfer Learning from other domains. (Taylor and Stone, 2009, Zhuo et al., 2008) are two techniques of this category.

Before going into deep details, the clarification of terminology used for domain Modelling is important. The terminology used to discuss the concept of planning domain models is confusing. The term *Domain* is used to refer a specific area that exists in real life. *Domain Description* is used when the area is discussed and described in natural language in coded/symbolic form. When the description is complete and brought to precise form, it can be called a *Domain Definition* while the operational version of the domain is called a *Domain Model. Modelling* with the domain is to make use of the model to predict performance in the domain area.

Classes of Learning for Domain Modelling

For learning domain models for AI planning, various authors developed different KE tools and techniques that learn from different mediums e.g. Learning domain models from text (Kintsch, 1986), from plan traces (Wu et al., 2005) (which in some cases can be noisy and incomplete), by human demonstrations (Ontanón et al., 2009), by crowdsourcing (Zhuo, 2015) and by experimentation. This is of increasing importance: domain-independent planners are now being used in a wide range of applications, but they should be able to refine their knowledge of the world in order to be exploited also in autonomous systems. Automated planners require domain models described using languages such as PDDL.

Methods of autonomous domain model learning can be classified into two different ways:

- Refine the already available knowledge

- Acquire new knowledge from the environment

For refining already available knowledge, an intelligent agent requires automated skill acquisition characteristics for learning heuristics for the self-optimisation property of autonomy. To acquire altogether new knowledge, it requires automated knowledge acquisition property that further leads to self-configuration/maintenance properties of autonomy.

The characteristic classification is based informally on the “amount of autonomy” or essential processing needed to effect a change of behaviour in an agent. Learning can be by rote, by being told, through input examples (supervised learning), by observation (unsupervised), by analogy and by discovery. The amount of autonomy and the amount of processing required, increases from learning by rote towards learning by discovery (figure 3.1).

1. The simplest type of learning which does not manipulate the input at all is learning by rote, which totally depends on memorising facts so it requires the least processing and provides the least autonomy. It cannot integrate the upcoming input with already available knowledge. A database of facts is an example.
2. The second type of learning that involve programming and can understand/integrate knowledge is learning by being told. It includes programmed facts in the form of procedures and functions.
3. An example of learning by training is learning by examples. Here the system is supervised by some assistant which provides some example training data to deduce new knowledge. It generalises new knowledge with the input inferred knowledge.
4. The self-taught learning is by observation of examples; it includes more autonomy and less manual processing as it is self-guided without any assistance and supervision. It uses already collected knowledge to classify further input observations.
5. Learning by analogy is a mechanism to formulate and generalise from past successful experience. The level of autonomy at this level of learning is much more than learning by rote.
6. The highest type of learning in terms of autonomy is learning by discovery. It has the highest required processing as this type of learning is inquiry based and the agent uses its own prior knowledge to uncover the truth to learn.

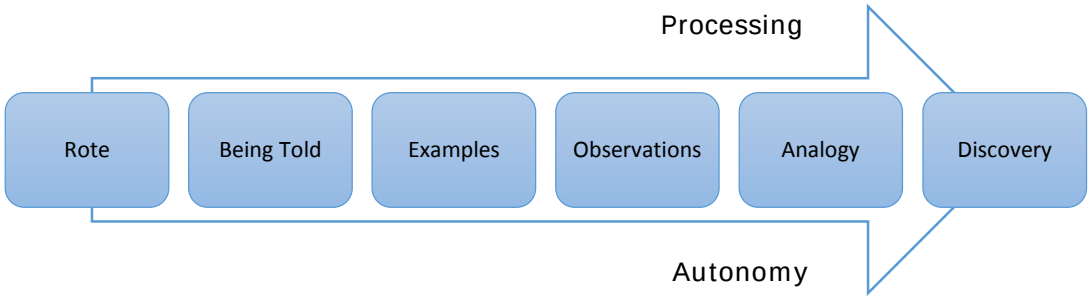


Figure 3.1: Autonomy and Processing required by Classes of Learning Sources

Concerns in Automated Domain Learning

Generally, domain model learning can have the following three concerns that are worth addressing before the formal setup (Tate et al., 2012):

i) What language will be used to encode domain model in?

Based on most of the research done on domain model generation, PDDL (Planning Domain Description Language) and its variants are being used to generate domain models (Fox and Long, 2006). It is based on an extension of first-order logic. It produces a domain model using a set of actions that represent operators. In the early days of automated planning, STRIPS was the most popular representation language. Later PDDL was developed for the first international planning conference (IPC) and since that date, it has become a standard language for the AI planning community (Fox and Long, 2003).

ii) What inputs are there to the learning process?

By inputs to the system, we mean the feed based on which the system generates a partial or full domain model. The input to the learning process could be training examples, observations, constraints, initial and goal states, predicates and in some systems a partial domain model to generate a full domain model. The input data can also vary based on its structure. Data can be unstructured, semi-structured or structured. The expressiveness of the learned domain model is directly proportional to the demand of more detailed input i.e. to learn expressive domain models, systems tend to require inputs that are more detailed. Some recent work on learning domain models has focused on learning with little or no supplied domain knowledge and LOCM (Cresswell et al., 2013) is one of such systems that learns from structured plan scripts only.

iii) What stage of learning is taking place i.e. how extensive the output domain model is?

The learning outcome is normally the acquisition of a set of operator schema or methods, but could also include a set of heuristics to be used to speed up planning. The output is a literal-based STRIPS-like domain model. Some systems generate an initial level of a domain model in first go that later could be refined and enhanced while some of them produce solver-ready domain models.

3.1.4 Specialised Knowledge Acquisition

This category of learning provides the basis for the research of this dissertation i.e. to learn a specialised component of the overall learning outcome in order to enhance or extend it. Our specialised knowledge acquisition focus, out of all available sources of learning, is to automatically learn a static constraints model for a planning domain model, from a logged sequence of action application knowledge (possibly plan traces). Our approach is called ASCoL (**A**utomatic **S**tatic **C**onstraints **L**earner) - a tool that exploits graph analysis, which has been designed to identify static facts automatically from an input set of actions, in order to enhance planning domain models.

Learning domain knowledge from plan traces as input training data has attracted much interest in research from early work in (Benson, 1995) to the work in the recent past in (Jilani et al., 2015).

For learning from training, the realistic input data becomes available through a dynamic environment using controllers. Controllers then interpret the data. By concentrating on the structure and type of input assistance, the objective input training data available in early research were limited to a structured form such as relational tables and transactions where each record is represented by one row in a table. However, the research within the last decade has extended the taxonomies of considered input training data to semi-structured form like multi-ordered graphs, symbolic sequences, and data represented in logic form. The main aim is to extract a meaningful pattern from data.

Significant differences can be found in terms of the quantity and quality of the required inputs by domain acquisition systems. The least quantity of required input to learn complete domain model is desirable for the system. These systems/techniques are discussed in Chapter 4 which also presents the critical analysis of these techniques. A shared feature of such systems is that they require the initial/ goal states and intermediate states between actions in sequence, in the input example plans. Some systems, in addition to example plans, also require static constraints, observations, and a partial domain model (with missing preconditions and effects) in the input to generate the final domain model. Other than plan traces obtained from the sensors (or observations), these input conditions

are hard to fulfil, since in many real-world situations we only have the action names and parameters in a given sequence, and do not know the states and existing partial domain model for the domain. The only situation when the intermediate states can be made available for learning is when the plan traces are acquired by a planning engine which can backtrack and find the intermediate states.

Learning Static Knowledge for Planning Domain Models

Apart from learning a dynamic domain model, many systems neglect one particular type of knowledge that cannot be learnt from the environment: The learning of a static object model.

Static relations represent the underlying structure of the domain that cannot be dynamically changed, but that affects the way in which actions can be performed – e.g. map learning, learning the fixed rules of domains (especially gaming domains), prefixed parameter units of certain quantities, ascending and descending order of quantities etc. Static facts are often thought of by their semantic construction (Gregory and Cresswell, 2015). They are essential for modelling correct action execution as static predicates are restrictions on the valid groundings of actions which help prune the search space in a planning phase. Our concern in the research is that the missing static knowledge can cause missing preconditions in the domain model. TPP domain example in Section 2.5 demonstrates the search space expansion problem due to missing static relations in the domain.

In all knowledge engineering for planning systems that learn from training data, the reason why static knowledge often gets neglected during the knowledge engineering phase is that for inducing a domain model automatically, efficient systems tend to input the least amount of knowledge and prefer not to exploit any input other than a sequence of actions (plan traces). Plan traces only contain knowledge about the dynamicity of the environment in the form of allowed sequence of action.

The LOCM family of algorithms is the best example of such systems. Moreover, for this reason, static knowledge or constraints cannot be easily observed by analysing only transitions of actions in plan traces. This is because static facts never change in the action sequence learnt from the environment. Such systems need manual assistance to declare static facts in the model after automatically inducing the dynamic model of the domain.

As already explained in the learning problem (section 2.2) we aim to acquire domain model from only action traces when the underlying domain model contains static knowledge. We have performed a number of experiments where systems produce a dynamic domain model from plan traces and we learnt a static object model using ASCoL approach to successfully build the comprehensive domain model.

What differentiates static constraints learning from other types of learning from plan traces is that the static facts are difficult to derive by observing transitions of actions alone. In ASCoL, we use syntactic interpretation for static graph analysis by exploiting a directed graph representation of graph theory, where the vertices of the graph are the instances of objects that belong to a type in the set of matching actions from the plan traces. Based on the output topology of a static graph, the ASCoL approach learns the semantics, order and type of static facts. We use the terms static facts, static knowledge, static constraints, static relations or static preconditions interchangeably in the next chapters.

3.2 LOCM Family of Algorithms

This section briefly describes the LOCM (**L**earning **O**bject-**C**entred **M**odels) and LOCM2 systems. It is included because our research aims to enhance the LOCM method and extend it to learn static knowledge as well besides learning dynamic part of the domain model. The basic aim of the LOCM family of algorithms (Cresswell et al., 2013) (Cresswell and Gregory, 2011) is to automatically induce domain models from training plans In the form of logged sequences of action applications. It is not given any prior knowledge of domain theory e.g. knowledge about predicates, sorts, actions, goals, initial states, intermediate states etc. It works on Machine Learning (ML) algorithms and supports knowledge engineering by automatic generation of planning domains in PDDL. The output of LOCM is a partial domain model consisting of sorts of objects, object behaviour defined by state machines, predicates defining associations between sorts, and action schema in a solver-ready PDDL domain model.

The LOCM approach is based on the idea of inductive generalisation i.e. generalise from the example data which illustrate the behaviour of one type of objects, to a principle about the whole type of objects. In automated planning, a set of plans generated by a planner can be a good training data as plans describe the set of actions and their possible occurrences in a sequence in order to achieve a goal(s). All the assumptions, hypothesis and the working steps of the LOCM algorithm are well explained by the authors already. The system is implemented in Prolog and incorporate their own algorithms.

3.2.1 LOCM

Assumptions in LOCM principle

LOCM uses certain assumptions about the action schema it has to induce. Following is the brief summary of such assumptions:

- That objects exist in the form of sets called sorts, where the behaviour of each object in a single sort is same. On action execution, all like objects in one sort can change certain specific states only.
- That LOCM has many observations to domain knowledge, and these observations are sequences of possible action applications.
- That action application on an object changes its state and bring it to some new object state. Each time when the same action will execute on the object, its preconditions and effects would remain the same.

Concisely, the LOCM produces the transition pattern of objects in each sort. By the coordination between transition patterns of all the sorts and the relationships between the objects of each sort, it induces the model in the form of Finite State Machines (FSM). Each FSM is augmented with parameters from same or different sorts in order to show the association with the rest of the FSMs. Each sort has a behaviour defined by a parameterized state machine. FSMs convert to the action schema comprising a domain model in PDDL. Following section briefly, describes the outlines of the complete algorithm in steps.

Outline - LOCM Algorithm

Input: Action training sequence

Output: PDDL domain model

Step 1. Create sort structure and Finite State Machines (FSM)

Step 2. Perform Zero Analysis and add new finite state machine if necessary

Step 3. Create and test hypotheses for state parameters

Step 4. Create and merge state parameters

Step 5. Remove parameter flaws

Step 6. Manually include static preconditions

Step 7. Form action schemas

End

Three Phases of LOCM

Generally, there are three phases of LOCM working principle. Each phase is briefly described below and has a separate algorithm.

Phase 1: Extraction of Finite State Machines (Steps 1 and 2)

LOCM develops a FSM for each of the identified sort. Many states in a sort's FSM represent state information of the objects of that sort. Generally, there is one state between two consecutive transitions within a FSM. One of the consecutive transitions in FSM is an incoming and the other is an outgoing transition. LOCM then learns the set of all transitions

taking place in the input plan sequences for each sort objects. A transition is represented as a combination of action name and action argument position e.g. for the transition of *block* object in predicate *putdown (?block ?table)*, would be represented as *putdown.1* in *block*'s FSM. Each action can give rise to multiple transitions where each transition represents a change in the state of an object. The FSM generates a separate starting and ending states for every new transition occurring in the input plans. It then captures the path of each object through each training sequence and generates the state machines for all the sorts.

Phase 2: Identification of state parameters (Steps 3-5)

LOCM follows object-centred representation that captures the dynamic relationship between objects in state parameters. Phase 2 of the algorithm identifies parameters of each state by observing patterns of object references in the original action sequence, corresponding to the transitions. This phase generally requires a larger amount of training data to identify state parameters than Phase 1.

Phase 3: Formation of action schema (Steps 6 and 7)

The output of Phase 2 (state parameters) provides correlations between the action parameters and state parameters occurring in the start/end states of transitions. The generated schema can be used directly in a planner after some manual developments. The LOCM algorithm has been evaluated on five different domain models based on three different criteria including Convergence, Equivalence and Adequacy.

As an example, Appendix A has a complete definition of the both Benchmark: Freecell Domain and LOCM: Freecell Domain in PDDL).

3.2.2 LOCM2

LOCM2 followed on from the LOCM idea. Experiments have revealed that there are many examples that have no model in the representation used by LOCM. A common feature of the domains, which exhibit this issue, is that the objects can have multiple aspects of their behaviour, and so they need multiple FSMs to represent each object's behaviour. LOCM2 generalises the domain induction of LOCM by allowing multiple parameterized state machines to represent a single object, with each FSM characterised by a set of transitions. This learns a varied range of domain models fully. LOCM2 uses a transition-centred representation instead of the state-centred representation used by LOCM. On conversion of the output PDDL representation from FSMs, LOCM and LOCM2, induced partial domain models only include dynamic facts with automatically generated unique labels. In order to use the domain model for planning, it needs a conversion of all of its states' labels to a sensible form.

The current LOCM and LOCM2 systems gather only the dynamic properties and cannot learn a static aspect of the planning domain. Domain designer has to manually include the static preconditions where required. Much research is carried out to fill that gap and make these systems able to induce both the dynamic and the static parts of domain models.

As mentioned in the last chapter and in the introduction section, this thesis explores the feasibility of learning static preconditions from the examples, rather than manually specifying them along with training sequences. We present a technique that automatically extracts the static knowledge from the same set of input action sequences used by LOCM (Chapter 5).

3.3.3 Experimental Work with LOCM

To investigate the behaviour of LOCM towards domain learning, we performed some additional experiments. The main purpose of the investigation is the comparison of some benchmark planning domain model, let us say X with the domain model, say Y that LOCM induced from plan traces (obtained from running a planner on X). By comparison, we checked what content i.e. predicates/states and objects (implicit) are missing in Y along with those preconditions that include a static aspect of the domain. Although LOCM authors have already done evaluations of the system on certain factors, we experimented with LOCM by following the Chapman's terminology (Chapman, 1987) used by Chrpa et al. in (Chrpa et al., 2013). We analysed domains based on the concept of Achiever and Clobberer. Chrpa et al. (Chrpa et al., 2013) define Achiever as follows:

Let o1 and o2 be planning operators. We say that o1 is an **Achiever** for o2 if and only if the intersection of positive effects of o1 and preconditions of o2 is not an empty set. Similarly, an operator is called a **Clobberer**, if and only if, the intersection of negative effects of o1 and preconditions of o2 is not an empty set.

Based on this concept of Achiever and Clobberer, we drew graphs of Blocks and Ferry world. We observed several differences in the graphs of real domains and the one generated by LOCM.

Ferry Domain

Following are the graphs of IPC Ferry domain (figure 3.2) and for Ferry domain generated by LOCM (figure 3.3).

As can be seen from the figures 3.2 and 3.3, the domain generated by LOCM based on goal-oriented input plans is adequate and it is equivalent to the domain model used to generate input training plans. Structurally, the only difference is that the LOCM does not induce the static fact (not-equal ?loc1 ?loc2). For that, initially, we used LOCM + statics

syntax to declare a static predicate of the domain in the training data for the domain, as the LOCM system cannot induce a domain with static aspects of objects. In LOCM + statics, this information is declared in the following form:

Static (not_equal (L1, L2), sail(L1, L2)).

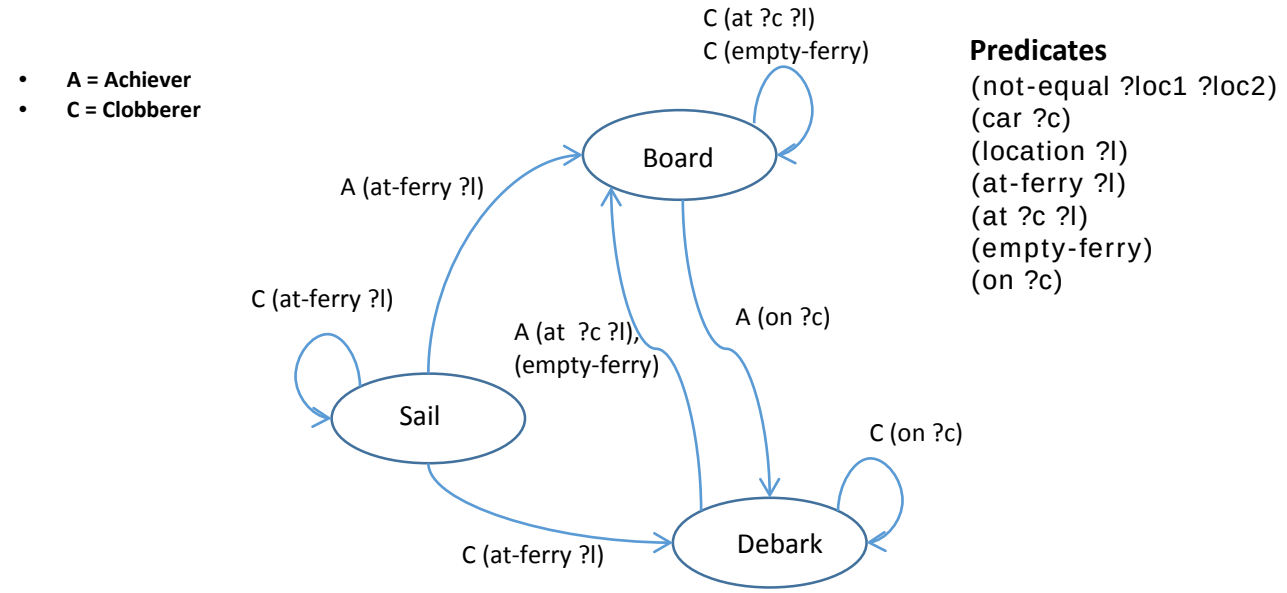


Figure 3.3: IPC Ferry Domain Graph

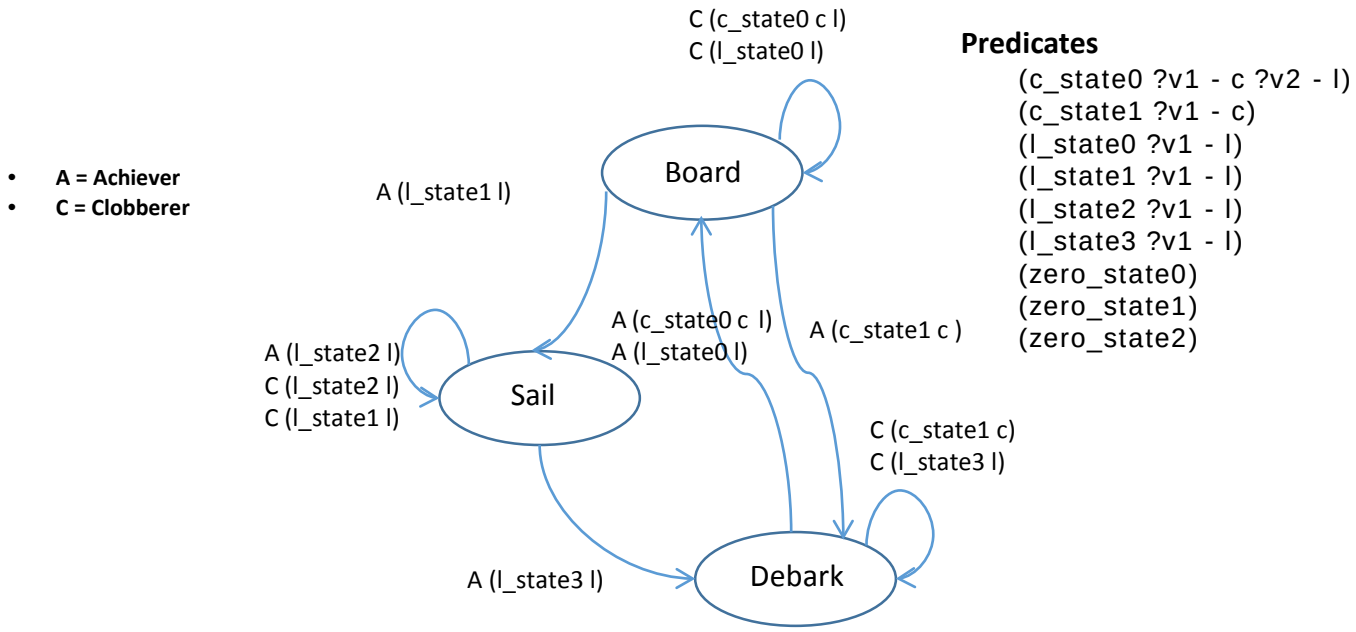
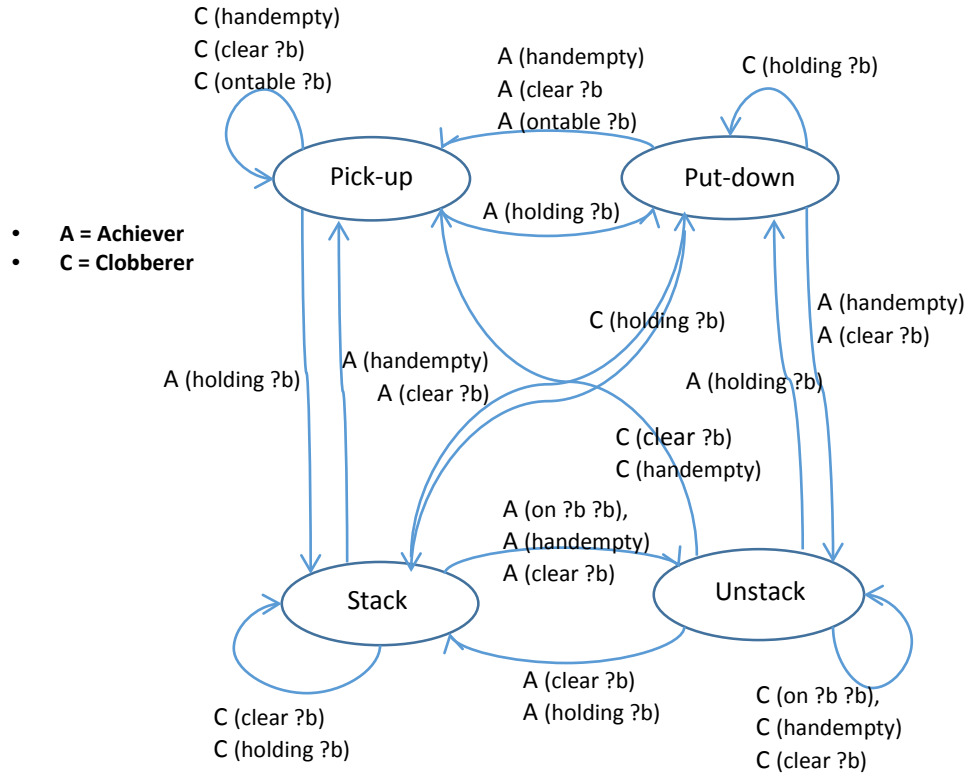


Figure 3.2: LOCM Induced Ferry Domain Graph

Blocks Domain

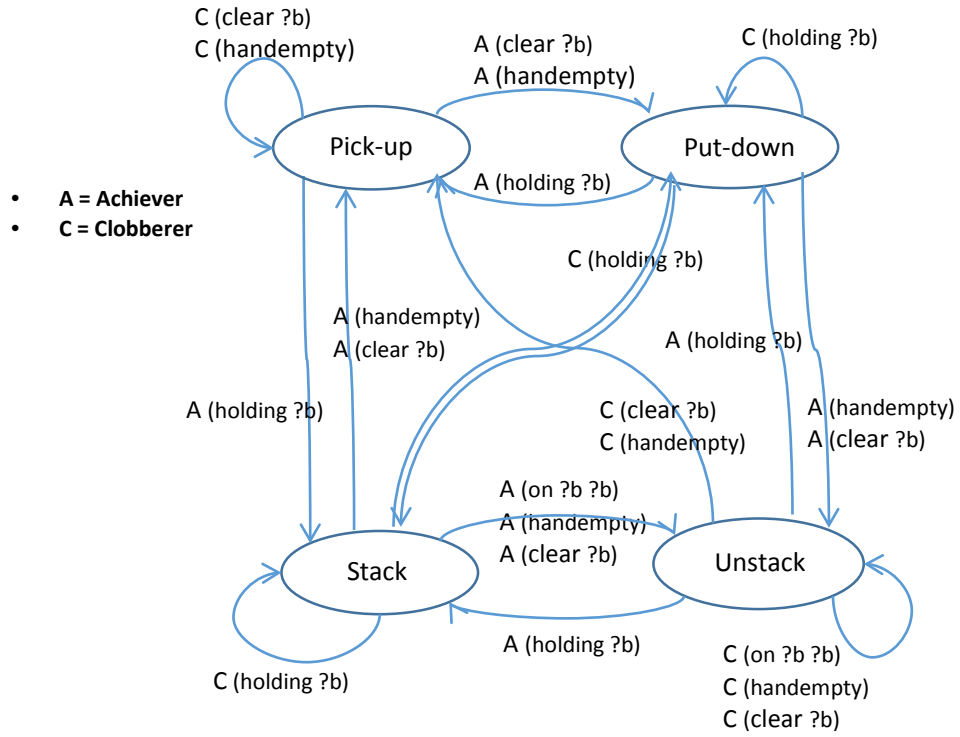
Following are the graphs of IPC four-operators blocks domain (figure 3.4) and for blocks domain generated by LOCM (figures 3.5).



Predicates

(on ?x - block ?y - block)
(ontable ?x - block)
(clear ?x - block)
(handempty)
(holding ?x - block)

Figure 3.4: Four-operator Blocks Domain Graph



Predicates

(b_state_1_1 ?b) = (clear)
(b_state_1_2 ?b) = (holding)
(b_state_1_3 ?b ?b) = (on)
(zero_state0) = (handempty)
(zero_state1) = (handholding)

Figure 3.5: Four-operator Blocks Domain Graph by LOCM

Both graphs are same except LOCM generates fewer predicates as compared to real blocks domain. The PDDL representation includes predicates with automatically generated unique labels (representing FSM states). This is why it is necessary for a human designer to examine and understand the state machines produced before making use of the induced model in a planner or for analysis.

Summary

This chapter discusses the approaches for learning in autonomous systems and describes in more detail the choice of approach we adopted in ASCoL for learning static knowledge for planning domain models. It also includes a section that briefly describes the LOCM (Learning Object-Centred Models) and LOCM2 systems. Towards the end, we presented our experimentation that we performed with the LOCM family of algorithms in order to make the comparison of some benchmark planning domain model.

Chapter 4 - KE Tools and Comparative Analysis

In this chapter, we present a brief overview of the automated tools that can be exploited to induce or assist in inducing planning domain models. While reviewing the literature on the existing tools for Knowledge Engineering (KE), we performed a comparative analysis. Outcome of this part of work has been published (Jilani et al., 2014). The analysis is based on a set of criteria. The aim of the analysis is to give insights into the strengths and weaknesses of the considered systems and to provide input for new, forthcoming research on KE tools in order to address future challenges in the automated KE area.

There have been reviews of existing knowledge engineering tools and techniques for AI Planning in (Vaquero et al., 2011) which provides a review of tools and methods that address the challenges encountered in each phase of a design process. Their work covers all the steps of the design cycles and is focused on tools that can be exploited by human experts for encoding domain models.

This chapter describes some of the KE tools for AI planning that automatically encode a domain model from observing plan traces. It also compares and analyses different state-of-the-art, automated KE tools that automatically discover action models from a set of successfully observed plans. Our special focus is to address the concerns of automated domain modelling discussed in the last section and analyse the design issues of automated KE systems, the extent of learning that can take place, the inputs that systems require and the competency in the output domain model that systems induce for dealing with complex real problems. We evaluate nine different KE tools against the following criteria: (i) Input Requirements (ii) Provided Output (iii) Language (iv) Noise in Plans (v) Refinement (vi) Operational Efficiency (vii) User Experience and (viii) Availability. By evaluating state-of-the-art tools, we can gain insight into the quality and efficiency of systems for encoding domain models, and better understand the improvements needed in the design of future supporting tools. It helps to give insight into which method and tools to use in order to better engineer a planning domain model.

4.1 KE Tools for Comparative Analysis

4.1.1 Opmaker

The Opmaker (McCluskey et al., 2002b) is a method for inducing primitive and hierarchical actions from examples, in order to reduce the human time needed for describing low-level details related to operators' pre- and post-conditions.

It induces parameterized, hierarchical operator descriptions from example sequences and declarative domain knowledge (object hierarchy, object descriptions, etc.)

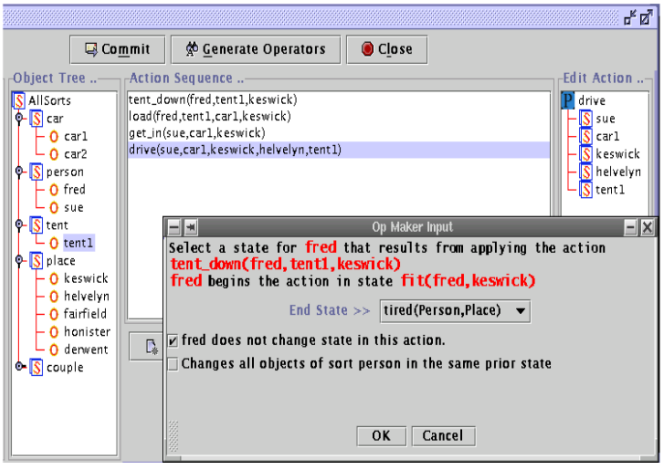


Figure 4.1: A screenshot of Opmaker

The user has to specify an action and identify associated objects as being affected or unaffected by the action. The system uses static domain knowledge, the initial and goal states and a planning sequence as input. Using this knowledge, it first deduces possible state-change pathways and then uses them to induce the needed actions. These actions can then be learnt, regenerated and improved according to the requirement. Opmaker extends GIPO, an integrated package for the construction of domain models, using a graphical user interface (Simpson et al., 2007). Figure 4.1 shows a screenshot of the graphical user interface.

4.1.2 SLAF

The SLAF (Simultaneous Learning and Filtering) algorithm (Shahaf and Amir, 2006) learns action models in partially observable (uncertain) domains. As inputs, SLAF includes specifications of fluents, as well as partial observations of intermediate states between action executions. The pre-conditions and effects that this system generates in output include implicit objects and unspecified relationships between objects by using the action schema language.

As output, the system learns action models (pre-conditions and effects) that also include conditional effects through a sequence of executed actions and partial observations. The action schema from this algorithm can be used in deterministic domains, which involve many actions, relations, and objects. This algorithm uses a Direct Acyclic Graph representation of the formula.

4.1.3 ARMS

The ARMS (Action-Relation Modelling System) (Yang et al., 2007) is a tool for learning action schema from observed plans with partial information. It is a system for automatically discovering action models from a set of observed plans where the intermediate states are either unknown or only partially known. To learn action schema, ARMS gathers knowledge on the statistical distribution of frequent sets of actions in the example plans. It then forms a weighted propositional satisfiability (weighted SAT) problem and resolves it using a weighted MAX-SAT solver. ARMS operates in two phases, where it first applies a frequent set mining algorithm to find the frequent subsets of plans that share a common set of parameters. It then applies a SAT algorithm for finding a consistent assignment of preconditions and effects.

ARMS needs partial intermediate states in addition to observed plan traces as input. The action model learnt from ARMS is not guaranteed to be completely correct, as the domain model induced is based on guesses with a minimal logical action model. This is why it can only serve as an additional component for the knowledge editors which provide advice for human users, such as GIPO (Simpson et al., 2007), and not as an independent, autonomous agent.

4.1.4 Opmaker2

Opmaker2 (an extension of Opmaker) (McCluskey et al., 2009) is a knowledge acquisition and formulation tool, which inputs a partial domain model and a training sequence, and outputs a set of PDDL operator schema including heuristics that can be used to make plan generation more efficient. It follows on from the original Opmaker idea. Its aim is similar to systems such as ARMS in that it supports the automated acquisition of a set of operator schema that can be used as input to an automated planning engine. Opmaker2 determines its own intermediate states of objects by tracking the changing states of each object in a training example sequence and making use of partial domain knowledge provided with input. Opmaker2 calls it the Determine State procedure. The output from Determine State is a map of states for each object in the example sequence. Parameterized operator

schemas are generated after applying the Opmaker algorithm for the generalisation of object references collected from example sequences.

4.1.5 LSO-NIO

The system LSO-NIO (Learning STRIPS Operators from Noisy and Incomplete Observations) (Mourao et al., 2012) has been designed for allowing an autonomous agent to acquire domain models from its raw experience in the real world. In such environments, the agent's observation can be noisy (incorrect actions) and incomplete (missing actions). In order to acquire a complete STRIPS (Fikes and Nilsson 1972) domain model, the system requires a partial model, which describes objects' attributes and relations, and operators' names.

LSO-NIO exploits a two-staged approach. As a first stage, LSO-NIO learns action models by constructing a set of kernel classifiers, which are able to deal with noise and partial observability. The resulting models are "implicit" in the learnt parameters of the classifiers (Mourao et al., 2010). The implicit models act as a noise-free and fully observable source of information for the subsequent step, in which explicit action rules are extracted. The final output of LSO-NIO is a STRIPS domain model, ready to use for domain-independent planners.

4.1.6 RIM

RIM (Refining Incomplete Planning Domain Models) (Zhuoa et al., 2013) is a system designed for situations where a planning agent has an incomplete model which it needs to refine through learning. This method takes as input an incomplete model (with missing pre-conditions and effects in the actions), and a set of plans that are known to be correct. By executing given plan traces and preconditions/ effects of the given incomplete model, it develops constraints and uses a MAX-SAT framework for learning the domain model (Zhuo et al., 2010). It outputs a "refined" model that not only captures additional precondition/effect knowledge about the given actions but also "macro actions". A macro-action can be defined as a set of actions applied at a single time that can quickly reach a goal at less depth in the search tree. Due to macro actions, problems that take a long time to solve might become solvable quickly.

In the first phase, it looks for candidate macros found from the plan traces, and in the second phase, it learns precondition/ effect models for both the primitive actions and the macro actions. Finally, it uses the refined model to plan. The running time of this system increases polynomially with the number of input plan traces.

In the RIM paper, the authors provide a comparison between RIM and ARMS by solving 50 different planning problems; through action models, refined and induced by the two systems. RIM uses both plans and incomplete domain models to induce a complete domain model but ARMS uses plans only, so to keep both systems' output on the same scale, RIM induces action models (used for comparison) based on plan traces only.

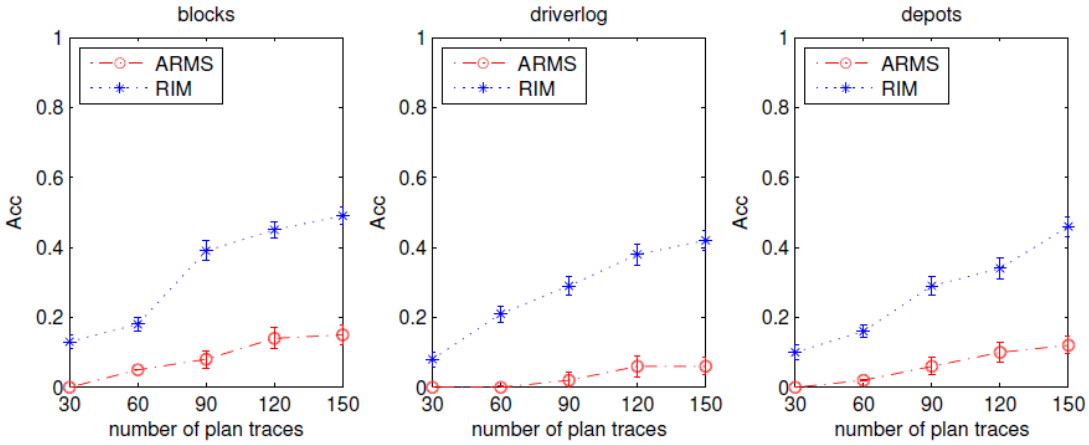


Figure 4.2: Comparison between RIM and ARMS

The average length of a plan is 18 when using action models learnt by ARMS; while the average length of plans (the same problems as solved by ARMS) is 21. This is when using preferences of macro-operators learnt by RIM. Figure 4.2 (Zhuoa et al., 2013) shows a comparison of three different domains; the correctness of RIM is better than ARMS, as RIM also learns macro-operators and it uses macros to increase the accuracy of plans generated with the refined system.

4.1.7 AMAN

AMAN (Action-Model Acquisition from Noisy plan traces) (Zhuo and Kambhampati, 2013) was designed to create domain models in situations where there is little or no possibility of collecting correct training data (plans). Usually, noisy plan traces are easier and cheaper to collect. Noise is inevitably introduced into plan traces when some sensors are occasionally damaged.

AMAN works as follows. It builds a graphical model to capture the relations between actions (in plan traces) and states, and then learns the parameters of the graphical model. After that, AMAN generates a set of action models according to the learnt parameters. Specifically, AMAN first exploits the observed noisy plan traces to predict correct plan traces and the domain model based on the graphical model and then executes the correct plan traces to calculate the reward of the predicted correct plan traces according to a

predefined reward function. Then, AMAN updates the predicted plan traces and domain model based on the reward. It iteratively performs the above-mentioned steps until it reaches a given number of iterations. Finally, it provides the predicted domain model.

In the AMAN paper (Yang et al., 2007), a comparison of AMAN and ARMS on noiseless inputs is provided.

4.2 Criteria for Evaluating Tools

Based on the function of the considered KE tools and focusing on what is important for the useful operation of the tools, several criteria are identified that are useful for evaluating the existing KE automated tools for inducing domain models. Such analysis is established for investigating the KE tools' practicality from different criteria including input, output, efficiency, availability, and usability.

Input Requirements

This is one of the challenging criteria in order to compare systems for their input demand. It investigates the required inputs by a system to refine/induce a partial or full domain model. The input to the learning process could be training plans, observations, constraints, initial and goal states, predicates, and in some systems a partial domain model (with missing pre-conditions and effects in the actions). The system which requires least amount of information in input and induce maximum and correct output domain model is considered more valuable.

Provided Output

What is the extent of learning that the system can do? To judge the correctness of the outcome, we compare it with the already existing benchmark domain models.

Language

What language does the system support to produce the output domain model? E.g. PDDL, STRIPS, and OCL etc.

Noise in Plans

Is the tool able to deal with noise in plans? Noise in plans can be either incomplete plan traces (i.e., missing actions) or noisy actions. An action in a plan is considered noisy if it is incorrectly observed.

Refinement

Does the tool refine existing domain models or does it build domain models from scratch?

Operational Efficiency

How efficiently are the models produced? In general terms, the efficiency of a system could be seen as the ratio between input given to the system to do the learning process and the output domain model that we get as a result of learning.

User Experience

Is the system/tool designed for inexperienced/beginner level planning users? Do users need to have a good knowledge of the system output language?

Availability and Usage

Is the system available for open use? Does the system provide a user manual?

4.3 Tools Evaluation

In this section, all the KE tools introduced in this paper are evaluated against the outlined criteria. Table 4.1 shows an overview of the comparison.

Inputs Requirements

The input to RIM, LOCM, LOCM2 and ARMS is a correct sequence of actions (training data in the form of plan traces), where each action in a plan is stated as a name and a list of objects that the action refers to. For some domains which require static knowledge, there is a need to mention static preconditions for the domain to be learnt; as LOCM and LOCM2 cannot learn static aspects of the domain. RIM in addition to a correct action sequence also requires an incomplete domain model (with missing pre-conditions and effects in the actions) as an input. ARMS makes use of background knowledge as input, comprising types, relations and initial and goal states to learn the domain model.

In comparison, Opmaker2 learns from a single, valid example plan but also requires a partial domain model (declarative knowledge of objects hierarchy, descriptions, etc.) as input.

AMAN and LSO-NIO, these systems learn from noisy (incorrect actions) and incomplete (missing actions) observations of real-world domains. Just like Opmaker2, LSO-NIO also requires a partial domain model, which describes objects (and their attributes and relations) as well as the name of the operators. The inputs to SLAF include specifications of fluents, as well as partial observations of intermediate states between action executions.

Provided Output

ARMS, Opmaker2, LOCM, LOCM2, LSO-NIO and RIM, the output of these systems is a complete domain model. In addition, LOCM also displays a graphical view of the finite state

machines, based on which the object behaviour in the output model is learnt. To increase the efficiency of plans generated, Opmaker2 also includes heuristics while RIM also learns macro operators.

SLAF, as output the system learns an action model (preconditions and effects) that also includes conditional effects through a sequence of executed actions and partial observations.

Given a set of noisy action plans, AMAN generates multiple (candidate) domain models. To capture domain physics it produces a graphical model and learns its parameters.

Language

The domain model (also called domain description or action model) is the description of the objects, structure, states, goals and dynamics of the domain of planning.

LOCM, LOCM2, and ARMS are able to provide a PDDL domain model representation. RIM, AMAN, and LSO-NIO can handle the STRIPS subset of PDDL. Opmaker and Opmaker2 use a higher-level language called Object-Centred Language (OCL) for domain modelling. Their output is an OCL domain model, but Opmaker can exploit the GIPO tool to translate the generated models into PDDL. Finally, SLAF System is able to exploit several languages to represent action schemas; starting from the most basic language SL, and then there is SL-V and SL-H (Shahaf and Amir, 2006). Domain-independent planners do not usually support such languages.

Noise in Plans

Most of the existing KE tools require valid plans. AMAN and LSO-NIO are the systems that can deal with noisy plan traces. Moreover, LSO-NIO is also able to handle incomplete plan traces. On the other hand, also LAMP (Zhuo et al., 2010), on which RIM is based, is able to exploit partial plan traces, in which some actions are missing.

Refinement

Most existing work on learning planning models learns a completely new domain model. The only tool among all those reviewed in the paper that is able to refine an existing domain model is RIM. RIM takes as input, correct plan traces as well as an incomplete domain model (with missing pre-conditions and effects in the actions), to refine it by capturing required pre-condition/effects.

Operational Efficiency

The efficiency of a system is seen as the ratio between the input given to the system for the learning process and the output domain model we get because of learning. As shown in the review of the considered tools, all systems have different and useful motivations behind their development. Given their relevant features of input requirements and learning extent, we can say that the system needing the least input assistance and which induces

the most complete domain model is the most efficient one. On such a scale, the AMAN, LOCM, and LOCM2 approaches have the best performance. In order to provide a complete domain model, they require only some plan traces (sequence of actions). Based on strong assumptions they output the solver-ready domain model. Similarly, ARMS, though requiring richer inputs, outputs a solution which is optimal; in that it checks error and redundancy rates in the domain model and reduces them. In contrast, Opmaker2 learns from a single example together with a partial domain model, and for output, it not only produces a domain model but also includes heuristics that is used to make plan generation through the domain model more efficient.

User Experience

By experience, we mean to evaluate how far the system is designed for use by inexperienced/beginner level planning users. The motivation behind most of these systems is to open up planning engines to general use. Opmaker is incorporated into GIPO as an action induction system, as GIPO is an integrated package for the construction of domain models in the form of a graphical user interface. It is used both as a research platform and in education. It has been used to support the teaching of artificial intelligence (AI) planning to students with a low experience level (Simpson et al., 2007).

The other systems are being used as standalone systems, they do not provide a GUI, and require the guidance of planning experts for usage. Certain systems also require separate formats for providing inputs, e.g., LOCM requires input plan traces in Prolog while many major planning engines use PDDL as a planning language. Therefore, the conversion from PDDL to Prolog is a time-consuming task and requires experienced users.

Availability and Usage

Very few systems are available on-line and open to download and practice. No systems provide documentation that make usage easy for beginners - except GIPO (Opmaker).
Following is the table that evaluates all nine (including LOCM and LOCM2) KE tools mentioned in the literature review section. Table 4.1 shows, P: Plan traces; BK: Background Knowledge; PDM: Partial Domain Model; Pr: Predicates; IS: Intermediate States; NP: Noisy Plans; DM: Domain Model; RDM: Refined Domain Model; H: Heuristics. Where available, +, i (intermediate) or - gives a qualitative evaluation w.r.t. the corresponding metric.

Criteria	AMAN	ARMS	LOCM	LOCM 2	LSO-NIO	Opmaker	Opmaker 2	RIM	SLAF
Inputs	NP	BK, P	P	P	PDM, NP	PDM, P	PDM, P	PDM, P	Pr, IS
Outputs	DM	DM	DM	DM	DM	DM	DM, H	RDM	DM
Language	STRIPS	PDDL	PDDL	PDDL	STRIPS	OCL	OCL	STRIPS	SL
Noise	+	-	-	-	+	-	-	-	-
Refinement	-	-	-	-	-	-	-	+	-
Efficiency	+	i	+	+	i	-	i	-	-
Experience	-	-	-	-	-	+	+	-	-
Availability	-	-	-	-	-	-	-	-	-

Table 4-1: Comparison of Knowledge Engineering Tools

4.4 Recommendations and Reviews

This section discusses the general structure, guidelines, and recommendations that are derived based on the review and assessment of the seven different state-of-the-art automated KE tools.

Technical review of different domain learning systems and methodologies reveals that a typical domain learning system consists of different components to accept plan traces or sometimes additional information from a user and deliver a full or partial domain model back to the user. These components are used to learn from the input information that user pass and through a sequential and logical arrangement of actions in plans, they exploit learning methodology of the system to generate a literal-based STRIPS-like domain model which consist of action schema/operators (figure 4.3).

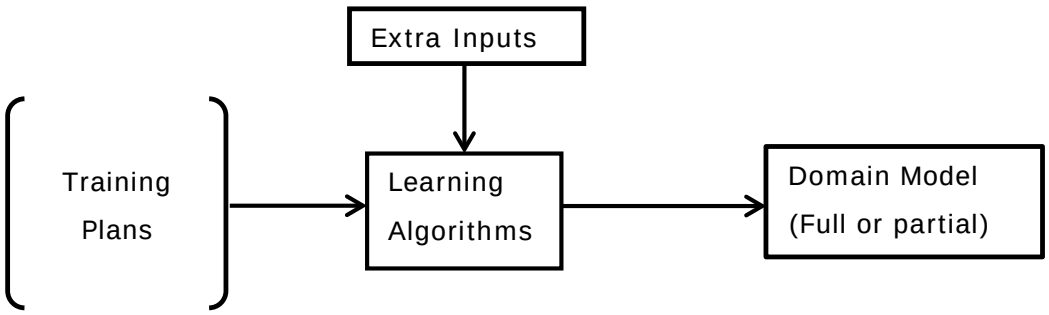


Figure 4.3: General Architecture of Domain Learning System

Some systems generate an initial level of domain model in the first attempt that later could be refined and enhanced while some of them produce solver-ready domain models as well.

Input to the learning process could be training plans, observations, constraints, initial, intermediate and goal states, and predicates and in some systems a partial domain model to generate a full domain model. The correctness and accuracy of the learned domain model increase with the increase in the demanded input assistance. The Latest work on such systems makes them enable to work only with multiple instances of plans as input.

Collection of training plans is one major concern at this stage. There are three general ways to collect example plans. The first is when plans are generated through goal-oriented solutions, the second through random walks and thirdly through observation of the environment by a human or by an agent. Goal-oriented plan solutions are generally expensive in that it requires a tool or a planner to generate a large enough number of correct plans. To do this one must also have a pre-existing domain model. Observation by an agent has high chances that noise can be introduced in the plan collection; which can clearly affect the learning process. Currently, most working systems assume the input knowledge to be correct and consequently are not suitable for real-world applications. To increase potential utility, systems should be able to show equal robustness to noise.

Another issue is the expressiveness of the output domain model. Observing the output of the current automated learning systems, there is a need to extend their development so that they can also learn metric domains that include durative actions, action costs and other resources. In other words, to increase the application range generation and produce expressive versions of PDDL, the systems should broaden the scope of domain model generation.

Systems that learn only from plan traces could make the output domain model more intelligible and useful by assigning meaningful names to all learnt fluents/predicates. To enhance the potential utility of the induced domain in the real world and to enhance the effectiveness of plans generated by planning engines, systems should perform error and redundancy checks.

To make learning systems more accessible and open to use by research students and the scientific community, these systems should be available on-line and include a GUI and user manual for ease of use by non-planning experts. A significant extension would be to create a consistent interface across all systems for specifying inputs. Having to convert PDDL plans into Prolog, for example, is likely to inhibit the uptake of automated KE tools by non-experts rather than encourage it.

Chapter 5 - ASCoL

After covering theoretical concepts, evolution, general architecture and overview of a few recent domain-learning systems in previous chapters, this chapter describes ASCoL (**A**utomatic **S**tatic **Co**nstraints **L**earner) - the outcome of this research and core methodology, in much detail supported by diagrams and other visual aids in order to support readers' understanding. The theoretical foundation in previous chapters provides the base of the proposed system. This chapter includes a discussion of the characteristics of the underlying idea, overall method structure and the implementation details of the developed system as the evidence of the concept of this research. All the concepts are described along with running examples for easy understanding. The assessment and evaluation methods used to test the validity of the system are described in the next chapter.

5.1 Introduction

Among many approaches to solving complex planning problem, one is to learn and exploit domain-specific constraints for supporting the performance of domain-independent planners by directing and controlling the search process. Since the early days of AI planning, much work has been done on different Machine Learning techniques with the purpose of automatically learning search-control knowledge. A few examples of these techniques are macro-actions (Fikes et al., 1972), control-rules (Borrajo and Veloso, 1997), and case-based and analogical planning (Veloso, 1994).

Static constraints or invariants restrict the traversal of unnecessary reachable states in solution search. Without using these constraints planners are slower because they have wider search space options to search for solution of a problem and most of the times produce invalid plans. Examples of static constraints include the connections of roads in the logistics domain, the level of floors in the Miconic domain and the fixed stacking relationships between specific cards in the Freecell domain.

In the domain modelling process, domain experts or knowledge engineers provide constraints. However, it is difficult for humans to formalise it. Most systems that synthesise constraints automatically do so by the analysis of the operators of the planning world, for example (Rintanen, 2000); some discover constraints from state descriptions (Mukherji and Schubert, 2005, Lin, 2004) while some use both initial states and operator descriptions (Gerevini and Schubert, 1998).

This chapter presents a general system ASCoL that learns static constraints from previous experience. The system aims to support domain-independent planning,

knowledge engineering, and even non-expert users in the modelling phase of planning domains. It learns domain-specific static constraints or invariants for domain models in the form of a static graph of ordered static rules (positive preconditions). It demonstrates the feasibility of automatically identifying static relations/constraints for domain models by considering application knowledge in the form of training plans for a range of application areas. In the section 2.3, we have explained the reason for using training plans as the source of learning. The system uses events or logs recorded from a real life process execution. These logs of actions or plan traces can be obtained from an instructor, control system execution or a planner. Like the LOCM systems, input training plans to ASCoL can be either optimal/suboptimal, goal-oriented or generated by using random walks.

ASCoL is a graph-based learning method that applies the concepts of graph embedding and combines it with graph analysis methods in order to discover the syntactically and semantically correct static knowledge structure. Here knowledge structure means that the semantically related pieces of knowledge should be grouped together and that different types of knowledge (constraints, facts etc.) should be represented by different knowledge constructs.

Graph embedding and analysis theory is particularly suitable to develop this logical static facts learning approach because graphs can be used to represent logical semantics (meaning of propositions and of their formal analogues) without using the language of logic but visual notions. In addition, because many efficient graph-processing algorithms exist, thus graphs can be exploited as a good computational mechanism to compute relations between objects (more on the benefits of using graph theory is discussed in Background section 2.1 - III). Graphs that represent static relationships show the permanent relationships between constant objects of a problem (Botea et al., 2005).

Complete detail on the implementation of each part of ASCoL method is provided in section 5.3. Here we present a quick overview of the system in the form of a workflow diagram in figure 5.1:

Input: In order to avoid manual syntax conversion of the numerous plan traces (generated by the planners) to be used as a test set, an additional software program was developed called *Sequence Generator*. It allows the users to transform the multiple input plan traces files (obtained from planners or any other source) to the single desirable Prolog format. LOCM domain model is the partial domain model generated by LOCM which needs improving with (desired) static knowledge. ASCoL captures the types (sorts) of objects from this input LOCM domain after parsing it along with plan traces' file.

Parsing: For the process of graph embedding, the File Parser parses both the sequence file and the domain model and tokenizes them to be used in the next step where edges, vertices and their labels are generated for the purpose of graph composition.

Graph Embedding / Analysis: After passing through multiple steps of generating edges, vertices and their labels, directed graphs are generated and analysed. The system induces static facts based on the order analysis of the graphic outcome. These static facts define the object behaviour in the form of predicates in PDDL.

The induced static predicates for an operator have the fixed arguments from the operator argument list, where each argument ranges through object instances belonging to some fixed type. ASCoL augments components of the LOCM generated domain with enriched static constraints in the form of positive preconditions as a set of first-order predicates.

The overall objective is to use the full system "LOCM + ASCoL" for generating a complete domain model including both static and dynamic knowledge by exploiting the minimum amount of input and to support the automated acquisition of a set of operator schema for automated planning engines.

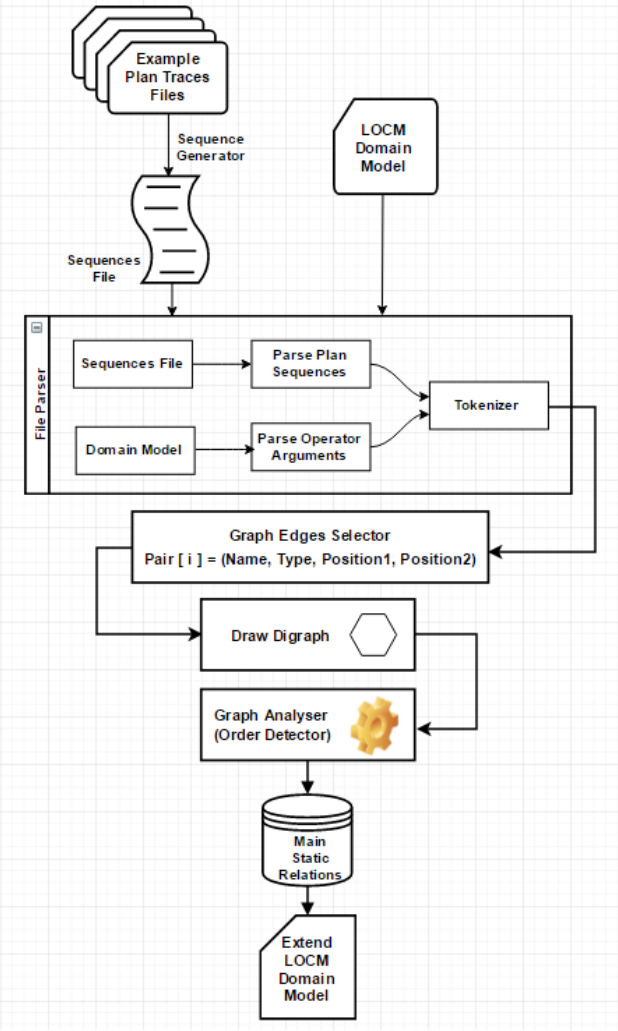


Figure 5.1: ASCoL Method Overview

5.1.1 Preliminaries

In this section, we define terms from a wide class of graph theory to better clarify the concepts in the following sections. We summarise the definitions at the semantic level; for syntactic details see section 5.2.2 – Generation of Digraphs:

Directed Graph / Digraph is a set of nodes/vertices that are connected together, where all the edges are directed from one vertex to another or in other words, the edges are comprised of ordered vertex pairs.

Indegree is a number of incoming edges to a vertex.

Outdegree is a number of outgoing edges from a vertex.

Source is a vertex with Indegree 0.

Sink is a vertex with Outdegree 0.

Loop is an edge that connects a vertex to itself.

Path is a sequence of edges which connect a sequence of distinct vertices. For v_1, v_2... v_n ordered vertices there are (v_i, v_{i+1}) edges where i = 1, 2... n–1.

Neighbour is an adjacent vertex of a vertex v in a graph that is connected to v by an edge.

Bipartite graphs / Bigraph is a graph whose nodes can be partitioned into two sets of vertices such that the edges only exist between the two sets but not within the vertex set. It is denoted by K_{mn}, where *m* and *n* represent the two sets of vertices.

Partially Ordered relation / graph A Relation R on a graph G is called a partial order if it is reflexive, antisymmetric and transitive. A graph G together with a partial ordering R is called a partially ordered graph or pograph for short.

Total Ordered relation / graph If (G, <) is a pograph and every two vertices of G are comparable then G is called a totally ordered graph. The relation < is said to be a total order.

Cyclic / Circular graph is a graph with some number of vertices connected in a closed chain.

Acyclic graph is a graph having no graph cycles. Acyclic graphs are bipartite.

5.1.2 Assumptions of ASCoL

In our technique, all kinds of static relations are represented as graphs, where the graph vocabulary can be seen as the object instances of particular type which can be taken from a live activity record(s) or from manually recorded log of action sequence. Although ASCoL does not require any other input, it makes the following minimal **assumptions** that must be fulfilled:

Assumption 1: That the dynamic structure of the domain model provided as input includes the Type of each operator argument.
Assumption 2: That the input training sequences (plan traces) are generated by considering problems that share the same objects, and object names.
Assumption 3: For the presence of a static fact, the operator must contain at least two arguments of the same type (argument provided in section 5.4).

ASCoL works by exploiting a directed graph representation, where the vertices of the graph are the instances of objects that belong to a type in the set of matching actions from the plan traces. In terms of plan traces, the second ASCoL assumption is particularly reasonable when training sequences come from real-world application observations; in that scenario, sensors are identifying and naming objects – they usually exploit fixed names. Moreover, it is acceptable to assume that the relevant objects of a real-world scenario will not change quickly. For example, machines in a factory or trucks and depots for logistics companies.

Moreover, ASCoL assumes that the input plan sequences are correct while the input domain file at least contains type information for all those operators that the algorithm aims to enhance.

Requirements: The ASCoL algorithm also has certain requirements. The training plan sequences are acceptable only if they are *correct* and *complete*: By correctness, it requires that the input training plans be noise-free. Noise can be introduced into plan sequences when some sensors are occasionally damaged or with unintentional mistakes in the recording of the action sequence. By completeness, it means that plan traces should cover all the actions and all the possible combination of actions in the domain based on Achiever and Clobberer rule (Chrpa et al., 2013). In addition, ASCoL restricts itself to the learning of binary static relationships only. Chapter 8 Section 8.1 discuss the complete list of system requirements and restrictions.

5.2 ASCoL Algorithm

This section explains ASCoL core methodology in detail supported by diagrams and other visual aids to assist the readers' understanding. Following is the summary of ASCoL algorithm:

1. Read and parse the (LOCM generated) partial domain model and the input plan sequences.
2. Identify, for each unique operator in the plan sequences, all pairs of arguments involving the same object types.
3. For each of the pairs, generate a directed graph by considering the objects involved in the matching actions from the plans.
4. Analyse the directed graphs and extract hidden static relations between arguments.
5. Run inequality check (Reflexive property).
6. Return the extended domain model that includes the identified operator-specific static relations.

Following are the three main steps of the methodology with the discussion on each in the following subsections:

Step 1: Generation of Vertices Pairs

Step 2: Generation of Digraphs

Step 3: Analysis of Directed Graphs

Step 1 of the methodology covers the first and second points of the algorithm summary. **Step 2** of the methodology covers the third and **step 3** covers the fourth and fifth points of the algorithm summary. Sixth point of the summary is the representation of the learnt knowledge and its conversion to PDDL format (discussed in a subsection 5.2.4 ahead).

In the Chapter, examples from a wide class of planning domains are included to better clarify the concept under discussion. All the examples quoted in the thesis are taken from IPC (ICAPS, 2002) domain collection and can be found in de facto official versions of PDDL.

5.2.1 Step 1: Generation of Vertices Pairs

The input to the static constraints learning algorithm ASCoL is specified as a tuple (P, T), Where, $P = P_1, P_2, \ldots, P_n$

P is the application knowledge in the form of a set of plan traces. In the input, plan traces are represented in a text-based format (the same as that supported by LOCM). Each plan contains an action sequence of N actions on numerous objects i.e. each $P_i \in P$ has the form:

P_i (A_1, A_2,..., A_N) for i = 1, ..., N

Where *A* is the action name. Each action A is made up of an identifier (the name of the action), and the names of objects that it affects, in the order of occurrence, which all have the form:

A_i (O_{i1},..., O_{ij}) for i = 1, ..., N

Where *O* is the action's object name. Each action *A* in the plan is stated as a name and a list of parameters. In each action A_i, there are *j* parameters where each parameter is an object *O* of some type T.

T is a set of types of action parameters in P. The ASCoL parses the types from the LOCM output domain such that:

n(T) ≤ n(O) types T = t1, t2,... tn.

The assumptions and requirements (in the previous section), and the constraints that they cause lead to an algorithm used in step 1 of the ASCoL. After reading and parsing the partial domain model and the input plan sequences, it induces the universe of affected actions and same-typed object instances. The identified same-typed object instances make the elements of the graph vocabulary. The algorithm is provided below in pseudo-code. In lines 2. through to 5., it removes from the parsed plan sequences file, all the actions which refer to operators that do not contain at least two parameters of the same type (Assumption 3).

Algorithm to detect valid actions

Input: A set of training (plan) sequences *P* of length *N*

Output: Action Set (AS) that contains all unique actions that satisfy Assumption 3 (figure 5.3)

1. Initialise Action Set AS to empty
2. **if** Plan *P* is not empty **then**
3. **Iterate** through A_i, i ∈ 1,..., N as follows:
4. **if** (A_i. parameters. sameTyped > 1) **then**
5. add A_i to AS

end

end if

Example to illustrate detection of valid actions: Freecell Domain

Figure 5.2 shows an example of one of the Freecell input action sequence to recognise the batch of valid actions (AS). The input action sequence satisfies assumption 3 to produce the output in figure 5.3. In figure 5.3 – *card*, *num* and *suit* represent the types T of objects.

```
A1: sendtofree (spade9, heart7, n1, n0),
A2: move (heart7, diamonda, spade8),
A3: sendtohome (diamonda, spade3, diamond, n1, diamond0, n0),
A4: move (diamond4, diamond9, club5),
A5: colfromfreecell (club8, diamond9, n0, n1),
A6: sendtohome_b (club3, club, n3, club2, n2, n0, n1),
A7: homefromfreecell (club4, club, n4, club3, n3, n0, n1),
A8: sendtonewcol (spade3, diamond4, n1, n0),
A8: sendtofree (diamond4, club5, n1, n0),
A9: sendtohome (club5, diamond8, club, n5, club4, n4),
A10: sendtohome (club6, heart7, club, n6, club5, n5),
A11: move_b (spade3, heart4, n0, n1),
```

Figure 5.2: Input - A training sequence of length 12 (Freecell)

```
sendtofree (card1, card2, num1, num2)
move (card1, card2, card3)
sendtohome (card1, card2, suit, num1, card3, num2)
colfromfreecell (card1, card2, num1, num2)
sendtohome_b (card1, suit, num1, card2, num2, num3, num4)
homefromfreecell (card1, suit, num1, card2, num2, num3, num4)
sendtonewcol (card1, card2, num1, num2)
```

Figure 5.3: Output - Action Set containing all actions that satisfy Assumption 3 (Freecell)

For some domains, the training sequences do not capture the complete set of actions because of the rare occurrence of some actions. An example of this occurs in the Freecell domain where the two operators including *sendtofree-b* and *newcolfromfreecell*, prevents ASCoL from learning the corresponding static relations from just a few plans. A large number of plans are required; at least 20 (of average 60 actions per plan). However, only 8 plan traces are required for correctly identifying the static relations of the remaining operators.

Identification of same-typed parameter pairs

The algorithm has identified the types (T) and the operators that satisfy the aforementioned assumptions and that provide the vocabulary for graphs. The next step is to identify all the possible combination pairs of same-type objects in each unique operator. The formula to calculate the total number of such pairs for each operator header is:

(n-1) + (n-2) +…+ (n-n); n = Total number of same typed arguments

For instance, if in an operator, the number of available same-typed arguments is four then the total number of pairs would be 3+2+1+0 = 6. This can better be represented in the form of general bipartite graph *(V1, E, V2)* where $V1 = O_{i1},\ldots,O_{i(n-1)}$, $V2 = O_{i2},\ldots,O_{i(n)}$, $i = 1,\ldots n$ and *O* represents the action's object names. *V1* and *V2* are not fully disjoint sets of vertices and *E* denotes the edges of the graph with V1 as source and V2 as sink nodes.

Example to illustrate same-typed parameter pairs: Freecell Domain

The Bipartite graph in figure 5.4 represents the ground action instance of *homefromfreecell* (?card - card ?suit - suit ?vcard - num ?homecard - card ?vhomecard - num ?cells ?ncells - num) that includes four same-typed object instances. i.e. of type *num*. Here V1 = [vcard, vhomecard, cells], V2 = [vhomecard, cells, ncells].

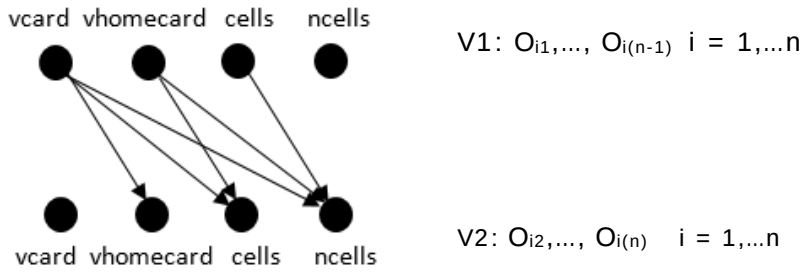


Figure 5.4: Bigraph for generating pairs of arguments from operator definition

An edge is generated based on the permutation of objects which makes the first argument of the pair the source and the second argument the sink of an edge in a graph. Therefore, an edge *(o1, o2)* is an outgoing edge from *o1* and an incoming edge to *o2*. All the vertices (arguments) in the bigraph must represent the same order as mentioned in the benchmark operator definition i.e. *vcard* comes before *vhomecard*, *cells*, and *ncells* in the operator. This is because the order in arguments is important in finding the relationship between object instances.

For type *num*, the following six vertex pairs are identified for the action *homefromfreecell*:

pair1 (?vcard, ?vhomecard), pair2 (?vcard, ?cells), pair3 (?vcard, ?ncells), pair4 (?vhomecard, ?cells), pair5 (?vhomecard, ?ncells), pair6 (?cells, ?ncells).

At the end of Step 1, ASCoL has induced a set of graph vertices and corresponding edges for each unique operator in the plan sequences. Each of the induced edges is identified as a pair of action parameters that fulfil assumptions 2 and 3.

5.2.2 Step 2: Generation of Digraphs

After identifying for each unique operator in the plan traces, all the pairs of arguments involving the same object types, this subsection describes the generation of digraphs after the generation of vertices pairs.

Consecutive Actions with respect to an Object Instance:

Let A_i^{th} and A_j^{th} be the two matching action instances in a plan. $O_{i,k}$ *and* $O_{j,k}$ are associated action objects which belong to the corresponding actions' pairs P_i and P_j. A_i^{th} and A_j^{th}, are considered consecutive w.r.t an object instance O such that $O = O_{i,k} = O_{j,k}$, then the sink vertex $O_{i,k}$ from the pair P_i in A_i is also a source vertex for $O_{j,k}$ in A_j.

The next step is to generate the embedding of graphs $G = (IDs, Conn, \mu, v)$ by considering all the pairs involved in the matching actions from the complete input set of plans.

Graph: G = (IDs, Conn, μ, v)

- IDs: finite set of nodes
- Conn ⊆ IDs × IDs denotes a set of edges
- μ: IDs → L_{IDs} denotes a node labelling function
- v: Conn → L_{Conn} denotes an edge labelling function

IDs is the set of vertices of the graph *G*, which are labelled by elements in the vocabulary and are observed from plan traces in the form of object instances. *Conn* is the finite set of edges for each of a particular pair in action instances across all the plan traces available. The order in arguments is important in finding the relationship between object instances. Therefore, an edge *(o1, o2)* is an outgoing edge from *o1* and an incoming edge to *o2*. This edge is added to *Conn* if the objects *o1* and *o2* appear, in this order, in the place of the considered arguments in one action of the plan trace. A node o2 is a neighbour of o1 in

(o1, o2). Being a neighbour does not imply symmetry, i.e. because o2 is a neighbour of o1 does not mean that o1 is essentially a neighbour of o2.

Example to illustrate Step 2: Freecell Domain

To explain the structure of *G* better, following is the continued example of *homefromfreecell* action. For each of the identified pairs of arguments, a directed graph is generated by considering the objects used in the plan traces. In order to exemplify how directed graphs are generated, consider the following instances of the *homefromfreecell* action which are collected from the input set of plan sequences.

Homefromfreecell (club4,club,n4,club3,n3,n0,n1)
homefromfreecell (diamond4,diamond,n4,diamond3,n3,n0,n1)
homefromfreecell (diamond5,diamond,n5,diamond4,n4,n1,n2)
homefromfreecell (spade2,spade,n2,spadea,n1,n0,n1)
homefromfreecell (spade3,spade,n3,spade2,n2,n1,n2)
homefromfreecell (heart3,heart,n3,heart2,n2,n0,n1)
homefromfreecell (heart4,heart,n4,heart3,n3,n1,n2)
homefromfreecell (spade7,spade,n7,spade6,n6,n0,n1)
homefromfreecell (heart6,heart,n6,heart5,n5,n1,n2)
homefromfreecell (spade8,spade,n8,spade7,n7,n2,n3)
homefromfreecell (heart8,heart,n8,heart7,n7,n3,n4)

In order to generate the directed graph for the pair1 arguments, i.e., pair1 (?vcard, ?vhomecard), ASCoL considers all of the objects used as the third and fifth arguments of the *homefromfreecell* action instances. Given the example, all the unique *IDs* include:

IDs = [n4, n3, n5, n2, n1, n7, n6, n8].
The Conn set includes the following unique edges:
Conn = [(n4, n3), (n5, n4), (n2, n1), (n3, n2), (n7, n6), (n6, n5), (n8, n7)].

This is because, in the first instance of *homefromfreecell*, *n4* is before *n3*, and in the second *n5* appears before *n4*. Figure 5.5 represents *IDs* and *Conns* in the form of a graph embedding. This linear order in the third and fifth arguments of the *homefromfreecell* action instances suggests that there is an important one-to-one relationship between the two positions. This linear relationship is implicit in the plan traces and cannot be captured by assembling the transition behaviour of individual type of objects.

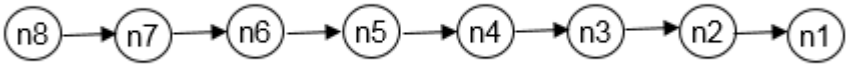


Figure 5.5: A directed graph with a linear structure (Pair 1: type num)

For pair2 (?vcard, ?cells), ASCoL considers all of the objects used as the third and sixth arguments of the *homefromfreecell* action instances. The unique *IDs* and *Conn* set are:

IDs = [n4, n0, n5, n1, n2, n3, n7, n6, n8].
The Conn set includes the following unique edges:
[(n4, n0), (n5, n1), (n2, n0), (n3, n1), (n3, n0), (n4, n1), (n7, n0), (n6, n1), (n8, n2), (n8, n3)].

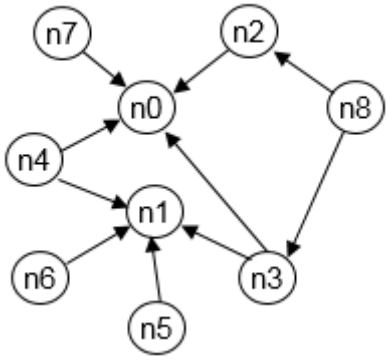


Figure 5.6: Directed acyclic connected graph (pair2: type num)

Figure 5.6 represents this in the form of a graph which is a partially ordered graph. Every node has at least one edge either incoming or outgoing. The structure of this graph does not allow us to find a path for reaching the starting node again and it indicates no meaningful fact to include in the preconditions for pair2. All the remaining pairs of type *num* for the operator have a partial ordering except for pair6 (?cells, ?ncells) which has a linear order like pair1.

To find the ordering in type T *card* for the same operator *homefromfreecell*, there is only one pair of arguments i.e., pair1 (?card, ?homecard). For the purpose of illustration, we shorten card suit names to the initials of the suit along with the digit on the face value of the card i.e. Club=C, Spade=S, Heart=H and Diamond=D. From the available instances of action *homefromfreecell*, all the unique IDs and Connections are included in figure 5.7:

IDs = [C4, C3, D4, D3, D5, S2, Sa, S3, H3, H2, H4, S7, S6, H6, H5, S8, S7, H8, H7].
Conn = [(C4, C3), (D4, D3), (D5, D4), (S2, Sa), (S3, S2), (H3, H2), (H4, H3), (S7, S6), (H6, H5), (S8, S7), (H8, H7)].

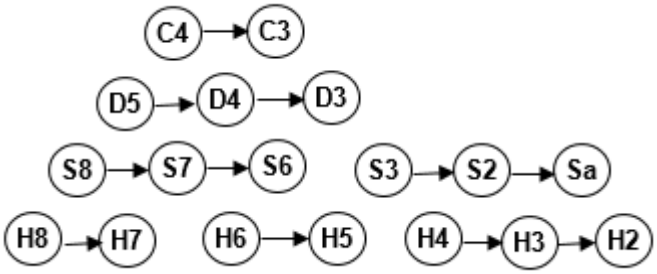


Figure 5.7: A non-fully connected directed graph (Pair 1: type card)

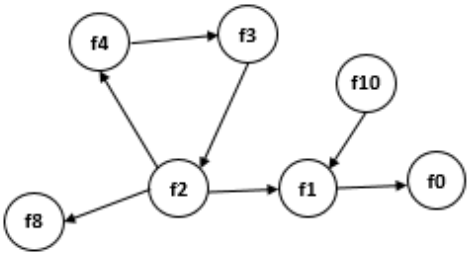


Figure 5.8: Example of a cyclic directed graph

ASCoL draws a graph structure based only on the available number of edges to learn the relationship. In principle, relations can be predicted by only looking at value-pairs of parameters but the difficulty level rises with the increase in the number of parameters of the same type. Using a graph structure makes it easy to analyse such situations where an action has more than two same typed parameters. It also makes the system visually explicit for human users when generating or debugging domains. To determine whether to define the relation, it uses a specific graph structure to decide the correct pair of arguments for producing a static fact.

5.2.3 Step 3: Analysis of the Directed Graphs

Step 1 and 2 of the ASCoL creates a graph for each pair of arguments found. Edges in each graph capture relational information about the arguments of that pair occupying the vertices. In this step, each generated directed graph is investigated to determine the type of relationship between its structure and content. To do this, the graph is analysed by the Order Detector (OD) component of the ASCoL algorithm. The OD will record pairwise static associations between objects. The difficulty lies in distinguishing relevant restrictions from irrelevant ones so that the domain model can be completed with the relevant static conditions (Cresswell et al., 2013).

Graph Outcomes and their Significance

The OD algorithm identifies and analyse the graph order to determine whether there is a specific ordering between the arguments of the corresponding operators. This approach is

suitable because the developed graphs can be analysed independently from the state of the world i.e. from other dynamic object instances and their transitions. For easy understanding, this section first explains the possible structures of the directed graphs and their respective significance and then present the algorithm of OD. The output digraphs from step 2 can have the following different shapes:

Totally ordered graph: Given the directed graph G, for each pair of vertices *o1, o2* we say that *o1* > *o2* if there exists a linear path that connects *o1* to *o2*. Figure 5.5 shows an example of a linear graph. Moreover, *o1* = *o2* is possible only if *o1* and *o2* are the same vertex. This ordered relation between vertices allows OD to check for a total ordering amongst object instances. Total ordering is also assessed by checking the trichotomy property for each pair of vertices (e.g. for natural numbers a and b, either a is greater than b, a equals b, or a is less than b). The trichotomy property states that given two objects o1 and o2, exactly one relation between o1 < o2, o1= o2 and *o1* > *o2* holds. In addition to totality in relationships, the following properties can also hold:

1. Transitivity: *o1* > *o2* and *o2* > *o3* implies *o1* > *o3*.
2. Anti-symmetry: *o1* > *o2* and *o2* > *o1* implies *o1* = *o2*.

The significance of a totally ordered graph in the relation search is that it indicates a strong static relation between the arguments. In particular, the arguments are used as a scale of values, i.e., it is always possible to identify, given two values, an ordering between them. We observed that this kind of linear structure is commonly used in STRIPS for modelling sequences, levels or quantities.

Example from Freecell Domain: in the running example of the *homefromfreecell* operator, pair 1 and pair 6 graphs indicate a linear relationship between the involved arguments, which is represented as (*successor ?vcard ?vhomecard)* and (*successor ?cells ?ncells)* in the benchmark operator. For domains like *Freecell*, such static game rules can be used as a rule of thumb for heuristic search, which require domain-specific information for efficient searching of solutions.

Directed Cyclic Graph: In a cyclic graph, there exists at least one path that can return to its starting vertex i.e. it is a path (o_0, o_1,…, o_k) such that $o_0 = o_k$ and $k \neq 0$. Figure 5.8 shows an example of a cyclic graph structure. Where f2 → f4 → f3 → f2 is a cycle in the graph. The Freecell domain provides no specific example of a cyclic ordered graph as all of the static facts are ordered sequences.

In the case of a Directed Cyclic Graph (DCG), ASCoL tests if G is fully connected or not. It has been observed that, mainly in transport-like domains, this type of graph signifies the presence of an underlying map of strongly connected locations where objects

can move between locations and are allowed to come back from the same route. This argument is supported by proof in the next chapter on evaluation.

Acyclic graph: In Directed Acyclic Graph (DAG) every node has at least one edge either incoming or outgoing. Starting from any vertex, there is no path to return to the starting point.

Finally, if the graph is a partially ordered or Directed Acyclic Graph (DAG), the current version of ASCoL cannot derive any information. Either an acyclic graph can indicate that no static relation is in place between the considered arguments, or there exists some partial ordering between them e.g. the presence of a weakly connected location map (Figure 5.6). Weakly connected location maps mostly exist if plan traces are not generated by random walks (Section 6.7 discuss the impact of differently generated plans).

Algorithm of Graph Order Detector (OD)

After explaining the significance of graphs this section in step 3 explains the formal algorithm for the Order Detector (OD) component of the ASCoL. OD analyzes and recognises the structure and type of generated static graphs. The description of the algorithm is given below in pseudo-code.

Step 3: Order Detection

Input: A set of binary pairs *Conn*, length L

Output: String *Order* that will contain the order result

1. Initialise Boolean *Non-Eq* to True
2. **if** *Conn* is not empty **then**
3. **Iterate** through *Conn*, $i \in 1,\ldots, L$ as follows:
4. **if** (Detects at least one cycle) **then**
5. **if** (reflexivity) **then**
6. *Non-Eq* = False
7. **if** (connectivity) **then**
8. return *Order* = “Connected”
9. **else** return *Order* = “No order”
10. end if
11. **else if** (for each pair *(a, b)* in *Conn*, either $a \leq b$ or $b \leq a$) **then**
12. **if** (reflexivity)
13. *Non-Eq* = False
14. return *Order* = “total order”

15. end else if
16. **else** return *Order* = “partial order”
17. end
18. end if

In **lines 2. through to 10** the algorithm iterates through the *Conn* set of each graph. If the graph entails the properties of a cyclic graph, it further checks for two possible properties i.e. Connectivity and Reflexive Property: -

Connectivity property: ASCoL tests whether G is fully connected, as this is significant for domains that traverse through connected nodes in the form of map of locations (or points) during the execution of actions.

Reflexive property: For each graph G that has the properties of linear or connected cyclic graph, ASCoL checks the presence of self-loops on vertices to identify inequality of action parameters. The significance of such inequality predicates is to force uniqueness in a same-typed parameter pair in an action definition, more specifically in transport actions. This requires the reflexivity (inequality) check between action parameters' values that are instances of the same type.

Example to illustrate reflexive property from Ferry Domain: Briefly, this domain concerns a ferry, which can transport a number of cars from their start to their goal locations. A ferry can *board* the car, *sail* from one location to another accessible location and can *debark* the car. To explain the reflexive property better, two vertices (say locations) in the graph, traversed by either a truck, car, pedestrian or a ferry, must be logically unique as the start and ends of the journey. This is the case of the *sail* operator (figure 5.9) from the *Ferry* domain. In *Ferry* and other domains, Inequality constraint is expressed as a (not-equal ?L1 ?L2) precondition i.e., a precondition that forces the two arguments to be unique.

```
(: action sail
    : parameters (?from ?to - location)
    : precondition (and
                (not-eq ?from ?to) (location ?from)
                (location ?to) (at-ferry ?from))
    : effect (and
                (at-ferry ?to)
                (not (at-ferry ?from))))
```

Figure 5.9: sail operator from Ferry domain

The Freecell domain does not contain any inequality constraint so we have included the *sail* operator from the Ferry domain for illustration purposes.

In **lines 11. through to 16,** if the condition of cyclic graphs fails, the presence of the most significant totally ordered graph property is investigated based on the linearity property. Again, the reflexive (inequality) constraint is also checked on top of all other constraints to assist efficient pruning of the search space.

If the input does not fulfil the conditions for any of the possible orders, OD declares it a partially ordered graph.

Exception for DCG: Experiments show that for some domains, the static facts which are induced from DCG, may become too permissive when some operator contains more than two arguments of the same type. An example of this occurs in the TPP domain (operator *Load*, *Unload* and *Buy* each contain four arguments of the same type). The reason is, in such cases the possible number of pairing combinations of all arguments are high and may give rise to many cyclic graphs. Such cyclic graphs do not always exhibit the correct static relation. Therefore, in the operators with more than two arguments of same type, linear or total order is preferred. This preference produced the desired results in the domain data set which is considered for experimentation.

5.2.4 Conversion to PDDL

The generated graphs have semantics in PDDL, that is, a graph can be translated into a set of PDDL predicates. For a totally ordered graph structure, ASCoL automatically embeds a predicate/relation labelled as *link (parameter1, parameter2)* into the relevant operator preconditions.

In case of the connected cyclic graphs, a static relation *connect (parameter1, parameter2)* is added to the pre-conditions of the operator. Similarly, for Reflexive property, if no self-loops are observed on any vertex, or no combination of two parameters is found equal for the action, implies that an inequality constraint exists. In this case, the constraint/relationship learnt in the form of a predicate *(not-equal ?Param1 ?Param2)* is added to the operator preconditions.

The inequality check might be redundant in the case when other static facts exist in the operator but for the situation of incomplete graphs due to the rare occurrence of a particular action, we prefer to add redundancy than to totally miss some relations.

5.2.5 Discussion

The outcome of the ASCoL is in the graphical form which is learnt from logs of actions (planning traces) and explain the logs in the form of static relations. For example, assuming the action alphabets of A, B, C, D, and E: the input to the algorithm is the number of plans based on some rational arrangement of action alphabets, such as ABAAC, ACBCAE, and ADEEDCDC etc. The system then induces static preconditions in the form of different topologies of graphs such as totally ordered graphs.

ASCoL identifies static knowledge for individual operators of a domain under observation. It does not combine or generate joint/general constraints for different operators or for the overall domain. Besides, the fact that different operators of the domain have their own specific static facts, it might be the case that in certain benchmark domains operators share the same static fact too. For instance, the complete Freecell domain uses four static facts sixteen times (in the ten operators). Each operator uses the same static condition for demonstrating different behaviour.

In terms of utility, a static graph can signify many things and it is more general than specific for some domains like transport domains. Regarding this point, it should be noted that ASCoL can be used without LOCM. In particular, it can be useful for debugging domain models, in order to identify missing static relations or further constrain the search space by pruning useless actions. Apart from its use for the planning and knowledge engineering community, other wider application areas can benefit from the methodology and involve synthesis of constraints or invariants. For instance,

- To detect anomalies in domain design.
- Extraction of maps of locations from past travelling activity.
- To detect increasing or decreasing trend in object levels.
- To learn the static rules of games like solitaires, draft games etc.
- Chapter 7 also covers the application of ASCoL with other techniques. It includes the analyses of planning domain models and generation of domain models corresponding to a number of solitaires.

ASCoL's language for describing static knowledge is PDDL, as the target output in most of the KE systems for AI planning is in de-facto standard PDDL with multiple extensions. PDDL also includes some optional supporting knowledge about a domain such as typing information.

A preliminary version of ASCoL has been presented in (Jilani, Crampton et al. 2014); this version was able to identify inequality constraints only. The present version (Jilani, Crampton et al. 2015) demonstrates ASCoL's ability to find static relations for enhancing domain models automatically acquired by LOCM; this follows from significant

experimental analysis in the next chapter. We demonstrate the scope and scalability of the proposed method on a wide variety of planning problems. Remarkable results have been achieved with regard to the number of static relations in complex, well-known benchmark domains, such as the Freecell domain.

5.3 Implementation

This section presents the implementation details of a prototype system developed and describes the specific details of requirements, design and implementation of the ASCoL system. Several factors together determine the design of the system. Some of them include the expected users of the system, the sources, language and format of the inputs, the quantity and quality of the inputs and the type and format of the expected output. Therefore, it is important to gather and evaluate the requirements of a software system that can best implement the proposed methodology.

As the ASCoL system is the first system of its kind, we compare it with the closest and general domain learning systems for implementation. ASCoL learns only the static aspect of a domain model and it is unique in that it requires only a set of plan traces and object types as its input. Both the quantity and quality of ASCoL's input and its potential output are different compared to the inputs and outputs of a traditional domain learning system. Therefore, the proposed ASCoL's technique requires some specific features apart from general requirements of a domain learning system.

The input in a traditional domain learning system generally includes a set of plan traces as well as the background information such as specification of fluents, intermediate states, initial/goal states or a partial domain model. The input to *ASCoL* is an action training sequence (also called plan traces), where each action is specified as a name followed by a sequence of affected objects as discussed in section 2.2. Input is in a sufficiently general format that it can be created from various sources. Similarly, the ASCoL output is a collection of constraints/static facts extracted from a set of plan traces by matching the action names and types for the enrichment of the domain model. This is different to the output of traditional domain learning systems that usually produce a complete domain model, macro operators, action model with conditional effects or multiple (candidate) domain models.

Flexibility and configurability were other significant requirements from the system implementing the ASCoL Methodology. This is because the effect of some of the concepts and features used by the ASCoL Methodology had to be explored to find the best performing set of features. Similarly, in some domains, ASCoL comes up with additional static facts e.g. in Logistics and Gripper etc. Hence, it was necessary to explore which of

the additionally learnt facts performs better in conjunction with other known facts and features for the specific domain exploited by ASCoL.

In order to implement a system as a component of other domain learning systems, we used the LOCM system. As already discussed in section 3.2 LOCM only produces the dynamic aspect of a domain model and not the static relationships. After identifying the importance and advantage of static relationships in the domain model, we decided to develop a separate method which can work both independently and as an extension of another system. ASCoL is designed to be compatible with LOCM system and this is why it uses the same input plan traces as used by LOCM and embed the output static relations into the domain model induced by LOCM, with the intention to extend and enrich the domain model and enhance the overall automatic domain learning systems.

ASCoL system is developed as a configurable desktop application. It is configurable to select the particular domain and its plan traces produced by certain planners. In order to fulfil the composition requirements of input, a side application is developed to convert the general plan traces' files to the ASCoL compatible single input file of action sequences in Prolog syntax. We have written a parser in the Java language to syntactically analyse and tokenize both the inputs, to make them applicable for operation, and to confirm them to the rules of the developed method.

The application is designed by keeping in mind the potential users of the system. The potential users of the system could be knowledge engineers, domain experts, other domain learning systems and even non-expert users as well. In order to visually aid the experience of the user, an option to view the developed graphs is also provided against each learnt static relation. In the following section on System Design and Development, a layered architecture of the system is illustrated and explained in the form of five functional layers with relevant block diagrams and screenshots.

5.3.1 System Design & Development

The ASCoL system is designed and implemented using standard software engineering procedures. An evaluation framework is also developed for preliminary experimentation to explore the effects and implementation of different concepts. The system is coded using the Java programming language and Eclipse IDE. The ASCoL system has been implemented using a layered architecture with five distinct layers. Each layer is responsible for a certain task. This "separation of concerns" is used to ensure that each layer should work independently of every other layer and can be altered, if required, without affecting the other modules of the system. Information about these layers are given below.

Sequence Generator

This is the first and optional layer in terms of usage. The main purpose of this stage is to convert the input plan traces to ASCoL compatible form in terms of syntax. For the sake of experimentation, different planners especially Metric-FF and LPG generated most of the plan trace test sets used. Each planner has its own syntax for generating a plan trace in its output. However, ASCoL inputs standard Prolog syntax for the input set of plan traces written one after the other in a single file.

In order to avoid manual syntax conversion of the number of plan traces generated by a planner to be used as a test set, an additional software program was developed called Sequence Generator. It allows the user to browse to the directory and folders (in Windows) which contain all the plan traces generated by a particular planner in the form of separate files. Sequence Generator then converts all the files in the directory to a single file, formatted according to the standard Prolog syntax which is accepted by the ASCoL parsing and processing layers. Figure 5.10 shows the working of Sequence Generator and figure 5.11 is a screenshot of Sequence Generator UI. The Sequence Generator is developed using the Java programming language and Eclipse IDE

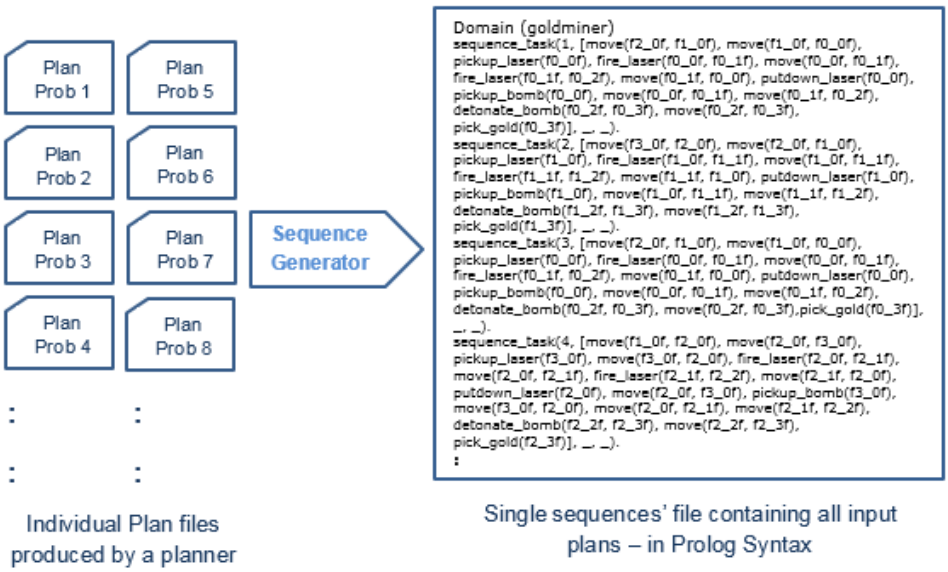


Figure 5.10: Working of Sequence Generator

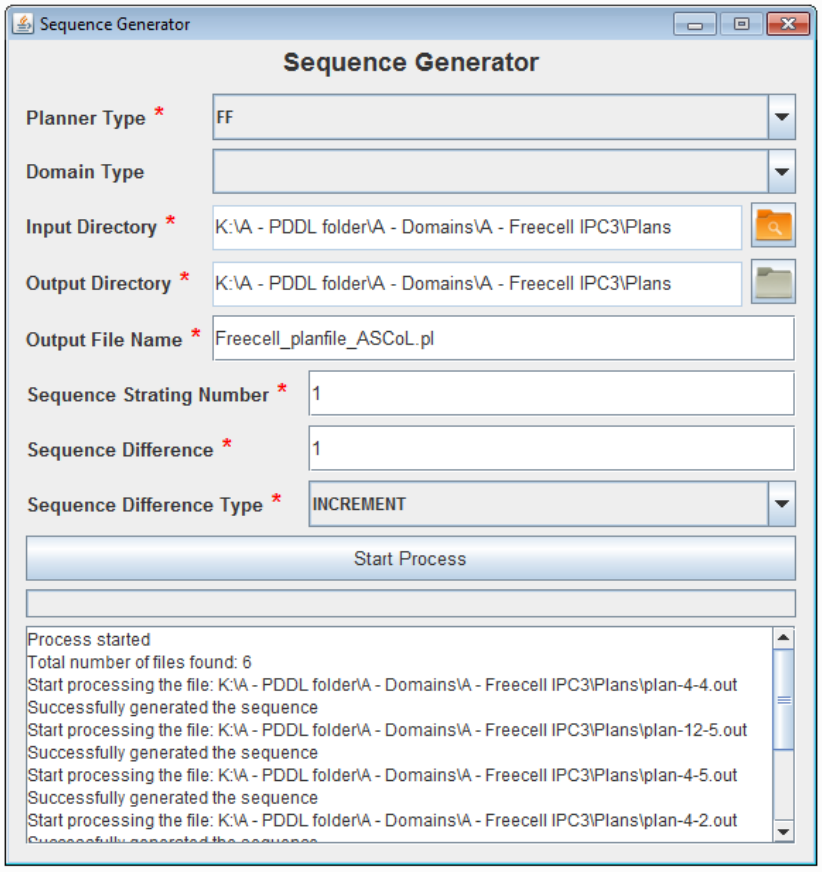


Figure 5.11: Working User Interface of Sequence Generator

Parsing Layer

This layer is responsible for parsing the inputs to the tokenized form that can then be utilised by the algorithm for further analysis. The input partial domain is parsed and type look-up is performed on the operator arguments in order to identify types for each parameter of each action in the plan traces. The input plan traces file (generated by Sequence Generator) is parsed in order to generate the embedding of graphs $G = (IDs, Conn, \mu, \nu)$ by considering all the pairs involved in the matching actions from the complete input set of plans. The graph embedding is a separate process and comes under the processing logic layer. Figure 5.12 represents the parsing process. Parser then passes the components (V, E) of the graph to the processing logic layer in order to run the main algorithm on it.

The rules for parsing the files are embedded into the developed system for now. These parsing rules can be presented in any formal way, such as by writing them in a separate XML file in order to share them for later use and to improve the system development experience in later prototypes.

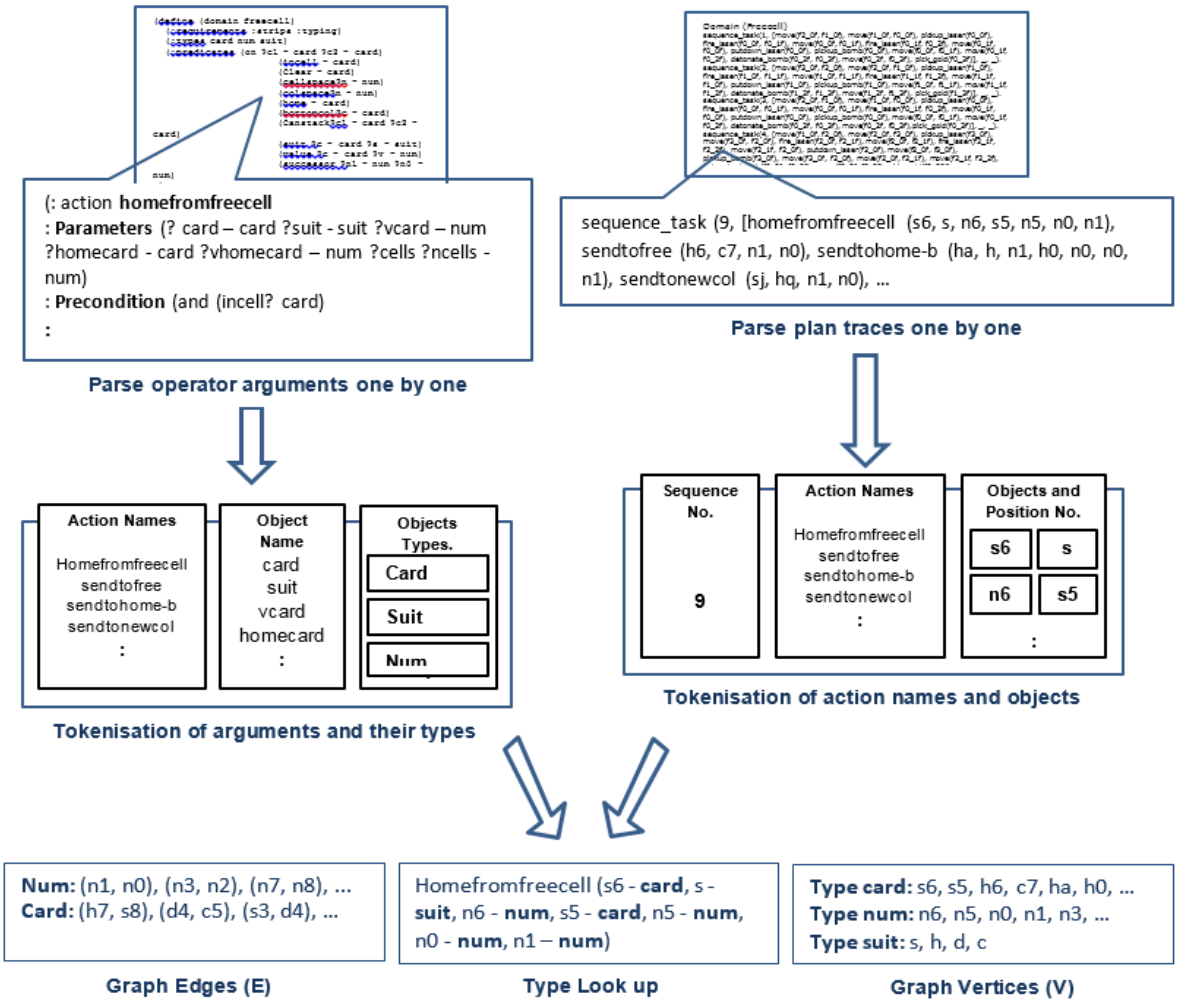


Figure 5.12: ASCoL Parsing Layer

Processing Logic Layer

Implementing the ASCoL methodology, this layer constitutes the core of ASCoL system. This layer (figure 5.13) performs all phases of the ASCoL methodology, starting from the selection of vertices pair to the generation of results. This layer uses input data provided by the parsing layer (after running it through Sequence Generator) and a partial domain model provided by the presentation layer to execute the methodology steps. In order to generate pairs or edges of graph, a mapping is made from the action object instances found in the plan traces to object types found from the domain model. Object instances are classified and

E = Conn, V = IDs from Parsing Layer

position in each

Constraints R to Embedding Layer

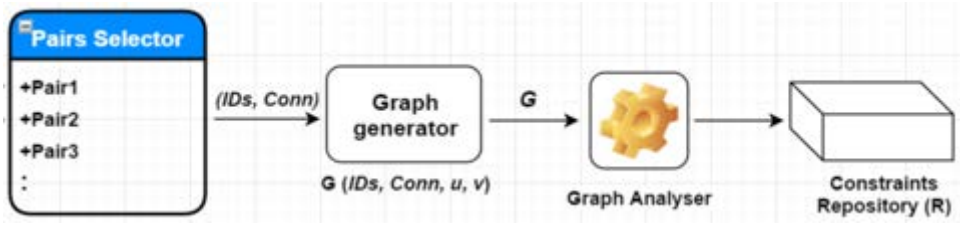


Figure 5.13: ASCoL Processing Logic Layer

Embedding Layer

This layer of the system is responsible for augmenting the input domain model by embedding the results produced by processing logic layer into the correct operator of the domain model to extend and improve it with static knowledge. Each learnt static relation is inserted into the relevant operator's preconditions (figure 5.14).

```
(:action homefromfreecell
        :parameters (?card - card ?suit - suit ?vcard – num ?homecard –
                        card ?vhomecard – num ?cells ?ncells - num)
        :precondition (and (zero_state0)
                        (card_state6 ?card)
                        (card_state0 ?suit)
                        (num_state0 ?vcard)
                        (card_state1 ?homecard ?suit ?vhomecard)
                        (num_state0 ?vhomecard)
                        (num_state0 ?cells)
                        (num_state0 ?ncells)
        :effect (and
                (card_state1 ?card ?suit ?vcard)
                (not (card_state6 ?card))
                (card_state7 ?homecard)
                        (not (card_state1 ?homecard ?suit ?vhomecard)))
)
```

```
(:action homefromfreecell
        :parameters (?card - card ?suit - suit ?vcard – num ?homecard –
                        card ?vhomecard – num ?cells ?ncells - num)
        :precondition (and (zero_state0)
                        (card_state6 ?card)
                        (card_state0 ?suit)
                        (num_state0 ?vcard)
                        (card_state1 ?homecard ?suit ?vhomecard)
                        (num_state0 ?vhomecard)
                        (num_state0 ?cells)
                        (num_state0 ?ncells)
                        (link ?ncells ?cells)
                        (link ?vcard ?vhomecard))
        :effect (and
                (card_state1 ?card ?suit ?vcard)
                (not (card_state6 ?card))
                (card_state7 ?homecard)
                        (not (card_state1 ?homecard ?suit ?vhomecard)))
)
```

Figure 5.14: LOCM operator without (left) and with (right) static relations

Presentation Layer

This layer provides a User Interface (UI) for the system interaction and better user experience. The user can enter the main plan traces file using this layer (or the Sequence Generator) that passes this on to the processing layer. In the evaluation system, this layer is also used to select different configurable options. The Graphical User Interface (GUI) provided by this layer is made up of Java *Swing* components and is designed with the help of the Window Builder plug-in.

The screenshots presented in figure 5.15 – 5.20 show the Graphical User Interface (GUI) for the application. The main user interface displays results in the form of three separate tabs on clicking Process button. The tabs titles comprise of Summary, Parameters

Must Not Be Equal and Orders. The screenshots are taken with the running example from Freecell domain.

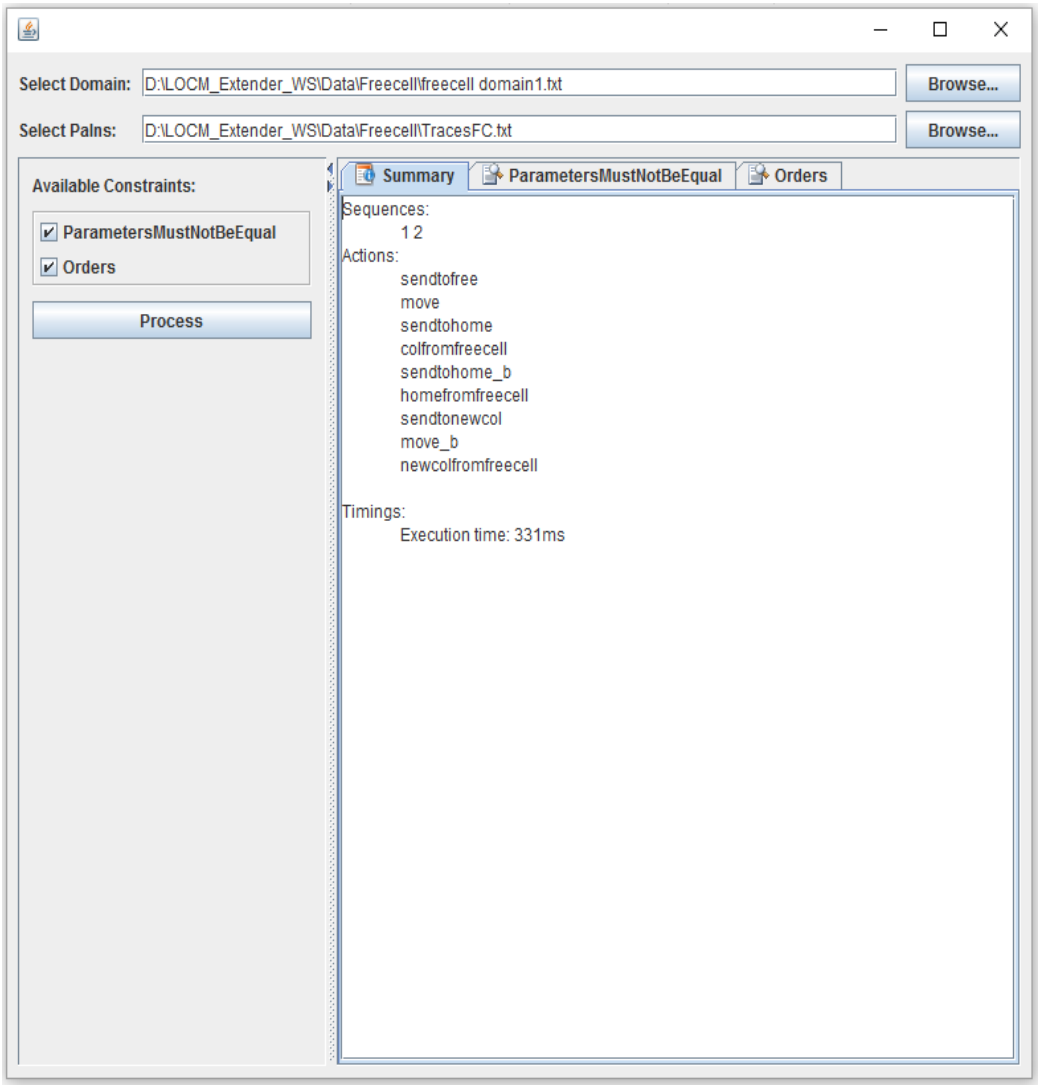


Figure 5.15: Screenshot: Displaying domain summary

Figure 5.15 shows the screenshot after browsing both the inputs (of Freecell in this UI), i.e. Domain file and Plan Traces file. This tab displays domain summary including number of sequences in input plan traces file, number of actions in input domain file and the total time to work out static relation (which is illustrated in next screenshot).

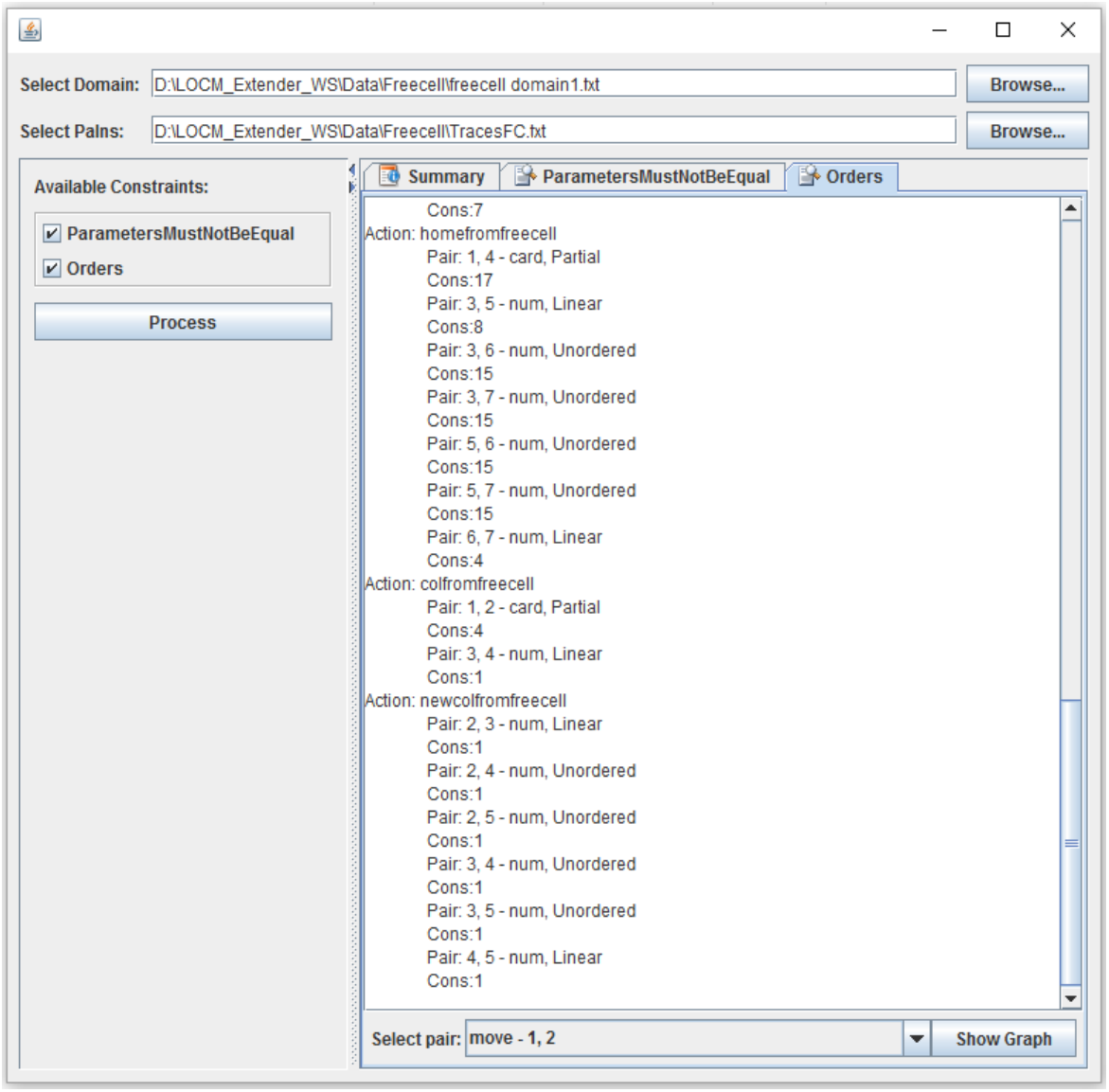


Figure 5.16: Screenshot: Tab showing Orders learnt for a particular actions

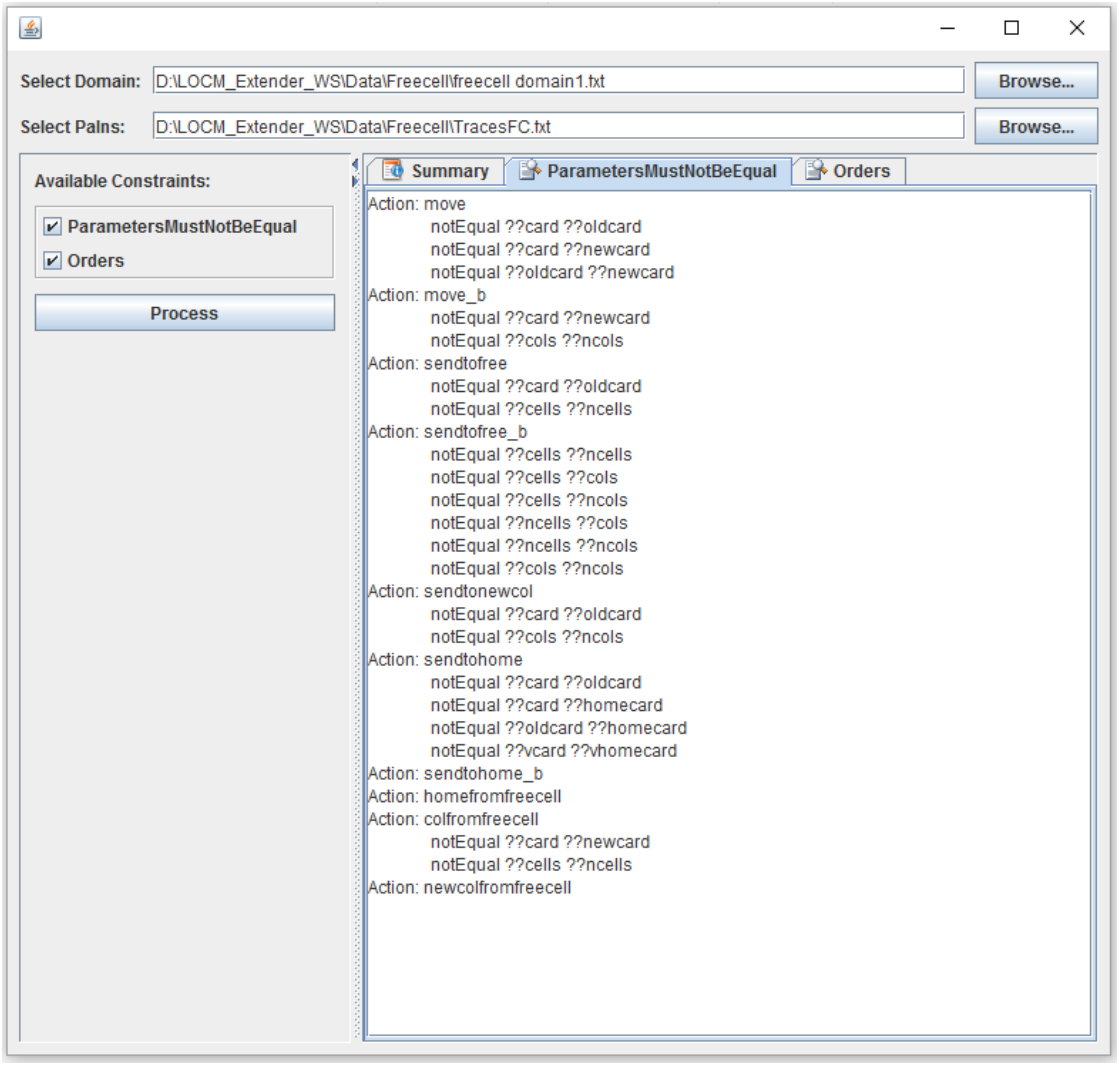


Figure 5.17: Screenshot: Tab for Non-equality constraints

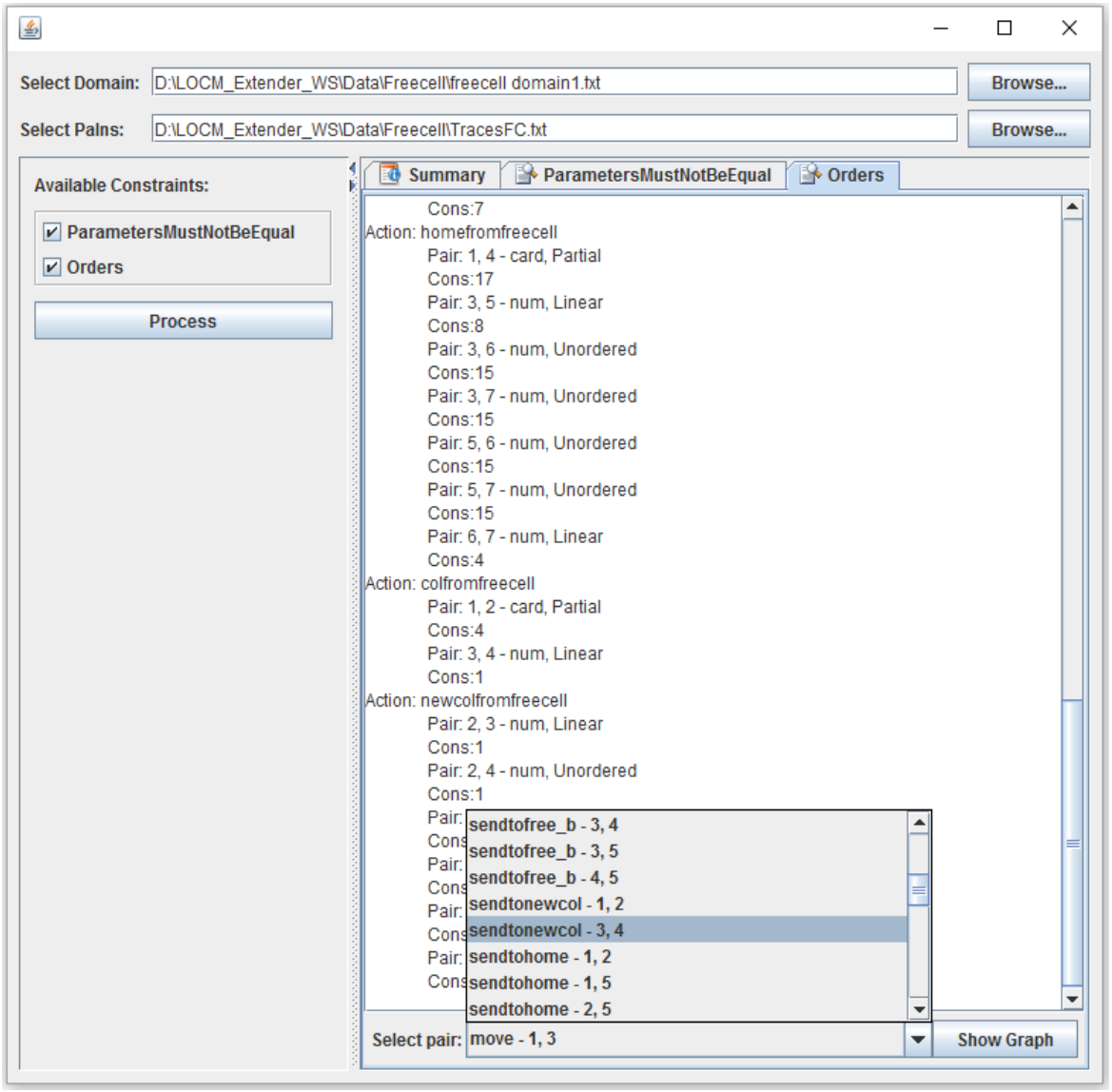


Figure 5.18: Screenshot: Selecting the argument pair

Figure 5.18 shows the screenshot for selecting the argument pair from the drop-down for any operator with static fact to draw the graph. The numbers at the end of each choice represent the parameter positions. Figure 5.19 shows the screenshot of the partially ordered output graph generated by the system for operator move (1, 3), where (1, 3) represents the parameters' positions. Figure 5.20 also shows the screenshot of the linear output graph generated by the system for operator *sendtohome* (4, 6), where (4, 6) represents the parameters' positions.

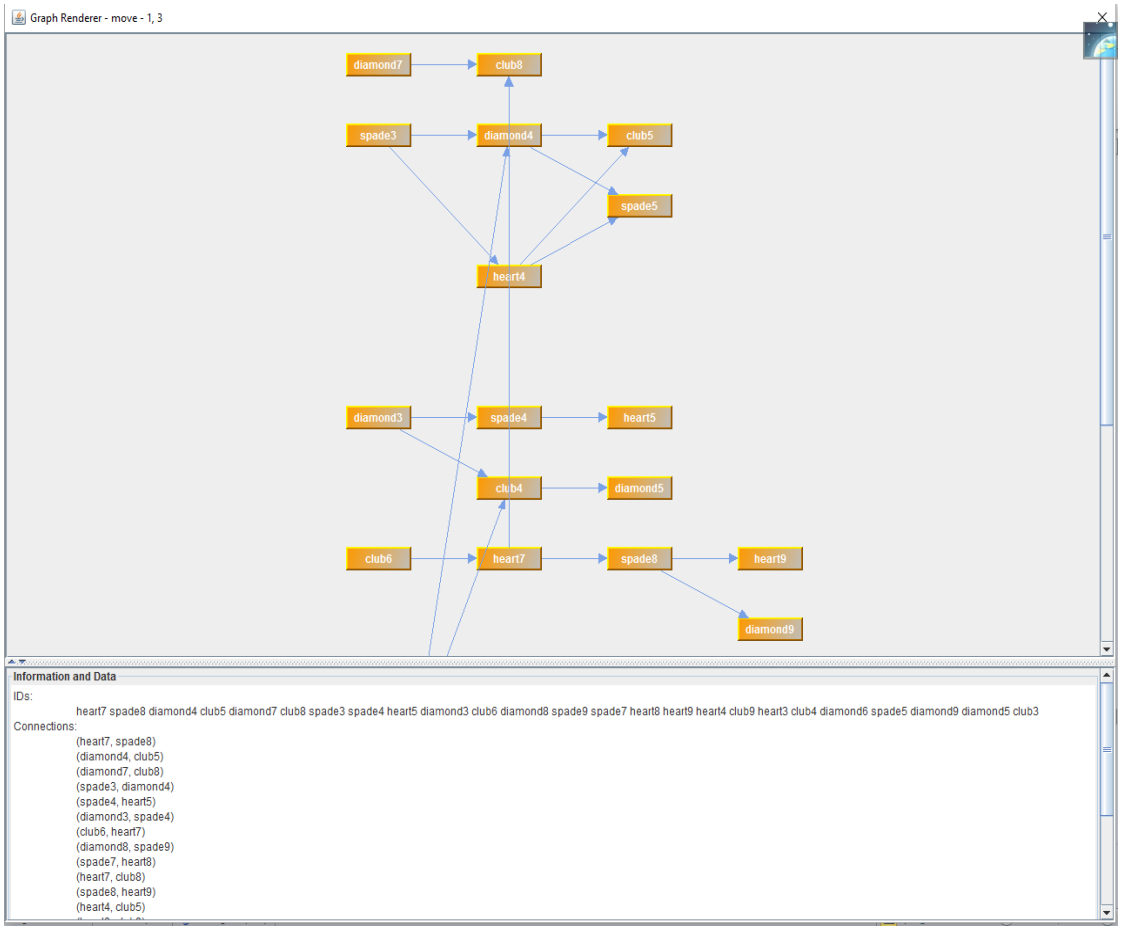


Figure 5.19: Screenshot: Output in graphical form

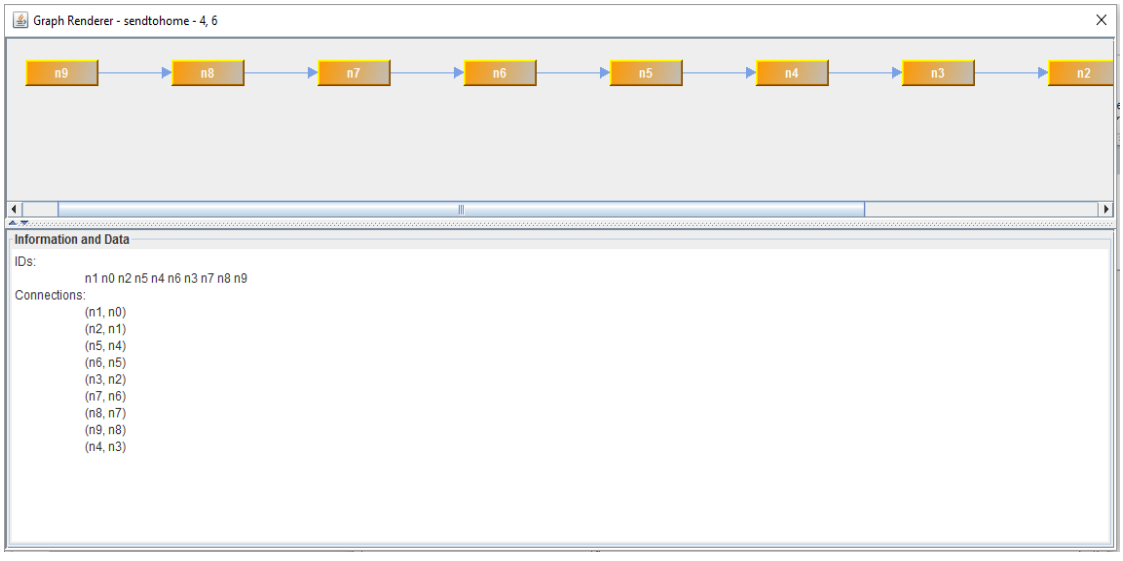


Figure 5.20: Screenshot: Output in graphical form

5.3.2 ASCoL Application Architecture

The ASCoL Static Knowledge Learning Application implements the ASCoL method. It allows the user to perform static knowledge discovery to discover potential static fact(s). The application takes in a user input in Prolog language and provides the relevant static relations and their graphs between variables as output.

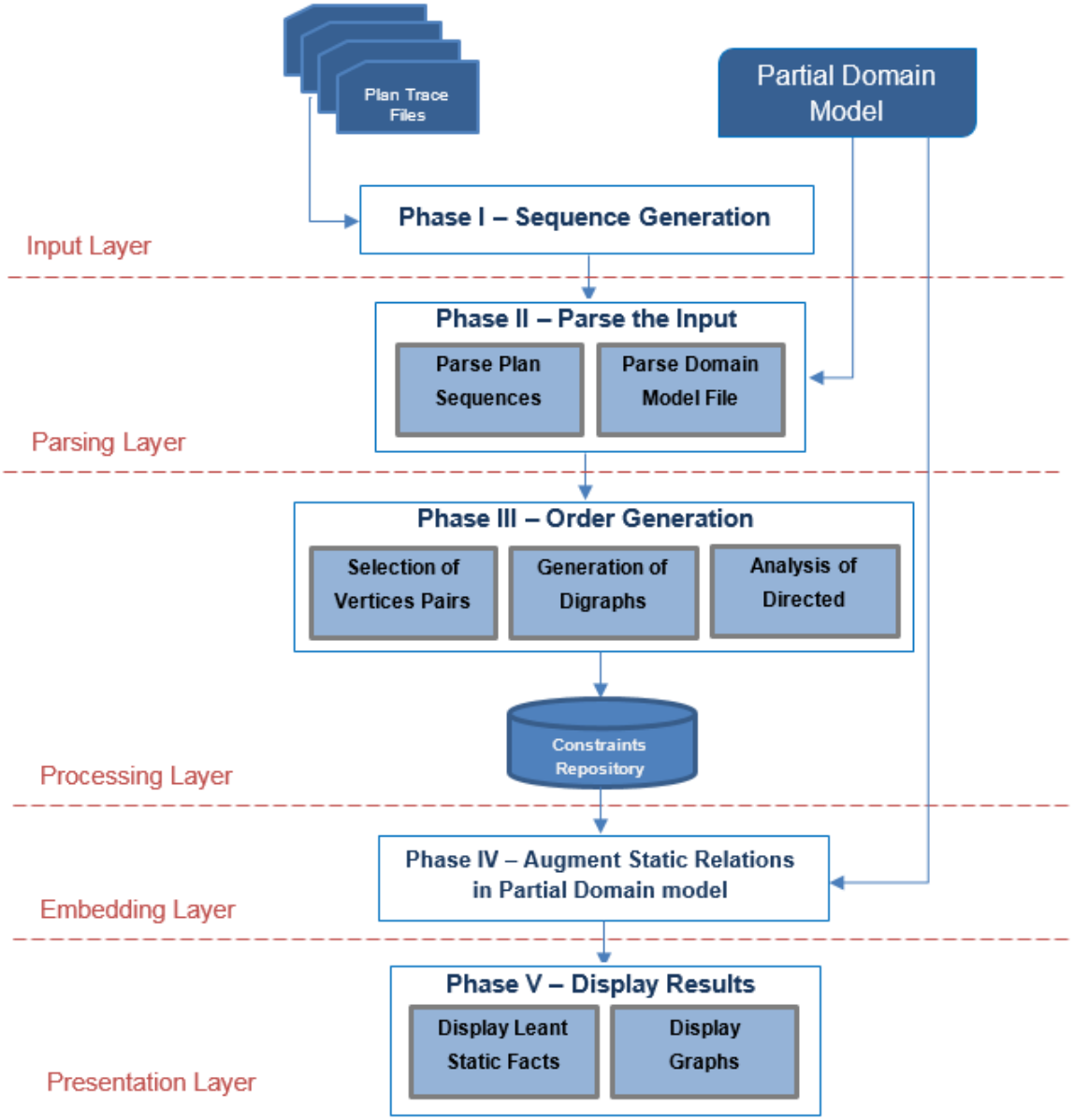


Figure 5.21: Block Diagram depicting ASCoL Application Architecture

When a user input is entered, the application uses a pre-set configuration to generate sequence file and to parse different inputs for particular concepts and methods used in the ASCoL methodology. On start-up, the application executes sequence generation and parsing process in Phase I and II of the ASCoL methodology. This is followed by execution of Phase III and IV, which extract and embed static relations in the partial domain model. At the end of this process, in Phase V the user is presented with results using a UI.

The architecture of the ASCoL Application has been presented in figure 5.21. This diagram also depicts the five different development layers i.e. the Input Layer, Parsing Layer, Processing Logic Layer, Embedding Layer, and Presentation Layer. It also indicates the ASCoL Method Phases I – V. Details of the individual phases have been provided in the previous section.

5.3.3 Testing

Testing is an investigation conducted to verify the quality and performance of a piece of software. General testing strategy is usually to compare the results of a program against already known benchmark. We applied a similar mechanism for testing ASCoL evaluation tool as we found it more suitable due to nature of the system. Hence, a testing strategy was devised for this purpose. The output of ASCoL evaluation tool was compared against known values that had been written by knowledge engineers and domain developers by hand. These values are well-known facts across planning domain and are best suited to verify system output.

The controlled execution of the system is performed to obtained output snippets at different key steps in the process to trace the overall operation and to validate the proposed relation look-up mechanism. We also performed the same steps on different domains manually to produce handcrafted results. The output of the manual steps was compared against the interim output produced by the ASCoL evaluation tool to verify their validity. In order to verify if both system-generated and manually generated results are equivalent, a type of equivalence was introduced between the two domains, one for the known planning benchmarks ($SR_{Benchmark}$) and the other for ASCoL generated static relationships (SR_{ASCoL}). $SR_{Benchmark}$ and SR_{ASCoL} are equivalent if and only if the two directed graphs and their edge labelling function (L_E) for static edges are isomorphic.

5.4 Argument for Extracting Same-Typed Static Relations

After discussing all the important steps and concepts exploited in the previous sections on the core method of the system, this section answers the important question “Why same-typed static facts?”

We use the term Main Static Relations (MSRs) for the binary static relations identified by ASCoL method. As quoted earlier, ASCoL discovers same-typed static relations to augment the LOCM generated domain model. As one factor, it is considered that only same-typed object pairs can reduce the computational time required for identifying relations and avoid the explosion in the size of the search space. In addition, this is due to the fact that, although theoretically, static relations can hold between objects of different types, they mostly arise between same-typed objects.

To justify the last statement, we investigated and looked for Dual-typed Static Relations (DSRs) and Extended Static Relations (ESRs) in our considered data set of fifteen domains.

Dual-typed Static Relation (DSR): is the binary relation that shows static precondition between two different types of objects.

Extended Static Relations (ESR): Given an automatically extracted Main Static Relations (MSRs), there could be other relations in the operator that connect the static feature to other objects of different types and which have a fixed relation to the MSR in the form of a graph too. We call such relations Extended Static Relations.

Let $O = [O_1, \ldots, O_{n(O)}]$ be a set of PDDL operators and P_i be the MSR learnt from the static graph, $G_{Pi} = (IDs, Conn, \mu, v)$ consisting of nodes called IDs and the finite set of directed edges called Conn where μ is a node labelling function and v is an edge labelling function. t_i is the object type of P_i. A type $t_j \neq t_i$ is called a node-fixed type if the following conditions hold:

- There must be an extended binary static relation P_j i.e. $a(P_j) = 2$
- P_j has one argument of type t_i and the other of type t_j.

ESR P_j is the relation that is not dynamic but its arguments are of two different types including the overlapping type (t_i) with MSR P_i. Each ESR can further be extended if it fulfils the above-mentioned conditions for producing further extension in the static graph by considering ESR's node-fixed static type t_j in place of MSR's static type t_i.

After manual analysis, there are only three out of fifteen domains that contain DSRs in their benchmark encoding. This is another motivating factor to look for same-typed static relations. Those three domains are the Freecell, Miconic, and Logistics domain. All the remaining domains only contain same-typed static facts. We now discuss all three of them one by one with respect to what LOCM and ASCoL systems learn.

5.4.1 Freecell Domain

Apart from the same-typed main static relations in benchmark Freecell domain, there are two more ESRs, *value(card, num)* and *suit(card, suit)* that are not same-typed.
For a static relation to be called as ESR, there must exist a Main Static Relation (MSR) which extends to produce that particular ESR. For the two mentioned ESRs in the chosen benchmark example, the MRS is a (*successor ?vcard ?vhomecard).*

```
(: action homefromfreecell
        : parameters (?card - card ?suit - card ?vcard - num ?homecard -
      card ?vhomecard - num ?cells - num ?ncells - num)
        : precondition (and (zero_state0)
                            (card_state6 ? card)
                            (card_state0 ? suit)
                            (num_state0 ?vcard)
                            (card_state1 ?homecard ?suit ?vhomecard)
                            (num_state0 ?vhomecard)
                            (num_state0 ?cells)
                            (num_state0 ?ncells))
      : effect (and (card_state1 ?card ?suit ?vcard)
                    (not (card_state6 ? card))
                    (card_state7 ?homecard)
                    (not (card_state1 ?homecard ?suit ?vhomecard)))
)
```

Figure 5.22: LOCM induced homefromfreecell operator of Freecell domain

Analysis: Upon deep investigation on LOCM induced domain, we noticed that LOCM and LOCM2 already learns ESRs, but (in some domains) misreport some of the ESRs as dynamic relations. For example, in the operator *homefromfreecell* definition (figure 5.22), LOCM learns and combines both the ESRs into the predicate (*card_state1 card1, suit, num1)* as precondition of the action, (*card_state1 card2, suit, num2)* as the positive effect^{+} of the action and (*card_state1 card1, suit, num1)* as the negative effect^{-} of that action. In order to correct this misreporting of static facts as dynamic facts, the LOCM method still needs some work.

5.4.2 Logistics Domain

This benchmark domain does not contain any same-typed MSR while ASCoL discovers an additional MSR in the Drive_Truck operator (figure 5.23) of the domain. The additional static relation (*connect (loc_from, loc_to)*) connects the two locations for the movement of truck across different cities.

```
(:action Drive-Truck
        :parameters (?truck ?loc-from ?loc-to ?city)
        :precondition (and
                    (truck ?truck) (location ?loc-from)
                    (location ?loc-to) (city ?city)(at ?truck ?loc-from)
                    (in-city ?loc-from ?city)(in-city ?loc-to ?city))
         :effect (and
                  (not (at ?truck ?loc-from)) (at ?truck ?loc-to))
)
```

Figure 5.23: Drive_Truck Operator of the benchmark Logistics Domain

Based on the additional MSR discovered by ASCoL, *Connect (loc_from, loc_to)* can be extended to two further already known ESRs:

- (In_city ?loc_from ?city)
- (In_city ?loc_to ?city)

The Logistics domain model induced by LOCM does not learn any of above mentioned static relations. ASCoL only learns same-typed MSRs. ASCoL cannot learn the two ESRs in operator Drive_Truck.

5.4.3 Miconic Domain

All four operators of the Miconic domain contain one static fact each. Operator Up and Down contain *(above ?floor1, ?floor2)* MSR as the precondition of the operator as shown in figure 5.24.

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

Figure 5.24: *Up* operator of the benchmark Miconic Domain

Board and Depart operators contain DSRs: (*origin ?passenger ?floor)* and *(destin ?passenger ?floor)*, respectively as shown in figure 5.25.

```
(:action board
        :parameters (?f - floor ?p - passenger)
        :precondition (and (lift-at ?f) (origin ?p ?f))
        :effect (boarded ?p)
)
(:action depart
        :parameters (?f - floor ?p - passenger)
        :precondition (and (lift-at ?f) (destin ?p ?f)
                (boarded ?p))
        :effect (and (not (boarded ?p))
            (served ?p))
)
```

Figure 5.25: Operator *Board* and *Depart* of benchmark Miconic domain

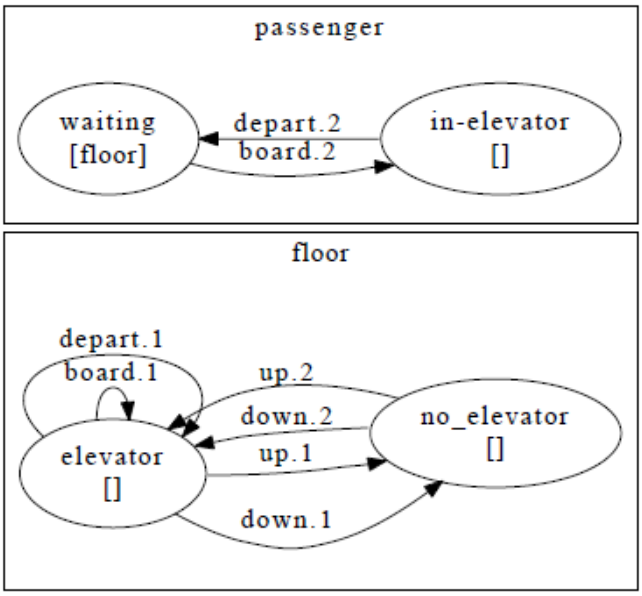


Figure 5.26: The FSMs produced by LOCM that describe the Miconic Domain

Analysis: In the latter case of operators Board and Depart, the DSRs do not fulfil the definition of ESRs and there is no same-typed MSR in the definition of the benchmark operators.

For Miconic Domain, just like Freecell domain above, LOCM cannot learn any of the MSRs ((*above ?floor1, ?floor2)*) while it does learn the DSRs but misreports these as dynamic relations. This can be seen in the action definition learnt by LOCM below (figure 5.27):

```
(:action depart
        :parameters(?F1 - f ?P2 - p)
        :precondition(and
            (zero_state0)
            (f_state0 ?F1)
            (in-elevator ?P2))
        :effect(and
            (waiting ?P2 ?f)
            (not (in-elevator ?P2)))
)
```

Figure 5.27: LOCM induced *Depart* operator definition

We have renamed the DSR as 'waiting' (from 'destin') in the LOCM version of the action definition for easy understanding. LOCM learns this dual-typed predicate but not as a static predicate. The LOCM generated *waiting* predicate is not wrong semantically as can be seen from LOCM generated FSM shown in figure 5.26. In figure 5.26, the first FSM represents the behaviour of passenger type who can board or depart from the elevator, while second FSM is for analysis of an individual floor in the building: on each floor, a passenger can board or depart from the elevator. In fact, hypothesis 3 of parameter association in LOCM methodology does not consider such dual-typed predicates as static facts. Instead, the parameter association step exploits the dynamicity of finite state machines to indicate before and after action execution values of static objects and the results produced are not wrong semantically.

ASCoL only learns the MSRs in Up and Down operators but not the DSRs (*origin ?passenger ?floor)* and *(waiting/destin ?passenger ?floor)* in Board and Depart operators, respectively, which are already learnt by LOCM.

The LOP method (Gregory and Cresswell, 2015) could not learn the MSRs between different floors in Up and Down actions but can detect the origin and destination predicates.

5.4.4 Conclusion

There are a number of problems that the LOCM version 1 is unable to handle which are fixed by LOCM version 2 e.g. the generalisation problem in finite state machine parameter generation which leads to over-simplified domain model generation. This is because some of the LOCM's assumptions are too strong to capture the correct domain description. Complete details and examples are thoroughly described in LOCM 1 and 2 ICAPS papers. LOCM2 weakens some of the assumptions of LOCM by introducing and allowing more than one state machine for representing the different aspects of each object.

The main drawback of both LOCM and LOCM2 is their inability to induce static aspects of the system. This is because LOCM relies on the assumption that all objects go through transitions. This is not true for all the scenarios especially when the domain

contains static aspect too. Apart from this downside, the problem is that the LOCM family of algorithm confuses some DSRs with dynamic relations. LOCM describes this aspect as 'dynamic association to another object' instead of calling it a static relation of two objects. The specific example is quoted in the previous section in Miconic domain where a passenger type is associated to floor type in figure 5.27 through (*waiting ?passenger ?floor)* predicate.

In terms of learning ability of ASCoL, it can only learn same-types static relations which is the most common way to express static aspect in majority of the IPC domains. The missing DSRs by ASCoL are already covered by LOCM in the form of dynamic relations.

Summary of the Chapter

Knowledge acquisition and AI Planning tools and techniques can be combined to implement a domain learning system. It is a challenging task to learn static constraints from plans without taking any hint from a knowledge engineer and planning problem. This is because the static information is not explicitly present in the plan traces which are expressed in terms of action names followed by the objects affected by those actions.

The ASCoL method has been presented in this chapter. ASCoL exploits the information which is implicit in the log of action sequences and demonstrates that this method is a viable approach. The method can be generalised to make constraint acquisition efficient in more complex AI planning domain models through syntactic interpretation of static relations. The implementation of the algorithm also demonstrates that the ASCoL discovers static data by analysing training plans for correlations in the data and can yield significant improvement when used for controlled search in complex domains.

ASCoL can work independently of LOCM as long as the types of objects involved in training plans are provided. The static nature of the relations makes the overall learning problem-independent because the static relations do not change in the state space with the changing problems and therefore does not have to be re-computed for each problem. Moreover, problem descriptions usually include structure related knowledge which usually does not change much between problems from the same domain.

The ASCoL method is implemented as the ASCoL System and an evaluation framework is developed to analyse the effects of different inputs and methods used within the ASCoL technique.

Impressive results have been achieved which prove that the graph-based learning mechanism is sound in terms of inference in the assigned logic and also have graphical representation in order to visualise the outcome.

Chapter 6 - Evaluation

This chapter introduces the ASCoL Evaluation System. It comprises the experiments for empirical evaluation and presents the ASCoL model data set to analyse and evaluate the ASCoL static facts learning system by its application to the learning of static knowledge for a variety of domains. It discusses the learning behaviour of ASCoL for particular types of static facts. It also explains the combined contribution of ASCoL and LOCM for producing useful benchmark domains for planners.

The ASCoL (**A**utomatic **S**tatic **Co**nstraints **L**earner) System has been developed and implemented as a proof of concept for this research. It has been developed with the intention to evaluate the proposed ASCoL technique. In order to extend and enhance the LOCM domain model, ASCoL accepts training plans as an external source of input. ASCoL provides output in the form of relationships that are static in nature with respect to the dynamic part of the model. The output Binary Static Relationships, which we call Main Static Relationships (MSRs) in the text, are generated in the standard PDDL format with a fixed general label 'link' that represents the static relationship or property between the two specific arguments of the operator. In the graphical representation, the first argument of the learnt MSR represents the source of the edge and the second argument represents the sink for the edge. Combining all such edges of the graph on grounding an operator using operator variables, gives rise to a complete static graph, which can further be used for domain analysis, verification and validation purposes.

The aims of the experimental analysis presented in this chapter are to:

(i) Assess the ability of the ASCoL technique in identifying static relationships between arguments, and

(ii) Investigate how different methods of plan generation affect the knowledge extraction processes in ASCoL and compare it with LOCM

Challenges that domain model evaluation faces: As a domain model acts as a knowledge base for generating a solution plan for a specific problem, the correctness of the domain model is a crucial aspect in the overall solution search process. Bensalem et al in (Bensalem et al., 2014) quotes that Validating and verifying domain models is the biggest challenge to the planning community. For the domain design process, among the first of the challenges of KE is that there is no standard and dedicated domain design process to be followed by knowledge engineers. Another major challenge, which need much attention from the research community, is that there is no principle evaluation system to judge the quality of domain models. As the sources of knowledge elicitation and model development

are not mathematical procedures, therefore a domain model cannot be measured on a correctness scale. To attract more research towards automatic domain model acquisition, the ICKEPS'09 (ICKEPS'09, 2009) was devoted to evaluating systems that involved the domain requirements to be captured in an application-oriented syntax for automatic or semi-automatic generation of a domain model.

We present the effectiveness of our idea on benchmarks from past IPCs. For evaluating ASCoL, here we use a batch of fifteen domains. We attempt to evaluate the acquisition of static facts by manually comparing the results generated by our system with the known benchmark domains available and also, when possible, comparing results with the only existing static knowledge learning algorithm in the literature i.e. LOP.

Explaining it briefly, a set of static predicates is retrieved for each domain in the form of graphs. The graphs in the output are ranked according to the order of graph in the operator, which is determined by the graph analyser component (explained in chapter 5). Analyses of the graphs show if the system was able to retrieve any static facts from the set of input training plans. In the outcome, a totally ordered graph indicates a strong static relationship hidden inside plan traces, while cyclic graphs sometimes, and partially ordered graph always, need a manual investigation to check if there is any trend or trace of static facts hidden in the output graph. For example, some graphs appear to be partially ordered and in fact contain strong static relationship in them. This could be due to more than one reason: The input plan traces do not contain enough instances of a particular action, the encoding of the domain, or the scenario where the same typed objects are subdivided into separate sorts and cannot be aligned into a linear total ordered graph.

This chapter's contents have been ordered such that the next section covers Experimental Setup. The following section then introduces the types of identified static facts to better understand the complexity of the learning problem. We developed the categories of static facts, based on the observation that they widely vary in their learning requirements and structural characteristics. A section also provides the details of the ASCoL model data set and details of the evaluation parameters used in the ASCoL technique with their settings for groups of experiments. Another section describes the evaluation metrics and impact of differently generated linear plans. Finally, we discuss some of the most interesting results.

6.1 Experimental Setup

The ASCoL approach has been evaluated using a set of fifteen standard planning benchmark domain models. We tested the system on large sets of linear plans generated from randomly selected problems from both classical (e.g., the blocks world, transportation planning) and artificial domains. The data set domain models are selected

either from past IPCs or from the FF domain collection (FFd). This is suitable because both these platforms are considered authentic due to their high credibility in the planning community. The domains include: PegSolitaire (IPC6), Driverlog (IPC3), Ferry (FFd), Freecell (IPC3), Gold-miner (IPC6), Gripper (IPC7), Hanoi (FFd), Logistics (IPC2), Miconic (IPC2), Mprime (IPC1), TPP (IPC5), Trucks (IPC5), Spanner (IPC6), Storage (IPC5) and ZenoTravel (IPC3). These domains have been selected because they have been manually encoded using different models and modelling strategies. Each of these domains have a variable number of operators which include variable numbers of static facts in the operator preconditions. All domains, except Gripper, Logistics, and Hanoi, exploit static relations in their benchmark encoding. Such domains have been included to test and prove the condition that ASCoL does not include any static relations which prevent the generation of valid plans. Among the considered domains are some with a hierarchical type structure, i.e., all the types defined in the domain are not on the same level (inheritance in types). Storage, Logistics, Driverlog and Trucks are the hierarchical domains considered (indicated by *(H)* in the table below). All domains are written in PDDL 1.2 and 2.1.

For each domain, available problem generators have been used to create the training problems, which are subsequently solved in order to obtain the required linear plan traces.

Domains	**Objects' range in problem instances**
TPP	1-20 goods, 1-8 trucks, 1-8 markets, 1-3 depots and 2-7 levels
Zenotravel	1-5 aircraft, 2-15 persons, 2-3 cities and 7-17 fuel levels
Miconic	3-19 passengers and 30-40 floors
Storage(H)	8-20 store-area, 2-3 hoists, 4-7 crates and 1-2 depots
FreeCell	52 cards, goal conditions vary for 4 suits in home cells between rank 4 - King
Hanoi	same number of objects, i.e. 4 disks and 3 pegs
Logistics(H)	1-2 air-planes, 4-10 locations, 2-5 cities, 2-5 trucks and 6-15 objects
Driverlog(H)	2-5 drivers, 2-5 trucks, 2-15 packages, 4-30 locations
Mprime	5-7 space levels and 2-11 cargo objects
Spanner	3-10 spanners, 10-11 nuts and 4-10 locations
Gripper	4-42 balls in two rooms
Ferry	5-9 locations, 300-600 cars
PegSolitaire	4-30 location objects
Gold-Miner	location grids of 4x4 - 7x7
Trucks(H)	3-9 packages, 3-4 locations, 7-13 time levels and 2-3 areas

There is very little existing work in literature that is close to the functionality that ASCoL performs. After LOP the closest systems that exist in this area of Knowledge Engineering

for P&S are domain-learning systems. Various domain-learning systems have been studied in order to get some inspiration for evaluating our fact learning system. Including the LOP system, most of the domain learning systems manually compare their results with the available planning benchmarks taken from the standard forum for researchers in AI planning and scheduling i.e. ICAPS IPC domain models. The benchmark domain models are considered standard in the community and are written by domain experts and engineers by hand.

The ASCoL evaluation framework is a batch processing application. It can process a batch of plan traces related to any domain. The framework finds the relationships in the provided inputs using the ASCoL method and then evaluates the batch of results by comparing the resulting relations against pre-defined standard planning benchmark domains. Each static fact in the output is considered formally correct if its reference matches with the one present in known benchmark static facts in the domain under study. If none of the static facts references matches the known benchmark static facts, the process is declared a 'failure' and the result is deemed incorrect. Different statistical measures regarding the total number of static facts retrieved and the correctness of each retrieved relationship are also calculated to quantify the analytical findings.

We considered total-ordered plans for experimentation that have been generated by using the Metric-FF planner (Hoffmann, 2003). Metric-FF planner is a forward search planner and enforce a total ordering on actions at all stages of the planning process. This is selected due to its runtime performance (it can very efficiently solve all of the generated problems) and its ability to provide good quality linear plans. In this context, quality is measured in terms of the number of actions. LPG (Gerevini and Serina, 2002) and LPG-td planner (Gerevini et al., 2004) are also used to generate random plans.

Metrics have been created to learn the number of input plans and types of plans (goal and Random walk) required to effectively learn these static facts. The same set of problems have been solved with both a goal-oriented planner and a random walk planner separately and the results observed from knowledge engineering point of view.

ASCoL has been implemented in Java. All the experiments have been performed on a Core 2 Duo/8GB processor.

Evaluation metrics, results of experiments and variations in the data set domain models that are tested on the system are discussed in the upcoming sections. Evaluation results are also logged in the tables for further analysis.

6.2 Types of Static Facts

This section describes the different kinds of static facts that we encountered whilst experimenting with a variety of well-known domain models. We broadly divide static relations into seven types depending upon their structural characteristics and the knowledge they contain:

Type 1: Locations Map: facts used for representing an underlying map of connected locations. Relations of this kind are mentioned as adjacent, link, next, path or connected predicates in the domains including TPP, Zenotravel, Storage, Logistics, Mprime, Spanner, Gripper, Trucks and Gold-miner. These usually appear as linear or connected cyclic digraphs.

Type 2: Resource Quantity: parameter objects include varying levels of goods, fuel, space and time depending on the scenario of the domain. Domains that mention such static facts include Mprime, Trucks, TPP and Zenotravel. These usually appear as linear digraphs.

Type 3: Unordered Array: series of objects but not in any specific order. The best example is the sequence of floors in the Miconic domain, which transports passengers from their origin to the destination floor (using up and down domain actions) through an elevator inside a multi-storied building. This is specified with the encoding of Miconic domain. Clearly, whilst floors have an order in the real world, such order is implicit and not important in the PDDL domain. An unordered array shows no ordering or structure in the problem definition. This appears as just a connected acyclic graph or disconnected graph in the output without any linear order.

Type 4: Ordered Sequence: mentioned as successor, next and link predicates in different domains including specifically Freecell and other card game domains. Here, such static facts allow the sequential arrangement of cards in card stacks among columns, reserve cells and home cells. Ordered sequences tend to produce linear graphs due to the allowed ordering/structure in the problem definition used to generate the plans.

Type 5: Compound Relations: static relations –usually exploited in the encoding of card games– that express two independent static relations in terms of one, i.e., can-stack (card1 card2). The intuition behind this is the conventional stacking rule based on card suit and rank order. This can be decomposed into two separate static facts that can then fulfil the criteria of graph analysis, i.e., can-stack-rank (rcard1 rcard2) and can-stack-suit

(scard1 scard2). Such kinds of compound static facts are not recommended as most of the time it mixes up the domain-independent aspect with the domain-specific aspect in the design phase. Details of this are described in section 6.6.1.

Type 6: Non-equality: the best example for these kinds of static facts are the Ferry domain and other general transportation domains where two location objects of travel (obviously of the same type) should be unique.

Type 7: Dual-Typed Relations: in our observations, static relationships between objects of two unique types are named as dual-typed static relations. LOCM does not treat such relationships as Static in most of the domains and demonstrates learning of such relationships through FSMs. In the experimental data set, three domains Freecell, Logistics and Miconic domain exhibit such relationships. Section 5.5 covered details on this.

Table 6-1: Number and Types of static facts in the benchmark domains

Domains	Type 1	Type 2	Type 3	Type 4	Type 5	Type 6	Type 7	TOTAL
TPP	1	6	x	x	x	x	x	7
Zenotravel	x	4	x	x	x	x	x	4
Miconic	x	x	2	x	x	x	2	4
Storage	5	x	x	x	x	x	x	5
FreeCell	x	x	x	13	3	x	x	16
Hanoi	x	x	x	1	x	x	x	1
Logistics	x	x	x	x	x	x	2	2
Driverlog	2	x	x	x	x	x	x	2
Mprime	1	6	x	x	x	1	x	8
Spanner	1	x	x	x	x	x	x	1
Gripper	x	x	x	x	x	x	x	0
Ferry	x	x	x	x	x	1	x	1
PegSolitaire	1	x	x	x	x	x	x	1
Gold-Miner	3	x	x	x	x	x	x	3
Trucks	1	2	x	x	x	x	x	3
TOTAL	15	18	2	14	3	2	4	58

Table 6-2: Number and Types of static facts in the benchmark domains

Domains	Static Relations
TPP	Type 1 - (connected ?p1 ?p2 - place) Type 2 - (next ?l1 ?l2 - level)
Zenotravel	Type 2 - (next ?l1 ?l2 - level)
Miconic	Type 3 - (above ?f1 ?f2 - floor) Type 7 - (origin ?p - passenger ?f - floor),(destin ?p - passenger ?f - floor)
Storage	Type 1 - (connected ?a1 ?a2 - area)
FreeCell	Type 4 - (successor ?n1 ?n0 - num) Type 5 - (canstack ?c1 ?c2 - card)
Hanoi	Type 4 - (smaller ?x ?y)
Logistics	Type 7 - (in-city ?object ?city)
Driverlog	Type 1 - (link ?x ?y - location), (path ?x ?y - location)
Mprime	Type 1 - (connected ?l1 ?l2 - location) Type 2 - (fuel-neighbor ?f1 ?f2 - level) Type 6 - (not-equal ?l1 ?l2 - location)
Spanner	Type 1 - (link ?x ?y - location)
Gripper	x
Ferry	Type 6 - (not-equal ?l1 ?l2 - location)
PegSolitaire	Type 1 - (in-line ?x ?y ?z - location)
Gold-Miner	Type 1 - (connected ?l1 ?l2 - location)
Trucks	Type 1 - (connected ?l1 ?l2 - location) Type 2 - (next ?t1 ?t2 - time), (le ?t1 ?t2 - time)

Table 6.1 shows the distribution of the types of static relations in each of the considered benchmark domains e.g. in the Freecell domain total of 16 static relations exist out of which 13 are of type 4 and the remaining are of type 5. Table 6.2 further describes the particular static predicates in each of the corresponding domains e.g. In the Freecell domain 16 static facts of Type 4 comprise of predicate (successor ?n1 ?n0 - num) in different operators while remaining 3 static facts of Type 5 are expressed by predicate (canstack ?c1 ?c2 - card).

The following sections provide further details of the evaluation metrics used the ASCoL model dataset domains and source adapted for gathering the sample plan traces used by ASCoL System. Finally, the results of experimentations performed are included.

6.3 Evaluation Metrics

The main part of the domain model is the representation of action schema that planning engines use and reason with. Static relations are among the key components of an operator and their correctness is an important factor in the overall correctness of the plan. It is essential for modelling correct action execution as static predicates are restrictions on the valid groundings of actions which help prune the search space in a planning phase. Missing static knowledge can cause missing preconditions in the domain model.

No standards evaluation metrics exist for this type of research yet. Here we introduce different metrics to evaluate the effectiveness and correctness of ASCoL system results. A learnt predicate is accepted as correct if it matches with the operational semantics of the known benchmark. The judgement of learnt static relation validity is performed manually by comparing it with the benchmark domains. The usage of additionally learnt static relations is evaluated by embedding the relations in the benchmark domain model and then dynamically testing it by running a planner to generate plans.

6.3.1 Accuracy

Accuracy is the measure of the familiarity of a measured value to a standard or acknowledged value. In the ASCoL Evaluation system, the Accuracy measure is defined as the proportion of correctly retrieved results amongst the total number of relationships present in the benchmark domain. This can be presented as follows:

$$\boldsymbol{Accuracy} = \frac{\boldsymbol{Number\ of\ \ correctly\ retrieved\ relations}}{\boldsymbol{Total\ number\ of\ relations\ in\ benchmark}}$$

The total number of relations varies in each of the data set domains. It depends on the scenario captured in the domain and on the way benchmarks are encoded. For instance, Freecell domain contains sixteen instances of static facts in ten different operators while Ferry domain contains one static relation in three unique operators.

In the calculation of accuracy, we do not consider additional relations (which are not present in the benchmark) learnt by ASCoL in some domains. We first calculate the percentage accuracy per domain and percentage accuracy based on types of static facts in each domain in order to determine the accuracy of the system in learning particular types of static facts. After explaining the remaining evaluation metrics and some peculiar domains, experimental results are presented in section 6.5 in the form of tables and figures.

We also evaluate the ASCoL and LOCM systems based on the types of input plan traces i.e. either the plan traces are goal-oriented or random-walk. We also evaluate how accurate overall the ASCoL system is in calculating static facts and show the trends discussed above in the form of graphs.

6.3.2 Precision

In the field of pattern recognition precision is the fraction of relevant values among the retrieved values. The ASCoL Evaluation System defines the Precision measure as the proportion of correct results amongst the total number of results retrieved by the system for a particular domain. This can be expressed as follows:

$$\boldsymbol{Precision} = \frac{\boldsymbol{Number\ of\ correctly\ learnt\ relations}}{\boldsymbol{Total\ number\ of\ learnt\ relations}}$$

The effectiveness of each of the learnt static relations is validated manually by experimenting and comparing the results to ensure that it fulfils the function of the known static fact in the benchmark domain. An important point here is that the additionally learnt relations, even if found semantically correct, or proved that they carry no-effect, reduces the precision of the system for the particular domain. Experimental results are presented in upcoming section 6.5.

The definitions of both accuracy and precision are adapted according to the developed evaluation framework, where computing accuracy means comparing the total number of retrieved facts with the total number of facts present in the benchmark domain. Precision describes how close semantically the discovered facts to the benchmarks are. It describes the correct facts out of all the learnt facts.

6.3.3 Statistical Binary Classification

Considering a classification terminology for test statistics and with regard to our measure of correction, we can divide the relations identified by ASCoL into four classes: true positive, true negative, false positive and false negative.

True positive: A true positive outcome is one that detects the fact when the fact is actually present.

True negative: A true negative result is one that does not detect the fact when the fact is absent. This indicates that no relationship is identified between two objects when there is no relationship used in the model exploited as ground truth.

False positive: A false positive result is the one that detects the fact when the fact is absent. In some domains, ASCoL infers one or two additional relations that are not included in the original domain model.

False negative: A false negative result is the one that does not detect the fact when the fact is present. Experimental results are presented in section 6.5 in the form of table.

6.4 Interesting/Peculiar Models

In this section, the term dataset is used by which we mean the collection of data that contains individual domain models to produce the input training data for our system and to which ASCoL's outcome is compared. The domain models are written in PDDL 1.2 or 2.1 format. Although there are no existing standards to adopt as a dataset in order to validate the outcome of facts learning method, most of the domain learning techniques (e.g. LOCM, ARMS, Opmaker, RIMS etc) use datasets taken from the domains section of the past IPCs. Various researchers and domain experts manually write IPC domains. These domains cover various scenarios, in order to validate the planning systems. As these are publically available and we consider them adequate in terms of needed characteristics, we chose our experimental dataset domains from both IPC and FF domain collection to validate the developed technique.

This section briefly explains the important and more complex domains of all the targeted fifteen dataset domains we have used to evaluate the developed technique. We aim to discuss only the static relationships in each domain and include those operator definitions that contain static facts in them. Freecell domain has already been explained in detail in section 2.5.

6.4.1 TPP Domain

Many domains contain operators with preconditions to measure the capacity or the level of objects. TPP (Travelling Purchase Problem), a known observation of the Travelling Salesman Problem, is an example of one of the most complex domains for the evaluation of the ASCoL algorithm. It is a generalisation of the Travelling Salesman Problem and we have used a propositional version of the original TPP domain. The scenario contains a set of Markets, Depots and Goods. Trucks are available for transportation of goods from Markets to Depots. Each market sells a limited quantity of each type of goods. Regardless of which Market is used to purchase the Goods, all Goods have the same price. The quantity of goods is represented as different discrete Levels. Trucks Load and Unload the Goods from Markets to Depots.

In the propositional TPP, out of a total of four operators, there are seven static relations to be learned which indicate the next level to load, unload or buy goods along

with a map of indirectly connected Markets and Depots. Three operators contain seven arguments each. Out of these seven, four arguments are of the same type (level). Following are the three complex operators of the domain with arguments. Argument names indicate the corresponding type.

Buy (goods truck market level1 level2 level3 level4)
Unload (goods truck depot level1 level2 level3 level4)
Load (truck goods market level1 level2 level3 level4)

The complexity of the domain increases with the increase in number of discrete levels of fuel because this leads to the increase in the number of ordered pairs made for each operator. The following figures (6.1 – 6.3) shows the three operators with two static facts next (level2 level1) next (level4 level3) in each:

```
(:action buy
        :parameters (?t - truck ?g - goods ?m - market ?l1 ?l2 ?l3 ?l4 - level)
        :precondition (and (at ?t ?m)
                        (on-sale ?g ?m ?l2) (ready-to-load ?g ?m ?l3)
                        (next ?l2 ?l1) (next ?l4 ?l3))
         :effect (and (on-sale ?g ?m ?l1)
                 (not (on-sale ?g ?m ?l2))(ready-to-load ?g ?m ?l4)
                 (not (ready-to-load ?g ?m ?l3)))
)
```

Figure 6.1: Operator *buy* from the benchmark TPP domain

```
(:action unload
        :parameters (?g - goods ?t - truck ?d - depot ?l1 ?l2 ?l3 ?l4 - level)
        :precondition (and (at ?t ?d)
                           (loaded ?g ?t ?l2)(stored ?g ?l3)
                           (next ?l2 ?l1) (next ?l4 ?l3))
        :effect (and (loaded ?g ?t ?l1)
                   (not (loaded ?g ?t ?l2))(stored ?g ?l4)
                   (not (stored ?g ?l3)))
)
```

Figure 6.2: Operator *unload* from the benchmark TPP domain

```
(:action load
        :parameters (?g - goods ?t - truck ?m - market ?l1 ?l2 ?l3 ?l4 - level)
         :precondition (and (at ?t ?m)
                        (loaded ?g ?t ?l3)(ready-to-load ?g ?m ?l2)
                        (next ?l2 ?l1) (next ?l4 ?l3))
        :effect (and (loaded ?g ?t ?l4)
                     (not (loaded ?g ?t ?l3)) (ready-to-load ?g ?m ?l1)
                     (not (ready-to-load ?g ?m ?l2)))
)
```

Figure 6.3: Operator *load* from the benchmark TPP domain

Along with propositional version of TPP domain, available random generators are used to create training problem instances and we have made use of problem instances from IPC5

which are subsequently solved in order to obtain the required plan traces. The considered TPP problems vary between 1-20 goods objects, 1-8 trucks, 1-8 markets, 1-3 depots and 2-7 level objects.

6.4.2 Zenotravel Domain

Just like TPP domain's static preconditions, Zenotravel domain also contain three out of five operators with preconditions to measure the capacity or the level of objects. It has four static facts, all of type 2 (Resource Quantity). It is another example of a complex domain for the evaluation of the ASCoL algorithm. In the domain scenario, an aircraft which can fly at two different speeds can Board and Debark persons in different cities. The Aircraft consumes fuel during flight and the quantity of consumption is dependent on the speed of flight. The quantity of fuel is represented as different discrete Levels. Zoom consumes more fuel then Fly. Refuel operator performs its actions by keeping a record of three unique levels of fuel. The level of fuel represented by (next ?level1 ?level2) precondition is the main static aspect of the domain which is necessary during Fly, Zoom and Refuel actions execution (figure 6.4 - 6.6). Zenotravel benchmark domain does not include the static precondition for connecting the two cities c1 and c2.

```
(:action fly
        :parameters (?a - aircraft ?c1 ?c2 - city ?l1 ?l2 - flevel)
        :precondition (and (at ?a ?c1)
                           (fuel-level ?a ?l1)
                           (next ?l2 ?l1))
        :effect (and (not (at ?a ?c1))
                     (at ?a ?c2)
                     (not (fuel-level ?a ?l1))
                     (fuel-level ?a ?l2))
)
```

Figure 6.4: Operator *Fly* from Zenotravel Benchmark Domain

```
(:action zoom
       :parameters (?a - aircraft ?c1 ?c2 - city ?l1 ?l2 ?l3 - flevel)
       :precondition (and (at ?a ?c1)
                          (fuel-level ?a ?l1)
                          (next ?l2 ?l1)
                          (next ?l3 ?l2))
       :effect (and (not (at ?a ?c1))
                    (at ?a ?c2)
                    (not (fuel-level ?a ?l1))
                    (fuel-level ?a ?l3))
)
```

Figure 6.5: Operator *Zoom* from Zenotravel Benchmark Domain

```
(:action refuel
        :parameters (?a - aircraft ?c - city ?l - flevel ?l1 - flevel)
        :precondition (and (fuel-level ?a ?l)
                           (next ?l ?l1)
                           (at ?a ?c))
 :effect (and (fuel-level ?a ?l1) (not (fuel-level ?a ?l)))
)
```

Figure 6.6: Operator *Refuel* from Zenotravel Benchmark Domain

The discrete Levels of fuel usage demands less expressive power of the planners that use the model (Long, 2003). This propositional version of the domain is enough to test the validity of our method. The Zenotravel problems used in order to generate plans have objects ranging between 1-5 aircraft, 2-15 persons, 2-3 cities and 7-17 fuel levels.

6.4.3 Mprime Domain

Mprime domain was first used in AIPS 1998 competition. This domain has a variety of static preconditions based on types of static facts we have defined in section 6.2. It contains one static relation of type 1 (Location Map) which is (conn ?loc1 ?loc2) in the Move operator, six relation of type 2 (Resource Quantity) which are (space-neighbour ?s1 ?s2) and (fuel-neighbour ?f1 ?f2) in the Unload, Load and Donate operators, and a relation of type 6 (Non-equality) which is (not-equal ?l1 ?l2) in the operator Donate. It is a logistics based domain where limited space capacity trucks transport cargo to different locations. Moving the truck from one location to another consumes an amount of fuel. Fuel decreases by one unit as the truck travels from one location to the next. Having more than one fuel unit at a location, a truck can donate a fuel unit to a different location. The following figures (6.7 – 6.10) include the encoding of important operators with static facts in them. Discovering static preconditions for Operator Donate is quite challenging as it contains five arguments of same type.

```
(:action move
     :parameters (?v - vehicle ?l1 ?l2 - location ?f1 ?f2 - fuel)
     :precondition (and (at ?v ?l1)
                        (conn ?l1 ?l2)
                        (has-fuel ?l1 ?f1)
                        (fuel-neighbor ?f2 ?f1))
     :effect (and (not (at ?v ?l1))
                  (at ?v ?l2)
                  (not (has-fuel ?l1 ?f1))
                  (has-fuel ?l1 ?f2))
)
```

Figure 6.7: Operator *Move* of Mprime Benchmark Domain

```
(:action load
      :parameters (?c - cargo ?v - vehicle ?l - location ?s1 ?s2 - space)
      :precondition (and (at ?c ?l)
                         (at ?v ?l)
                         (has-space ?v ?s1)
                         (space-neighbor ?s2 ?s1))
      :effect (and (not (at ?c ?l))
                   (in ?c ?v)
                   (not (has-space ?v ?s1))
                   (has-space ?v ?s2))
)
```

Figure 6.8: Operator *Load* of Mprime Benchmark Domain

```
(:action unload
      :parameters (?c - cargo ?v - vehicle ?l - location ?s1 ?s2 - space)
      :precondition (and (in ?c ?v)
                         (at ?v ?l)
                         (has-space ?v ?s1)
                         (space-neighbor ?s1 ?s2))
   :effect (and (not (in ?c ?v))
                (at ?c ?l)
                (not (has-space ?v ?s1))
                (has-space ?v ?s2))
)
```

Figure 6.9: Operator *Unload* of Mprime Benchmark Domain

```
    (:action donate
          :parameters (?l1 ?l2 - location ?f11 ?f12 ?f13 ?f21 ?f22 - fuel)
          :precondition (and (not-equal ?l1 ?l2)
                             (has-fuel ?l1 ?f11)
                             (fuel-neighbor ?f12 ?f11)
                             (fuel-neighbor ?f13 ?f12)
                             (has-fuel ?l2 ?f21)
                             (fuel-neighbor ?f21 ?f22))
       :effect (and (not (has-fuel ?l1 ?f11))
                    (has-fuel ?l1 ?f12)
                    (not (has-fuel ?l2 ?f21))
                    (has-fuel ?l2 ?f22))
    )
```

Figure 6.10: Operator *Donate* of Mprime Benchmark Domain

The level of fuel represented by (fuel-neighbour ?f1 ?f2) precondition is the main static aspect of the domain which is necessary during Move and Donate action execution. The Mprime problems used in order to generate plans have objects ranging between 5-7 space levels and 2-11 cargo objects.

6.5 Learning Static Relations Using ASCoL

In the previous sections, we described the main evaluation framework adopted along with the experimental setup and the domains used to carry out experiments. It also described different types of static facts we discovered from different domains and the metrics to

evaluate system results. This section focuses on the capacity of ASCoL in identifying static relations unrecognised by LOCM. Here we consider – for each domain – the best set of plan traces, in order to maximise the number of identified relations.

Table 6.3 shows the results of the experimental analysis. For each original domain, the number of operators (# Operators), and the total number of static relations (# SR) are presented. ASCoL results are shown in terms of the number of identified static relationships (ASCoL SR) and number of additional static relations learnt (Additional SR) which were not included in the original hand-coded benchmark domain model. The last column in the table shows the CPU-time in milliseconds for finding static facts for each domain. Empirical analysis shows that ASCoL is able to identify all of the static relations of the considered domains except a few static facts in domains including Miconic, Freecell, PegSolitaire and Gold-Miner. All four domains with missing static facts are discussed in detail in upcoming section 6.6.

Moreover, in some domains including Zenotravel, Logistics, Gripper and Ferry, ASCoL is providing additional static relations, which are not included in the benchmark domain models. After generating plans by using domains with additionally learnt static facts, we found that these do not reduce the solvability of problems, but can reduce the size of the search space by pruning useless instantiations of operators in some domains. Later, we also compare our results with results mentioned by LOP system for a few prominent domains.

Table 6-3: Overall results of ASCoL on considered domains

Domain	#Operators	#SR	ASCoL SR	Additional SR	CPU-Time
TPP	4	7	7	0	171
Zenotravel	5	4	6	2	109
Miconic	4	4	2	0	143
Storage	5	5	5	0	175
FreeCell	10	16	13	0	320
Hanoi	1	1	1	0	140
Logistics	6	0	1	1	98
Driverlog	6	2	2	0	35
Mprime	4	8	8	0	190
Spanner	3	1	1	0	144
Gripper	3	0	1	1	10
Ferry	3	1	2	1	130
PegSolitaire	1	1	0	0	00
Gold-Miner	7	3	1	0	128
Trucks	4	3	3	0	158

Table 6.4 shows, for each considered domain, the percentages of true positive and negative, false positive and negative relations (defined in section 6.3.3) identified by ASCoL.

True positive: In terms of ASCoL results, these are correctly identified static relations. Relations identified are always static relations which are included in the original domain models.

True negative: ASCoL did not identify a static relation between arguments that are connected by a dynamic relation in any of the considered domains.

False positive: In some domains, ASCoL infers one or two additional relations that are not included in the original domain model. From a Knowledge Engineering point of view and after testing the solvability of problems with new domain models, we can say that such learnt preconditions do not reduce the solvability of problems.

False negative: In Freecell and Gold-miner domains, ASCoL does not identify all of the static relations.

In Table 6.4, where the TP results are shown to be 100.0 indicates that ASCoL learnt 100% of the facts while 0.0 shows no learning at all. In TN column, 100.0 indicates that ASCoL did not find any static relation that is actually connected by a dynamic relation. Column FP shows 0.0 in most of the domains which indicates that it does not discover any percentage of additional relations that are not present in the real domain. IN FN column, 0.0 indicates that no percentage of static relations is left that is not learnt by the ASCoL system.

The ability of ASCoL to correctly identify static relations, that should be included as preconditions of specific operators, depends on the number of times the particular operator appears in the provided plan traces. The higher the number of instances of the operator in the plan, the higher the probability that ASCoL will correctly identify all the static relations.

Table 6-4: Results of Statistical Binary Classification

Domains	TP	TN	FP	FN
TPP	100.0	100.0	0.0	0.0
Zenotravel	100.0	100.0	33.3	0.0
Miconic	50.0	100.0	0.0	50.0
Storage	100.0	100.0	0.0	0.0
FreeCell	81.3	100.0	0.0	18.7
Hanoi	100.0	100.0	0.0	0.0

Logistics	100.0	100.0	100.0	0.0
Driverlog	100.0	100.0	0.0	0.0
Mprime	100.0	100.0	0.0	0.0
Spanner	100.0	100.0	0.0	0.0
Gripper	100.0	100.0	100.0	0.0
Ferry	100.0	100.0	50.0	0.0
PegSolitaire	00.0	100.0	0.0	100.0
Gold-Miner	33.0	100.0	0.0	66.6
Trucks	100.0	100.0	0.0	0.0

Table 6-5: The Accuracy and Precision of the ASCoL system per domain

Domains	Accuracy (%)	Precision (%)
TPP	7/7 =100.0	7/7 = 100.0
Zenotravel	4/4 = 100.0	4/6 = 66.6
Miconic	2/4 = 50.0	2/2 = 100.0
Storage	5/5 = 100.0	5/5 = 100.0
FreeCell	13/16 = 81.25	13/13 = 100.0
Hanoi	1/1 = 100.0	1/1 = 100.0
Logistics	---	---
Driverlog	2/2 = 100.0	2/2 = 100.0
Mprime	8/8 = 100.0	8/8 = 100.0
Spanner	1/1 = 100.0	1/1 = 100.0
Gripper	---	---
Ferry	1/1 = 100.0	1/2 = 50.0
PegSolitaire	00.0	00.0
Gold-Miner	1/3 = 33.3	1/1 = 100.0
Trucks	3/3 = 100.0	3/3 = 100.0

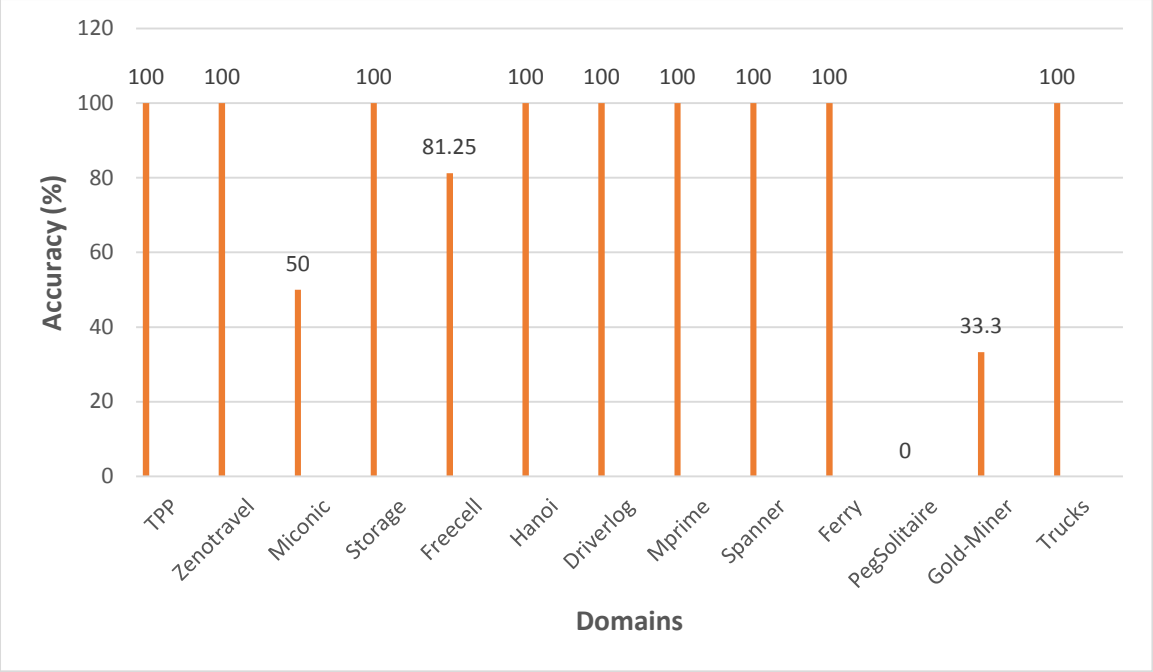


Figure 6.11: Overall Success Trend of ASCoL system per domain

Table 6.5 presents the group of experiments carried out to calculate the accuracy and precision of the system for each particular domain in the dataset. The **Accuracy** defines the proportion of correctly retrieved results amongst the total number of relationships present in the benchmark domain. For instance, the system has only 50% accuracy for Miconic domain as it cannot learn two of the four static preconditions present in the planning benchmark domain. Figure 6.11 shows the graph for the overall success trend of ASCoL experimental analysis for each domain in terms of percentage accuracy. In accordance with the table 6.3, for four domains including Miconic, Freecell, PegSolitaire and Gold-Miner, ASCoL is only 50%, 81.25%, 0.00% and 33.3% accurate, respectively. There are no static facts in benchmark Logistics and Gripper domains so accuracy cannot be calculated.

The **Precision** column mentions the measure of the proportion of correct results amongst the total number of results retrieved by the system for a particular domain. Most of the domains have 100% precision except for a few, e.g. In Zenotravel and Ferry domains, the system learnt two extra static facts which, although semantically correct and useful for the extended domain model, but are not correct according to the principle benchmark definition of the Zenotravel domain and this makes system less precise. For PegSolitaire, ASCoL does not learn the only static fact in the benchmark domain.

Table 6-6: The percentage Accuracy of the ASCoL system per static facts' type

Types	Learnt/Total	Accuracy (%)	Domain(s) Affected
1. Location Map	13/15	86.6	Gold-miner

2. Resource Quantity	18/18	100.0	None
3. Unordered Array	2/2	100.0	None
4. Ordered Sequence	14/14	100.0	None
5. Compound Facts	0/3	0.0	Freecell
6. Non-equality	2/2	100.0	None
7. Dual-Typed	0/4	0.0	Miconic, Logistics
TOTAL	49/58	84.4	---

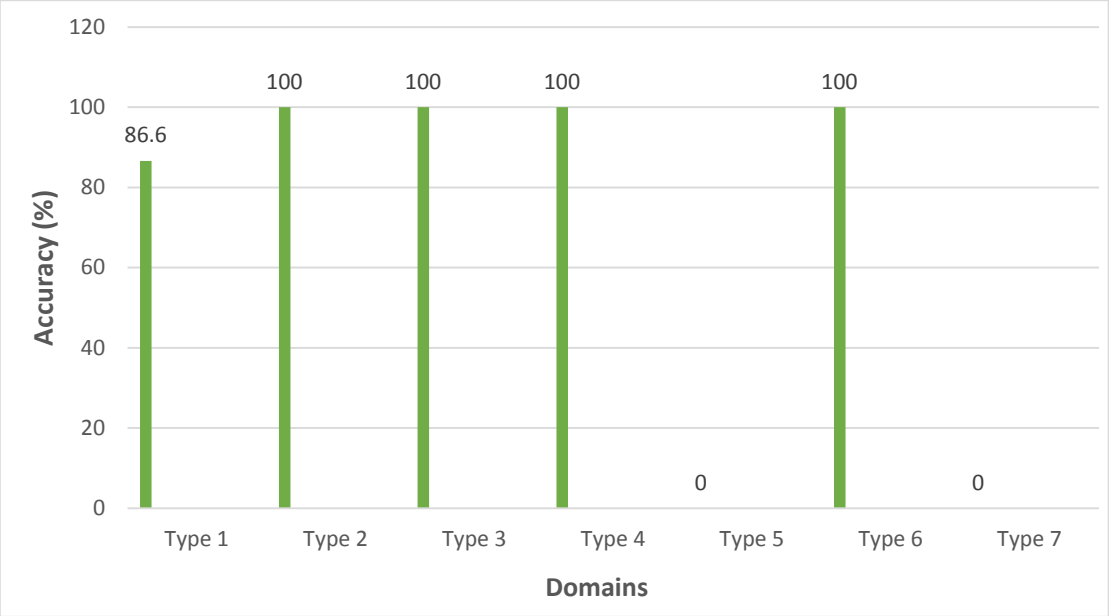


Figure 6.12: Learning Trend of ASCoL based on Types of Static Relations

Table 6.6 displays the accuracy of the system based on the types of static facts we categorised in section 6.2 and the domains affected with missing static relations. The last row also shows the total number of static relations learnt correctly and the net accuracy (%) of the ASCoL system. A column indicates the Domain(s) Affected with missing facts that ASCoL cannot learn.

Figure 6.12 shows the graph for the learning trend of ASCoL based on types of Static Relations. For Type 1: Location Map the compromise in accuracy is due to the Gold-miner domain. This is because of the rare occurrence of detonate-bomb and fire-laser actions per input plan in the domain. The accuracy of the system to learn Compound Facts and Dual-Typed Facts is 0%. The semantics of Compound static facts graph is such that no single connected graph can cover the entire relationship. This is because most often Compound static facts such as those is the Freecell domain (canstack ?c1 ?c2 - card) mix up the domain-independent aspect with the domain-specific aspect in the design phase.

6.6 Significant Experimental Results

This section discusses some of the most interesting results including those where ASCoL is not very competent. There are several reasons behind this and we discuss them under the relevant domain headings below. The LOP system outcome for each domain is also discussed towards the end of each subsection.

6.6.1. Freecell Domain

We used the original version of the Freecell domain model for running our experiments and observed the lack in performance of ASCoL due to poor knowledge engineering practice in the encoding of the benchmark domain. In this domain, there are three cases out of sixteen where ASCoL cannot identify static relations due to a partially ordered graph in the outcome. These three static relations are the instances of the same static fact canstack(card1 card2 – suit) in three unique operators.

We observe that such relations are peculiar/specific to the domain and are overloaded to cover more than one static kind of knowledge in one relation. As an example of this here is canstack(card1 card2 - suit) which describes the stacking rule relation between cards of different suits. In particular, canstack(card1 card2 - suit) at the same time, model the fact that one card c1 can be stacked over another card c2. This fact applies if (i) the colour of the suit of c1 is red, and the colour of the suit of c2 is black (or vice-versa), and (ii) the face value of c1 is smaller (higher if stacking at home) then the value of c2. So overall, the predicate is expressing two unique static aspects under one predicate.

We observed and recommend that the canstack(card1 card2 - suit) can be decomposed into two separate static facts that can then fulfil our criteria of graph analysis, i.e. canstack-rank(rcard1 rcard2 - num) and canstack-suit(scard1 scard2 - suit). By this decomposition, the linear relation of rank can easily be identified by ASCoL. This way the domain independent characteristic can be identified and separated from the domain-specific aspect. For the same reason, we discourage such a modelling approach of combining multiple aspects of the domain into one relation during the design phase of domains.

The rare occurrence of the two operators which include sendtofree-b and newcolfromfreecell prevents ASCoL from learning the corresponding relations from just a few plans. A large number of plans are required; at least 20. However, only 8 plan traces are required for correctly identifying the static relations of the remaining operators. The relatively high number of plans needed is also because the *num* type models different aspects of the game. Empirically, we observed that for reducing the number of plans, splitting the *num* type is useful. *num* in the benchmark domain is used for modelling three different aspects of the game; therefore, we split it as follows: *free-Cell* keeps a record of

the free cells available, *free-Col* keeps a record of empty columns and *home* is used to record the arrangement of cards in the home cell using cards' face value or rank. Splitting type *num* does not split or affect the number of static facts in the domain. This way ASCoL only requires eight plans to learn all the relations.

LOP system is based on the LOCM2 system (Cresswell and Gregory, 2011). LOP targets only 10 static facts to learn the complete Freecell domain model. The authors of the system are of the view that the LOCM2-induced domain already encodes this information as dynamic relations.

6.6.2 TPP Domain

TPP is an example of one of the most complex domains for the evaluation of the ASCoL algorithm. As already mentioned in subsection 6.4.1, that out of four operators, three contain seven arguments each. The large number of arguments of the same type gives rise to six arguments' pairs per operator. Out of six pairs, ASCoL correctly identifies the two static binary relations; as the benchmark domain also contains only two static facts in each operator, i.e. next(level2 level1) next(level4 level3). In contrast, the LOP system is not able to identify these facts as it depends on a LOCM2-encoded domain model for its input. LOCM2 fails to induce the complete dynamics of the TPP domain. Appendix B-2 contains the result produced by ASCoL for type *Level* (last four arguments) in each of challenging operators *Unload*, *Load* and *Buy* of TPP Domain.

6.6.3. Miconic Domain

Miconic domain is a STRIPS encoding of a system to transport a number of passengers within an elevator from their origin to their destination floors. It has four static predicates: two of type 3 – Unordered Array and the remaining two of type 7 – Dual-typed relations. The Unordered Array type static relation is (above ?floor1 ?floor2) and is used in operator *Up* and *Down* for indicating the ordering of arguments of type floor, i.e. the order between the different floors which are connected by the elevator. The Dual-typed static relations are (origin ?passenger ?floor) and (destination ?passenger ?floor). These two are used in operator *board* and *depart*, respectively, to indicate the origin and destination floors of the passenger.

In this domain, the predicate *above* is used for modelling the fact that one floor is above the lower one. This predicate is used both for up and down movements of the elevator, but in a different way. ASCoL successfully detects the presence of a link between floors but generates two different static relations, one per operator. Although different from the original model, such an encoding is an alternative way for modelling the relation between floors. ASCoL cannot learn the two dual-typed static relations because such cases

(although rare) contain two different types of arguments which cannot be expressed by the ASCoL graph technique.

The LOP system is not able to identify the relations between different floors in the *Up* and *Down* operators. According to authors, this is due to more groundings for up and down actions in the plans which never leads to a shorter plan, and even an empty static fact does not preserve the optimality of input plans (Gregory and Cresswell, 2015).

6.6.4. Gold-Miner Domain

From the learning track of IPC6, Gold-miner domain is a STRIPS encoding of a scenario where a robot is in a mine (grid) and has the goal of reaching a location that contains gold by shooting through the rocks. It exploits a static relation (*connected*), in three operators: move, detonate-bomb and fire-laser. *Connected* predicate indicates that the two adjacent cells are connected.

In our experimental analysis, we generated plans using the problem set provided for IPC6. By considering the corresponding plans, ASCoL easily learns the connection fact used in the move operator. Interestingly, ASCoL is not able to correctly identify the same static relationship in the remaining two operators. This is due to the reason that both *detonate-bomb* and *fire-laser* actions are rarely used, usually only once per plan. Moreover, in the considered IPC6 problems, the structure of the problems forces the robot to fire the laser (or to detonate bombs) within the same row of the grid, but not across different columns. Changing the structure of the problems slightly, so that the robot can detonate bombs (or fire the laser) also between cells of different columns, allows ASCoL to learn all of the static relations of the Gold-miner domain model. It should be noted that changing the problems does not modify the overall structure of the domain model.

The LOP authors did not include this domain in their experimental dataset and the system itself is not publically available.

6.6.5. PegSolitaire Domain

PegSolitaire is a board game with typically 33 holes on the board. In some versions of the game, the board may vary in size. Each peg is arranged in holes on the board with only one hole left free in middle. Pegs can make moves like draughts both horizontally or vertically. To win the game, the goal is to gradually remove all the pegs until only one remains at the location where initially there was a hole. There are several versions of the domain including a temporal domain proposed in IPC6. We used the propositional version with only one operator *Jump*. *Jump* operator (figure 6.13) includes only one static precondition (in-line ?from ?over ?to - location). This precondition restricts the action execution to move pegs in a line be it vertical or horizontal.

```
(:action jump
            :parameters (?from - location ?over - location ?to - location)
            :precondition (and (occupied ?from)
                               (IN-LINE ?from ?over ?to)
                               (occupied ?over)
                               (free ?to))
            :effect (and (not (occupied ?from))
                         (not (occupied ?over))
                         (not (free ?to))
                         (free ?from)
                         (free ?over)
                         (occupied ?to))
)
```

Figure 6.13: Operator *Jump* from Benchmark PegSolitaire Domain

The fact that ASCoL is a binary static constraints learner and because by the standard definition of static facts, all the objects are linked pairwise, ASCoL failed to discover any useful results from the domain.

6.6.6. Mprime Domain

As already described in the Interesting Models section 6.4.3 that the Mprime domain is among the most challenging domain scenarios. In our experimental analysis, we generated plans using the problem set provided for IPC1. By considering the corresponding plans, ASCoL easily learns all eight relations used in the four operators *Move*, *Load*, *Unload* and *Donate*. The *Donate* operator is especially the most challenging operator with five arguments (nine pairs of arguments) of the same type (figure 6.14). ASCoL correctly learns all three static facts of the *Donate* operator as the linear ordered graphs with 100% accuracy and precision. Appendix B-1 contains the results produced by ASCoL for *Donate* operator.

```
(:action donate
    :parameters (?l1 ?l2 - location ?f11 ?f12 ?f13 ?f21 ?f22 - fuel)
    :precondition (and (not-equal ?l1 ?l2)
                       (has-fuel ?l1 ?f11)
                       (fuel-neighbor ?f12 ?f11)
                       (fuel-neighbor ?f13 ?f12)
                       (has-fuel ?l2 ?f21)
                       (fuel-neighbor ?f21 ?f22))
  :effect (and (not (has-fuel ?l1 ?f11))
               (has-fuel ?l1 ?f12)
               (not (has-fuel ?l2 ?f21))
               (has-fuel ?l2 ?f22))
)
```

Figure 6.14: Operator *Donate* of Mprime Benchmark domain

6.7 Impact of Differently-Generated Plans

This section explains how different types of plan traces can affect the ability of both LOCM and ASCoL in generating domain models and static knowledge, respectively. Investigative

experiments have been performed to try to determine the best method of generating plan traces. After the comparison between goal-oriented and randomly generated plans, it was learnt that both the methods have their own drawbacks and advantages.

For the better understanding of the complexity of the learning problem faced by ASCoL, metrics were created to learn the amount of input plans and types of plans required to effectively learn these static facts. We calculated the precision between results generated using goal-oriented (GO) and random-walk (RW) plans separately and compared them to each other and to the benchmarks. The same set of planning instances have been used for generating both goal-oriented and random-walk traces. The latter have been created by using a Java-based tool able to parse a given domain model and problem to generate subsequent valid traces of a given length. Clearly, such traces usually do not reach the goal required by the planning instance, but if the goal is reached, they provide richer information in terms of the number of transitions for different types of static facts when compared to the GO plan sequences. GO solutions are generally expensive in that a tool or a planner is needed to generate a large enough number of correct plans to be used by the system, but they can also provide useful heuristic information.

In order to evaluate the number and structure of plan traces required by LOCM and ASCoL for learning, it is important to specify when the learning process ends. For this specific purpose, the notion of convergence has been exploited. We say that ASCoL (LOCM) converges after N plan traces if increasing the value of N –while maintaining the overall structure of the traces– does not produce any change in the output of the learning system. In other words, if adding more plan traces does not affect the generated domain models, the learning system is deemed to have converged.

ASCoL generally requires a larger amount of example data (N') than LOCM (N) in order to converge. As previously described, ASCoL being the static knowledge learner, relies on the generation of directed graphs' order. The more complete the graphs, the more accurate is the analysis, thus the largest number of static relations can be identified. On the other hand, LOCM learns the dynamic domain model and extracts single transitions between parameters from plan traces, thus requiring a smaller amount of information. We can call the input set of ASCoL the superset of the input set for LOCM ($N \subseteq N'$).

As the outcome of the system, it is possible that we sometimes get less precise but accurate results and sometimes, inaccurate very precise results. For instance, in the first case, a benchmark domain contains fewer numbers of static facts as compared to the number of facts that the ASCoL learns. In the second case, in some domains, ASCoL learns fewer but semantically correct relationships compared to the benchmark domain. Similarly, for a particular domain, either the results produced by GO or RW plans could be imprecise as for certain domain scenarios only one of either GO or RW plans serve to produce accurate and precise results. If GO results does not match with RW results, one

of the results are considered imprecise by matching against the benchmark relationships. Tables 6.7 and 6.8 show the analysis done on the number of plan traces –GO and randomly generated RW plans – required by LOCM and ASCoL, respectively, for reaching convergence. Results also show the average plan length and highlight the presence of differences in the generated domain models.

It has long been recognised that accuracy and precision cannot be calculated for domain learning system like LOCM as there is no scale of either syntactically or semantically correct outcome on which domain model can be measured and also because there can be variation in the encodings of a single domain. LOCM paper (Cresswell et al., 2013) describes the evaluation of the system on five different domains in much detail based on the factors of convergence, equivalence and adequacy. We evaluated LOCM system on some more domains as discussed in chapter 3. We now examine the LOCM algorithm based on variety in its input plan traces which has not been covered by LOCM authors.

6.7.1 LOCM

In Table 6.7, it can be noted that in five of the considered domains the final models which are generated by exploiting GO or RW traces do not match with each other. As a general observation, we noticed that GO plans allow LOCM to produce better and more complete Finite State Machines (FSMs) for involved objects. Specifically, in TPP, Miconic and Logistics, FSMs generated by considering RW plan traces are incomplete. This is because many operators appear rarely (or not at all) in the randomly generated plan traces. For each original domain in Table 6.7, it is expressed in terms of types of plan traces, i.e. goal-oriented and random-walk, the number of input Plans (# of Plans) provided and the average number of actions per plan (Actions/Plan) that allows LOCM to converge. The last column indicates if the models generated by considering the two types of plan traces are equal (Yes or No). Figure 6.15 shows a Venn diagram including all the domains and classify them based on the type of plan traces required to learn them correctly.

On the other hand, we also observed that in two domains, Mprime and Trucks, the use of RW traces allows LOCM to provide more accurate domain models i.e. one close to the benchmark. In these domains, the use of GO plans lead to the generation of redundant states for some of the involved objects.

Table 6-7: LOCM evaluation based on the type and quantity of plans

Domain	LOCM				
	Goal Oriented		Random-Walk		Match
	# of Plans	Actions/Plan	# of Plans	Actions/Plan	Yes/No
TPP	1	250	-	-	No
Zenotravel	1	60	3	10	Yes
Miconic	3	30	1	297	No
Storage	10	15	1	142	Yes
FreeCell	1	135	1	90	Yes
Hanoi	1	41	1	30	Yes
Logistics	20	51	1	45	No
Driverlog	11	19	4	22	Yes
Mprime	10	18	1	78	No
Spanner	5	25	3	20	Yes
Gripper	1	78	1	53	Yes
Ferry	1	120	1	47	Yes
PegSolitaire	7	40	1	250	yes
Gold-Miner	1	20	1	42	Yes
Trucks	10	30	8	25	No

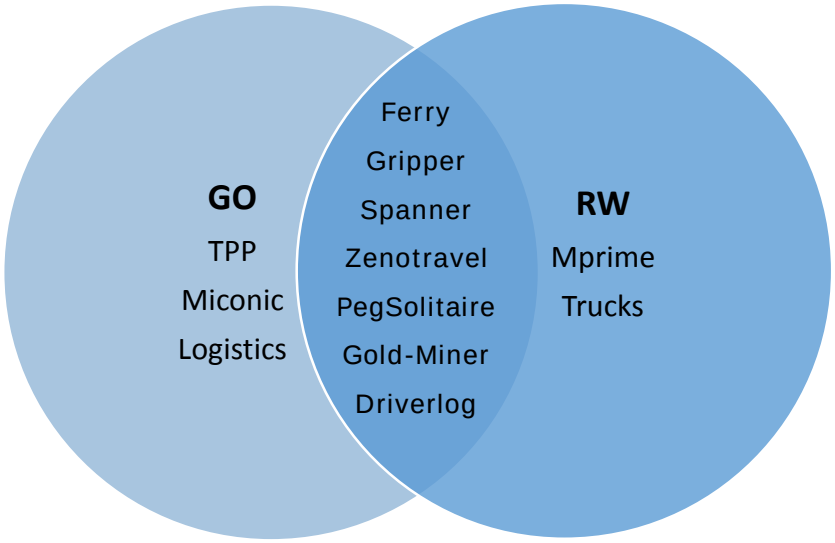


Figure 6.15: Venn diagram - Learning by LOCM based on plan types

6.7.2 ASCoL

Table 6.8 shows the results of using different types of plan traces for the ASCoL method. There are some domains, including TPP, Logistics and Gold-Miner, where the use of GO plans does not allow ASCoL to learn static preconditions fully. In some domains, RW plans do not allow ASCoL to learn static facts altogether due to the rare appearance of actions in plans that contain static constraints (Hanoi, gold-miner). Neither RW nor GO plans can learn all static constraints in the Gold-Miner domain; the reason is already discussed in section 6.6.4. For each original domain in Table 6.8, it is expressed in terms of types of plan traces, i.e. goal-oriented and random-walk, the number of input Plans (# of Plans) provided and the average number of actions per plan (Actions/Plan) that allows ASCoL to converge. The last column indicates if the models generated by considering the two types of plan traces are equal (Yes) or not (No). Figure 6.16 shows a Venn diagram including all the domains and classify them based on the type of plan traces required to learn the static knowledge correctly.

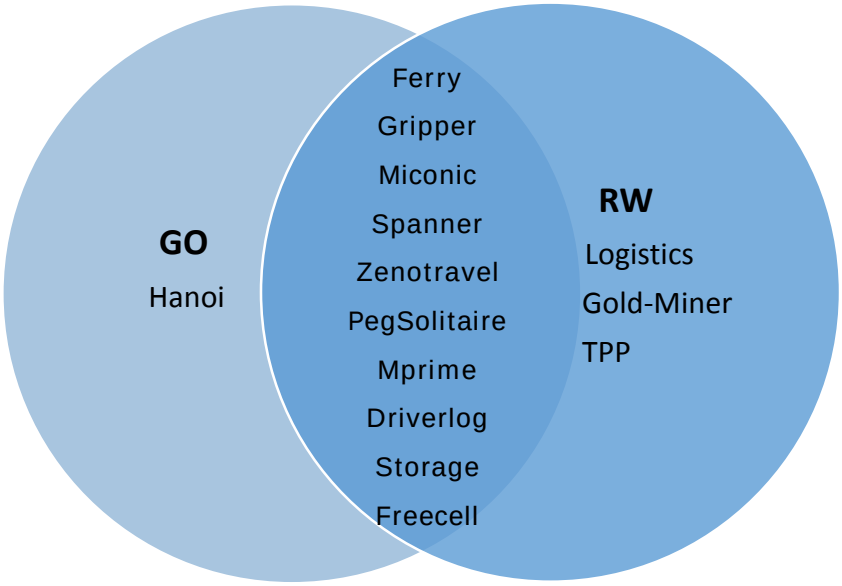


Figure 6.16: Venn diagram – Learning by ASCoL based on plan types

Table 6-8: ASCoL evaluation based on the type and quantity of plans

Domain	ASCoL				
	Goal-Oriented		Random-Walk		Match
	# of Plans	Actions/Plan	# of Plans	Actions/Plan	Yes/No
TPP	10	123	3	256	No
Zenotravel	3	35	5	17	Yes
Miconic	5	54	1	177	Yes
Storage	25	22	24	15	Yes
FreeCell	20	60	9	115	Yes
Hanoi	1	60	-	-	No
Logistics	25	50	1	117	No
Driverlog	3	12	3	20	Yes
Mprime	13	19	4	41	Yes
Spanner	1	8	1	13	Yes
Gripper	2	14	1	50	Yes
Ferry	1	20	1	15	Yes
PegSolitaire	-	-	-	-	-
Gold-Miner	17	29	14	24	No
Trucks	6	25	2	31	Yes

6.7.3 Discussion

This section presents the analysis on our findings after detailed experiments with the targeted fifteen planning benchmark domain models. The main aim is to investigate the effects of using different types of plan traces for producing static knowledge.

It is observed that for ASCoL, no strict balance is required between the number and type of plans, or actions per plan, required for the number of operators and number of static facts in the domain. This is due to the reason that ASCoL extrapolates actions and it depends on the types of static precondition used in the domain. However, for LOCM, it does matter, as it designs overall action schema. The interested readers may refer to (Cresswell et al., 2009).

The density of actions in plan traces that contain static preconditions acts as a minimum quality measure for training plans to be used for extracting a static graph. Operators containing static preconditions must exist at least twice in each plan, otherwise, the plan trace is considered useless as the likelihood of acquiring a graph structure becomes less. For instance, in the Gold-miner domain, the operators *fire-laser* and

detonate-bomb occur most of the time only once in a plan. This is a specific issue related to the random generator made available by the IPC organisers. Therefore, in Gold-miner we redesigned the problem generator to enable us to generate more suitable initial conditions and goals.

Based on our analysis, we concluded that different kinds of static facts require a very different amount of knowledge to be identified. For instance, relationships of type Location Map, Resource quantity and Ordered Sequence require the maximum number of plan traces, while, Unordered Array and Non-equality types require comparatively fewer number of plans to converge. This is because in the former case, these types produce linear graphs to discover them and one missing edge in linear graph could lead to poor results in the form of partially ordered graphs. In the latter case, where ASCoL requires fewer number of plans (or actions per plan) because these types are learnt using connected cyclic graphs or by the comparison between arguments to check inequality.

Domain models vary on the basis of scenarios they capture and how the domain engineer encodes the object associations in benchmark action schema. Many domains contain operators with preconditions to measure capacity or the level of objects. One of the most complex examples of such domains is TPP. Having a set of products and markets, the amount of goods is discrete and represented by multiple levels. Each of three operators (load, unload and buy) in the domain contains two constraints indicating the next level to load, unload or buy goods. For such constraints, on average, 7 plans of 28 actions per plan usually learn the static facts. This is complex because the number of ordered pairs made for each operator increases with the increase in the levels. The TPP (four Level parameters) and Mprime (five Level parameters) require most plans. In these two domains, RW plans cover a wider scope of transitions and produce richer information in the output graph compared to GO plan sequences. These static facts usually appear as a linear graph in the majority of domains.

Summary

The aim of the evaluation was to test the proposed method in order to identify the best results. The evaluation has been carried out to test the performance of domain-independent methods, i.e. without using domain-specific resources, and compare it to the known planning benchmark encodings of the domains.

The first two sections of this chapter discuss the experimental setup along with the types of static facts that we introduced for the first time in the planning literature. The next few sections of the chapter introduced experimentation metrics used to evaluate results and described some interesting and challenging domains and their structures. Experiments were conducted to observe the learning effects of ASCoL system based on certain parameters that include the types of static facts. These experiments also made it

possible to determine which types of static facts provide significant results over others. Section 6.6 also discuss six significant results in detail. The findings of these experiments were studied in much more detail to carry out further experimentation using different types of plan traces as discussed in section 6.7.

Chapter 7 - Extended Uses of ASCoL

The developed system ASCoL can provide a foundation for many Knowledge Engineering Systems. This chapter presents the extended or alternative uses of ASCoL technique when combined with other techniques. This chapter's contents have been ordered such that the first section covers the combined use of ASCoL with the system of Wickler (Wickler, 2013) in order to analyse planning domains using static analysis. Section 7.2 describes the case study that exploits the combined use of ASCoL system with LOCM system in order to generate domain models corresponding to a number of solitaires.

Static graph relation: Wickler in (Wickler, 2013) defines a static graph relation as follows:
Let $O = [O_1,..., O_{n(O)}]$ be a set of PDDL operators and $P = [P_1,..., P_{n(P)}]$ be a set of n(P) predicate symbols. $a(P_i)$ be the corresponding arities of predicates. Then P_i is a static graph relation if and only if:

- P_i is a static relation;
- P_i is a binary relation i.e. $a(P_i) = 2$; and
- The two arguments of P_i are of the same type $T = arg P_i(1) = arg P_i(2)$.

Gerhard uses static graph relations from the planning domain specification instead of plan traces. Given a static graph relation, it defines the graph as:

Static graph: Let (s_i, g, O) be a planning problem and let (C, T, τ) be the O-type partition derived from the planning problem where:

- s_i is a set of initial states and g is a set of goal states
- O is the Operator set
- C is a set of constants $n(C) \geq 1$ and $C = [c_1, c_2, c_3 ,..., c_{n(C)}]$
- T is a set of types where $n(T) \leq n(C)$ and $T = [t_1, t_2, t_3 ,..., t_{n(T)}]$
- τ is C -->T is a function that defines the type of each constant

If P_i be the static graph relation for the set of operators O, then P_i defines a static graph: $G_{P_i} = (V, E)$ consisting of nodes V and directed edges E, where:

- $V = [c \in C \mid \tau(c) = arg_{P_i}(1) = arg_{P_i}(2)]$ and
- $E = [(c, c') \mid P_i (c, c') \in s_i]$.

7.1 Analysis of Domain Model using Static Graphs

ASCoL + Wickler 2013 method

Static analysis is an analysis method known for the helpful internal validation of a domain description, such as to examine the effects of static knowledge or to discover state invariants. (Wickler, 2013) presents a method for investigation and validation of a STRIPS-like domain model through finding a static graph from the given domain model. As ASCoL automatically learns static relations from plan traces in order to enhance domain models, we decided to use both systems together to improve the overall process of domain validation with the help of both domain operators and training plans.

In almost all domains which contain static knowledge, all the actions that include static graph relations must have other dynamic properties or objects related to the main static relation(s). Such properties or objects define the relationship between static and dynamic aspects of objects in the pre and post execution of that action. Wickler calls operators with such dynamic properties *Shift Operators*.

In the combined use of both methods, we first learn the Main Static Relations (MSRs) using ASCoL. Based on those MSRs, Wickler's method then verify and validates the domain definition based on static analysis of Extended Static Relations (ESRs – fully described in section 5.5) and Shift Operators.

7.1.1 Extended Static Relations (ESRs)

This section uses the same running example of Freecell domain in order to demonstrate the combined usage of both techniques (ASCoL + Wickler's) to search for ESRs.

Example 1 - Freecell Domain: To make it easy to understand, we continue with our example of operator *homefromfreecell* from the benchmark Freecell domain (figure 7.1). Here (successor ?vcard ?vhomecard) and (successor ?ncells ?cells) are two MSRs in the operator which ASCoL successfully learns by identifying a totally ordered static graph. There are two ESRs in the benchmark domain for (successor ?vcard ?vhomecard) predicate:

- (value ?card ?vcard)
- (value ?homecard ?vhomecard)

Both these above mentioned ESRs explain the objects of *(successor ?vcard ?vhomecard)* in terms of face *value* that both *card* objects contain.

- MSR = P_i = (successor ?vcard ?vhomecard)
- ESR = P_j = (value ?card ?vcard) and (value ?homecard ?vhomecard)
- Static type of MSR = t_i = *num (cell)*
- Node-fixed type = t_j = *card*

Each identified ESR in an operator explains the MSR to further level of details in different ways depending upon the nature of the domain e.g. ESR relates some physical objects to one node in the graph in transportation domains. Similarly, in skill-based games it explains the value/cost/property of the node. The same property or ESR may repeat for more than one node of MSR graph.

```
(: action homefromfreecell
        : parameters (?card - card ?suit - suit ?vcard - num ?homecard - card
                ?vhomecard - num ?cells ?ncells - num)
        : precondition (and (incell ?card)
                        (home ?homecard)
                        (suit ?card ?suit)
                        (suit ?homecard ?suit)
                        (value ?card ?vcard)
                        (value ?homecard ?vhomecard)
                        (successor ?vcard ?vhomecard)
                        (cellspace ?cells)
                        (successor ?ncells ?cells))
        : effect (and (home ?card)
                        (cellspace ?ncells)
                        (not (incell ?card))
                        (not (cellspace ?cells))
                        (not (home ?homecard)))
)
```

Figure 7.1: MSR (marked blue), ESRs (marked red), ESR_{L2} (marked green)

In order to represent extended static functional properties in the form of a graph, we extend the totally ordered graph for MSR (successor ?vcard ?vhomecard) P_i produced by ASCoL. It is represented in black colour with circular nodes, while node-fixed type objects for ESR P_j are represented using rectangular nodes in figure 7.2.

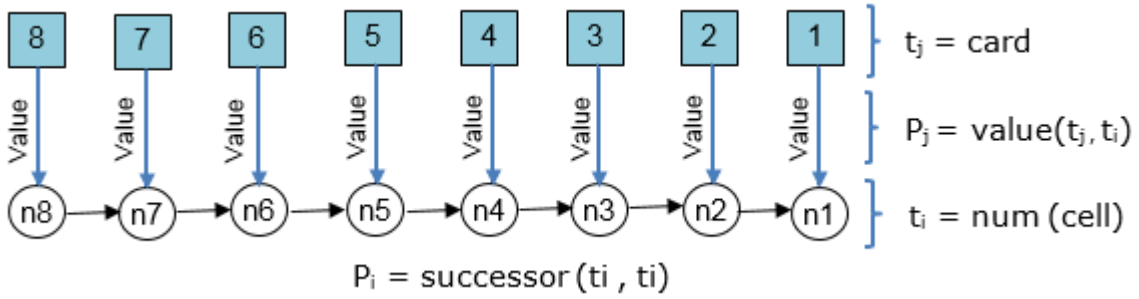


Figure 7.2: ESRs over a linear graph for *homefromfreecell*

Level-two ESRs (ESR_{L2})

Each ESR can further be extended if it fulfils the conditions for ESR for producing further extension in static graph with respect to the node-fixed type i.e. considering ESR's node-fixed static type t_j in place of MSR's static type t_i. We call it Level-two ESRs (ESR_{L2}).

To explain this, the same example can further be expanded in the form of *(suit ?card ?suit)* and *(suit ?homecard ?suit)* considering previous $t_j = t_i$ = card and level-two $t_j = t_k$ = suit as demonstrated in figure 7.3. ESR_{L2} P_k are represented using rectangular orange nodes.

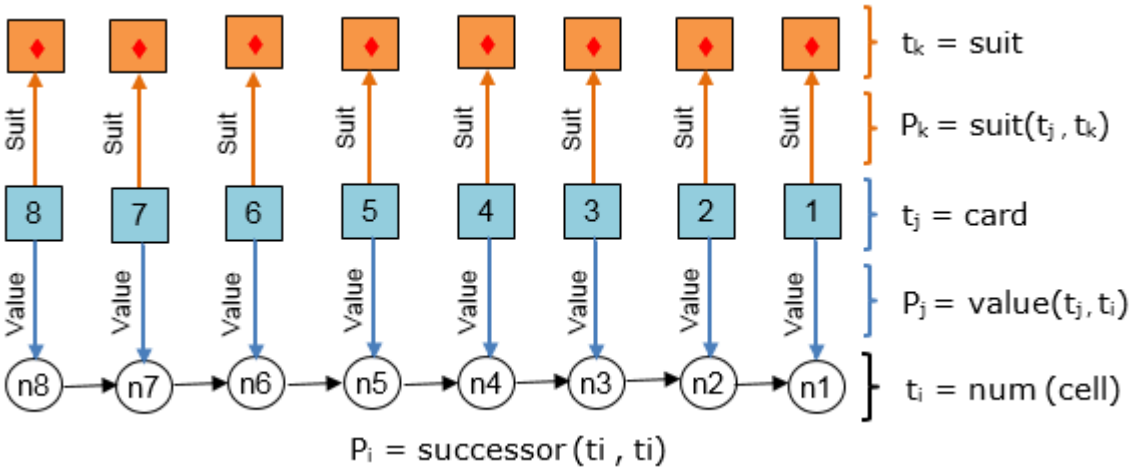


Figure 7.3: ESRs and ESR_{L2} over a linear graph with MSR

The extended part of the graph contains directed edges with node-fixed type as a source of each edge. This is because the ESR P_j could possibly be functional in both directions. This mainly depends upon the order of predicate arguments in the domain under verification.

Example 2 - Grid Domain: The strips version of Grid domain was used in the first planning competition. Without going into the details of the domain description, we just describe how the Unlock operator of this domain can be used as an example and the source of finding ESR and ESR_{L2}. Operator *Unlock* (Figure 7.4) describes the lock position for the lock with a particular key that has a particular shape. Here the MSR or the static graph is defined by *Conn* predicate that indicates the relationship between current position and lock position of the lock. Keeping in mind the definition of ESRs, there is a predicate *lock-shape* which fulfils both the conditions of being ESR and have node fixed-type i.e. a binary predicate as well as having two variables of different types including the type of objects from MSRs:

Here

- MSR = P_i = (conn ?curpos ?lockpos)
- ESR = P_j = (lock-shape ?lockpos ?shape)
- Static type of MSR = t_i = *position*
- Node-fixed type = t_j = *shape*

To find ESR_{L2}, corresponding static graph analysis can further be expanded in the form of predicate *(key-shape ?key ?shape)* considering previous $t_j = t_i$ = *shape* and at level-two $t_j = t_k$ = key.

```
(:action unlock
        :parameters (?curpos ?lockpos - position ?key - key ?shape - shape)
        :precondition (and (place ?curpos) (place ?lockpos)
                    (key ?key)(shape ?shape)
                    (conn ?curpos ?lockpos)
                    (lock-shape ?lockpos ?shape)
                    (key-shape ?key ?shape)
                    (at-robot ?curpos)
                    (locked ?lockpos) (holding ?key))
        :effect (and  (open ?lockpos) (not (locked ?lockpos)))
)
```

Figure 7.4: *unlock* with MSR (blue), ESR (red) and ESR_{L2} (green)

By using this search and analysis method, it not only produces the static relationships between objects in the plan traces but also discovers a further level of networking between the objects depending upon the nature of the domain. For instance, in patience card games, it can provide the relationship between a card, its face value and its suit as shown in the Freecell example already. This combined function of both the methods can be applied to first learn the static facts for the enhancement of LOCM generated domain model and then to do static analysis of the already learnt domain model.

7.1.2 Shift Operators or Static Modifier (O_{SM})

Given a domain model that exhibits static aspects in addition to dynamic behaviour in operator definition, there are often dynamic first-order relations which support the transitions of object states by representing the dynamicity or movement of static graph objects in the dynamic domain scenario. Such first-order dynamic predicates carry different functional properties with the same semantics and it depends upon the domain scenario. Wickler in (Wickler, 2013) calls operators that encompass such a property, Shift Operators and defines them as follows:

If O be a particular planning operator having preconditions p_1 ,….,p_n, positive effects$^+$ (e^+_1 ,…., e^+_n) and negative effects$^-$ (e^-_1,….,e^-_n) where each precondition and positive/negative effect is a first-order atom.

If P_i = static graph relation for the O,
t_i = static type for P_i
t_j = node-fixed type for P_i.

Then, O is a shift operator wrt t_j iff:

- O has a MSR $P_i(v, v')$;
- O has a precondition pre_i with argument v (or v');
- O has a negative effect$^-$ = pre_i;
- O has a positive effect$^+$ = pre_i but the argument v (or v') must alternate with v' (or v), respectively.

According to this definition, an operator O is a shift operator if it contains preconditions, the same negative effect$^-$ and a positive effect$^+$ (but with alternate static argument) 'with respect to t_j '. We altered the definition by also including unary predicates with only static object of type t_i (and no necessity of t_j as a second argument) in it, in addition to binary predicates with reference to t_j. Because only a few of domains exist which contain this shifting property wrt t_j, we extended the definition in order to bring a larger range of domains in the application focus of this verification process in addition to transport domains. We call such operators Static Modifiers (O_{SM}). Biundo et al in (Biundo et al., 2003a) refer to such relationships as cardinality relationships.

Example 1 – Freecell Domain: In our example of Freecell domain, we use *move-b* operator as an illustrative example of O_{SM} (figure 7.5). There are two MSRs involved in this operator i.e. *(successor ?ncols ?cols*) and *(canstack ?card ?newcard).* Here, *(colspace ?cols)* is the modifier predicate of *(successor ?ncols ?cols*) which represents the transition of number of empty columns before and after the action execution in the form of the action precondition, effect$^+$ and effect$^-$, respectively. No predicate fulfils the conditions to be the shift predicate for *(canstack ?card ?newcard).*

```
(:action move-b
        :parameters (?card ?newcard - card ?cols ?ncols - num)
        :precondition (and (clear ?card)
                           (bottomcol ?card)
                           (clear ?newcard)
                           (canstack ?card ?newcard)
                           (colspace ?cols)
                           (successor ?ncols ?cols))
        :effect (and (on ?card ?newcard)
                     (colspace ?ncols)
                     (not (bottomcol ?card))
                     (not (clear ?newcard))
                     (not (colspace ?cols)))
)
```

Figure 7.5: *move-b* - MSR (marked red) and pre_{SM} (marked blue)

From the same domain Freecell, the operators *homefromfreecell*, *colfromfreecell*, *sendtofree*, *newcolfromfreecell* and *sendtofree-b* also modifies (cellspace ?cells) using MSR (successor ?cells ?ncells) to (cellspace ?ncells). Operators *sendtohome-b*, *sendtonewcol*, *newcolfromfreecell* and *sendtofree-b* has a modifier predicate (colspace ?cols) along (successor ?ncols ?cols) to *(colspace* ?ncols). The complete benchmark encoding of Freecell domain is provided in Appendix A.

Example 2 – Grid Domain: In our example of Grid domain, we use the *move* operator as an illustrative example of O_{SM} (figure 7.6). *(Conn ?curpos ?nextpos)* is the MSR involved in this operator. Here, *(at-robot ?curpos)* is the shift predicate for *(conn ?curpos ?nextpos)* which represents the transition of the position of the robot from its current position to its next position in the form of the action precondition, $effect^+$ and $effect^-$, respectively.

```
(:action move
        :parameters (?curpos ?nextpos)
        :precondition (and (place ?curpos) (place ?nextpos)
                           (conn ?curpos ?nextpos)
                           (at-robot ?curpos) (open ?nextpos))
        :effect (and (at-robot ?nextpos) (not (at-robot ?curpos)))
)
```

Figure 7.6: *move* - MSR (marked red) and pre_{SM} (marked blue)

Other Examples: Fourteen more examples have been found in addition to Freecell and Grid domain that we considered for experimentation. Table 7.1 shows the names of domains, the names of the shift/modifier operators (O_{SM}) in each domain, the MSR of the

O_{SM}, the discovered shift/modifier predicate pre_{SM} and the arity of pre_{SM}. Among all fourteen examples, Move operator in Gripper domain, Fly operator in Zenotravel domain and Fly-Airplane operator in Logistics domain do not contain any MSR in the benchmark hand-coded real versions of the domains. As already discussed in section 5.2 that the ASCoL generates additional static facts for some domain that are not present in the original domain model. After manual investigation, we concluded that such additional relations do not reduce the solvability of problems.

Table 7-1: Examples of Static Modifier Operators (O_{SM})

Domains	Operator(s) O_{SM}	Binary MSR	Shift/Modifier Predicate (pre_i)	Unary/Binary
Ferry	Sail	Not-equal	(at-ferry ?location)	Unary
Gold-miner	Move	Connected	(at-robot ?location)	Unary
Gripper	Move	Connected	(at-robby ?room)	Unary
Logistics	Fly-Airplane	Route	(at ?airplane ?location)	Binary
Miconic	Up, Down	Above	(Lift-at ?floor)	Unary
TPP	Drive	Connected	(at ?truck ?location)	Binary
Trucks	Drive	Connected	(at ?truck ?location)	Binary
Trucks	Drive	Next	(time-now ?time)	Unary
Visitall	Move	Connected	(at-robot ?place)	Unary
Spanner	Walk	Link	(at ?man ?location)	Binary
Storage	Move, Go-in, Go-out	Connected	(at ?hoist ?area)	Binary
Zenotravel	Fly	Next	(Fuel-level ?aircraft ?level)	Binary
Zenotravel	Fly	Connected	(at ? aircraft ?city)	Binary
Zenotravel	Refuel	Next	(Fuel-level ?aircraft ?level)	Binary

7.1.3 Conclusion

Wickler's method of domain analysis depends on the domain features that can be extracted automatically by using ASCoL rather than manually. Clearly, there is much more in an operator of a domain that makes a planning domain and problem complex. This combined method described in Section 7.1 builds on the static domain analysis in terms of automatic identification of static graph relations (MSRs, as ASCoL names it), ESRs and O_{SM}. Specifically, by using Static Modifier property, it becomes easy to understand the manipulation of world states in guiding the search space in planning.

The combined usage of both the systems can be exploited in automatic verification and validation of domain models, to ensure consistency of the representation and to

investigate the effect of static knowledge. Although the combined method works on domains, problem instances can also be analysed based on the fact that often the initial conditions in a planning problem that comprise of the static relations are reusable and does not change across a range of problems. Even KE systems like GIPO and itSIMPLE cannot perform domain analysis in terms of static relations as we defined here.

This combined analysis method and the definitions that it is based on exploits certain representational choices that are mostly used when formally representing knowledge in PDDL domain models. There may be substitutes that we have not reflected here that may make the analysis unsuccessful even with the presence of static graphs.

7.2 Benchmarking Planning Domains – ASCoL + LOCM

In this section, we envisage the exploitation of solitaire card games as a pool of interesting benchmarks. In order to "access" such benchmarks, we exploit LOCM and ASCoL for automated generation of domain models corresponding to a number of solitaires. We want to learn problem-specific information and constraints by extracting the underlying game rules (e.g., the initial setup, stacking constraints etc.) which come from problem models. The contribution of this work is twofold. On the one hand, the analysis of the generated models and the learning process itself gives insights into the strengths and weaknesses of the approaches, highlighting lessons learned regarding sensitivity to observed traces. On the other hand, an experimental analysis shows that generated solitaires are challenging for the state-of-the-art of satisficing planning: therefore, solitaires can provide a set of interesting and easy-to-extract benchmarks.

Clearly, domain models include knowledge about the dynamic side, i.e. describe actions which are under the control of the planner, but the actual instantiation and possible execution of such actions depend on the specific problem. Moreover, problem descriptions usually include structure related knowledge, which usually does not change much between problems from the same domain. This is the case, for instance, for patience card games. The dynamic side of games is very similar: cards can be stacked, dealt or moved. Very few differences can then be spotted in the domain models of each patience games. What makes a great difference is mainly the initial setup –in terms of number of columns, presence and number of reserve cells, distribution of cards, etc. – and the goal that has to be reached and this is what differentiates patience games.

Given plan traces P, LOCM automatically generates a domain model D. Moreover, LOCM provides a finite state machine F for each identified type of object in the domain. LOCM also provides part of the initial and goal state descriptions I for each plan trace. ASCoL receives as input part of the knowledge extracted by LOCM, in order to improve the domain model D with preconditions involving static facts: such preconditions are not

identified by LOCM. For extracting static constraints, ASCoL analyses relations between objects taken from the provided plan traces and produce the Enhanced domain model (D_E). Static Relations (s) represent part of the knowledge from the problem description. The exploited learning framework is described in the following Algorithm:

Algorithm: Learning Process

1: **procedure** LEARN (P)

2: D, F, I = LOCM (P)

3: D_E, s = ASCoL (P, D)

4: r = identifyPatienceRules (s, I, D_E)

5: end procedure

Considering solitaire patience games, static constraints here are analogous to static game rules e.g. the fixed stacking relationships between cards keeping the concept of suit (that generally alternate between red and black in columns and remains the same in home cells) and rank (that generally is in descending order in columns and in ascending order in home cells) intact. Also in most solitaire games, the knowledge about available free cells and empty columns act as a resource and plays a vital role in game-winning or goal achievement.

```
(: action homefromfreecell
        : parameters (?card - card ?suit - suit ?vcard - num ?homecard - card
                ?vhomecard - num ?cells ?ncells - num)
        : precondition (and (incell ?card)
                        (home ?homecard)
                        (suit ?card ?suit)
                        (suit ?homecard ?suit)
                        (value ?card ?vcard)
                        (value ?homecard ?vhomecard)
                        (successor ?vcard ?vhomecard)
                        (cellspace ?cells)
                        (successor ?ncells ?cells))
        : effect (and (home ?card)
                        (cellspace ?ncells)
                        (not (incell ?card))
                        (not (cellspace ?cells))
                        (not (home ?homecard)))
)
```

```
(: action homefromfreecell
        : parameters (?card - card ?suit - card ?vcard - num ?homecard -
    card ?vhomecard - num ?cells - num ?ncells - num)
        : precondition (and (zero_state0)
                        (card_state6 ? card)
                        (card_state0 ? suit)
                        (num_state0 ?vcard)
                        (card_state1 ?homecard ?suit ?vhomecard)
                        (num_state0 ?vhomecard)
                        (num_state0 ?cells)
                        (num_state0 ?ncells))
        : effect (and (card_state1 ?card ?suit ?vcard)
                (not (card_state6 ? card))
                (card_state7 ?homecard)
                (not (card_state1 ?homecard ?suit ?vhomecard)))
)
```

Figure 7.7: *homefromfreecell* - Freecell domain (Left) and from LOCM (Right)

As evidence of the concept of this learning, we continue with our running example from Freecell domain i.e. homefromfreecell operator. Figure 7.7 (left) shows the real operator while (right) shows the LOCM induced operator. LOCM creates one predicate corresponding to one state in each FSM. The predicates card_state6, card_state1, card_state0 and

card_state7 can be understood as incell, home, suit and in_homecell_and_covered, respectively.

As LOCM does not learn the background knowledge about objects i.e. the successor(?vcard ?vhomecard) and successor(?ncells ?cells) predicates, the adjacency between particular cards and the alternating sequence of black and red cards. ASCoL induces these static facts as shown in figure 7.8.

```
(:action homefromfreecell
        :parameters (?card - card ?suit - suit ?vcard – num ?homecard –
                        card ?vhomecard – num ?cells ?ncells - num)
        :precondition (and (zero_state0)
                        (card_state6 ?card)
                        (card_state0 ?suit)
                        (num_state0 ?vcard)
                        (card_state1 ?homecard ?suit ?vhomecard)
                        (num_state0 ?vhomecard)
                        (num_state0 ?cells)
                        (num_state0 ?ncells)
                        (link ?ncells ?cells)
                        (link ?vcard ?vhomecard))
        :effect (and
                (card_state1 ?card ?suit ?vcard)
                (not (card_state6 ?card))
                (card_state7 ?homecard)
                        (not (card_state1 ?homecard ?suit ?vhomecard)))

)
```

Figure 7.8: *homefromfreecell* operator induced by LOCM and ASCoL

Finally, the extended domain model D_E, the initial and goal states generated by LOCM and the relations (s) are analysed in order to extract useful knowledge about the structure of problems, constraints of the patience game and goals that need to be achieved. IdentifyPatienceRules method compares initial and goal states provided by LOCM and identifies elements that do not change –given a confidence threshold. Specifically, while the initial distribution of cards varies significantly, the number of columns and reserves do not, as well as goal states. This can provide some important insights into the structure of problems. Moreover, relations (s) provide information about predicates and facts that need to be declared in the initial state.

7.2.1 Assumptions

We mainly focused on three patience games: Freecell, Klondike Solitaire and Bristol. In order to generate the plans required as input to the learning framework, we manually developed three domain models and corresponding problem instances. These patience games share the presence of reserve cells to hold cards during play, foundation/home cells for all four suits, multiple columns for the initial card deal and the subsequent movement of cards to achieve winning conditions.

Starting from an initial state with a standard 52-card deck and a random configuration of cards across a number of columns, the user can move cards in a specified order onto four home cells following typical card stacking rules that are game-specific. The user can use a number of free/reserve cells and empty columns as a resource. The static game rules encoded in preconditions of actions in domain model and initial states of problem models are the allowed sequential arrangement of cards in the free cells, the home cells (played in ascending rank order) and among the card columns (played in descending rank order). Home cells build four stacks of four suits starting from ace to king.

In existing PDDL encodings of card games –i.e., the well-known Freecell and Thoughtful domains, considered in IPCs– *Canstack* and *Successor* constraints are used. The latter provides an ordering on the value of cards and is usually considered for specifying rules about the order of cards in home cells. The Canstack (card1 card2) constraint is used to control the movement of cards in columns; e.g., card1 must be played in descending order, and alternating colours to card2. In our encoding, we decomposed the Canstack constraint into two sub-constraints i.e. Rank-successor (rank1 rank2) and Compatible (suit1 suit2), as some games will require red and black cards to be alternated, but others may require cards to be built in suit or may ignore suit altogether. Considered games are briefly described below.

1. Freecell: in this game, all the 52 cards are already available and distributed among eight columns. They can be moved between columns following specific stack rules involving suit and rank; reserve cells can be used for moving cards. Reserve cells are represented by four empty free cells at the start of the game.

2. Klondike Solitaire: This domain represents one of the most popular solitaire card games. Klondike does not use reserve cells; it deals cards into seven columns. In the initial configuration, 28 cards are dealt in seven columns, containing 1, 2, 3, 4, 5, 6 and 7 cards respectively, keeping 24 cards in the deal stack. Only the top seven cards in columns are face up initially.

3. Bristol: in this game, cards can be stacked regardless of suit both in home cells and in columns. Three cards are dealt to each column with the remaining cards staying in the deal stack. Cards are dealt from this stack onto the three reserve cells, one card each per deal. The top cards in three reserve cells are available to play. Reserve cells can only be filled from the deal stack.

These three domains have been selected because they are encoded using the same Modelling strategy and contain partly different designs of game layout but common game rules. Many other patience games share the same general playing rules.

7.2.2 Complexity of Input

The complexity of input training plans increases with the complexity of problems used to generate them. Mainly it depends on to following two factors:

1. The high card rank included in problem goals: Including cards of higher ranks to achieve the goals requires more moves and makes it complex exponentially, as the home cells build four stacks of four suits starting from ace up to king.
2. The initial configuration of cards: Initial random configuration of cards matters more in some solitaire games than others. For example, in Bristol solitaires, the restriction on reusing empty columns makes the initial configuration the deciding factor in achieving the goal. Less complex configurations include low-ranked cards near to top of column piles in order to easily work towards achieving the goal as the home cells build stacks of four suits starting from low rank up to higher ranks.

For Freecell and Klondike, we included problem instances starting from the goal to have all four home piles filled till rank 8, to having complete suit of 52 cards in home cells. For Bristol, planner exceed the memory limit and run out of time for problems with goal above rank 9.

Freecell, Klondike and Bristol required 4, 6 and 10 plan traces with 58, 45 and 28 average actions per plan, respectively. As already explained before, ASCoL generally requires a larger amount of input example data to converge than LOCM as it relies on the generation of directed graphs. Freecell, Klondike and Bristol required 327, 409 and 376 milliseconds to learn static game constraints by ASCoL. ASCoL requires larger number of plans for some operators that are rarely used, or when the number of arguments of the same type are high, per operator. Klondike and Bristol spent more time for identifying static preconditions. This is due to the reason that many operators require more objects of the same type in both of these domains than Freecell.

7.2.3 Complexity of Card Games Modelling

What makes card games modelling complex to extract a usable domain model is the large set of operators that each has different effects. E.g. To send a card to home cell there must at least be 3 + n actions. One, to send card from the top of the pile of cards in columns, where in the effects of the action the card under it must become clear now. Second, to send the last card of the pile in column to the home cell, where the effects of the action would leave the column empty. Third, to send a card to the home cell straight from the reserve cells. Here, n indicate the variable that changes according to game layout. e.g. in Bristol, to send required card to home it first needs deal action to populate reserve

cells and then *homefromreserve* action to fulfil the sub-goal. Besides this, all static game rules needs mentioning in problem definition.

7.2.4 Performance of Automatic Models Generation

LOCM generated domain model is not understandable due to predicates with automatically generated unique labels. We evaluated the output based on Equivalence and Adequacy of the generated domain model by using the concept of Achiever and Clobberer (Chrpa et al., 2013). For evaluation, we examined the output finite state machines (FSMs) and compared the transition with the actual domain model.

There are number of flaws that LOCM cannot handle. The structural difference between induced and actual domains is that the former contains extra states for the cards in home cells that are covered by another card. We translated the LOCM auto-generated predicate labels as follows and realised that card_state2, card_state3 and card_state7 are fulfilling the same effect and that two of them are redundant.

The auto-labelled predicates and their meaning
card_state0 - suit
card_state1 - home
card_state2 - in_home_cell_and_covered
card_state3 - in_home_cell_and_covered
card_state4 - in_col_and_covered
card_state5 - on_top_of_col
card_state6 - incell
card_state7 - in_home_cell_and_covered

Appendix A has a complete definition of the both Benchmark: Freecell Domain and LOCM: Freecell Domain in PDDL).

LOCM also generates some planner-oriented knowledge including initial and goal states corresponding to each input plan (see Appendix A for LOCM: Freecell Problem Instance). These initial and goal states are captured from the generated dynamic finite state machines (FSMs), thus these do not include the static states required for complete definition of a problem. In other words, LOCM uses generated actions to specify states e.g. in cards game, initial states of the cards are obtained by applying deal (sendtocolumn or sendtoemptycolumn) actions on cards and considering transitions ending of each object as its initial state.

Similarly, by specifying all actions that send cards to home cell, goal states are achieved but this level of problem specification is of limited use provided the plan traces already exist for those problem instances. Also, the output domain model and problem

instances are inadequate due to the absence of static preconditions which are analogous to game rules in card domains.

Running ASCoL on the same set of plan traces learns 90% of problem specific static game rules i.e. static knowledge to enhance LOCM output, including the knowledge of stacking cards according to ranks in home and column piles. It also learns knowledge about number of available free cells and empty columns which is vital in game winning. By combining this static information along with the LOCM generated planner oriented knowledge including initial and goal states, our learning framework can generate the problem instances along with the benchmark domain models. Thus, it is possible to state a planning task independently of the knowledge of state representation.

The remaining 10% of static facts are those for which ASCoL generates a cyclic graph (work in progress). ASCoL generates static facts for such facts but requires manual investigation of the graph for now. This includes the suit compatibility of cards in column piles.

We found the combined results of LOCM + ASCoL for patience games domain model learning, to be adequate for use by solvers to solve low to medium complexity problems. The only factor that requires further work is that the ASCoL system generates static constraints that are generally based on a complete set of input plan traces, while the LOCM generated problem instances are specific to each input plan. To manually fix this problem it needs a human designer to allow only the required subset of static rules according to the objects involved in the problem instance (or input plan). Besides, as mentioned by LOCM authors, to use the action schema model for planning, it is not understandable due to predicates with automatically generated unique labels. To understand the commonly generated problem instances, the automatically generated state labels must be converted to a sensible form before general use.

7.2.5 Performance of State-of-the-Art Planners

Patience card games are considered some of the most difficult domains in classical planning and the evidence is the poor performance of most of the general purpose planners on the domain. The critical and the most complex situation while solving Patience card games problems is the deadlock situation. It is a situation when one particular action, say A, cannot be executed because it requires action B as prerequisite, which in turn requires action A to occur and the generalisation of such situations leads to a waiting condition.

We empirically evaluated the complexity of the previously introduced patience games, in order to check their usefulness as benchmarks for comparing planners' performance. Here we consider FF, Lpg, Madagascar (Rintanen, 2014)(hereinafter Mp) and Yahsp3 (Vidal, 2014). All of them took part in IPC, with notable results. Specifically, Yahsp3 was the winner of the Agile track of IPC 2014, which focused on runtime, while

Madagascar was the runner-up (Vallati et al. 2015). Moreover, FF and Lpg, due to their good performance and to the fact that they are easy to use, are currently exploited in a number of real-world planning applications. Planners have been run on a set of instances from the different patience games. The complexity of the problem instances increases due to initial arrangement and number of cards included in the game setup. We designed the instances starting from low complexity by including only 32 cards (all four suits till rank 8) and went up to 40 cards (up to rank 10) to be placed in the four home piles to achieve the goal conditions. Experiments were run on 3.0 Ghz machine CPU with 4GB of RAM. Cut off time was 1800 CPU-time seconds.

Table 7-2: Performance of State-of-the-Art Planners

Domains	FF		Lpg		Mp	
	Cov	Time	Cov	Time	Cov	Time
Freecell (6)	100	58	100	1252	11	4
Klondike (8)	100	69	100	1442	0	–
Bristol (8)	100	76	0	–	17	14

Table 7.2 shows the performance of FF, Lpg and Mp on instances from the considered patience games in terms of percentage of solved problems and CPU-time seconds. The table displays the percentage of solved instances (Cov) and average CPU-time (seconds) of FF, Lpg and Madagascar on benchmarks from considered patience games. The total number of instances is shown in brackets. Yahsp3 results are not shown since the planner did not solve any considered instance. Interestingly, we observe that most recent planners, i.e. Mp and Yahsp3, perform less well than FF and Lpg in terms of both CPU-time and coverage. None of the considered planners was able to solve problems with a larger number of cards. Most of the failures are due to the planners running out of time, while a few are due to memory limits being exceeded.

In terms of quality, solutions look very similar for all the considered solvers. Number of actions ranges between tens and few hundreds.

7.2.6 Lessons Learnt

Many domain acquisition systems use other inputs in addition to plan traces to learn the output domain model. Empirical analysis suggests that only plan traces can be an interesting test bed to generate a domain model that is complete or near to complete. We learnt that plan traces to act as a fruitful source of knowledge, should not include only single instance of static types in operator parameters, rather it should be in the form of pairs to learn static factors effectively. Especially, for our framework one of the

assumptions of ASCoL is based on the condition of at least two objects of same type to allow analysis of totally ordered graphs.

To decide the types of plan traces in accordance with the above-mentioned facts and our empirical analysis, LOCM supports (Goal-Oriented) GO solutions better than (Random Walk) RW. GO allows LOCM to produce better and more complete finite state machines (FSMs) for involved objects while most of FSMs generated with RW solutions are incomplete. This is because many operators appear rarely (or not at all) in the randomly generated plan traces. In addition, because RW makes it nearly impossible to identify goal states and rules controlling home cells. In general, for ASCoL, no strict balance exists between the number of plans, or actions per plan, required for the number of operators or number of static facts in the domain. Rather, it depends on the types of static precondition and the frequency of actions in plan traces exploited by the domain. Specifically, card games' facts require a large set of GO plan traces in order to learn a complete graph. This is because one missing edge can prevent linearity in the output and so leads to a partially ordered graph with no utility.

Summary

This chapter presents the extended or alternative uses of ASCoL technique by combining its application with other systems. Section 7.1 presents the analysis we carried out for domain models by combining ASCoL system with the system of Wickler. Section 7.2 describes the combined use of ASCoL with LOCM system in order to generate domain models corresponding to a number of solitaires. They exploit the solitaire card games as a pool of interesting benchmarks.

Chapter 8 - Conclusion & Future Work

This chapter presents the summary of our findings and the contributions made in the area of Knowledge engineering for AI planning. It also covers the future perspective for the enhancement of the system.

8.1 Thesis Summary

AI Planning or Automated Planning (AP) - a form of general problem solving, is a pivotal task that has to be performed by every autonomous system. In the past decade, AP has seen important advances and, nowadays, automated planners have the capability of producing plans of hundreds of actions in a wide variety of domains. AP uses a domain model for a particular domain that encodes the knowledge of the domains in terms of actions that can be executed and relevant properties. The manual domain designing process (of both dynamic and static knowledge) is time-consuming, error-prone and is still a challenge for AP community as it is far from being autonomous.

Learning is fundamental to Autonomous behaviour and it can be defined in many different ways. From the point of view of Machine learning, it can be defined as a change in behaviour to allow improvement in performance. To learn domain models for AP is what triggered research into developing automated learning systems from training data.

The research of this thesis concerns the area of automated acquisition of a full or partial domain model from one or more examples of action sequences within the domain under study. There are several knowledge engineering interfaces, translation and debugging tools with varying capabilities. These Interfaces, translation and tools support AP, not only in the knowledge elicitation process but also for the design, validation and verification of developed models. Most of these systems, in addition to plan traces, require additional planner oriented knowledge such as a partial domain model, initial, goal and/or intermediate states. Hence, a question arises whether we can learn a dynamic domain model from real-time action sequence traces only.

The LOCM and LOCM2 systems require as input some plan traces only, and are effectively able to automatically encode the dynamic part of the domain knowledge. However, the static part of the domain is missed, since it can hardly be derived by observing transitions only. The static part of the domain is the underlying structure of the domain that cannot be dynamically changed, but that affects the way in which actions can be performed e.g. predicates representing the connections of roads; the level of floors in the Miconic domain, or the fixed stacking relationships between specific cards in card games. Based on these premises and using the principles of graph theory, the research in this thesis tackles a learning problem. It provides a method for learning static knowledge

for AP domains from training data. The method can be used to automatically produce a large pool of interesting benchmarks for automated planning applications, along with the support of other state of the art KE tools. In broad terms, the topic can be classified as Knowledge Engineering for Automated Planning.

We have developed a generic static constraints learning technique ASCoL (**A**utomatic **S**tatic **Co**nstraints **L**earner), which supports the process of knowledge engineering for AI planning and scheduling. The novelty of ASCoL is that it demonstrates the feasibility of automatically extracting static information from a large collection of structured application knowledge in the form of action sequences. The aim of the ASCoL system is to discover the semantically correct static constraints in order to enhance automatically induced planning domain models. It generates a directed graph representation of operator arguments' relations observed in the plan traces. By analysing such graph topologies, it identifies the types of relations that pairs of predicates show in the form of enhances PDDL domain model.

ASCoL exploits the graph analysis which has been designed to identify different types of static relations automatically from input training data. This also strengthens the performance of domain-independent planners. The author has also developed the ASCoL evaluation tool, which is tailored and adapted to cope with the specific challenges of this genre.

Constraints/Facts learning systems can be thought of as offspring systems of the domain learning systems. The ASCoL system is among the pioneering systems for automatically learning the static knowledge from only a set of plan traces. Traditionally, in learning a complete domain model, systems require more than training plans in the input which include static knowledge of the domain as well. ASCoL has been tested against a variety of IPC domains that have been manually encoded and are based on different scenarios and modelling strategies. Some domain encodings inherit the complexity and peculiar characteristics of the scenario, which poses unique challenges to ASCoL. Through a significant empirical analysis, ASCoL has been shown to be effective.

8.1.1 Requirements and Restrictions of the System

This section introduces the Requirements and Restrictions of the ASCoL System in this thesis. By restrictions, we mean the facts that limit the overall application of the system while requirements include the necessary conditions to fulfil in order to run the system effectively.

Restrictions

Following are some of limiting conditions or restrictions that ASCoL system faces:

- Tackling real-world problems often requires taking various types of constraints into account. Such constraint types range from simple numerical and static comparators to complex resources. In planning, planners have been developed which cast complex reasoning and scheduling problems as a constraint satisfaction problem (CSP). In order to use efficient problem-solving techniques, most search frameworks use only a restricted scenario e.g. Puzzle or Card Game operates under a very limited set of rules. However, some real-world planning systems operate under a much greater set of rules due to the incorporation of time and resource (global or higher level) constraints, which provide highly specialised functionality. These constraints may be available only through auxiliary “black box" problem solvers. Sometimes human user must intervene to guide the search through heuristics. Examples include small Military unit operations, emergency medical treatment, multi-agent robotics applications etc. where each has some specific constraints unique to that scenario. Such problems are called the tactical planning problems (Nareyek et al., 2005). The real life example incidence is of the crash of the Black Hawk Helicopter which was compromised by the terrain in Afghanistan (Drive, 2017). Briefly, a plan and a set of constraints, which might be suitable elsewhere, is not feasible in the high mountains.

  ASCoL is a learning system with the speciality to produce static domain constraints only. ASCoL learns the restriction on the possible value combinations of static variables from the training data in the form of example plans. Many efficient methods have been developed that exploit the higher-level domain knowledge to support solution search in planning but there is still a need for learning systems that can learn global constraints from training data.
- The language of its required input also restricts the ASCoL system, i.e. it requires its inputs to be written in Prolog language. This is related to the coding of the software tool developed to run ASCoL technique. This is covered by designing a separate sequence generator program to produce required format of the input plans.

Requirements

Following is the complete list of requirements of the ASCoL system:

- ASCoL technique requires the correct and complete plan traces. The input training plans are assumed to be noise-free and that plan traces should cover all the possible actions of the world. In terms of accuracy of plan traces, the user of the system cannot really say if the plan traces are accurate. In the experimental domains that have narrow scope like tyre world and blocks domain, it is easy to tell accuracy of the input training data. However, in real world situations the completeness and accuracy of plan traces are never guaranteed to cover all possible constraints, set of rules, entities and eventualities possible e.g. in Military operations and disaster management situations.

- Plans must have been generated by considering problems that share the same objects and object names. This is important because ASCoL generalises the relationship by learning the order of the output directed graph representation, where the vocabulary in the graph are the instances of objects that belong to a type in the set of matching actions from the plan traces. If the same object name changes from plan to plan, this produces discontinuous (incomplete) linear or partial ordered graphs and the outcome of the learning system becomes less accurate and useless. From the real-world application observations, we noticed that when sensors identify and name the objects – they usually exploit fixed names. Moreover, it is acceptable to assume that the relevant objects of a real-world scenario will not change quickly. For example, machines in a factory or trucks and depots for logistics companies.
- The dynamic part of the domain model provided as input should be correct and must include the type of each operator argument. This allows the method to generate right vertices pairs based on matching object type.
- The reported work requires domain models that at least exist in the experimental PDDL domain category in order to produce inputs for the system using planners and to evaluate the system's results as well. This is important because for producing a high number of plans from real world by manually annotating every state in a plan example from snapshots of the environment requires prohibitively high cost in labelling the training examples.
- The user of the systems needs to have at least the basic level of knowledge about the PDDL as the outcome of the system is combined with a PDDL domain model.

8.1.2 Summary of Chapters

After the generic introduction of the area, the rationale and motivation of the research in *Chapter 1*, *Chapter 2* aims to provide a background within which the contributions of this research can fit in and to show how it links up to recent and past research in the area. *Chapter 3* went deeper into the discussion on approaches for learning in autonomous systems. It also described in more detail the choice of approach we adopted in ASCoL for learning static knowledge for planning domain models. It also includes an overview of LOCM family of algorithms and our research on LOCM system.

Chapter 4 is an already published overview of the state of the art in automated tools and techniques that can be exploited to induce or assist in inducing planning domain models. It also compares different techniques according to different criteria including input requirements of the systems, quantity and quality of the provided output, language of model generation, noise in plans, refinement of domains, operational efficiency, user experience and availability of the systems. Towards the end, it discusses the general

structure, guidelines, and recommendations that were derived based on the review and assessment of the eight different state-of-the-art automated KE tools.

Chapter 5 is the description of the core ASCoL algorithm and its learning behaviour after the discussion of ASCoL assumptions. A section is included that presents argument for extracting same typed knowledge from plans. It also includes the implementation details of the ASCoL batch processing application.

Chapter 6 is an extensive empirical evaluation of the ASCoL system where we evaluate the technique both quantitatively and qualitatively. Evaluation is performed based on learning capability of ASCoL and on different parameters. Parameters include different types of input training plans as well as seven different types of static constraints. Graphs and tables in different sections show the learning abilities of ASCoL approach. The chapter also discusses significant results in more detail which provide views into the strengths and weaknesses of the ASCoL system. It also leads to a range of interesting insights for future work for the system. It also includes the discussion on analysis of domain structures by combining both ASCoL and Gerhard's approach of domain validation.

Chapter 7 covers the extended or alternative uses of ASCoL technique by combining its application with other systems. First section presents the analysis we carried out for domain models by combining ASCoL system with the system of Wickler. Section 7.2 describes the combined use of ASCoL with LOCM system in order to generate domain models corresponding to a number of solitaires and exploit the solitaire card games as a pool of interesting benchmarks.

We have shown the usefulness of the developed technique and this thesis can be viewed as a contribution towards an increase in the autonomy of domain construction. It would allow the automated planning to be embedded into the autonomous agents so that intelligent agents can plan more independently and autonomously without requiring additional knowledge.

8.1.3 Potential Application areas of this research

This section introduces the potential applications of the research contributions presented in this thesis.

AI Planning and domain construction: The system can be used to provide support to the users including planning experts and particularly domain experts and knowledge engineers. It can also benefit users who do not have strong understanding of the dynamic and static behaviour of a domain and need support for planning and modelling for planning applications. ASCoL can also be used by areas that exploit induced domain models based on the evidences which may also come in the form of plan traces. Such application areas include robotics and autonomous vehicles, automatic control of industrial processes, automatic crisis management, and automatic user support for project planning, generation

of automatic control programmes like chemical plant control, automatic business processes modelling, autonomous agents and many more. Apart from AI planning and Knowledge Engineering, other wider application areas that involve synthesis of constraints or invariants can benefit from the method. Examples include the detection of anomalies in domain design, extraction of maps of locations from past activity, learning of the static rules of games like solitaires, draft games etc. and the detection of increasing or decreasing trend in object levels.

Analysis and diagnosis: The ASCoL technique specifically works well in planning area. It can also be used by other systems/individuals to extract patterns in large real-world case studies which cannot be dynamically discovered using finite state automata e.g. to extract data from large databases, to analyse social networks or work markets, the diagnosis of certain diseases like spreading of virus. Hence, the internet which is a network on its own can be a good application area where different HTML documents act as the nodes of the graph and the hyperlinks act as the edges in the graph. An example includes Google, which uses the structure of Internet in its famous PageRank algorithm (Ma et al., 2008) for websites ranking.

ASCoL application face some limitations outside the planning community. That include the accessibility to log of actions. In many systems other than AI planning, generating training data for learning is not a big challenge. Following are some examples:

In many operating systems, batch commands consist of the name of each command along with some partial information about directory location, structure and content. If we view such a scenario as a planning domain, the batch command is a list of actions. We can easily get the action name and the parameters of each action by reading the system manually. This corresponds to an example of training plans. In this domain, although action model is not specified, it can easily be generated by using a large number of batch commands, viewed as plan examples.

Another important application concerned with wireless sensor networks and machine learning is activity recognition. It aims to recognise the actions and goals of one or more agents from a series of observations on the agents' actions and the environmental conditions. Training data for model learning algorithms can be generated from activity recognition scenario. This can be done by using the sensor readings in a pervasive environment to understand what activities are going on by collecting a large number of sensor reading sequences. Corresponding activity sequences can be obtained by performing activity recognition on sensor reading sequences. Here, the downside is that the noise can be introduced when some sensors are occasionally damaged or with unintentional mistakes in the recording of the action sequence thus, the activities recognised by activity recognition algorithms may sometimes be incorrect.

8.2 Future Work

The learning technique we employed is suitable for improvements. We see several avenues for future development of the system. These can be summarised as below:

- Apply the system in more real life case studies e.g. the once discussed in previous section on potential application areas. The evaluation of ASCoL should be done by using large amount of data from different source. For instance, real-time activity log recorded from a car manufacturing company or from the sequence list of batch commands (discussed in previous section).
- Although much work has been done in identifying static knowledge that is described by AI planning domains, the dual-typed static constraints are still unexplored. The ASCoL system should be extended to deal with it so that extended static facts can also be learnt and the application focus of the technique could be improved.
- An established open problem in AI is to construct plans that can solve multiple problem instances. Learning from generalised plans is one of interesting future work. The example is a DISTILL (Winner and Veloso, 2003, Winner and Veloso, 2007) system that uses example plans to learn domain-specific planners.
- To improve ASCoL to produce game rules for a specific range of objects in a particular problem instance without manual handling and to identify domain models for other games including broad games and other logic-based combinatorial number placement puzzle games.

Bibliography

AARTS, F., DE RUITER, J. & POLL, E. Formal models of bank cards for free. Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on, 2013. IEEE, 461-468.

AMIR, E. & CHANG, A. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 349-402.

ARMSTRONG, A. 2016. Artificial Intelligence - Strong and Weak.

AVESANI, P., PERINI, A. & RICCI, F. 2000. Interactive case-based planning for forest fire management. *Applied Intelligence,* 13, 41-57.

BARREIRO, J. E. A. EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization. In Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12) – The 4th International Competition on Knowledge Engineering for Planning and Scheduling., 2012.

BARTÁK, R., FRATINI, S. & MCCLUSKEY, T. L. 2013. Preface to special issue on Knowledge Engineering for Planning and Scheduling. *The Knowledge Engineering Review,* 28, 117.

BARTAK, R. & MCCLUSKEY, L. 2006. The first competition on knowledge engineering for planning and scheduling. *AI Magazine,* 27, 97.

BAUER, A., WOLLHERR, D. & BUSS, M. 2008. Human–robot collaboration: a survey. *International Journal of Humanoid Robotics,* 5, 47-66.

BENSALEM, S., HAVELUND, K. & ORLANDINI, A. 2014. Verification and validation meet planning and scheduling. *International Journal on Software Tools for Technology Transfer,* 16, 1-12.

BENSON, S. Action model learning and action execution in a reactive agent. Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95), 1995.

BENSON, S. S. 1996. *Learning action models for reactive autonomous agents.* PhD PhD Thesis.

BERNARDI, G., CESTA, A., ORLANDINI, A. & FINZI, A. 2013. A knowledge engineering environment for p&s with timelines. *Knowledge Engineering for Planning and Scheduling*, 16.

BERNARDINI, S. & SMITH, D. E. Developing domain-independent search control for europa2. Proceedings of the Workshop on Heuristics for Domain-independent Planning at ICAPS, 2007.

BERTOLI, P., PISTORE, M. & TRAVERSO, P. 2010. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence,* 174, 316-361.

BESSIERE, C., COLETTA, R., DAOUDI, A., LAZAAR, N., MECHQRANE, Y. & BOUYAKHF, E.-H. Boosting Constraint Acquisition via Generalization Queries. ECAI, 2014. 99-104.

BIUNDO, S., AYLETT, R., BEETZ, M., BORRAJO, D., CESTA, A., GRANT, T., MCCLUSKEY, L., MILANI, A. & VERFAILLE, G. 2003a. Technological roadmap on AI planning and scheduling. *Informe Técnico IST-2000-29656, PLANET, the European Network of Excellence in AI Planning*.

BIUNDO, S., AYLETT, R., BEETZ, M., BORRAJO, D., CESTA, A., GRANT, T., MCCLUSKEY, L., MILANI, A. & VERFAILLE, G. 2003b. Technological roadmap on AI planning and scheduling. *Planet (ENoE in AI Planning)*.

BJÖRNSSON, Y. Learning Rules of Simplified Boardgames by Observing. ECAI, 2012. 175-180.

BLUM, A. L. & FURST, M. L. 1997. Fast planning through planning graph analysis. *Artificial intelligence,* 90, 281-300.

BONET, B., LOERINCS, G. & GEFFNER, H. A robust and fast action selection mechanism for planning. AAAI/IAAI, 1997. 714-719.

BOOCH, G. 2005. *The unified modeling language user guide*, Pearson Education India.

BORDES, A., WESTON, J., COLLOBERT, R. & BENGIO, Y. Learning structured embeddings of knowledge bases. Conference on Artificial Intelligence, 2011.

BORRAJO, D. & VELOSO, M. 1997. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Lazy learning.* Springer.
BOTEA, A., ENZENBERGER, M., MÜLLER, M. & SCHAEFFER, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research,* 24**,** 581-621.
BOUILLET, E. E. A. Mario: middleware for assembly and deployment of multi-platform flow-based applications. In Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, Middleware '09, 26:1–26:7. New York, NY, USA: Springer-Verlag New York, Inc., 2009.
BRATKO, I. & URBANČIČ, T. 1997. Transfer of control skill by machine learning. *Engineering Applications of Artificial Intelligence,* 10**,** 63-71.
BREDEWEG, B., SALLES, P., BOUWER, A., LIEM, J., NUTTLE, T., CIOACA, E., NAKOVA, E., NOBLE, R., CALDAS, A. L. R. & UZUNOV, Y. 2008. Towards a structured approach to building qualitative reasoning models and simulations. *Ecological Informatics,* 3**,** 1-12.
BRESINA, J., DRUMMOND, M. & KEDAR, S. 1993. Reactive, integrated systems pose new problems for machine learning. *Machine Learning Methods for Planning***,** 159-195.
BRESINA, J. L., JÓNSSON, A. K., MORRIS, P. H. & RAJAN, K. Activity Planning for the Mars Exploration Rovers. ICAPS, 2005. 40-49.
CAKMAK, M., CHAO, C. & THOMAZ, A. L. 2010. Designing interactions for robot active learners. *Autonomous Mental Development, IEEE Transactions on,* 2**,** 108-118.
CHAPMAN, D. 1987. Planning for conjunctive goals. *Artificial intelligence,* 32**,** 333-377.
CHEIN, M., MUGNIER, M.-L. & CROITORU, M. 2013. Visual reasoning with graph-based mechanisms: the good, the better and the best. *The knowledge engineering review,* 28**,** 249-271.
CHIEN, S., KNIGHT, R., STECHERT, A., SHERWOOD, R. & RABIDEAU, G. Integrated planning and execution for autonomous spacecraft. Aerospace Conference, 1999. Proceedings. 1999 IEEE, 1999. IEEE, 263-271.
CHRPA, L., VALLATI, M. & MCCLUSKEY, T. L. Determining Linearity of Optimal Plans by Operator Schema Analysis. SARA, 2013.
CLOUARD, R., ELMOATAZ, A., PORQUET, C. & REVENU, M. 1999. Borg: A knowledge-based system for automatic generation of image processing programs. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 21**,** 128-144.
COLES, A., COLES, A., FOX, M. & LONG, D. 2011. popf2: a forward-chaining partial order planner. *The 2011 International Planning Competition***,** 65.
CRESSWELL, S. 2009. LOCM: A tool for acquiring planning domain models from action traces. *ICKEPS 2009*.
CRESSWELL, S. & GREGORY, P. Generalised Domain Model Acquisition from Action Traces. ICAPS, 2011.
CRESSWELL, S., MCCLUSKEY, T. L. & WEST, M. M. Acquisition of Object-Centred Domain Models from Planning Examples. ICAPS, 2009.
CRESSWELL, S. N., MCCLUSKEY, T. L. & WEST, M. M. 2013. Acquiring planning domain models using LOCM. *Knowledge Engineering Review*.
DAVIDSSON, P., HENESEY, L., RAMSTEDT, L., TÖRNQUIST, J. & WERNSTEDT, F. 2005. An analysis of agent-based approaches to transport logistics. *Transportation Research part C: emerging technologies,* 13**,** 255-271.
DE VRIES, P. H. 1989. *PhD Thesis: Representation of scientific texts in knowledge graphs*.
DRIVE, T. 2017. The US Plan to Give Afghanistan a Fleet of Black Hawks Is Deeply Flawed.
ELYASAF, A., HAUPTMAN, A. & SIPPER, M. GA-FreeCell: Evolving solvers for the game of FreeCell. Proceedings of the 13th annual conference on Genetic and evolutionary computation, 2011. ACM, 1931-1938.
ENGINEERING, C. 2015. Artificial intelligence for control engineering, accessed 14 July 2016. *http://www.controleng.com/single-article/artificial-intelligence-for-control-engineering/1284e9173857dba3c4fa632978e2209f.html*.
ERSEN, M. & SARIEL-TALAY, S. Learning interactions among objects, tools and machines for planning. Computers and Communications (ISCC), 2012 IEEE Symposium on, 2012. IEEE, 000361-000366.

FEIGENBAUM, E. A. 1984. Knowledge engineering. *Annals of the New York Academy of Sciences,* 426**,** 91-107.
FENG, Z., DEARDEN, R., MEULEAU, N. & WASHINGTON, R. Dynamic programming for structured continuous Markov decision problems. Proceedings of the 20th conference on Uncertainty in artificial intelligence, 2004. AUAI Press, 154-161.
FERRER, A. G. 2012. *PhD Thesis: Knowledge Engineering Techniques for the Translation of Process Models Into Temporal Hierarchical Planning and Scheduling Domains*, Editorial de la Universidad de Granada.
FIKES, R. E., HART, P. E. & NILSSON, N. J. 1972. Learning and executing generalized robot plans. *Artificial intelligence,* 3**,** 251-288.
FIKES, R. E. & NILSSON, N. J. 1972. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence,* 2**,** 189-208.
FOX, M. & LONG, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research***,** 367-421.
FOX, M. & LONG, D. Hybrid STAN: Identifying and managing combinatorial optimisation sub-problems in planning. Proceedings of International Joint Conference on Artificial Intelligence, 2001. 445-452.
FOX, M. & LONG, D. 2003. PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res.(JAIR),* 20**,** 61-124.
FOX, M. & LONG, D. 2006. Modelling Mixed Discrete-Continuous Domains for Planning. *J. Artif. Intell. Res.(JAIR),* 27**,** 235-297.
FOX, M. & LONG, D. 2011. The automatic inference of state invariants in TIM. *arXiv preprint arXiv:1105.5451*.
FOX, M., LONG, D. & MAGAZZENI, D. 2011. Automatic Construction of Efficient Multiple Battery Usage Policies. *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS-11).*
FRANK, J. & JÓNSSON, A. 2003. Constraint-based attribute and interval planning. *Constraints,* 8**,** 339-364.
GARVEY, P. R. & PINTO, C. A. Introduction to functional dependency network analysis. The MITRE Corporation and Old Dominion, Second International Symposium on Engineering Systems, MIT, Cambridge, Massachusetts, 2009.
GEREVINI, A. 2004. The LPG-td version, accessed 14 April 2016. *Available at: http://lpg.unibs.it/lpg/*.
GEREVINI, A. & LONG, D. 2005. Bnf description of pddl3. 0. *Unpublished manuscript from the IPC-5 website*.
GEREVINI, A., SAETTI, A. & SERINA, I. 2003. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research***,** 239-290.
GEREVINI, A., SAETTI, A., SERINA, I. & TONINELLI, P. LPG-TD: a fully automated planner for PDDL2. 2 domains. In Proc. of the 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04) International Planning Competition abstracts, 2004. Citeseer.
GEREVINI, A. & SCHUBERT, L. Inferring state constraints for domain-independent planning. AAAI/IAAI, 1998. 905-912.
GEREVINI, A. & SERINA, I. LPG: A Planner Based on Local Search for Planning Graphs with Action Costs. AIPS, 2002. 281-290.
GIL, Y. 1992. Acquiring domain knowledge for planning by experimentation. DTIC Document.
GONZÁLEZ-FERRER, A., FERNÁNDEZ-OLIVARES, J. & CASTILLO, L. 2009. JABBAH: a Java application framework for the translation between business process models and HTN. *Proceedings of the 3rd International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS'09)*.
GONZALEZ, J., JONYER, I., HOLDER, L. B. & COOK, D. J. Efficient mining of graph-based data. Proceedings of the AAAI Workshop on Learning Statistical Models from Relational Data, 2000. 21-28.
GRANT, T. Assimilating planning domain knowledge from other agents. Proceedings of the 26th Workshop of the UK Planning and Scheduling Special Interest Group, Prague, Czech Republic, 2007.

GRANT, T. 2010. Identifying Domain Invariants from an Object-Relationship Model. *PlanSIG2010*, 57.
GREGORY, P., BJÖRNSSON, Y. & SCHIFFEL, S. The GRL System: Learning Board Game Rules With Piece-Move Interactions. The IJCAI-15 Workshop on General Game Playing. 55.
GREGORY, P. & CRESSWELL, S. Domain Model Acquisition in the Presence of Static Relations in the LOP System. ICAPS, 2015. 97-105.
GUPTA, N. & NAU, D. S. 1992. On the complexity of blocks-world planning. *Artificial Intelligence,* 56**,** 223-254.
HALPIN, T. A. & PROPER, H. A. 1995. Subtyping and polymorphism in object-role modelling. *Data & Knowledge Engineering,* 15**,** 251-281.
HELMERT, M. 2003. Complexity results for standard benchmark domains in planning. *Artificial Intelligence,* 143**,** 219-262.
HELMERT, M. A Planning Heuristic Based on Causal Graph Analysis. ICAPS, 2004. 161-170.
HELMERT, M. 2008. Changes in PDDL 3.1. *Unpublished summary from the IPC-2008 website*.
HOFFMANN, J. 2003. The Metric-FF Planning System: Translating``Ignoring Delete Lists''to Numeric State Variables. *Journal of Artificial Intelligence Research*, 291-341.
HOFFMANN, J. & NEBEL, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 253-302.
HOFFMANN, J., WEBER, I. & KRAFT, F. Planning@ sap: An application in business process management. 2nd International Scheduling and Planning Applications woRKshop (SPARK'09), 2009.
HOWEY, R., LONG, D. & FOX, M. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on, 2004. IEEE, 294-301.
ICAPS-KEPS 2014. Knowledge Engineering for Planning and Scheduling (KEPS), accessed: 19 June 2016. *available at: http://icaps14.icaps-conference.org/workshops tutorials/keps.html*.
ICAPS-VVPS 2011. VVPS. Verification and Validation of Planning and Scheduling Systems, accessed: 13 Jan 2016. *http://icaps11.icaps-conference.org/workshops/vvps.html*.
ICAPS International Planning Competitions, accessed : 02 Jan 2016. *Available at: http://icaps-conference.org/index.php/main/competitions*.
ICAPS 2002. The 2002 Competition, accessed: 20 June 2016. *Available at: http://www.icaps-conference.org/index.php/Main/Competitions*.
ICKEPS'09 2009. International Competition on Knowledge Engineering for Planning and Scheduling, accessed: 11 April 2016. *Available at: http://kti.mff.cuni.cz/~bartak/ICKEPS2009/*.
JILANI, R., CRAMPTON, A., KITCHIN, D. & VALLATI, M. ASCoL: a tool for improving automatic planning domain model acquisition. Congress of the Italian Association for Artificial Intelligence, 2015. Springer, 438-451.
JILANI, R., CRAMPTON, A., KITCHIN, D. E. & VALLATI, M. 2014. Automated Knowledge Engineering Tools in Planning: State-of-the-art and Future Challenges. *KEPS 2014*.
JIMÉNEZ, S., DE LA ROSA, T., FERNÁNDEZ, S., FERNÁNDEZ, F. & BORRAJO, D. 2012. A review of machine learning for automated planning. *The Knowledge Engineering Review,* 27**,** 433-467.
KAUTZ, H. & SELMAN, B. Pushing the envelope: Planning, propositional logic, and stochastic search. Proceedings of the National Conference on Artificial Intelligence, 1996. 1194-1201.
KAUTZ, H. A. Deconstructing planning as satisfiability. Proceedings of The National Conference on Artificial Intelligence, 2006. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 1524.
KHOUADJIA, M. R., SCHOENAUER, M., VIDAL, V., DRÉO, J. & SAVÉANT, P. Multi-objective AI Planning: Evaluating DaE YAHSP on a Tunable Benchmark. International Conference on Evolutionary Multi-Criterion Optimization, 2013. Springer, 36-50.

KINTSCH, W. 1986. Learning from text. *Cognition and instruction,* 3**,** 87-108.
KLEIN, G. 1998. Sources of Power. How people make decisions. Massachusetts Institute of Technology. MIT Press, Cambridge, Massachusetts.
KONIDARIS, G., KAELBLING, L. P. & LOZANO-PEREZ, T. 2014. Constructing symbolic representations for high-level planning. *Proceedings of the 28th AAAI Conference on Artificial Intelligence*.
LEVITT, R. E., KARTAM, N. A. & KUNZ, J. C. 1988. Artificial intelligence techniques for generating construction project plans. *Journal of Construction Engineering and Management,* 114**,** 329-343.
LIN, F. 2004. Discovering State Invariants. *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning,* 4**,** 536-544.
LONG, D. 2003. The Zeno-Travel Domain, accessed: 24 April 2016. *Available at: https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/long03a-html/node40.html*.
LONG, D. & FOX, M. 1999. Efficient Implementation of the Plan Graph in STAN. *J. Artif. Intell. Res.(JAIR),* 10**,** 87-115.
LUGER, G. F. 2005. *Artificial intelligence: structures and strategies for complex problem solving*, Pearson education.
MA, N., GUAN, J. & ZHAO, Y. 2008. Bringing PageRank to the citation analysis. *Information Processing & Management,* 44**,** 800-810.
MALIK GHALLAB, A. H., CRAIG KNOBLOCK ET AL 1998. PDDL | The Planning Domain Denition Language. *AIPS-98 Planning Competition*.
MALIK GHALLAB, D. N., PAOLO TRAVERSO 2004. Automated Planning Theory and Practice. *Morgan Kaufmann Publishers Inc. San Francisco, CA, USA*.
MCCARTHY, J. & HAYES, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Readings in artificial intelligence***,** 431-450.
MCCLUSKEY, T. 2000. Knowledge engineering for planning roadmap (unpublished work, University of Huddersfield).
MCCLUSKEY, T., CRESSWELL, S., RICHARDSON, N., SIMPSON, R. & WEST, M. M. 2008. An evaluation of Opmaker2. *The 27th Workshop of the UK Planning and Scheduling Special Interest Group, December 11-12th, 2008, Edinburgh.***,** 65-72.
MCCLUSKEY, T., FOX, M. & AYLETT, R. 2002a. Planform: An open environment for building planners. *PLANET Newsletter of the European Network of Excellence in AI Planning***,** 38-45.
MCCLUSKEY, T. L., CRESSWELL, S., RICHARDSON, N. E. & WEST, M. M. 2009. Automated acquisition of action knowledge with Opmaker2. *International Conference on Agents and Artificial Intelligence ICAART 2009: Agents and Artificial Intelligence* 137-150.
MCCLUSKEY, T. L. & PORTEOUS, J. M. 1997. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence,* 95**,** 1-65.
MCCLUSKEY, T. L., RICHARDSON, N. E. & SIMPSON, R. M. An Interactive Method for Inducing Operator Descriptions. AIPS, 2002b. 121-130.
MCCLUSKEY, T. L., VAQUERO, T. & VALLATI, M. 2016. Issues in Planning Domain Model Engineering.
MEHTA, N., TADEPALLI, P. & FERN, A. Autonomous learning of action models for planning. Advances in Neural Information Processing Systems, 2011. 2465-2473.
MENZE, M. & GEIGER, A. Object scene flow for autonomous vehicles. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015. 3061-3070.
MOURAO, K., PETRICK, R. P. & STEEDMAN, M. Learning action effects in partially observable domains. ECAI, 2010. 973-974.
MOURAO, K., ZETTLEMOYER, L. S., PETRICK, R. & STEEDMAN, M. 2012. Learning strips operators from noisy and incomplete observations. *arXiv preprint arXiv:1210.4889*.
MUGGLETON, S. Bayesian inductive logic programming. Proceedings of the seventh annual conference on Computational learning theory, 1994. ACM, 3-11.
MUKHERJI, P. & SCHUBERT, L. K. Discovering Planning Invariants as Anomalies in State Descriptions. ICAPS, 2005. 223-230.

MYERS, K. & WILKINS, D. 2001. PRS-CL: A Procedural Reasoning System. *User's Guide., SRI International, Center, Menlo Park, CA*.

NAREYEK, A., FREUDER, E. C., FOURER, R., GIUNCHIGLIA, E., GOLDMAN, R. P., KAUTZ, H., RINTANEN, J. & TATE, A. 2005. Constraints and AI planning. *Intelligent Systems, IEEE,* 20**,** 62-72.

NAU, D. S. 2007. Current trends in automated planning. *AI magazine,* 28**,** 43.

NAVEED, M., KITCHIN, D., CRAMPTON, A., CHRPA, L. & GREGORY, P. A Monte-Carlo path planner for dynamic and partially observable environments. Computational Intelligence and Games (CIG), 2012 IEEE Conference on, 2012. IEEE, 211-218.

NEWELL, A. & SIMON, H. 1963. GPS a program that simulates human thoughts in EA Feigenbaum and J. Feldman eds., Computer and Thoughts. McGraw-Hill, New-York.

NUNAMAKER JR, J. F., WEBER, E. S. & CHEN, M. 1989. Organizational crisis management systems: planning for intelligent action. *Journal of management information systems,* 5**,** 7-32.

ONTANÓN, S., BONNETTE, K., MAHINDRAKAR, P., GÓMEZ-MARTÍN, M. A., LONG, K., RADHAKRISHNAN, J., SHAH, R. & RAM, A. 2009. Learning from human demonstrations for real-time case-based planning. *CiteSeerX*.

PALAO, F., FDEZ-OLIVARES, J., CASTILLO, L. & GARCıA, O. 2011. An extended htn knowledge representation based on a graphical notation. *KEPS 2011***,** 126.

PAUL, G. & HELMERT, M. 2016. Optimal Solitaire Game Solutions using A Search and Deadlock Analysis. *Heuristics and Search for Domain-independent Planning (HSDIP)***,** 52.

PEDNAULT, E. P. 1988. Synthesizing plans that contain actions with context-dependent effects1. *Computational Intelligence,* 4**,** 356-372.

PETRES, C., PAILHAS, Y., PATRON, P., PETILLOT, Y., EVANS, J. & LANE, D. 2007. Path planning for autonomous underwater vehicles. *IEEE Transactions on Robotics,* 23**,** 331-341.

PLCH, T., CHOMUT, M., BROM, C. & BARTÁK, R. 2012. Inspect, Edit and Debug PDDL Documents: Simply and Efficiently with PDDL Studio. *and Exhibits***,** 15.

POMERLEAU, D. A. 1991. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation,* 3**,** 88-97.

PORTEOUS, J. & SEBASTIA, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research,* 22**,** 215-278.

RAILRESEARCHASSOSIATION 2012. Robotics research call information day.

REDINGTON, D. 1985. A Review of Knowledge Engineering and Expert Systems: Towards Expert Operators in Forth. *The Journal of Forth Applicationa and Research,* 4**,** 85-94.

REFANIDIS, I. & SEKALLARIOU, I. A Systematic and Complete Algorithm to Compute Higher Order Exclusion Relations. Proceeding of the ICAPS Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (COPLAS02009), 2009. 33-42.

REGER, G., BARRINGER, H. & RYDEHEARD, D. 2015. Automata-based pattern mining from imperfect traces. *ACM SIGSOFT Software Engineering Notes,* 40**,** 1-8.

RICHARDSON, N. E. 2008. *An operator induction tool supporting knowledge engineering in planning.* University of Huddersfield.

RICHTER, S. & WESTPHAL, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research,* 39**,** 127-177.

RINTANEN, J. An iterative algorithm for synthesizing invariants. AAAI/IAAI, 2000. 806-811.

RINTANEN, J. 2014. Madagascar: Scalable planning with sat. *Proceedings of the 8th International Planning Competition (IPC-2014)*.

RIVEST, R. L. & SCHAPIRE, R. E. 1993. Inference of finite automata using homing sequences. *Machine Learning: From Theory to Applications.* Springer.

RUSSELL, S. J., NORVIG, P., CANNY, J. F., MALIK, J. M. & EDWARDS, D. D. 2003. *Artificial intelligence: a modern approach*, Prentice hall Upper Saddle River.

SACERDOTI, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial intelligence,* 5**,** 115-135.

SACERDOTI, E. D. 1975. The nonlinear nature of plans. DTIC Document.
SAMUEL, A. L. 2000. Some studies in machine learning using the game of checkers. *IBM Journal of research and development,* 44**,** 206-226.
SAUER, J. Planning and Scheduling An Overview. Planen und Konfigurieren (PuK-2003), Proceedings des Workshops zur KI, 2003. Citeseer, 158-161.
SCHREIBER, G., WIELINGA, B., DE HOOG, R., AKKERMANS, H. & VAN DE VELDE, W. 1994. CommonKADS: A comprehensive methodology for KBS development. *IEEE expert,* 9**,** 28-37.
SHADBOLT, N. & MILTON, N. 1999. From knowledge engineering to knowledge management. *British Journal of Management,* 10**,** 309-322.
SHAH, M., CHRPA, L., JIMOH, F., KITCHIN, D., MCCLUSKEY, T., PARKINSON, S. & VALLATI, M. 2013a. Knowledge engineering tools in planning: State-of-the-art and future challenges. *Knowledge Engineering for Planning and Scheduling***,** 53.
SHAH, M. M. S., CHRPA, L., KITCHIN, D. E., MCCLUSKEY, T. L. & VALLATI, M. Exploring Knowledge Engineering Strategies in Designing and Modelling a Road Traffic Accident Management Domain. IJCAI, 2013b. 2373-2379.
SHAHAF, D. & AMIR, E. Learning partially observable action schemas. Proceedings of the National Conference on Artificial Intelligence, 2006. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 913.
SHEN, W.-M. 1994. Autonomous Learning from the Environmen. *In WH Freeman and Company.*
SHOEEB, S. 2012. Investigation into the Theoretical Properties of, and the Relationship Between, AI Planning Domain Models (unpublished Report, University of Huddersfield).
SHOEEB, S. & MCCLUSKEY, T. On comparing planning domain models. Proceedings of the Annual PLANSIG Conference, 2011. 92-94.
SIMPSON, R. M., KITCHIN, D. E. & MCCLUSKEY, T. 2007. Planning domain definition using GIPO. *Knowledge Engineering Review,* 22**,** 117-134.
SIPPER, M. & ELYASAF, A. 2014. Lunch isn't free---but cells are: evolving FreeCell players. *ACM SIGEVOlution,* 6**,** 2-10.
SKIBNIEWSKI, M. & GOLPARVAR-FARD, M. 2016. Toward a Science of Autonomy for Physical Systems: Construction. *arXiv preprint arXiv:1604.03563*.
STAFF, J. C. O. 2011. Joint Operation Planning.
STEFIK, M. 1981. Planning with constraints (MOLGEN: Part 1). *Artificial intelligence,* 16**,** 111-139.
STERRITT, R. & BUSTARD, D. Autonomic Computing-a means of achieving dependability? Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the, 2003. IEEE, 247-251.
STERRITT, R., GARRITY, G., HANNA, E. & O'HAGAN, P. Survivable security systems through autonomicity. Workshop on Radical Agent Concepts, 2005. Springer, 379-389.
STOKMAN, F. N. & DE VRIES, P. H. 1988. Structuring knowledge in a graph. *Human-Computer Interaction.* Springer.
TATE, A. 1998. Roots of SPAR—shared planning and activity representation. *The Knowledge Engineering Review,* 13**,** 121-128.
TATE, A., POLYAK, S. T. & JARVIS, P. 1998. *TF Method: An Initial Framework for Modelling and Analysing Planning Domains*, AIAI, University of Edinburgh.
TATE, A., WICKLER, G., MCCLUSKEY, L. & CHRPA, L. 2012. Machine Learning and Adaptation of Domain Models to Support Real Time Planning in Autonomous Systems. *HEdLAMP – Huddersfield + Edinburgh: Learning and Adaptation of Models for Planning, University of Edinburgh*.
TAYLOR, M. E. & STONE, P. 2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research,* 10**,** 1633-1685.
TRUSZKOWSKI, W., HALLOCK, H., ROUFF, C., KARLIN, J., RASH, J., HINCHEY, M. & STERRITT, R. 2009. *Autonomous and autonomic systems: With applications to NASA intelligent spacecraft operations and exploration systems*, Springer Science & Business Media.

TSOUMAKAS, G., VRAKAS, D., BASSILIADES, N. & VLAHAVAS, I. Lazy adaptive multicriteria planning. ECAI, 2004. 693.
VAQUERO, T., TONACO, R., COSTA, G., TONIDANDEL, F., SILVA, J. R. & BECK, J. C. itSIMPLE4. 0: Enhancing the modeling experience of planning problems. Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12)–System Demonstration, 2012. 11-14.
VAQUERO, T. S., ROMERO, V., TONIDANDEL, F. & SILVA, J. R. itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains. ICAPS, 2007. 336-343.
VAQUERO, T. S., SILVA, J. R. & BECK, J. C. 2011. A brief review of tools and methods for knowledge engineering for planning & scheduling. *KEPS 2011*, 7.
VELOSO, M., CARBONELL, J., PEREZ, A., BORRAJO, D., FINK, E. & BLYTHE, J. 1995. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental & Theoretical Artificial Intelligence,* 7, 81-120.
VELOSO, M. M. 1994. *Planning and learning by analogical reasoning*, Springer Science & Business Media.
VIDAL, V. 2011. YAHSP2: Keep it simple, stupid. *International Planning Competition 2011*, 83-90.
VIDAL, V. 2014. YAHSP3 and YAHSP3-MT in the 8th international planning competition. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 64-65.
VIEIRA, M. A. M. 2010. *PhD Thesis: The interplay between networks and robotics: networked robots and robotic routers.* University of Southern California.
VODRÁZKA, J. & CHRPA, L. Visual design of planning domains. Proceedings of ICAPS 2010 workshop on Scheduling and Knowledge Engineering for Planning and Scheduling (KEPS), 2010. 68-69.
WALDNER, J.-B. 1992. *CIM: principles of computer-integrated manufacturing*, John Wiley & Sons.
WALSH, T. J. & LITTMAN, M. L. Efficient Learning of Action Schemas and Web-Service Descriptions. AAAI, 2008. 714-719.
WANG, X. Learning by observation and practice: An incremental approach for planning operator acquisition. ICML, 1995. 549-557.
WATKINS, C. J. C. H. 1989. *PhD Thesis: Learning from delayed rewards.* University of Cambridge England.
WICKLER, G. 2011. Using planning domain features to facilitate knowledge engineering. *KEPS 2011*, 39.
WICKLER, G. 2013. Using Static Graphs in Planning Domains to Understand Domain Dynamics. *Knowledge Engineering for Planning and Scheduling*, 69.
WICKLER, G., CHRPA, L. & MCCLUSKEY, T. L. KEWI-A Knowledge Engineering Tool for Modelling AI Planning Tasks. KEOD, 2014. 36-47.
WINNER, E. & VELOSO, M. DISTILL: Learning domain-specific planners by example. ICML, 2003. 800-807.
WINNER, E. & VELOSO, M. LoopDISTILL: Learning looping domain-specific planners from example plans. International Conference on Automated Planning and Scheduling, Workshop on Artificial Intelligence Planning and Learning, 2007. Citeseer.
WU, K., YANG, Q. & JIANG, Y. 2005. Arms: Action-relation modelling system for learning action models. *CKE*, 50.
XING, Z., CHEN, Y. & ZHANG, W. Maxplan: Optimal planning by decomposed satisfiability and backward reduction. Proceedings of the Fifteenth International Planning Competition, International Conference on Automated Planning and Scheduling, 2006. 53-56.
YANG, Q., WU, K. & JIANG, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence,* 171, 107-143.
YOUNES, H. L. & LITTMAN, M. L. 2004. PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*.
ZHUO, H., YANG, Q., HU, D. H. & LI, L. 2008. Transferring knowledge from another domain for learning action models. *PRICAI 2008: Trends in Artificial Intelligence.* Springer.
ZHUO, H. H. Crowdsourced action-model acquisition for planning. Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015.

ZHUO, H. H., HU, D. H., HOGG, C., YANG, Q. & MUNOZ-AVILA, H. Learning HTN Method Preconditions and Action Models from Partial Observations. IJCAI, 2009a. 1804-1810.

ZHUO, H. H., HU, D. H., YANG, Q., MUNOZ-AVILA, H. & HOGG, C. Learning applicability conditions in AI planning from partial observations. Workshop on Learning Structural Knowledge From Observations at IJCAI, 2009b.

ZHUO, H. H. & KAMBHAMPATI, S. Action-model acquisition from noisy plan traces. Proceedings of the Twenty-Third international joint conference on Artificial Intelligence, 2013. AAAI Press, 2444-2450.

ZHUO, H. H., YANG, Q., HU, D. H. & LI, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence,* 174**,** 1540-1569.

ZHUOA, H. H., NGUYENB, T. & KAMBHAMPATIB, S. Refining incomplete planning domain models through plan traces. Proceedings of IJCAI, 2013.

ZIMMERMAN, T. & KAMBHAMPATI, S. 2003. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine,* 24**,** 73.

Appendices

Appendix A

This appendix contains PDDL files describing Freecell, BlocksWorld, ZenoTravel, Depots, Driver-Log, Briefcase, Elevator and Rover domains.

Benchmark: Freecell Domain

```
(define (domain freecell)
  (:requirements :strips :typing)
  (:types card num suit)
  (:predicates (on ?c1 - card ?c2 - card)
               (incell ?c - card)
               (clear ?c - card)
               (cellspace ?n - num)
               (colspace ?n - num)
               (home ?c - card)
               (bottomcol ?c - card)
               (canstack ?c1 - card ?c2 - card)
               (suit ?c - card ?s - suit)
               (value ?c - card ?v - num)
               (successor ?n1 - num ?n0 - num)
 )

  (:action move
          :parameters (?card ?oldcard ?newcard - card)
          :precondition (and (clear ?card)
                             (clear ?newcard)
                             (canstack ?card ?newcard)
                             (on ?card ?oldcard))
          :effect (and
                   (on ?card ?newcard)
                   (clear ?oldcard)
                   (not (on ?card ?oldcard))
                   (not (clear ?newcard))))

  (:action move-b
          :parameters (?card ?newcard - card ?cols ?ncols - num)
```

```
        :precondition (and (clear ?card)
                           (bottomcol ?card)
                           (clear ?newcard)
                           (canstack ?card ?newcard)
                           (colspace ?cols)
                           (successor ?ncols ?cols))
        :effect (and (on ?card ?newcard)
                     (colspace ?ncols)
                     (not (bottomcol ?card))
                     (not (clear ?newcard))
                     (not (colspace ?cols))))

(:action sendtofree
        :parameters (?card ?oldcard - card ?cells ?ncells - num)
        :precondition (and
                        (clear ?card)
                        (on ?card ?oldcard)
                        (cellspace ?cells)
                        (successor ?cells ?ncells))
        :effect (and
                  (incell ?card)
                  (clear ?oldcard)
                  (cellspace ?ncells)
                  (not (on ?card ?oldcard))
                  (not (clear ?card))
                  (not (cellspace ?cells))))

(:action sendtofree-b
        :parameters (?card - card ?cells ?ncells ?cols ?ncols - num)
        :precondition (and (clear ?card)
                           (bottomcol ?card)
                           (cellspace ?cells)
                           (successor ?cells ?ncells)
                           (colspace ?cols)
                           (successor ?ncols ?cols))
        :effect (and
                  (incell ?card)
                  (colspace ?ncols)
```

```
                (cellspace ?ncells)
                (not (bottomcol ?card))
                (not (clear ?card))
                (not (colspace ?cols))
                (not (cellspace ?cells))))

(:action sendtonewcol
        :parameters (?card ?oldcard - card ?cols ?ncols - num)
        :precondition (and
                        (clear ?card)
                        (colspace ?cols)
                        (successor ?cols ?ncols)
                        (on ?card ?oldcard))
        :effect (and
                (bottomcol ?card)
                (clear ?oldcard)
                (colspace ?ncols)
                (not (on ?card ?oldcard))
                (not (colspace ?cols))))

(:action sendtohome
        :parameters (?card ?oldcard - card ?suit - suit ?vcard - num ?homecard - card
?vhomecard - num)
        :precondition (and
                        (clear ?card)
                        (on ?card ?oldcard)
                        (home ?homecard)
                        (suit ?card ?suit)
                        (suit ?homecard ?suit)
                        (value ?card ?vcard)
                        (value ?homecard ?vhomecard)
                        (successor ?vcard ?vhomecard))
        :effect (and (home ?card)
                    (clear ?oldcard)
                    (not (on ?card ?oldcard))
                (not (home ?homecard))
                    (not (clear ?card))))
```

```
(:action sendtohome-b
        :parameters (?card - card ?suit - suit  ?vcard - num
                        ?homecard - card
                        ?vhomecard - num
                        ?cols ?ncols - num)
        :precondition (and (clear ?card)
                        (bottomcol ?card)
                        (home ?homecard)
                        (suit ?card ?suit)
                        (suit ?homecard ?suit)
                        (value ?card ?vcard)
                        (value ?homecard ?vhomecard)
                        (successor ?vcard ?vhomecard)
                        (colspace ?cols)
                        (successor ?ncols ?cols))
        :effect (and (home ?card)
                    (colspace ?ncols)
                    (not (home ?homecard))
                    (not (clear ?card))
                    (not (bottomcol ?card))
                    (not (colspace ?cols))))

(:action homefromfreecell
        :parameters (?card - card ?suit - suit ?vcard - num
                        ?homecard - card ?vhomecard - num
                        ?cells ?ncells - num)
   ;;Send a card home from a free cell.
        :precondition (and
                        (incell ?card)
                        (home ?homecard)
                        (suit ?card ?suit)
                        (suit ?homecard ?suit)
                        (value ?card ?vcard)
                        (value ?homecard ?vhomecard)
                        (successor ?vcard ?vhomecard)
                        (cellspace ?cells)
                        (successor ?ncells ?cells))
        :effect (and
```

```
                (home ?card)
                (cellspace ?ncells)
                (not (incell ?card))
                (not (cellspace ?cells))
                (not (home ?homecard))))

(:action colfromfreecell
        :parameters (?card ?newcard -  card ?cells ?ncells - num)
        :precondition (and  (incell ?card)
                            (clear ?newcard)
                            (canstack ?card ?newcard)
                            (cellspace ?cells)
                            (successor ?ncells ?cells))
        :effect (and (cellspace ?ncells)
                    (clear ?card)
                    (on ?card ?newcard)
                    (not (incell ?card))
                    (not (cellspace ?cells))
                    (not (clear ?newcard))))

(:action newcolfromfreecell
        :parameters (?card - card ?cols ?ncols ?cells ?ncells - num)
        :precondition (and (incell ?card)
                            (colspace ?cols)
                            (cellspace ?cells)
                            (successor ?cols ?ncols)
                            (successor ?ncells ?cells))
        :effect (and (bottomcol ?card)
                    (clear ?card)
                    (colspace ?ncols)
                    (cellspace ?ncells)
                    (not (incell ?card))
                    (not (colspace ?cols))
                    (not (cellspace ?cells)))))
```

LOCM: Freecell Domain

The auto-labelled predicates and their meaning
card_state0 - suit
card_state1 - home
card_state2 - in_home_cell_and_covered
card_state3 - in_home_cell_and_covered
card_state4 - in_col_and_covered
card_state5 - on_top_of_col
card_state6 - incell
card_state7 - in_home_cell_and_covered

Domain

```
(define
  (domain freecell)
  (:requirements :typing)
  (:types club n zero)
  (:predicates
    (club_state0 ?v1 - club)
    (club_state1 ?v1 - club ?v2 - club ?v3 - n)
    (club_state2 ?v1 - club)
    (club_state3 ?v1 - club)
    (club_state4 ?v1 - club)
    (club_state5 ?v1 - club)
    (club_state6 ?v1 - club)
    (club_state7 ?v1 - club)
    (n_state0 ?v1 - n)
    (zero_state0))

  (:action
  colfromfreecell
  :parameters
  (?Club1 - club ?Club2 - club ?N3 - n ?N4 - n)
  :precondition
  (and
    (zero_state0)
```

```
    (club_state6 ?Club1)
    (club_state5 ?Club2)
    (n_state0 ?N3)
    (n_state0 ?N4))

  :effect
  (and
    (club_state5 ?Club1)
    (not (club_state6 ?Club1))
    (club_state4 ?Club2)
    (not (club_state5 ?Club2)))
)

  (:action
  homefromfreecell
  :parameters
  (?Club1 - club ?Club2 - club ?N3 - n ?Club4 - club ?N5 - n ?N6 - n ?N7 - n)
  :precondition
  (and
    (zero_state0)
    (club_state6 ?Club1)
    (club_state0 ?Club2)
    (n_state0 ?N3)
    (club_state1 ?Club4 ?Club2 ?N5)
    (n_state0 ?N5)
    (n_state0 ?N6)
    (n_state0 ?N7))

  :effect
  (and
    (club_state1 ?Club1 ?Club2 ?N3)
    (not (club_state6 ?Club1))
    (club_state7 ?Club4)
    (not (club_state1 ?Club4 ?Club2 ?N5)))
)

  (:action
  move
```

```
  :parameters
  (?Club1 - club ?Club2 - club ?Club3 - club)
  :precondition
  (and
    (zero_state0)
    (club_state5 ?Club1)
    (club_state4 ?Club2)
    (club_state5 ?Club3))

  :effect
  (and
    (club_state5 ?Club2)
    (not (club_state4 ?Club2))
    (club_state4 ?Club3)
    (not (club_state5 ?Club3)))
)

  (:action
  move_b
  :parameters
  (?Club1 - club ?Club2 - club ?N3 - n ?N4 - n)
  :precondition
  (and
    (zero_state0)
    (club_state5 ?Club1)
    (club_state5 ?Club2)
    (n_state0 ?N3)
    (n_state0 ?N4))

  :effect
  (and
    (club_state4 ?Club2)
    (not (club_state5 ?Club2)))
)

  (:action
  newcolfromfreecell
  :parameters
```

```
  (?Club1 - club ?N2 - n ?N3 - n)
  :precondition
  (and
    (zero_state0)
    (club_state6 ?Club1)
    (n_state0 ?N2)
    (n_state0 ?N3))

  :effect
  (and
    (club_state5 ?Club1)
    (not (club_state6 ?Club1)))
)

  (:action
  sendtofree
  :parameters
  (?Club1 - club ?Club2 - club ?N3 - n ?N4 - n)
  :precondition
  (and
    (zero_state0)
    (club_state5 ?Club1)
    (club_state4 ?Club2)
    (n_state0 ?N3)
    (n_state0 ?N4))

  :effect
  (and
    (club_state6 ?Club1)
    (not (club_state5 ?Club1))
    (club_state5 ?Club2)
    (not (club_state4 ?Club2)))
)

  (:action
  sendtofree_b
  :parameters
  (?Club1 - club ?N2 - n ?N3 - n)
```

```
  :precondition
  (and
    (zero_state0)
    (club_state5 ?Club1)
    (n_state0 ?N2)
    (n_state0 ?N3))

  :effect
  (and
    (club_state6 ?Club1)
    (not (club_state5 ?Club1)))
)

  (:action
  sendtohome
  :parameters
  (?Club1 - club ?Club4 - club ?Club2 - club ?N3 - n ?Club5 - club ?N6 - n)
  :precondition
  (and
    (zero_state0)
    (club_state5 ?Club1)
    (club_state4 ?Club4)
    (club_state0 ?Club2)
    (n_state0 ?N3)
    (club_state1 ?Club5 ?Club2 ?N6)
    (n_state0 ?N6))

  :effect
  (and
    (club_state1 ?Club1 ?Club2 ?N3)
    (not (club_state5 ?Club1))
    (club_state5 ?Club4)
    (not (club_state4 ?Club4))
    (club_state2 ?Club5)
    (not (club_state1 ?Club5 ?Club2 ?N6)))
)

  (:action
```

```
  sendtohome_b
  :parameters
  (?Club1 - club ?Club2 - club ?N3 - n ?Club4 - club ?N5 - n ?N6 - n ?N7 - n)
  :precondition
  (and
    (zero_state0)
    (club_state5 ?Club1)
    (club_state0 ?Club2)
    (n_state0 ?N3)
    (club_state1 ?Club4 ?Club2 ?N5)
    (n_state0 ?N5)
    (n_state0 ?N6)
    (n_state0 ?N7))

  :effect
  (and
    (club_state1 ?Club1 ?Club2 ?N3)
    (not (club_state5 ?Club1))
    (club_state3 ?Club4)
    (not (club_state1 ?Club4 ?Club2 ?N5)))
)

  (:action
  sendtonewcol
  :parameters
  (?Club1 - club ?Club2 - club ?N3 - n ?N4 - n)
  :precondition
  (and
    (zero_state0)
    (club_state5 ?Club1)
    (club_state4 ?Club2)
    (n_state0 ?N3)
    (n_state0 ?N4))

  :effect
  (and
    (club_state5 ?Club2)
    (not (club_state4 ?Club2)))
```

))

LOCM: Freecell Problem Instance

A Problem instance generated by LOCM against the following Plan

Plan:

sequence_task(20,[sendtofree(spade2,spade4,n4,n3),sendtofree(heart8,diamond7,n3,n2),sendtofree(club8,diamond4,n2,n1),sendtofree(spade9,heart7,n1,n0),move(heart7,diamonda,spade8),sendtohome(diamonda,spade3,diamond,n1,diamond0,n0)],[],[]).

Problem:

```
(define
  (problem task_20)
  (:domain freecell)
  (:objects club8 diamond diamond0 diamond4 diamond7 diamonda heart7 heart8 spade2
spade3 spade4 spade8 spade9 - card n0 n1 n2 n3 n4 - num)
  (:init
    (zero_state0)
    (card _state0 diamond)
    (card _state1 diamond0 diamond n0)
    (card _state4 diamond4)
    (card _state4 diamond7)
    (card _state4 diamonda)
    (card _state4 heart7)
    (card _state4 spade3)
    (card _state4 spade4)
    (card _state5 club8)
    (card _state5 heart8)
    (card _state5 spade2)
    (card _state5 spade8)
    (card _state5 spade9)
    (num _state0 n0)
    (num _state0 n1)
    (num _state0 n2)
    (num _state0 n3)
    (num _state0 n4))
  (:goal
    (and
    (zero_state0)
    (card _state0 diamond)
    (card _state1 diamonda diamond n1)
    (card _state2 diamond0)
```

(card _state4 spade8)

(card _state5 spade4)

(card _state5 diamond4)

(card _state5 diamond7)

(card _state5 heart7)

(card _state5 spade3)

(card _state6 club8)

(card _state6 heart8)

(card _state6 spade2)

(card _state6 spade9)

(num _state0 n0)

(num _state0 n1)

(num _state0 n2)

(num _state0 n3)

(num _state0 n4))))

Appendix B

B-1. Result of type Fuel in Donate operator of Mprime Domain.

In the following figure, the colour scheme in arguments indicate the learning of same coloured static fact in the operator definition. The corresponding ASCoL results are also provided in the same colours after the figure.

```
(:action donate
     :parameters (?l1 ?l2 - location ?f11 ?f12 ?f13 ?f21 ?f22 - fuel)
     :precondition (and (not-equal ?l1 ?l2)
                        (has-fuel ?l1 ?f11)
                        (fuel-neighbor ?f12 ?f11)
                        (fuel-neighbor ?f13 ?f12)
                        (has-fuel ?l2 ?f21)
                        (fuel-neighbor ?f21 ?f22))
  :effect (and (not (has-fuel ?l1 ?f11))
               (has-fuel ?l1 ?f12)
               (not (has-fuel ?l2 ?f21))
               (has-fuel ?l2 ?f22))
)
```

Figure B 1: operator *Donate* from Mprime Domain

donate

3

4

o(f4,f3) o(f3,f2)

o(f2,f1)

Linear type - add o(X,Y) to operator

donate
3
5
o(f4,f2) o(f3,f1)
o(f2,f0)
Partial order type

donate
3
6
o(f4,f0) o(f3,f0)
o(f2,f0)
Partial order type

donate
3
7
o(f4,f1) o(f3,f1)
o(f2,f1)
Partial order type

donate
4
5
o(f3,f2) o(f2,f1)
o(f1,f0)
Linear type - add o(X,Y) to operator

donate
4
6

o(f3,f0) o(f2,f0)

o(f1,f0)

Partial order type

donate

4

7

o(f3,f1) o(f2,f1)

o(f1,f1)

Partial order type

donate

5

6

o(f2,f0) o(f1,f0)

o(f0,f0)

Partial order type

donate

5

7

o(f2,f1) o(f1,f1)

o(f0,f1)

Partial order type

donate

6

7

o(f0,f1)

Linear type - add o(X,Y) to operator

B-2. Result of Unload, Load and Buy operators in TPP Domain

In the following figure (B 2, B 3, B 4), the colour scheme in arguments indicate the learning of same coloured static fact in the operator definition. The corresponding ASCoL results are also provided in the same colours after the figure.

```
(:action unload
        :parameters (?g - goods ?t - truck ?d - depot ?l1 ?l2 ?l3 ?l4 - level)
        :precondition (and (at ?t ?d)
                           (loaded ?g ?t ?l2)(stored ?g ?l3)
                           (next ?l2 ?l1) (next ?l4 ?l3))
        :effect (and (loaded ?g ?t ?l1)
                     (not (loaded ?g ?t ?l2))(stored ?g ?l4)
                     (not (stored ?g ?l3)))
)
```

Figure B 2: Operator *unload* from the benchmark TPP domain

```
(:action load
        :parameters (?g - goods ?t - truck ?m - market ?l1 ?l2 ?l3 ?l4 - level)
         :precondition (and (at ?t ?m)
                            (loaded ?g ?t ?l3)(ready-to-load ?g ?m ?l2)
                            (next ?l2 ?l1) (next ?l4 ?l3))
        :effect (and (loaded ?g ?t ?l4)
                     (not (loaded ?g ?t ?l3)) (ready-to-load ?g ?m ?l1)
                     (not (ready-to-load ?g ?m ?l2)))
)
```

Figure B 3: Operator *load* from the benchmark TPP domain

```
(:action buy
        :parameters (?t - truck ?g - goods ?m - market ?l1 ?l2 ?l3 ?l4 - level)
        :precondition (and (at ?t ?m)
                           (on-sale ?g ?m ?l2) (ready-to-load ?g ?m ?l3)
                           (next ?l2 ?l1) (next ?l4 ?l3))
         :effect (and (on-sale ?g ?m ?l1)
                      (not (on-sale ?g ?m ?l2))(ready-to-load ?g ?m ?l4)
                      (not (ready-to-load ?g ?m ?l3)))
)
```

Figure B 4: Operator *buy* from the benchmark TPP domain

```
unload
4
5
o(level2,level3)    o(level1,level2)
o(level0,level1)
Linear type - add o(X,Y) to operator
```

unload
4
6
o(level2,level6) o(level1,level4)
o(level0,level8) o(level0,level7)
o(level0,level6) o(level0,level5)
o(level0,level4) o(level1,level1)
o(level2,level0) o(level2,level1)
o(level1,level2) o(level1,level0)
o(level0,level3) o(level0,level2)
o(level0,level1) o(level0,level0)
Objects not ordered

unload
4
7
o(level2,level7) o(level1,level5)
o(level0,level9) o(level0,level8)
o(level0,level7) o(level0,level6)
o(level0,level5) o(level1,level2)
o(level2,level1) o(level2,level2)
o(level1,level3) o(level1,level1)
o(level0,level4) o(level0,level3)
o(level0,level2) o(level0,level1)
Objects not ordered

unload
5
6
o(level3,level6) o(level2,level4)
o(level1,level8) o(level1,level7)
o(level1,level6) o(level1,level5)
o(level1,level4) o(level2,level1)
o(level3,level0) o(level3,level1)
o(level2,level2) o(level2,level0)
o(level1,level3) o(level1,level2)
o(level1,level1) o(level1,level0)
Objects not ordered

```
unload
5
7
o(level3,level7)    o(level2,level5)
o(level1,level9)    o(level1,level8)
o(level1,level7)    o(level1,level6)
o(level1,level5)    o(level2,level2)
o(level3,level1)    o(level3,level2)
o(level2,level3)    o(level2,level1)
o(level1,level4)    o(level1,level3)
o(level1,level2)    o(level1,level1)
Objects not ordered

unload
6
7
o(level8,level9)    o(level7,level8)
o(level6,level7)    o(level5,level6)
o(level4,level5)    o(level3,level4)
o(level2,level3)    o(level1,level2)
o(level0,level1)
Linear type - add o(X,Y) to operator


load
4
5
o(level2,level3)    o(level3,level4)
o(level4,level5)    o(level5,level6)
o(level1,level2)    o(level0,level1)
Linear type - add o(X,Y) to operator


load
4
6
o(level2,level1)    o(level3,level0)
```

o(level4,level0) o(level5,level0)
o(level1,level1) o(level0,level2)
o(level1,level0) o(level0,level1)
o(level0,level0)
Objects not ordered

load
4
7
o(level2,level2) o(level3,level1)
o(level4,level1) o(level5,level1)
o(level1,level2) o(level0,level3)
o(level1,level1) o(level0,level2)
o(level0,level1)
Objects not ordered

load
5
6
o(level3,level1) o(level4,level0)
o(level5,level0) o(level6,level0)
o(level2,level1) o(level1,level2)
o(level2,level0) o(level1,level1)
o(level1,level0)
Objects not ordered

load
5
7
o(level3,level2) o(level4,level1)
o(level5,level1) o(level6,level1)
o(level2,level2) o(level1,level3)
o(level2,level1) o(level1,level2)
o(level1,level1)
Objects not ordered

load
6
7
o(level2,level3) o(level1,level2)
o(level0,level1)
Linear type - add o(X,Y) to operator

buy
4
5
o(level3,level4) o(level4,level5)
o(level2,level3) o(level0,level1)
o(level1,level2)
Linear type - add o(X,Y) to operator

buy
4
6
o(level3,level0) o(level4,level0)
o(level3,level1) o(level4,level1)
o(level1,level1) o(level2,level0)
o(level0,level1) o(level0,level0)
o(level1,level0)
Objects not ordered

buy
4
7
o(level3,level1) o(level4,level1)
o(level3,level2) o(level4,level2)
o(level1,level2) o(level2,level1)
o(level0,level2) o(level0,level1)
o(level1,level1)
Objects not ordered

buy

5
6
o(level4,level0) o(level5,level0)
o(level4,level1) o(level5,level1)
o(level2,level1) o(level3,level0)
o(level1,level1) o(level1,level0)
o(level2,level0)
Objects not ordered

buy
5
7
o(level4,level1) o(level5,level1)
o(level4,level2) o(level5,level2)
o(level2,level2) o(level3,level1)
o(level1,level2) o(level1,level1)
o(level2,level1)
Objects not ordered

buy
6
7
o(level1,level2) o(level0,level1)
Linear type - add o(X,Y) to operator