REASONING AND PLANNING WITH INCOMPLETE INFORMATION IN THE
PRESENCE OF STATIC CAUSAL LAWS

BY

TU HUY PHAN

A dissertation submitted to the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy

Subject: Computer Science

New Mexico State University
Las Cruces, New Mexico
June 2007

Copyright © 2007 by Tu Huy Phan
“Reasoning and Planning with Incomplete Information in the Presence of Static Causal Laws,” a dissertation prepared by Tu Huy Phan in partial fulfillment of the requirements for the degree, Doctor of Philosophy, has been approved and accepted by the following:

Linda Lacey
Dean of the Graduate School

Son Cao Tran
Chair of the Examining Committee

Date

Committee in charge:

Dr. Son Cao Tran, Chair
Dr. Enrico Pontelli
Dr. Hing Leung
Dr. Ou Ma
DEDICATION

This dissertation is dedicated to my wife, my daughter, and my parents.
I would like to express my deep and sincere thanks to my advisor as well as my mentor, Dr. Son Cao Tran, for his valuable support and guidance. During the five years of my doctoral study at New Mexico State University, he has shared with me not only his knowledge and insight in the field of reasoning about action and change, but also his real-life experience. He is the person who inspired my passion for the field, without which I could not have this dissertation done. Moreover, had it not been for his suggestions, comments, ideas along with his quick feedback, this work would not be possible. My special thanks are extended to Dr. Enrico Pontelli for his favorable support and fruitful discussions on my research.

My many thanks also go to Dr. Chitta Baral for giving me a chance to work with him. His state-of-the-art thinking and papers have influenced my research. I would be especially grateful to Dr. Michael Gelfond for his creative ideas and beneficial assistance which have enabled me to better explore the field. He has taught me how to approach the depth and the breadth of a problem, even a small one, whether throughout face-to-face conversations or throughout emails.

I would like to thank the other two committee members, Dr. Hing Leung and Dr. Ou Ma, for their time and patience.

I would also like to thank Dr. AnHai Doan who encouraged me to study in the United States.
I am deeply indebted to my wife and my lovely little daughter, my parents and my sisters whose love, support, and encouragement lie behind my efforts. They have walked with me every step of my study.

Furthermore, I wish to thank Xin Zhang, Yu Pan, Ricardo Morales, Son To, and Vien Tran for their collaboration. I appreciate the help of Computer Science Department and all the folks at the Knowledge Representation, Logic, and Advanced Programming Laboratory (KLAP) where I had the opportunity to get access to wonderful facilities and to exchange ideas with great minds throughout weekly seminars.

Finally, I wish to thank all of my friends at New Mexico State University who have created a great community, which I am a part of, for the joyful days I have had in Las Cruces.
VITA


1999–2002  Software and Telecoms Engineer, Research Institute of Post and Telecommunication, Vietnam

2002–2007  Graduate Assistant, Department of Computer Science, New Mexico State University

2002–2007  Ph.D., Computer Science, New Mexico State University

Professional Societies

Member of American Association for Artificial Intelligence

Member of Knowledge Representation, Logic, and Advanced Programming Laboratory

Publications


**Field of Study**

Major field: Computer Science

Reasoning about Action and Change, Planning
ABSTRACT

REASONING AND PLANNING WITH INCOMPLETE INFORMATION IN THE PRESENCE OF STATIC CAUSAL LAWS

BY

TU HUY PHAN

Doctor of Philosophy

New Mexico State University

Las Cruces, New Mexico, 2007

Dr. Son Cao Tran, Chair

This dissertation extends the framework of the 0-approximation to handle domains with static causal laws. It is divided into three parts. In the first part, we present a sufficient condition for the completeness of the 0-approximation and propose a new method to achieve complete reasoning and planning with the 0-approximation. In the second part, we tackle the problem of designing approximations for domains with static causal laws. Specifically, we present two different approaches to the problem which result in four different approximations. In the last part, we extend the approximations to incorporate sensing actions. In each part, the results are evaluated through the development of conformant or conditional planner(s).
# TABLE OF CONTENTS

LIST OF TABLES ....................................................... xvii
LIST OF FIGURES ...................................................... xviii

1 INTRODUCTION ....................................................... 1
   1.1 Situation Calculus and The Frame Problem ....................... 1
   1.2 Other Formalisms ............................................... 4
   1.3 Planning .................................................... 6
   1.4 Domain Constraints and The Ramification Problem ............. 8
   1.5 Incomplete Information and Sensing Actions .................. 12
   1.6 Outline Of Dissertation and Main Contributions .............. 14

2 BACKGROUND ....................................................... 20
   2.1 Action Description Language $\mathcal{AL}$ ........................ 20
       2.1.1 Syntax ............................................... 20
       2.1.2 Semantics ........................................... 23
       2.1.3 The Entailment Relationship $\models$ ...................... 31
   2.2 Reasoning With Incomplete Information ........................ 34
       2.2.1 The Possible World Approach ............................ 36
       2.2.2 The 0-Approximation Approach .......................... 40
   2.3 Planning with Incomplete Information .......................... 45
   2.4 Logic Programming Under Answer Set Semantics ................ 47
       2.4.1 General Logic Programs ................................ 48
       2.4.2 Extended General Logic Programs ......................... 50
       2.4.3 Disjunctive Logic Programs ............................ 52
2.4.4 Splitting Set and Splitting Sequence Theorems ........................... 54
2.5 Summary .................................................................................. 57

3 COMPLETENESS OF THE 0-APPROXIMATION ................................. 59
3.1 Introduction ............................................................................. 59
3.2 A Sufficient Condition for the Completeness of $|=^0$ .................... 65
3.3 Complete Reasoning Using $|=^0$ ............................................. 74
3.4 Application to Conformant Planning .......................................... 79
  3.4.1 The Conformant Planning System CPA$^+$ .................................. 79
  3.4.2 Experiments ...................................................................... 83
3.5 Discussion and Related Work .................................................... 89
3.6 Summary .................................................................................. 93

4 APPROXIMATIONS OF DOMAIN DESCRIPTIONS WITH STATIC CAUSAL LAWS ................................................................. 94
4.1 Introduction ............................................................................. 94
4.2 Approximations of $\mathcal{AL}$ Domain Descriptions ....................... 97
4.3 The Possibly-Holds Approach ................................................... 104
  4.3.1 An Algorithm for Computing $\Phi^{ph}$ .................................... 107
  4.3.2 Examples .......................................................................... 110
  4.3.3 An Enhanced Version of $T^{ph}$ ........................................... 116
4.4 The Possibly-Changes Approach ............................................... 122
  4.4.1 An Algorithm For Computing $\Phi^{pc}$ .................................... 124
  4.4.2 Examples .......................................................................... 126
  4.4.3 An Enhanced Version of $T^{pc}$ ........................................... 130

xi
4.5 Properties Of the Approximations ................................................. 135
  4.5.1 The Relationship between the Approximations ......................... 136
  4.5.2 A Sufficient Condition For the Completeness of \( \models^A \) ........ 141
  4.5.3 Complexity of Planning with respect to \( \models^A \) .................... 145
4.6 An Approximation Based Conformant Planner in Logic Programming . 146
  4.6.1 Soundness of \( \text{CPASP} \) .................................................. 153
  4.6.2 Completeness of \( \text{CPASP} \) .............................................. 154
4.7 A C++ Implementation of \( \text{CPASP} \) ......................................... 155
4.8 Experiments ................................................................. 156
  4.8.1 Performance of \( \text{CPASP} \) .................................................. 156
    4.8.1.1 Benchmarks .......................................................... 157
    4.8.1.2 Performance ......................................................... 159
  4.8.2 Performance of \( \text{CPA} \) .................................................... 163
4.9 Discussion ................................................................. 168
  4.9.1 Handling Disjunctive Information about the Initial State in \( \text{CPASP} \) .................................................................. 168
  4.9.2 Implementation: Imperative Language vs Logic Programming ........ 170
4.10 Related Work ............................................................. 173
  4.10.1 Approximate Reasoning about Action and Change ...................... 173
  4.10.2 Conformant Planning ...................................................... 175
4.11 Summary ................................................................. 177
5 INCORPORATING SENSING ACTIONS ........................................ 180
A.3 \textit{C-PLAN} \hspace{1cm} 239
A.4 \textit{DLV}^K \hspace{1cm} 239
A.5 \textit{KACMBP} \hspace{1cm} 240
A.6 \textit{MBP} \hspace{1cm} 240
A.7 \textit{POND} \hspace{1cm} 240
A.8 \textit{SGP (Sensory Graph Plan)} \hspace{1cm} 241

B \hspace{0.5cm} \textbf{PROOFS OF THE RESULTS IN CHAPTER 2} \hspace{1cm} 242
B.1 Proof of Proposition 2.1 \hspace{1cm} 242
B.2 Proof of Proposition 2.2 \hspace{1cm} 243
B.3 Proof of Theorem 2.1 \hspace{1cm} 245

C \hspace{0.5cm} \textbf{PROOFS OF THE RESULTS IN CHAPTER 3} \hspace{1cm} 247
C.1 Proofs of the Results in Section 3.2 \hspace{1cm} 249
C.1.1 Proof of Proposition 3.1 \hspace{1cm} 249
C.1.2 Proof of Proposition 3.2 \hspace{1cm} 250
C.1.3 Proof of Proposition 3.3 \hspace{1cm} 251
C.1.4 Proof of Theorem 3.1 \hspace{1cm} 253
C.2 Proofs of the Results in Section 3.3 \hspace{1cm} 254
C.2.1 Proof of Theorem 3.2 \hspace{1cm} 254
C.2.2 Proof of Proposition 3.4 \hspace{1cm} 255
C.2.3 Proof of Proposition 3.5 \hspace{1cm} 256

D \hspace{0.5cm} \textbf{PROOFS OF THE RESULTS IN CHAPTER 4} \hspace{1cm} 257
D.1 Proofs of the Results in Section 4.2 \hspace{1cm} 259
D.1.1 Proof of Theorem 4.1 \hspace{1cm} 259

xiv
F.1.1 Proof of Proposition 5.1 ......................... 308
F.1.2 Proof of Proposition 5.2 ......................... 309
F.1.3 Proof of Proposition 5.3 ......................... 310
F.1.4 Proof of Theorem 5.1 ............................ 310
F.2 Proofs of Results in Section 5.4 .................. 311
  F.2.1 Proof of Theorem 5.2 ........................... 312
  F.2.2 Proof of Proposition 5.4 ....................... 338
  F.2.3 Proof of Theorem 5.3 ........................... 341
G A SAMPLE ENCODING OF ASCP ...................... 357
  G.1 Input Domain ..................................... 357
  G.2 The Output of smodels ......................... 364
  G.3 The Output of cmodels ......................... 365
REFERENCES ........................................... 366
## LIST OF TABLES

3.1 Performance of $CPA^+$ on the Logistics domain ........................................... 86
3.2 Performance of $CPA^+$ on the Ring domain .................................................. 87
3.3 Performance of $CPA^+$ on the BTUC domain ............................................... 88
3.4 Performance of $CPA^+$ on the Cleaner domain ............................................. 89

4.1 Performance of $CPASP$ on the BT and BTC domains ..................................... 159
4.2 Performance of $CPASP$ on the Ring domains ............................................... 160
4.3 Performance of $CPASP$ on the Domino domains ......................................... 160
4.4 Performance of $CPASP$ on the Gaspipe domain ......................................... 161
4.5 Performance of $CPASP$ on the Cleaner domain .......................................... 161
4.6 Performance of $CPASP$ on the BT$^p$ and BTC$^p$ domains ............................. 162
4.7 Performance of $CPASP$ on the Gaspipe$^p$ domain ..................................... 162
4.8 Performance of $CPASP$ on the Cleaner$^p$ domain ..................................... 163
4.9 Performance of CPA on the Logistics domain .............................................. 164
4.10 Performance of CPA on the Ring domain .................................................. 164
4.11 Performance of CPA on the BTUC domain ............................................... 165
4.12 Performance of CPA on the Cleaner domain ............................................. 166
4.13 Performance of CPA on the Domino domain ............................................. 167

5.1 Performance of ASCP on the Bomb domains ............................................... 227
5.2 Performance of ASCP on the Medicate and Sick domains ................................. 228
5.3 Performance of ASCP on the Ring domain .................................................. 229
5.4 Performance of ASCP on the Domino domain ............................................. 229
## LIST OF FIGURES

2.1 Description of the bomb-in-the-toilet domain, $D_{2.1}$ .......................... 22
2.2 The possible world approach ................................................................. 37
2.3 The 0-approximation approach .............................................................. 41

3.1 The possible world approach ................................................................. 61
3.2 The 0-approximation approach .............................................................. 62
3.3 The 0-approximation approach with sets of partial states ................. 63
3.4 An algorithm for computing a decisive set ........................................... 77
3.5 Encoding of the bomb-in-the-toilet domain .......................................... 80
3.6 The search algorithm of CPAM$^+$ ......................................................... 82
3.7 Partitioning the initial belief partial state .............................................. 82
3.8 Computing the successor partial state .................................................... 83

4.1 Computing the closure of a set of fluent literals ................................... 107
4.2 An algorithm for computing $\Phi^{ph}$ ..................................................... 108
4.3 An algorithm for computing $\Phi^{pc}(a, \delta)$ .......................................... 125
4.4 The search algorithm of CPAM ............................................................. 156

5.1 Sample plan trees .......................................................... .......................... 202
5.2 Possible mappings for the trees in Figure 5.1 ........................................ 203
5.3 Grid representation of conditional plans ............................................... 204

D.1 The relationship between subprograms of $\pi_h(P)$ ............................. 290

F.1 The relationships between subprograms of $\pi_{h,w}(P)$ ....................... 322
CHAPTER 1
INTRODUCTION

Reasoning about action and change (RAC) has become an active area of research since the early days of Artificial Intelligence (AI). In essence, it can be understood as the field of automated reasoning about dynamic domains whose states change due to the execution of actions. The main goal of this dissertation is to develop an efficient method for reasoning and planning with incomplete information in the presence of static causal laws.

1.1 Situation Calculus and The Frame Problem

One of the first attempts to formalize action and change was situation calculus due to McCarthy and Hayes [94, 95, 99]. Central to situation calculus are the concepts of situations, fluents, and actions. In essence, a situation can be viewed as a snapshot of the domain at a specific time and a fluent describes a property of the domain whose value changes over time. To assert that a fluent $F$ holds in a situation $S$, situation calculus uses the predicate $\text{holds}(F, S)$. In addition, the function $\text{do}(A, S)$ is used to denote the successor situation of a situation $S$ after the execution of an action $A$. An action theory in situation calculus is a first order logic theory that includes (i) axioms to describe the initial situation, (ii) precondition axioms to describe the conditions under which actions are executable, and (iii) effect axioms to describe effects of actions.
In their 1969 landmark paper “Some Philosophical Problems from the Standpoint of Artificial Intelligence” [99], McCarthy and Hayes noticed that in order for an action theory to yield correct results, in addition to precondition and effect axioms, the theory must include other axioms, which they called frame axioms, to describe what remains unchanged after the execution of an action. The problem of representing what is not affected by an action is known as the frame problem and it became one of the most challenging problems in AI for several decades. Shanahan [120] discussed that a good solution to the frame problem should meet three criteria: (1) representational parsimony, (2) expressive flexibility, and (3) elaboration tolerance. The first criterion states that any representation should be compact, proportional to the complexity (or size) of the domain; a good estimate of the size of the domain might be the number of fluents plus the number of actions. According to this criterion, using frame axioms to represent unchanged fluents in situation calculus is not a good solution since the number of frame axioms is roughly the product of the number of actions and the number of fluents. The second criterion says that any solution should be able to represent complicated domains, i.e., domains with features like domain constraints, concurrent and non-deterministic actions, multi-valued objects, continuous change, multi-agents, etc., which are ubiquitous in many dynamic domains. The criterion of elaboration tolerance, a term coined by McCarthy [98], implies that any formalism should be easily expanded in the sense that adding new knowledge to the representation should not require too much effort.
Up to date, many solutions to the frame problem have been proposed and their success varies (see [105, 120] for a detailed discussion about advantages and disadvantages of the existing solutions to the frame problem). In general, these solutions can be divided into two main groups: monotonic solutions and non-monotonic solutions.

Toward a monotonic solution to the frame problem, Haas [57] proposed so-called explanation closure axioms and this approach was later elaborated by Schubert in [119]. Reiter [114] then observed that effect axioms and explanation closure axioms can combined together into successor state axioms to make the representation more compact.

Some other researchers believe that the key idea to deal with the frame problem is to formalize the following commonsense law of inertia:

**Normally, given any action [or event type] and any fluent, the action does not affect the fluent ([99]).**

At first glance, non-monotonic logics\(^1\) such as McCarthy’s circumscription [96], Reiter’s default logic [113], or Moore’s auto-epistemic logic [103] seem to be adequate for formalizing this law. In [97], McCarthy applied the theory of circumscription to minimize fluents that are needed to change. Specifically, he introduced the predicate \(ab(A, F, S)\) to denote whether the value of fluent \(F\) changes after the execution of the action \(A\) in the situation \(S\) and then circumscribed the predicate \(ab(A, F, S)\), while letting \(holds\) to vary. Unfortunately, Hanks and McDermott [58] found an example –

\(^1\)The term non-monotonic logic is referred to formal frameworks for capturing and representing defeasible (sometimes called default) inferences
the famous “Yale shooting scenario” – which demonstrates that this approach yields unintuitive results.

Some authors, for example, Kautz [66], Lifschitz [77], and Shoham [122], followed the chronological minimization approach where changes are postponed until as late as possible. Although this approach can solve the Yale shooting problem correctly, Kautz [66] found a counter-intuitive example, called the “stolen car scenario”.

Lifschitz [78] introduced two new predicates causes and precond to describe the effects and the preconditions of actions respectively. To solve the frame problem, he suggested that causes and precond should be minimized in parallel, allowing holds to vary. This approach is known as causal minimization and a similar solution can be found in [60].

Baker [6] proposed another non-monotonic solution to the frame problem, known as state-based minimization. Unlike naive minimization which allows holds to vary, state-based minimization allows the do function to vary. Furthermore, an axiom is added to guarantee the existence of a situation for every legitimate combination of fluents.

1.2 Other Formalisms

In addition to situation calculus, two other formalisms that have influenced the research in reasoning about action and change are event calculus, and fluent calculus. Event calculus was first introduced, in the form of a logic program, by Kowalski and
Sergot in [72]. Although originally intended for database updates and narrative understanding [71, 72], event calculus has been used in many other reasoning tasks. Fluent calculus [138, 139, 141] was developed by Thieltscher in an attempt to address the frame problem in situation calculus. It is very similar to situation calculus except that a situation is related to the state of the domain in that situation, which is in turn represented as a concatenation of fluent terms.

Nowadays there has been an increasing interest in using action description languages as a formalism for reasoning about action and change because of their flexible, declarative syntax and transition-based semantics. The first language of this family is the action description language $A$ by Gelfond and Lifschitz [49]. The semantics of an $A$ domain description is given in terms of a transition diagram between states where each transition $\langle s, a, s' \rangle$ represents the fact that $s'$ is a possible successor state of the state $s$ as the result of execution of the action $a$. Since $A$ was introduced, several dialects of $A$ have been proposed to address various limitations of this language. For instance, [53] extended $A$ by adding domain constraints, non-propositional (sometimes referred to as multi-valued) fluents, nondeterministic actions and actions with indirect effects. [89, 126] incorporated sensing actions into $A$. [10] takes into account concurrent actions. [12] addresses observations and hypotheses. [129] extended $A$ to handle actions with duration. Recently, [63] generalized $A$ by allowing erroneous beliefs and non-Markovian belief change. Action description languages have been also applied in many real world applications, for example, in modelling molecular interactions in cells.
of living organisms [142, 143], or in verifying plans concerning the reaction control system of space shuttles [148].

1.3 Planning

Planning is one of the main applications of reasoning about action and change. In classical meaning, a planning problem is the task of choosing a sequence of actions, called solution, whose execution transforms the domain from one state into another state that satisfies a predetermined goal.

A straightforward (but efficient) method in planning is to formulate a planning problem as the search problem where nodes represent states of the domain and arcs correspond to transitions of the domain. The start node is the initial state of the domain and a goal node represents a state that satisfies the goal. To find a solution, search algorithms such as $A^*$, best-first search, hill-climbing, etc. can be applied and search direction can be either forward or backward. Examples of successful planning systems following this idea are HSP [21, 22] by Bonet, Loerincs, and H. Geffner, and its regression version HSP-R [18], or FF [61] by Hoffmann and Nebel.

In [116], Earl Sacerdoti developed a planner called NOAH which searches through the plan space instead of the state space. Basically, the plan space is a directed graph in which each node is a structure of actions corresponding to a (portion of) plan and each edge denotes a plan refinement operation such as the addition of an action to a plan. The start node represents the null plan and each goal node corresponds to a
solution of the planning problem. This technique, called partial-order planning (POP), was later made precise by McAllester and Rosenblitt [92] in their planner SNLP (Systematic Non-Linear Planner). In 1992, Penberthy and Weld [109] extended the POP algorithm to handle problems expressed in a more expressive language in their planner UCPOP (Universal, Conditional Partial-Order Planner). Some other authors, e.g. [51, 65, 108], used heuristics to improve the performance of the POP algorithm.

Instead of searching in the state space or the plan space, the GRAPHPLAN algorithm by Blum and Furst [17] first constructs a special, compact structure called planning graph and then performs a backward search in this graph. Shortly after published, GRAPHPLAN have attracted the planning community. Originally, GRAPHPLAN is limited to simple STRIPS [45] operators and thus several work extended GRAPHPLAN to handle more expressive languages. For example, [1, 46, 69] extended GRAPHPLAN to handle conditional effects. Work by Iwen and Mali [64] exploited the potential of GRAPHPLAN in distributed planning by developing a pair of two-agent planners DGP (distributed GRAPHPLAN) and IG-DGP (interaction graph-based DGP). Some other authors, e.g. [23, 36, 61, 107], used the planning graph as a means of deriving heuristic. In [91], Lopez and Bacchus presented an efficient method to formulate the planning problem as a constraint satisfaction problem. Their transformations uncover additional structure in the planning problem that subsumes the structure uncovered by GRAPHPLAN algorithm. The CSP encoded planning problem is then solved by using standard CSP algorithms.
A planning problem can also be viewed as a satisfiability (SAT) problem and this approach has been adopted by several researchers, e.g. [67, 68], in the field. In this approach, a planning problem is encoded as a SAT problem and then a SAT solver is used to solve the resulting SAT problem. The output of the SAT solver, which is a model, is then converted back into solutions of the planning problem.

Last but not least, in the answer set planning approach, a planning problem is translated into a logic program and then an answer set solver is used to generate answer sets of the logic program. These answer sets represent solutions of the original planning problem. This approach was first used by Subrahmanian and Zaniolo [135] and then pursued by several other authors [40, 41, 80, 127]. Similarly to the SAT approach, the translation is in general independent of the solver. As a result, the performance of an answer set based planner depends heavily on the underlying solver.

### 1.4 Domain Constraints and The Ramification Problem

Domain constraints constitute an important part of every dynamic domain. A domain constraint differs from a dynamic law in that it expresses a relationship among fluents rather than a relationship between an action and fluents. In general, a domain constraint is of the form

\[ \psi \Rightarrow \varphi \]  

(1)

where \( \varphi \) and \( \psi \) are fluent formulas. Intuitively, this constraint says that in any state of the domain, if the fluent formula \( \psi \) is true then so is \( \varphi \). For example,
(a) in the travel domain, the fact that “an object cannot be at two different places at
the same time” can be represented as the following domain constraint

\[ at(X) \Rightarrow \neg at(Y) \]

where \(X\) and \(Y\) are two different locations,

(b) in the block worlds domain, the fact that “if block \(X\) is on block \(Y\) and \(Y\) is
above block \(Z\) then \(X\) is above \(Z\)” can be represented as the following domain
constraint

\[ on(X, Y), above(Y, Z) \Rightarrow above(X, Z) \]

The use of domain constraints in reasoning about action and change offers several ad-
vantages. First, it provides a natural, lucid way to represent knowledge. Second, it
helps avoid duplications in the specification of action effects. As an example, in the
travel domain, without domain constraints, for every action that changes the location
of the object, in addition to asserting that the agent would be in the new location, one
also has to assert that the agent would no longer be at the previous location. Further-
more, to represent the fact that initially the object is at exactly one location, say \(A\), one
has to add to the knowledge base not only the fact \(at(A)\) for location \(A\) but also the
fact \(\neg at(X)\) for every other location \(X\). Third, it is more elaboration-tolerant, that is,
adding new knowledge requires only small change in the description of the domain.
For example, when there are some new locations, without domain constraints, one has
to assert that initially the object is not at these new locations. Finally, compiling away
domain constraints is difficult, even impossible in practice.

In the presence of domain constraints, an action may have *indirect effects*. For
example, in the above travel domain, the action that moves the object from location \( A \)
to location \( B \) (action \( \text{move}(A, B) \)) causes not only the fluent \( \text{at}(B) \) to be true but also
the fluent \( \text{at}(A) \) to be false. Similarly, in the blocks world domain, the action of putting
the block \( A \) atop the block \( B \) (action \( \text{put}(A, B) \)) causes not only \( \text{on}(A, B) \) to be true
but also \( \text{above}(A, C) \) to be true for every block \( C \) that is under \( B \). In these examples,
\( \neg \text{at}(A) \) and \( \text{above}(A, C) \) are indirect effects of actions \( \text{move}(A, B) \) and \( \text{put}(A, B) \)
respectively. Identifying indirect effects of actions is known as the *ramification problem*
in the field. The main difficulty of this problem lies in the fact that an action may cause
a long chain of indirect effects. Imagine, for example, a complicated electronic circuit
where many components are interconnected with each other. Turning on a switch may
cause a series of changes in many other components of the circuit.

To address the ramification problem, Winslett [150] presented a solution to the
ramification problem based on the principle of minimal changes. In her solution, along
with direct effects of the action, a minimum number of additional changes are selected
so as to satisfy domain constraints. Lifschitz [79] suggested a distinction between
two types of fluents: frame fluents and non-frame fluents. Only frame fluents follow
the principle of inertia and their values completely determine the values of non-frame
problems. McIlraith [101] devised a procedure for compiling (a restricted class of)
domain constraints into successor state axioms in situation calculus. Shanahan [121] showed that indirect effects of actions can be accounted for in event calculus without introducing a lot of extra machineries.

Some other researchers [9, 84, 93, 140] argued that in reasoning about action and change, to describe static relationships in the domain, it is better to use another form of domain constraints, called static causal laws and written as

\[ \psi \rightarrow \varphi \]  

(2)

The intuitive meaning of the above static causal law is that if \( \psi \) is caused to be true then \( \varphi \) is also caused to be true. The main difference between a domain constraint of the form (1) and a static causal law of the form (2) is that the latter is not contrapositive, i.e., if \( \neg \varphi \) is caused to be true then \( \neg \psi \) is not necessarily caused to be true. This dissertation follows this causality approach to deal with static causal laws.

While thoroughly studied by the reasoning about action and change community, static causal laws have rarely been directly considered by the planning community. Although the original specification of the Planning Domain Description Language (PDDL) – a language frequently used for the specification of planning problems by the planning community – includes axioms (a restricted form of domain constraints) [52], most of the planning domains used in the recent planning competitions [3, 38, 90] do not include axioms. This is in part true due to the fact that the semantics for PDDL with axioms is not clearly specified. In [8, 136], it was shown that adding axioms not
only increases the expressiveness and elegance of the action representation language but also improves the performance of planners.

1.5 Incomplete Information and Sensing Actions

Intelligent agents acting in real world environments have to face both the absence of complete information and limited observability about the domain. The latter refers to situations in which some certain properties of the domain are invisible to the agent at the reasoning (or planning) time. They can be observed only after some actions, called sensing actions, have been executed. For example, a robot cannot determine whether there is an obstacle on its way unless it uses one of its sensors to spot it. Or, one cannot decide whether there is a bomb in a package unless we open the package or use a special device to detect it.

Reasoning and planning with incomplete information and sensing actions have been intensively studied by many researchers in the field. Most of them rely on the possible world approach that was introduced in [102]. The basic idea of this approach lies in that to reason about the effects of an action or a plan, with its incomplete knowledge about the current state of the domain, an agent should consider all possible states of the domain that are consistent with its knowledge. The main weakness of this approach is its high complexity. It was proved in [13], in conformant setting\(^2\), for example, the

\(^2\)In the presence of incomplete information, we may have more than one possible initial state. Conformant planning is the problem of determining a sequence of actions that leads to a state that satisfies a given goal from every possible initial state.
problem of finding a (polynomial length) conformant plan using the possible world approach is $\Sigma_2^P$-complete.

An alternative to the possible world approach is the approximation approach. This approach, which was adopted by several authors in [7, 50, 56, 87, 111, 126], is motivated by the fact that reasoning and planning in the presence of incomplete information is intractable and many queries can be answered based on the knowledge of the agent (rather than possible states of the domain) at a time. One of the proposals in this direction is the 0-approximation due to Son and Baral [126]. In this approximation, the set of possible initial states is approximated by a single partial state and reasoning is performed based on a transition function between partial states. The main advantage of the 0-approximation in particular and approximation approach in general is their lower complexity in reasoning and planning tasks, in comparison with the possible world approach as shown in [13]. The price that one has to pay when using an approximation is its incompleteness with respect to the possible world approach, i.e., a reasoner based on this approach may answer a query about the truth value of a fluent formula after the execution of a sequence of actions with ‘unknown’ while another reasoner using the possible world approach would answer with either ‘true’ or ‘false’. This also implies that a planner based on an approximation cannot solve some problems that are solvable by other planners based on the possible world approach. Another limitation of the existing approximations is that the underlying representation languages either do not allow for domain constraints or just allow for a limited class of domain constraints. For ex-
ample, the approximations in \[7, 87, 126\] do not take into account domain constraints; the approximation in [50] is guaranteed to be sound with respect to the possible world approach only when the bodies of domain constraints contain at most one literal.

1.6 Outline Of Dissertation and Main Contributions

This dissertation studies an efficient method for reasoning with incomplete information in the presence of static causal laws and its application in planning. The action description language $\mathcal{AL}$ [11] is selected as the underlying representation language for the framework as it supports both concurrent actions and static causal laws. We employ both logic programming with answer set semantics and imperative languages in the implementation of planners. A brief review of the $\mathcal{AL}$ language, the possible world approach and the 0-approximation approach, planning, and logic programming with answer set semantics is given in Chapter 2.

The dissertation is divided into three main parts. In the first part (Chapter 3), we introduce a new method for reasoning and planning with action theories in the presence of incomplete information about the initial state. Although primarily based on the 0-approximation, the method is guaranteed to achieve completeness with respect to the possible world semantics. To this end, we first study a sufficient condition for which an action theory under the 0-approximation semantics is complete with respect to possible world semantics. We then introduce the notion of decisive sets of fluents for partial states with respect to fluent formulas. Based on decisive sets, an action theory can
be modified into another action theory such that the 0-approximation semantics of the modified one agrees with the possible semantics of the original one on the truth value of a given fluent formula. Furthermore, given an action theory, a decisive set for a partial state with respect to a fluent formula can be computed in polynomial time in the size of the domain description and the size of the formula. These results are then used in the development of a forward, best-first search conformant planner, CPA+. Despite its simple built-in heuristic, CPA+ can be competitive with other state-of-the-art conformant planners.

In the second part (Chapter 4), we address the problem of designing approximations for domain descriptions with static causal laws. In particular, we introduce two different approaches to the problem. In the first approach, called the possibly-holds approach, the successor partial state is defined based on the set of fluent literals that may hold in some possible successor state. The idea of second approach, called the possibly-changes approach, is rooted in the observation that every change must be caused by either a dynamic causal law or a static causal law. As a result, it defines the successor partial state based on identifying changes that might occur as the execution of an action. These two approaches result in four different approximations of $\mathcal{AL}$ descriptions which can be computed very efficiently.

Utilizing recent advances in answer set programming [81], we develop a conformant planner, called CPASP, which implements one of the proposed approximations. The experimental results shows that CPASP can solve a broad spectrum of conformant
benchmarks, and especially performing well on concurrent benchmarks. The main weakness of CPASP is that it does not scale up well on sequential benchmarks due to the fact that existing answer set solvers require the input to be grounded. To overcome this weakness, we develop another conformant planner in C++, called CpA. This planner is shown to be competitive with other state-of-the-art conformant planners.

In the last part of the dissertation (Chapter 5) we extend the approximations to incorporate sensing actions. To this end, we first develop an action description language called $\mathcal{AL}_K$ for sensing actions and define the notions of queries and conditional plans of $\mathcal{AL}_K$. We show that with some restrictions, the conditional planning with respect to the newly developed approximations is NP-complete. This facilitates the development of ASCP, an answer set programming based planner that is capable of generating both conformant and conditional planners. ASCP is proved to be sound and complete with respect the 0-approximation it relies on.

The main contributions of this dissertation can be summarized as follows.

1. A sufficient condition for which reasoning with an action theory using the 0-approximation is complete. This condition is constructed based on the dependencies between fluents, and between actions and fluents.

2. A new method that extends the 0-approximation framework to achieve complete reasoning and planning.

3. A sound and complete conformant planning system for domains without static
causal laws, whose performance is competitive with other state-of-the-art con-
formant planners.

4. A general definition of approximations of $\mathcal{AC}$ domain descriptions.

5. Two different approaches to tackling the problem of designing approximations for domains without static causal laws.

6. Four different approximations for domains with static causal laws, namely $T^{ph}(D)$, $T^{pc}(D)$, $T^{ph+}(D)$, and $T^{pc+}(D)$. These approximations are proved to be complete with respect to the possible world semantics on a restricted class of $\mathcal{AC}$ domain descriptions.

7. A logic programming based conformant planner for domains with static causal laws that are capable of generating both concurrent plans and sequential plans.

8. A forward, best-first conformant planner which implements the $T^{ph}(D)$ and $T^{pc}(D)$ approximations. This planner can generate sequential conformant plans and has capability of dealing with disjunctive information in the specification of the initial state.

9. An action description language $\mathcal{A}L_\mathcal{K}$ that allows for concurrent actions, sensing actions, and static causal laws.

10. Four different approximations of $\mathcal{A}L_\mathcal{K}$ domain descriptions, each of which extends an approximation of $\mathcal{AC}$ domain descriptions to incorporate sensing ac-
11. A logic programming based conditional planner.

The proposal for this dissertation has been presented in the doctoral consortium of the “22nd International Conference on Logic Programming (ICLP’06)” [144]. Most of results in this dissertation have been published as refereed articles in several international and national workshops, conferences, and journals. In particular, the completeness condition for the 0-approximation, the procedure for complete reasoning based on the 0-approximation, and the planner CPA⁺ (Results (1)–(3)) have been published in the proceedings of the “10th International Conference on Principles of Knowledge Representation and Reasoning (KR’06)” [130].

The approximations of \( \mathcal{AL} \) domain descriptions and their application in planning (Results (4)–(8)) have been published in the proceedings of the “20th National Conference on Artificial Intelligence (AAAI’05)” [124] and in the proceedings of the “8th International Conference on Logic Programming and Nonmonotonic Conference (LPNMR’05)” [125]. The extension of CPASP to deal with disjunctive information about the initial state has been published in the proceedings of the “20th International Joint Conference on Artificial Intelligence (IJCAI’07)” [104].

The action description language \( \mathcal{AL}_K \), its approximation semantics, and the planner ASCP (Results (9)–(11)) have been reported in the proceedings of [131] and in the “Theory and Practice of Logic Programming (TPLP)” journal [145].
tension of $\mathcal{AL}_K$ to deal with multi-valued fluents has been published in the “Studia Logica” journal [133]. An abstract version of this paper has been published in the proceedings of the “IJCAI-03 Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC’03)” [132].
CHAPTER 2
BACKGROUND

This chapter reviews background related to the work of this dissertation. We
start by introducing a widely used formalism for representing and reasoning about ac-
tion and change – the action description language $\mathcal{AL}$ – including its syntax and se-
mantics, and the entailment relationship between an action theory and a query (Section
2.1). In Section 2.2, we review two approaches to dealing with incomplete information,
the possible world approach, and the 0-approximation approach. In Section 2.3, we in-
troduce the concepts of planning problem instances and their solutions. In Section 2.4,
we review logic programming with answer set semantics (or answer set programming).
Section 2.5 summarizes the chapter. The proofs of propositions, and theorems in the
chapter are listed in Appendix B.

2.1 Action Description Language $\mathcal{AL}$

2.1.1 Syntax

The signature $\Sigma$ of a domain description of $\mathcal{AL}$ consists of two disjoint, non-
empty sets of symbols: the set $F$ of fluents, and the set $A$ of elementary actions. An
action is a non-empty set of elementary actions. When an action contains more than
one elementary action, it is called a concurrent action. Intuitively, an execution of an
action is interpreted as a simultaneous execution of its components. In this sense, there
is no distinction between an elementary action $e$ and the action $\{e\}$. For this reason, we identify an elementary action $e$ with the action $\{e\}$. A fluent literal $l$ is a fluent $f$ or its negation $\neg f$. For a fluent literal $l$, we denote by $\neg l$ the fluent literal complementary to $l$, i.e., $\neg(f) = \neg f$ and $\neg(\neg f) = f$. By $L$ we denote the set of fluent literals \{ $f, \neg f \mid f \in F$\}. A fluent formula $\varphi$ is a formula constructed from fluent literals using connectives $\land$, $\lor$ and $\neg$ as usual. An $\mathcal{AL}$ domain description is a set of statements of the following forms:

$$e \text{ causes } l \text{ if } \psi$$

(3)

$$l \text{ if } \psi$$

(4)

$$\text{impossible } b \text{ if } \psi$$

(5)

where $e$ is an elementary action, $b$ is an action, $l$ is a fluent literal, and $\psi$ is a set of fluent literals from the signature $\Sigma$. The set of fluent literals $\psi$ is referred to as the precondition of the corresponding statement. When the precondition of a statement is empty, its if part can be omitted. Statement (3), called a dynamic causal law, says that if $e$ is executed in a state satisfying $\psi$ then $l$ will hold in any resulting state. Statement (4), called a static causal law, says that any state satisfying $\psi$ must satisfy $l$. Statement (5), called an impossibility condition, says that $b$ cannot be executed in any state satisfying $\psi$.

To illustrate the syntax of $\mathcal{AL}$, let us consider an instance of (a variant of) the bomb-in-the-toilet domain [100].
Example 2.1 It has been alarmed that there may be a bomb in the airport. There are two suspicious packages. Dunking a package that contains a bomb into a toilet of the airport (there are two of them) can disarm the bomb. This action also clogs the toilet. Flushing a toilet can make it unclogged. We are safe only if both packages are disarmed.

<table>
<thead>
<tr>
<th>Variables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$'s stand for packages, $i \in {1, 2}$, $p_1 \neq p_2$</td>
</tr>
<tr>
<td>$t_j$'s stand for toilets, $j \in {1, 2}$, $t_1 \neq t_2$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fluents:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{armed}(p_i)$: package $p_i$ contains the bomb</td>
</tr>
<tr>
<td>$\text{clogged}(t_j)$: toilet $t_j$ is clogged</td>
</tr>
<tr>
<td>$\text{safe}$: all the bombs are disarmed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{dunk}(p_i, t_j)$: dunk package $p_i$ into toilet $t_j$</td>
</tr>
<tr>
<td>$\text{flush}(t_j)$: flush toilet $t_j$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>impossible ${\text{dunk}(p_i, t_j), \text{flush}(t_j)}$</td>
</tr>
<tr>
<td>impossible ${\text{dunk}(p_1, t_j), \text{dunk}(p_2, t_j)}$</td>
</tr>
<tr>
<td>impossible ${\text{dunk}(p_i, t_1), \text{dunk}(p_i, t_2)}$</td>
</tr>
<tr>
<td>impossible $\text{dunk}(p_i, t_j)$ if $\text{clogged}(t_j)$</td>
</tr>
<tr>
<td>$\text{dunk}(p_i, t_j)$ causes $\neg \text{armed}(p_i)$</td>
</tr>
<tr>
<td>$\text{dunk}(p_i, t_j)$ causes $\text{clogged}(t_j)$</td>
</tr>
<tr>
<td>$\text{flush}(t_j)$ causes $\neg \text{clogged}(t_j)$</td>
</tr>
<tr>
<td>$\text{safe}$ if $\neg \text{armed}(1), \neg \text{armed}(2)$</td>
</tr>
</tbody>
</table>

Figure 2.1: Description of the bomb-in-the-toilet domain, $\mathcal{D}_{2.1}$

Figure 2.1 shows an encoding of the domain in the action language $\mathcal{AL}^1$. As can be seen, the domain description, denoted by $\mathcal{D}_{2.1}$, contains four types of impossibility conditions. The first one says that “it is impossible to dunk a package into a toilet

---

1Note that in the description of a domain, we sometimes use typed variables. A statement with variables is interpreted as the collection of its ground instances.
that is being flushed”. The second one states that “it is impossible to dunk two different packages into the same toilet at the same time”. The third one says that “it is impossible to dunk a package into two different toilets at the same time”. Unlike the first three conditions that specify physical impossibilities of concurrent actions, the last one specifies physical impossibility of an elementary action. It says that “it is impossible to dunk a package into a toilet which is clogged”.

In addition to the impossibility conditions, \( D_{2.1} \) also includes three dynamic causal laws to describe the effects of actions. The first two state that dunking a package into a toilet disarms the package and makes the toilet clogged. The last one law states that flushing a toilet makes it unclogged.

The last statement in the description is a static causal law that describes the relationship between fluent \( \text{safe} \) and fluents \( \text{armed}'s: \) we are safe if both packages have been disarmed.

\subsection{2.1.2 Semantics}

The semantics of an \( A\mathcal{L} \) domain description is given by a transition \( \Phi \) that maps actions and states into sets of states. Before providing the formal definition of the transition function \( \Phi \), let us introduce some terminology and notation.

Given a domain description \( \mathcal{D} \), a set \( \sigma \) of fluent literals is \textit{consistent} if it does not contain two complementary fluent literals. We say that \( \sigma \) is \textit{complete} if for every fluent \( f \in F \), either fluent literal \( f \) or \( \neg f \) belongs to \( \sigma \). A fluent literal \( l \) holds in a set
of fluent literals $\sigma$ if $l$ belongs to $\sigma$; $l$ possibly holds in $\sigma$ if $\neg l$ does not belong to $\sigma$. A set $\gamma$ of fluent literals holds (resp. possibly holds) in $\sigma$ if every fluent literal in $\gamma$ holds (resp. possibly holds) in $\sigma$.

A set of fluent literals $\sigma$ is closed under a static causal law (4) if $l$ holds in $\sigma$ whenever $\psi$ holds in $\sigma$. By $Cl_D(\sigma)$ we denote the smallest set of fluent literals that contains $\sigma$ and is closed under the static causal laws of $D$.

**Definition 2.1** Let $D$ be a domain description. A state $s$ of $D$ is a complete, consistent set of fluent literals closed under the static causal laws of $D$.

**Definition 2.2** A partial state $\delta$ of $D$ is a consistent set of fluent literals closed under the static causal laws of $D$. A partial state $\delta$ is valid if there is a state $s$ such that $\delta \subseteq s$.

For convenience, for a partial state $\delta$ (resp. a set of partial states $\Delta$) we denote by $ext(\delta)$ (resp. $ext(\Delta)$) the set of all states that contain $\delta$ (resp. a partial state in $\Delta$).

That is,

$$ext(\delta) = \{s \mid s \text{ is a state s.t. } \delta \subseteq s\}$$  \hspace{1cm} (6)

and

$$ext(\Delta) = \bigcup_{\delta \in \Delta} ext(\delta)$$  \hspace{1cm} (7)

The sets $ext(\delta)$ and $ext(\Delta)$ are called completions of $\delta$ and $\Delta$ respectively.

The following example illustrates the meanings of states, partial states, valid partial states, and completions of partial states and sets of partial states.
Example 2.2 For the domain description with the two static causal laws\(^2\)

\[ D_{2.2} = \{ f \text{ if } g, h, \ f \text{ if } g, \neg h \} \]

1. \( s_0 = \{ f, g, h \} \) and \( s_1 = \{ f, g, \neg h \} \) are states, while \( s_2 = \{ \neg f, g, h \} \) and \( s_3 = \{ \neg f, g, \neg h \} \) are not (because they are not closed under the set of static causal laws of \( D_{2.2} \)),

2. \( \delta_0 = \{ \neg f, g \} \) and \( \delta_1 = \{ f, g \} \) are partial states, while \( \delta_2 = \{ \neg f, g, h \} \) is not (because it is not closed under the first static causal law),

3. of the above two partial states, \( \delta_0 = \{ \neg f, g \} \) is not a valid partial state because it is not part of any state; \( \delta_1 \), on the contrary, is a valid partial state because it is part of the state \( \{ f, g, h \} \),

4. the completions of \( \delta_0 \), and \( \delta_1 \), and \( \Delta_0 = \{ \delta_0, \delta_1 \} \) are

\[
\begin{align*}
ext(\delta_0) &= \emptyset & \next(\delta_1) &= \{ \{ f, g, h \}, \{ f, g, \neg h \} \} \\
next(\Delta_0) &= next(\delta_0) \cup next(\delta_1) = \{ \{ f, g, h \}, \{ f, g, \neg h \} \}
\end{align*}
\]

\( \square \)

The truth value of a fluent formula \( \varphi \) in a partial \( \delta \), denoted by \( \delta(\varphi) \), is defined as follows:

\(^2\)From now on, we assume that all the fluents and actions of a domain description are only those appearing in the statements of the domain description unless otherwise stated

25
1. If $\varphi \equiv l$ for some fluent literal $l$ then $\delta(\varphi) = true$ iff $l \in \delta$; $\delta(\varphi) = false$ iff $\neg l \in \delta$; $\delta(\varphi) = unknown$ otherwise.

2. If $\varphi \equiv \varphi_1 \land \varphi_2$ then $\delta(\varphi) = true$ iff $\delta(\varphi_1) = true$ and $\delta(\varphi_2) = true$; $\delta(\varphi) = false$ iff $\delta(\varphi_1) = false$ or $\delta(\varphi_2) = false$; $\delta(\varphi) = unknown$ otherwise.

3. If $\varphi \equiv \varphi_1 \lor \varphi_2$ then $\delta(\varphi) = true$ if $\delta(\varphi_1) = true$ or $\delta(\varphi_2) = true$; $\delta(\varphi) = false$ iff $\delta(\varphi_1) = false$ and $\delta(\varphi_2) = false$; $\delta(\varphi) = unknown$ otherwise.

4. If $\varphi \equiv \neg \varphi_1$ then $\delta(\varphi) = true$ iff $\delta(\varphi_1) = false$; $\delta(\varphi) = false$ iff $\delta(\varphi_1) = true$; $\delta(\varphi) = unknown$ otherwise.

When $\varphi$ is true or false in a partial state $\delta$, we say that it is known in $\delta$. Note that when $\delta$ is complete, i.e., it is a state, then every fluent formula $\varphi$ is known in $\delta$. For a set of partial states $\Delta$, $\varphi$ is true (resp. false) in $\Delta$ if it is true (resp. false) in every $\delta \in \Delta$.

An action $a$ is executable in a state $s$ if $\mathcal{D}$ does not contain any impossibility condition (5) such that $\psi$ holds in $s$ and $b \subseteq a$. Given a state $s$ and an action $a$ that is executable in $s$, a fluent literal $l$ is called a direct effect of $a$ in $s$ if there exists a dynamic causal law (3) such that $e \in a$ and $\psi$ holds in $s$. By $de(a, s)$ we denote the set of all direct effects of $a$ in $s$ 3, i.e.,

$$de(a, s) = \{l \mid \exists \text{ a dynamic causal law}(3) \text{ in } \mathcal{D} \text{ such that } e \in a \text{ and } \psi \text{ holds in } s\} \quad (8)$$

The transition function $\Phi$ of a domain description is defined as follows.

\(^{3\text{“de” stands for “direct effects”}}\)
Definition 2.3 Let $D$ be a domain description. For any action $a$ and state $s$

1. if $a$ is not executable in $s$ then

$$\Phi(a, s) = \bot$$

2. otherwise,

$$\Phi(a, s) = \{s' \mid s' \text{ is a state such that } s' = \text{cl}_D(de(a, s) \cup (s \cap s'))\}$$

where $\bot$ denotes an undefined (or a failure) situation.

It is worth to note here that for domain descriptions without concurrent actions, the equation in Definition 2.3 is equivalent to the one defined in [93].

Intuitively, the transition function $\Phi$ of a domain description $D$ describes a transition diagram $T(D)$ whose nodes correspond to physically possible states of the domain and whose arcs are labeled with actions. A transition $\langle s, a, s' \rangle$ belongs to $T(D)$ iff $s'$ belongs to $\Phi(a, s)$ and such a state $s'$ is called a possible successor state of $s$ as a result of the execution of $a$. If action $a$ is clear from the context then we simply say that $s'$ is a possible successor state of $s$. Furthermore, throughout the dissertation, we will use the notions $\langle s, a, s' \rangle \in T(D)$ and $s' \in \Phi(a, s)$ interchangeably.

Observe that the above definition implies that a fluent literal $l$ holds in a possible successor state $s'$ of a state $s$ as a result of the execution of an action $a$ iff either (i) it is a direct effect of $a$ in $s$, i.e., $l \in de(a, s)$, (ii) it holds by inertia (see Chapter 1), i.e., $l \in (s \cap s')$, or (iii) it is an indirect effect of $a$ in $s$, i.e., $l$ holds because of the operator
This observation is very important to understand the proposed approximations that will be presented in the subsequent chapters.

**Example 2.3** Consider the domain description $D_{2.1}$ from Example 2.1. Let

$$s_0 = \{\text{armed}(1), \text{armed}(2), \neg \text{clogged}(1), \neg \text{clogged}(2), \neg \text{safe}\}$$

and

$$a_1 = \{\text{dunk}(1, 1), \text{dunk}(2, 2)\}$$

We can easily verify that $a_1$ is executable in $s_0$ because no precondition of the impossibility conditions of $D_{2.1}$ whose action is a subset of $a_1$ holds in $s_0$. Furthermore, the state

$$s_1 = \{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\}$$

is a possible successor state of $s_0$ because

$$Cl_{D_{2.1}}(de(a_1, s_0) \cup (s_0 \cap s_1)) =$$

$$Cl_{D_{2.1}}(\{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2)\} \cup \emptyset) =$$

$$\{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\} = s_1$$

Note that $safe$ belongs to the closure of $\sigma = \{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2)\}$ because $D_{2.1}$ contains the static causal law

$$safe \text{ if } \neg \text{armed}(1), \neg \text{armed}(2)$$
and both $\neg\text{armed}(1)$ and $\neg\text{armed}(2)$ hold in $\sigma$. Of the fluent literals in $s_1$, $\neg\text{armed}(1)$, $\neg\text{armed}(2)$, $\text{clogged}(1)$ and $\text{clogged}(2)$ are direct effects of action $a_1$, and $\text{safe}$ is an indirect effect of action $a_1$.

We can also check that no state other than $s_1$ is a possible successor state of $s_0$. As a result, we have

$$\Phi(a_1, s_0) = \{s_1\}$$

Now let

$$a_2 = \{\text{dunk}(1, 1), \text{flush}(2)\}$$

Similarly, we can check that $a_2$ is executable in $s_1$ and

$$s_2 = \{\neg\text{armed}(1), \text{armed}(2), \text{clogged}(1), \neg\text{clogged}(2), \neg\text{safe}\}$$

is the unique successor state of $s_0$, i.e.,

$$\Phi(a_2, s_0) = \{s_2\}$$

In this case, fluent literals $\neg\text{armed}(1)$, $\text{clogged}(1)$, and $\neg\text{clogged}(2)$ hold in $s_2$ because they are direct effects of $a_2$; fluent literals $\text{armed}(2)$, $\neg\text{safe}$, and $\text{clogged}(2)$ hold by inertia; and there is no indirect effect. □

For a domain description, we are often interested in two important properties: whether it is consistent and whether is is deterministic. The first property is defined as follows.
Definition 2.4 A domain description \( \mathcal{D} \) is consistent if for any action \( a \) and state \( s \), if \( a \) is executable in \( s \) then \( \Phi(a, s) \neq \emptyset \). Otherwise, it is inconsistent.

Example 2.4 Consider the following domain description:

\[
\mathcal{D}_{2.4} = \left\{ \begin{array}{l}
e \text{ causes } f \text{ if } g \\
e \text{ causes } \neg f \text{ if } h \\
\end{array} \right\}
\]

Then, the set of fluent literals \( s = \{f, g, h\} \) is a state because it is complete and consistent. The action \( e \) is executable in \( s \) because the domain description has no impossibility condition. If \( s' \) is a successor state of \( s \) then it is easy to see that both \( f \) and \( \neg f \) belong to \( s' \), which implies \( s' \) is inconsistent. Hence, no successor state of \( s \) exists, i.e., \( \Phi(e, s) = \bot \). According to the definition \( \mathcal{D}_{2.4} \) is inconsistent.

However, if we add to \( \mathcal{D}_{2.4} \) the following impossibility condition

\[ \text{impossible } e \text{ if } g, h \]

then the domain description would become consistent because \( e \) could not be executed in any state in which both \( g \) and \( h \) holds. \( \Box \)

The inconsistency of a domain description indicates that there might be something wrong with the description: either we did not encode the domain correctly or maybe we were missing some actions, fluents, or statements that should have been incorporated in the description. In this sense, it is reasonable to require that domain descriptions be consistent. In the remainder of the dissertation, whenever a domain description is mentioned, we mean that it is consistent.
The determinism (or non-determinism) of a domain description is defined as follows.

**Definition 2.5** A domain description \( D \) is deterministic if for any action \( a \) and state \( s \), if \( a \) is executable in \( s \) then \( |\Phi(a, s)| \leq 1 \).

It should be noted that if a domain description \( D \) does not contain any static causal laws then it is deterministic as implied by in the following proposition.

**Proposition 2.1** Let \( D \) be a domain description without static causal laws, \( s \) be a state and \( a \) be an action that is executable in \( s \). A state \( s' \) is a possible successor state of \( s \) as a result of the execution of \( a \) iff \( s' = de(a, s) \cup (s \setminus de(a, s)) \).

**Proof.** See Section B.1.

On the contrary, a domain description with static causal laws may be non-deterministic as illustrated by the following example.

**Example 2.5** Consider the domain description:

\[
D_{2.5} = \left\{ \begin{array}{l}
e \text{ causes } f \\
g \text{ if } f, \neg h \\
h \text{ if } f, \neg g \\
\end{array} \right\}
\]

Let \( s_0 = \{\neg f, \neg g, \neg h\} \). We can verify that \( s_1 = \{f, h, \neg g\} \) and \( s_2 = \{f, g, \neg h\} \) are possible successor states of \( s_0 \). Hence, by definition, \( D_{2.5} \) is non-deterministic.

**2.1.3 The Entailment Relationship \( \models \)**

One of the main problems in the field of reasoning about action and change is to answer the question: given a domain description \( D \), whether a fluent formula \( \varphi \) is
true after the execution of a sequence of actions \( \alpha \) in an initial state \( s_0 \). In the action language \( \mathcal{AL} \), a domain description \( D \) together with an initial state \( s_0 \) is called an \textit{action theory}, and the question about the truth value of \( \varphi \) is called a \textit{query} and expressed as

\[
\varphi \textbf{ after } \alpha
\]  

(9)

The answer to the above query is determined by an entailment relationship \( \models \) between the action theory and the query. The definition of \( \models \) is based on an extension of the transition function \( \Phi \). The extended function, whose definition is given below, is a mapping from sequences of actions and states into sets of states.

**Definition 2.6** Let \( D \) be a domain description. For any sequence of actions \( \alpha \) and state \( s \),

1. if \( \alpha = \langle \rangle \) then

\[
\hat{\Phi}(\alpha, s) = \{s\}
\]

2. if \( \alpha = \langle \beta, a \rangle \) where \( \beta \) is a sequence of actions and \( a \) is an action then

\[
\hat{\Phi}(\alpha, s) = \begin{cases} 
\bot & \text{if } \hat{\Phi}(\beta, s) = \bot, \text{ or } \Phi(a, s') = \bot \text{ for some } s' \in \hat{\Phi}(\beta, s) \\
\bigcup_{s' \in \hat{\Phi}(\beta, s)} \Phi(a, s') & \text{otherwise}
\end{cases}
\]

Intuitively, for a state \( s \) and a sequence of actions \( \alpha \), \( \hat{\Phi}(\alpha, s) \) is the set of possible final states after the execution of \( \alpha \) in \( s \). If we think of \( D \) as the transition diagram \( T(D) \) then \( s' \in \hat{\Phi}(\alpha, s) \) corresponds to the fact that there is a path from \( s \) to \( s' \) in the diagram whose arcs are labeled with actions in \( \alpha \).
The entailment relationship between an action theory and a query is defined as follows.

**Definition 2.7** An action theory \((\mathcal{D}, s)\) is said to entail a query \(\varphi\) after \(\alpha\), denoted by

\[(\mathcal{D}, s) \models \varphi\ \text{after}\ \alpha,\]

if \(\hat{\Phi}(\alpha, s) \neq \bot\) and \(\varphi\) is true in any state belonging to \(\hat{\Phi}(\alpha, s)\).

Let us illustrate the intuitive meaning of \(\models\) by a concrete example.

**Example 2.6** Consider the Bomb domain, \(\mathcal{D}_{2,1}\), from Example 2.1 and let

\[s_0 = \{\text{armed}(1), \text{armed}(2), \neg\text{clogged}(1), \neg\text{clogged}(2), \neg\text{safe}\}\]

Let

\[\alpha = \langle\text{dunk}(1, 1), \text{dunk}(2, 2)\rangle\]

We can verify that \(\text{dunk}(1, 1)\) is executable in \(s_0\) and

\[s_1 = \{\neg\text{armed}(1), \text{armed}(2), \text{clogged}(1), \neg\text{clogged}(2), \neg\text{safe}\}\]

is the only possible successor state of \(s_0\), i.e.,

\[\Phi(\text{dunk}(1, 1), s_0) = \{s_1\}\]

Hence, we have

\[\hat{\Phi}(\langle\text{dunk}(1, 1)\rangle, s_0) = \Phi(\text{dunk}(1, 1), s_0) = \{s_1\}\]
Next, we can also verify that \(dunk(2, 2)\) is executable in \(s_1\) and

\[
s_2 = \{ ¬\text{armed}(1), ¬\text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\}\]

is the only possible successor state of \(s_1\). Hence, we have

\[
\hat{\Phi}(\alpha, s_0) = \bigcup_{s' \in \hat{\Phi}(\langle dunk(1, 1) \rangle, s_0)} \Phi(dunk(2, 2), s') = \Phi(dunk(2, 2), s_1) = \{s_2\}
\]

Because \(\text{safe}\) is true in \(s_2\), by the definition of \(\models\), we have

\[
(D, s_0) \models \text{safe} \text{ after } \langle dunk(1, 1), dunk(2, 2) \rangle
\]

2.2 Reasoning With Incomplete Information

Intelligent agents operating in real world environments usually do not have complete information about the domain they are working on. For this reason, it is unrealistic to describe the initial state of the domain as a single state in the corresponding action theory. We, therefore, represent an action theory as a pair \((D, \Delta)\) where \(D\) is a domain description and \(\Delta\) is a non-empty set of partial states. Each partial state \(\delta\) of \(\Delta\) describes a set of possible initial states of the domain – the set of possible states \(ext(\delta)\). Furthermore, we require that each \(\delta\) of \(\Delta\) is valid, i.e., \(ext(\delta) \neq \emptyset\). With this representation, both disjunctive information and incomplete information about the initial state of the domain can easily be expressed. As an example, in our bomb-in-toilet example (Example 2.1), to describe the initial state of the domain
in which exactly one of the packages is armed (disjunctive information) and the status of the toilets are unknown (incomplete information), we can use the set of partial states \( \Delta = \{\{\text{armed}(1), \neg \text{armed}(2)\}, \{\neg \text{armed}(1), \text{armed}(2)\}\} \). In this particular example, \( \Delta \) corresponds to \( 2 \times 2^3 = 16 \) possible initial states because for each \( \delta \) in \( \Delta \) there are three fluents unknown in \( \delta \): safe, clogged(1), and clogged(2), each of which has two possible truth values. Note that when no disjunctive information about the initial state is present, \( \Delta \) has exactly one element and when neither incomplete information nor disjunctive information is present, \( \Delta \) is a set which contains exactly one complete state.

In the presence of incomplete information, queries of the form (9) about the truth value of a fluent formula can be answered by using the possible world approach [102]. Besides, the approximation approach, e.g. [7, 50, 126], provides an alternative. In the next subsections, we review the possible world approach and one of the existing proposed approximations, the 0-approximation approach from [126]. Although the possible world approach can be applied to arbitrary action theories, including ones that have static causal laws and concurrent actions, the 0-approximation approach only considers action theories without static causal laws. Note that our presentation of the 0-approximation approach here is slightly different from its original version: we do allow for concurrent actions, whereas the original version does not.

To illustrate the two approaches, we use the following simple instance of the bomb-in-the-toilet domain.
Example 2.7 In this simplified version of the bomb-in-the-toilet domain, there are one package and one toilet only. The package may or may not contain a bomb. The following is an encoding of the domain in the language $\mathcal{AL}$.

\[
D_{2.7} = \begin{cases} 
\text{dunk causes } \neg \text{armed} \\
\text{dunk causes clogged} \\
\text{flush causes } \neg \text{clogged} \\
\text{impossible dunk if clogged}
\end{cases}
\]

Notice that unlike the $D_{2.1}$ domain description in Example 2.1, this domain description does not contain any static causal laws.

The question of our interest in this particular example is whether or not the bomb will be disarmed after the sequence of actions $\langle \text{flush, dunk} \rangle$ is executed, provided that initially we do not know whether or not the toilet is clogged and whether or not the package contains a bomb.

2.2.1 The Possible World Approach

A possible way to reason about the truth value of the above question is as follows (see Figure 2.2). Since both fluents $\text{armed}$ and $\text{clogged}$ are unknown in the beginning, we consider four possible initial states (called initial belief state) corresponding to different possible assignments of their truth values. After executing the action $\text{flush}$, the belief state changes to $\{\{\text{armed, }\neg \text{clogged}\}, \{\neg \text{armed, }\neg \text{clogged}\}\}$ as flushing the toilet makes it unclogged. After executing $\text{dunk}$, the final belief state of the domain is $\{\{\neg \text{armed, clogged}\}\}$. Because the fluent $\text{armed}$ is false in this final belief state, we can conclude that the bomb would be disarmed at the end. This process of reasoning
is known as the possible world approach, which relies on the possible world semantics to reason about the truth value of a query. Central to the possible world semantics are the concepts of belief states, the transition function $\Phi^P$ between belief states, and the entailment relationship $|=^P$ between an action theory and a query, whose definitions are given next.

![Figure 2.2: The possible world approach](image)

A belief state $S$ is a non-empty set of states. A fluent literal $l$ (resp. a set of fluent literals $\gamma$) holds in a belief state $S$ if it holds in every state in $S$. The truth value of a fluent formula $\varphi$ in a belief state $S$ is true if it is true in every $s$ in $S$, and false if otherwise. An action $a$ is executable in a belief state $S$ if it is executable in every state in $S$. The successor belief state of a belief state $S$ after the execution of $a$, denoted by $\Phi^P(a, S)$, is defined as follows.

**Definition 2.8** Let $D$ be a domain description. For any action $a$ and belief state $S$, 

37
1. if $a$ is not executable in $S$ then

$$\Phi^P(a, S) = \bot$$

2. otherwise,

$$\Phi^P(a, S) = \bigcup_{s \in S} \Phi(a, s)$$

Since we are assuming $D$ is consistent, $\Phi(a, s) \neq \emptyset$ for any action $a$ and state $s$ such that $a$ is executable in $s$. Hence, if $S$ is a belief state and $a$ is an action executable in $S$ then $\Phi^P(a, S)$ is indeed a belief state. Similarly to the transition function $\Phi$, we extends $\Phi^P$ to define the final belief state of a belief state after the execution of a sequence of actions as follows.

**Definition 2.9** Let $D$ be a domain description. For any sequence of actions $\alpha$ and belief state $S$,

1. if $\alpha = \langle \rangle$ then

$$\hat{\Phi}^P(\alpha, S) = S$$

2. if $\alpha = \langle \beta, a \rangle$ where $\beta$ is a sequence of actions and $a$ is an action then

$$\hat{\Phi}^P(\alpha, S) = \begin{cases} \bot & \text{if } \hat{\Phi}^P(\beta, S) = \bot \\ \Phi^P(a, \hat{\Phi}^P(\beta, S)) & \text{otherwise} \end{cases}$$

The entailment relationship between an action theory and a query with respect to the possible world semantics is defined as follows.
Definition 2.10 An action theory $(D, \Delta)$ is said to entail a query $\varphi$ after $\alpha$ with respect to the possible world semantics, denoted by,

$$(D, \Delta) \models^P \varphi \text{ after } \alpha$$

if $\hat{\Phi}P(\alpha, \text{ext}(\Delta)) \neq \perp$ and $\varphi$ is true in $\hat{\Phi}P(\alpha, \text{ext}(\Delta))$.

Example 2.8 Consider the domain $D_{2.7}$ from Example 2.7 and let $\Delta_0 = \{\emptyset\}$. We have

$S_0 = \text{ext}(\Delta_0) = \{s_0, s_1, s_2, s_3\}$ where $s_0 = \{\text{armed, clogged}\}$, $s_1 = \{\text{armed, } \neg \text{clogged}\}$, $s_2 = \{\neg \text{armed, clogged}\}$, and $s_3 = \{\neg \text{armed, } \neg \text{clogged}\}$.

For the state $s_0$ and the action $\text{flush}$, we have

$de(\text{flush}, s_0) = \{\neg \text{clogged}\}$

Because $D_{2.7}$ does not have static causal laws, by Proposition 2.1, we have

$$\Phi(\text{flush}, s_0) = \{de(a, s_0) \cup (s_0 \setminus de(a, s_0))\}$$

$$= \{\neg \text{clogged}\} \cup (\{\text{armed, clogged}\} \setminus \{\text{clogged}\})$$

$$= \{\{\text{armed, } \neg \text{clogged}\}\} = \{s_1\}$$

Likewise, we have

$$\Phi(\text{flush}, s_1) = \{\{\text{armed, } \neg \text{clogged}\}\} = \{s_1\}$$

$$\Phi(\text{flush}, s_2) = \{\{\neg \text{armed, } \neg \text{clogged}\}\} = \{s_3\}$$

$$\Phi(\text{flush}, s_3) = \{\{\neg \text{armed, } \neg \text{clogged}\}\} = \{s_3\}$$
Hence, we have

\[ \hat{\Phi}^P(\langle flush \rangle, S_0) = \Phi^P(flush, S_0) = \{s_1, s_3\} = S_1 \]

On the other hand, we can check that

\[ \Phi(dunk, s_1) = \Phi(dunk, s_3) = \{s_2\} \]

Accordingly, we have

\[ \Phi^P(dunk, S_1) = \{s_2\} = S_2 \]

The final belief state is

\[ \hat{\Phi}^P(\langle flush, dunk \rangle, S_0) = \Phi^P(dunk, \hat{\Phi}^P(\langle flush \rangle, S_0)) = \Phi^P(dunk, S_1) = S_2 \]

By Definition 2.10, this means that

\[ (\mathcal{D}_{2.7}, \Delta_0) \models^P \neg \text{armed after } \langle flush, dunk \rangle \]  \hspace{1cm} (11) \]

2.2.2 The 0-Approximation Approach

In the 0-approximation approach, a set of possible states of the domain at a time is approximated by a single partial state and reasoning is based on the transition function between partial states. Consider the bomb-in-the-toilet toilet in Example 2.8 for instance. As no information about the initial state of the domain is available to the agent, it initiates its knowledge with an empty partial state. After executing the action
flush, the agent’s knowledge changes to the partial state \{\neg clogged\} because the effect of the action flush is to make the toilet unclogged. After executing the action dunk, its knowledge becomes \{\neg armed, clogged\} because \neg armed and clogged are the two effects of action dunk. Since \neg armed holds in the final partial state, it concludes that the bomb will be disarmed after the above actions are executed. This reasoning process is depicted in Figure 2.3.

![Figure 2.3: The 0-approximation approach](image)

We now make this more formal. As usual, let us begin with some notations. An action \(a\) is executable in a partial state \(\delta\) if there is no impossibility condition (5) such that \(b \subseteq a\) and \(\psi\) possibly holds\(^4\) in \(\delta\). For a partial state \(\delta\) and an action \(a\) that is executable in \(\delta\), let

\[
de(a, \delta) = \{l \mid \exists \text{ a dynamic causal law (3) s.t. } e \in a \text{ and } \psi \text{ holds in } \delta\}
\]

(12)

and

\[
pe(a, \delta) = \{l \mid \exists \text{ a dynamic causal law (3) s.t. } e \in a \text{ and } \psi \text{ possibly holds in } \delta\}
\]

(13)

Each \(l \in de(a, \delta)\) (resp. \(l \in pe(a, \delta)\)) is called a direct effect (resp. possible direct effect) of \(a\) in \(\delta\). Note that the definition of \(de(a, \delta)\) extends the definition of \(de(a, s)\)

\(^4\)Whether a fluent literal or a set of fluent literals holds or possibly holds in a partial state, which is a consistent set of fluent literals, has been defined in Section 2.1.2
(see Eq. (8)) to the case of partial states. Intuitively, \(de(a, \delta)\) specifies the set of fluent literals that the agent knows to hold in the successor state. The set \(pe(a, \delta)\) on the other hand defines all fluent literals that may hold in the successor state and hence \(\neg pe(a, \delta)\) covers all the fluent literals that do not hold in the successor state. As a result, the set of fluent literals \(de(a, \delta) \cup (\delta \setminus \neg pe(a, \delta))\) definitely holds in the successor state. The transition function of the 0-approximation, \(\Phi^0\), between partial states is defined based on this observation.

**Definition 2.11** Let \(\mathcal{D}\) be a domain description without static causal laws. For any partial state \(\delta\) and action \(a\),

1. if \(a\) is not executable in \(\delta\) then

   \[\Phi^0(a, \delta) = \bot\]

2. otherwise,

   \[\Phi^0(a, \delta) = de(a, \delta) \cup (\delta \setminus \neg pe(a, \delta))\]

The partial state \(\Phi^0(a, \delta)\) is called the successor partial state of \(\delta\) as a result of the execution of \(a\) in \(\delta\). When \(a\) is clear from the context, we simply say that it is the successor partial state of \(\delta\). Observe that even if the domain description is non-deterministic, there is at most one successor partial state for an action and a partial state.

The transition function \(\Phi^0\) of a domain description \(\mathcal{D}\) can also be viewed as a transition diagram \(T^0(\mathcal{D})\) whose nodes are partial states and whose transitions are
defined by $\Phi^0$. Unlike $T(D)$, in the $T^0(D)$ diagram, for any state $\delta$ and action $a$ at most one of the arcs originating from $\delta$ is labeled with $a$.

We have the following proposition.

**Proposition 2.2** Let $D$ be a domain description without static causal laws. Let $\delta$ be a partial state and $s$ be a state such that $\delta \subseteq s$. For any action $a$, if $a$ is executable in $\delta$ then there exists a state $s'$ such that $\Phi(a, s) = \{s'\}$ and $\Phi^0(a, \delta) \subseteq s'$.

**Proof.** See Section B.2

Similarly to $\Phi$, we extend $\Phi^0$ to define the final partial state after the execution of a sequence of actions. The extended function, denoted by $\hat{\Phi}^0$ and defined below, is a mapping from partial states and sequences of actions into partial states.

**Definition 2.12** Let $D$ be a domain description without static causal laws. For any sequence of actions $\alpha$ and partial state $\delta$,

1. if $\alpha = \langle \rangle$ then

$$\hat{\Phi}^0(\alpha, \delta) = \delta$$

2. if $\alpha = \langle \beta, a \rangle$ where $\beta$ is a sequence of actions and $a$ is an action then

$$\hat{\Phi}^0(\alpha, \delta) = \begin{cases} \perp & \text{if } \hat{\Phi}^0(\beta, \delta) = \perp \\ \Phi^0(a, \hat{\Phi}^0(\beta, \delta)) & \text{otherwise} \end{cases}$$

Given the extended transition function $\hat{\Phi}^0$, the entailment relationship, $\models^0$, between an action theory and a query with respect to the 0-approximation semantics is defined as follows (recall that $\Delta$ is a set partial states).
Definition 2.13  Let \((\mathcal{D}, \Delta)\) be an action theory where \(\mathcal{D}\) is a domain description without static causal laws. We say that \((\mathcal{D}, \Delta)\) entails a query of the form \(\varphi\) after \(\alpha\) with respect to the 0-approximation semantics and write

\[(\mathcal{D}, \Delta) \models^0 \varphi\;\text{after}\;\alpha,\]

if for every \(\delta \in \Delta, \hat{\Phi}^0(\alpha, \delta) \neq \bot\) and \(\varphi\) is true in \(\hat{\Phi}^0(\alpha, \delta)\).

The following example demonstrates how \(\models^0\) can be used to reason about the effects of actions.

Example 2.9  Consider the domain description \(\mathcal{D}_{2.7}\) from Example 2.7 and let \(\Delta_0 = \{\emptyset\}\). We have

\[de(\text{flush}, \emptyset) = \{\neg \text{clogged}\}\;\text{and}\;pe(\text{flush}, \emptyset) = \{\neg \text{clogged}\}\]

Hence,

\[\hat{\Phi}^0((\text{flush}), \emptyset) = \Phi^0(\text{flush}, \emptyset) = \{\neg \text{clogged}\} = \delta_1\]

Furthermore, we have

\[de(\text{dunk}, \delta_1) = \{\text{clogged, } \neg \text{armed}\}\;\text{and}\;
\]

\[pe(\text{dunk}, \delta_1) = \{\text{clogged, } \neg \text{armed}\}\]

Thus,

\[\hat{\Phi}^0((\text{flush, dunk}), \emptyset) = \Phi^0(\text{dunk}, \delta_1) = \{\neg \text{armed, clogged}\}\]
This implies that

\[(D_{2.7}, \Delta_0) \models^0 \neg \text{armed after } \langle \text{flush, dunk} \rangle\]  

(14)

The soundness of the 0-approximation semantics with respect to the possible world semantics for action theories without concurrent actions is proved in its original paper [126]. The following theorem extends that result to action theories with concurrent actions.

**Theorem 2.1** Let \((D, \Delta)\) be an action theory, where \(D\) is a domain description without static causal laws. For any sequence of actions \(\alpha\) and fluent formula \(\varphi\),

\[(D, \Delta) \models^0 \varphi \text{ after } \alpha \text{ implies } (D, \Delta) \models^P \varphi \text{ after } \alpha\]

**Proof.** See Section B.3. \(\square\)

### 2.3 Planning with Incomplete Information

Planning is the problem of determining a sequence of actions \(\alpha = \langle a_1, a_2, \ldots, a_n \rangle\), called *plan*, that when executed can transform the domain from one state to another state that satisfies a predefined goal. When \(\alpha\) contains no action it is called an empty plan and denoted by \(\langle \rangle\). Furthermore, for a plan \(\alpha\) by \(\alpha[i]\) we denote the plan that consists of the \(i\) initial actions of \(\alpha\), i.e., \(\alpha[i] = \langle a_1, \ldots, a_i \rangle\) and by convenience \(\alpha[0] = \langle \rangle\). A planning problem instance is defined as follows.
Definition 2.14 A planning problem instance \( \mathcal{P} \) is a tuple \( \langle \mathcal{D}, \Delta, \mathcal{G} \rangle \) where \( \langle \mathcal{D}, \Delta \rangle \) is an action theory, and \( \mathcal{G} \) is a set of fluent literals describing the goal.

The notion of solutions of a planning problem instance is defined as follows.

Definition 2.15 Let \( \mathcal{P} = \langle \mathcal{D}, \Delta, \mathcal{G} \rangle \) be a planning problem instance. A plan \( \alpha \) is a solution of \( \mathcal{P} \) if \( \langle \mathcal{D}, \Delta \rangle \models^P \mathcal{G} \) after \( \alpha \) (where \( \mathcal{G} \) is interpreted as the conjunction of fluent literals in it).

If \( \Delta \) contains exactly one state and the domain description \( \mathcal{D} \) is deterministic then such \( \alpha \) is often referred to as a classical solution; otherwise it is called a conformant solution. Furthermore, if each \( a_i \) of \( \alpha \) is an elementary action then \( \alpha \) is called a sequential solution; otherwise it is called a concurrent solution.

Example 2.10 Consider the domain description \( \mathcal{D}_{2.1} \) from Example 2.1 Then, \( \mathcal{P}_{2.10} = \langle \mathcal{D}, \{\emptyset\}, \{\text{safe}\} \rangle \) is a planning problem instance.

We can check that

\[
\alpha_1 = \langle \text{flush}(1), \text{dunk}(1, 1), \text{flush}(1), \text{dunk}(2, 1) \rangle
\]

and

\[
\alpha_2 = \langle \{\text{flush}(1), \text{flush}(2)\}, \{\text{dunk}(1, 1), \text{dunk}(2, 2)\} \rangle
\]

are solutions of \( \mathcal{P}_{2.10} \). The former is a sequential solution, while the latter is a concurrent solution. \( \Box \)
The complexity of planning has been studied widely in-depth in the planning community, see e.g. [13, 28, 29, 39, 43, 147]. In this thesis, we are interested in the complexity of the following planning problem, which we call polynomial-length planning problem.

**Definition 2.16** The polynomial-length planning problem is stated as follows.

1. Given a polynomial $Q(n) \geq n$, a domain description $\mathcal{D}$, a partial state $\delta$, and a goal $G$,

2. determine whether there exists a solution $\alpha$ of $\mathcal{P} = \langle \mathcal{D}, \delta, G \rangle$ with $|\alpha| \leq Q(|\mathcal{D}|)$.

We have the following results [13] about the complexity of planning with respect to the possible world semantics and planning with respect to the 0-approximation semantics.

**Theorem 2.2** For situations without static causal laws, the polynomial-length planning problem is $\Sigma^2_P$-complete.

**Theorem 2.3** For situations without static causal laws, the polynomial-length planning problem with respect to the 0-approximation\(^5\) is NP-complete.

### 2.4 Logic Programming Under Answer Set Semantics

The idea of logic programming is due to Kowawski [70]. In this section, we will review the syntax of logic programs and the answer set semantics [47, 48].

\(^5\)A solution of a planning problem $\langle \mathcal{D}, \Delta, G \rangle$ with respect to the 0-approximation is defined as a sequence of actions $\alpha$ such that $(\mathcal{D}, \Delta) \models^0 G$ after $\alpha$
2.4.1 General Logic Programs

The language of a general logic program, like a first order language, contains object constants, functions constants and predicate constants. Terms are defined as in first order logic. Atoms are of the form $p(t_1, \ldots, t_n)$ where $t_i$'s are terms and $p$ is a predicate symbol of arity $n$.

A general logic program (sometimes referred to as normal logic program) $P$ is a collection of rules of the following form:

$$A_0 \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n$$

(15)

where $m \leq n$. The not in the above rule is called negation-as-failure. When $P$ is negation-as-failure free, it is called a definite program. For a rule $r$ of the form (15), we will denote by head($r$), pos($r$), neg($r$), body($r$), the atom $A_0$, the set of atoms $\{A_1, \ldots, A_m\}$, the set of atoms $\{A_{m+1}, \ldots, A_n\}$, and the set of atoms $\{A_1, \ldots, A_n\}$ respectively.

Formulae and rules not containing variables are said to be ground. The set of all ground atoms in the language of a program $P$ is called the Herbrand base of $P$ and will be denoted by $HB(P)$. A rule $r$ of the form (15) with variables will be viewed as the set of its ground instantiations. For this reason, whenever we say about a rule of the above form, we mean one of its instantiations unless otherwise stated.

There are several proposals for the semantics of a general logic program. In the following, we review the stable model semantics, which is due to Gelfond and Lifschitz,
from [47]. We will begin with the stable model semantics for a definite program and then extend it for a general logic program.

**Definition 2.17** Let $P$ be a definite program. The stable model of $P$ is the smallest subset $S$ of $HB(P)$ such that for each rule $r \in P$, if $\text{pos}(r) \subseteq S$ then $\text{head}(r) \in S$.

Note that in the above definition, we used “the” in front of “stable model” because it is proved that every definite program $P$ has exactly one stable model.

Let $P$ be a general logic program. For any set $S$ of atoms, let $P^S$ be the program obtained from $P$ by removing

1. every rule $r$ s.t. $\text{neg}(r) \cap S \neq \emptyset$, and

2. not $A$ in the bodies of the remaining rules.

This transformation is often referred to as the Gelfond-Lifschitz transformation and the program $P^S$ is called the reduct of $P$ with respect to $S$. Note that this transformation guarantees that $P^S$ is a definite program and thus it has the unique stable model as defined in Definition 2.17. The stable models of $P$ are defined as follows.

**Definition 2.18** Let $P$ be a general logic program. A set $S \subseteq HB(P)$ is a stable model of $P$ if $S$ is the stable model of $P^S$.

Let $S$ be a set of atoms. A ground atom $p$ is true in $S$ if $p \in S$, otherwise $p$ is false. The truth value of a formula $\varphi$ in $S$ is defined as usual. The following defines the entailment relationship between a general logic program and a (ground) query $\varphi$. 

49
Definition 2.19 Let $P$ be a general logic program and $\varphi$ is a ground query. We say that

- $P$ entails $\varphi$ and write $P \models \varphi$ if $\varphi$ is true in all stable models of $P$,

- $P$ entails $\neg \varphi$ and write $P \models \neg \varphi$ if $\neg \varphi$ is true in all stable models of $P$,

- otherwise $\varphi$ is unknown.

An interesting property of general logic programs is non-monotonicity. That is, adding new rules to a logic program may withdraw some of the previous conclusions. Another property is that a general logic program $P$ may have no stable model, one stable model or more than one stable model.

2.4.2 Extended General Logic Programs

There have been a number of extensions of general logic programs, one of which is the so-called extended general logic programs. Extended general logic program allows for another type of negation, called classical negation $\neg$ in addition to negation-as-failure $\text{not}$.

A literal $L$ is either an atom $A$ or its negation $\neg A$. Two literals $A$ and $\neg A$, where $A$ is an atom, are said complementary to each other. An extended logic program $P$ is a set of rules of the following form

$$L_0 \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$$ (16)
where \( L_i \)'s are literals. For an extended general logic program \( P \), let \( \text{lit}(P) \) denote the set of all ground literals in the language of \( P \). Similarly as in general logic programs, a rule (16) with variable will be viewed as the collection of all its ground instantiations. Therefore, whenever such a rule is referred to, we mean one of its instantiations unless otherwise stated.

In [48], the semantics of an extended logic program \( P \) is given in terms of answer sets. Basically, answer sets of an extended general logic program correspond to their counterpart, stable models, in general logic programs. Similarly to general logic programs, we first define the answer set for an extended general logic program without the negation-as-failure \( \text{not} \) and then extend it for an arbitrary extended general logic program.

**Definition 2.20** Let \( P \) be an extended logic program without \( \text{not} \). The answer set of \( P \) is the smallest set \( S \) of literals such that

- for any rule \( r \) of the form (16), if \( \text{pos}(r) \subseteq S \) then \( L_0 \in S \),
- if \( S \) contains a pair of complementary literals then \( S = \text{lit}(P) \).

For a set \( S \) of literals, the reduct of \( P \) with respect to \( S \), denoted by \( P^S \), is the program obtained from \( P \) by deleting

- each rule \( r \) such that \( \text{neg}(r) \cap S \neq \emptyset \), and
- \( \text{not} \) \( L \) in the remaining rules.
Clearly, \( P^S \) does not contain negation-as-failure. Therefore, it has a unique answer set as defined by Definition 2.20. If \( S \) is this unique answer set then it is an answer set of \( P \). Formally, the answer sets for \( P \) is defined as follows.

**Definition 2.21** Let \( P \) be an extended logic program. A set \( S \) of literals is an answer set of \( P \) if it is the answer set of \( P^S \).

Note that an answer set of \( P \) may contain both positive and negative literals. We will say that a literal \( l \) is true in an answer set \( S \) if \( l \in S \). The answer of \( P \) to a literal query \( l \) is true (resp. false) if \( l \) (resp. \( \neg l \)) is true in every answer set of \( P \); and unknown otherwise. This definition is then extended to the case of a ground formula \( \varphi \) as usual.

### 2.4.3 Disjunctive Logic Programs

A disjunctive logic program \( P \) is a set of rules \( r \) of the following form

\[
L_1 | \cdots | L_l \leftarrow L_{l+1}, \ldots, L_m, \neg L_{m+1}, \ldots, \neg L_n
\]  

(17)

The meaning of “\( | \)” in the head of the above rule is *or*. We will denote by \( \text{head}(r) \), \( \text{pos}(r) \), \( \text{neg}(r) \), and \( \text{lit}(r) \) the sets of literals \( \{L_1, \ldots, L_l\} \), \( \{L_{l+1}, \ldots, L_m\} \), \( \{L_{m+1}, \ldots, L_n\} \), and \( \{L_1, \ldots, L_n\} \).

When \( P \) does not contain \( \neg \), it is called a *positive disjunctive logic program*.

The answer sets for a positive disjunctive program are defined as follows.

**Definition 2.22** Let \( P \) be a positive disjunctive logic program. An answer set of \( P \) is a minimal (with respect to set inclusion \( \subseteq \)) set \( S \) of literals such that
• for any rule \( r \) of the form (17), if \( \text{pos}(r) \subseteq S \) then \( \text{head}(r) \cap S \neq \emptyset \),

• if \( S \) contains a pair of complementary literals then \( S \) is the set of all literals.

Note that unlike a positive extended logic program, a positive disjunctive logic program may have more than one answer set. For example, the following program

\[
p \mid q \leftarrow
\]

has two answer sets \( \{p\} \) and \( \{q\} \).

We now define the answer sets for an arbitrary disjunctive logic program \( P \). For a set \( S \) of literals, let \( P^S \) be the program obtained from \( P \) by deleting

• each rule \( r \) such that \( \neg \text{pos}(r) \cap S \neq \emptyset \), and

• not \( L \) in the remaining rules.

**Definition 2.23** Let \( P \) be an extended logic program. A set \( S \) of literals is an answer set of \( P \) if it is an answer set of \( P^S \).

We expand the notion of a query to a formula constructed from literals using the connectives \( \neg, \land, \lor \). The satisfaction of a query \( \varphi \) by a set \( S \) of literals is defined as follows

• if \( \varphi \) is a literal \( l \) then \( \varphi \) is true (resp. false) in \( S \) if \( l \in S \) (resp. \( \neg l \in S \)).

• if \( \varphi = \varphi_1 \land \varphi_2 \) then \( \varphi \) is true (resp. false) in \( S \) if \( \varphi_1 \) and \( \varphi_2 \) are true (resp. \( \varphi_1 \) or \( \varphi_2 \) is false) in \( S \).
• if $\varphi = \varphi_1 \lor \varphi_2$ then $\varphi$ is true (resp. false) in $S$ if $\varphi_1$ or $\varphi_2$ is true (resp. $\varphi_1$ and $\varphi_2$ are false) in $S$.

• if $\varphi = \neg \varphi_1$ then $\varphi$ is true (resp. false) in $S$ if $\varphi_1$ is false (resp. true) in $S$.

A formula $\varphi$ is said to be true (resp. false) with respect to a disjunctive logic program if it is true (resp. false) in all answer sets of the program; otherwise, it is unknown.

2.4.4 Splitting Set and Splitting Sequence Theorems

In this section, we will review two important theorems related to the answer set semantics – the splitting set theorem and the splitting sequence theorem – from [82]. Basically, these theorems say that a disjunctive logic program $P$ can be divided into several subprograms whose answer sets can be constructed from the answer sets of $P$ and vice versa. These theorems have proved their usefulness in proving the correctness of logic programs. Notice that because the class of general logic programs is a subset of the class of extended general logic programs, which is a subset of the class of disjunctive logic programs, the theorems presented here can be applied to general logic programs and extended general logic programs as well.

Let $P$ be a disjunctive logic program. For simplicity, we will assume that $P$ contains only ground rules.

A splitting set for $P$ is a set of literals $U$ such that for every rule $r \in P$, if $head(r) \cap U \neq \emptyset$ then $lit(r) \subseteq U$. The bottom of $P$ with respect to the splitting set $U$, denoted by $b_U(P)$, is the set of rules $r \in P$ such that $lit(r) \subseteq U$. The top part of $P$ is
\[ t_U(P) = P \setminus b_U(P). \]

Let \( U \) be a splitting set of \( P \) and \( X \) be a set of literals. For each \( r \in t_U(P) \) such \( pos(r) \cap X \subseteq U \) and \( neg(r) \cap U \) is disjoint from \( X \), consider the rule \( r' \) defined by:
\[
head(r') = head(r), pos(r') = pos(r) \setminus U, neg(r') = neg(r) \setminus U
\]

Let \( e_U(P, X) \) be the program consisting of rules \( r' \) obtained in this way. A solution to \( P \) (with respect to \( U \)) is a pair \( \langle X, Y \rangle \) of sets of literals such that
\begin{itemize}
  \item \( X \) is an answer set for \( b_U(P) \).
  \item \( Y \) is an answer set for \( e_U(t_U(P), X) \),
  \item \( X \cup Y \) is consistent.
\end{itemize}

**Theorem 2.4 (Splitting Set Theorem)** Let \( U \) be a splitting set for a program \( P \). A set of literals \( A \) is a consistent answer set for \( P \) if and only if \( A = X \cup Y \) for some solution \( \langle X, Y \rangle \) to \( P \) with respect to \( U \).

**Example 2.11** Consider the program \( P_1 \). Let \( U = \{a, b\} \). It is easy to see that \( U \) is a splitting set of \( P_1 \). \( b_U(P_1) = \{a \leftarrow \text{not } b, b \leftarrow \text{not } a\} \) and \( t_U(P_1) = \{c \leftarrow a, c \leftarrow b\} \). \( b_U(P_1) \) has two answer sets \( \{a\} \) and \( \{b\} \). \( e_U(t_U(P_1), \{a\}) = \{c \leftarrow \} \). Thus, \( e_U(t_U(P_1), \{a\}) \) has one answer set \( \{c\} \).

**Example 2.12** Consider the program \( P_2 \) as follows:

\[
a \leftarrow b, \text{not } c
\]
Let $U = \{c\}$. We have $b_U(P_2) = \{c\}$ and $t_U(P_2) = \{a \leftarrow b, not c; b \leftarrow c, not a\}$.

$b_U(P_2)$ has the unique answer set $X = \{c\}$. The evaluation of $t_U(P_2)$ with respect to $X$ is

$$e_U(t_U(P_2), X) = \{b \leftarrow not a\}$$

$Y = \{b\}$ is an answer set for $e_U(t_U(P_2), X)$, $X \cup Y$ is consistent. Thus, $\langle X, Y \rangle$ is a solution to $P_2$. According to the splitting theorem, $A = X \cup Y = \{b, c\}$ is an answer set for $P_2$.

The splitting set theorem is extended to the splitting sequence theorem as follows.

A sequence is a family whose index set is an initial segment of ordinals, $\{\alpha \mid \alpha < \mu\}$. The ordinal $\mu$ is called the length of the sequence. A sequence $\langle U_\alpha \rangle_{\alpha < \mu}$ of sets is monotone if for any $\alpha < \beta$, $U_\alpha \subseteq U_\beta$. It is continuous if for each limit ordinal $\alpha < \mu$, $U_\alpha = U_{\nu < \alpha}$.

A splitting sequence for $P$ is a monotone, continuous sequence $U = \langle U_\alpha \rangle_{\alpha < \mu}$ of splitting sets for $P$ such that $\bigcup_{\alpha < \mu} U_\alpha = \text{lit}(P)$. A solution to $P$ with respect to $U$ is a sequence $\langle X_\alpha \rangle_{\alpha < \mu}$ of sets of literals such that

- $X_0$ is an answer set for $b_{U_0}(P)$,
• for any $\alpha$ such that $\alpha + 1 < \mu$, $X_{\alpha+1}$ is an answer set for

$$e_{U_{\alpha}}(b_{U_{\alpha+1}}(P) \setminus b_{U_{\alpha}}(P), \bigcup_{\nu \leq \alpha} X_{\nu}),$$

• for any limit ordinal $\alpha < \mu$, $X_\alpha = \emptyset$,

• $\bigcup_{\alpha < \mu} X_\alpha$ is consistent.

**Theorem 2.5 (Splitting Sequence Theorem)** Let $U = \langle U_\alpha \rangle_{\alpha < \mu}$ be a splitting sequence for a program $P$. A set $A$ of literals is a consistent answer set for $P$ iff $A = \bigcup_{\alpha < \mu} X_\alpha$ for some solution $\langle X_\alpha \rangle_{\alpha < \mu}$ to $P$ with respect to $U$.

### 2.5 Summary

In this chapter, we reviewed one of the existing formalisms for reasoning about action and change, the action description language $\mathcal{AL}$, which can be used to describe domains with impossibility conditions, conditional effects, concurrent actions, and static causal laws. Central to $\mathcal{AL}$ are concepts of domain descriptions, action theories, states, transition function, queries, and entailment relationship. We also demonstrated by examples how $\mathcal{AL}$ can answer queries of a certain form. Then, we reviewed two approaches that extend the $\mathcal{AL}$ framework to deal with incompleteness of information: the possible world approach and the 0-approximation approach. While the former is applicable to arbitrary action theories, the latter only deals with action theories without static causal laws. The two approaches were illustrated by several concrete examples. Then, we briefly reviewed the concepts of planning problem, planning problem...
instances, and solutions of a planning problem instance, and the complexity of planning
with respect to the possible world semantics and the 0-approximation semantics. Also
in this chapter, we reviewed logic programming with answer set semantics.

In the next chapter, we will discuss about some of limitations of the possible
world approach and the 0-approximation approach, and show how we can address them.
CHAPTER 3
COMPLETENESS OF THE 0-APPROXIMATION

3.1 Introduction

Recall that in the possible world approach, to reason about the effects of an action (or a sequence of actions) with incomplete information about the initial state of the domain, an agent has to consider all possible initial states of the domain that are consistent with its knowledge. The truth value of a fluent formula after the execution of a sequence of actions is then determined by whether the formula is true in all possible final states. Following this approach, the problem of finding a (polynomial length) conformant solution is \( \Sigma_p^P \)-complete [13]. The 0-approximation approach, on the other hand, approximates the set of possible initial states by a single partial state and reasoning is performed based on the transitions between partial states. Consequently, using the 0-approximation approach can reduce the complexity of reasoning and planning tasks [13].

One of the main drawbacks of the 0-approximation approach, however, is its incompleteness with respect to the possible world approach, i.e., a reasoner using the 0-approximation semantics may answer a query about the truth value of a fluent formula after the execution of a sequence of actions with “unknown” while another reasoner using the possible world semantics would answer with either “true” or “false”. This
also implies that some planning problem instances that are “solvable” by conformant planners using possible world semantics turn out to be “unsolvable” by those using the 0-approximation semantics\(^1\).

A trivial method to overcome this drawback of the 0-approximation is to do exactly what the possible world approach does: considers all possible initial states. This solution is not satisfactory since (i) it does not scale up well; and (ii) in several cases, it is not necessary as can be seen in the following example.

**Example 3.1** Consider the following modification of the bomb-in-the-toilet domain description \(D_{2.7}\) from Example 2.7.

\[
D_{3.1} = \left\{ \begin{array}{l}
\text{dunk causes } \neg\text{armed if armed} \\
\text{dunk causes clogged} \\
\text{flush causes } \neg\text{clogged} \\
\text{impossible dunk if clogged}
\end{array} \right\}
\]

This domain description differs from its previous version, \(D_{2.7}\), only in the encoding of the effects of action dunk. While \(D_{2.7}\) states that the bomb will be disarmed after action dunk is executed no matter what the current state is, \(D_{3.1}\) states that the bomb will be disarmed only if it is not armed in the current state. With this minor change, one may expect that there should not be any change in the reasoning process of the agent. Unfortunately, this is not the case for the 0-approximation approach. Consider, for example, the following query from Example 2.7: whether the bomb is disarmed after the execution of the sequence of actions \(\langle\text{flush, dunk}\rangle\), given that the agent knows

\(^1\)By “solvable” (resp. “unsolvable”) we mean that the conformant planner returns a solution (resp. no solution)
nothing about the initial state of the domain, i.e., the set of initial partial states is $\Delta = \{\emptyset\}$.

Following the possible world approach (Figure 3.1), since both fluents $\text{armed}$ and $\text{clogged}$ are unknown in the beginning, the agent considers four possible initial states corresponding to different assignments of their truth values. After executing the action $\text{flush}$, the agent comes up with two possible states $s_1 = \{\text{armed}, \neg \text{clogged}\}$ and $s_2 = \{\neg \text{armed}, \neg \text{clogged}\}$ as flushing the toilet makes it unclogged. Executing the action $\text{dunk}$ afterwards leads to the final state $s_3 = \{\neg \text{armed}, \text{clogged}\}$ because if the previous state of the domain was $s_1$ then the precondition of the first dynamic causal law holds, causing the value of the fluent $\text{armed}$ to be false (or equivalently, the fluent literal $\neg \text{armed}$ holds) in the final state; otherwise (i.e., the previous state of the domain was $s_2$), the first dynamic causal law would not take effect and thus there would be no change in the value of fluent $\text{armed}$, i.e., the value of fluent $\text{armed}$ in the final state is still false. As the value of the fluent $\text{armed}$ is false in the final state, we can conclude
that the bomb will be disarmed if the above sequence of actions are executed.

\[ \text{flush} \rightarrow \neg \text{clogged} \quad \text{dunk} \rightarrow \text{clogged} \]

Figure 3.2: The 0-approximation approach

Following the 0-approximation approach (Figure 3.2), on the contrary, does not help us to draw that conclusion. The reason is that it approximates the four possible initial states by an empty partial state \(\delta_0 = \emptyset\). Executing the action \(\text{flush}\) in \(\delta_0\) changes the state of the domain to the partial state \(\delta_1 = \{\neg \text{clogged}\}\), according to the agent’s belief. Because \(\delta_1\) does not contain any information about the fluent \(\text{armed}\), when the action \(\text{dunk}\) is executed, the agent’s belief about the state of the domain is updated in the following steps. First, it adds to its current knowledge, i.e., \(\delta_1\), the fluent literal \(\text{clogged}\) because the precondition of the second dynamic causal law holds. It, however, does not add \(\neg \text{armed}\) to its current knowledge because it does not know whether the precondition of the first dynamic causal law holds or not. Second, it removes from its current knowledge what may no longer hold, which, in this case, is fluent literal \(\text{armed}\). Following these steps, the agent’s knowledge about the state of the domain after the action \(\text{flush}\) is executed is \(\delta_2 = \{\text{clogged}\}\). As \(\delta_2\) does not contain any information about the fluent \(\text{armed}\) (i.e., neither fluent literal \(\text{armed}\) nor fluent literal \(\neg \text{armed}\) belongs to \(\delta_2\)), the query is unknown to the agent.

It is easy to see that if the truth values of fluent \(\text{armed}\) were considered separately in the beginning (by partitioning the partial state \(\emptyset\) into two possible partial states.
Figure 3.3: The 0-approximation approach with sets of partial states

\{armed\} and \{-armed\}) by the agent then the value of fluent \textit{armed} after the execution of \langle \textit{flush}, \textit{dunk} \rangle would be known (see Figure 3.3) to the agent. As a result, it would be able to conclude that the bomb will be disarmed after that sequence of actions is executed. Partitioning the initial partial state over \{-clogged\}, on the contrary, does not help the agent to draw the conclusion.

The above example shows that in some cases, the 0-approximation approach may lose some conclusions with respect to the possible world approach. This also implies that any conformant planner based on the 0-approximation is in general incomplete, i.e, it may not find a solution even when one exists. This raises the following question: "Under what conditions does the 0-approximation agree with the possible world semantics on answering the truth value of a fluent formula?" Answering this question is important for different reasons. First, it provides a characterization for situations in which the 0-approximation can be used instead of the possible world semantics, thus, reducing the complexity of reasoning and planning in theories with incomplete information. Second, it can help identify the relevant knowledge that an agent needs to acquire about
the initial state of the domain in order for him/her to find a course of actions achieving a predefined goal. Furthermore, it can provide us with the theoretical basis for the development of a reasoning method that does not need to examine all possible initial states, making it more efficient than reasoning method based on the possible world semantics.

In this chapter, we present a sufficient condition for which reasoning with action theories using the 0-approximation semantics is complete. Then, we introduce a notion called decisive sets for partial states that allow us to partition the set of initial states so as to achieve complete reasoning with the 0-approximation semantics. To evaluate our method, we develop a sound and complete conformant planning system based on this idea, called CPA$^+$, and evaluate its performance with some other state-of-the-art planners. Since the 0-approximation only considers action theories without static causal laws, we limit ourselves to that case.

The main contributions of the work presented in this chapter are:

- A sufficient condition for which reasoning with an action theory using the 0-approximation is complete.

- The notion of decisive sets which can be used to modify an action theory in such a way that the 0-approximation semantics of the modified theory is complete with respect to the possible world semantics of the original theory.

- A polynomial time algorithm for computing decisive sets.

- A sound and complete conformant planning system based on the 0-approximation
whose performance is experimentally shown to be competitive with other state-of-the-art conformant planners.

The remainder of the chapter is organized as follows. Section 3.2 presents a sufficient condition for which the 0-approximation semantics is complete. Section 3.3 discusses how to make reasoning with the 0-approximation semantics complete. Section 3.4 describes the conformant planner $\text{CPA}^+$ and shows some experimental results. Section 3.5 provides discussion and related work. Section 3.6 summarizes the chapter. The proofs of propositions and theorems in the chapter are listed in Appendix C.

3.2 A Sufficient Condition for the Completeness of $|=^0$

As discussed previously, the main disadvantage of the 0-approximation is its incompleteness if $\Delta$ — the set of initial partial states — does not contain sufficient information for its reasoning. In this section, we provide a condition on action theories for which the 0-approximation semantics yields the same answer to a query as the possible world semantics does. Let us begin with some definitions.

**Definition 3.1** Let $(\mathcal{D}, \Delta)$ and $(\mathcal{D}, \Delta^*)$ be two action theories and $\varphi$ be a fluent formula. We say that reasoning with $(\mathcal{D}, \Delta^*)$ using the 0-approximation semantics and reasoning with $(\mathcal{D}, \Delta)$ using the possible world semantics agree with each other on $\varphi$, denoted by

$$ (\mathcal{D}, \Delta^*)|=^0 \varphi \simeq (\mathcal{D}, \Delta)|=^p \varphi $$. 

65
if for any sequence of actions $\alpha$, we have that

$$(D, \Delta^*) \models^0 \varphi \text{ after } \alpha \iff (D, \Delta) \models^P \varphi \text{ after } \alpha.$$  

The intuitive meaning of $\simeq$ is that $(D, \Delta^*) \models^0 \varphi \Leftrightarrow (D, \Delta) \models^P \varphi$ implies that reasoning with action theory $(D, \Delta^*)$ using the 0-approximation semantics yields the same answer to the query

$$\varphi \text{ after } \alpha$$

as reasoning with $(D, \Delta)$ using the possible world semantics.

Then the problems of our interest are: given an action theory $(D, \Delta)$ and a fluent formula $\varphi$,

(1) under what conditions,

$$(D, \Delta) \models^0 \varphi \Leftrightarrow (D, \Delta) \models^P \varphi$$

and

(2) how to find a set of partial states $\Delta^*$ such that

$$(D, \Delta^*) \models^0 \varphi \Leftrightarrow (D, \Delta) \models^P \varphi.$$  

In this section, we tackle the first problem. In the next section, we will present a solution to the second problem. Without loss of generality, we assume that $\varphi$ is given in the conjunctive normal form (CNF), i.e.,

$$\varphi = \gamma_1 \land \gamma_2 \land \cdots \land \gamma_n$$
where $\gamma_i$ is a disjunction of fluent literals

$$\gamma_i = l_{i_1} \lor \cdots \lor l_{i_k}$$

Observe that the incompleteness of $|=^0$ comes from the lack of information about necessary fluents in the beginning. Let us go back to the example in the very beginning of this chapter. The reason that the fluent literal $\neg armed$ does not appear in the final partial state is because the agent could not determine the truth value of the precondition, $armed$, of the dynamic causal law

$$dunk \ causes \ \neg armed \ if \ armed$$

in the partial state $\delta_1 = \{\neg clogged\}$ (the partial state after the execution of the action $flush$). In other words, the fluent literal $\neg armed$ (the head of the dynamic causal law) depends on the fluent literal $armed$ (the precondition of the law) and the information about the latter is missing in the partial state $\delta_1$. This missing indeed is due to the missing of the information about the fluent $armed$ in the initial partial state.

The above observation indicates that by analyzing the dependencies among fluent literals and actions we may be able to identify fluents whose truth values need to be known in the beginning in order for the 0-approximation to be complete. Let us formalize this idea.

**Definition 3.2** Let $\mathcal{D}$ be a domain description without static causal laws. A fluent literal $l$ depends on a fluent literal $g$, written as $l \preceq g$, iff one of the following conditions holds.
1. \( l = g \).

2. \( D \) contains a dynamic causal law

   \( e \text{ causes } l \text{ if } \psi \)

   such that \( g \in \psi \).

3. There exists a fluent literal \( h \) such that \( l \rightarrow h \) and \( h \rightarrow g \).

4. The complement of \( l \) depends on the complement of \( g \), i.e., \( \neg l \rightarrow \neg g \).

Note that the dependency relationship between fluent literals is reflexive, transitive but not symmetric. The next definition is about the dependency between actions and fluent literals.

**Definition 3.3** Let \( D \) be a domain description without static causal laws. An action \( a \) depends on a fluent literal \( l \), written as \( a \rightarrow l \), iff one of the following conditions is satisfied.

1. \( D \) contains an impossibility condition

   \( \text{impossible } b \text{ if } \psi \)

   such that \( b \subseteq a \) and \( \neg l \in \psi \).

2. There exists a fluent literal \( g \) such that \( a \rightarrow g \) and \( g \rightarrow l \).
For a fluent literal \(l\) (resp. action \(a\)), we denote by \(\Omega(l)\) (resp. \(\Omega(a)\)) the set of fluent literals that \(l\) (resp. \(a\)) depends on. A fluent literal \(l\) (resp. an action \(a\)) depends on a set of fluent literals \(\sigma\), denoted by \(l \triangleleft \sigma\) (resp. \(a \triangleleft \sigma\)), iff \(l \triangleleft g\) (resp. \(a \triangleleft g\)) for some \(g \in \sigma\).

A disjunction of fluent literals \(\gamma = l_1 \lor \cdots \lor l_k\) depends on a set of literals \(\sigma\), denoted by \(\gamma \triangleleft \sigma\), if there exists \(1 \leq j \leq k\) such that \(l_j \triangleleft \sigma\); otherwise, \(\gamma\) does not depend on \(\sigma\), denoted by \(\gamma \ntriangleleft \sigma\).

**Example 3.2**  For the domain description \(D_{3.1}\) (Example 3.1), we have

\[
\begin{align*}
\Omega(clogged) &= \{clogged\} & \Omega(\neg clogged) &= \{\neg clogged\} \\
\Omega(armed) &= \{armed, \neg armed\} & \Omega(\neg armed) &= \{armed, \neg armed\} \\
\Omega(dunk) &= \{\neg clogged\} & \Omega(flush) &= \emptyset \\
\end{align*}
\]

We have defined dependencies among fluent literals and actions. Let us see how this information can be used in determining essential fluents whose truth values need to be known in the beginning in order for the 0-approximation to be complete. Recall that, in the possible world approach, when the initial state of the domain is described by a single partial state \(\delta_0\), i.e., \(\Delta = \{\delta_0\}\), the agent initiates its belief state to \(S = \text{ext}(\delta_0)\).

When an action is executed, this belief state is updated according to the \(\Phi^P\) function. To reason about the truth value of a formula \(\varphi\) at a time, it checks whether the formula is true in the belief current state \(S\). In the 0-approximation approach, on the contrary, the agent only maintains a partial state \(\delta\) that represents its knowledge about the current state of the domain. Initially, \(\delta\) is set to \(\delta_0\) and each time an action is executed, this
partial state is updated according to the $\Phi^0$ function. To reason about the truth value of $\varphi$, it checks whether the formula is true in the current partial state.

From this observation we can see that to find a sufficient condition for the completeness of $\models^0$, it is important to characterize when a belief state $S$ and a partial state $\delta$ agree with each other on the truth value of a fluent formula. We therefore define the notion of reducibility as follows.

**Definition 3.4** Let $D$ be a domain description without static causal laws, $S$ be a belief state, $\delta$ be a partial state, and $\varphi = \gamma_1 \land \cdots \land \gamma_n$ be a fluent formula. We say that $S$ is reducible to $\delta$ with respect to $\varphi$, denoted by $S \gg^\varphi \delta$ if

1. $\delta$ is a subset of every state $s$ in $S$,
2. for every $1 \leq i \leq n$, there exists a state $s \in S$ such that $\gamma_i \not\models (s \setminus \delta)$, and
3. for any action $a$, there exists a state $s \in S$ such that $a \not\models (s \setminus \delta)$.

**Example 3.3** Consider the domain description $D_{3.1}$ from Example 3.1. Let $\delta_1 = \emptyset$. Then, $S_1 = \text{ext}(\delta_1)$ is not reducible to $\delta_1$ with respect to $\varphi = \neg\text{armed}$, i.e.,

$$\text{ext}(\emptyset) \not\gg_{\neg\text{armed}} \emptyset$$

because for each $s \in S_1$, either fluent literal $\text{armed}$ or $\neg\text{armed}$ belongs to $s \setminus \delta_1$ and the fluent literal $\neg\text{armed}$ depends on both $\text{armed}$ and $\neg\text{armed}$ (Condition 2 in Definition 2).
3.4 is not satisfied). For the same reason, the following results hold

\[ \text{ext}\{\text{clogged}\} \succ_{\neg \text{armed}} \{\text{clogged}\} \]

\[ \text{ext}\{\neg \text{clogged}\} \succ_{\neg \text{armed}} \{\neg \text{clogged}\} \]

Now let \( \delta_2 = \{\text{armed}\} \). Then we have \( \text{ext}(\delta_2) = \{s_1, s_2\} \), where \( s_1 = \{\text{armed, clogged}\} \) and \( s_2 = \{\text{armed, \neg clogged}\} \). We will show that

\[ \text{ext}(\delta_2) \succ_{\neg \text{armed}} \delta_2 \]

because of the following reasons.

1. \( \neg \text{armed} \not\in (s_1 \setminus \delta_2) = \{\text{clogged}\} \) because \( \neg \text{armed} \not\in \text{clogged} \).

2. For every action \( a \), it is easy to see that \( a \not\in (s_1 \setminus \delta_2) = \{\text{clogged}\} \) because both elementary actions \( \text{flush} \) and \( \text{dunk} \) do not depend on \( \text{clogged} \).

Hence, we have

\[ \text{ext}\{\text{armed}\} \succ_{\neg \text{armed}} \{\text{armed}\} \]

Similarly, we can check that

\[ \text{ext}\{\neg \text{armed}\} \succ_{\neg \text{armed}} \{\neg \text{armed}\} \]
There are two interesting properties about the reducibility of a belief state. First, if $S \gg_{\varphi} \delta$ and $S$ represents the set of possible states of the domain then to reason about the fluent formula $\varphi$, it suffices to know $\delta$. Second, the reducibility of a belief state is preserved along the course of the execution of actions. These properties are stated in the next two propositions.

**Proposition 3.1** Let $\mathcal{D}$ be a domain description without static causal laws. Let $S$ be a belief state, $\delta$ be a partial state, and $\varphi$ be a fluent formula such that $S \gg_{\varphi} \delta$. Then, $\varphi$ is true in $S$ iff $\varphi$ is true in $\delta$.

**Proof.** See Section C.1.1.

**Proposition 3.2** Let $\mathcal{D}$ be a domain description without static causal laws. Let $S$ be a belief state, $\delta$ be a partial state, and $\varphi$ be a fluent formula such that $S \gg_{\varphi} \delta$. For any action $a$, if $a$ is executable in $S$ then

1. $a$ is executable in $\delta$, and

2. $\Phi^P(a, S) \gg_{\varphi} \Phi^0(a, \delta)$.

**Proof.** See Section C.1.2.

The second property can be extended to a sequence of actions $\alpha$ as follows.
Proposition 3.3 Let $\mathcal{D}$ be a domain description without static causal laws. Let $S$ be a belief state, $\delta$ be a partial state, and $\varphi$ be a fluent formula such that $S \gg_\varphi \delta$. For any sequence of actions $\alpha$, if $\hat{\Phi}^P(\alpha, S) \neq \bot$ then

1. $\hat{\Phi}^0(\alpha, \delta) \neq \bot$, and

2. $\hat{\Phi}^P(\alpha, S) \gg_\varphi \hat{\Phi}^0(\alpha, \delta)$.

Proof. See Section C.1.3.

From Propositions 3.1 and 3.3, we have the following theorem.

Theorem 3.1 Let $(\mathcal{D}, \Delta)$ be an action theory without static causal laws and $\varphi$ be a fluent formula. If $\text{ext}(\delta) \gg_\varphi \delta$ for every $\delta \in \Delta$ then $(\mathcal{D}, \Delta) \models^0 \simeq_\varphi (\mathcal{D}, \Delta) \models^p$.

Proof. See Section C.1.4.

This theorem serves as a sufficient condition for the completeness of $\models^0$ with respect to $\varphi$. Let us illustrate the intuitive meaning of the theorem by an example.

Example 3.4 From Example 3.3, for the domain $\mathcal{D}_{3.1}$, we have

$$\text{ext}(\{\text{armed}\}) \gg_{\neg \text{armed}} \{\text{armed}\}$$

and

$$\text{ext}(\{\neg \text{armed}\}) \gg_{\neg \text{armed}} \{\neg \text{armed}\}$$
Therefore, it follows from the above theorem that for

\[(D_{3.1}, \{(\text{armed}), \{\neg\text{armed}\}\}) |\not\models_{\text{armed}} (D_{3.1}, \{(\text{armed}), \{\neg\text{armed}\}\}) |\not\models_{\text{armed}}.\]

\[\square\]

Observe that the condition in the theorem is a sufficient but not necessary condition for the completeness of \(\models^0\). The following illustrates this point.

**Example 3.5** Consider the following action theory

\[D_{3.5} = \begin{cases} 
  e \text{ causes } f \text{ if } g \\
  e \text{ causes } f \text{ if } g, \neg f
\end{cases}\]

and let \(\Delta = \{g\}\). We can check that the fluent literal \(f\) depends on both fluent literals \(f\) and \(\neg f\) and thus \(ext(\{g\}) \not\models_f \{g\}\). However, we have

\[(D_{3.5}, \Delta) |\not\models^0 \not\models_f (D_{3.5}, \Delta) |\not\models^0.\]

\[\square\]

### 3.3 Complete Reasoning Using \(\models^0\)

The results presented in the previous section can be used to achieve complete reasoning using the 0-approximation. In Example 3.1, we mentioned that it is possible to make \(\models^0\) complete with respect to \(\models^P\) without having to examine all possible initial states, by partitioning the initial partial state \(\emptyset\) into the set of partial states
{{\{\texttt{armed}\}, \{\neg \texttt{armed}\}\}}. Why did we choose the set of fluents {\texttt{armed}} but not {\texttt{clogged}} to partition the initial partial state, although both fluents {\texttt{armed}} and {\texttt{clogged}} are unknown in the beginning? In this section, we will provide an answer to this question.

Let us begin with some definitions.

**Definition 3.5** Let \( \mathcal{D} \) be a domain description without static causal laws. Let \( \delta \) be a partial state, and \( F \) be a (possibly empty) set of fluents unknown in \( \delta \). A partition of \( \delta \) over \( F \), denoted by \( \Delta_\delta^F \), is the following set of partial states

\[
\Delta_\delta^F = \{ \delta \cup I \mid I \text{ is an interpretation of } F \}
\]

**Definition 3.6** Let \( \mathcal{D} \) be a domain description without static causal laws, \( \delta \) be a partial state, and \( \varphi \) be a fluent formula. A set \( F \) of fluents is called a decisive set for \( \delta \) with respect to \( \varphi \) if the following conditions are satisfied.

- Every fluent \( f \in F \) is unknown in \( \delta \).
- For every interpretation \( I \) of \( F \), \( \text{ext}(\delta \cup I) \gg_{\varphi} (\delta \cup I) \).

**Example 3.6** Consider the domain description \( \mathcal{D}_{3.1} \) and let \( \delta_1 = \emptyset \). Let \( F_1 = \{\texttt{armed}\} \).

Then, \( F_1 \) has two interpretations \( I_1 = \{\texttt{armed}\} \) and \( I_2 = \{\neg \texttt{armed}\} \). It follows from Example 3.3 that

\[
\text{ext}(\delta_1 \cup I_1) = \text{ext}(\{\texttt{armed}\}) \gg_{\neg \texttt{armed}} \{\texttt{armed}\} = \delta_1 \cup I_1
\]

and

\[
\text{ext}(\delta_1 \cup I_2) = \text{ext}(\{\neg \texttt{armed}\}) \gg_{\neg \texttt{armed}} \{\neg \texttt{armed}\} = \delta_1 \cup I_2
\]
As a result, by definition, $F_1$ is a decisive set for $\delta$ with respect to $\neg \text{armed}$. Similarly, we can check that $F_2 = \{\text{armed, clogged}\}$ is a decisive set for $\delta$ with respect to $\neg \text{armed}$.

Now consider $F_3 = \{\text{clogged}\}$. From Example 3.3, we have

$$\text{ext}(\delta_1 \cup \{\text{clogged}\}) = \text{ext}(\{\text{clogged}\}) \not\gg_{\neg \text{armed}} \{\text{clogged}\}$$

As a result, by definition, $F_3$ is not a decisive set for $\delta_1$ with respect to $\neg \text{armed}$.

We have the following theorem (recall that $\Delta^\delta_F$ is the partition of $\Delta$ over $F$).

**Theorem 3.2** Let $(D, \Delta)$ be an action theory without static causal laws and let $\varphi$ be a fluent formula. For every $\delta \in \Delta$, let $F_\delta$ be a decisive set for $\delta$ with respect to $\varphi$. Define

$$\Delta^* = \bigcup_{\delta \in \Delta} \Delta^\delta_F$$

Then, we have

$$(D, \Delta^*)_{\models^0} \simeq_{\varphi} (D, \Delta)_{\models^P}$$

**Proof.** See Section C.2.1.

This theorem implies that we can make $\models^0$ complete if for each $\delta \in \Delta$, a decisive set for $\delta$ can be found. Trivially, for every $\delta$, the set of fluents unknown in $\delta$, $U_\delta$, is always a decisive set for $\delta$. It is, however, important to note that the number of
interpretations of $U_\delta$ is exponential in the size of $U_\delta$. Hence, given a partial state $\delta$, we wish to find a decisive set for $\delta$ that is as small as possible (with respect to set inclusion $\subseteq$). To do so, we develop an algorithm for computing a decisive set (Figure 3.4). The algorithm is based on the notion of dependencies in Definitions 3.2 & 3.3.

\begin{verbatim}
DECISIVE(D, \delta, \varphi)
INPUT: a domain $D$, a partial state $\delta$, and a formula $\varphi = \gamma_1 \land \cdots \land \gamma_n$
OUTPUT: a decisive set for $\delta$ w.r.t. $\varphi$
1. BEGIN
2. $F = \emptyset$
3. compute dependencies between fluent literals
4. compute dependencies between actions and fluent literals
5. for each fluent $f$ unknown in $\delta$ do
6. if there exists $1 \leq i \leq n$ s.t. $\gamma_i$ depends on both $f$ and $\neg f$ or
7. an action $a$ s.t. $a$ depends on both $f$ and $\neg f$
8. then $F = F \cup \{f\}$
9. return $F$;
10. END
\end{verbatim}

Figure 3.4: An algorithm for computing a decisive set

The following proposition states the correctness of the algorithm.

**Proposition 3.4** Let $D$ be a domain description without static causal laws, $\delta$ be a partial state and $\varphi$ be a fluent formula. Then, the set of fluents returned by DECISIVE($D, \delta, \varphi$) is a decisive set for $\delta$ with respect to $\varphi$.

**Proof.** See Section C.2.2.

Although simple and somewhat naive, the algorithm is worth some discussion. According to the algorithm, an unknown fluent $f$ belongs to the returned set $F$ if there
exists a conjunct $\gamma \in \varphi$ or an action $a$ that depends on both $f$ and $\neg f$. This guarantees that $F$ is a decisive set for $\delta$ with respect to $\varphi$ because for every interpretation $I$ of $F$, $\text{ext}(\delta \cup I)$ is reducible to $\delta \cup I$ with respect to $\varphi$. The main weakness of this algorithm is that it does not guarantee the minimality of $F$. Observe that we could derive a generate-and-test algorithm which returns a smaller decisive sets, based on the definition of the reducibility of a set of states (Definition 3.4). Nevertheless, we still adopt the above algorithm in the development of our planner (to be described in the next section) for two reasons. First, it is computationally efficient (its run time is polynomial in the size of the domain). Second, for a majority of the benchmark problems, we notice that the decisive set returned by the algorithm is empty, which is already as small as possible.

For a domain description $D$, if we define the size of $D$ to be the sum of (1) the number of fluents; (2) the number of actions; and (3) the number of statements in $D$, then the running time of the algorithm is polynomial in the size of $D$ as stated in the following proposition.

**Proposition 3.5** Let $D$ be a domain description without static causal laws, $\delta$ be a partial state, and $\varphi$ be a fluent formula. Then, the running time of the algorithm DECISIVE($D, \delta, \varphi$) is polynomial in the size of $D$ and $\varphi$.

**Proof.** See Section C.2.3.

**Example 3.7** Consider the action theory ($D_{3.1}, \emptyset$). Then, DECISIVE($D_{3.1}, \emptyset, \neg \text{armed}$)
returns \( \{\text{armed}\} \). The partition of \( \emptyset \) over \( \{\text{armed}\} \) is \( \{\{\text{armed}\}, \{\neg \text{armed}\}\} \). By Theorem 3.2, we have

\[
(D_{3.1}, \{\{\text{armed}\}, \{\neg \text{armed}\}\})|_{\emptyset} \simeq_{\neg \text{armed}} (D_{3.1}, \{\emptyset\})|_{\emptyset}
\]

\( \square \)

### 3.4 Application to Conformant Planning

Recall that in conformant planning, we are given an action theory and a goal and the task is to find a sequence of actions that leads to a goal state (i.e., a state satisfying the goal) from any possible initial state. In this section, we describe a conformant planning system, called \( \text{CPA}^+ \), that exploits the results in the previous sections. \( \text{CPA}^+ \) takes as input a planning problem instance and returns as output a solution of the problem or \text{FAILURE}. It can handle domains without static causal laws and without concurrent actions only. As a result, solutions returned by \( \text{CPA}^+ \) do not contain concurrent actions. \( \text{CPA}^+ \) is a complete conformant planner in the following sense. If the input problem has a conformant solution then \( \text{CPA}^+ \) will return a solution. If the input problem has no solution then \( \text{CPA}^+ \) returns \text{FAILURE}.

#### 3.4.1 The Conformant Planning System \( \text{CPA}^+ \)

A typical input file for a planning problem instance \( P = \langle D, \Delta, G \rangle \) is made up of four sections. The first section declares actions and fluents that are used in the planning problem instance using keywords \text{action} and \text{fluent}. The second section
% fluent and action declarations
fluent armed, clogged;
action dunk, flush;

% dynamic causal laws
dunk causes -armed if armed;
dunk causes clogged;
flush causes -clogged;

% impossibility conditions
impossible dunk if clogged;

% initial state: nothing is known
% goal: the bomb is disarmed
goal -armed;

Figure 3.5: Encoding of the bomb-in-the-toilet domain

describes effects and impossibility conditions of actions, i.e., statements of the forms (3) & (5). The third section describes the initial state of the domain, which contains statements of the form

initially \( \varphi \)

where \( \varphi \) is a fluent formula CNF formula. The last section describes the goal \( \mathcal{G} \) of the planning problem instance, which consists of statements of the form

goal \( \varphi \)

where \( \varphi \) is a fluent formula in CNF. Figure 3.5 shows a sample input file for a planning problem instance in CPA\(^+\).

CPA\(^+\) is implemented in C++. It employs the forward, best-first search strategy with repeated state avoidance and uses the number of fulfilled subgoals as the heuristic
function. The search algorithm of CPA+ is outlined in Figure 3.6. It uses a priority queue, Queue, to store belief partial states\(^3\) that have been explored so far and a table Visited to store visited belief partial states. The priority queue Queue contains elements of the form \((\Delta_1, p)\), where \(\Delta_1\) is a belief partial state and \(p\) is the path from the initial belief partial state to \(\Delta_1\), sorted according to the heuristic value of belief partial states.

The first step of the algorithm (Line 2) is to compute \(\Delta^*\) – the partition of the initial belief partial state \(\Delta\) over decisive sets for \(\Delta\) with respect to \(G\). This computation is done by a call to PARTITION\((D,\Delta,G)\) (see Figure 3.7) which computes decisive sets for each \(\delta\) in \(\Delta\) using the algorithm in Figure 3.4 and then returns the partition of \(\Delta\) over these decisive sets. Then, the algorithm checks whether \(\Delta^*\) satisfies \(G\) (i.e., \(G\) holds in every \(\delta \in \Delta^*\)). If so, then the algorithm returns the empty plan \(\langle \rangle\) as the solution of the planning problem (Line 3). Otherwise, it initializes the queue and the visited table to \\{\((\Delta^*, \langle \rangle)\)\} and \\{\(\Delta^*\)\} respectively (Line 4).

The main part of the algorithm is a while loop (Lines 5-13). In each iteration of the loop, the algorithm picks up a belief partial state \(\Delta_1\) with the best heuristic function from the queue (Line 6), computes the successor belief partial state \(\Delta_2\) for each action that is executable in \(\Delta_1\) (Lines 7–8), and checks whether \(\Delta_2\) satisfies the goal or not (Line 9). If so, it returns the path to \(\Delta_2\) as the solution of a planning problem instance.

\(^3\)A belief partial state is a set of partial states
\textbf{FwdPlan}(\mathcal{D}, \Delta, \mathcal{G})
\textbf{INPUT}: A planning problem \mathcal{P} = \langle \mathcal{D}, \Delta, \mathcal{G} \rangle 
\textbf{OUTPUT}: A solution of \mathcal{P} or \text{FAILURE}
1. \text{BEGIN}
2. \Delta^* = \text{Partition}(\mathcal{D}, \Delta)
3. \text{if } \Delta^* \text{ satisfies } \mathcal{G} \text{ then return } \langle \rangle 
4. \text{Queue} = \{ (\Delta^*, \langle \rangle) \} \quad \text{Visited} = \{ \Delta^* \}
5. \text{while } \text{Queue is not empty}
6. \text{select } (\Delta_1, p) \text{ with the best heuristic value from } \text{Queue}
7. \text{for each action } a \text{ executable in } \Delta_1
8. \Delta_2 = \bigcup_{\delta \in \Delta_1} \text{Resolv}(a, \delta)
9. \text{if } \Delta_2 \text{ satisfies } \mathcal{G} \text{ then return } (p, a)
10. \text{else if } \Delta_2 \notin \text{Visited}
11. \text{compute heuristic for } \Delta_2
12. \text{insert } (\Delta_2, \langle p, a \rangle) \text{ into } \text{Queue}
13. \text{insert } \Delta_2 \text{ into } \text{Visited}
14. \text{return } \text{FAILURE}
15. \text{END}

Figure 3.6: The search algorithm of CPA$^+$

Otherwise, it computes the heuristic for $\Delta_2$ and inserts it into the queue, and the visited table as well (Lines 10–13) if $\Delta_2$ has not been visited. If the queue is empty and no solution is found then it returns \text{FAILURE}.

\textbf{Partition}(\mathcal{D}, \Delta, \varphi)
\textbf{INPUT}: A domain description $\mathcal{D}$, a set of partial states $\Delta$, and a formula $\varphi$
\textbf{OUTPUT}: The partition of $\Delta$ over decisive sets for $\Delta$ w.r.t. $\varphi$
1. \text{BEGIN}
2. \text{for each } \delta \in \Delta
3. \quad F_\delta = \text{Decisive}(\mathcal{D}, \delta, \varphi)
4. \text{return } \bigcup_{\delta \in \Delta} \Delta^\delta_{F_\delta}
5. \text{END}

Figure 3.7: Partitioning the initial belief partial state

The procedure for computing the successor partial state is depicted in Figure 3.8. It takes as input a domain description $\mathcal{D}$, an action $a$ and a partial state $\delta$ and
returns as output the successor partial state of $\delta$ after $a$ is executed. It iterates over the set of dynamic causal laws of $\mathcal{D}$ to compute the set of direct effects of $a$, $de(a, \delta)$, and the set of possible direct effects of $a$, $pe(a, \delta)$ and then returns $de(a, \delta) \cup (\delta \setminus pe(a, \delta))$ as the successor partial state.

<table>
<thead>
<tr>
<th>RES0(D,a,\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT: An action theory $\mathcal{D}$, an action $a$, and a partial state $\delta$</td>
</tr>
<tr>
<td>OUTPUT: successor partial state of $\delta$</td>
</tr>
<tr>
<td>1. BEGIN</td>
</tr>
<tr>
<td>2. $de = \emptyset$ $pe = \emptyset$</td>
</tr>
<tr>
<td>3. for each dynamic causal law (3) in $\mathcal{D}$ do</td>
</tr>
<tr>
<td>4. if $\psi$ possibly holds in $\delta$ then</td>
</tr>
<tr>
<td>5. $pe = pe \cup {l}$</td>
</tr>
<tr>
<td>6. if $\psi$ holds in $\delta$ then</td>
</tr>
<tr>
<td>7. $de = de \cup {l}$</td>
</tr>
<tr>
<td>8. return $(de \cup (\delta \setminus pe))$</td>
</tr>
<tr>
<td>9. END</td>
</tr>
</tbody>
</table>

Figure 3.8: Computing the successor partial state

The heuristic for a belief partial state employed in the algorithm is defined to be the number of subgoals that hold in the first partial state of the belief partial state.

### 3.4.2 Experiments

To evaluate the performance of CPA$^+$, we compared it with three state-of-the-art conformant planners Conformant-FF (CFF) [23], KACMBP [34], and POND [36]. These planners are known as the current fastest conformant planners on most of the conformant planning benchmarks in the literature: CFF is superior to other state-of-the-art conformant planners like GPT [20], MBP [33]; KACMBP is reported [34] to outperform $\mathbb{DLV}^K$ and $\mathcal{C}$-PLAN in many domains in the literature. A short description
of these planners is given in Appendix A.

The domains used in our experiments are the bomb-in-the-toilet, Ring, Logistics, and Cleaner. A short description of these domains is given below.

- **Bomb-In-The-Toilet domain** [100]: In this domain, it has been alarmed that there might be a bomb in a lavatory. There are \( m \) suspicious packages and \( n \) toilets and a package can be disarmed by dunking it into a toilet. The goal is to have the bomb defused.

  There are several variants of this domain. In the Bomb-in-the-toilet without clogging variant (BT\((m, n)\)), we assume that dunking a package into a toilet does not make the toilet clogged. In the Bomb-in-the-toilet with clogging variant (BTC\((m, n)\)), we assume that dunking a package makes the toilet clogged; furthermore, the domain has an extra action *flush* whose effect is to make a toilet unclogged. In another variant, denoted by BTUC\((m, n)\), we assume that whether or not a toilet is clogged is initially unknown.

- **Ring(n)** [34]: In this domain, one can move in a cyclic fashion (either forward or backward) around a \( n \)-room building to lock windows. Each room has a window which can be locked if it is closed. Initially, neither the location of the robot nor the states (open/closed) of the windows is known. The goal is to have all windows locked. A possible conformant plan is to perform a sequence of actions *forward, close, lock* repeatedly.
- **Logistics (Log(l,c,p))** [23]: In this domain, we need to transport \( p \) packages between locations of \( c \) cities (each city has \( l \) locations) via trucks and airplanes. Initially the exact locations of the packages are unknown. The goal is to transport packages to predetermined locations.

- **Cleaner(n,p)** [124]: This domain is a modified version of the Ring domain in which instead of locking the window, the goal of the robot is to go around \( n \)-room building to clean objects (there are \( p \) objects in each room). Initially, the robot is at the first room and does not know whether or not objects are cleaned.

In this testing, we used the BTUC variant of the Bomb-In-The-Toilet domain and experimented with \( m = 10, 20, 50, 100 \) packages and \( n = 1, 5, 10 \) toilets. In the Logistics domain we did experiments with 5 instances, corresponding to \( l = 2, 3, 4 \) and \( c = p = 2, 3 \), where \( l, c, \) and \( p \) are the numbers of locations per city, cities, and packages respectively, (only Log(4,2,2) is not available). In the Ring domain, we tested with \( n = 2, 5, 10, \) and 20, where \( n \) is the number of rooms. In the Cleaner domain, we tested with 6 instances corresponding to \( n = 2, 5 \) and \( p = 10, 50, 100 \) respectively, where \( n \) is the number of rooms and \( p \) is the number of objects.

All experiments were run on a 2.4 GHz CPU, 768MB RAM machine, running Slackware 10.0 operating system. Time limit is set to half an hour. The testing results are shown in Tables 3.1–3.4. In each table, the left most column shows the planning instance. Columns 2–4 show the characteristics of the instance: the number of initial
partial states (i.e., size of $\Delta$), the total number of fluents, and the number of unknown fluents in each initial partial state. The next columns report the solving time of the corresponding planner. Times are shown in seconds; "-" denotes that the corresponding planner ran out of time without returning a solution or stopped abnormally due to some reason. We ran two versions of the planner, one of which uses the possible world semantics ($\text{CPA}^*$) and the other uses the 0-approximation semantics embedded with the module of computing decisive sets ($\text{CPA}^+$).

Table 3.1: Performance of $\text{CPA}^+$ on the Logistics domain

| Problem   | $|\Delta|$ | $|F|$ | $|U|$ | KACMBP | POND  | CFF | $\text{CPA}^*$ | $\text{CPA}^+$ |
|-----------|-----------|------|------|--------|-------|-----|---------------|---------------|
| Log(2,2,2)| 4         | 20   | 0    | 0.19   | 1.11  | 0.03| 0.05          | 0.06          |
| Log(2,3,3)| 8         | 39   | 0    | 355.96 | 11.89 | 0.06| 2.24          | 2.21          |
| Log(3,2,2)| 9         | 26   | 0    | 2.10   | 4.02  | 0.06| 1.38          | 1.36          |
| Log(3,3,3)| 27        | 51   | 0    | 29.8   | 24.66 | 0.12| 93.90         | 93.17         |
| Log(4,3,3)| 64        | 63   | 0    | -      | 40.12 | 0.14| -             | -             |

As can be seen in Table 3.1, CFF is superior to KACMBP and $\text{CPA}^+$ over the Logistics domain. It took only 0.14 seconds to solve the hardest instance Logistics(4,3,3) while both KACMBP and $\text{CPA}^+$ reported a time out. The second best is POND which can solve all the instances. $\text{CPA}^+$ is better than KACMBP over the first three instance but slower on Logistics(3,3,3). It should be noted here that one of the characteristics of this domain is that all the partial states in $\Delta$ are complete, i.e., they are states indeed; hence, using the 0-approximation to implement $\text{CPA}^+$ does not help solve the problems more quickly than if its possible world semantic version ($\text{CPA}^*$).
Another factor that may cause the slow performance of \( \text{CPA}^+ \) is because of its simple heuristic function.

Table 3.2: Performance of \( \text{CPA}^+ \) on the Ring domain

| Problem  | \( |\Delta| \) | \( |F| \) | \( |U| \) | KACMBP | POND | CFF | \( \text{CPA}^* \) | \( \text{CPA}^+ \) |
|----------|-----------|-------|-------|--------|------|-----|-------|-------|
| Ring(2)  | 2         | 6     | 4     | 0.00   | 0.15 | 0.06 | 0.00  | 0.00  |
| Ring(3)  | 3         | 9     | 6     | 0.00   | 0.08 | 0.23 | 0.09  | 0.00  |
| Ring(4)  | 4         | 12    | 8     | 0.00   | 0.25 | 3.86 | 0.77  | 0.00  |
| Ring(5)  | 5         | 15    | 10    | 0.00   | 0.96 | 63.67| 5.50  | 0.01  |
| Ring(10) | 10        | 30    | 20    | 0.02   | -    | -    | -     | 0.11  |
| Ring(15) | 15        | 45    | 30    | 0.04   | -    | -    | -     | 0.38  |
| Ring(20) | 20        | 60    | 40    | 0.15   | -    | -    | -     | 0.92  |
| Ring(25) | 25        | 75    | 50    | 0.32   | -    | -    | -     | 1.92  |

In the Ring domain (Table 3.2), \( \text{KACMP} \) is the best. This domain, however, is really problematic for \( \text{CFF} \). As explained in [23], this is because of the lack of informativity of the heuristic function in the presence of non-unary effect conditions and the problem with checking repeated states. Both \( \text{POND} \) and \( \text{CFF} \) can solve only the first four instances within the time limit. \( \text{CPA}^+ \) is much better than \( \text{CFF} \) and \( \text{POND} \) but a little slower than \( \text{KACMBP} \). In this domain, and in all the other domains that follow as well, we can see there is a big difference between the performance of \( \text{CPA}^* \) and \( \text{CPA}^+ \). The reason for the good performance of \( \text{CPA}^+ \) over \( \text{CPA}^* \) is because the uncertainty degree of the instances in these domains is high, thus, making the performance of \( \text{CPA}^* \) gets worse quickly because it has to consider all possible initial states which is exponential in the number of unknown fluents.

In the BTUC domain (Table 3.3), \( \text{CPA}^+ \) outperforms all the other planners.
Table 3.3: Performance of C\(\text{P}\)A\(^+\) on the BTUC domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>(\Delta)</th>
<th></th>
<th>(F)</th>
<th></th>
<th>(U)</th>
<th>KACMBP</th>
<th>POND</th>
<th>CFF</th>
<th>C(\text{P})A(^+)</th>
<th>C(\text{P})A(^+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTUC(5,1)</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>0.00</td>
<td>0.03</td>
<td>0.03</td>
<td>0.01</td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(10,1)</td>
<td>1</td>
<td>11</td>
<td>10</td>
<td>0.01</td>
<td>0.07</td>
<td>0.05</td>
<td>1.03</td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(20,1)</td>
<td>1</td>
<td>21</td>
<td>20</td>
<td>0.05</td>
<td>0.57</td>
<td>0.17</td>
<td>-</td>
<td>0.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(50,1)</td>
<td>1</td>
<td>51</td>
<td>50</td>
<td>0.51</td>
<td>28.69</td>
<td>5.33</td>
<td>-</td>
<td>0.31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(100,1)</td>
<td>1</td>
<td>101</td>
<td>100</td>
<td>3.89</td>
<td>682.33</td>
<td>121.8</td>
<td>-</td>
<td>2.28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(5,5)</td>
<td>1</td>
<td>10</td>
<td>5</td>
<td>0.04</td>
<td>0.10</td>
<td>0.04</td>
<td>0.14</td>
<td>0.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(10,5)</td>
<td>1</td>
<td>15</td>
<td>10</td>
<td>0.09</td>
<td>0.65</td>
<td>0.07</td>
<td>6.00</td>
<td>0.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(20,5)</td>
<td>1</td>
<td>25</td>
<td>20</td>
<td>0.30</td>
<td>7.28</td>
<td>0.16</td>
<td>-</td>
<td>0.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(50,5)</td>
<td>1</td>
<td>55</td>
<td>50</td>
<td>1.66</td>
<td>348.28</td>
<td>4.70</td>
<td>-</td>
<td>0.68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(100,5)</td>
<td>1</td>
<td>105</td>
<td>100</td>
<td>6.92</td>
<td>-</td>
<td>113.95</td>
<td>-</td>
<td>4.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(5,10)</td>
<td>1</td>
<td>15</td>
<td>5</td>
<td>0.11</td>
<td>0.35</td>
<td>0.03</td>
<td>0.26</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(10,10)</td>
<td>1</td>
<td>20</td>
<td>10</td>
<td>0.30</td>
<td>2.50</td>
<td>0.05</td>
<td>15.00</td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(20,10)</td>
<td>1</td>
<td>30</td>
<td>20</td>
<td>0.97</td>
<td>27.69</td>
<td>0.13</td>
<td>-</td>
<td>0.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(50,10)</td>
<td>1</td>
<td>50</td>
<td>50</td>
<td>5.39</td>
<td>960.00</td>
<td>4.04</td>
<td>-</td>
<td>1.26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTUC(100,10)</td>
<td>1</td>
<td>110</td>
<td>100</td>
<td>35.83</td>
<td>-</td>
<td>102.56</td>
<td>-</td>
<td>7.44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It took only around 7 seconds to solve the hardest instance, BTUC(100,10), whereas that solving time for KACMP and CFF are more than 35 seconds and 100 seconds respectively; POND reported a time out. The reason for this good result of C\(\text{P}\)A\(^+\) is because this domain exposes a very high degree of uncertainty (i.e., the number of unknown fluents in the initial partial state is almost the same as the total number of fluents). In addition, the planner detects that none of the unknown fluents is needed for its reasoning. As a result, at any time during the search, the planner needs to consider only one partial state and the size of search space is rather small.

Similarly to the BTUC domain, C\(\text{P}\)A\(^+\) also works well with the Cleaner domain (Table 3.4). For the hardest instance, it took around 68 seconds, outputting a plan of length 504. None of the other planners was able to solve the last instance within the
Table 3.4: Performance of CPA\(^+\) on the Cleaner domain

| Problem         | |Δ| | |U| | KACMBP | POND   | CFF | CPA\(^+\) | CPA\(^+\) |
|-----------------|---|---|---|---|---|---|---|---|---|---|
| Cleaner(2,5)    | 1 | 12| 10| 0.01| 0.17| 0.03| 6.87| 0.04|   |
| Cleaner(2,10)   | 1 | 22| 20| 0.08| 0.85| 0.07|   | - | 0.04|   |
| Cleaner(2,20)   | 1 | 42| 40| 0.62| 15.87| 0.15| - | 0.09|   |
| Cleaner(2,50)   | 1 | 102| 100| 13.55| - | 0.80| - | 1.03|   |
| Cleaner(2,100)  | 1 | 202| 200| 185.39| - | 5.72| - | 8.09|   |
| Cleaner(5,5)    | 1 | 30| 25| 0.01| 1.46| 0.11| - | 0.02|   |
| Cleaner(5,10)   | 1 | 55| 50| 0.09| 12.86| 0.24| - | 0.09|   |
| Cleaner(5,20)   | 1 | 105| 100| 7.82| 214.83| 0.85| - | 0.58|   |
| Cleaner(5,50)   | 1 | 255| 250| 227.82| - | 14.36| - | 8.73|   |
| Cleaner(5,100)  | 1 | 505| 500| - | - | - | - | 68.02|   |

Among the others, CFF is the best. It outperforms both KACMBP and POND on this domain. The reason for good performance of CPA\(^+\) on this domain can be explained similarly as with the BTUC domain.

3.5 Discussion and Related Work

One of the main advantages of the 0-approximation is its lower complexity in comparison with the possible world semantics in planning and reasoning when the initial state is described by a single partial state \(\Delta\) \([13]\), i.e., \(|\Delta|\). The price one has to pay for this lower complexity is the incompleteness of reasoning and planning tasks. It is therefore natural to see this advantage (low complexity) to disappear even when we deal with action theories with more than one initial partial state, i.e., \(|\Delta| > 1\).

\(^4\)CFF stopped because the maximum length of a plan is exceeded. We believe that it can be easily fixed by increasing this constant in the source code

\(^5\)We observe that most of the benchmarks in conformant planning satisfy this property
As can be seen, complete reasoning using 0-approximation and decisive sets of fluents offers certain advantages over reasoning based on possible world semantics if partial states are represented explicitly. Given a domain $D$ with $n$ fluents and an initial partial state $\delta$ with a decisive set $F_\delta$, the number of partial states that needs to be considered by the 0-approximation is $2^{|F_\delta|}$ whereas the number of states that needs to be considered by the possible world semantics is $2^{n-|\delta|}$. Because $|F_\delta| \leq n-|\delta|$, reasoning based on the 0-approximation is certainly more efficient than reasoning based on the possible world semantics. It is interesting to note that in most of the benchmarks for conformant planning, we found that $F_\delta = \emptyset$ or $|F_\delta|$ is much smaller than $n-|\delta|$. This explains why CPA$^+$, even built with a simple heuristic, can achieve very good performance as shown in the previous section. Nevertheless, due to explicit representation of partial states, CPA$^+$ has poor performance in domains for which $F_\delta$ is large for some initial partial state $\delta$ as only enumerating all the possible initial partial states might already take exponential time in the size of $F_\delta$. We consider this as a disadvantage of CPA$^+$ and at present, we are investigating alternative representations of partial states to address this issue.

We believe that our work can be applied not only to reasoners/planners that represent (partial) states explicitly but also to ones that represent (partial) states implicitly. We conducted some experiments with CFF, KACMBP, and POND by adding to the bomb domain several irrelevant fluents and specifying them to be unknown in the initial state. We observed that when the number of irrelevant fluents is the same as
the number of fluents of the original problem, these planners were around 2, 10 and 2 times, respectively, i.e., slower. This indicates that the performance of CFF, KACMBP, and POND could be improved if irrelevant fluents of this type were identified.

Regarding the related work, in the past, a number of researchers, e.g., [59, 83, 106], already realized that a planning problem instance may contain some irrelevant information, including irrelevant fluents, irrelevant actions, etc. In most of these work, if irrelevant information is found then it can be safely removed from the problem. In our approach, on the contrary, if a fluent does not belong to a decisive set with respect to a fluent formula, it does not mean that we can safely remove it from the theory without affecting the reasoning process. Rather, it only means that considering the truth values of the fluent separately in the beginning is not necessary. For example, in the bomb domain, we cannot remove \textit{clogged} although it does not belong to the decisive set \{\textit{armed}\} for the partial state $\emptyset$ with respect to $\neg\textit{armed}$. Let us now discuss in more detail the relationship between our work and some existing work.

In [83], the idea of relevant actions of a planning problem instance $P$ with the action domain $D$ is made precise by the notion of an isolated set. Basically, an isolated set $\sigma$ with respect to $D$ could be viewed as a partition of the original domain. If every fluent appearing in the goal is contained in an isolated set $\sigma$, then solutions to $P$ can be found by using $\sigma$ instead of $D$. An isolated set differs from a decisive set of fluents in that (i) it contains not only fluents but also actions and laws; (ii) it does not take into consideration the knowledge about the initial state. We observe that a problem
might not have a non-trivial isolated set (the complete domain is the trivial isolated set) but can still have a decisive set of fluents. For example, for the planning problem instance $P_3 = \langle D_1, \{\emptyset\}, \{\neg \text{armed, clogged}\} \rangle$, no non-trivial isolated set can be found but $\{\text{armed}\}$ is a decisive set for $(D, \{\emptyset\})$. We hypothesize that if $\sigma$ is an isolated set with respect to $D$ containing the goal $G$ then the set of fluents in $\sigma$, which are unknown in an initial state $\delta$, constitutes a decisive set for $(D, \{\delta\}, G)$.

In [59], the notion of a reduced operator set (of a planning problem instance) is defined and algorithms for computing such sets are presented. Intuitively, a reduced operator set consists of operators needed for solving the problem and does not contain any redundant operator (an operator is redundant if it can be replaced by a sequence of operators). They also implemented a preprocessor for computing a reduced operator set of a planning problem instance and demonstrated its usefulness in various planners. We note that this work concentrates on domains with complete initial state while we focus on domains with incomplete information.

In [106], information relevant to a planning problem instance is selected by backchaining from the goals. This is done as follows. First, given a planning problem instance, a fact-generation tree – an AND-OR-tree where the AND-nodes are sets of ground facts and the OR-nodes contains a single ground fact; the root is the set of goals – is constructed. Then, based on the structure of this tree, the set of sets of initial facts possibly needed for the goals, called possibility set for the goals, is determined. Based on this set, they propose several methods of selecting “probably relevant” pieces
of information. This approach differs from ours in that (i) it is intended for planning with complete initial state; and (ii) it may exclude information that is indeed relevant, making the problem unsolvable even if it is solvable.

3.6 Summary

In this chapter, we presented a sufficient condition for the completeness of the 0-approximation semantics and proposed a method for complete reasoning in the presence of incomplete information based on the 0-approximation semantics. The basic idea of the latter is to determine decisive sets for initial partial states. These decisive sets can be used to partition the set of possible initial states into a smaller set of partial states so as to achieve complete reasoning with the 0-approximation. We also presented an algorithm for computing a decisive set of fluents given an action theory and a fluent formula. The efficiency of our method was demonstrated through the development of a sound and complete conformant planner.
CHAPTER 4

APPROXIMATIONS OF DOMAIN DESCRIPTIONS WITH STATIC CAUSAL LAWS

4.1 Introduction

In the previous chapter, we have demonstrated the usefulness of the 0-approximation approach in reasoning and planning. We have also pointed out that the incompleteness of the 0-approximation can be addressed by modifying action theories adequately. Nevertheless, as mentioned in the introduction chapter, in addition to incompleteness, most of existing approximations, including the 0-approximation, cannot handle domains with domain constraints – an important feature of dynamic domains, or only support for a limited class of domain constraints. The main goal of this chapter is to develop approximations for $\mathcal{AL}$ domain descriptions with domain constraints (or, static causal laws in the context of the causality approach).

We start by introducing a general definition of approximations of $\mathcal{AL}$ domain descriptions, outlining some desired properties of approximations. Then, we present two different approaches to the problem of designing approximations for domains with static causal laws. The outcomes are four different approximations which can be computed very efficiently. We also study properties of the proposed approximations, including their completeness condition and complexity in planning. Although an approxima-
tion can be non-deterministic in general, our focus in this chapter is on deterministic approximations only.

Utilizing recent advances in answer set programming and the existing answer set solvers (e.g. cmodels [76], smodels [123], DLV [35], ASSAT [86], or NoMoRe [2]), we develop an approximation based conformant planner, called CPASP, which can solve a large body of benchmark problems. One of the advantages of this planner is the ability to generate concurrent conformant plans. Although CPASP performs well on most of concurrent benchmarks, it does not scale up well to large, sequential benchmarks. The main reason is that existing answer set solvers require the input to be grounded, which often results in a very large grounding. To address this issue, we develop a forward, best-first search sequential conformant planner in C++, called CPA. The search algorithm of CPA is similar to the search algorithm of the CPA+ planner (see Chapter 3).

The main contributions of the work presented in this chapter are:

- a general definition of approximations of $\mathcal{AL}$ domain descriptions,
- two different approaches to tackling the problem of designing approximations for $\mathcal{AL}$ domain descriptions,
- four different approximations of $\mathcal{AL}$ domain descriptions,
- a logic programming based conformant planner, called CPASP, which is capable of generating both sequential plans and concurrent plans,
• an extension of CPASP to deal with disjunctive information about the initial state,

• a best-first search, sequential conformant planner written in C++, called CPA,

• various experimental results demonstrating that conformant planners based on approximations can perform reasonably well against state-of-the-art conformant planners in benchmark problems.

The rest of the chapter is organized as follows. In Section 4.2, we introduce a general definition of the concept of $\mathcal{AL}$ approximations. In Sections 4.3 and 4.4, we present two different approaches to tackling the problem of designing approximations and four different approximations. In Section 4.5 we study properties of these approximations, including the relationship between them, their completeness with respect to the possible world semantics, and their complexity in reasoning and planning. In Section 4.6, we describe the implementation of CPASP in details and prove the correctness of the implementation. In Section 4.7, we describe the planner CPA. In Section 4.8, we compare our planners against state-of-the-art conformant planners. Section 4.9 provides some discussion, including how to handle disjunctive information about the initial state in the logic programming based implementation and advantages/disadvantages of using logic programming in the implementation. Section 4.10 relates our work to existing work. Section 4.11 summarizes main points of the chapter.
4.2 Approximations of $\mathcal{AL}$ Domain Descriptions

Recall that the (full) semantics of a domain description $\mathcal{D}$ can be viewed as the transition diagram $T(\mathcal{D})$ between states where each transition of $T(\mathcal{D})$ represents a possible transition of the domain from one state to another state as a result of the execution of some action. From this point of view, an approximation can be understood as a transition diagram $\tilde{T}(\mathcal{D})$ that “approximates” $T(\mathcal{D})$. The following is our general definition of an approximation of the semantics of $\mathcal{D}$.

**Definition 4.1** Let $\mathcal{D}$ be a domain description and $T(\mathcal{D})$ be the transition diagram of $\mathcal{D}$. An approximation of the semantics of $\mathcal{D}$ (or an approximation of $\mathcal{D}$ for short) is a transition diagram $\tilde{T}(\mathcal{D})$ that satisfies the following conditions.

1. Nodes of $\tilde{T}(\mathcal{D})$ are partial states of $\mathcal{D}$ and arcs of $\tilde{T}(\mathcal{D})$ are labeled with actions.

2. If $\langle \delta, a, \delta' \rangle \in \tilde{T}(\mathcal{D})$ then for every state $s \in \text{ext}(\delta)$,

   (a) $a$ is executable in $s$ and,

   (b) $\delta' \subseteq s'$ for every $s'$ such that $\langle s, a, s' \rangle \in T(\mathcal{D})$.

Intuitively, the first condition describes that an approximation $\tilde{T}(\mathcal{D})$ is a transition diagram between partial states and the second condition requires $\tilde{T}(\mathcal{D})$ to be *sound* with respect to $T(\mathcal{D})$. By this definition, both states and transitions of $T(\mathcal{D})$ are approximated in $\tilde{T}(\mathcal{D})$: states are approximated by partial states and transitions between states.
are approximated by transitions between partial states. With this definition, clearly for a domain description $D$ without static causal laws, the transition diagram $T^0(D)$ (Chapter 2) is an approximation of $D$.

Given an approximation $\tilde{T}(D)$, for partial states $\delta, \delta'$, and sequence of actions $\alpha$, we will write $\langle \delta, \alpha, \delta' \rangle \in \tilde{T}(D)$ to denote the fact that there exists a path corresponding to $\alpha$ from $\delta$ to $\delta'$ in $\tilde{T}(D)$; furthermore, by convention, $\langle \delta, \langle \rangle, \delta \rangle \in \tilde{T}(D)$ for every partial state $\delta$. We say that $\tilde{T}(D)$ is deterministic if for any partial state $\delta$ and action $a$, there exists at most one $\delta'$, called successor partial state of $\delta$, such that $\langle \delta, a, \delta' \rangle \in \tilde{T}(D)$. It is easy to see that when $\tilde{T}(D)$ is deterministic, for any partial state $\delta$ and sequence of actions $\alpha$, there is at most one partial state $\delta'$, called the final partial state of $\delta$ after the execution of $\alpha$, such that $\langle \delta, \alpha, \delta' \rangle \in \tilde{T}(D)$. In this chapter, we assume that approximations are deterministic.

The intended usage of approximations is for reasoning and planning with action theories in the presence of incomplete information, i.e, theories that are described by a pair $(D, \Delta)$ where $D$ is a domain description and $\Delta$ is a non-empty set of valid partial states. In order to do so, first of all we have to define what are consequences of an action theory with respect to an approximation, and what are solutions of a planning problem instance with respect to an approximation.

**Definition 4.2** Let $(D, \Delta)$ be an action theory and $\tilde{T}(D)$ be an approximation of $D$. 

We say that \((\mathcal{D}, \Delta)\) entails a query

\[ \varphi \text{ after } \alpha \]

with respect to \(\tilde{T}(\mathcal{D})\), denoted by,

\[ (\mathcal{D}, \Delta) \models \tilde{T}(\mathcal{D}) \varphi \text{ after } \alpha \]

if for each \(\delta \in \Delta\), (i) there exists a partial state \(\delta'\) such that \(\langle \delta, \alpha, \delta' \rangle \in \tilde{T}(\mathcal{D})\) and (ii) \(\varphi\) is true in \(\delta'\).

The following theorem states the soundness of approximations with respect to the possible world semantics.

**Theorem 4.1** Let \((\mathcal{D}, \Delta)\) be an action theory and \(\tilde{T}(\mathcal{D})\) be an approximation of \(T(\mathcal{D})\).

For any sequence of actions \(\alpha\) and fluent formula \(\varphi\),

\[ (\mathcal{D}, \Delta) \models \tilde{T}(\mathcal{D}) \varphi \text{ after } \alpha \]

implies

\[ (\mathcal{D}, \Delta) \models P \varphi \text{ after } \alpha \]

**Proof.** See Section D.1.1.

Abusing notations, we will use the notations “\(\models \tilde{T}(\mathcal{D})\)” and “\(\models P\)” to refer to reasoning using approximation \(\tilde{T}(\mathcal{D})\) and reasoning using the possible world semantics respectively. Although sound, \(\models \tilde{T}(\mathcal{D})\) may be *incomplete* with respect to \(\models P\) in the sense...
that there may be some queries that can be answered by $\models P$ but not $\models \tilde{T}(D)$. Formally, we define the completeness (or incompleteness) of $\models \tilde{T}(D)$ with respect to $\models P$ as follows (Note that this definition is similar to Definition 3.1).

**Definition 4.3** Let $(D, \Delta)$ be an action theory and $\tilde{T}(D)$ be an approximation of $D$. We say that $\models \tilde{T}(D)$ is complete with respect to $\models P$ on a fluent formula $\varphi$ if for any sequence of actions $\alpha$, 

$$(D, \Delta) \models P \varphi \text{ after } \alpha$$

implies

$$(D, \Delta) \models \tilde{T}(D) \varphi \text{ after } \alpha$$

We now define what it means by a solution of a planning problem instance with respect to an approximation.

**Definition 4.4** Let $P = \langle D, \Delta, G \rangle$ be a planning problem instance and $\tilde{T}(D)$ be an approximation of $D$. A sequence of actions $\alpha$ is a solution of $P$ with respect to $\tilde{T}(D)$ iff $(D, \Delta) \models \tilde{T}(D) G \text{ after } \alpha$.

It follows from Theorem 4.1 and the definition of solutions of planning problem instances (Definition 2.15) that if $\alpha$ is a solution of $P$ with respect to an approximation then it is also a solution of $P$ (with respect to the possible world semantics). Note that the reverse direction does not hold in general, however.

We have formally presented general definitions of an approximation and the entailment relationship between an action theory and a query with respect to an ap-
proximation. We have also pointed out that approximations can be used to compute solutions of planing problem instances. Now the question is how to design and what we would expect from an approximation.

Observe that any deterministic approximation, which is our focus in this chapter, can be described by a transition function \( \tilde{\Phi} \) that maps actions and partial states into partial states and must satisfy the following conditions for any action \( a \) and partial state \( \delta \).

1. If there exists a partial state \( \delta' \) such that \( \langle \delta, a, \delta' \rangle \in \tilde{T}(D) \) then \( \tilde{\Phi}(a, \delta) = \delta' \), and vice versa.

2. If there exists no \( \delta' \) such that \( \langle \delta, a, \delta' \rangle \in \tilde{T}(D) \) then \( \tilde{\Phi}(a, \delta) \) is undefined, denoted by \( \tilde{\Phi}(a, \delta) = \bot \), and vice versa.

3. Furthermore, the transition function must be sound, i.e., if \( \delta' = \tilde{\Phi}(a, \delta) \neq \bot \) then
   
   (a) \( a \) is executable in every state \( s \in ext(\delta) \), and
   
   (b) \( \delta' \subseteq s' \) for any states \( s \) and \( s' \) such that \( s \in ext(\delta) \) and \( s' \in \Phi(a, s) \).

It is easy to see that there are many transition functions that satisfy these conditions but the question is which one(s) we would prefer and why. In general, it does not make much sense to create an approximation that can only answer a very few queries compared to the possible world semantics since such an approximation will not be much useful in practice. Hence, the first desirable property of an approximation is that
it should contain as much information as possible in the sense that given a partial state \( \delta \) and an action \( a \) that is executable in \( \delta \), we want to maximize the successor partial state \( \tilde{\Phi}(a, \delta) \). Ideally, \( \tilde{\Phi}(a, \delta) \) should be the intersection of all successor states of \( ext(\delta) \), i.e., the set of fluent literals

\[
\tilde{\Phi}_{\text{max}}(a, \delta) = \bigcap_{\exists s \in ext(\delta) | s' \in \Phi(a, s)} s'
\]  

(18)

Second, an approximation should be computationally efficient, in the sense that \( \tilde{\Phi}(a, \delta) \) can be efficiently computed, say, in polynomial time in the size of domain. This property certainly gives the approximation an advantage over the possible world semantics because, as we will see in Section 4.5, for the polynomial-length planning problem (Definition 2.16), with some restrictions, it indeed cuts off one level from the complexity. With this property, the condition of maximum information (18), unfortunately, cannot be achieved. Finally, an approximation should be able to handle or can be easily extended to complicated domains, e.g., domains with characteristics like sensing actions, continuous change, multivalued fluents, etc.

Once an approximation has been created, we need to evaluate its usefulness, e.g., what kind of queries it can answer or what kind of planning problem instances it can solve, and how efficiently it operates. These evaluations can be performed both theoretically and practically. On the theoretical side, we may need to study a completeness condition and the complexity of the approximation in reasoning and planning tasks. On the practical side, we can implement a reasoner and/or a planner based on the
approximation and compare its performance against other systems in the same setting.

In the next two sections, we will present two different approaches to approximating $\mathcal{AL}$ domain descriptions which result in four different approximations. We will consider domain descriptions with concurrent actions and static causal laws, i.e., domains described by statements of the form (3)–(5) without any restriction. As an approximation can be described by a transition function $\tilde{\Phi}$ that maps actions and partial states into partial states, we will define our approximations in terms of their transition functions. Both of the two approaches are based on the following observations, suppose a domain description $\mathcal{D}$ is given.

**Observation 4.1** According to Definition 2.3, a possible successor state $s'$ of a state $s$ after the execution of an action $a$ consists of three parts:

1. **the direct effects of $a$:** $\text{de}(a, s)$,

2. **the inertial part:** $s' \cap s$, and

3. **the indirect effects of $a$:** $s' \setminus (\text{de}(a, s) \cup (s \cap s'))$.

**Observation 4.2** For any action $a$ and valid partial state $\delta$ the set of fluent literals

$$E(a, \delta) = \text{Cl}_{\mathcal{D}}(\text{de}(a, \delta))$$

(19)

always holds in any possible successor state of a state $s \in \text{ext}(\delta)$, given that $a$ is executable in $\text{ext}(\delta)$. 

103
4.3 The Possibly-Holds Approach

To define the successor partial state of a partial state $\delta$ after the execution of an action $a$, the possibly-holds approach first determines a set of fluent literals, denoted by $ph(a, \delta)$, that covers all the fluent literals that hold in some possible successor state $s'$ of a state $s \in ext(\delta)$. From Observation 4.1, it follows that a fluent literal $l$ holds in $s'$ if either $l$ is a direct effect of $a$ in $s$, $l$ holds by inertia, or $l$ is an indirect effect of $a$ in $s$. For the first case, it is easy to see that $l$ is a possible direct effect of $a$ in $\delta$ as $de(a, s) \subseteq pe(a, \delta)$. For the second case, we have that $l$ possibly holds in $\delta$, i.e., $\neg l \notin \delta$, as $\delta \subseteq s$. For the third case, it implies that there must be a static causal law

\[ l \text{ if } \psi \]

in $D$ such that $\psi$ holds in $s'$. On the other hand, as $s' \subseteq ph(a, \delta)$ (the property of the set of fluent literals that we wish to find, $ph(a, \delta)$) and $E(a, s) \subseteq s'$, it follows that $\psi \subseteq ph(a, \delta)$ and $\psi$ possibly holds in $E(a, s)$. In addition, by Observation 4.2, such an $l$ does not belong to $\neg E(a, \delta)$.

We therefore compute $ph(a, \delta)$ as follows. A fluent literal $l$ belongs to $ph(a, \delta)$ if $l$ does not belong to $\neg E(a, \delta)$ (the set of fluent literals that is known not to hold in any possible successor state) and one of the following conditions is satisfied.

1. $l$ is a possible direct effect of $a$, i.e., $l \in pe(a, \delta)$,

2. $l$ possibly holds in $\delta$, i.e., $\neg l \notin \delta$.
3. \( l \) is a possible indirect effect of \( a \), i.e., there is a static causal law

\[
l \textbf{if } \psi
\]

such that \( \psi \subseteq ph(a, \delta) \) and \( \psi \) possibly holds in \( E(a, \delta) \), i.e., \( \neg \psi \cap E(a, \delta) = \emptyset \).

Formally, the set of fluent literals \( ph(a, \delta) \) is defined as follows.

\[
ph(a, \delta) = \bigcup_{i=0}^{\infty} ph^i(a, \delta) \tag{20}
\]

where

\[
ph^0(a, \delta) = (pe(a, \delta) \cup \{ l \mid \neg l \notin \delta \}) \setminus \neg E(a, \delta) \tag{21}
\]

and for \( i \geq 0 \)

\[
ph^{i+1}(a, \delta) = ph^i(a, \delta) \cup \left\{ l \mid \begin{array}{l}
\text{there exists a static causal law } [ l \textbf{ if } \psi ] \text{ in } D \text{ s.t. } \\
l \notin \neg E(a, \delta), \psi \subseteq ph^i(a, \delta), \text{ and } \neg \psi \cap E(a, \delta) = \emptyset
\end{array} \right\}
\]

(22)

We have the following property about the set of fluent literals \( ph(a, \delta) \).

**Lemma 4.1** Let \( D \) be a domain description, \( \delta \) be a partial state, and \( a \) be an action that is executable in \( \delta \). Let \( s \) be a state in \( ext(\delta) \). Then, each state \( s' \in \Phi(a, s) \) is a subset of \( ph(a, \delta) \).

**Proof.** See Section D.2.1.

\( \square \)

By this lemma, for each state \( s \in ext(\delta) \), \( ph(a, \delta) \) covers all the fluent literals belonging to the possible successor states of \( s \), i.e.,

\[
\bigcup_{s' \in \Phi(a, s)} s' \subseteq ph(a, \delta)
\]
and this is what we expected. This also implies that the set of fluent literals \( \neg ph(a, \delta) \) covers all the fluent literals that cannot hold in any possible successor state. As a result, the set of fluent literals \( \{ l \mid l \not\in \neg ph(a, \delta) \} \) certainly holds in any possible successor state. This leads us to define a transition function between partial states, namely \( \Phi^{ph} \), as follows.

**Definition 4.5** Let \( \mathcal{D} \) be a domain description. For any action \( a \) and partial state \( \delta \),

1. if \( a \) is not executable in \( \delta \) then

\[
\Phi^{ph}(a, \delta) = \bot
\]

2. otherwise,

\[
\Phi^{ph}(a, \delta) = Cl_{\mathcal{D}}(\{ l \mid l \not\in \neg ph(a, \delta) \})
\]

Note that although we could include \( E(a, \delta) \) in the inside part of the \( Cl_{\mathcal{D}} \) operator in the above definition, it is not necessary because it is not difficult to show that \( E(a, \delta) \subseteq \{ l \mid l \not\in \neg ph(a, \delta) \} \).

Let \( T^{ph}(\mathcal{D}) \) be the transition diagram described by \( \Phi^{ph} \) where \( (\delta, a, \delta') \in T^{ph}(\mathcal{D}) \) iff \( \Phi^{ph}(a, \delta) = \delta' \neq \bot \). It follows from Lemma 4.1 that \( T^{ph}(\mathcal{D}) \) is indeed an approximation of \( \mathcal{D} \).

**Theorem 4.2** Let \( \mathcal{D} \) be a domain description. Then, \( T^{ph}(\mathcal{D}) \) is an approximation of \( \mathcal{D} \).

**Proof.** See Section D.2.2.
4.3.1 An Algorithm for Computing $\Phi_{ph}$

Figure 4.2 presents an algorithm, namely RESPH, that implements the function $\Phi_{ph}$. The algorithm takes as input a domain description $D$, an action $a$ and a partial state $\delta$ and returns as output the successor partial state of $\delta$, $\Phi_{ph}(a, \delta)$. In the algorithm, the variables $de$, $E$, $pe$, and $ph$ are used to store the values of the sets of fluent literals $de(a, \delta)$, $E(a, \delta)$, $pe(a, \delta)$ and $ph(a, \delta)$ respectively. The variable $lit$ is used to store the set of fluent literals $L$ of the domain description.

```
closure(D, \sigma)
INPUT: A domain description $D$ and a set of fluent literal $\sigma$
OUTPUT: $Cl_D(\sigma)$
1. BEGIN
2. $\sigma_1 = \sigma_2 = \sigma$
3. repeat
4. $fixpoint = true$
5. for each static causal law (4) in $D$ do
6. if $\psi$ holds in $\sigma_1$ and $l \notin \sigma_2$ then
7. $\sigma_2 = \sigma_2 \cup \{l\}$ $fixpoint = false$
8. $\sigma_1 = \sigma_2$
9. until $fixpoint$
10. return $\sigma_1$
11. END
```

Figure 4.1: Computing the closure of a set of fluent literals

Initially, the variables $de$ and $pe$ are set to $\emptyset$, and the variable $lit$ is set to $L$ (Line 2). Then, the algorithm iterates over the set of dynamic causal laws in $D$ to add to $de$ and $pe$ fluent literals that are direct effects of $a$ and possible direct effects of $a$ (the for loop, Lines 4–8). Next, the algorithm calls the $closure$ function to set the value of $E$ to $E(a, \delta)$ (Line 9) and then sets the variable $ph$ to $ph^0(a, \delta)$ (Line 10). The $closure$
function, depicted in Figure 4.1, takes as input a domain description $\mathcal{D}$ and a set of fluent literals $\sigma$ and returns as output $\text{Cl}_\mathcal{D}(\sigma)$.

This function is performed in the following steps. First, the closure is set to $\sigma$. Then the function loops over the static causal laws, adding to the closure the head of static causal laws whose preconditions belong to the closure. The loop terminates when no more fluent literals can be added to the closure.

The repeat loop of the RESPH algorithm (Lines 11–16) iterates over the set of static causal laws in $\mathcal{D}$. In each iteration, if there exists a static causal law

\[ l \text{ if } \psi \]
in \( D \) such that \( l \notin \neg E, \psi \subseteq ph, \neg \psi \cap E = \emptyset \), and \( l \) does not belong to the current value of \( ph \) then \( l \) will be added to \( ph \); when no more fluent literal can be added to \( ph \), the loop stops (see (20)–(22)). Finally, the algorithm makes another call to the function \text{CLOSURE} to compute the closure of the set of fluent literals \( \{ l \mid l \notin \neg ph \} \) and returns this value as the successor partial state (Line 11).

The correctness of the algorithm is stated in the following proposition.

**Proposition 4.1** Let \( D \) be a domain description, \( a \) be an action, and \( \delta \) be a partial state such that \( a \) is executable in \( \delta \). Then, the value returned by \( \text{RESPH}(D, a, \delta) \) is \( \Phi_{ph}^\delta(a, \delta) \).

**Proof.** See Section D.2.3.

We have the following proposition about the complexity of the \( \text{RESPH} \) algorithm.

**Proposition 4.2** Let \( D \) be a domain description, \( a \) be an action, and \( \delta \) be a partial state. Then \( \text{RESPH}(D, a, \delta) \) runs in polynomial time in the size of \( D \).

**Proof.** It is easy to see that \( \text{CLOSURE}(D, \sigma) \) runs in polynomial time in the size of the domain because the \text{repeat} loop takes at most \( |L| \) steps and the number of iterations of the nested \text{for} loop is at most as many steps as the number of static causal laws in \( D \). Hence, Lines 9 & 17 in the \( \text{RESPH} \) algorithm takes polynomial time. In addition, the \text{for} loop and the \text{repeat} loop of this algorithm also run in polynomial time. As a result, \( \text{RESPH} \) also runs in polynomial time.
4.3.2 Examples

In this section, we will illustrate the semantics of $T^{ph}$ by several concrete examples. For brevity, for a domain description $D$, we will use $\models_{ph}^D$ to denote $\models_{T^{ph}(D)}$.

**Example 4.1** Consider the bomb-in-the-toilet domain $D_{2.1}$ from Example 2.1. Let

$$\delta = \{\neg clogged(1), \neg clogged(2)\}$$

and

$$a = \{dunk(1, 1), dunk(2, 2)\}$$

We have

$$de(a, \delta) = pe(a, \delta) = \{\neg armed(1), \neg armed(2), clogged(1), clogged(2)\}$$

$$E(a, \delta) = Cl_{D_{2.1}}(de(a, \delta)) = \{\neg armed(1), \neg armed(2), clogged(1), clogged(2), safe\}$$

$$\{l \mid \neg l \notin \delta\} = \{armed(1), \neg armed(1), armed(2), \neg armed(2),$$

$$\neg clogged(1), \neg clogged(2), safe, \neg safe\}$$

Hence,

$$ph^0(a, \delta) = (pe(a, \delta) \cup \{l \mid \neg l \notin \delta\}) \setminus E(a, \delta)$$

$$= \{\neg armed(1), \neg armed(2), clogged(1), clogged(2), safe\}$$
We can check that

\[ ph^0(a, \delta) = ph^1(a, \delta) = \ldots = \{\neg\text{armed}(1), \neg\text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\} \]

Therefore, we have

\[ ph(a, \delta) = \{\neg\text{armed}(1), \neg\text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\} \]

\[ \{l \mid l \not\in \neg ph(a, \delta)\} = \{\neg\text{armed}(1), \neg\text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\} \]

From this, it follows that

\[ \Phi_{ph}^a(a, \delta) = Cl_{D_{2,1}}(\{l \mid l \not\in \neg ph(a, \delta)\}) \]

\[ = Cl_{D_{2,1}}(\{\neg\text{armed}(1), \neg\text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\}) \]

\[ = \{\neg\text{armed}(1), \neg\text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\} \]

It is not difficult to verify that

\[ \Phi_{ph}^a(a, \delta) = \bar{\Phi}_{max}(a, \delta) \]

where \( \bar{\Phi}_{max}(a, \delta) \) is defined by Equation (18), i.e., for this action and this partial state, \( \Phi_{ph} \) obtains maximum information that an approximation can obtain.

As \( \text{safe} \) is true in \( \Phi_{ph}^a(a, \delta) \), it follows that

\[ (D_{2,1}, \{\neg\text{clogged}(1), \neg\text{clogged}(2)\}) \models_{ph} \text{safe} \text{ after } (\{\text{dunk}(1, 1), \text{dunk}(2, 2)\}) \]

(23)
Example 4.2 Consider the domain $\mathcal{D}_{2.5}$ from Example 2.5 and let $\delta = \{\neg g, \neg h\}$. Suppose $e$ is performed in $\delta$. We have

$$de(e, \delta) = pe(e, \delta) = \{f\}$$

$$E(e, \delta) = Cl_{\mathcal{D}_{2.5}}(de(e, \delta)) = \{f\}$$

$$\{l \mid l \notin \delta\} = \{f, \neg f, \neg g, \neg h\}$$

Hence,

$$ph^0(e, \delta) = (pe(e, \delta) \cup \{l \mid l \notin \delta\}) \setminus \neg E(e, \delta)$$

$$= \{f, \neg g, \neg h\}$$

$$ph^1(e, \delta) = ph^0(e, \delta) \cup \{g, h\}$$

$$= \{f, g, \neg g, h, \neg h\}$$

$$ph^2(e, \delta) = ph^1(e, \delta)$$

$$\ldots$$

Hence, we have

$$ph(e, \delta) = \{f, g, \neg g, h, \neg h\}$$

Therefore, we have

$$\{l \mid l \not\in \neg ph(e, \delta)\} = \{f\}$$
Accordingly, we have

\[ \Phi_{ph}(e, \delta) = \text{Cl}_{D_{2.5}}(\{ l \mid l \notin \neg \Phi_{ph}(e, \delta) \}) = \text{Cl}_{D_{2.5}}(\{ f \}) = \{ f \} \]

On the other hand, we have

\[ \text{ext}(\delta) = \{ \neg f, \neg g, \neg h \} \]

and

\[ \Phi(e, \{ \neg f, \neg g, \neg h \}) = \{ s_1, s_2 \} \]

where \( s_1 = \{ f, g, \neg h \} \) and \( s_2 = \{ f, \neg g, h \} \). As a result, we have

\[ \Phi_{ph}(e, \delta) = \{ f \} = s_1 \cap s_2 = \tilde{\Phi}_{\max}(e, \delta) \]

So, in this case, we still obtain maximum information with \( \Phi_{ph} \).

Since \( f \) is true in \( \Phi_{ph}(e, \delta) \), we have

\[ (D_{2.5}, \{ \neg g, \neg h \}) \models_{ph} f \text{ after } \langle e \rangle \] (24)

For the above two examples, we see that \( \Phi_{ph} \) always obtains maximum information that a transition function of an approximation can have. However, this is not always the case as shown in the next examples.
Example 4.3 Consider the domain

\[ \mathcal{D}_{4.3} = \begin{cases} e \text{ causes } f \\ \neg h \text{ if } f, g \\ k \text{ if } f, g \\ \neg p \text{ if } f, \neg k \end{cases} \]

Let \( \delta = \{g, h, p\} \). It is easy to see that any completion \( s \) of \( \delta \) must contain \( \neg f \) for if otherwise then the first static causal law could not be satisfied. Furthermore, the value of the fluent \( k \) can be either true and false in \( s \). Therefore, \( \text{ext}(\delta) = \{s_1, s_2\} \) where

\[ s_1 = \{\neg f, g, h, k, p\} \]

and

\[ s_2 = \{\neg f, g, h, \neg k, p\} \]

We can check that both \( s_1 \) and \( s_2 \) have only one possible successor state

\[ \Phi(e, s_1) = \Phi(e, s_2) = \{\{f, g, \neg h, p, k\}\} \]

(note that in this successor state, \( f \) is a direct effect of \( e \), \( \neg h \) and \( k \) are indirect effects, and \( p \) holds by inertia). Since \( p \) holds in \( \{f, g, \neg h, p, k\} \), we have

\[ (\mathcal{D}_{4.3}, \{\{g, h, p\}\}) \models^P p \text{ after } \langle e \rangle \] (25)

Now let us consider the approximation \( T^{ph}(\mathcal{D}) \). Intuitively, after the execution of \( e \), we would expect the successor partial state \( \Phi_{\text{max}}(e, \delta) \) to be

\[ \Phi_{\text{max}}(e, \delta) = \{f, g, \neg h, k, p\} \]
Let us see if this is the case. We have

\[ de(e, \delta) = pe(e, \delta) = \{f\} \]

\[ E(e, \delta) = Cl_{D_{4.3}}(de(e, \delta)) = \{f\} \]

\[ \{l \mid \neg l \not\in \delta\} = \{f, \neg f, g, h, k, \neg k, p\} \]

Hence,

\[ ph^0(e, \delta) = (pe(e, \delta) \cup \{l \mid \neg l \not\in \delta\}) \setminus \neg E(e, \delta) \]

\[ = \{f, \neg f, g, h, k, \neg k, p\} \setminus \{\neg f\} = \{f, g, h, k, \neg k, p\} \]

We can see that the preconditions of all static causal laws in \( D_{4.3} \) are subsets of \( ph^0(e, \delta) \) and possibly hold in \( E(e, \delta) \). As the heads of these static causal laws do not belong to \( \neg E(a, \delta) \), by definition, they belong to \( ph^1(a, \delta) \)

\[ ph^1(e, \delta) = ph^0(e, \delta) \cup \{\neg h, k, \neg p\} \]

\[ = \{f, g, h, \neg h, k, \neg k, p, \neg p\} \]

We can verify that

\[ ph^1(e, \delta) = ph^2(e, \delta) = \cdots = \{f, g, h, \neg h, k, \neg k, p, \neg p\} \]

Hence,

\[ ph(e, \delta) = \{f, g, h, \neg h, k, \neg k, p, \neg p\} \]

Consequently, we have

\[ \Phi^{ph}(e, \delta) = Cl_{D_{4.3}}(\{l \mid l \not\in \neg ph(e, \delta)\}) = Cl_{D_{4.3}}(\{f, g\}) = \{f, g, \neg h, k\} \]

115
As $p$ is missing in the successor partial state $\Phi_{ph}(e, \delta)$, we have

$$\left( \mathcal{D}_{4.3}, \{\{g, h, p\}\} \right) \not\models_{hp} p \text{ after } \langle e \rangle$$

(26)

4.3.3 An Enhanced Version of $T_{ph}$

Recall that in $T_{ph}(\mathcal{D})$, for an action $a$ and a partial state $\delta$, the successor partial state $\delta'$ of $\delta$ is computed based on the observation that the closure of the set of direct effects of $a$, $E(a, \delta)$, holds in any possible successor state (Observation 4.2). On the other hand, it follows from Theorem 4.2 that $\delta'$ also holds in any possible successor state. Therefore, if we repeat this computation again except that $E(a, \delta)$ is replaced with $\delta'$ then intuitively the resulting partial state $\delta''$ still holds in any possible successor state. It is not difficult to see that the sequence $E(a, \delta), \delta', \delta'', \ldots$, is monotonically non-decreasing and bounded by $\bar{\Phi}_{\max}(a, \delta)$, the maximum value that an approximation can obtain with the successor partial state. As a result, this sequence eventually converges to a fixpoint, say, the partial state $\delta^{\text{limit}}$, which still holds in any possible successor state and furthermore $\Phi_{ph}(a, \delta) \subseteq \delta^{\text{limit}}$. Accordingly, we can define this partial state as the successor partial state of $\delta$ without losing the soundness property of the approximation. We now make this idea more precise.

Let $\delta$ be a partial state and $a$ be an action that is executable in $\delta$. We modify the operator $ph$ (defined by (20)) to include an extra argument $\delta'$ that is assumed to hold in
every possible successor state as follows.

\[
ph(a, \delta, \delta') = \bigcup_{i=0}^{\infty} ph^i(a, \delta, \delta')
\]  

(27)

where

\[
ph^0(a, \delta, \delta') = (pe(a, \delta) \cup \{ l \mid \neg l \notin \delta \}) \setminus \neg \delta'
\]

\[
ph^{i+1}(a, \delta, \delta') = ph^i(a, \delta, \delta') \cup \{ l \mid \text{there exists a static causal law}\ [l\ \text{if}\ \psi]\ \text{in}\ \mathcal{D}\ \text{s.t.}\ l \notin \neg \delta', \ \psi \subseteq ph^i(a, \delta), \ \text{and}\ \neg \psi \cap \delta' = \emptyset\}
\]

Observe that the definition of \(ph^i(a, \delta, \delta')\) is similar to the definition of \(ph^i(a, \delta)\) except that \(E(a, \delta)\) is replaced with \(\delta'\). Hence, we have

\[
ph^i(a, \delta, E(a, \delta)) = ph^i(a, \delta)
\]

and

\[
ph(a, \delta, E(a, \delta)) = ph(a, \delta)
\]

The following lemma is an extension of Lemma 4.1.

**Lemma 4.2** Let \(\mathcal{D}\) be a domain description, \(\delta, \delta'\) be partial states and \(a\) be an action that is executable in \(\delta\). Let \(s\) be a state in \(ext(\delta)\). For each state \(s' \in \Phi(a, s)\), if \(\delta' \subseteq s'\), then \(s'\) is a subset of \(ph(a, \delta, \delta')\).

**Proof.** Similar to the proof of Lemma 4.1 (presented in Section D.2.1) except that \(E(a, \delta)\) is replaced with \(\delta'\) and \(ph^i(a, \delta)\) is replaced with \(ph^i(a, \delta, \delta')\).
Then we construct a sequence of partial states \( \langle \delta_i^{ph} \rangle_{i=0}^\infty \) as follows

\[
\delta_i^{ph} = \begin{cases} 
E(a, \delta) & \text{if } i = 0 \\
\text{Cl}_D(\{l \mid l \not\in \neg ph(a, \delta, \delta_{i-1}^{ph})\}) & \text{if } i \geq 1
\end{cases}
\] (28)

It is worth noting that by this definition, as \( ph(a, \delta, E(a, \delta)) = ph(a, \delta) \), we have

\[
\Phi^{ph}(a, \delta) = \delta_1^{ph}
\]

The following lemma states the desirable property of this sequence of partial states.

**Lemma 4.3** Let \( D \) be a domain description, \( \delta \) be a valid partial state and \( a \) be an action executable in \( \delta \). Then, the sequence of partial states \( \langle \delta_i^{ph} \rangle_{i=0}^\infty \) (defined by (28)) is monotonically non-decreasing and bounded by \( \tilde{\Phi}_{\text{max}}(a, \delta) \).

**Proof.** See Section D.2.4.

By this lemma, the partial state

\[
\bigcup_{i=0}^{\infty} \delta_i^{ph}
\]

belongs to any possible successor state after the execution of \( a \). This leads to the following definition of a transition function between partial states.

**Definition 4.6** Let \( D \) be a domain description. For any action \( a \) and partial state \( \delta \),

1. if \( a \) is not executable in \( \delta \) then

\[
\Phi^{ph^+}(a, \delta) = \bot
\]
2. otherwise,

\[ \Phi^{ph+}(a, \delta) = \bigcup_{i=0}^{\infty} \delta_i^{ph} \]

where \( \delta_i^{ph} \) is defined by (28).

Let \( T^{ph+}(D) \) be the transition diagram described by the transition function \( \Phi^{ph+} \) where \( (\delta, a, \delta') \in T^{ph+}(D) \) if and only if \( \delta' = \Phi^{ph+}(a, \delta) \neq \bot \). We have the following theorem.

**Theorem 4.3** Let \( D \) be a domain description. Then, \( T^{ph+}(D) \) is an approximation of \( D \).

**Proof.** Follows from Lemma 4.3, the definition of \( \Phi^{ph+} \) (Definition 4.6), and the definition of \( T^{ph+}(D) \).

\[
\sqrt{\text{Remark 4.1}} \text{ Let } a \text{ be an action and } \delta \text{ be a valid partial state such that } a \text{ is executable in } \delta. \text{ Since } \Phi^{ph}(a, \delta) = \delta_1^{ph}(a, \delta), \text{ by Lemma 4.3, it follows that } \Phi^{ph}(a, \delta) \text{ is always a subset of } \Phi^{ph+}(a, \delta). \text{ Hence, } T^{ph+}(D) \text{ is stronger than } T^{ph}(D) \text{ in the sense that for any action theory, the set of conclusions that can be derived by } \models^{ph} \text{ is always a subset of the set of conclusions that can be derived by } \models^{T^{ph+}(D)}. \text{ In the next example we will see that in some cases, } \models^{T^{ph+}(D)} \text{ is actually strictly stronger than } \models^{ph}. \text{ For brevity, we will write } \models^{ph+} \text{ to denote } \models^{T^{ph+}(D)}.
\]

\[
\sqrt{119}
\]
Example 4.4 Consider again the domain $D_{4,3}$ and let $\delta = \{g, h, p\}$. From Example 4.3 we have that

$$\delta_1^{ph} = \Phi^{ph}(a, \delta) = \{f, g, \neg h, k\}$$

Now let us compute $\delta_2^{ph}$. We have

$$ph^0(e, \delta, \delta_1^{ph}) = (pe(a, \delta) \cup \{l \mid \neg l \notin \delta\}) \setminus \neg \delta_1^{ph}$$

$$= \{f, \neg f, g, h, k, \neg k, p\} \setminus \{\neg f, \neg g, h, \neg k\} = \{f, g, k, p\}$$

Notice that unlike $ph^0(e, \delta)$ (see Example 4.3), the fluent literal $\neg k$ no longer appears in the set of $ph^0(e, \delta, \delta_1^{ph})$ and thus the precondition of the last static causal law is not a subset of $ph^0(e, \delta, \delta_1^{ph})$. Therefore, the head of this static causal law, $\neg p$, does not belong to the set of fluent literals $ph^1(a, \delta, \delta_1^{ph})$. Consequently, we have

$$ph^1(e, \delta, \delta_1^{ph}) = ph^0(e, \delta, \delta_1^{ph}) \cup \{\neg h, k\}$$

$$= \{f, g, k, p\} \cup \{\neg h, k\} = \{f, g, \neg h, k, p\}$$

We can check that

$$ph^1(e, \delta, \delta_1^{ph}) = ph^2(e, \delta, \delta_1^{ph}) = \cdots = \{f, g, \neg h, k, p\}$$

As a result, we have

$$\delta_2^{ph} = Cl_{4,3}(\{f, g, \neg h, k, p\}) = \{f, g, k, p, \neg h\} = \delta_3^{ph} = \cdots$$

Hence

$$\Phi^{ph+}(e, \delta) = \{f, g, \neg h, k, p\}$$
which implies

\[
(D_{4.3}, \{g, h, p\}) \models^{ph+} p \text{ after } \langle e \rangle
\]  

That is, unlike \(\models^{ph}\), using \(\models^{ph+}\) we can conclude that \(p\) holds after the execution of \(e\).

\[ \square \]

The above example shows that \(\models^{ph+}\) can answer more queries than \(\models^{ph}\). Nevertheless, \(\models^{ph+}\) is still not complete with respect to \(\models^P\) in general, i.e., there are some action theories such that some queries can be answered by \(\models^P\) but not by \(\models^{ph+}\). Studying a condition for the completeness of \(\models^{ph+}\) or \(\models^{ph}\) with respect to \(\models^P\) is very important but also very challenging. We will go back to this issue in Section 4.5.

Similarly to \(\Phi^{ph}\), \(\Phi^{ph+}\) has a nice property that it can be computed in polynomial time in the size of the domain.

**Proposition 4.3** Let \(D\) be a domain description, \(a\) be an action, and \(\delta\) be a partial state such that \(a\) is executable in \(\delta\). Then \(\Phi^{ph+}(a, \delta)\) can be computed in polynomial time in the size of \(D\).

**Proof.** We can easily modify the algorithm \textsc{RESPH} (presented in Figure 4.2) to return \(\Phi^{ph+}(a, \delta)\). The idea is to add another loop for Lines 10–17 and at the end of each iteration of the loop, set the value \(E\) to \(\text{CLOSURE}(D, \text{lit } \neg ph)\). The loop terminates when \(E\) reaches a fixpoint. Then, return this value as the output. It is easy to see that the new algorithm still runs in polynomial time. Hence, \(\Phi^{ph+}(a, \delta)\) can be computed in polynomial time.
4.4 The Possibly-Changes Approach

Given a partial state $\delta$ and an action $a$ that is executable in $\delta$, we say that a fluent literal $l$ possibly changes (after the execution of $a$) if there is a state $s \in \text{ext}(\delta)$ such that $l$ does not belong to $s$ but belongs to some successor state of $s$. Unlike the possibly-holds approach where the successor partial state is defined based on estimating the set of fluent literals that possibly hold after the execution of $a$, the possibly-changes approach, on the contrary, defines the successor partial state based on estimating the set of fluent literals that possibly changes.

It follows from Observations 4.1 and 4.2 that a fluent literal $l$ possibly changes if it does not belong to $\delta$, i.e., $l \not\in \delta$, and one of the following conditions is satisfied.

1. It is a possible direct effect of $a$ in $\delta$, i.e., $l \in pe(a, \delta)$.

2. $D$ contains a static causal law

   \[ l \text{ if } \psi \]

   such that $\psi$ contains at least one fluent literal that possibly changes and $\psi$ possibly holds in $E(a, \delta)$.

So, we estimate the set of fluent literals that possibly changes as follows.

\[ pc(a, \delta) = \bigcup_{i=0}^{\infty} pc^i(a, \delta) \]  \hspace{1cm} (30)
where
\[ pc^0(a, \delta) = pe(a, \delta) \setminus \delta \]
and for \( i \geq 0 \),
\[ pc^{i+1}(a, \delta) = pc^i(a, \delta) \cup \{ l \mid \text{there exists a static causal law } [l \text{ if } \psi]\text{ in } D \text{ s.t. } l \not\in \delta, \psi \cap pc^i(a, \delta) \neq \emptyset, \text{ and } \neg\psi \cap E(a, \delta) = \emptyset \} \]

Clearly, by this definition, \( pc(a, \delta) \) covers all the possible changes that are directly caused by the action as \( de(a, s) \subseteq pe(a, \delta) \) for every \( s \in \text{ext}(\delta) \). The following lemma states that it also covers all the possible changes that are indirectly caused by the action, i.e., by static causal laws.

**Lemma 4.4** Let \( D \) be a domain description, \( \delta \) be a partial state and \( a \) be an action that is executable in \( \delta \). Let \( s \) be a state in \( \text{ext}(\delta) \). Then, for each \( s' \in \Phi(a, s) \), we have
\[ s' \setminus (de(a, s) \cup (s \cap s')) \subseteq pc(a, \delta) \]

**Proof.** See Section D.3.1.

Since \( pc(a, \delta) \) covers all the fluent literals that possibly change, the set of fluent literals \( \delta \setminus \neg pc(a, \delta) \) belongs to the inertial part of any possible successor state \( s' \) of a state \( s \in \text{ext}(\delta) \). Furthermore, by Observation 4.2, we know that the set of fluent literals \( E(a, \delta) \) also holds in any possible successor state. As a result, the set of fluent literals \( E(a, \delta) \cup (\delta \setminus \neg pc(a, \delta)) \), and thus, \( \text{Cl}_D(E(a, \delta) \cup (\delta \setminus \neg pc(a, \delta))) \) as well, definitely hold in any possible successor state. We, therefore, define the successor partial state according to the possibly-changes approach as follows.
**Definition 4.7** Let $D$ be a domain description. For any partial state $\delta$ and action $a$,

1. if $a$ is not executable in $\delta$ then

   \[ \Phi^{pc}(a, \delta) = \bot \]

2. otherwise,

   \[ \Phi^{pc}(a, \delta) = Cl_D(E(a, \delta) \cup (\delta \setminus \neg pc(a, \delta))) \]

Let $T^{pc}(D)$ be the transition diagram described by $\Phi^{pc}$ where $\langle \delta, a, \delta' \rangle \in T^{pc}(D)$ iff $\Phi^{pc}(a, \delta) = \delta' \neq \bot$. Like $T^{ph}(D)$ and $T^{ph+}(D)$, $T^{pc}(D)$ is also an approximation of $D$ as stated in the following theorem.

**Theorem 4.4** Let $D$ be a domain description. Then, $T^{pc}(D)$ is an approximation of $D$.

**Proof.** See Section D.3.2.

\[ \square \]

### 4.4.1 An Algorithm For Computing $\Phi^{pc}$

It is not difficult to devise an algorithm to compute the successor partial state of a partial state $\delta$ according the transition function $\Phi^{pc}$. Figure 4.3 presents such an algorithm, called RESPC, for this purpose. It takes as input a domain description $D$, an action $a$, and a partial state $\delta$ and returns as output the set of fluent literals $\Phi^{pc}(a, \delta)$. In the algorithm, the variables $de$, $E$, and $pc$ are used to store the values of $de(a, \delta)$, $E(a, \delta)$, and $pc(a, \delta)$ respectively.
Initially, the variables \(de\) and \(pc\) are initialized to \(\emptyset\). The for loop (Lines 3–8) iterates over the set of dynamic causal laws to add to \(pc\) and \(de\) the possible changes that are directly caused by the action, i.e., fluent literals belonging to \(pc^0(a, \delta)\), and the direct effects of the action in \(\delta\), i.e., fluent literals belonging to \(de(a, \delta)\), respectively.

---

**RESPC(\(D, a, \delta\))**

**INPUT:** A domain description \(D\), an action \(a\), and a partial state \(\delta\)

**OUTPUT:** \(\Phi_{pc}(a, \delta)\)

1. BEGIN
2. \(de = \emptyset\) \hspace{1cm} \(pc = \emptyset\)
3. for each dynamic causal law (3) in \(D\) do
   4. if \(\psi\) possibly holds in \(\delta\) then
      5. if \(l \not\in \delta\) then
         6. \(pc = pc \cup \{l\}\)
      7. if \(\psi\) holds in \(\delta\) then
         8. \(de = de \cup \{l\}\)
   9. \(E = \text{Closure}(D, de)\)
10. repeat
11. \(stop = true\)
12. for each static causal law (4) in \(D\) do
13. if \(\neg \psi \cap E = \emptyset\) and \(\psi \cap pc \neq \emptyset\) and \(l \not\in \delta\) then
14. \(pc = pc \cup \{l\}\) \hspace{1cm} \(stop = false\)
15. until \(stop\)
16. return \(\text{Closure}(D, E \cup (\delta \setminus pc))\)
17. END

---

Figure 4.3: An algorithm for computing \(\Phi_{pc}(a, \delta)\)

In the next line in the algorithm (Line 9), the algorithm calls the \(\text{Closure}\) function (presented in Figure 4.1) to set the value of the variable \(E\) to \(Cl_D(de) = Cl_D(de(a, \delta))\). The repeat loop (Lines 10–15) iterates over the set of static causal laws in \(D\) to add to \(pc\) possible changes that are caused by static causal laws until no more fluent literal can be added.
It is not difficult to see that the algorithm correctly implements the transition function $\Phi_{pc}$ as stated in the following proposition.

**Proposition 4.4** Let $D$ be a domain description, $a$ be an action, and $\delta$ be a partial state such that $a$ is executable in $\delta$. Then the value returned by $\text{RESPC}(D, a, \delta)$ is $\Phi_{pc}(a, \delta)$.

**Proof.** See Section D.3.3.

Furthermore, since the procedure $\text{CLOSURE}(D, \sigma)$ runs in polynomial time, $\text{RESPC}$ also runs in polynomial time.

**Proposition 4.5** Let $D$ be a domain description, $a$ be an action, and $\delta$ be a partial state. Then $\text{RESPC}(D, a, \delta)$ runs in polynomial time in the size of $D$.

**Proof.** Trivial because $\text{CLOSURE}(D, \sigma)$ runs in polynomial time in the size of the domain.

4.4.2 Examples

In this section, we will illustrate the semantics of $T_{pc}(D)$ by some examples. Similarly to $\models_{ph}$ and $\models_{ph+}$ we will use $\models_{pc}$ as a shorthand for $\models_{T_{pc}(D)}$.

**Example 4.5** Consider the domain $D_{2,1}$ from Example 2.1. Let

$$\delta = \{ \neg \text{clogged}(1), \neg \text{clogged}(2) \}$$
and

\[ a = \{\text{dunk}(1, 1), \text{dunk}(2, 2)\} \]

We have

\[ de(a, \delta) = pe(a, \delta) = \{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2)\} \]

\[ E(a, \delta) = \text{Cl}_{D_2} (de(a, \delta)) = \{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\} \]

Hence,

\[ pc^0(a, \delta) = pe(a, \delta) \setminus \delta = \{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2)\} \]

Since the precondition \( \psi \) of the static causal law

\[ \text{safe if } \neg \text{armed}(1), \neg \text{armed}(2) \]

possibly holds in \( E(a, \delta) \) and \( \psi \cap pc^0(a, \delta) \neq \emptyset \), we have

\[ pc^1(a, \delta) = pc^0(a, \delta) \cup \{\text{safe}\} \]

\[ = \{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\} \]

It is easy to see that for \( i > 1 \),

\[ pc^i = pc^1(a, \delta) \]

As a result, we have

\[ pc(a, \delta) = \bigcup_{i=0}^{\infty} pc^i(a, \delta) \]

\[ = \{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\} \]
Hence,
\[ \delta \setminus \neg \text{pc}(a, \delta) = \emptyset \]

Accordingly, we have
\[
\Phi^{\text{pc}}(a, \delta) = \text{Cl}_{D_{2.1}}(E(a, \delta) \cup (\delta \setminus \neg \text{pc}(a, \delta)))
\]
\[
= \text{Cl}_{D_{2.1}}(\{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\})
\]
\[
= \{\neg \text{armed}(1), \neg \text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\}
\]

As with \( \Phi^{\text{ph}}(a, \delta) \) (see Example 4.1), we can obtain maximum information with \( \Phi^{\text{pc}}(a, \delta) \)
\[
\Phi^{\text{pc}}(a, \delta) = \bar{\Phi}_{\text{max}}(a, \delta)
\]

Since \text{safe} is true in \( \Phi^{\text{pc}}(a, \delta) \), it follows that
\[
(D_{2.1}, \{\neg \text{clogged}(1), \neg \text{clogged}(2)\}) \models^{\text{pc}} \text{safe after } \langle\{\text{dunk}(1, 1), \text{dunk}(2, 2)\}\rangle
\]
(31)

\(\square\)

**Example 4.6** Consider the domain \( D_{2.5} \) from Example 2.5. Let \( \delta = \{\neg g, \neg h\} \). Suppose \( e \) is performed in \( \delta \). We have
\[
de(e, \delta) = \text{pe}(e, \delta) = \{f\}
\]
\[
E(e, \delta) = \text{Cl}_{D_{2.5}}(de(e, \delta)) = \{f\}
\]

Hence,
\[
\text{pc}^0(e, \delta) = \text{pe}(e, \delta) \setminus \delta = \{f\}
\]
\[ pc^1(e, \delta) = pc^2(e, \delta) = \cdots = \{f, g, h\} \]

Therefore,
\[ pc(e, \delta) = \{f, g, h\} \]

As a result, we have
\[ \Phi_{pc}(e, \delta) = C_{D_{2,5}}(E(e, \delta) \cup (\delta \setminus pc(e, \delta))) \]
\[ = C_{D_{2,5}}(\{f\}) = \{f\} \]

As with \( \Phi_{pb} \), with \( \Phi_{pc} \), we still can get maximum information for the successor partial state, i.e.,
\[ \Phi_{pc}(e, \delta) = \tilde{\Phi}_{\text{max}}(e, \delta) \]

Hence,
\[ (D_{2,5}, \{f\}) \models_{pc} \{f\} \text{ after } \langle e \rangle \quad (32) \]

\[ \square \]

**Example 4.7** Consider the domain \( D_{4,3} \) and let \( \delta = \{g, h, p\} \). Then, we have
\[ E(e, \delta) = de(e, \delta) = pe(e, \delta) = \{f\} \]

Hence,
\[ pc^0(e, \delta) = pe(e, \delta) \setminus \delta = \{f\} \]
Since } f \in pc^0(e, \delta), f \text{ belongs to the preconditions of all static causal laws in } D, \text{ and these preconditions possibly hold in } E(e, \delta), \text{ we have}

\[ pc^1(e, \delta) = pc^0(e, \delta) \cup \{ \neg h, k, \neg p \} \]

\[ = \{ f, \neg h, k, \neg p \} \]

We can verify that

\[ pc^1(e, \delta) = pc^2(e, \delta) = \cdots = \{ f, \neg h, k, \neg p \} \]

As a result, we have

\[ pc(e, \delta) = \{ f, \neg h, k, \neg p \} \]

Therefore, we have

\[ \Phi^{pc}(e, \delta) = Cl_{D_{4.3}}(E(e, \delta) \cup (\delta \setminus pc(e, \delta))) \]

\[ = Cl_{D_{4.3}}(\{ f, g \}) = \{ f, g, \neg h, k \} \]

Hence,

\[ (D_{4.3}, \{ \{ g, h, p \} \}) \not\models^{pc} p \text{ after } \langle e \rangle \quad (33) \]

\[ \square \]

4.4.3 An Enhanced Version of } T^{pc}

As with } \Phi^{ph}, \text{ we can improve } \Phi^{pc} \text{ by repeating the computation of the successor partial state until it reaches a fixpoint. To this end, given an action } a \text{ and a partial
state $\delta$ such that $a$ is executable in $\delta$, we first extend the operator $pc(a, \delta)$ to accommodate for another parameter $\delta'$ which is assumed to hold in any possible successor state. Specifically, we define

$$pc(a, \delta, \delta') = \bigcup_{i=0}^{\infty} pc^i(a, \delta, \delta')$$  \hspace{1cm} (34)

where

$$pc^0(a, \delta, \delta') = pc^0(a, \delta) = pc(a, \delta) \setminus \delta$$

and for $i \geq 0$,

$$pc^{i+1}(a, \delta, \delta') = pc^i(a, \delta, \delta') \cup \left\{ l \mid \text{there exists a static causal law} \left[ l \text{ if } \psi \right] \text{ in } D \text{ s.t. } l \not\in \delta, \psi \cap pc^i(a, \delta, \delta') \neq \emptyset, \text{ and } \neg \psi \cap \delta' = \emptyset \right\}$$

Notice that when $\delta' = E(a, \delta)$ then $pc^i(a, \delta)$ is exactly the same as $pc^i(a, \delta, \delta')$.

As a result, we have

$$pc(a, \delta, E(a, \delta)) = pc(a, \delta)$$

**Lemma 4.5** Let $D$ be a domain description, $\delta$, $\delta'$ be partial states and $a$ be an action that is executable in $\delta$. Let $s$ be a state in $\text{ext}(\delta)$. For each state $s' \in \Phi(a, s)$, if $\delta' \subseteq s'$ then

$$s' \setminus (de(a, s) \cup (s \cap s')) \subseteq pc(a, \delta, \delta')$$

**Proof.** Similar to the proof of Lemma 4.4 (presented in Section D.3.1) except that $E(a, \delta)$ is replaced with $\delta'$ and note that $\delta'$ is a subset of $s'$. \hfill $\square$
Given the operator $pc(a, \delta, \delta')$, we construct a sequence of partial states $\langle \delta_{\text{pc}}^i \rangle_{i=0}^\infty$ as follows.

$$
\delta_{\text{pc}}^i = \begin{cases} 
E(a, \delta) & \text{if } i = 0 \\
\text{Cl}_D(\delta_{\text{pc}}^{i-1} \cup (\delta \setminus pc(a, \delta, \delta_{\text{pc}}^{i-1}))) & \text{if } i > 0
\end{cases}
$$

(35)

It is worth noticing that by this definition, we have

$$
\Phi_{\text{pc}}(a, \delta) = \delta_{\text{pc}}^1
$$

Similarly to $\langle \delta_{\text{ph}}^i \rangle_{i=0}^\infty$, the sequence $\langle \delta_{\text{pc}}^i \rangle_{i=0}^\infty$ has the following property.

**Lemma 4.6** Let $D$ be a domain description, $\delta$ be a valid partial state and $a$ be an action such that $a$ is executable in $\delta$. The sequence $\langle \delta_{\text{pc}}^i \rangle_{i=0}^\infty$ (defined by (35)) is monotonically non-decreasing and bounded by $\Phi_{\text{max}}(a, \delta)$.

**Proof.** See Section D.3.4.

$\Box$

It follows from this lemma that the sequence $\langle \delta_{\text{pc}}^i \rangle_{i=0}^\infty$ converges to a partial state and furthermore this partial state is guaranteed to hold in any possible successor state. This lead us to define another transition function between partial states as follows.

**Definition 4.8** Let $D$ be a domain description. For any partial state $\delta$ and action $a$,

1. if $a$ is not executable in $\delta$ then

$$
\Phi_{\text{pc}^+}(a, \delta) = \perp
$$
2. otherwise,

\[ \Phi^{pc^+}(a, \delta) = \bigcup_{i=0}^{\infty} \delta_i^{pc} \]

where \(\delta_i^{pc}\) is defined by (35).

Let \(T^{pc^+}(D)\) be the transition diagram described by \(\Phi^{pc^+}\) where \(\langle \delta, a, \delta' \rangle \in T^{pc^+}(D)\) iff \(\Phi^{pc^+}(a, \delta) = \delta' \neq \bot\). The following theorem states that \(T^{pc^+}(D)\) is indeed an approximation of \(T(D)\).

**Theorem 4.5** Let \(D\) be a domain description. Then, \(T^{pc^+}(D)\) is an approximation of \(T(D)\).

**Proof.** Follows from Lemma 4.6, the definition of \(\Phi^{pc^+}\) (Definition 4.8), and the definition of \(T^{pc^+}(D)\).

\(\Box\)

**Remark 4.2** For any action \(a\) and partial state \(\delta\) such that \(a\) is executable in \(\delta\), we have noted previously that

\[ \Phi^{pc}(a, \delta) = \delta_1^{pc}(a, \delta) \]

By Lemma 4.6, this implies that \(\Phi^{pc}(a, \delta)\) is always a subset of \(\Phi^{pc^+}(a, \delta)\). Hence, \(T^{pc^+}(D)\) is stronger than \(T^{pc}(D)\). In the next example, we will see that in some cases, \(\models^{pc^+}\) (which is a shorthand for \(\models^{T^{pc}(D)}\)) can actually derive more conclusions than \(\models^{pc}\).

\(\Box\)
Example 4.8 Consider the domain $D_{4.3}$ and let $\delta = \{g, h, p\}$. It follows from Example 4.3 that

$$\delta_1^{pc} = \Phi^{pc}(e, \delta) = \{f, g, \neg h, k\}$$

Now let us compute $\delta_2^{pc}$. We have

$$pc^0(e, \delta, \delta_1^{pc}) = pe(e, \delta) \setminus \delta_1^{pc} = \{f\}$$

$$pc^1(e, \delta, \delta_1^{pc}) = pc^2(e, \delta, \delta_1^{pc}) = \cdots = \{f, \neg h, k\}$$

Note that $\neg p$ does not belong to $pc^1(e, \delta, \delta_1^{pc})$ as the precondition of the static causal law

$$\neg p \text{ if } f, \neg k$$

does not possibly hold in $\delta_1^{pc}$. Hence, we have

$$pc(e, \delta, \delta_1^{pc}) = \{f, \neg h, k\}$$

Accordingly,

$$\delta_2^{pc} = Cl_{D_{4.3}}(\delta_1^{pc} \cup (\delta \setminus pc(e, \delta, \delta_1^{pc})))$$

$$= Cl_{D_{4.3}}(\{f, g, \neg h, k\} \cup (\{g, h, p\} \setminus \{f, h, \neg k\})) = Cl_{D_{4.3}}(\{f, g, \neg h, k, p\})$$

$$= \{f, g, \neg h, k, p\}$$

Since $\delta_2^{pc}$ is already maximum, we do not need to repeat this computation again. As a result, we have

$$\Phi^{pc^+}(e, \delta) = \{f, g, \neg h, k, p\}$$
That is

\[(\mathcal{D}_{4.3}, \{\{g, h, p\}\}) \models^{pc+} p \text{ after } \langle e \rangle\]  \hspace{1cm} (36)

In other words, in contrast to $|\models^{pc}$, $|\models^{pc+}$ allows us to conclude that $p$ holds after the execution of $e$.

\[\Box\]

**Proposition 4.6** Let $\mathcal{D}$ be a domain description, $a$ be an action, and $\delta$ be a partial state such that $a$ is executable in $\delta$. Then $\Phi^{pc+}(a, \delta)$ can be computed in polynomial time in the size of $\mathcal{D}$.

**Proof.** Without much effort, we can modify the algorithm RESPC presented in Figure 4.3 to compute $\Phi^{pc+}(a, \delta)$. Furthermore, the new algorithm are still guaranteed to run in polynomial time. As a result, $\Phi^{pc+}(a, \delta)$ can be computed in polynomial time.

\[\Box\]

### 4.5 Properties Of the Approximations

This section is devoted to discussing about some important properties of the proposed approximations. We begin with the relationship between them, followed by a sufficient condition for their completeness with respect to $|\models^P$. Then, we present some complexity results about reasoning and planning with the approximations.

Let $\tilde{T}(\mathcal{D})$ be a deterministic approximation and let $\tilde{\Phi}$ be the transition of $\tilde{T}(\mathcal{D})$. We extend $\tilde{\Phi}$ to define the final partial state after the execution of a sequence of actions as follows (the new function is denoted by $\hat{\Phi}$).
Definition 4.9 Let $D$ be a domain description. For any sequence of actions $\alpha$ and valid partial state $\delta$,

1. if $\alpha = \langle \rangle$ then

   \[
   \widehat{\Phi}(\alpha, \delta) = \delta
   \]

2. if $\alpha = \langle \beta, a \rangle$ where $\beta$ is a sequence of actions and $a$ is an action then

   \[
   \widehat{\Phi}(\alpha, \delta) = \begin{cases} 
   \bot & \text{if } \widehat{\Phi}(\beta, \delta) = \bot \\
   \Phi(a, \widehat{\Phi}(\beta, \delta)) & \text{otherwise}
   \end{cases}
   \]

It is easy to see that with this definition a transition $\langle \delta, \alpha, \delta' \rangle$, where $\alpha$ is a sequence of actions, belongs to $\widehat{T}(D)$ if and only if $\delta' = \widehat{\Phi}(\alpha, \delta) \neq \bot$. Hence, an action theory $(D, \Delta)$ entails a query

$\varphi$ after $\alpha$

with respect to $\widehat{T}(D)$ if and only if for every $\delta \in \Delta$, $\widehat{\Phi}(\alpha, \delta) \neq \bot$ and $\varphi$ is true in $\widehat{\Phi}(\alpha, \delta)$.

In what follows, we will use the superscript $^A$ after $\Phi$, $\widehat{\Phi}$, or $\models$, to refer to the corresponding component of either $T_{ph}(D)$, $T_{ph}^+(D)$, $T_{pc}(D)$, or $T_{pc}^+(D)$.

4.5.1 The Relationship between the Approximations

From Remark 4.1, we know that the set of conclusions that $\models_{ph}$ can draw is always a subset of the set of conclusions that $\models_{ph}^+$ can draw. Furthermore, Examples 4.3 and 4.4 show that $\models_{ph}^+$ is strictly stronger than $\models_{ph}$ in some cases. Likewise, from
Remark 4.2 and Examples 4.7 and 4.8, we know that $\models^{pc^+}$ is stronger than $\models^{pc}$ in general and in some specific cases it is indeed strictly stronger $\models^{pc}$. Nevertheless, in all examples presented so far, both $\models^{ph}$ and $\models^{pc}$ (or $\models^{ph^+}$ and $\models^{pc^+}$) return the same answer for a query. This is not the case in general, however, and we cannot say which one is stronger than the other.

Example 4.9 Consider the domain

$$D_{4.9} = \{ \begin{array}{l} e \text{ causes } f \\ g \text{ if } g, f \end{array} \}$$

and let $\delta_1 = \{ \neg g \}$. We can easily check that $\Phi^{pc}_{D_{4.9}}(e, \delta_1) = \Phi^{pc^+}_{D_{4.9}}(e, \delta_1) = \{ f \}$ and $\Phi^{ph}_{D_{4.9}}(e, \delta_1) = \Phi^{ph^+}_{D_{4.9}}(e, \delta_1) = \{ f, \neg g \}$. Hence, we have

$$(D_{4.9}, \{ \delta_1 \}) \models^{ph} \neg g \text{ after } \langle e \rangle$$

$$(D_{4.9}, \{ \delta_1 \}) \models^{ph^+} \neg g \text{ after } \langle e \rangle$$

but

$$(D_{4.9}, \{ \delta_1 \}) \not\models^{pc} \neg g \text{ after } \langle e \rangle$$

$$(D_{4.9}, \{ \delta_1 \}) \not\models^{pc^+} \neg g \text{ after } \langle e \rangle$$

Remark 4.3 Observe that the static causal law of $D_{4.9}$ is irrelevant to the domain description in the sense that it can be safely removed without affecting the full semantics of the domain description. Of course, we could have excluded this type of static causal laws from the definition of $pc^i$’s so that $\models^{pc}$ can answer the above query.
Example 4.10 Consider the domain

\[
D_{4.10} = \begin{cases} 
e \text{ causes } h \\ \neg f \text{ if } g \end{cases}
\]

and let \( \delta = \{f\} \). Then, we have

\[
E(e, \delta) = de(a, \delta) = pe(e, \delta) = \{h\}
\]

Observe that the value of fluent \( g \) is false in any completion \( s \) of \( \delta \), for if otherwise, by the static causal law, we would have \( \neg f \in s \) which results in inconsistency (both \( f \) and \( \neg f \) belong to \( s \)). Therefore, we have \( ext(\delta) = \{s_1, s_2\} \) where \( s_1 = \{f, \neg g, h\} \) and \( s_2 = \{f, \neg g, \neg h\} \). Executing \( e \) in either \( s_1 \) and \( s_2 \) will cause the fluent \( h \) to be true in any possible successor state. Thus, we have

\[
\Phi(e, s_1) = \Phi(e, s_2) = \{\{f, \neg g, h\}\}
\]

As a result, we have

\[
(D_{4.10}, \{\{f\}\}) \models^P f \text{ after } \langle e \rangle \tag{37}
\]

Let us see if we can obtain a similar result with \( \models^P \). First, let us compute \( \Phi^{ph}(e, \delta) \).

\[
\begin{align*}
l \not\in \delta & \Rightarrow \{f, g, \neg g, h, \neg h\} \\
\Phi^{ph}(e, \delta) & = (pe(e, \delta) \cup \{l \mid \not\in \delta\}) \setminus E(e, \delta) \\
& = \{f, g, \neg g, h\}
\end{align*}
\]
As the precondition of the static causal law, $g$, belongs to $\text{ph}^0(e, \delta)$ and possibly holds in $E(e, \delta)$, the head of the static causal law, $\neg f$, belongs to $\text{ph}^1(e, \delta)$

$$\text{ph}^1(e, \delta) = \{f, \neg f, g, \neg g, h\}$$

We can also check that

$$\text{ph}^1(e, \delta) = \text{ph}^2(e, \delta) = \cdots = \{f, \neg f, g, \neg g, h\}$$

Hence, we have

$$\text{ph}(e, \delta) = \bigcup_{i=0}^{\infty} \text{ph}^i(e, \delta) = \{f, \neg f, g, \neg g, h\}$$

Consequently, we have

$$\Phi^{\text{ph}}(e, \delta) = \text{Cl}_{D_{4.10}}(\{l \mid l \notin \neg \text{ph}(e, \delta)\}) = \text{Cl}_{D_{4.10}}(\{h\}) = \{h\}$$

Hence,

$$(D_{4.10}, \{\{f\}\}) \not\vdash^{\text{ph}} f \text{ after } \langle e \rangle$$

(38)

Repeating this process does not add any new information to the successor partial state. As a result, using $\not\vdash^{\text{ph}+}$ does not allow us to answer the above query. That is,

$$(D_{4.10}, \{\{f\}\}) \not\vdash^{\text{ph}+} f \text{ after } \langle e \rangle$$

(39)

Nevertheless, for this example, both $T_{\text{pc}}(D)$ and $T_{\text{pc}+}(D)$ can derive that $f$ is true after the execution of $\langle e \rangle$. The following computation shows this.

$$\text{pc}^0(e, \delta) = \text{pc}(a, \delta) \setminus \delta = \{h\}$$

$$\text{pc}^1(e, \delta) = \text{pc}^2(e, \delta) = \cdots = \{h\}$$
As a result, we have

$$pc(e, \delta) = \{h\}$$

Hence,

$$\Phi_{pc}(e, \delta) = \delta_{pc}^1(e, \delta) = Cl_{D_{4.10}}(\{h\} \cup \{f\}) = \{f, h\}$$

Since $\Phi_{pc}(e, \delta)$ is already maximum, we have

$$\Phi_{pc}^+(e, \delta) = \{f, h\}$$

Thus, we have

$$(D_{4.10}, \{\{f\}\}) \models^{pc} f \text{ after } \langle e \rangle \quad (40)$$

$$(D_{4.10}, \{\{f\}\}) \models^{pc} f \text{ after } \langle e \rangle \quad (41)$$

\[\square\]

**Remark 4.4** It is worth noting that the main reason that

$$\Phi_{ph}(e, \delta) \neq \tilde{\Phi}_{\text{max}}(e, \delta)$$

in Examples 4.3 and 4.10 is because $\delta$ is not the intersection of $s_1$ and $s_2$, i.e.,

$$\delta \neq \bigcap_{s \in ext(\delta)} s$$

If we were defining the set of fluent literals that initially possibly holds, $ph^0(e, \delta)$, to be

$$(pe(e, \delta) \cup \bigcap_{s \in ext(\delta)} s) \setminus \neg E(a, \delta)$$
then we would drive, using $\models^{ph}$, that $p$ in Example 4.3 or $f$ in Example 4.10 would hold after the execution of $\langle e \rangle$. Nevertheless, only enumerating all completions of $\delta$ may already take exponential time, which violates the efficiency desiratum of the approximation.

Although not identical in general, $\models^{ph}$, $\models^{ph+}$, $\models^{pc}$, and $\models^{pc+}$ agree with each other and with the $\models^0$ on action theories without static causal laws as stated in the following theorem.

**Theorem 4.6** Let $(D, \Delta)$ be an action theory where $D$ contains no static causal laws. Then, for any sequence of actions $\alpha$ and fluent formula $\varphi$, we have

$$(D, \Delta) \models^A \varphi \text{ after } \alpha \Leftrightarrow (D, \Delta) \models^0 \varphi \text{ after } \alpha$$

**Proof.** See Section D.4.1.

**4.5.2 A Sufficient Condition For the Completeness of $\models^A$**

Since both $\models^{ph+}$ and $\models^{pc+}$ are sound with respect to $\models^P$ and they are not identical with each other it follows that both of them, and thus $\models^{ph}$ and $\models^{pc}$ as well, are not complete with respect to $\models^P$ in general. Characterizing action theories for which $\models^A$ is complete is not a simple task. It follows from Theorem 4.6, that for domain descriptions without static causal laws, the sufficient condition presented in Chapter 3...
can be applied to $|=^A$. In this section, we present our initial study of the completeness of $|=^A$ on answering a fluent formula $\varphi$ which extends this result to domains with static causal laws. Without loss of generality, we assume that $\varphi$ is in CNF.

First of all, let us extend the definition of dependencies between fluent literals in Definition 3.2 to domain descriptions with static causal laws.

**Definition 4.10** Let $D$ be a domain description. A fluent literal $l$ depends on a fluent literal $g$, written as $l \triangleleft g$, if and only if one of the following conditions holds.

1. $l = g$.

2. $D$ contains a dynamic causal law $e$ causes $l$ if $\psi$ such that $g \in \psi$.

3. $D$ contains a static causal law $l$ if $\psi$ such that $g \in \psi$.

4. There exists a fluent literal $h$ such that $l \triangleleft h$ and $h \triangleleft g$.

5. The complement of $l$ depends on the complement of $g$, i.e., $\neg l \triangleleft \neg g$.

Notice that the only difference between the definition of $\triangleleft$ and the definition of $\triangleleft$ (Definition 3.2) is the addition of the third condition where the roles of static causal laws
are taken into account in the definition of a dependency. The dependency between an action \( a \) and fluent literal \( l \), denoted by \( a \leftarrow l \), and the reducibility of a belief state \( S \) to a partial state \( \delta \) with respect to a fluent formula \( \varphi \), denoted by \( S \gg_{\varphi} \delta \), are defined similarly as in Definitions 3.3 and 3.4 except that \( \triangleright \) is replaced with \( \leftarrow \). We now define a class of domain descriptions, called \emph{simple domain descriptions}, for which the reducibility of a belief state is preserved during the course of the execution of actions.

**Definition 4.11** A static causal law

\[
l \text{ if } \psi
\]

is simple if \( \psi \) contains at most one literal. A domain description \( \mathcal{D} \) is simple if each of its static causal laws is simple.

Note with this definition, domain descriptions without static causal laws are always simple. Furthermore, we observe that many of real world static causal laws are simple in nature. For example, to represent the unique location of a robot at a time, we can use the following collection of static causal laws:

\[
\{ \neg \text{at}(X) \text{ if } \text{at}(Y) \mid X \neq Y \}
\]

Likewise, to represent the unique value of a multi-valued object \( \text{obj} \), we can use:

\[
\{ \neg \text{value}(\text{obj}, X) \text{ if } \text{value}(\text{obj}, Y) \mid X \neq Y \}
\]

The following propositions are extensions of Propositions 3.1 and 3.2.
Proposition 4.7 Let $D$ be a simple domain description. Let $S$ be a belief state, $\delta$ be a valid partial state, and $\sigma$ be a fluent formula such that $S \gg_{\phi} \delta$. Then, $\phi$ is true in $S$ iff $\phi$ is true in $\delta$.

Proof. Similar to the proof of Proposition 3.1 (Section C.1.1) except that $\lhd$ is replaced with $\triangleleft$.

Proposition 4.8 Let $D$ be a simple domain description. Let $S$ be a belief state, $\delta$ be a partial state, and $\sigma$ be a fluent formula such that $S \gg_{\phi} \delta$. For any action $a$, if $a$ is executable in $S$ then

1. $a$ is executable in $\delta$,

2. $\Phi^P(a, S) \gg_{\phi} \Phi^A(a, \delta)$.

Proof. See Section D.4.2.

Similarly to Proposition 3.2, the latter proposition can be extended to $\hat{\Phi}^A$ as follows.

Proposition 4.9 Let $D$ be a simple domain description. Let $S$ be a belief state, $\delta$ be a partial state, and $\phi$ be a fluent formula such that $S \gg_{\phi} \delta$. For any sequence of actions $\alpha$, if $\hat{\Phi}^P(\alpha, S) \neq \bot$ then

1. $\hat{\Phi}^A(\alpha, \delta) \neq \bot$, and
2. \( \Phi^P(\alpha, S) \gg_{\varphi} \Phi^A(\alpha, \delta) \).

**Proof.** Similar to the proof of Proposition 3.3 except that we have to replace “\( \Phi^0 \)” with “\( \Phi^A \)”, and “Proposition 3.3” with “Proposition 4.9”.

From Propositions 4.7 and 4.9, we have the following theorem.

**Theorem 4.7** Let \((D, \Delta)\) be an action theory, where \(D\) is a simple domain description, and let \( \varphi \) be a fluent formula. If \( \text{ext}(\delta) \gg_{\varphi} \delta \) for every \( \delta \in \Delta \) then \( \models^A \) is complete on \( \varphi \).

**Proof.** Similar to the proof of Theorem 3.1 but makes use of the result in Proposition 4.9 instead of the result in Proposition 3.3.

**4.5.3 Complexity of Planning with respect to \( \models^A \)**

Interestingly, the use of \( \models^A \) cuts off at least one level from the complexity compared to the possible world semantics for the polynomial-length planning problem (Definition 2.16).

**Theorem 4.8** The polynomial-length planning problem with respect to \( \models^A \) is NP-complete.

**Proof.** See Section D.4.3.

So, for the polynomial-length planning problem, the use of \( \models^A \) is actually computationally efficient.
4.6 An Approximation Based Conformant Planner in Logic Programming

To evaluate the usefulness of the proposed approximations, we develop a pair of approximation based conformant planners then compare their performance with other state-of-the-art conformant planners. One of them, called CPASP, is implemented in logic programming paradigm and the other, called CPA, is implemented in C++. One of the main differences between these two conformant planners lies in the approximations that they use. While CPASP is based on the $T^{ph}(D)$ approximation, CPA provides the user with an option to select either $T^{ph}(D)$ or $T^{pc+}(D)$ that will be used in searching for a solution. Furthermore, CPASP can generate concurrent conformant solutions but CPA only can generate sequential conformant solutions. The version of CPASP that will be presented next is designated for solving planning problem instances with only one initial partial state, i.e., instances of the form $\langle D, \delta, G \rangle$. In the next section, we will discuss how to extend it to handle disjunctive information about the initial state. The CPA conformant planner will be described later, in Section 4.7, and the experimental results will be presented in Section 4.8.

In the CPASP framework, a planning problem instance $\mathcal{P} = \langle D, \delta, G \rangle$ is translated into a logic program $\pi_h(\mathcal{P})$, where $h$ is an input parameter to denote the length of the solution we wish to find. The answer sets of the program $\pi_h(\mathcal{P})$ represent solutions of $\mathcal{P}$. In what follows, we describe $\pi_h(\mathcal{P})$ in the syntax of the answer set solver smodels [123] (note that $\mathcal{P} = \langle D, \delta, G \rangle$) but it should be noted that with some minor
changes, $\pi_h(D)$ can be run on any other existing answer set solver as well. A sample of a logic program $\pi_h(P)$ is listed in Appendix E.

1. **Constants.** In addition to the constant $h$, $\pi_h(P)$ has other constants to denote fluents, literals and actions in the domain. Due to the fact that smodels does not allow symbol $\neg$, to represent a literal constant $\neg f$, we will use $\text{neg}(f)$.

2. **Predicates.** The program uses the following predicates.

   - $\text{time}(T)$ is true if $1 \leq T \leq h$.
   - $\text{time1}(T_1)$ is true if $1 \leq T_1 \leq h + 1$.
   - $\text{fluent}(F)$ is true if $F$ is a fluent.
   - $\text{literal}(L)$ is true if $L$ is a fluent literal.
   - $\text{contrary}(L, L_1)$ is true if $L$ and $L_1$ are two complementary fluent literals.
   - $\text{action}(E)$ is true if $E$ is an elementary action
   - $\text{holds}(L, T)$ is true if fluent literal $L$ holds at $T$.
   - $\text{occ}(E, T)$ is true if an elementary action $E$ occurs at $T$.
   - $\text{ef}(L, T)$ is true if fluent literal $L$ is an effect of an action occurring at $T$.
   - $\text{ph}(L, T)$ is true if fluent literal $L$ may change at $(T + 1, P)$.
   - $\text{goal}(T)$ is true if the goal is satisfied at $T$.

3. **Variables.** The following variables are used in the program.
• $F$: a fluent variable.

• $L$ and $L_1$: fluent literal variables.

• $T$ and $T_1$: time variables, in ranges 1..$h$ and 1..$h + 1$ respectively,

• $E$: an elementary action variable,

• $A$: an action variable.

The domains of these variables are declared in $\pi_h(\mathcal{P})$ using the keyword #domain of smodels.

4. Rules. The program $\pi_h(\mathcal{P})$ has the following facts to define variables of sort time:

$$
time(1..h) \leftarrow
$$

$$
time1(1..h + 1) \leftarrow
$$

Furthermore, for each elementary action $e$, or fluent $f$, $\pi_h(\mathcal{P})$ contains the following facts respectively

$$
action(e) \leftarrow
$$

$$
fluent(f) \leftarrow
$$

The remaining rules of $\pi_h(\mathcal{P})$ are divided into three groups: (i) domain dependent rules; (ii) goal representation and (iii) domain independent rules and listed below. Note that they are shown in a shortened form in which the following shortening conventions are used.
• Two complementary literal variables are written as $L$ and $\neg L$.

• For a predicate symbol $p$, and a set $\gamma$ of literals or actions, we will write $p(\gamma, \ldots)$ to denote the set of atoms $\{p(x, \ldots) \mid x \in \gamma\}$.

• For a literal constant $l$, $\neg l$ stands for $\text{neg}(f)$ (resp. $f$) if $l = f$ (resp. $l = \neg f$) for some fluent $f$.

For example, the rule (53) is the shorten form of the following rule

$$\text{holds}(L, T + 1) \leftarrow \text{contrary}(L, L_1), \text{not ph}(L_1, T)$$

**Domain dependent rules**

• *Rules encoding the initial state of the domain.* For each $l \in \delta$ (recall that $\delta$ is the only initial partial state of $\mathcal{P}$), $\pi_h(\mathcal{P})$ contains the fact

$$\text{holds}(l, 1) \leftarrow$$

$$(42)$$

• *Rules encoding impossibility conditions.* For each impossibility condition $\text{impossible} b$ if $\psi$

in $\mathcal{D}$, $\pi_h(\mathcal{P})$ contains the rule

$$\leftarrow \text{occ}(b, T), \text{not holds}(\neg \psi, T)$$

$$(43)$$

This rule states that if the precondition $\psi$ of the impossibility condition possibly holds at time step $T$ (note that by our convention $\text{not holds}(\neg \psi, T)$ is the short-
hand for \( \{ \text{not} \ holds(-l, T) \mid l \in \psi \} \) which means that \( \psi \) possibly holds at time step \( T \) then any action \( a \supseteq b \) cannot occur at that time step.

- **Rules for reasoning about the effects of actions.** For each dynamic causal law

  \[ e \text{ causes } l \text{ if } \psi \]

  in \( D \), we add to \( \pi_h(P) \) the following rules:

  \[
  \begin{align*}
  ef(l, T) & \leftarrow \text{occ}(e, T), holds(\psi, T) \tag{44} \\
  ph(l, T) & \leftarrow \text{occ}(e, T), \text{not} \ ef(-l, T), \text{not} \ holds(-\psi, T) \tag{45}
  \end{align*}
  \]

  The first rule, when used along with \( (46) \), encodes what certainly holds in the successor state as the effects of the action that occurs at time step \( T \), i.e., the set of fluent literals \( E(a, \delta) \), where \( a = \{ e \mid \text{occ}(e, T) \text{ is true} \} \). The second rule, together with \( (47) \) and \( (52) \), describes what possibly holds in any possible successor state, i.e., the set of fluent literals \( ph(a, \delta) \). All of these rules will be used in corporation with \( (48) \) and \( (53) \) to define the successor partial state according to the \( T^{ph}(D) \) approximation.

- **Rules for reasoning about static causal laws.** For each static causal law

  \[ l \text{ if } \psi \]
in $\mathcal{D}$, $\pi_h(\mathcal{P})$ contains the rules

$$ef(l, T) \leftarrow ef(\psi, T)$$  \hspace{1cm} (46)

$$ph(l, T) \leftarrow \neg e(l, T), ph(\psi, T), \neg ef(\neg \psi, T)$$ \hspace{1cm} (47)

$$holds(l, T_1) \leftarrow holds(\psi, T_1)$$ \hspace{1cm} (48)

The first and the third rules state that the sets of fluent literals $E(a, \delta)$, and the successor partial state are closed under the set of static causal laws. The second rule says that for the above static causal law, if $l$ possibly holds in $E(a, \delta)$, the precondition $\psi$ belong to $ph(a, \delta)$ and $\psi$ possibly holds in $E(a, \delta)$ then $l$ also belong to $ph(a, \delta)$.

**Goal representation**

The following rules encode the goal and make sure that it is always achieved at the final step.

$$goal(T_1) \leftarrow holds(G, T_1)$$ \hspace{1cm} (49)

$$goal(T + 1) \leftarrow goal(T) \leftarrow \neg goal(h + 1)$$ \hspace{1cm} (50)

The first rule says that the goal is satisfied at time step $T$ if all of its subgoals are satisfied at that time step. The second rule states that if the goal is satisfied at time step $T$ then it is also satisfied at time step $T + 1$, which implies that it is also satisfied at the final time step $h + 1$. The last rule enforces the goal to be satisfied at the final step.
Domain independent rules

- **Rules encoding the successor partial state.** $\pi_h(\mathcal{P})$ contains the following two independent rules of $\pi_h(\mathcal{P})$

\[
\begin{align*}
ph(L, T) & \leftarrow \text{not holds}(\neg L, T), \text{not ef}(-L, T) \\
holds(L, T+1) & \leftarrow \text{not ph}(\neg L, T)
\end{align*}
\] (52)

The first rule says that a fluent literal $L$ belongs to $ph(a, \delta)$ if it possibly holds in $\delta$ and does not belong to $\neg E(a, \delta)$. The second rule defines the successor partial state according to the $T^{ph}(\mathcal{D})$ approximation.

- **Rules for generating action occurrences.**

\[
1\{occ(A, T)\} \leftarrow \text{not goal}(T)
\] (54)

This rule says that at least one elementary action must occur at time step $T$ if the goal has not been achieved.

- **Auxiliary Rules.**

\[
\begin{align*}
literal(F) & \leftarrow \\
literal(\neg F) & \leftarrow \\
\text{contrary}(F, \neg F) & \leftarrow \\
\text{contrary}(\neg F, F) & \leftarrow
\end{align*}
\] (55) (56) (57) (58)
We have presented the program $\pi_h(P)$. In the subsequent subsections we will prove its soundness and completeness with respect to the $T_{ph}(D)$ approximation.

### 4.6.1 Soundness of CPASP

Let $P = \langle D, \delta, G \rangle$ be a planning problem instance and $S$ be an answer set for $\pi_h(P)$. For any integer $1 \leq i \leq h$, let $a_i = \{ e \mid \text{occ}(e, i) \in S \}$ and let

$$
\alpha_i(S) = \begin{cases} 
\langle \rangle & \text{if } i = 0 \\
\langle \alpha_{i-1}(S), a_i \rangle & \text{if } a_i \neq \emptyset \\
\alpha_{i-1}(S) & \text{otherwise}
\end{cases}
$$

It is easy to see that by this definition, $\alpha_h(S) = \langle a_1, \ldots, a_k \rangle$ where $k$ is some integer between $1$ and $h$ such that $a_i \neq \emptyset$ for $1 \leq i \leq k$ and $a_{k+1} = \emptyset$.

We have the following theorem about the soundness of CPASP with respect to $T_{ph}(D)$.

**Theorem 4.9** Let $P = \langle D, \delta, G \rangle$ be a planning problem instance and $h \geq 1$ be an integer. Let $S$ be an answer set of $\pi_h(P)$. Then $\alpha_h(S)$ is a solution of $P$ with respect to $T_{ph}(D)$.

**Proof.** See Section D.5.1.

Since a solution of a planning problem $P$ with respect to an approximation is also a solution of $P$, it follows from the theorem that any solution returned by CPASP is also a solution of $P$ (Definition 2.15).
4.6.2 Completeness of CPASP

We now turn our attention to what kind of solutions CPASP can generate. It is not difficult to see that CPASP cannot generate every possible solution of $P = \langle D, \delta, G \rangle$ with respect to the $T^{ph}(D)$ approximation. Rather, each solution generated by CPASP must satisfy the property that the goal is not achieved at anywhere else in the execution path of the solution other than at the end. Therefore, CPASP is not complete in the sense that there is no one-to-one correspondence between the answer sets and solutions of $P$ with respect to $T^{ph}(D)$. However, it is complete in the sense that for each solution $\alpha$ of $P$ with respect to $T^{ph}(D)$, there exists an integer $h$ such that $\pi_h(P)$ has an answer set $S$ whose corresponding plan, $\alpha_h(S)$, can be obtained from $\alpha$ by applying the following transformation (called reduct operation).

**Definition 4.12** Let $P = \langle D, \delta, G \rangle$ and $\alpha$ be a solution of $P$ with respect to the $T^{ph}(D)$ approximation. The reduct of $\alpha$, denoted by $\text{reduct}(\alpha)$, is defined as $\alpha[i]$ where $i$ is the smallest integer such that $\alpha[i]$ is a solution of $P$ with respect to $T^{ph}(D)$.

Clearly, by this definition, for a solution $\alpha$ of $P$ with respect to $T^{ph}(D)$, $\text{reduct}(\alpha)$ always exists. We have the following theorem about the completeness of CPASP.

**Theorem 4.10** Let $P = \langle D, \delta, G \rangle$ be a planning problem instance and $\alpha$ be a solution of $P$ with respect to $T^{ph}(D)$. Then, there exists an integer $h$ such that $\pi_h(P)$ has an answer set $S$ and $\alpha_h(S) = \text{reduct}(\alpha)$.

**Proof.** See Section D.5.2.
Note that this theorem does not imply that CPASP can find any solution $\alpha$ of $\mathcal{P}$ that satisfies the property $\alpha = \text{reduct}(\alpha)$. Rather it only states that CPASP can find any solution of this type of $\mathcal{P}$ with respect to the $T_{ph}(\mathcal{D})$ approximation.

4.7 A C++ Implementation of CPASP

In the previous section, we have presented an approximation based conformant planner in logic programming paradigm. Our experiments (presented in Section 4.8) show that CPASP is relatively good in planning with concurrent actions and in domains rich in static causal laws. However, it does not perform well in domains with large grounded representation. One of the main reasons for this problem is because the existing answer set solvers do not scale up well to programs that require large grounded representation. Various approaches have been proposed to attack this issue from different perspectives (see for example, [15, 42]). To further investigate the usefulness of approximations, we developed a sequential conformant planner, called CPA, in the C++ programming language. Two approximations implemented in CPA are $T_{ph}(\mathcal{D})$ and $T_{pc}(\mathcal{D})$ and the planner provides the user with an option of selecting the approximation being used. Like CPA+, CPA accepts domains written in the $\mathcal{AL}$ language with the initial state being described as CNF formulas.

The search algorithm of CPA (depicted Figure 4.4) is similar to the search algorithm of CPA+ where the best first strategy is employed and the number of fulfilled
Algorithm FwdPLAN(\(\mathcal{D}, \Delta, \mathcal{G}\))

INPUT: A planning problem \(\mathcal{P} = (\mathcal{D}, \Delta, \mathcal{G})\)

OUTPUT: A solution of \(\mathcal{P}\) or FAILURE

1. BEGIN
2. if \(\Delta\) satisfies \(\mathcal{G}\) then return \(\langle\rangle\)
3. Queue = \{\(\langle\Delta, \langle\rangle\rangle\)\} \(\text{Visited} = \{\Delta\}\)
4. while Queue is not empty
5. select \(\langle\Delta_1, p\rangle\) with the best heuristic value from Queue
6. for each action \(a\) executable in \(\Delta_1\)
7. \(\Delta_2 = \bigcup_{\delta \in \Delta_1} \text{RESA}(a, \delta)\)
8. if \(\Delta_2\) satisfies \(\mathcal{G}\) then return \(\langle p, a \rangle\)
9. else if \(\Delta_2 \not\in \text{Visited}\)
10. compute heuristic for \(\Delta_2\)
11. insert \(\langle\Delta_2, \langle p, a \rangle\rangle\) into Queue
12. insert \(\Delta_2\) into Visited
13. return FAILURE
14. END

Figure 4.4: The search algorithm of CpA

subgoals is used as the heuristic function. Depending on the input option, the algorithm
will invoke the procedure RESPH or RESPC to compute the successor belief partial
state (procedure RESA, Line 7).

4.8 Experiments

In this section, we will present our experimental results with CPASP and CPASP.

The platform for testing is a 2.4 GHz CPU, 768MB RAM machine, running Slackware
10.0 operating system. Time limit was set to half an hour.

4.8.1 Performance of CPASP

We compared CPASP with three other conformant planners CMBP, DLV\(\kappa\), and
\(\mathcal{C}\)-PLAN (see Appendix A for a brief description of these planners) because these plan-
ners are in the similar spirit as CPASP (that is, a planning problem is translated into an
equivalent problem in a more general setting which can be solved by an off-the-shelf software system).

4.8.1.1 Benchmarks

We prepared two test suites, one of which contains sequential, conformant plan-
ning benchmarks and the other contains concurrent, conformant planning benchmarks.

The sequential test suite includes the domains BT, BTC, Ring\footnote{It should be noted that the Ring domain has disjunctive knowledge about the initial state and the framework presented in Section 4.6 does not allow for this kind of knowledge. However, in Section 4.9 we discuss how CPASP has been extended to cope with disjunctive information} and Cleaner that have been described in Chapter 3. Furthermore, this test suite also includes two new domains:

- **Domino(n)** [124]: We have $n$ dominos standing on a line in such a way that if one of them falls then the domino on its right side also falls. There is a ball hanging close to the leftmost one. Touching the ball causes the first domino to fall. Initially, the states of dominos are unknown. The goal is to have the rightmost one to fall.

- **Gaspipe(n)** [125]: We need to start a flame in a burner which is connected to a gas tank through a pipe line. The gas tank is on the left-most of the pipeline and the burner is on the right-most. The pipe line contains sections that connect with each other by valves. The states of pipe sections can be either pressured or
un-pressured. Opening a valve causes the section on its right side to be pressured if the section on its left is pressured. Moreover, to be safe, a valve can be opened only if the next valve on the line is closed. Closing a valve causes the pipe section on its right side to be un-pressured. There are two kinds of static causal laws. The first one is that if a valve is open and the section on its left is pressured then the section on its right will pressured. Otherwise (either the valve is closed or the section on the left is un-pressured), the pipe on the right side is un-pressured. The burner will start a flame if the pipe connecting to it is pressured. The gas tank is always pressured. The uncertainty we introduce with the initial situation is that the states of valves are unknown. A possible conformant plan will be closing all valves except the first one (that is, the one that directly connects to the gas tank) in the right-to-left order and then opening them in the reverse order.

The concurrent test suite contains four domains $\text{BT}^p$, $\text{BTC}^p$, $\text{Gaspipe}^p$ and $\text{Cleaner}^p$. The $\text{BT}^p$ and $\text{BTC}^p$ domains are modifications the BT and BTC domains respectively in which we allow to dunk different packages into different toilets at the same time. The $\text{Gaspipe}^p$ domain is a modification of the Gaspipe domain, which allows to close multiple valves at the time. In addition, it is possible to open a valve while closing other valves. However, it is not allowed to open and close the same valve or open two different valves at the same time. The final domain in the test suite, $\text{Cleaner}^p$, is a relaxed version of the Cleaner domain where we allow the robot to concurrently
clean multiple objects in the same room.

4.8.1.2 Performance

We ran CPASP on both smodels and cmodels and observed that cmodels yielded better performance in general. The running times of CPASP reported here are with cmodels. We did not test C-PLAN on the sequential planning benchmarks since it is supposed to use for concurrent planning\textsuperscript{2}. In these tables, times are shown in seconds; “-”, denotes a time out or an abnormal termination. Since both DLV\textsuperscript{K} and CPASP require as an input parameter the length of a plan to search for, we ran them by incrementally increasing the plan length, starting from 1\textsuperscript{3}, until a plan is found.

Table 4.1: Performance of CPASP on the BT and BTC domains

<table>
<thead>
<tr>
<th>Problem</th>
<th>CMBP</th>
<th>DLV\textsuperscript{K}</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT(2,2)</td>
<td>0.03</td>
<td>0.04</td>
<td>0.20</td>
</tr>
<tr>
<td>BT(4,2)</td>
<td>0.17</td>
<td>0.55</td>
<td>0.41</td>
</tr>
<tr>
<td>BT(6,2)</td>
<td>0.21</td>
<td>216.55</td>
<td>0.77</td>
</tr>
<tr>
<td>BT(8,4)</td>
<td>0.63</td>
<td>-</td>
<td>6.73</td>
</tr>
<tr>
<td>BT(10,4)</td>
<td>1.5</td>
<td>-</td>
<td>890.06</td>
</tr>
<tr>
<td>BTC(2,2)</td>
<td>0.16</td>
<td>0.12</td>
<td>0.22</td>
</tr>
<tr>
<td>BTC(4,2)</td>
<td>0.26</td>
<td>72.44</td>
<td>0.71</td>
</tr>
<tr>
<td>BTC(6,2)</td>
<td>0.74</td>
<td>-</td>
<td>2.72</td>
</tr>
<tr>
<td>BTC(8,4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BTC(10,4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The running times (in seconds) for sequential benchmarks are shown in Tables 4.1–4.5. As can be seen in Table 4.1, in the BT and BTC domains, CMBP outperforms

\textsuperscript{2}The authors told us that C-PLAN was not intended for searching sequential plans

\textsuperscript{3}We did not start from 0 because none of the benchmarks has a plan of length 0
both $\text{DLV}^\mathcal{K}$ and CPASP on most instances. CPASP, however, has better performance than $\text{DLV}^\mathcal{K}$ in general. As an example, $\text{DLV}^\mathcal{K}$ took more than three minutes to solve the BT(6,2), while it took only 0.77 seconds for CPASP to solve the same instance. In addition, within the time limit, CPASP was able to solve more instances than $\text{DLV}^\mathcal{K}$.

Table 4.2: Performance of CPASP on the Ring domains

<table>
<thead>
<tr>
<th>Problem</th>
<th>CMBP</th>
<th>$\text{DLV}^\mathcal{K}$</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring(2)</td>
<td>0.06</td>
<td>0.10</td>
<td>0.52</td>
</tr>
<tr>
<td>Ring(4)</td>
<td>0.10</td>
<td>2.14</td>
<td>2.98</td>
</tr>
<tr>
<td>Ring(6)</td>
<td>0.48</td>
<td>-</td>
<td>44.43</td>
</tr>
<tr>
<td>Ring(8)</td>
<td>-</td>
<td>-</td>
<td>1424.07</td>
</tr>
<tr>
<td>Ring(10)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

In the Ring domain (Table 4.2), although outperformed by both CMBP and $\text{DLV}^\mathcal{K}$ in some small instances, CPASP is the only planner that was able to solve the Ring(8).

Table 4.3: Performance of CPASP on the Domino domains

<table>
<thead>
<tr>
<th>Problem</th>
<th>CMBP</th>
<th>$\text{DLV}^\mathcal{K}$</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domino(100)</td>
<td>0.26</td>
<td>0.1</td>
<td>0.21</td>
</tr>
<tr>
<td>Domino(200)</td>
<td>1.79</td>
<td>0.35</td>
<td>0.28</td>
</tr>
<tr>
<td>Domino(500)</td>
<td>7.92</td>
<td>2.40</td>
<td>0.74</td>
</tr>
<tr>
<td>Domino(1000)</td>
<td>13.20</td>
<td>13.10</td>
<td>1.23</td>
</tr>
<tr>
<td>Domino(2000)</td>
<td>66.60</td>
<td>62.42</td>
<td>2.41</td>
</tr>
<tr>
<td>Domino(5000)</td>
<td>559.46</td>
<td>-</td>
<td>6.07</td>
</tr>
<tr>
<td>Domino(10000)</td>
<td>-</td>
<td>-</td>
<td>12.584</td>
</tr>
</tbody>
</table>

CPASP seems to work well with domains rich in static causal laws like Domino and Gaspipe. In the Domino domain (Table 4.3), CPASP outperforms all the other planners on most of instances. It took only 2.41 seconds to solve Domino(2000), while
Table 4.4: Performance of CPASP on the Gaspipe domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>CMBP</th>
<th>DLVK</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaspipe(3)</td>
<td>NA</td>
<td>0.13</td>
<td>1.34</td>
</tr>
<tr>
<td>Gaspipe(5)</td>
<td>NA</td>
<td>0.42</td>
<td>2.22</td>
</tr>
<tr>
<td>Gaspipe(7)</td>
<td>NA</td>
<td>42.62</td>
<td>6.18</td>
</tr>
<tr>
<td>Gaspipe(9)</td>
<td>NA</td>
<td>-</td>
<td>39.32</td>
</tr>
<tr>
<td>Gaspipe(11)</td>
<td>NA</td>
<td>-</td>
<td>868.10</td>
</tr>
</tbody>
</table>

both DLVK and CMBP took more than one minute. In fact CPASP could scale up very well to larger instances, e.g., Domino(10000). In the Gaspipe domain (Table 4.4), CPASP also outperforms DLVK: it was able to solve all the instances while DLVK was able to solve only the first three instances.

Table 4.5: Performance of CPASP on the Cleaner domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>CMBP</th>
<th>DLVK</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cleaner(2,2)</td>
<td>0.1</td>
<td>0.104</td>
<td>0.49</td>
</tr>
<tr>
<td>Cleaner(2,5)</td>
<td>0.61</td>
<td>214.69</td>
<td>3.88</td>
</tr>
<tr>
<td>Cleaner(2,10)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cleaner(4,2)</td>
<td>0.13</td>
<td>14.82</td>
<td>2.09</td>
</tr>
<tr>
<td>Cleaner(4,5)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cleaner(4,10)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cleaner(6,2)</td>
<td>4.1</td>
<td>-</td>
<td>224.39</td>
</tr>
<tr>
<td>Cleaner(6,5)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cleaner(6,10)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The Cleaner domain (Table 4.5) turns out to be hard for all the planners: only very small instances could be solved within the time limit. In this domain, CPASP is outperformed by CMBP. To solve the Cleaner(6,2), CMBP took only 4.1 seconds while

\[\text{We tried to test this domain with CMBP but had some problem with the encoding. We contacted with the author of CMBP and are still waiting for response}\]
CPASP took more than 3 minutes. However, CPASP performs better than $DLV^K$ in general: $DLV^K$ reported a timeout with the problem Cleaner(6,2).

We have seen that CPASP can be competitive with CMBP and $DLV^K$ on the sequential benchmarks. Let us move our attention to the concurrent benchmarks (Tables 4.6–4.8).

Table 4.6: Performance of CPASP on the BT$^p$ and BTC$^p$ domains

<table>
<thead>
<tr>
<th>Problem</th>
<th>$C$-PLAN</th>
<th>$DLV^K$</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT$^p$(2,2)</td>
<td>0.07</td>
<td>0.07</td>
<td>0.11</td>
</tr>
<tr>
<td>BT$^p$(4,2)</td>
<td>0.05</td>
<td>0.09</td>
<td>0.26</td>
</tr>
<tr>
<td>BT$^p$(6,2)</td>
<td>1.81</td>
<td>3.06</td>
<td>0.34</td>
</tr>
<tr>
<td>BT$^p$(8,4)</td>
<td>4.32</td>
<td>10.52</td>
<td>0.24</td>
</tr>
<tr>
<td>BT$^p$(10,4)</td>
<td>-</td>
<td>-</td>
<td>1.91</td>
</tr>
<tr>
<td>BTC$^p$(2,2)</td>
<td>0.05</td>
<td>0.05</td>
<td>0.13</td>
</tr>
<tr>
<td>BTC$^p$(4,2)</td>
<td>0.07</td>
<td>0.90</td>
<td>0.30</td>
</tr>
<tr>
<td>BTC$^p$(6,2)</td>
<td>7.51</td>
<td>333.27</td>
<td>0.67</td>
</tr>
<tr>
<td>BTC$^p$(8,4)</td>
<td>-</td>
<td>-</td>
<td>0.50</td>
</tr>
<tr>
<td>BTC$^p$(10,4)</td>
<td>-</td>
<td>-</td>
<td>1192.45</td>
</tr>
</tbody>
</table>

Table 4.7: Performance of CPASP on the Gaspipe$^p$ domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>$C$-PLAN</th>
<th>$DLV^K$</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaspipe$^p$(3)</td>
<td>-</td>
<td>0.08</td>
<td>0.40</td>
</tr>
<tr>
<td>Gaspipe$^p$(5)</td>
<td>-</td>
<td>0.17</td>
<td>0.75</td>
</tr>
<tr>
<td>Gaspipe$^p$(7)</td>
<td>-</td>
<td>0.44</td>
<td>1.22</td>
</tr>
<tr>
<td>Gaspipe$^p$(9)</td>
<td>-</td>
<td>17.44</td>
<td>3.17</td>
</tr>
<tr>
<td>Gaspipe$^p$(11)</td>
<td>-</td>
<td>-</td>
<td>8.83</td>
</tr>
</tbody>
</table>

As can be seen from Tables 4.6 & 4.7, CPASP outperforms both $DLV^K$ and $C$-PLAN on most instances of the BT$^p$, BTC$^p$, and Gaspipe$^p$ domains. Furthermore, CPASP is the only planner that was able to solve all the instances in the test suite.
In the Cleaner\textsuperscript{p} domain (Table 4.8), \texttt{C-PLAN} is the best. To solve the Cleaner\textsuperscript{p}(6,10) problem, \texttt{C-PLAN} took only 0.35 seconds, whereas \texttt{DLV}\textsuperscript{K} reported a timeout and \texttt{CPASP} needs 3.73 seconds.

4.8.2 Performance of CPA

We compared CPA with three planners CFF, KACMBP, and POND (see Appendix A for a short description of these systems). The running times for CFF, KACMBP, and POND on the benchmark domains have been reported in Chapter 3. However, to be easy for our discussion, we copy and paste the results herewith. Furthermore, we want to note that the encodings of some domains in CPA are slightly different from those in CPA\textsuperscript{+} (Chapter 3). Specifically, the encodings in CPA contain some static causal laws which were compiled away in the CPA\textsuperscript{+} encodings.

The results are shown in Table 4.9–4.12 where CPA\textsuperscript{ph} and CPA\textsuperscript{pc} denote the planner CPA with the approximation option being the \texttt{T}\textsuperscript{ph}(D) and \texttt{T}\textsuperscript{pc}(D) respectively. As usual, times are shown in seconds; “-“ indicates a timeout or an abnormal termina-
Table 4.9: Performance of CPA on the Logistics domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>KACMBP</th>
<th>POND</th>
<th>CFF</th>
<th>CPA^ph</th>
<th>CPA^pc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log(2,2,2)</td>
<td>0.19</td>
<td>1.11</td>
<td>0.03</td>
<td>0.15</td>
<td>0.16</td>
</tr>
<tr>
<td>Log(2,3,3)</td>
<td>355.96</td>
<td>11.89</td>
<td>0.06</td>
<td>8.95</td>
<td>9.543</td>
</tr>
<tr>
<td>Log(3,2,2)</td>
<td>2.10</td>
<td>4.02</td>
<td>0.06</td>
<td>11.87</td>
<td>4.54</td>
</tr>
<tr>
<td>Log(3,3,3)</td>
<td>29.8</td>
<td>24.66</td>
<td>0.12</td>
<td>409.68</td>
<td>435.55</td>
</tr>
<tr>
<td>Log(4,3,3)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

In the logistic domain (Table 4.9), both KACMBP and CPA had difficulty in finding plans. Although KACMBP is better than CPA \(^5\), its performance is far from that of CFF which solved every instance in less than one second. We believe that the poor performance of CPA on this domain lies in the simple built-in heuristic function.

Table 4.10: Performance of CPA on the Ring domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>KACMBP</th>
<th>POND</th>
<th>CFF</th>
<th>CPA^ph</th>
<th>CPA^pc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring(2)</td>
<td>0.00</td>
<td>0.15</td>
<td>0.06</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Ring(3)</td>
<td>0.00</td>
<td>0.08</td>
<td>0.23</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Ring(4)</td>
<td>0.00</td>
<td>0.25</td>
<td>3.86</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Ring(5)</td>
<td>0.00</td>
<td>0.96</td>
<td>63.67</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>Ring(10)</td>
<td>0.02</td>
<td>-</td>
<td>-</td>
<td>1.01</td>
<td>1.05</td>
</tr>
<tr>
<td>Ring(15)</td>
<td>0.04</td>
<td>-</td>
<td>-</td>
<td>6.76</td>
<td>6.10</td>
</tr>
<tr>
<td>Ring(20)</td>
<td>0.15</td>
<td>-</td>
<td>-</td>
<td>27.44</td>
<td>22.68</td>
</tr>
<tr>
<td>Ring(25)</td>
<td>0.32</td>
<td>-</td>
<td>-</td>
<td>79.58</td>
<td>64.60</td>
</tr>
</tbody>
</table>

In the Ring domain (Table 4.10), although CPA could solve all the instances, its performance is not as good as KACMBP, e.g., to solve the biggest instance, Ring(25), CPA needed more than one minute, while KACBMP needed only 0.32 seconds. CFF

\(^5\)From now on, we use CPA to refer to both CPA^ph and CPA^pc
and POND on the contrary did not perform well on this domain. Both of them could scale up to Ring(5) only.

Table 4.11: Performance of C\(\text{P}A\) on the BTUC domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>KACMBP</th>
<th>POND</th>
<th>CFF</th>
<th>C(\text{P}A)(^{ph})</th>
<th>C(\text{P}A)(^{pc})</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTUC(5,1)</td>
<td>0.00</td>
<td>0.03</td>
<td>0.03</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>BTUC(10,1)</td>
<td>0.01</td>
<td>0.07</td>
<td>0.05</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>BTUC(20,1)</td>
<td>0.05</td>
<td>0.57</td>
<td>0.17</td>
<td>0.07</td>
<td>0.03</td>
</tr>
<tr>
<td>BTUC(50,1)</td>
<td>0.51</td>
<td>28.69</td>
<td>5.33</td>
<td>0.82</td>
<td>0.33</td>
</tr>
<tr>
<td>BTUC(100,1)</td>
<td>3.89</td>
<td>682.33</td>
<td>121.8</td>
<td>6.24</td>
<td>2.36</td>
</tr>
<tr>
<td>BTUC(5,5)</td>
<td>0.04</td>
<td>0.10</td>
<td>0.04</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>BTUC(10,5)</td>
<td>0.09</td>
<td>0.65</td>
<td>0.07</td>
<td>0.04</td>
<td>0.02</td>
</tr>
<tr>
<td>BTUC(20,5)</td>
<td>0.30</td>
<td>7.28</td>
<td>0.16</td>
<td>0.18</td>
<td>0.09</td>
</tr>
<tr>
<td>BTUC(50,5)</td>
<td>1.66</td>
<td>348.28</td>
<td>4.70</td>
<td>1.90</td>
<td>0.83</td>
</tr>
<tr>
<td>BTUC(100,5)</td>
<td>6.92</td>
<td>-</td>
<td>113.95</td>
<td>12.13</td>
<td>5.266</td>
</tr>
<tr>
<td>BTUC(5,10)</td>
<td>0.11</td>
<td>0.35</td>
<td>0.03</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>BTUC(10,10)</td>
<td>0.30</td>
<td>2.50</td>
<td>0.05</td>
<td>0.07</td>
<td>0.04</td>
</tr>
<tr>
<td>BTUC(20,10)</td>
<td>0.97</td>
<td>27.69</td>
<td>0.13</td>
<td>0.40</td>
<td>0.19</td>
</tr>
<tr>
<td>BTUC(50,10)</td>
<td>5.39</td>
<td>960.00</td>
<td>4.04</td>
<td>3.74</td>
<td>1.63</td>
</tr>
<tr>
<td>BTUC(100,10)</td>
<td>35.83</td>
<td>-</td>
<td>102.56</td>
<td>20.94</td>
<td>9.80</td>
</tr>
</tbody>
</table>

In the BTUC domain (Table 4.11), C\(\text{P}A\) is the best in general. For example, to solve the largest instance, BTUC(100,10), C\(\text{P}A\)\(^{ph}\) and C\(\text{P}A\)\(^{pc}\) took 20.94 and 9.80 seconds respectively, while KACMBP and CFF took 35.83 and 102.56 seconds respectively and POND reported a timeout. However, CFF seems to have no problem when the number of toilets increases, while there is a significant increase in the amount of solving time for both KACMBP and C\(\text{P}A\). The increase in the amount of time for C\(\text{P}A\) is more reasonable than that for KACMBP. For example, with a fixed number of packages 100, when the number of toilets increase from 5 to 10, the amount of solving time for CFF even decreases, while that for KACMBP about 5 times increases and C\(\text{P}A\)’s
is around doubled. It should be noted that the performance of $\text{CPA}^{pc}$ is better than the performance of $\text{CPA}^{ph}$ in this domain. One of the reasons for this is that in the computation of the successor partial state according to the $T^{ph}(\mathcal{D})$ we need to take into account all the fluents, while that is not the case with $T^{pc}(\mathcal{D})$.

Table 4.12: Performance of CPA on the Cleaner domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>KACMBP</th>
<th>POND</th>
<th>CFF</th>
<th>CPA$^{ph}$</th>
<th>CPA$^{pc}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cleaner(2,5)</td>
<td>0.01</td>
<td>0.17</td>
<td>0.03</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>Cleaner(2,10)</td>
<td>0.08</td>
<td>0.85</td>
<td>0.07</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>Cleaner(2,20)</td>
<td>0.62</td>
<td>15.87</td>
<td>0.15</td>
<td>0.19</td>
<td>0.07</td>
</tr>
<tr>
<td>Cleaner(2,50)</td>
<td>13.55</td>
<td>-</td>
<td>0.80</td>
<td>2.76</td>
<td>0.92</td>
</tr>
<tr>
<td>Cleaner(2,100)</td>
<td>185.39</td>
<td>-</td>
<td>5.72</td>
<td>22.71</td>
<td>7.51</td>
</tr>
<tr>
<td>Cleaner(5,5)</td>
<td>0.01</td>
<td>1.46</td>
<td>0.11</td>
<td>0.07</td>
<td>0.04</td>
</tr>
<tr>
<td>Cleaner(5,10)</td>
<td>0.09</td>
<td>12.86</td>
<td>0.24</td>
<td>0.26</td>
<td>0.16</td>
</tr>
<tr>
<td>Cleaner(5,20)</td>
<td>7.82</td>
<td>214.83</td>
<td>0.85</td>
<td>1.78</td>
<td>0.88</td>
</tr>
<tr>
<td>Cleaner(5,50)</td>
<td>227.82</td>
<td>-</td>
<td>14.36</td>
<td>26.66</td>
<td>11.66</td>
</tr>
<tr>
<td>Cleaner(5,100)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>214.27</td>
<td>92.81</td>
</tr>
</tbody>
</table>

CFF also obtained good performance in the Cleaner domain (Table 4.12) and it is the only planner that could solve all instances. KACMBP behaves well only on small problems but does not scale up as well as CFF and CPA. CFF is very good at all instances of the domain, except Cleaner(5,100) where the constant denoting the maximum length of the plan was exceeded.

In the Domino domain (Table 4.13), except CPA, all the other planners did not perform well. The reason is that this domain is a rich source of static causal laws – a feature not supported by KACMBP, CFF, and POND. Thus, we had to compile them away when encoding this domain for these planners and this complication requires the
Table 4.13: Performance of CPA on the Domino domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>KACMBP</th>
<th>POND</th>
<th>CFF</th>
<th>CPA^ph</th>
<th>CPA^pc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domino(10)</td>
<td>0.01</td>
<td>1.72</td>
<td>0.05</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Domino(50)</td>
<td>0.27</td>
<td>-</td>
<td>4.44</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Domino(100)</td>
<td>2.56</td>
<td>-</td>
<td>-</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>Domino(200)</td>
<td>29.10</td>
<td>-</td>
<td>-</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Domino(500)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>Domino(1000)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>Domino(2000)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.63</td>
<td>0.65</td>
</tr>
<tr>
<td>Domino(5000)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3.81</td>
<td>4.01</td>
</tr>
</tbody>
</table>

Introduction of extra actions and fluents. As a result, the performance of these planners is not as good as the performance of CPA.

As stated, our planner is sound but not complete, i.e., theoretically speaking, CPA cannot solve some planning problems, even when the initial state is complete. To make sure our approach can cover a broader spectrum of practical planning problems, we also tested CPA with classical planning problems. The first domain considered is the Blocks World with five problems described in [41]. We then tested with problems in the Rover domain\(^6\). Five problems, different from each other in the numbers of waypoints, rovers, cameras, rock and soil samples, and objectives, were experimented with. It turns out that CPA can solve all those problems.

\(^6\)[http://planning.cis.strath.ac.uk/competition/]
4.9 Discussion

4.9.1 Handling Disjunctive Information about the Initial State in CPASP

One of the limitations of CPASP is that it does not consider planning problem instances with disjunctive information about the initial state, i.e., instances of the form $\mathcal{P} = \langle \mathcal{D}, \Delta, \mathcal{G} \rangle$ where $\Delta$ contains more than one initial partial state, which we call disjunctive planning problem instances. In this section, we discuss how to extend the program $\pi_h(\mathcal{P})$ to handle this type of planning problem instances. For more details, we refer the reader to [104].

The new extended program, denoted by $\hat{\pi}_h(\mathcal{P})$, has an extra constant called $worlds$, to denote the number of initial partial states in $\Delta$, i.e., $worlds = |\Delta|$. The predicates of $\hat{\pi}_h(\mathcal{P})$ are similar to the predicates of $\pi_h(\mathcal{P})$ except an extra parameter $W$ in each predicate $holds$, $ef$, and $ph$, which is used to indicate a possible world:

- $holds(l, T, W)$ is true if the fluent literal $l$ holds at time-step $T$ in the world $W$,
- $ef(l, T, W)$ is true if the fluent literal $l$ is a direct effect of an action that occurs at the previous time step in the world $W$,
- $ph(l, T, W)$ is true if fluent literal $l$ possibly holds at time step $T$ in the world $W$, and
- $goal(T, W)$ is true if the goal $\mathcal{G}$ holds at time step $T$ in the world $W$.

The set of rules of $\hat{\pi}_h(\mathcal{P})$ is very similar to the set of rules of $\pi_h(\mathcal{P})$. Suppose $\Delta =$
\{\delta_1, \ldots, \delta_k\}$. First, \(\hat{\pi}_h(P)\) modifies the rule (54) of \(\pi_h(P)\) to take into consideration disjunctive information about the initial state. Specifically, for each \(1 \leq i \leq k\), \(\hat{\pi}_h(P)\) contains the following set of facts

\[
\text{holds}(\delta_i, 1, i) \leftarrow \tag{59}
\]

Second, \(\hat{\pi}_h(P)\) has the following set of the constraints which is similar to the rule (51) of \(\pi_h(P)\): for each \(1 \leq i \leq k\), the constraint

\[
\leftarrow \text{not \text{goal}}(h + 1, i) \tag{60}
\]

belongs to \(\hat{\pi}_h(P)\).

Finally, the remaining rules of \(\hat{\pi}_h(P)\) are obtained from the remaining rules of \(\pi_h(P)\) by replacing predicates \(\text{holds}(l, T)\), \(\text{ef}(l, T)\), \(\text{ph}(l, T)\), and \(\text{goal}(T)\) with the new predicates \(\text{holds}(l, T, W)\), \(\text{ef}(l, T, W)\), \(\text{ph}(l, T, W)\), and \(\text{goal}(T, W)\) respectively (if applicable). For example, the rule (45) will become

\[
\text{ph}(l, T + 1, W) \leftarrow \text{o}(e, T), \text{not \text{e}}(\neg l, T, W), \text{not \text{h}}(\neg \psi, T, W). \tag{61}
\]

Similarly, the rule (53) becomes

\[
\text{holds}(L, T + 1, W) \leftarrow \text{not \text{ph}}(\neg L, T, W). \tag{62}
\]

It is easy to see that if \(S\) is an answer set of \(\hat{\pi}_h(P)\) then \(\alpha_h(S)\) is a solution of \(P\) with respect to the \(T^{ph}(D)\).
4.9.2 Implementation: Imperative Language vs Logic Programming

The experimental results in the previous section indicate that the C++ planner CPA yields better performance than the logic programming based planner CPASP in most of the sequential benchmarks. However, CPASP is competitive for the planning problems with short solutions. Our experimental results also show that in problems with complex static causal laws, CPASP seems to do better than CPA and other state-of-the-art planners but it does not do well on large problems. One of the main reasons for this weakness of CPASP is its reliance on a general answer set solver in computing a solution. On the other hand, the use of logic programming brings a number of advantages. We will now discuss some of these advantages in detail.

1. **Generating Concurrent Plans**: Most of planners implemented in logic programming paradigm such as CPASP and DLV$^C$ can generate concurrent plans. In CPASP, this is achieved by the rule (54). On the other hand, most of the state-of-the-art planners do not compute concurrent plans.

2. **Incorporation of Control Knowledge and/or Preferences**: Logic programming provides an ideal environment for adding control knowledge and/or preferences in searching for plans satisfying some qualitative criteria. For example, in the bomb in the toilet domain with one package, flushing the toilet followed by dunking the package ($flush; dunk$) is a plan achieving the goal of having the package disarmed, even when we know that the toilet is unclogged in the initial situation.
Although it is a valid solution, this plan is hardly a preferable one. The problem is that the action theory does not include knowledge stipulating the planner not to consider such a plan when the plan with a single action *dunk* would be sufficient. This type of knowledge cannot be represented by an executability condition for *flush*. As such, for planners relying on a fixed representation language (e.g. PDDL), the incorporation of knowledge in planning requires a lot of work (e.g. TPlan [4] develops separate modules to deal with temporal knowledge). In logic programming, we can easily express the above knowledge by the constraint

\[ \leftarrow \text{occ}(\text{flush}(t), T), \text{holds}(\neg \text{clogged}(t), T). \]

which disallows answer sets in which the action *flush* occurs when the toilet is *unclogged*. The paper [128] discusses in detail how different types of control knowledge can be easily incorporated in answer set planning.

3. **Dealing with Complex Initial Situations**: Section 4.9.1 discusses how CPASP can be easily adapted to handle disjunctive information about the initial situations. This type of knowledge generates multiple branches in the search trees and can be dealt with fairly efficiently by state-of-the-art sequential planners. Yet, the specification of the initial situation in CPASP and its extension described in Section 4.9.1 remains simple in the sense that the initial state can be expressed by a set of facts. While this is adequate in all benchmarks found in the literature, the following example, discussed in [50], shows that allowing the specification
of more complex initial situations is necessary and useful sometime.

Example 4.11 (Complex initial situation) Assume that the packages in the bomb in the toilet domain are coming from different sources belonging to one of two hierarchically structured organizations, called \( b \) (bad) and \( g \) (good). The hierarchies are described in the usual way using relation \( \text{link}(D_1, D_2) \) which indicates that a department \( D_1 \) is a subdivision of a department \( D_2 \). Organization \( g \) for instance can be represented by a collection of atoms:

\[
\begin{align*}
\text{link}(d_1, g) \leftarrow \\
\text{link}(d_2, g) \leftarrow \\
\text{link}(d_3, d_1) \leftarrow \\
\text{link}(d_4, d_1) \leftarrow 
\end{align*}
\]

These atoms represent the fact that \( d_i \ (i = 1, 2, 3, 4) \) are departments in the good company. It is known that packages coming from the organization \( g \) are safe and the bomb is sent by someone working for \( b \). There are packages labeled by the name of the department the sender works for, which can be recorded by the atom \( \text{from}(P, D) \) - package \( P \) came from department \( D \). There are also some unlabeled packages. The initial situation of the modified bomb in the toilet problem will be described by the program \( H \) consisting of the above atoms and the following rules which define the organization the package came from and our trust in the good guys from \( g \).

\[
\begin{align*}
\text{from}(P, D_2) & \leftarrow \text{from}(P, D_1), \text{link}(D_1, D_2) \\
\text{holds}(\neg \text{armed}(P), 1) & \leftarrow \text{from}(P, g)
\end{align*}
\]
As usual, $P$ ranges over packages and $D$’s range over the departments. It is easy to see that such a program has a unique answer set and can be used to specify the initial situation.

It is easy to see that for $\pi_h(P)$ (as well as for $\tilde{\pi}_h(P)$) to work with the planning problems with this type of initial situation, we only need to replace the rule (42) by the above program.

4.10 Related Work

4.10.1 Approximate Reasoning about Action and Change

One of the existing work to approximate reasoning which is closest to our work is the 0-approximation of Son & Baral [126]. In essence, our approximations is an extension of the 0-approximations to domains with static causal laws. As have been shown in Section 4.5, our approximations coincide with the 0-approximation in domains without static causal laws.

In [41], Eiter et. al. propose a language called $\mathcal{K}$ for formalizing action and change. Actions are modeled in $\mathcal{K}$ in terms of how they change the agent’s knowledge rather than how they change the physical states of the domain. Action theories in $\mathcal{K}$ resemble logic programs in that both classical negations and default negations are allowed in the specification of actions’ effects as well as static causal laws. The semantics of $\mathcal{K}$ is defined based on what they call knowledge states which are very similar to our notion of partial states here. Nevertheless, for an action and a partial state, the semantics of
\(\mathcal{K}\) allows for more than one possible successor knowledge state but we have only one successor partial state. Another work that follows the knowledge based approach is that of Bacchus and Petrick [5, 111] with the addition of sensing actions. Their approach is very similar to the STRIPS approach [45] where effects of actions are described in terms of what would be added to and what would be deleted from the agent’s knowledge. Our work differ from these work in that we still describe actions and effects in terms of how they change the physical states of the domain but our reasoning and planning are based on the agent’s knowledge (partial states) which approximates the set of possible states of the domain.

In [87] Liu and Levesque introduce a procedure, called PV, to reasoning about action and change in which action theories are described in situation calculus. To the best of our knowledge, although there have been several efforts, e.g., [85], to translate an action theory with static causal laws into an equivalent situation calculus theory, the degree of satisfaction varies. Without static causal laws and sensing actions, our approximations coincides with the procedure PV on answering queries in propositional action theories.

Recently, Gelfond & Morales [50] propose an approximation of \(\mathcal{AL}\) action theories which is described in the terms of logic program that capture the transitions between (partial) states but is guaranteed to be sound with the possible world semantics only when the bodies of static causal laws contain at most one literal. The possibly-holds approach is very much influenced by this work.
4.10.2 Conformant Planning

Conformant planning has been approached from many directions by many authors in the field. One of the simplest (but efficient) approaches is to formulate a conformant planning instance as a search problem in the space of belief states and then use a search algorithm such as best-first search, A*, etc. to find solutions. Our planner CPA follows this approach. Perhaps, the planner GPT by Bonet and Geffner [19] is the first conformant planner that makes this idea precise. GPT uses the $A^*$ algorithm to implement the search. The heuristic function used by GPT is based on the estimate of the distance from the current belief state $S$ to the goal, $h_{dp}(S)$. Basically, $h_{dp}(S)$ is the maximum among estimates of distances, $V^*_{dp}(s)$, from an individual state $s \in S$ to the goal. The heuristic function $h_{dp}(S)$ can be computed efficiently (in particular, in polynomial time in the size of $S$) if the size of $S$ is not too large.

Another belief state search planner is CMBP by Cimatti and Roveri [31], which uses the breadth-first strategy. Unlike GPT that appeals to the heuristic technique to improve the search, CMBP takes advantage of recent enhancement in model checking techniques. Specifically, CMBP uses binary decision diagrams (BDD) [24] to represent belief states and expand the search space and to provide an efficient computational platform. Recently, symbolic techniques and heuristic techniques have been combined together in the implementation of a conformant planner, called KACBMP [34]. The main advantage of using BDDs to represent belief states is that the size of a BDD seems
not to depend on the uncertainty degree of the belief state. However, one of the main
disadvantages of this method is that the size of BDDs becomes too large in some cases,
affecting the overall performance of the system. Brafman and Hoffmann [23] therefore
suggest another method for representing belief states. In their planner Conformant-FF
(CFF), each belief state is represented just by the initial state specification together with
the sequence of actions that leads to that belief state. To check if a belief state satisfies
the goal, it checks if each goal proposition is contained in the intersection of all states
in the belief state. This reasoning process is done by deciding the solvability of a CNF
formula (using a SAT solver) that captures the execution of the sequence of actions that
leads to the belief state. The heuristic used by CFF is an extension of the heuristic of its
counterpart in classical setting, FF [61]. KACMBP and Conformant-FF are now known
as ones of the fastest state-of-the-art conformant planners.

Some other planners, e.g., CAltAlt by Bryce and Kambhampati [27], imple-
ments a backward search algorithm in the belief space where belief states. The heuris-
tics of CAltAlt are extracted from the planning graph(s) for the relaxed problem. Mul-
tiple planning graphs are compactly represented as a labeled uncertainty graph (LUG)
[26].

Another approach is to translate a planning problem into a satisfiability (SAT)
problem whose satisfying truth assignments correspond to the plans of the planning
problem. Some other researchers [30, 115] followed the SAT-based approach [67, 68]
to solve conformant planning problem. The basic idea is to generate a possible so-
sultion to the original problem and then test if it is indeed a solution to the problem. A possible solution (of length $n$) to a planning problem $P$ is a sequence of actions $\pi = a_0a_1 \ldots a_{n-1}$ such that there exists a sequence of states $s_0, \ldots, s_n$ and (i) $s_0$ is a possible initial state, (ii) $s_i \in T(a_{i-1}, s_{i-1})$, and (iii) $s_n \in G$. Since, in conformant planning, we may have uncertainty in the initial state and the outcomes of actions, there may be different sequences of such states. As a result, a possible solution is not guaranteed to be a solution to the planning problem. Therefore, once a possible solution is generated, we need to verify if it is actually a solution. This is done by checking the validity of a propositional formula.

CPASP follows the so-called answer set planning – the term coined by Lifschitz [80] – in which a planning problem is described as a logic program and the answer sets of this logic program correspond to plans of the planning problem. A typical system using this approach is $\mathsf{DLV^{K}}$ by Eiter et. al. [41] which is mentioned before. $\mathsf{DLV^{K}}$ generates conformant solutions by asking the underlying solver, $\mathsf{DLV}$, to repeatedly generate a possible solution and verify if it is a solution, which is in the same spirit as the extension of the SAT-based approach for conformant planning.

4.11 Summary

In this chapter, we introduce a general definition of approximations of $\mathcal{ALC}$ domain descriptions and discuss a methodology for designing approximations. We then present two different approaches to approximating domain descriptions, resulting in
four different approximations. The proposed approximations are deterministic and can be computed very efficiently. We also study the soundness, completeness and complexity of the approximations.

To evaluate the usefulness of the approximations, we develop two conformant planners, one of which is a logic programming based planner (called CPASP) and the other is a heuristic search based planner (CPA). Unlike many state-of-the-art conformant planners, these planners are able to deal with static causal laws directly. Their performance is comparable with state-of-the-art conformant planners over typical benchmark domains as well as over newly invented domains. Due to the simple heuristic used in the implementation of CPA, we believe that the good performance of CPA lies in the efficiency of the approximation. It also demonstrates that research in reasoning about action and change in general, and approximations in particular, can positively impact the development of practical planners.

The work presented in this chapter can be extended in many directions. First, we believe that the sufficient conditions for the completeness of $\models^A$ is somewhat limited and can be strengthened. Second, we can combine the two approaches to create a new approximation that is stronger than both $T^{ph+}(D)$ and $T^{pc+}(D)$ by computing the set of fluent literals that possibly holds and the set of fluent literals that possibly changes in parallel. Third, the performance of CPASP and CPA can be improved by adding domain specific knowledge [4] and/or by implementing a more informative heuristic. Finally, we can extend our approximations to domains with richer expressiveness features, such
as, multi-valued fluents, sensing actions, continuous changes, etc. In the next chapter, we will extend them to incorporate sensing actions.
5.1 Introduction

In addition to concurrent actions and static causal laws, sensing actions (a.k.a knowledge producing actions) play an important part of every dynamic domain. Unlike a non-sensing action, the execution of a sensing action does not change the state of the domain; instead, it only affects the knowledge of the agent about the state of the domain. Sensing actions are needed not only for domains with incomplete information about the initial state but also for domains with uncertainty about the effect of actions or for domains with exogenous actions [74].

Two important questions arise in the presence of sensing actions: how to reason about sensing actions and what is a plan. The first question leads to the development of mechanisms for representing and reasoning about sensing actions such as [56, 89, 102, 118, 126, 139]. The second question gives rise to the notion of conditional plan which takes into account contingencies that may occur during the execution of actions. Unlike a conformant plan which is simply a sequence of actions, a conditional plan may contain sensing actions and test conditions such as if-then-else, and case-endcase statements (e.g. [74, 89, 126]), or in a more complicated form, a conditional plan may contain while-do statements (e.g., [75]). Consequently, there are planning instances
solvable by conditional planners but not by conformant planners as demonstrated in the following example.

**Example 5.1** Consider a window with a lock that behaves as follows. The window can be in one of the three states *opened*, *closed*\(^1\) or *locked*\(^2\). When the window is closed or opened, pushing it *up* or *down* will get it *open* or *closed* respectively. When the window is closed or locked, flipping the lock will get it locked or closed respectively. Now, consider a security robot that needs to make sure that the window is locked after 9 pm. Suppose that the robot has been told that the window is not open (but whether it is locked or closed is unknown).

Intuitively, the robot can achieve this goal by performing the following conditional plan. First, (1) determine the status of the window. If it is closed, (2.a) lock the window; otherwise (i.e., the window is already locked), simply (2.b) do nothing. Note that in this conditional plan, we do not need to take into account whether the window is open because initially it is assumed that this is not the case.

Observe that no sequence of actions can achieve the goal from every possible initial state. In other words, no conformant solution for this planning problem exists.

\[\square\]

In this chapter we introduce an action language, called \(\mathcal{AL}_\mathcal{K}\), which extends

\(^1\)The window is closed and unlocked

\(^2\)The window is closed and locked
the action language $\mathcal{AL}$ by adding sensing actions and knowledge-producing laws. We then define what is a conditional plan and what is a query in the presence of sensing actions, and extend the approximations presented in the previous chapter to incorporate sensing actions. Interestingly, with some restrictions, the conditional planning problem with respect to the newly developed approximations is NP-complete. This facilitates the development of $\text{ASCP}$, an answer set programming based planner that is capable of generating both conditional and conformant solutions. The framework of $\text{ASCP}$ is similar to the framework of the planner $\text{CPASP}$ (presented in Chapter 4): Given a planning instance with incomplete information about the initial state and sensing actions, we translate it into a logic program whose answer sets – which can be computed using existing answer set solvers (e.g. $\text{cmodels, smodels, DLV, ASSAT, or NoMoRe}$) – correspond to conformant or conditional plans that satisfy the goal. Our experiments show that conditional and conformant planning based on answer set programming can be competitive with other approaches.

The chapter is organized as follows. Section 5.2 presents the syntax of $\mathcal{AL}_K$ and the definitions of conditional plans and queries. Section 5.3 introduces the approximations of $\mathcal{AL}_K$ domain descriptions and discuss about their properties. Section 5.4 describes the implementation of $\text{ASCP}$ in logic programming paradigm. Section 5.5 discusses several properties of $\text{ASCP}$. Section 5.6 compares $\text{ASCP}$ with some other state-of-the-art conformant/conditional planners on several benchmarks in the literature. Section 5.7 relates our work to other existing work. Section 5.8 summarizes the
chapter. The proofs of results in this chapter is listed in Appendix F. An example of the translation of a planning instance is given in Appendix G.

5.2 The Action Language $\mathcal{AL}_K$

5.2.1 Syntax

The alphabet of an $\mathcal{AL}_K$ domain description consists of two disjoint, non-empty sets of symbols: the set $\mathbf{F}$ of fluents, and the set $\mathbf{A}$ of elementary actions. Furthermore, we distinguish two (disjoint) types of elementary actions in $\mathbf{A}$ which we call elementary non-sensing actions and elementary sensing actions. An action $a$ is either a non-empty set of elementary non-sensing actions (called a non-sensing action) or a set of a single elementary sensing action (called a sensing action). For an elementary action $e$, we will treat the action $\{e\}$ as the elementary action $e$. An $\mathcal{AL}_K$ domain description $D$ is a set of statements of the following forms

\[
e \text{causes } l \text{ if } \psi
\]  
\[l \text{ if } \psi
\]  
\[\text{impossible } b \text{ if } \psi
\]  
\[c \text{ determines } \theta
\]

where $e$ is an elementary non-sensing action, $b$ is an action, $c$ is a sensing action, $l$ is a fluent literal, and $\psi$ and $\theta$ are sets of fluent literals. Fluent literals in $\theta$ are sometimes called sensed literals.
Statements of the forms (63)–(65) are similar to statements of the forms (3)–(5) of $\mathcal{AL}$ domain descriptions and are still called dynamic causal law, static causal law, and impossibility condition respectively. When the precondition $\psi$ of these laws is empty we can omit the `if` part from the laws. The main difference between an $\mathcal{AL}_K$ domain description and an $\mathcal{AL}$ domain description is the presence of knowledge-producing laws of the form (66). The intuitive meaning of a knowledge-producing law of the form (66) is that if the sensing action $c$ is executed then the truth values of fluent literals in $\theta$ will be known to the agent. Through this thesis, we will use the following abbreviations.

- For a set of fluent literals $\gamma$,

  \[
  \text{oneof } \gamma
  \]

  denotes the set of static causal laws of the forms

  \[
  \neg g' \text{ if } g
  \]

  and

  \[
  g \text{ if } \neg (\gamma \setminus \{g\})
  \]

  where $g \neq g'$ are two fluent literals in $\gamma$. Basically, this set of static causal laws guarantees that fluent literals in $\gamma$ are mutually exclusive, i.e., in any state of the domain, exactly one of fluent literals in $\gamma$ holds.
• For a sensing action $c$ and a fluent $f$,

\[ c \text{ determines } f \]

denotes the knowledge-producing law:

\[ c \text{ determines } f, \neg f \]

Apparently, it is natural to assume that a sensing action $c$ has to determine the truth value of at least one fluent, i.e., the set of fluent literals $\theta$ in every knowledge-producing law of the form (66) contains at least two fluent literals. In addition, we require that if $\theta$ is not a set of two complementary fluent literals $f$ and $\neg f$ then the fluent literals in $\theta$ are mutually exclusive, i.e., the domain description contains

\[ \text{oneof } \theta \]

Finally, we also assume that each sensing action cannot appear in more than one knowledge-producing law in the domain description.

An $\mathcal{AL}_K$ domain description $\mathcal{D}$ is consistent if the $\mathcal{AL}$ domain description obtained from $\mathcal{D}$ by removing knowledge-producing laws of the form (66) is consistent. In this work, we assume that every $\mathcal{AL}_K$ domain description is consistent.
Example 5.2 The domain in Example 5.1 can be represented in $\mathcal{AL}_K$ as follows.

$$D_{5,2} = \begin{cases} 
\text{impossible } \text{push}_\up \text{ if } \neg \text{closed} \\
\text{impossible } \text{push}_\down \text{ if } \neg \text{open} \\
\text{impossible } \text{flip}_\text{lock} \text{ if } \neg \text{open} \\
\text{impossible } \{ \text{push}_\up, \text{push}_\down \} \\
\text{impossible } \{ \text{push}_\up, \text{flip}_\text{lock} \} \\
\text{impossible } \{ \text{push}_\down, \text{flip}_\text{lock} \} \\
\text{push}_\down \text{ causes } \text{closed} \\
\text{push}_\up \text{ causes } \text{open} \\
\text{flip}_\text{lock} \text{ causes } \text{locked} \text{ if } \text{closed} \\
\text{flip}_\text{lock} \text{ causes } \text{closed} \text{ if } \text{locked} \\
\text{oneof } \text{open, locked, closed} \\
\text{check determines } \text{open, closed, locked} 
\end{cases}$$

The domain description has three elementary non-sensing actions $\text{push}_\up$, $\text{push}_\down$, and $\text{flip}_\text{lock}$ and a sensing action, $\text{check}$, to determine the status of the window. Notice the presence of the static causal law $\text{oneof}$ to establish a mutual exclusion constraint on the fluent literals sensed by the $\text{check}$ action.

Similarly to an $\mathcal{AL}$ action theory, an $\mathcal{AL}_K$ action theory is defined by an $\mathcal{AL}_K$ domain description $\mathcal{D}$ and a non-empty set of valid partial states $\Delta$. A conditional planning instance is a 3-tuple $(\mathcal{D}, \Delta, \mathcal{G})$, where $(\mathcal{D}, \Delta)$ is an $\mathcal{AL}_K$ action theory and $\mathcal{G}$ is a set of fluent literals. In this chapter, we will use the term “a domain description” or “an action theory” to refer to “an $\mathcal{AL}_K$ domain description” or “an $\mathcal{AL}_K$ action theory” unless otherwise stated explicitly.
Remark 5.1  For a domain description $\mathcal{D}$, if $\mathcal{D}$ contains a static causal law

$$l \text{ if } \psi$$

such that $\psi = \emptyset$ then it is easy to see that fluent literal $l$ holds in every possible state. As a result, queries about the truth value of $l$ (or $\neg l$) have a trivial answer and the theory can be simplified by removing all instances of $l$ in other statements. Furthermore, if $\mathcal{D}$ also contains a dynamic causal law of the form

$$e \text{ causes } \neg l \text{ if } \psi$$

then the execution of $e$ (or of an action containing $e$) in a state satisfying $\psi$ will result in an inconsistent state of the world. Thus, the presence of $l$ in the action theory is either redundant or erroneous. For this reason, without loss of generality, we will assume that domain descriptions do not contain any static causal law (64) with $\psi = \emptyset$.

\[ \square \]

Remark 5.2  Since an empty plan can always be used to achieve an empty goal, we will assume that conditional planning instances have non-empty goals.

\[ \square \]

5.2.2 Conditional Plans and Queries

As aforementioned, in the presence of sensing actions, we need to extend the notion of a plan from a sequence of actions so as to allow conditional statements such as if-then-else, while-do, or case-endcase. Notice that an if-then-else statement can
be replaced by a case-endcase statement. Besides, if we are only interested in plans with bounded length then whatever can be represented by a while-do statement with a non-empty body can also be represented by a set of case-endcase statements as well. Therefore, we limit ourselves to conditional plans with the case-endcase construct only. Formally, we consider conditional plans defined as follows (note that our notion of conditional plans here is very similar to the ones introduced in [74, 89, 126]).

**Definition 5.1** A conditional plan is defined recursively as follows.

1. An empty sequence of actions, $\langle \rangle$, is a conditional plan.

2. If $a$ is a non-sensing action and $\beta$ is a conditional plan then $\langle a, \beta \rangle$ is a conditional plan.

3. If $a$ is a sensing action associated with the knowledge-producing law

   $a \text{ determines } g_1, \ldots, g_n$

   and $\beta_j$'s are conditional plans then $\langle a, \text{cases}\{g_j \rightarrow \beta_j\}_{j=1}^n \rangle$ is a conditional plan.

4. Nothing else is a conditional plan.

By this definition, clearly a sequence of actions is also a conditional plan. The execution of a conditional plan of the form $\langle a, \beta \rangle$, where $a$ is a non-sensing action and $\beta$ is another conditional plan, is performed sequentially as usual, i.e., $a$ is executed first,
followed by $\beta$. To execute a conditional plan of the form $\langle a, \text{cases}(\{g_j \rightarrow \beta_j\}_{j=1}^n) \rangle$, $a$ is executed first and then each $g_j$ is evaluated with respect to the agent’s current knowledge. If one of the $g_j$’s, say $g_k$, holds, then the corresponding sub-plan $\beta_k$ will be executed. Observe that because fluent literals $g_j$’s are mutual exclusive, such $g_k$ always exists and is unique.

**Example 5.3** The following are conditional plans of the domain description in Example 5.2:

\[
\begin{align*}
\alpha_1 &= \langle \text{push\_down, flip\_lock} \rangle \\
\alpha_2 &= \langle \text{check, cases} \left( \begin{array}{c}
\text{open} \rightarrow \langle \rangle \\
\text{closed} \rightarrow \langle \text{flip\_lock} \rangle \\
\text{locked} \rightarrow \langle \rangle 
\end{array} \right) \rangle \\
\alpha_3 &= \langle \text{check, cases} \left( \begin{array}{c}
\text{open} \rightarrow \langle \text{push\_down, flip\_lock} \rangle \\
\text{closed} \rightarrow \langle \text{flip\_lock, flip\_lock, flip\_lock} \rangle \\
\text{locked} \rightarrow \langle \rangle 
\end{array} \right) \rangle \ (69)
\end{align*}
\]

\[
\begin{align*}
\alpha_4 &= \langle \text{check, cases} \left( \begin{array}{c}
\text{open} \rightarrow \langle \rangle \\
\text{closed} \rightarrow \alpha_2 \\
\text{locked} \rightarrow \langle \rangle 
\end{array} \right) \rangle \\
\end{align*}
\]

In the rest of the chapter, the terms “plan” and “conditional plan” will be used alternatively.

A query of $\mathcal{AL}_{\mathcal{K}}$ has the following form

$$\varphi \text{ after } \alpha \quad (71)$$
where $\alpha$ is a *conditional plan* and $\varphi$ is a fluent formula. As an example, the following are queries of $\mathcal{AL}_K$

\[
\text{locked after } \alpha_1
\]

\[
\text{locked after } \alpha_2
\]

where $\alpha_1$ and $\alpha_2$ are defined in Example 5.3. Since a sequence of actions is also a conditional plan, the notion of $\mathcal{AL}_K$ queries is broader than the notion of $\mathcal{AL}$.

### 5.3 Approximations of $\mathcal{AL}_K$

The concepts of states, partial states, and belief states in the $\mathcal{AL}_K$ language are defined similarly as in the $\mathcal{AL}$ language. The full semantics of an $\mathcal{AL}_K$ domain description $\mathcal{D}$ can be understood as a transition diagram $T(\mathcal{D})$ between belief states\(^3\) where for a non-sensing action $a$, \(\langle S, a, S' \rangle \in \mathcal{D}\) iff \(S' = \bigcup_{s \in S} \Phi(a, s) \neq \bot\), where $\Phi$ is given in Definition 2.3 (Section 2.1.2). For a sensing action $a$ with the knowledge producing law of the form

\[
a \text{ determines } \theta
\]

\(\langle S, a, S' \rangle \in T(\mathcal{D})\) iff \(S' = \{s' \mid s' \in S, g \in s'\}\) for some $g \in \theta$. Note that as $\theta$ may contain several fluent literals, there may be more than one such $S'$. As a result, the transition diagram $T(\mathcal{D})$ is non-deterministic for sensing actions. However, for

\(^3\)Recall that a belief state is a non-empty set of states
non-sensing actions the transition diagram is deterministic.

We now introduce a family of approximations of $\mathcal{ALC}$ domain descriptions which extend the approximations presented in Chapter 5.3 to incorporate sensing actions. Recall that in Chapter 4, to define an approximation of an $\mathcal{ALC}$ domain description $\mathcal{D}'$, we first define the general concept of approximations (in terms of transition diagrams), then define the transition function for the approximation, and finally formally prove that the transition diagram described by the transition function is indeed an approximation according to our general definition of approximations. Here, we omit the first and the third steps – we only define the transition functions for the approximations. Specifically, for each $\mathcal{ALC}$ approximation $T^A(\mathcal{D})$ presented in Chapter 4, where $A$ is either $ph$, $ph+$, $pc$, or $pc+$, we extend the transition function $\Phi^A$ to a new transition function $\Phi^A_S$ which takes into account sensing actions. Unlike $\Phi^A$ which maps actions and partial states into partial state, $\Phi^A_S$ maps actions and partial states into sets of partial states, i.e., they are non-deterministic.

**Definition 5.2** Let $\mathcal{D}$ be a domain description. For any action $a$ and partial state $\delta$, $\Phi^A_S(a, \delta)$ is defined as follows.

1. If $a$ is not executable in $\delta$ then

   $$\Phi^A_S(a, \delta) = \bot$$

2. If $a$ is a non-sensing action then

   $$\Phi^A_S(a, \delta) = \begin{cases} \emptyset & \text{if } \Phi^A(a, \delta) \text{ is consistent} \\ \{ \Phi^A(a, \delta) \} & \text{otherwise} \end{cases}$$
3. If \( a \) is a sensing-action associated with the knowledge-producing law

\[ a \text{ determines } \theta \]

then

\[ \Phi^A_S(a, \delta) = \{ \text{Cl}_D(\delta \cup \{g\}) \mid g \in \theta \text{ and } \text{Cl}_D(\delta \cup \{g\}) \text{ is consistent} \} \]

Consequently, we have four different approximations of \( \mathcal{AL}_K \) domain descriptions, each of which is an extension of an approximation of \( \mathcal{AL} \) domain descriptions that was described in the previous chapter. Similarly as with \( \Phi^A \), for each action \( a \) and partial state \( \delta \), each partial state in \( \Phi^A_S(a, \delta) \) is called a possible *successor partial state* of \( \delta \) after the execution of \( a \). It is not difficult to see that without sensing actions then \( \Phi^A_S \) is the same as \( \Phi^A \) (if we ignore the set notations \( \{, \} \) and the inconsistency). For a sensing action \( a \), the execution of the action may result in several possible successor partial states, each of which corresponds to a fluent literal in \( \theta \). However, some of them might be inconsistent with the current knowledge of the agent about the domain. For example, if the security robot in Example 5.1 knows that the window is not open then after the robot *checks* the window, it should not consider the case that the window is *open* because this is inconsistent with its current knowledge. Thus, in the definition of the set of possible successor partial states resulting from the execution of a sensing action, we have excluded such inconsistent partial states. To make it more precise, let us take an example.
Example 5.4 Consider again the domain description $D_{5.2}$ from Example 5.2. Let

$$\delta_1 = \{\neg \text{open}\}$$

Then, we have

$$\text{Cl}_{D_{5.2}}(\delta_1 \cup \{\text{open}\}) = \{\text{open}, \neg \text{open}, \text{closed}, \neg \text{closed}, \text{locked}, \neg \text{locked}\} = \delta_{1,1}$$

$$\text{Cl}_{D_{5.2}}(\delta_1 \cup \{\text{closed}\}) = \{\neg \text{open}, \text{closed}, \neg \text{locked}\} = \delta_{1,2}$$

$$\text{Cl}_{D_{5.2}}(\delta_1 \cup \{\text{locked}\}) = \{\neg \text{open}, \neg \text{closed}, \text{locked}\} = \delta_{1,3}$$

Among them, $\delta_{1,1}$ is inconsistent. Therefore, we have

$$\Phi_S^A(\text{check}, \delta_1) = \{\delta_{1,2}, \delta_{1,3}\}$$

\[\Box\]

The next proposition shows that if an action (either a non-sensing or a sensing one) is executed in a valid partial state then the set of possible successor partial states will contain at least one valid partial state. This corresponds to the fact that if the agent’s knowledge about the domain is consistent with the actual state of the domain, its knowledge will remain consistent with the state of the domain after the execution of the action.

Proposition 5.1 Let $D$ be a domain description. Let $\delta$ be a valid partial state and $a$ be an action such that $\Phi_S^A(a, \delta) \neq \bot$. Then, $\Phi_S^A(a, \delta)$ contains at least one valid partial state.
Proof. See Section F.1.1.

The transition function $\Phi^A_S$ describes a transition diagram $T^A_S(D)$ with nodes being partial states and arcs labeled with non-sensing actions and sensing actions where a transition $<\delta, a, \delta'>$ belongs to $T^A_S(D)$ iff $\Phi^A_S(a, \delta) \neq \bot$ and $\delta \in \Phi^A_S(a, \delta)$. The transition diagram $T^A_S(D)$ is called an approximation of (the full semantics) $D$. Similarly to $\Phi^A$, we extend $\Phi^A_S$ to define the set of possible partial states after the execution of a plan. The extended transition function, called $\hat{\Phi}^A_S$, is given in the following definition.

**Definition 5.3** Let $D$ be a domain description. For any plan $\alpha$ and partial state $\delta$, $\hat{\Phi}^A_S(\alpha, \delta)$ is defined as follows.

1. If $\alpha = \langle \rangle$ then
   $$\hat{\Phi}^A_S(\alpha, \delta) = \{\delta\}$$

2. If $\alpha = \langle a, \beta \rangle$, where $a$ is a non-sensing action and $\beta$ is a sub-plan, then
   $$\hat{\Phi}^A_S(\alpha, \delta) = \begin{cases} \bot & \text{if } \Phi^A_S(a, \delta) = \bot \\ \hat{\Phi}^A_S(\beta, \delta') & \text{otherwise, where } \Phi^A_S(a, \delta) = \{\delta'\} \end{cases}$$

3. If $\alpha = \langle a, \text{cases}((\{g_j \rightarrow \beta_j\}_j^n)) \rangle$, where $a$ is a sensing action and $\beta_j$'s are sub-plans, then
   $$\hat{\Phi}^A_S(\alpha, \delta) = \begin{cases} \bot & \text{if } \Phi^A_S(a, \delta) = \bot \\ \bigcup_{1 \leq j \leq n, \delta' \in \Phi^A_S(a, \delta)} \Phi^A_S(\beta_j, \delta') & \text{otherwise} \end{cases}$$

where, by convention, $\cdots \cup \bot \cup \cdots = \bot$. 

194
If $\tilde{\Phi}_S^A(\alpha, \delta) = \bot$ then we say that $\alpha$ is not executable in $\delta$. Otherwise, $\alpha$ is executable in $\delta$ and each partial state in $\tilde{\Phi}_S^A(\alpha, \delta)$ is called a final partial state of $\delta$ after the execution of $\alpha$.

According to this definition, during the execution of a plan $\alpha$, when a non-sensing action $a$ is encountered (Item 2), by Definition 5.2, there are three possibilities: $\Phi_S^A(a, \delta) = \bot$, $\Phi_S^A(a, \delta) = \emptyset$, or $\Phi_S^A(a, \delta) = \{\delta'\}$ for some partial state $\delta'$. If the first case occurs then the result of execution of $\alpha$ in $\delta$ by the definition is also $\bot$, i.e., $\alpha$ is not executable in $\delta$. If the second case occurs then according to the definition, $\tilde{\Phi}_S^A(\alpha, \delta)$ is also empty. By Proposition 5.1, this case happens only if $\delta$ is invalid. When $\Phi_S^A(a, \delta) = \{\delta'\}$, then the result of the execution of $\alpha$ in $\delta$ is exactly as the result of the execution of the rest of $\alpha$ in $\delta'$.

On the other hand, if $\alpha = \langle a, \text{cases} \{g_j \rightarrow \beta_j\}_{j=1}^n \rangle$, where $a$ is a sensing action and $\beta_j$'s are sub-plans (Item 3), and $\Phi_S^A(a, \delta) \neq \bot$ then by Definitions 5.2, we know that $\Phi_S^A(a, \delta)$ may contain several partial states $\delta_j$'s. Each $\delta_j$ corresponds to a partial state in which fluent literal $g_j$ holds. Therefore, we define $\tilde{\Phi}_S^A(p, \delta)$ to be the union of the sets of possible partial states that are the results of the execution of $\beta_j$ in $\delta_j$. Note that by including $g_j$ in the partial state $\delta_j$ of $\delta$, we have made an assumption that $g_j$ holds. However, if afterwards, at some point during the execution of the rest of $\alpha$, $\beta_j$, we discover that $\tilde{\Phi}_S^A(\beta_j, \delta_j) = \emptyset$, then our assumption about $g_j$ is not correct. Therefore, such a $\delta_j$ contributes nothing to the set of possible partial states in $\tilde{\Phi}_S^A(a, \delta)$. To make it more precise, consider the following domain description...
Let us see what are the final possible partial states of

$$\delta = \{g\}$$

after the execution of plan

$$\alpha = \langle c, \text{cases}(\{ f \to \langle e \rangle, \neg f \to \langle e \rangle \}) \rangle$$

When $c$ is executed, we generate two possible successor partial states $\delta_1 = \{g, f\}$ (which is valid), and $\delta_2 = \{g, \neg f\}$ (which is invalid). Executing $e$ in $\delta_2$ results in no possible successor partial state because $Cl_{D_3}(\{g, \neg f, h\}) = \{g, \neg f, h, f\}$ is not consistent. This means that $\Phi^A_S(e, \delta_2)$, and thus $\hat{\Phi}^A_S(\langle e \rangle, \delta_2)$, become $\emptyset$. Therefore, the set of possible final partial states is $\hat{\Phi}^A_S(\alpha, \delta) = \hat{\Phi}^A_S(\langle e \rangle, \delta_1) = \{\{f, g, h\}\}$.

Note that in this example, we did not notice that $\delta_2$ is invalid at the time the action $c$ was executed; rather, this invalidity was only realized after the execution of $e$. In other words, our assumption that $\neg f$ holds was not correct and this can be noticed only after the execution of $e$. Similarly to the execution of a non-sensing action, we can show that the execution of a sensing action $a$ results in no possible successor partial state only if $\delta$ is invalid.

From these observations, we can see that, in some cases, for a plan $\alpha$ and a partial state $\delta$, $\hat{\Phi}^A_S(\alpha, \delta)$ may be empty. However, as implied by the following proposition,
which is an extension of Proposition 5.1, this case occurs only if \( \delta \) is invalid.

**Proposition 5.2** Let \( \mathcal{D} \) be a domain description. Let \( \delta \) be a valid partial state and \( \alpha \) be a conditional plan such that \( \hat{\Phi}_S^A(\alpha, \delta) \neq \perp \). Then \( \hat{\Phi}_S^A(\alpha, \delta) \) contains at least one valid partial state.

**Proof.** See Section F.1.2.

\( \square \)

It follows from this proposition that if \( \delta \) is valid then the execution of any conditional plan \( \alpha \) will yield at least a valid trajectory\(^4\), provided that \( \alpha \) is executable in \( \delta \). This is consistent with the fact that the execution of an executable plan in the real environment always ends up with some final state.

The entailment between an \( \mathcal{ALK} \) action theory and a query with respect to the approximation \( T_S^A(\mathcal{D}) \) is defined as follows (Note that this definition is similar to Definition 4.2 but given in terms of the transition function).

**Definition 5.4** Let \( (\mathcal{D}, \Delta) \) be an action theory. For a plan \( \alpha \) and a fluent formula \( \varphi \), we say that \( (\mathcal{D}, \Delta) \) entails the query \([\varphi \text{ after } \alpha]\) with respect to \( T_S^A(\mathcal{D}) \) and write

\[
(\mathcal{D}, \Delta) \models_S^A \varphi \text{ after } \alpha
\]

if for every \( \delta \in \Delta \), \( \hat{\Phi}_S^A(\alpha, \delta) \) \( \neq \perp \) and \( \varphi \) is true in \( \hat{\Phi}_S^A(\alpha, \delta) \)\(^5\).

\(^4\)A trajectory is an alternate sequence of partial states and actions, \( \delta_0\alpha_1\delta_1a_2\ldots a_n\delta_n \), such that \( \delta_i \in \Phi_S^A(a_i, \delta_{i-1}) \) for \( i = 1, \ldots, n \); A trajectory is valid if \( \delta_i \)'s are valid

\(^5\)A formula is true in a set of partial states \( \Delta \) if it is true in each partial state in \( \Delta \)
Example 5.5 For the domain description $\mathcal{D}_{5.2}$ in Example 5.2, we will show that

$$(\mathcal{D}_{5.2}, \{\{\neg \text{open}\}\}) \models \text{locked after } \alpha_2 \quad (72)$$

where $\alpha_2$ is given in Example 5.3.

Let $\delta_1 = \{\neg \text{open}\}$ and $\alpha_{2,1} = \langle \rangle$, $\alpha_{2,2} = \langle \text{flip lock} \rangle$ and $\alpha_{2,3} = \langle \rangle$. It follows from Example 5.4 that

$$\hat{\Phi}^A_S(\text{check}; \delta_1) = \{\delta_{1,2}, \delta_{1,3}\}$$

On the other hand, we can check that

$$\hat{\Phi}^A_S(\alpha_{2,2}, \delta_{1,2}) = \{\{\text{locked}, \neg \text{open}, \neg \text{closed}\}\}$$

and

$$\hat{\Phi}^A_S(\alpha_{2,3}, \delta_{1,3}) = \{\{\text{locked}, \neg \text{open}, \neg \text{closed}\}\}$$

Therefore, we have

$$\hat{\Phi}^A_S(\alpha_2, \delta_1) = \hat{\Phi}^A_S(\alpha_{2,2}, \delta_{1,2}) \cup \hat{\Phi}^A_S(\alpha_{2,3}, \delta_{1,3}) = \{\{\text{locked}, \neg \text{open}, \neg \text{closed}\}\}$$

Since $\text{locked}$ is true in $\{\text{locked}, \neg \text{open}, \neg \text{closed}\}$, we have (72) holds. Likewise, we can prove that

$$(\mathcal{D}_{5.2}, \{\{\neg \text{open}\}\}) \models \text{locked after } \alpha_3$$

$$(\mathcal{D}_{5.2}, \{\{\neg \text{open}\}\}) \models \text{locked after } \alpha_4$$
We now define the solutions of a planning instance with respect to the approximation $T^A_S(D)$.

**Definition 5.5** A plan $\alpha$ is a solution of a planning instance $\mathcal{P} = (\mathcal{D}, \Delta, \mathcal{G})$ with respect to $T^A_S(D)$ iff

$$(\mathcal{D}, \Delta) \models^A_S \mathcal{G} \text{ after } \alpha$$

According to this definition, it is easy to see that plans $\alpha_2$, $\alpha_3$, and $\alpha_4$ in Example 5.3 are solutions of $\mathcal{P}_1 = (\mathcal{D}_{5.2}, \{\neg \text{open}\}, \{\text{locked}\})$.

We will now discuss some properties of $T^A_S(D)$. For a domain description $\mathcal{D}$, we define the size of $\mathcal{D}$ to be the sum of (1) the number of fluents; (2) the number of actions; and (3) the number of propositions in $\mathcal{D}$. The size of a planning instance $\mathcal{P} = (\mathcal{D}, \Delta, \mathcal{G})$ is defined as the size of $\mathcal{D}$. The size of a plan $\alpha$, denoted by $\text{size}(\alpha)$, is defined as follows.

1. $\text{size}(\alpha) = 0$ if $\alpha = \langle \rangle$,

2. $\text{size}(\langle \alpha \rangle) = 1 + \text{size}(\beta)$ if $\alpha = \langle a, \beta \rangle$ for some non-sensing action $a$,

3. $\text{size}(\alpha) = 1 + \sum_{j=1}^{n} (1 + \text{size}(\beta_j))$ if $\alpha = \langle a, \text{cases}([g_j \rightarrow \beta_j]_{j=1}^{n}) \rangle$ for some sensing action $a$.

Then, we have the following proposition.

**Proposition 5.3** Let $\mathcal{D}$ be a domain description, $a$ be an action, and $\delta$ be a partial state. Then, computing $\Phi^A_S(a, \delta)$ can be done in polynomial time in the size of $\mathcal{D}$. 

199
Proof. See Section F.1.3.

From this proposition, we have the following corollary.

**Corollary 5.1** Determining whether or not a plan $\alpha$ is a solution of the planning instance $P = (D, \delta, G)$ with respect to $T_S^\Lambda (D)$ can be done in polynomial time in the size of $\alpha$ and $D$.

**Definition 5.6** The polynomial-length conditional planning problem with respect to $T_S^\Lambda (D)$ is defined as follows.

- Given a polynomial $Q(n) \geq n$, a domain description $D$, a partial state $\delta$, and a goal $G$,

- determine whether there exists a solution $\alpha$ of $P = (D, \delta, G)$ with $|\alpha| \leq Q(n)$ with respect to $T_S^\Lambda (D)$.

**Theorem 5.1** The polynomial-length conditional planning problem with respect to $T_S^\Lambda (D)$ is NP-complete.

Proof. See Section F.1.4.

\[ \square \]

5.4 A Logic Programming Based Conditional Planner

In this section, we describe a conditional planner, called ASCP, which is based on the $T_S^{gc} (D)$ approximation. This planner can find conditional solutions for planning
instances with a single initial partial state, i.e., planning instances of the form $\mathcal{P} = \langle D, \delta, G \rangle$. Similarly to CPASP, ASCP translates a planning instance $\mathcal{P} = \langle D, \delta, G \rangle$ into a logic program $\pi_{h,w}(\mathcal{P})$, where the parameter $h$ is similar to the parameter $h$ of the program $\pi_{h}(\mathcal{P})$ and the meaning of the parameter $w$ will become clear shortly. Then we use an answer set solver (e.g., smodels or cmodels) to compute answer sets for $\pi_{h,w}(D)$ from which solutions of $\mathcal{P}$ can be extracted. Our intuition behind this task rests on the observation that each plan $\alpha$ (Definition 5.1) corresponds to a labeled tree $T_{\alpha}$ below.

- If $\alpha = \langle \rangle$ then $T_{\alpha}$ is a tree with a single node.
- If $\alpha = \langle a \rangle$, where $a$ is a non-sensing action, then $T_{\alpha}$ is a tree with a single node labeled with $a$.
- If $\alpha = \langle a, \beta \rangle$, where $a$ is a non-sensing action and $\beta$ is a non-empty plan, then $T_{\alpha}$ is a tree whose root is labeled with $a$ and has only one subtree which is $T_{\beta}$. Furthermore, the link between $a$ and the root of $T_{\beta}$ is labeled with an empty string.
- If $\alpha = \langle a, \text{cases}(\{g_{j} \rightarrow \beta_{j}\}_{j=1}^{n}) \rangle$, where $a$ is a sensing action that determines $g_{j}$’s, then $T_{\alpha}$ is a tree whose root is labeled with $a$ and has $n$ subtrees $\{T_{\beta_{j}} \mid j \in \{1, \ldots, n\}\}$. For each $j$, the link from $a$ to the root of $T_{\beta_{j}}$ is labeled with $g_{j}$.

Observe that each trajectory of the plan $\alpha$ corresponds to a path from the root to a leave of $T_{\alpha}$. As an example, Figure 5.1 depicts the labeled trees for plans $\alpha_{1}$, $\alpha_{2}$, $\alpha_{3}$ and $\alpha_{4}$ in
Example 5.3 (black nodes indicate that there exists an action occurring at those nodes, while white nodes indicate that there is no action occurring at those nodes).

\[ T_{P_1} \quad T_{P_2} \quad T_{P_3} \quad T_{P_4} \]

Figure 5.1: Sample plan trees

For a plan \( \alpha \), let \( W \) be the number of leaves of \( T_\alpha \) and \( H \) be the number of nodes along the longest path from the root to the leaves of \( T_\alpha \). \( W \) and \( H \) are called the \textit{width} and \textit{height} of \( T_\alpha \) respectively. Let \( w \) and \( h \) be two integers such that \( w \geq W \) and \( h \geq H \).

Let us denote the leaves of \( T_\alpha \) by \( x_1, \ldots, x_W \). We map each node \( y \) of \( T_\alpha \) to a pair of integers \( n_y = (t_y, p_y) \), where \( t_y \) is the number of nodes along the path from the root to \( y \), and \( p_y \) is defined in the following way.

- For each leaf \( x_i \) of \( T_\alpha \), \( p_{x_i} \) is an arbitrary integer between 1 and \( w \). Furthermore, there exists a leaf \( x \) with \( p \)-value of 1, i.e., \( p_x = 1 \), and there exist no \( i \neq j \) such that \( p_{x_i} = p_{x_j} \).

- For each interior node \( y \) of \( T_\alpha \) with children \( y_1, \ldots, y_r \), \( p_y = \min\{p_{y_1}, \ldots, p_{y_r}\} \).
For instance, Figure 5.2 shows some possible mappings with \( h = 4 \) and \( w = 5 \) for the trees in Figure 5.1. It is easy to see that if \( w \geq W \) and \( h \geq H \) then such a mapping always exists. Furthermore, from the construction of \( T_\alpha \), independently of how the leaves of \( T_\alpha \) are numbered, we have the following properties.

1. For every node \( y \), \( t_y \leq h \) and \( p_y \leq w \).

2. For a node \( y \), all of its children have the same \( t \)-value. That is, if \( y \) has \( r \) children \( y_1, \ldots, y_r \) then \( t_{y_i} = t_{y_j} \) for every \( 1 \leq i, j \leq r \). Furthermore, the \( p \)-value of \( y \) is the smallest one among the \( p \)-values of its children.

3. The root of \( T_\alpha \) is always mapped to \((1,1)\).

Our encoding is based on the above mapping. We observe that a conditional plan \( \alpha \) can be represented on a grid \( h \times w \) where each node \( y \) of \( T_\alpha \) is placed at the position \((t_y, p_y)\) relative to the leftmost top corner of the grid. By this way, it is guaranteed that the root of \( T_\alpha \) is always placed at the leftmost top corner. Figure 5.3 depicts...
the $4 \times 5$ grid representation of conditional plans $T_{\alpha_3}$ and $T_{\alpha_4}$ in Figure 5.2. As it can be seen in Figure 5.3, each path (trajectory) of the plan can end at an arbitrary time point. For example, the leftmost and rightmost trajectories of $T_{\alpha_4}$ end at 2, whereas the others end at 3. On the other hand, to check if the plan is a solution, we need to check the satisfaction of the goal at every leaf node of the plan, that is, at the end of each trajectory. In our encoding, this task is simplified by extending all the trajectories of the plan so that they have the same height $h + 1$; furthermore, the partial states on the extended part of each trajectory are guaranteed to be the same as the one at the end node of the original trajectory. Hence, to check if the goal is satisfied at the end of a trajectory, we only need to check if it is satisfied at the end of the extended trajectory.

![Figure 5.3: Grid representation of conditional plans](image)

We now describe the program $\pi_{h,w}(P)$ in the syntax of smodels (for a concrete example of $\pi_{h,w}(P)$, see Appendix G). In $\pi_{h,w}(P)$, variables of sorts *time* and
path correspond to rows and columns of the grid. The program \( \pi_{h,w}(\mathcal{P}) \) contains the following elements.

1. **Constants.** The first two constants of \( \pi_{h,w}(\mathcal{P}) \) are \( h \) and \( w \) which serve as the input parameters for the maximum height and width of the condition plan we wish to find. In addition, we have constants to denote fluents, fluent literals and actions in the domain.

2. **Predicates.** Like \( \pi_h(\mathcal{P}) \), \( \pi_{h,w} \) uses the predicates \( \text{time}(T) \), \( \text{time}_1(T_1) \), \( \text{fluent}(F) \), \( \text{l literal}(L) \), and \( \text{contrary}(L, L_1) \), and \( \text{action}(E) \) to describe variables of sorts \( \text{time} \), \( \text{fluent} \), \( \text{literal} \), \( \text{complementary literals} \), and \( \text{elementary action} \). Additionally, it uses the following predicates.

   - \( \text{path}(\mathcal{P}) \) is true if \( 1 \leq P \leq w \).
   - \( \text{sensed}(L) \) is true if \( L \) is a sensed fluent literal.
   - \( \text{nonsensing}(NE) \) is true if \( NE \) is an elementary non-sensing action
   - \( \text{sensing}(SE) \) is true if \( SE \) is an elementary sensing action
   - \( \text{holds}(L, T, P) \) is true if fluent literal \( L \) holds at \( (T, P) \).
   - \( \text{occ}(E, T, P) \) is true if an elementary action \( E \) occurs at \( (T, P) \).
   - \( \text{ef}(L, T, P) \) is true if fluent literal \( L \) is an effect of a non-sensing action occurring at \( (T, P) \).
   - \( \text{pc}(L, T, P) \) is true if fluent literal \( L \) may change at \( (T + 1, P) \).
• \textit{goal}(T, P) is true if the goal is satisfied at \((T, P)\).

• \textit{br}(G, T, P, P_1) is true if there exists a branch from \((T, P)\) to \((T + 1, P_1)\) labeled with \(G\) in \(T_\alpha\). For example, in the grid representation of \(T_{\alpha_3}\) (Figure 5.3), we have \(\textit{br}(\text{open}, 1, 1, 1), \textit{br}(\text{closed}, 1, 1, 2),\) and \(\textit{br}(\text{locked}, 1, 1, 5)\).

• \textit{used}(T, P) is true if \((T, P)\) belongs to some extended trajectory of the plan. This allows us to know which paths are used in the construction of the plan and thus to be able to check if the plan satisfies the goal. As an example, for \(T_{\alpha_3}\) in Figure 5.3, we have \(\textit{used}(t, 1)\) for \(1 \leq t \leq 5\), and \(\textit{used}(t, 2)\) and \(\textit{used}(t, 5)\) for \(2 \leq t \leq 5\). The goal satisfaction, hence, will be checked at nodes \(\textit{used}(5, 1), \textit{used}(5, 2),\) and \(\textit{used}(5, 5)\).

3. \textbf{Variables.} The following variables are used in the program.

• \(T\) and \(T_1\): \textit{time} variables, in ranges \(1..h\) and \(1..h + 1\) respectively.

• \(P, P_1,\) and \(P_2\): \textit{path} variables, in range \(1..w\).

• \(F\): a \textit{fluent} variable.

• \(L\) and \(L_1\): fluent literal variables.

• \(G, G_1\) and \(G_2\): sensed literal variables.

• \(E\): an elementary action variable.

• \(NE\): an elementary non-sensing action variable.

• \(SE\): an elementary sensing action variable.
The domains of these variables are declared in $\pi_{h,w}(\mathcal{P})$ using the keyword \texttt{#domain} of \texttt{smodels} (see Appendix G for details).

The program $\pi_{h,w}(\mathcal{P})$ has the following facts to define variables of sort \textit{time} and \textit{path}:

\begin{align*}
time(1..h) & \leftarrow \\
time1(1..h + 1) & \leftarrow \\
path(1..w) & \leftarrow
\end{align*}

For each elementary non-sensing action $ne$, each elementary sensing action $se$, fluent $f$, or sensed-fluent literal $g$ in the domain, $\pi_{h,w}(\mathcal{P})$ contains the following fact respectively

\begin{align*}
nonsensing(ne) & \leftarrow \\
sensing(se) & \leftarrow \\
fluent(f) & \leftarrow \\
sensed(g) & \leftarrow
\end{align*}

The remaining rules of $\pi_{h,w}(\mathcal{P})$ are divided into three groups: (i) domain dependent rules; (ii) goal representation and (iii) domain independent rules, which are given next (We will use the same shortening conventions as with CP\textsc{asp} in describing the rules).
5.4.1 Domain Dependent Rules

- **Rules encoding the initial state.** For each \( l \in \delta \) (recall that \( \delta \) is the initial partial state of \( P \)), \( \pi_{h,w}(P) \) contains the fact

\[
\text{\texttt{holds}}(l, 1, 1) \leftarrow \tag{73}
\]

- **Rules encoding actions’ impossibility conditions.** For each impossibility condition

\[
\text{impossible } b \text{ if } \psi
\]

in \( D \), \( \pi_{h,w}(P) \) contains the rule

\[
\leftarrow \text{occ}(b, T, P), \neg \text{holds}(\neg\psi, T, P) \tag{74}
\]

- **Rules for reasoning about the effect of non-sensing actions.** For each dynamic causal law

\[
\text{e causes } l \text{ if } \psi
\]

in \( D \), we add to \( \pi_{h,w}(P) \) the following rules:

\[
e f(l, T, P) \leftarrow \text{occ}(e, T, P), \text{holds}(\psi, T, P) \tag{75}
\]

\[
\text{pc}(l, T, P) \leftarrow \text{occ}(e, T, P), \neg \text{holds}(l, T, P), \neg \text{holds}(\neg\psi, T, P) \tag{76}
\]

The first rule, when used along with (81), encodes what certainly holds in the successor state as the effect of the action that occurs at \((T, P)\), i.e., the set of fluent literals \( E(a, \delta) \), where \( a = \{e \mid \text{occ}(e, T, P) \text{ is true}\} \). The second rule,
when used along with (80), describes what could potentially be changed by \( a \), i.e., the set of fluent literals \( pc(a, \delta) \). These rules will be used in cooperation with (82), (86), and (87) to define the successor partial state after the execution of a non-sensing action according to the transition function \( \Phi_{s}^{pc}(a, \delta) \).

- **Rules for reasoning about the effect of sensing actions.** For each knowledge-producing law

  \[ c \text{ determines } \theta \]

  in \( D \), \( \pi_{h,w}(P) \) contains the following rules:

  \[
  \leftarrow \text{occ}(c, T, P), \not\text{br}(\theta, T, P, P)(77)
  
  1\{\text{br}(g, T, P, X):\text{new}\_\text{br}(P, X)\}1 \leftarrow \text{occ}(c, T, P) \quad (78)
  
  (g \in \theta)
  
  \leftarrow \text{occ}(c, T, P), \text{holds}(g, T, P) \quad (79)
  
  (g \in \theta)
  
  The first rule is to make sure that if a sensing action \( c \) occurs at \( (T, P) \) then there must be a branch from \( (T, P) \) to \( (T + 1, P) \). The second rule is to make sure that a new branch, corresponding to a new successor partial state, will be created for each fluent literal sensed by the action. The last rule is a constraint that prevents \( c \) from taking place if one of the fluent literals sensed by the action is already known. With this rule, the returned plan is guaranteed to be optimal in
the sense that a sensing action does not occur if one of the fluent literals sensed by
the action already holds. Observe that the semantics of $\mathcal{AL}_k$ does not prevent a
sensing action to occur when some of its sensed-fluents is known. For this reason,
some solutions of a planning instance might not be found using this encoding.
However, as we will see later in Section 5.5.2, the program will generate an
“equivalent” solution to those solutions.

- **Rules for reasoning about static causal laws.** For each static causal law

  \[ l \text{ if } \psi \]

in $\mathcal{D}$, $\pi_{h,w}(P)$ contains the rules

\[
\begin{align*}
pc(l,T,P) & \leftarrow \text{not holds}(l,T,P), pc(l',T,P), \\
& \text{not ef}(\neg\psi, T, P) \\
& (l' \in \psi)
\end{align*}
\]

(80)

\[
\begin{align*}
ef(l,T,P) & \leftarrow \ef(\psi, T, P) \\
\text{holds}(l,T_1,P) & \leftarrow \text{holds}(\psi,T_1,P)
\end{align*}
\]

(81) (82)

The first rule defines possible changes that are caused by static causal laws. The
second rule computes what certainly holds in the successor state as an indirect
effect of the action occurring at $(T, P)$. The last rule states that the successor
state must be closed under static causal laws.
5.4.2 Goal Representation

The following rules encode the goal and make sure that it is always achieved at the end of every possible branch of the plan.

\[
\text{goal}(T_1, P) \leftarrow \text{holds}(G, T_1, P) \quad (83)
\]

\[
\text{goal}(T_1, P) \leftarrow \text{holds}(L, T_1, P), \text{holds}(\neg L, T_1, P) \leftarrow \text{used}(h+1, P), \text{not} \text{goal}(h+1, P) \quad (84)
\]

The first rule says that the goal is satisfied at a node if all of its subgoals holds at that node. The last rule guarantees that if a path \( P \) is used in the construction of a plan then the goal must be satisfied at the end of this path, that is, at node \( (h + 1, P) \).

Rule (84) deserves some explanation. Intuitively, the presence of both \( \text{holds}(L, T, P) \) and \( \text{holds}(\neg L, T, P) \) indicates that the partial state at the node \( (T, P) \) is inconsistent. This means that no action should be generated at this node as inconsistent partial states will be ruled out in the extended transition function (Definition 5.3). To achieve this effect\(^6\), we say that the “goal” has been achieved at \( (T, P) \). The inclusion of this rule might raise the question: is it possible for the program to generate a plan whose execution yields inconsistent partial states only. Fortunately, due to Proposition 5.2, this is not the case.

\(^6\)The same effect can be achieved by (i) introducing a new predicate, say \( \text{stop}(T, P) \), to represent that the partial state at \( (T, P) \) is inconsistent; (ii) adding \( \text{not stop}(T, P) \) in the body of rule (94) to prevent action to occur at \( (T, P) \); and (iii) modifying the rule (85) accordingly
5.4.3 Domain Independent Rules

- Rules encoding the effect of non-sensing actions. Rules (75) – (76) specify what definitely holds and what could potentially be changed in the successor partial state as the effect of a non-sensing action. The following rules encode the effect and frame axioms for non-sensing actions.

\[
\text{holds}(L, T+1, P) \leftarrow \text{ef}(L, T, P) \tag{86}
\]

\[
\text{holds}(L, T+1, P) \leftarrow \text{holds}(L, T, P), \neg \text{pc}(\neg L, T, P) \tag{87}
\]

As aforementioned, together with (75) – (76), they define the successor partial state according to the $\Phi_{pc}^S$ function.

- Inertial rules for sensing actions. This group of rules encodes the fact that the execution of a sensing action does not change the world. However, there is one-to-one correspondence between sensed fluent literals and possible successor
partial states after the execution of a sensing action.

\[
\leftrightarrow P_1 < P_2, P_2 < P, br(G_1, T, P_1, P),
\]

\[
br(G_2, T, P_2, P) \quad \quad \quad (88)
\]

\[
\leftrightarrow P_1 \leq P, G_1 \neq G_2, br(G_1, T, P_1, P),
\]

\[
br(G_2, T, P_1, P) \quad \quad \quad (89)
\]

\[
\leftrightarrow P_1 < P, br(G, T, P_1, P), used(T, P)
\]

\[
used(T+1, P) \quad \leftrightarrow P_1 < P, br(G, T, P_1, P) \quad \quad \quad (90)
\]

\[
holds(G, T+1, P) \quad \leftrightarrow P_1 \leq P, br(G, T, P_1, P)
\]

\[
holds(L, T+1, P) \quad \leftrightarrow P_1 < P, br(G, T, P_1, P), holds(L, T, P_1)
\]

The first three rules make sure that there is no cycle in the plan that we are encoding. The next rule is to mark a node as used if there exists a branch in the plan that coming to that node. This allows us to know which paths on the grid are used in the construction of the plan and thus to be able to check if the plan satisfies the goal (see rule (85)).

The last two rules, along with rule (82), encode the possible successor partial state corresponding to the branch denoted by fluent literal \( G \) after a sensing action is performed in a state \( \delta \). They say that such partial state should contain \( G \) (rule (92)) and fluent literals that hold in \( \delta \) (rule (93)).

Note that because for each fluent literal \( G \) sensed by a sensing action \( c \), we create
a corresponding branch (rules (77) and (78)), the rules of this group guarantee that all possible successor partial states after $c$ is performed are generated.

- **Rules for generating action occurrences.**

\[
\begin{align*}
1\{\text{occ}(X, T, P) : \text{action}(X)\} & \leftarrow \text{used}(T, P), \text{not goal}(T, P) \\
& \leftarrow \text{occ}(SE, T, P), \text{occ}(E, T, P), SE \neq E
\end{align*}
\]  

(94)

The first rule enforces at least one elementary action to take place at a node that was used but the goal has not been achieved. The second rule guarantees that if a sensing action already occurs at a node then no other action (either sensing or non-sensing one) could occur at the same node.

To find sequential solutions, simply change the head of the rule to \(1\{\text{occ}(X, T, P) : \text{action}(X)\}\) and remove the second rule.
• Auxiliary Rules.

\[
\text{literal}(F) \leftarrow \quad (96)
\]

\[
\text{literal}(\neg F) \leftarrow \quad (97)
\]

\[
\text{contrary}(F, \neg F) \leftarrow \quad (98)
\]

\[
\text{contrary}(\neg F, F) \leftarrow \quad (99)
\]

\[
\text{action}(NE) \leftarrow \quad (100)
\]

\[
\text{action}(SE) \leftarrow \quad (101)
\]

\[
\text{new}_{\text{br}}(P, P_1) \leftarrow P \leq P_1 \quad (102)
\]

\[
\text{used}(1, 1) \leftarrow \quad (103)
\]

\[
\text{used}(T+1, P) \leftarrow \text{used}(T, P) \quad (104)
\]

The first four rules define literals and contrary literals. Rule (102) says that a newly created branch should outgo to a path number greater than the current path. The last two rules mark nodes that have been used.

5.5 Properties of **ASCP**

This section discusses some important properties of **ASCP**. We begin with how to extract a solution from an answer set returned by **ASCP**. Then, we argue that **ASCP** is sound and complete with respect to the \( T_{Spe}^S(D) \) semantics. We also show that **ASCP** can be used as a conformant planner. Finally, we present how to modify **ASCP** to act as a reasoner.
5.5.1 Solution Extraction

In CPASP as well as in some previous answer set based planners [37, 41, 80], reconstructing a plan from an answer set for a logic program encoding the planning instance is quite simple: we only need to collect the action occurrences in the model and then order them by the time they occur. In other words, if the answer set contains \( \text{occ}(a_1, 1), \ldots, \text{occ}(a_m, m) \) then the plan is \( a_1, \ldots, a_m \). For ASCP, it is not that simple because each answer set for \( \pi_{h,w}(\mathcal{P}) \) represents a conditional plan which may contain conditionals in the form \( \text{br}(l, t, p, p_1) \). The following procedure describes how to extract such a plan from an answer set.

Let \( \mathcal{P} = \langle \mathcal{D}, \delta, \mathcal{G} \rangle \) be a planning instance and \( S \) be an answer set for \( \pi_{h,w}(\mathcal{P}) \). Let

\[
a_{i,k} = \{ e \mid \text{occ}(e, i, k) \in S \}
\]

For any pair of integers, \( 1 \leq i \leq h + 1, 1 \leq k \leq w \), we define \( \alpha^k_i(S) \) as follows:

\[
\alpha^k_i(S) = \begin{cases} 
\langle \rangle & \text{if } i = h + 1 \text{ or } a_{i,k} = \emptyset \\
\langle a_{i,k}, \alpha^k_{i+1}(S) \rangle & \text{if } a_{i,k} \text{ is a non-sensing action} \\
\langle a_{i,k}, \text{cases}(\{g_j \rightarrow p_{j+1}^{k_j}(S)\}_{j=1}^n) \rangle & \text{if } a_{i,k} \text{ is a sensing action, and} \\
\text{br}(g_j, i, k, k_j) \in S & \text{for } 1 \leq j \leq n
\end{cases}
\]

5.5.2 Soundness and Completeness of ASCP

We have the following theorem.

**Theorem 5.2** Let \( \mathcal{P} = \langle \mathcal{D}, \delta, \mathcal{G} \rangle \) be a planning instance and \( h \geq 1 \) and \( w \geq 1 \) be integers. Let \( S \) be an answer set of \( \pi_{h,w}(\mathcal{P}) \). Then \( \alpha^1_1(S) \) is a solution of \( \mathcal{P} \) with respect to \( T^{pc}_S(\mathcal{D}) \).
Theorem 5.2 shows the soundness of $\pi_{h,w}(P)$. We will now turn our attention to the completeness of $\pi_{h,w}(P)$. Observe that solutions generated by $\pi_{h,w}(P)$ are optimal in the following senses.

1. An action (either a sensing or non-sensing one) does not occur once the goal is achieved or a possible successor partial state does not exist.

2. A sensing action does not occur if one of the fluent literals sensed by the action holds.

The first property holds because of rule (94) and the second property holds because of constraint (79). Since the definition of a conditional plan in general does not rule out non-optimal plans, obviously $\pi_{h,w}(P)$ will not generate all possible solutions of $P$.

For example, consider the planning instance $P_1$ from Examples 5.2. We have mentioned that the plans $\alpha_2$, $\alpha_3$, and $\alpha_4$ in Example 5.3 are all solutions of $P_1$. However, $\alpha_3$ and $\alpha_4$ are not optimal because they do not satisfy the above two properties.

The above example shows that $\pi_{h,w}(P)$ is not complete with respect to $T^p_{pc}(D)$ in the sense that no one-to-one correspondence between its answer sets and solutions of $P$ exists. However, we will show next that it is complete in the sense that for each solution $\alpha$ of $P$, there exist two integers $h$ and $w$ such that $\pi_{h,w}(P)$ has an answer set $S$ whose corresponding plan, $\alpha_1^1(S)$, can be obtained from $\alpha$ by applying the following
transformation (called the **reduct** operation).

**Definition 5.7** Let $\mathcal{P} = \langle \mathcal{D}, \delta, \mathcal{G} \rangle$ be a planning instance, $\alpha$ be a plan and $\delta'$ be a partial state such that $\hat{\Phi}^p_S(\alpha, \delta') \neq \bot$. The reduct of $\alpha$ with respect to $\delta'$, denoted by $\text{reduct}_{\delta'}(\alpha)$, is defined as follows.

1. If $\alpha = \langle \rangle$ or $\mathcal{G}$ is true in $\delta'$ then

$$\text{reduct}_{\delta'}(\alpha) = \langle \rangle$$

2. If $\alpha = \langle a, \beta \rangle$, where $a$ is a non-sensing action and $\beta$ is a plan, then

$$\text{reduct}_{\delta'}(\beta) = \begin{cases} 
\langle a, \text{reduct}_{\delta''}(\beta) \rangle & \text{if } \Phi^p_S(a, \delta') = \delta'' \\
\langle a \rangle & \text{otherwise}
\end{cases}$$

3. If $\alpha = \langle a, \text{cases}(\{g_j \rightarrow \beta_j\}_{j=1}^n) \rangle$, where $a$ is a sensing action that senses $g_1, \ldots, g_n$, then

$$\text{reduct}_{\delta'}(\alpha) = \begin{cases} 
\text{reduct}_{\delta'}(\beta_k) & \text{if } g_k \text{ holds in } \delta' \text{ for some } k \\
\langle a, \text{cases}(\{g_j \rightarrow \beta'_j\}_{j=1}^n) \rangle & \text{otherwise}
\end{cases}$$

where

$$\beta'_j = \begin{cases} 
\langle \rangle & \text{if } Cl_D(\delta' \cup \{g_j\}) \text{ is inconsistent} \\
\text{reduct}_{Cl_D(\delta' \cup \{g_j\})}(\beta_j) & \text{otherwise}
\end{cases}$$

**Example 5.6** Consider the planning instance $\mathcal{P}_1 = \langle \mathcal{D}_{5.2}, \{\neg \text{open}\}, \{\text{locked}\} \rangle$, and plans $\alpha_2$, $\alpha_3$, and $\alpha_4$ from Example 5.3. Let $\delta = \{\neg \text{open}\}$. We will show that

$$\text{reduct}_{\delta}(\alpha_3) = \alpha_2$$

(105)
and

\[
\text{reduct}_\delta(\alpha_4) = \alpha_2
\]  

(106)

Because \textit{open}, \textit{closed}, and \textit{locked} do not hold in \(\delta\), we have

\[
\text{reduct}_\delta(\alpha_3) = \langle \text{check}, \text{cases}(\{\text{open} \rightarrow \beta'_1, \text{closed} \rightarrow \beta'_2, \text{locked} \rightarrow \beta'_3\}) \rangle
\]

where \(\beta'_j\)'s are defined as in Definition 5.7.

Let

\[
\delta_1 = \text{Cl}_{D_2}(\delta \cup \{\text{open}\}) = \{\text{open}, \neg\text{open}, \text{closed}, \neg\text{closed}, \text{locked}, \neg\text{locked}\}
\]

\[
\delta_2 = \text{Cl}_{D_2}(\delta \cup \{\text{closed}\}) = \{-\text{open}, \text{closed}, \neg\text{locked}\}
\]

\[
\delta_3 = \text{Cl}_{D_2}(\delta \cup \{\text{locked}\}) = \{-\text{open}, \neg\text{closed}, \text{locked}\}
\]

It is easy to see that \(\beta_1 = \langle \rangle\) (because \(\delta_1\) is inconsistent) and \(\beta_3 = \langle \rangle\) (because the sub-plan corresponding to the branch \textit{locked} in \(\alpha_3\) is empty).

Let us compute \(\beta'_2\). We have

\[
\beta'_2 = \text{reduct}_{\delta_2}(\langle \text{flip\_lock, flip\_lock, flip\_lock} \rangle)
\]

Because \(\delta_2\) does not satisfy \(G\) and \(\Phi_{S}^{\text{pc}}(\text{flip\_lock}, \delta_2) = \{\delta_{2,1}\} \neq \emptyset\), where

\[
\delta_{2,1} = \{-\text{open}, \neg\text{closed}, \text{locked}\},
\]

we have

\[
\beta'_2 = \langle \text{flip\_lock, reduct}_{\delta_{2,1}}(\langle \text{flip\_lock, flip\_lock} \rangle) \rangle
\]
As $\delta_{2,1}$ satisfies $G$, we have $\text{reduct}_{\delta_{2,1}}(\langle \text{flip\_lock, flip\_lock} \rangle) = \langle \rangle$. Hence, $\beta_2 = \langle \text{flip\_lock} \rangle$. Accordingly, we have

$$\text{reduct}_\delta(\alpha_3) = \langle \text{check, cases}\{\text{open} \to \langle \rangle, \text{closed} \to \langle \text{flip\_lock} \rangle, \text{locked} \to \langle \rangle\} \rangle = \alpha_2$$

That is, (105) holds.

We now show that (106) holds. It is easy to see that

$$\text{reduct}_\delta(\alpha_4) = \langle \text{check, cases}\{\text{open} \to \langle \rangle, \text{closed} \to \text{reduct}_{\delta_2}(\alpha_2), \text{locked} \to \langle \rangle\} \rangle$$

Because $\text{closed}$ holds in $\delta_2$, we have

$$\text{reduct}_{\delta_2}(\alpha_2) = \text{reduct}_{\delta_2}(\langle \text{flip\_lock} \rangle) = \langle \text{flip\_lock} \rangle$$

Thus,

$$\text{reduct}_\delta(\alpha_4) = \langle \text{check, cases}\{\text{open} \to \langle \rangle, \text{closed} \to \langle \text{flip\_lock} \rangle, \text{locked} \to \langle \rangle\} \rangle = \alpha_2$$

As a result, we have (106) holds.

We have the following proposition.

**Proposition 5.4** Let $\mathcal{P} = \langle D, \delta, G \rangle$ be a planning instance. Then, for every solution $\alpha$ of $\mathcal{P}$ with respect to $T^\text{pe}_S(D)$, $\text{reduct}_\delta(\alpha)$ is unique and also a solution of $\mathcal{P}$ with respect to $T^\text{pe}_S(D)$. 

220
Proof. See Section F.2.2.

The following theorem shows the completeness of our planner with respect to the $T_{S}^{pc}(D)$ approximation.

**Theorem 5.3** Let $\mathcal{P} = \langle D, \delta, G \rangle$ be a planning instance, and $\alpha$ be a solution of $\mathcal{P}$. Then, there exist two integers $h$ and $w$ such that the program $\pi_{h,w}(\mathcal{P})$ has an answer set $S$ that satisfies $\alpha_{1}^{1}(S) = \text{reduct}_{\delta}^{pc}(\alpha)$.

Proof. See Section F.2.3.

5.5.3 Special Case: \textsc{ASCP} as a Conformant Planner

Consider the program $\pi_{h,w}(\mathcal{P})$ with $w = 1$ and let $S$ be an answer set for this program. As we assume that each sensing action senses at least two literals, $w = 1$ implies $S$ does not contain atoms of the form $\text{occ}(c, \ldots)$ where $c$ is a sensing action because if otherwise rules (78) and (89) cannot be satisfied. Thus, $\alpha_{1}^{1}(S)$ is a sequence of non-sensing actions. By Theorem 5.2, we know that $\alpha_{1}^{1}(S)$ achieves the goal of $\mathcal{P}$ from every possible initial partial state of the domain, which implies that $\alpha_{1}^{1}(S)$ is a conformant solution of $\mathcal{P}$. From this observation, it turns out that the planner \textsc{ASCP} can also be used as a conformant planner.
5.5.4 Special Case: ASCP as a Reasoner

It is easy to see that with minor changes, ASCP can be used to compute the consequences of a plan. This can be done as follows. Given an action theory \((\mathcal{D}, \delta)\), for any integers \(h, w\), let \(\pi_{h,w}(\mathcal{D}, \delta)\) be the set of rules: \(\pi_{h,w}(\mathcal{P}) \setminus \{(77)-(79), (83)-(85), (88)-(90), (94), (102)\}\). In other words, \(\pi_{h,w}(\mathcal{D}, \delta)\) is the program obtained from \(\pi_{h,w}(\mathcal{P})\) by removing the rules for (i) generating the branches when sensing actions are executed; (ii) checking the satisfaction of the goal; (iii) representing the constraints on branches; and (iv) generating action occurrences. For a plan \(\alpha\), let \(T_\alpha\) be the corresponding tree for \(\alpha\) that is numbered according to the principles described in the previous section. Let \(h\) and \(w\) be the height and width of \(T_\alpha\) respectively. Let \(\epsilon(\alpha)\) be the following set of atoms

\[
\{\text{occ}(e, t, p) \mid \exists \text{ a node } x \text{ in } T_\alpha \text{ labeled with action } a \text{ and numbered with } (t, p) \text{ s.t. } e \in a\} \cup \\
\{\text{br}(g, t, p, p') \mid \exists \text{ a link labeled with } g \text{ that connects the node numbered with } (t, p) \text{ to the node numbered with } (t+1, p') \text{ in } T_\alpha\}.
\]

It is easy to see that the program \(\Pi = \pi_{h,w}(\mathcal{D}, \delta) \cup \epsilon(\alpha)\) has a unique answer set which corresponds to \(\widehat{\Phi}_S^{pc}(\alpha, \delta)\). Specifically, we can check that

- \(\Pi\) has a unique answer set \(S\);

- \(\mathcal{D}, \delta \models^{pc} \varphi\) after \(\alpha\) if and only if
  
  - there exists some \(j, 1 \leq j \leq w, \delta_{h+1,j}(S) \neq \perp\); and
– for every $j$, $1 \leq j \leq w$ and $\delta_{h+1,j}(S) \neq \bot$, $\varphi$ is true in $\delta_{h+1,j}(S)$.

where

$$\delta_{t,j}(S) = \begin{cases} 
\{l \mid holds(l, t, j) \in S\} & \text{if } used(t, j) \in S \text{ and } \\
\bot & \text{is consistent} \\
\text{otherwise} 
\end{cases}$$

5.6 Evaluation

In this section, we presented our experimental results between ASCP with three other conditional planners SGP, POND, and MBP. For a comparison between ASCP and these planners on conformant benchmarks, we refer the reader to [145].

We begin with a description of the testing benchmarks, followed by the experimental results.

5.6.1 Benchmarks

First of all, we would like to note that although we attempted to encode the planning problem instances given to the systems in a uniform way (in terms of the number of actions, fluents, and effects of actions), due to the differences in the representation languages of these systems, there are situations in which the encoding of the problems might be different for each system.

The test suite consists the following sets of problems.

- **Bomb in the Toilet with Sensing Actions (BTS):** This set of examples is taken from [149]. They are variations of the BTC problem (described in Chapter 3)
that allow sensing actions to be used to determine the existence of a bomb in a specific package. There are $m$ packages and only one toilet. We can use one of the following methods to detect a bomb in a package: (1) use a metal detector (action \textit{detect\_metal}); (2) use a trained dog to sniff the bomb (action \textit{sniff}); (3) use an x-ray machine (action \textit{xray}); and, finally, (4) listen for the ticking of the bomb (action \textit{listen\_for\_ticking}).

This set of examples contains four subsets of problems, namely BTS1($m$), BTS2($m$), BTS3($m$), and BTS4($m$) respectively, where $m$ is the number of suspicious packages. These subsets differ from each other in which ones of the above methods are allowed to use. The first subset allows only one sensing action (1); the second one allows sensing actions (1)-(2); and so on.

- **Medical Problem (Med):** This set of problems is from [149]. A patient is sick and we want to find the right medication for her. Using a wrong medication may be fatal. Performing a throat culture will return either \textit{red}, \textit{blue}, or \textit{white}, which determines the group of illness the patient is infected with. Inspecting the color (that can be performed only after the throat culture is done) allows us to observe the color returned by a throat culture, depending on the illness of the patient. Analyzing a blood sample tells us whether or not the patient has a high white cell count. This can be done only after a blood sample is taken. In addition, we know that in the beginning, the patient is not dead but infected. In addition, none of the
tests have been done.

There are five problems in this set, namely, Med1, . . . , Med5. These problems are different from each other in how much we know about the illness of the patient in the beginning.

- **Sick Domain (Sick):** This set of problems is similar to Med. A patient is sick and we need to find a proper medication for her. There are $n$ kinds of illness that she may be infected with and each requiring a particular medication. Performing throat culture can return a particular color. Inspecting that color determine what kind of illness the patient has. Initially, we do not know the exact illness that the patient is infected with.

  The characteristic of this domain is that the length of the plan is fixed (only 3) but the width of the plan may be large, depending on the number of illnesses. We did experiments with five problems in the domain, namely, Sick(2), Sick(4), . . . , Sick(10). They differ from each other in the number of illnesses that the patient may have.

- **Ring (RingS):** This domain is a modification of the Ring domain. In this modified version, the agent can close a window only if it is open. It can lock a window only if it is closed. The agent can determine the status of a window by observing it (sensing action `observe_window`). Furthermore, we assume that the initial location of the agent is in the first room (This is because that the current version of
ASCP is unable to handle the domain with disjunctive knowledge about the initial state of the domain.

- **Domino (DomS):** This is a variant of the Domino domain (Chapter 4) in which some dominos may be glued to the table. Unlike the original version of the Domino domain, in this variant, when a domino falls, the next one falls only if it is not glued. The agent can do an action to unglue a glued domino. We introduce a new sensing action $\text{observe\_domino}(X)$ to determine whether a domino $X$ is glued or not.

### 5.6.2 Performance

We ran our experiments on a 2.4 GHz CPU, 768MB RAM, DELL machine, running Slackware 10.0 operating system. Time limit was set to 30 minutes. The CMU Common Lisp version 19a was used to run SGP examples. We ran ASCP examples on both cmodels and smodels. By convention, in what follows, we will use ASCP$^c$ and ASCP$^s$ to refer to the planner ASCP when it was run on cmodels and smodels respectively. Sometimes, if the distinction between the two is not important, by ASCP we mean both.

The experimental results for conformant and conditional planning are shown in Tables 5.1–5.4 respectively. Times are in seconds. “-” indicates that the corresponding planner does not return a solution within the time limit or stopped abnormally due to some reasons.
In general, ASCP is outperformed by both POND and MBP in the benchmarks, except in the last two problems of the BTS3 domain or in the last three of the BTS4 (Table 5.1), where MBP had a problem with segmentation fault or memory excess, or in the Med(5) problem (Table 5.2) where POND stopped abnormally. Both POND and MBP did very good at testing domains. POND took just a few seconds to solve each instance in the testing domains. In what follows, we discuss the performance of ASCP against the performance of SGP.

Table 5.1: Performance of ASCP on the Bomb domains

<table>
<thead>
<tr>
<th>Problem</th>
<th>Min. Plan</th>
<th>ASCP</th>
<th>SGP</th>
<th>POND</th>
<th>MBP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cmodels</td>
<td>smodels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTS1(2)</td>
<td>2x2</td>
<td>0.166</td>
<td>0.088</td>
<td>0.11</td>
<td>0.188</td>
</tr>
<tr>
<td>BTS1(4)</td>
<td>4x4</td>
<td>0.808</td>
<td>1.697</td>
<td>0.22</td>
<td>0.189</td>
</tr>
<tr>
<td>BTS1(6)</td>
<td>6x6</td>
<td>5.959</td>
<td>83.245</td>
<td>2.44</td>
<td>0.233</td>
</tr>
<tr>
<td>BTS1(8)</td>
<td>8x8</td>
<td>25.284</td>
<td>-</td>
<td>24.24</td>
<td>0.346</td>
</tr>
<tr>
<td>BTS1(10)</td>
<td>10x10</td>
<td>85.476</td>
<td>-</td>
<td>-</td>
<td>0.918</td>
</tr>
<tr>
<td>BTS2(2)</td>
<td>2x2</td>
<td>0.39</td>
<td>0.102</td>
<td>0.19</td>
<td>0.186</td>
</tr>
<tr>
<td>BTS2(4)</td>
<td>4x4</td>
<td>1.143</td>
<td>3.858</td>
<td>0.32</td>
<td>0.198</td>
</tr>
<tr>
<td>BTS2(6)</td>
<td>6x6</td>
<td>19.478</td>
<td>1515.288</td>
<td>3.23</td>
<td>0.253</td>
</tr>
<tr>
<td>BTS2(8)</td>
<td>8x8</td>
<td>245.902</td>
<td>-</td>
<td>0.452</td>
<td>109.867</td>
</tr>
<tr>
<td>BTS2(10)</td>
<td>10x10</td>
<td>345.498</td>
<td>-</td>
<td>-</td>
<td>1.627</td>
</tr>
<tr>
<td>BTS3(2)</td>
<td>2x2</td>
<td>0.357</td>
<td>0.13</td>
<td>0.22</td>
<td>0.185</td>
</tr>
<tr>
<td>BTS3(4)</td>
<td>4x4</td>
<td>1.099</td>
<td>5.329</td>
<td>0.44</td>
<td>0.195</td>
</tr>
<tr>
<td>BTS3(6)</td>
<td>6x6</td>
<td>7.055</td>
<td>-</td>
<td>3.89</td>
<td>0.258</td>
</tr>
<tr>
<td>BTS3(8)</td>
<td>8x8</td>
<td>56.246</td>
<td>-</td>
<td>28.41</td>
<td>0.549</td>
</tr>
<tr>
<td>BTS3(10)</td>
<td>10x10</td>
<td>248.171</td>
<td>-</td>
<td>-</td>
<td>2.675</td>
</tr>
<tr>
<td>BTS4(2)</td>
<td>2x2</td>
<td>0.236</td>
<td>0.149</td>
<td>0.26</td>
<td>0.194</td>
</tr>
<tr>
<td>BTS4(4)</td>
<td>4x4</td>
<td>1.696</td>
<td>3.556</td>
<td>0.64</td>
<td>0.191</td>
</tr>
<tr>
<td>BTS4(6)</td>
<td>6x6</td>
<td>13.966</td>
<td>149.723</td>
<td>4.92</td>
<td>0.264</td>
</tr>
<tr>
<td>BTS4(8)</td>
<td>8x8</td>
<td>115.28</td>
<td>-</td>
<td>30.34</td>
<td>0.708</td>
</tr>
<tr>
<td>BTS4(10)</td>
<td>10x10</td>
<td>126.439</td>
<td>-</td>
<td>-</td>
<td>4.051</td>
</tr>
</tbody>
</table>

In the BTS1–BTS4 domains (Table 5.1), SGP outperforms ASCP on the small
instances. However, $\textit{ASCP}^c$ scales up better than SGP to larger instances. As can be seen from the table, SGP could not solve any largest instance of the domains in this group within the time limit.

### Table 5.2: Performance of $\textit{ASCP}$ on the Medicate and Sick domains

<table>
<thead>
<tr>
<th>Problem</th>
<th>Min. Plan</th>
<th>$\textit{ASCP}$</th>
<th>SGP</th>
<th>POND</th>
<th>MBP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\textit{models}$</td>
<td>$\textit{smodels}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Med(1)</td>
<td>1x1</td>
<td>1.444</td>
<td>1.434</td>
<td>0.09</td>
<td>0.187</td>
</tr>
<tr>
<td>Med(2)</td>
<td>5x5</td>
<td>35.989</td>
<td>9.981</td>
<td>0.59</td>
<td>0.193</td>
</tr>
<tr>
<td>Med(3)</td>
<td>5x5</td>
<td>42.791</td>
<td>9.752</td>
<td>1.39</td>
<td>0.2</td>
</tr>
<tr>
<td>Med(4)</td>
<td>5x5</td>
<td>39.501</td>
<td>10.118</td>
<td>7.18</td>
<td>0.205</td>
</tr>
<tr>
<td>Med(5)</td>
<td>5x5</td>
<td>35.963</td>
<td>9.909</td>
<td>44.64</td>
<td>-</td>
</tr>
<tr>
<td>Sick(2)</td>
<td>3x2</td>
<td>0.234</td>
<td>0.121</td>
<td>0.21</td>
<td>0.189</td>
</tr>
<tr>
<td>Sick(4)</td>
<td>3x4</td>
<td>0.901</td>
<td>0.797</td>
<td>10.29</td>
<td>0.19</td>
</tr>
<tr>
<td>Sick(6)</td>
<td>3x6</td>
<td>5.394</td>
<td>3.9</td>
<td>-</td>
<td>0.201</td>
</tr>
<tr>
<td>Sick(8)</td>
<td>3x8</td>
<td>17.18</td>
<td>14.025</td>
<td>-</td>
<td>0.221</td>
</tr>
<tr>
<td>Sick(10)</td>
<td>3x10</td>
<td>82.179</td>
<td>43.709</td>
<td>-</td>
<td>0.261</td>
</tr>
</tbody>
</table>

In the Medicate domain (Table 5.2), both $\textit{ASCP}$ and SGP can solve all the instances and again, the performance of SGP is better than the performance of $\textit{ASCP}$ on small instances. However, it is noticeable that there is an exponential increase in the solving times of SGP when scaling up to larger instances. The running times of $\textit{ASCP}$ seem not to increase much when scaling up to larger instances. In this domain, CPA$^s$ is better than $\textit{ASCP}^c$ on all instances of this domain.

In the Sick domain, SGP could solve only the first two instances and on these problems. $\textit{ASCP}$, on the other hand, can solve all the instances. The performance of $\textit{ASCP}^s$ is slightly better than the performance of $\textit{ASCP}^c$.

Both SGP and $\textit{ASCP}$ did not perform well on the Ring domain (Table 5.3).
Table 5.3: Performance of ASCP on the Ring domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>Min. Plan</th>
<th>ASCP</th>
<th>SGP</th>
<th>POND</th>
<th>MBP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cmodels</td>
<td>smodels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RingS(1)</td>
<td>3x3</td>
<td>0.768</td>
<td>0.14</td>
<td>0.67</td>
<td>0.198</td>
</tr>
<tr>
<td>RingS(2)</td>
<td>7x9</td>
<td>-</td>
<td>1386.299</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RingS(3)</td>
<td>11x27</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.391</td>
</tr>
<tr>
<td>RingS(4)</td>
<td>15x64</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3.054</td>
</tr>
</tbody>
</table>

ASCP can solve the first two instances only, while ASCP and SGP can solve only the first instance.

Table 5.4: Performance of ASCP on the Domino domain

<table>
<thead>
<tr>
<th>Problem</th>
<th>Min. Plan</th>
<th>ASCP</th>
<th>SGP</th>
<th>POND</th>
<th>MBP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>cmodels</td>
<td>smodels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DominoS(1)</td>
<td>3x1</td>
<td>0.117</td>
<td>0.203</td>
<td>0.11</td>
<td>0.08</td>
</tr>
<tr>
<td>DominoS(2)</td>
<td>5x4</td>
<td>0.306</td>
<td>0.325</td>
<td>48.82</td>
<td>0.183</td>
</tr>
<tr>
<td>DominoS(3)</td>
<td>7x8</td>
<td>3.646</td>
<td>53.91</td>
<td>-</td>
<td>0.19</td>
</tr>
<tr>
<td>DominoS(4)</td>
<td>9x16</td>
<td>87.639</td>
<td>-</td>
<td>-</td>
<td>0.248</td>
</tr>
<tr>
<td>DominoS(5)</td>
<td>11x32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.687</td>
</tr>
</tbody>
</table>

In the Domino domain (Table 5.4), SGP did not perform well. It can solve only the first two instances. ASCP is a little bit better than SGP as it can solve the DominoS(3); ASCP can solve up to 4 dominos. The running times of ASCP and ASCP are also better than the running time of SGP on DominoS(2).

5.7 Related Work

There have been several action description languages that extended the language $A$ to account for sensing actions. Indeed, the action language $\mathcal{AL}_K$ is an extension of the action language $\mathcal{A}_K$ [126] in which causal laws and concurrent actions are allowed.
Unlike $A_K$ where sensing actions can sense only one fluent, $AL_K$ allows sensing actions to sense more than one fluent. Consequently, the notion of conditional plans in $AL_K$ is different from the notion of conditional plans in $A_K$. There is also a similarity between $AL_K$ and the action language $L_{DS}$ introduced in [14] as both languages allow for conditional effects, static causal laws and knowledge-producing laws. Nevertheless, like $A_K$, $L_{DS}$ restricts sensing actions to sense only one fluent. Apart from this, $AL_K$ can be viewed as a subset of $L_{DS}$.

Most of the existing formalisms for reasoning about sensing actions, e.g. [14, 44, 73, 117], are based on the possible world approach, which apparently differs from our work. More closely related to ours are the work by Petrick and Bacchus [111], Golden and Weld [56], Son and Baral [126], Liu and Levesque [87] which can be viewed as a kind of approximate reasoning in the presence of sensing actions. The way we handle sensing actions is similar to these work. The main difference lies in the underlying representation language and/or in the way the agent’s knowledge is updated when a non-sensing action is executed (see Section 4.10 for more information relating our work to these work).

Approaches to conditional planning can be characterized by the techniques employed in their search process or by the formalism that supports their reasoning process. Most of the early conditional planners, for example, XII by Golden, Etzioni, and Weld [55], Cassandra by Pryor and Collins [112], CNLP [110] Peot and Smith, or CoPlaS by Lobo [88], extended the partial order planning algorithm [92, 109, 116] to handle
sensing actions. Some other conditional planners translate the conditional planning problem into another problem in another setting and appeal to off-the-shell solvers to find solutions. Along this line of research were the constraint logic programming based conditional planner FLUX by Thielscher [137], and the QBF theorem prover based conditional planner by Ritannen [115]. In [149], Weld, Anderson and Smith presented a conditional planner called SGP which extends the planning graph algorithm [17] to deal with sensing actions. A majority of recent conditional planners, for example GPT by Bonet and Geffner [20], POND by Bryce, Kambhampati, and Smith [25], MBP by Bertoli, Cimatti, and Roveri [16], or Contingent-FF by Hoffmann and Brafman [62], search for a solution in the belief state space, using Binary Decision Diagrams (BDD) or CNF formulas to compactly represent the search space and exploiting a heuristic function to guide the search. To the best of our knowledge, no answer set based conditional planner was developed before ASCP.

5.8 Summary

In this chapter, we introduce an action language, called $\mathcal{AL}_K$, for specifying sensing actions and then define the concepts of conditional plans and queries of $\mathcal{AL}_K$. We then present how to extend the approximations presented in Chapter 4 to incorporate sensing actions. The results are four different approximations of $\mathcal{AL}_K$ domain descriptions. To evaluate the approximations, we develop a conditional planner in logic programming paradigm, called ASCP which is based on one of these approximations.
The experimental results show that on some benchmarks, ASCP is competitive with the planner SGP.
CHAPTER 6

CONCLUSION

This dissertation is developed along the line of research in reasoning and planning with incomplete information.

In the first part of the dissertation (Chapter 3), we present a sufficient condition (Section 3.2) for the completeness of the 0-approximation, suggest a method that extends this framework to achieve an efficient method for complete reasoning for domain descriptions without static causal laws (Section 3.3), and study its application in conformant planning (Section 3.4).

The sufficient condition for the completeness of the 0-approximation is based on a dependency relationship $\triangleright$ between two fluent literals or between an action and a fluent literal. Intuitively, for fluent literals $l$ and $g$, and an action $a$, $l \triangleright g$ means that the value of $l$ in a state may depend on the value of $g$ in that state or in previous states; $a \triangleright g$ means that the executability of $a$ depends on the value of fluent literal $g$. We then define a property between a belief state $S$ and a partial state $\delta$ with respect to a fluent formula $\varphi$, called reducibility and denoted by $S \gg_{\varphi} \delta$. With this property it is guaranteed that $S$ and $\delta$ agree with each other on answering the truth value of $\varphi$. Furthermore, we prove that the reducibility is preserved during the execution course of actions. More precisely, if $S \gg_{\varphi} \delta$ and $S'$ and $\delta'$ are the final belief state of $S$ and the final partial
state of $\delta$ after the execution of a sequence of actions $\alpha$ then we have $S' \gg \varphi \delta'$. The completeness of the 0-approximation with respect to the possible world semantics on answering a fluent formula is then defined based on this property.

From these results, we suggest a new method that allows for “complete reasoning with the 0-approximation”. This method is based on the notion of a decisive set for a partial state with respect to a fluent formula. Intuitively, decisive sets are sets of unknown fluents that can be used to group possible initial states of the domain into partial states such that the 0-approximation semantics of the modified one and the possible world semantics of the original one agree with each other on the truth value of the fluent formula. We then devise a polynomial time algorithm for computing a decisive set for a partial state with respect to a fluent formula.

This result is applied in the development of a sound and complete conformant planner, called CPA$^+$. The experiments show that, even built with a simple heuristic, the planner can be competitive with other state-of-the-art conformant planners on many benchmarks in the literature.

The second part of the dissertation (Chapter 4) focuses on extending the 0-approximation to reason about domains with static causal laws. Specifically, we introduce two different approaches to the problem, one of which is called the possibly-holds approach (Section 4.3) and the other the possibly-changes approach (Section 4.4). In the former, a successor partial state of a state after the execution of an action is determined based on specifying what would possibly hold in a possible successor state. In
the latter, a successor partial state is determined through identifying possible changes that may be directly caused by the action or indirectly caused by a static causal law. The outcome is four different approximations which can be computed very efficiently. In particular, for a given action and a partial state, a successor partial state can be computed in polynomial time in the size of the domain. As a result, the polynomial-length planning problem with respect to the proposed approximations is NP-complete. We also present our initial study on the completeness of the approximations.

To evaluate the performance of the proposed approximations, we build two conformant planners, one of which, called CPASP (Section 4.6), is implemented in logic programming paradigm and the other of which, called CPA (Section 4.7), is a forward, best-first search planner. The experiments show that CPASP is good on concurrent benchmarks, whereas CPA is good on sequential benchmarks. We also discuss how to extend CPASP to handle disjunctive information about the initial state of the domain.

In the last part of the dissertation (Chapter 5), we extend the approximations in the second part to incorporate sensing actions. To this end, we first develop a language, called \(\mathcal{AL}_K\), to handle sensing actions and then defined the concepts of conditional plans and queries of \(\mathcal{AL}_K\) domain descriptions (Section 5.2). The approximations of \(\mathcal{AL}_K\) domain descriptions are presented in Section 5.3. We also show that the polynomial-length conditional planning problem with respect to these new approximations is NP-complete.

We then build a logic programming based conditional planner, ASCP that imple-
ments one of the proposed approximations and proved the correctness of the implementation (Section 5.4). ASCP is capable of generating both conformant and conditional plans, as well as sequential and concurrent plans.

The work in this dissertation can be extended in many directions. First, the sufficient condition of the completeness of 0-approximation can be strengthened in one way or another. This may be done by extending the dependency relationship ◁ to fluent formulas. Second, the approximations of \( \mathcal{AL} \) and \( \mathcal{AL}_K \) domain descriptions are not ultimate ones. In fact, we can combine the two approaches, “possibly-holds” and “possibly-changes”, to create a stronger approximation without loosing their computational efficiency. Furthermore, we would like to extend the approximations to handle other features of dynamic domains, e.g., multi-valued fluents, continuous change, etc. As an initial step toward this direction, we already extend \( \mathcal{AL}_K \) to support fluents with multi-valued and develop a full semantics for this language in [132, 133].

Finally, we would like to extend the planners to cope with domain knowledge, or user’s preferences on plans. Indeed, we already build a planner with preferences in constraint logic programming [146]. However, this planner is limited to domains with complete information about the initial state. In the future, we would like to extend it to handle incomplete information.

The heuristic used in CPA^+ and CPA is very simple and in some cases, it is not effective. Hence, at present, we are developing a more informative heuristic function for domains with static causal laws. We are also investigating the application of parallel
programming to improve the performance of the planners. Preliminary results toward this direction have been reported in [134].
This appendix contains the description of the planning systems that were mentioned in the dissertation.

A.1 Conformant-FF (CFF)

CFF [23], to our best knowledge, is one of the current fastest conformant planners in most of the benchmark domains in the literature. It extends the classical FF planner [61] to deal with uncertainty in the initial state. The basic idea is to represent a belief state $s$ just by the initial belief state (which is described as a CNF formula) together with the action sequence that leads to $s$. In addition, the reasoning is done by checking the satisfiability of CNF formulae.

The input language of CFF is a subset of PDDL with a minor change that allows the users to specify the initial state as a CNF formula. Both sequential and conformant planning are supported in CFF. However, it does not support concurrent and conditional planning.

A.2 CMBP (Conformant Model Based Planner)

CMBP [32, 33] is a conformant planner developed by Cimatti and Roveri. A planning domain in CMBP is represented as a finite state automaton. BDD (Binary
Decision Diagram) techniques are employed to represent and search the automaton. CMBP allows nondeterministic domains with uncertainty in both the initial state and action effects. Nevertheless, it does not have the capability of generating concurrent and conditional plans. The input language of CMBP is $AR$ described in [53]. The version used for testing was downloaded from http://www.cs.washington.edu/research/jair/contents/v13.html.

**A.3 C-PLAN**

$C$-PLAN [30] is a SAT-based conformant planner, which works based on a generate-and-test method: candidates for solutions are generated and then verified. The input language is the action language $C$ [54]. $C$-PLAN is designated for generating concurrent plans.

**A.4 DLV$^K$**

$DLV^K$ is a declarative, logic-based planning system built on top of the DLV system (http://www.dbai.tuwien.ac.at/proj/dlv/). The input language $K$ is a logic-based planning language described in [41]. The version we used for testing is available at http://www.dbai.tuwien.ac.at/proj/dlv/K/. $DLV^K$ is capable of generating both concurrent and conformant plans. It, however, does not support sensing actions and cannot generate conditional plans.

239
A.5 KACMBP

Similarly to CMBP, KACMBP [34] uses techniques from symbolic model checking to search in the belief space. However, in KACMBP, the search is guided by a heuristic function which is derived based on knowledge associated with a belief state.

KACMBP is designated for sequential and conformant setting. It, however, does not support concurrent planning and conditional planning. The input language of KACMBP is SMV. The system was downloaded from http://sra.itc.it/tools/mbp/AIJ04/.

A.6 MBP

MBP [16] is a previous version of CMBP. Unlike CMBP which only deals with conformant planning, MBP supports conditional planning as well. The version used for testing was downloaded from http://sra.itc.it/tools/mbp/.

A.7 POND

POND [25] extends the planning graph algorithm [17] to deal with sensing actions. Conformant planning is also supported as a feature of POND. The input language is a subset of PDDL. The version used for testing was downloaded from http://rakaposhi.eas.asu.edu/belief-search/.
A.8 SGP (Sensory Graph Plan)

SGP [1, 149] is a planner based on the planning graph algorithm proposed by Blum and Furst in [17]. SGP supports conditional effects, universal and existential quantification. It also handles uncertainty and sensing actions. SGP has the capability of generating both conformant and conditional plans, as well as concurrent plans. Nevertheless, static laws are not allowed in SGP. The input syntax is PDDL (Planning Domain Definition Language). The version used for testing is 1.0h (dated January 14th, 2000), written in Lisp, available at http://www.cs.washington.edu/ai/sgp.html.
APPENDIX B

PROOFS OF THE RESULTS IN CHAPTER 2

B.1 Proof of Proposition 2.1

1. “⇒”: Suppose \( s' \) is a possible successor state of \( s \). By definition, we have

\[
s' = \text{Cl}_D(\text{de}(a, s) \cup (s \cap s')) = \text{de}(a, s) \cup (s \cap s')
\]  

(107)

as \( D \) contains no static causal laws. We need to show that

\[
s' = \text{de}(a, s) \cup (s \setminus \neg\text{de}(a, s))
\]

(108)

Let \( \sigma \) denote the right hand side of (108). First of all, notice that \( \text{de}(a, s) \) is consistent because it is a subset of state \( s' \).

Let \( l \) be an arbitrary fluent literal in \( s' \). By (107), there are two cases.

(a) \( l \in \text{de}(a, s) \). Clearly, we have \( l \in \sigma \).

(b) \( l \in s \cap s' \). This implies that

\[
l \in s
\]

(109)

If \( l \in \neg\text{de}(a, s) \) then we have \( \neg l \in \text{de}(a, s) \subseteq s' \). This is a contradiction because \( s' \) is consistent, i.e., it cannot contain both \( l \) and \( \neg l \).
Hence, we have

\[ l \not\in \neg de(a, s) \]  

(110)

By (109) and (110) we have \( l \in \sigma \).

We have showed that in either case, \( l \in s' \) implies \( l \in \sigma \). Hence, we have

\[ s' \subseteq \sigma \]

On the other hand, as \( de(a, s) \) is consistent and \( s \) is consistent, we have \( \sigma \) is consistent. Observe that the number of elements in a consistent set of fluent literals cannot exceed the number of elements in a state. As a result, it follows from \( s' \subseteq \sigma \) that (108) holds.

2. “\( \Leftarrow \)”: let

\[ s' = de(a, s) \cup (s \setminus \neg de(a, s)) \]

We need to show that it is a possible successor state of \( s \).

Because \( D \) is consistent and \( a \) is executable in \( s \), there exists at least one possible successor of \( s \), say \( s'' \). From the previous result, we have \( s'' = de(a, s) \cup (s \setminus \neg de(a, s)) = s' \), which implies that \( s' \) is a possible successor state of \( s \).

B.2 Proof of Proposition 2.2

Suppose \( a \) is executable in \( \delta \). If \( a \) is not executable in \( s \) then there exists an impossibility condition (5) such that \( b \subseteq a \) and \( \psi \) holds in \( s \). Because \( \delta \subseteq s \), this implies that \( \psi \)
possibly holds in $\delta$. By definition, this implies that $a$ is not executable in $\delta$, which is a contradiction. Hence, $a$ is executable in $s$.

By the definition of $\Phi$ and Proposition 2.1, it follows that $\Phi(a, s) = \{s'\}$ where

$$s' = de(a, s) \cup (s \setminus \neg de(a, s))$$

To complete the proof, we only need to show that

$$\Phi^0(a, \delta) \subseteq s'$$

Observe that

$$de(a, \delta) \subseteq de(a, s)$$

On the other hand, because what holds in $s$ definitely possibly holds in $\delta$, we have

$$de(a, s) \subseteq pe(a, \delta)$$

Hence, we have

$$\neg de(a, s) \subseteq \neg pe(a, \delta)$$

Accordingly, we have

$$de(a, \delta) \cup (\delta \setminus \neg pe(a, \delta)) \subseteq de(a, s) \cup (s \setminus \neg de(a, s)) = s'$$

That is,

$$\Phi^0(a, \delta) \subseteq s'$$

244
B.3 Proof of Theorem 2.1

Let \( \alpha = \langle a_1, \ldots, a_n \rangle \) be a plan and \( \varphi \) be a query such that \( (D, \Delta) \models^0 \varphi \).

Let \( s \) be an arbitrary state in \( ext(\Delta) \) and let \( \delta \in \Delta \) be a partial state such that \( s \in ext(\delta) \). Define a sequence of partial states \( \langle \delta_i \rangle_{i=1}^{n+1} \) as follows:

\[
\delta_i = \begin{cases} 
\delta & \text{if } i = 1 \\
\Phi^0(a_{i-1}, \delta_{i-1}) & \text{otherwise}
\end{cases}
\]

Because \( (D, \Delta) \models \varphi \) after \( \alpha \), such sequence of partial states exists and is unique.

Furthermore, we have \( a_i \) is executable in \( \delta_i \) for \( 1 \leq i \leq n \), and

\[ \delta_{n+1}(\varphi) = true \]

We now show that for \( 1 \leq i \leq n + 1 \), there exists a state \( s' \) such that

\[ \widehat{\Phi}(\alpha[i], s) = \{ s' \} \text{ and } \delta_i \subseteq s' \]  \hspace{1cm} (111)

(recall that \( \alpha[i] \) denotes the plan consisting of the \( i \) initial actions of \( \alpha \)).

1. Base case: \( i = 1 \). Clearly, we have \( \widehat{\Phi}(\langle \rangle, s) = \{ s \} \) and \( \delta_1 \subseteq s \).

2. Inductive step: Suppose (111) is true for \( i \leq k \leq n \), we need to show that it is also true for \( i = k + 1 \).

By the inductive hypothesis, there exists a state \( s'' \) such that

\[ \widehat{\Phi}(\alpha[k], s) = \{ s'' \} \text{ and } \delta_k \subseteq s'' \]
Because $a_k$ is executable in $\delta_k$, by Proposition 2.2, it follows that there exists a state $s'$ such that

$$\Phi(a_k, s'') = \{s'\} \text{ and } \delta_{k+1} \subseteq s'$$

As $\tilde{\Phi}(\alpha[k], s) = \{s''\} \neq \bot$, by the definition of $\tilde{\Phi}$, we have

$$\tilde{\Phi}(\alpha[k + 1], s) = \bigcup_{s_1 \in \hat{\Phi}(\alpha[k], s)} \Phi(a_k, s_1) = \Phi(a_k, s'') = \{s'\}$$

This allows us to conclude the inductive step.

We have proved that (111) holds. This means that for every $s \in ext(\Delta)$, there exists a state $s'$ such that

$$\hat{\Phi}(\alpha, s) = \{s'\} \text{ and } \delta_{n+1} \subseteq s'$$

Because $\varphi$ is true in $\delta_{n+1}$, it is also true in $s'$.

On the other hand, it is easy to see that

$$\tilde{\Phi}^P(\alpha, ext(\Delta)) = \bigcup_{s \in ext(\Delta)} \tilde{\Phi}(\alpha, s)$$

Consequently, we have that $\varphi$ holds in $\tilde{\Phi}^P(\alpha, ext(\Delta))$. By the definition of $\tilde{\Phi}^P$, it follows that

$$(D, \Delta) \models^P \varphi \text{ after } \alpha$$
APPENDIX C

PROOFS OF THE RESULTS IN CHAPTER 3

This section lists the proofs of theorems and propositions in the chapter. Some
of the proofs will use the results from the following lemmas.

Lemma C.1  Let \( D \) be a domain description, \( s \) be a state, \( \delta \) be a partial state, and \( \psi \) be
a set of fluent literals. Then, the following hold.

1. If \( \psi \) holds in \( \delta \) then \( \neg \psi \cap \delta = \emptyset \).
2. If \( \psi \) does not hold in \( s \) then \( \neg \psi \cap s \neq \emptyset \).

Proof.

1. Suppose \( \psi \) holds in \( \delta \). By definition, we have \( \psi \subseteq \delta \). Because \( \delta \) is consistent,
   this implies that \( \neg \psi \cap \delta = \emptyset \).

2. Suppose \( \psi \) does not hold in \( s \), i.e., \( \psi \nsubseteq s \). We have \( \psi \subseteq L = s \cup \neg s \). Therefore,
it follows that \( \psi \cap \neg s \neq \emptyset \). That is, \( \neg \psi \cap s \neq \emptyset \).

Lemma C.2  Let \( D \) be a domain description without static causal laws. Let \( s \) be a
state, \( \delta \subseteq s \) be a partial state, and \( a \) be an action that is executable in \( \delta \). Let \( s' \) be
the successor state of \( s \) (see Proposition 2.2) and \( \delta' \) be the successor partial state of \( \delta \).
Then, for each fluent literal \( l \in s' \setminus \delta' \), there exists a fluent literal \( l_1 \) in \( s \setminus \delta \) such that \( l \triangleleft l_1 \).

**Proof.** Let \( l \) be a fluent literal in \( s' \setminus \delta' \). If \( l \in s \setminus \delta \) then we can take \( l_1 = l \) to have \( l \triangleleft l_1 \), which implies that the lemma holds. Suppose otherwise, i.e., \( l \not\in s \setminus \delta \). There are two possible cases:

- \( l \not\in s \), or
- \( l \in s \) and \( l \in \delta \).

Let us consider each in turn.

1. \( l \not\in s \). Because \( l \in s' \), it follows that \( l \in de(a, s) \) (see Proposition 2.1), i.e., there must be a dynamic causal law

   \[
   e \text{ causes } l \text{ if } \psi
   \]

   such that \( e \in a \) and \( \psi \) holds in \( s \), i.e., (i) \( \psi \subseteq s \). On the other hand, \( l \in s' \setminus \delta' \) implies \( l \not\in \delta' \). By definition of \( \delta' \), we have \( l \not\in de(a, \delta) \). Hence, \( \psi \) does not hold in \( \delta \), i.e., (ii) \( \psi \not\subseteq \delta \). By (i) and (ii), there exists a fluent literal \( l_1 \) such that \( l_1 \in \psi \) and \( l_1 \in s \setminus \delta \). By the definition of dependencies, we have \( l \triangleleft l_1 \in (s \setminus \delta) \). Hence, the lemma holds.

2. \( l \in s \) and \( l \in \delta \). Because \( l \in s' \) but \( l \not\in \delta' \), there must be a dynamic causal law

   \[
   e \text{ causes } \neg l \text{ if } \psi
   \]

   248
such that $e \in a$, $\psi$ does not hold in $s$, and $\psi$ possibly holds in $\delta$. By Lemma C.1, it follows that $\neg \psi \cap s \neq \emptyset$ and $\neg \psi \cap \delta = \emptyset$. Hence, there exists $l_1 \in \neg \psi$ such that $l_1 \in s \setminus \delta$. By the definition of dependencies, we have $\neg l \triangleright \neg l_1$ and thus $l \triangleright l_1$.

\[ \square \]

C.1 Proofs of the Results in Section 3.2

C.1.1 Proof of Proposition 3.1

By the definition of $\triangleright_\varphi$, we have that $\delta$ is a subset of every state $s$ in $S$. Hence, it is easy to see that if $\varphi$ is true in $\delta$ then it is also true in every $s \in S$, i.e., it is true in $S$. We now prove that if $\varphi$ is true in $S$ then it is also true in $\delta$.

Let $\varphi = \gamma_1 \land \gamma_2 \land \cdots \land \gamma_n$. Suppose that $\varphi$ is not true in $\delta$. This implies that there exists $1 \leq i \leq n$ such that $\gamma_i$ is not true in $\delta$. Let $\gamma_i = l_1 \lor l_2 \lor \cdots \lor l_k$. Then, as $\gamma_i$ is not true in $\delta$, for every $1 \leq j \leq k$, $l_j$ is not true in $\delta$, i.e.,

\[ l_j \notin \delta \quad (112) \]

Consider an arbitrary state $s \in S$. As $\varphi$ is true in $s$, we have that $\gamma_i$ is true in $s$. As a result, there exists at least one $1 \leq j \leq k$ such that $l_j$ is true in $s$, i.e.,

\[ l_j \in s \quad (113) \]

By (112) and (113), we have

\[ l_j \in s \setminus \delta \]

249
which implies that $\gamma_i \triangleleft (s \setminus \delta)$. By Definition 3.4, $S \nRightarrow_\varphi \delta$ (Condition 2 is not satisfied), which is a contradiction. As a result, we have $\varphi$ is true in $\delta$.

C.1.2 Proof of Proposition 3.2

Assume that $a$ is executable in $S$.

1. We first show that $a$ is executable in $\delta$. Suppose otherwise. This means that there is an impossibility condition

$$\text{impossible } b \text{ if } \psi$$

in $\mathcal{D}$ such that $b \subseteq a$ and $\neg \psi \cap \delta = \emptyset$.

Let $s$ be an arbitrary state in $S$. As $a$ is executable in $s$, $\psi$ does not hold in $s$, which implies that $\neg \psi \cap s \neq \emptyset$ (see Lemma C.1). Hence, there is a fluent literal $l$ such that $l \in \neg \psi$ and $l \in s$. As $\neg \psi \cap \delta = \emptyset$, it follows that $l \in s \setminus \delta$. On the other hand, as $l \in \neg \psi$ and $b \subseteq a$, by the definition of dependencies, we have $a \triangleleft l$ and thus $a \triangleleft (s \setminus \delta)$. Hence, we have $S \nRightarrow_\varphi \delta$, which is a contradiction.

2. Let $\varphi = \gamma_1 \land \cdots \land \gamma_n$. Let $S' = \Phi^P(a, S)$ and $\delta' = \Phi^0(a, \delta)$. We need to show that

$$S' \nRightarrow_\varphi \delta'$$

Suppose otherwise, that is, $S' \nRightarrow_\varphi \delta'$. According to Definition 3.4, there are three possible cases:
(a) \( \delta' \) is not a subset of every state in \( S' \). Let \( s' \) be a state in \( S' \) such that \( \delta' \not\subseteq s' \)
and let \( s \in S \) be the previous state of \( s' \), i.e., \( \Phi(a, s) = \{s'\} \) (such \( s \) always
exists, by Proposition 2.2 and by the definition of \( S' \)). As \( S \gg_{\varphi} \delta \), we have
\( \delta \subseteq s \). By Proposition 2.2, it follows that \( \delta' \subseteq s' \). This is a contradiction.

(b) there exists an integer \( 1 \leq i \leq n \) such that \( \gamma_i \prec s' \setminus \delta' \) for every \( s' \in S' \).
Consider an arbitrary state \( s \in S \) and let \( \Phi(a, s) = \{s'\} \) (by Proposition
2.2, such \( s' \) exists). Since \( \gamma_i \prec s' \setminus \delta' \), there exists \( l_1 \in s' \setminus \delta' \) such that \( \gamma_i \prec l_1 \).
By Lemma C.2, there must be a fluent literal \( l_2 \in s \setminus \delta \) such that \( l_1 \prec l_2 \). It
is easy to see that \( \gamma_i \prec l_2 \). This implies that \( \gamma_i \prec s \setminus \delta \). Since \( s \) is an arbitrary
state in \( S \), it follows that \( S \gg_{\varphi} \delta \). This is a contradiction because \( S \gg_{\varphi} \delta \).

(c) there exists an action \( b \) such that \( b \prec (s' \setminus \delta') \) for every \( s' \in S' \). Let \( b \) be such
an action. Consider an arbitrary state \( s \in S \) and let \( \Phi(a, s) = \{s'\} \). Since
\( b \prec s' \setminus \delta' \), there exists a fluent literal \( l \in s' \setminus \delta' \) such that \( b \prec l \). By Lemma
C.2, there exists \( l_1 \in s \setminus \delta \) such that \( l \prec l_1 \). It follows from the definition of
dependencies that \( b \prec l_1 \in s \setminus \delta \). That is, \( b \prec s \setminus \delta \). Since \( s \) is an arbitrary
state in \( S \), this implies that \( b \prec s \setminus \delta \) for every \( s \in S \). This is a contradiction
because \( S \gg_{\varphi} \delta \).

C.1.3 Proof of Proposition 3.3

Let \( \alpha = \langle a_1, \ldots, a_n \rangle \) and assume that \( \alpha \) is executable in \( S \). For \( 1 \leq i \leq n + 1 \),
let \( S_i = \hat{\Phi}^P(\alpha[i - 1], S) \). By this definition, we have \( S_1 = S \) and \( S_{n+1} = \hat{\Phi}^P(\alpha, S) \).
First of all, we will show that

\[ \widehat{\Phi}^0(\alpha[k], \delta) \neq \bot \quad (114) \]

\[ S_{k+1} \gg_{\varphi} \widehat{\Phi}^0(\alpha[k], \delta) \quad (115) \]

for \(0 \leq k \leq n\) by induction on \(k\).

- **Base case:** \(k = 0\). Clearly, \(\widehat{\Phi}^0(\alpha[0], \delta) = \widehat{\Phi}^0(\langle \rangle, \delta) = \delta \neq \bot\). Hence (114) holds.

As \(S \gg_{\varphi} \delta, S_1 = \widehat{\Phi}^P(\alpha[0], S) = S\), and \(\widehat{\Phi}^0(\alpha[0], \delta) = \delta\), we have that (115) holds.

- **Inductive Step:** Suppose (114) & (115) hold for \(k < i\) (1 \(\leq i \leq n\), we need to show that they also hold for \(k = i\). Let \(\delta' = \widehat{\Phi}^0(\alpha[i-1], \delta)\). By the inductive hypothesis, we have \(\delta' \neq \bot\) and \(S_i \gg_{\varphi} \delta'\).

As \(\widehat{\Phi}^P(\alpha, S) \neq \bot\), \(a_i\) is executable in \(S_i\). By Proposition 3.2, it follows that

(i) \(a_i\) is executable in \(\delta'\)

(ii) \(\Phi^P(a_i, S_i) \gg_{\varphi} \Phi^0(a_i, \delta')\)

From (i), it follows that \(\widehat{\Phi}^0(\alpha[i], \delta) \neq \bot\), and from (ii), it follows that \(\widehat{\Phi}^P(\alpha[i], S) \gg_{\varphi} \widehat{\Phi}^0(\alpha[i], \delta)\). Hence, the inductive hypothesis holds.

The proposition follows immediately from results (114) and (115) as \(\alpha[n] = \alpha\).
C.1.4 Proof of Theorem 3.1

By Definition 3.1, to prove the theorem, we need to show that for any sequence of actions $\alpha$

$$(D, \Delta) \models^0 \varphi \text{ after } \alpha \iff (D, \Delta) \models^P \varphi \text{ after } \alpha$$

1. “$\Rightarrow$”: follows from Theorem 2.1.

2. “$\Leftarrow$”. Suppose

$$(D, \Delta) \models^P \varphi \text{ after } \alpha \tag{116}$$

we need to show that

$$(D, \Delta) \models^0 \varphi \text{ after } \alpha \tag{117}$$

where $\alpha$ is a sequence of actions.

Let $S = ext(\Delta)$. By the definition of the $\models^P$, it follows from (116) that $\hat{\Phi}^P(\alpha, S) \neq \bot$ and $\varphi$ is true in $\hat{\Phi}^P(\alpha, S)$.

Consider a partial state $\delta \in \Delta$ and let $S' = \hat{\Phi}^P(\alpha, ext(\delta))$. Since $\hat{\Phi}^P(\alpha, S) \neq \bot$ and $\varphi$ is true in $\hat{\Phi}^P(\alpha, S)$, we have that $S' \neq \bot$ and $\varphi$ is true in $S'$. On the other hand, as $ext(\delta) \gg_{\varphi} \delta$, by Proposition 3.3, we have $\delta' = \hat{\Phi}^0(\alpha, \delta) \neq \bot$ and $S' \gg_{\varphi} \delta'$. By Proposition 3.1, it follows that $\varphi$ is true in $\delta'$.

So, for every $\delta \in \Delta$, we have $\delta' = \hat{\Phi}^0(\alpha, \delta) \neq \bot$ and $\varphi$ is true in $\delta'$. By the definition of $\models^0$, this means that (117) holds.
C.2 Proofs of the Results in Section 3.3

C.2.1 Proof of Theorem 3.2

From the construction of $\Delta^*$ and by the definition of decisive sets, it follows that for each $\delta^* \in \Delta^*$, we have $\text{ext}(\delta^*) \gg \varphi \delta^*$. Hence, by Theorem 3.1, we have

$$(D, \Delta^*)_{\models^0} \simeq_{\varphi} (D, \Delta)_{\models^P}$$

This implies that for any sequence of actions $\alpha$, we have

$$(D, \Delta^*) \models^0 \varphi \text{ after } \alpha \iff (D, \Delta) \models^P \varphi \text{ after } \alpha \quad (118)$$

On the other hand, observe that because $D$ does not contain static causal law, we have $\text{ext}(\Delta_{FA}) = \text{ext}(\delta)$. Hence,

$$\text{ext}(\Delta^*) = \text{ext}(\bigcup_{\delta \in \Delta} \Delta_{FA}^\delta) = \bigcup_{\delta \in \Delta} \text{ext}(\Delta_{FA}^\delta) = \bigcup_{\delta \in \Delta} \text{ext}(\delta) = \text{ext}(\Delta)$$

This implies that for any sequence of actions $\alpha$ and fluent formula $\varphi$, we have

$$(D, \Delta^*) \models^P \varphi \text{ after } \alpha \iff (D, \Delta) \models^P \varphi \text{ after } \alpha \quad (119)$$

From (118) and (119), it follows that for any sequence of actions $\alpha$, we have

$$(D, \Delta^*) \models^0 \varphi \text{ after } \alpha \iff (D, \Delta) \models^P \varphi \text{ after } \alpha$$

That is,

$$(D, \Delta^*)_{\models^0} \simeq_{\varphi} (D, \Delta)_{\models^P}$$

254
C.2.2 Proof of Proposition 3.4

Let $F$ be the set of fluents returned by the algorithm. First, notice that $F$ contains only fluents unknown in $\delta$. Second, we will prove that for every interpretation $I$ of $F$, $\text{ext}(\delta^*) \not\gg_{\phi} \delta^*$, where $\delta^* = \delta \cup I$. Suppose otherwise, i.e., $\text{ext}(\delta^*) \gg_{\phi} \delta^*$. There are two possibilities

1. there exists a conjunct $\gamma \in \varphi$ such that for every state $s \in \text{ext}(\delta^*)$, $\gamma \not\prec s \setminus \delta^*$. Let $G$ be the set of fluents unknown in $\delta^*$. According to the algorithm, there exists no $f \in G$ such that $\gamma$ depends on both $f$ and $\neg f$. As a result, for each $f \in G$, either

   - $\gamma \prec f$ but $\gamma \not\not\prec \neg f$,
   - $\gamma \not\not\prec f$ but $\gamma \prec \neg f$, or
   - $\gamma \not\not\prec f$ and $\gamma \not\not\prec \neg f$.

and these cases are mutually exclusive.

Let

$$\gamma_1 = \{ \neg f \mid f \in G, \gamma \prec f \text{ and } \gamma \not\not\prec \neg f \}$$

$$\gamma_2 = \{ f \mid f \in G, \gamma \not\not\prec f \text{ and } \gamma \prec \neg f \}$$

$$\gamma_3 = \{ f \mid f \in G, \gamma \not\not\prec f \text{ and } \gamma \not\not\prec \neg f \}$$

and let $J = \gamma_1 \cup \gamma_2 \cup \gamma_3$. Then we have the following results.
• \( J \) is consistent.

• For each \( f \in G \), either \( f \) or \( \neg f \) belong to \( J \).

• \( l \) does not depend on any literal in \( J \).

From the first two results, we can derive that \( J \) is an interpretation of \( G \). Hence, \( s = \delta^* \cup J \) is a state in \( \text{ext}(\delta^*) \). From the last result, we have \( \gamma \not\triangleright s \setminus \delta^* \). This is a contradiction.

2. there exists an action \( a \) such that for every state \( s \in \text{ext}(\delta^*) \), \( a \triangleleft s \setminus \delta^* \). Similarly to the above case, we can construct a state \( s \in \text{ext}(\delta^*) \) such that \( a \) does not depend on \( s \setminus \delta^* \) and thus it is a contradiction.

Therefore, the proposition holds.

C.2.3 Proof of Proposition 3.5

In the first two steps, we can use the Warshall’s algorithm to compute dependencies, which takes polynomial time in the size of the domain. The for loop runs in polynomial time as well. As a result, the algorithm runs in polynomial time.
APPENDIX D

PROOFS OF THE RESULTS IN CHAPTER 4

This appendix contains the proofs of the results presented in Chapter 4. Some of them will make use of the following notations and lemmas.

Given a domain description $\mathcal{D}$, for a set of fluent literals $\sigma$, let

$$\Lambda(\sigma) = \sigma \cup \{l \mid \exists \psi \in \mathcal{D} \text{ such that } \psi \subseteq \sigma\} \quad (120)$$

Let $\Lambda^0(\sigma) = \sigma$ and $\Lambda^{i+1}(\sigma) = \Lambda(\Lambda^i(\sigma))$ for $i \geq 0$. Since, by the definition of $\Lambda$, for any set of fluent literals $\sigma'$ we have $\sigma' \subseteq \Lambda(\sigma')$, the sequence $\langle \Lambda^i(\sigma) \rangle_{i=0}^\infty$ is monotonic with respect to the set inclusion operation. In addition, $\langle \Lambda^i(\sigma) \rangle_{i=0}^\infty$ is bounded by the set of fluent literals $L$. Thus, there exists a set of fluent literals $\sigma^{\text{limit}}$ such that

$$\sigma_{D}^{\text{limit}} = \bigcup_{i=0}^\infty \Lambda^i(\sigma) \quad (121)$$

Furthermore, $\sigma_{D}^{\text{limit}}$ is unique and is closed under the set of static causal laws in $\mathcal{D}$.

**Lemma D.1** *For any set of fluent literals $\sigma$, we have $\sigma_{D}^{\text{limit}} = \text{Cl}_{\mathcal{D}}(\sigma)$.*

**Proof.** By induction we can easily show that $\Lambda^i(\sigma) \subseteq \text{Cl}_{\mathcal{D}}(\sigma)$ for all $i \geq 0$. Hence, we have

$$\sigma_{D}^{\text{limit}} \subseteq \text{Cl}_{\mathcal{D}}(\sigma)$$

Furthermore, from the construction of $\Lambda^i(\sigma)$, it follows that $\sigma^{\text{limit}}$ is closed under the set of static causal laws in $\mathcal{D}$. Because of the minimality property of $\text{Cl}_{\mathcal{D}}(\sigma)$, we have

$$\text{Cl}_{\mathcal{D}}(\sigma) \subseteq \sigma_{D}^{\text{limit}}$$
Accordingly, we have
\[
\sigma_D^{\text{limit}} = \text{Cl}_D(\sigma)
\]

\[\square\]

**Corollary D.1** For two sets of literals \(\sigma \subseteq \sigma', \text{Cl}_D(\sigma) \subseteq \text{Cl}_D(\sigma')\).

**Proof.** Follows immediately from the above lemma.

\[\square\]

**Lemma D.2** Let \(\sigma\) and \(\gamma\) be two sets of fluent literals. Then,
\[
\text{Cl}_D(\gamma \cup \sigma) = \text{Cl}_D(\text{Cl}_D(\gamma) \cup \sigma)
\]

**Proof.** It follows from Corollary D.1 that the left hand side of the equation in the lemma is always a subset of the right hand side. Therefore, to prove this lemma, we only need to show that
\[
\text{Cl}_D(\text{Cl}_D(\gamma) \cup \sigma) \subseteq \text{Cl}_D(\gamma \cup \sigma)
\]

By Corollary D.1, as \(\gamma \subseteq \gamma \cup \sigma\) we have
\[
\text{Cl}_D(\gamma) \subseteq \text{Cl}_D(\gamma \cup \sigma)
\]

Furthermore, we have
\[
\sigma \subseteq \text{Cl}_D(\gamma \cup \sigma)
\]

Hence, we have
\[
\text{Cl}_D(\gamma) \cup \sigma \subseteq \text{Cl}_D(\gamma \cup \sigma)
\]
Accordingly, by Corollary D.1, we have

\[ Cl_D(Cl_D(\gamma) \cup \sigma) \subseteq Cl_D(Cl_D(\gamma \cup \sigma)) = Cl_D(\gamma \cup \sigma) \]

Proof done.

D.1 Proofs of the Results in Section 4.2

D.1.1 Proof of Theorem 4.1

Let \( \alpha = \langle a_1, \ldots, a_n \rangle \) be a plan and \( \varphi \) be a fluent formula such that

\[(D, \Delta) \models_{\tilde{T}(D)} \varphi \text{ after } \alpha\]

By Definition 4.2, this implies that for each \( \delta \in \Delta \), there is a partial state \( \delta' \) such that \( \langle \delta, \alpha, \delta' \rangle \in \tilde{T}(D) \) and \( \varphi \) is true in \( \delta' \). As \( \langle \delta, \alpha, \delta' \rangle \in \tilde{T}(D) \), there is a sequence of partial states \( \langle \delta_i \rangle_{i=1}^{n+1} \) such that (i) \( \delta_1 = \delta \), (ii) for \( 1 \leq i \leq n \), \( \langle \delta_i, a_i, \delta_{i+1} \rangle \in \tilde{T}(D) \), and (iii) \( \delta_{n+1} = \delta' \).

Let us show by induction on \( i \) (\( 1 \leq i \leq n + 1 \)) that for each \( \delta \in \Delta \),

\[ S_i = \hat{\Phi}^P(\alpha[i-1], ext(\delta)) \neq \perp \] (122)

\[ \delta_i \subseteq \bigcap_{s \in S_i} s \] (123)

1. **Base case:** \( i = 1 \). In this case, by definition, we have

\[ S_1 = \hat{\Phi}^P(\alpha[0], ext(\delta)) = \hat{\Phi}^P(\langle \rangle, ext(\delta)) = ext(\delta) \]

Therefore, (122) holds. Furthermore, we have \( \delta_1 = \delta \subseteq \bigcap_{s \in ext(\delta)} s = \bigcap_{s \in S_i} s \) and thus (123) holds.
2. **Inductive Step:** Suppose (122) and (123) hold for \( i \leq k \) (1 \( \leq k \leq n \)). We need to show that they also hold for \( i = k + 1 \), i.e.,

\[
S_{k+1} \neq \bot
\]

\[
\delta_{k+1} \subseteq \bigcap_{s \in S_{k+1}} s
\]

Consider an arbitrary state \( s \in S_k \). By the inductive hypothesis, we have \( \delta_k \subseteq s \). On the other hand, as \( \langle \delta_k, a_k, \delta_{k+1} \rangle \in \tilde{T}(D) \), by the definition of an approximation, it follows that \( a_k \) is executable in \( s \). Since \( s \) is an arbitrary state in \( S_k \), it implies that \( s \) is executable in \( S_k \).

As \( a_k \) is executable in \( S_k \), by the definition of \( \Phi^P \) (Definition 2.8), we have

\[
\Phi^P(a_k, S_k) = \bigcup_{s \in S_k} \Phi(a, s) = \{ s' \mid \langle s, a_k, s' \rangle \in T(D) \text{ for some } s \in S_k \}
\]

Accordingly, by the definition of \( \hat{\Phi}^P \) (Definition 2.8) we have

\[
S_{k+1} = \hat{\Phi}^P(\alpha[k], \text{ext}(\delta)) = \Phi^P(a_k, \hat{\Phi}^P(\alpha[k-1], \text{ext}(\delta))) = \Phi^P(a_k, S_k) = \{ s' \mid \langle s, a_k, s' \rangle \in T(D) \text{ for some } s \in S_k \}
\]

So, we have

\[
S_{k+1} \neq \bot
\]

In addition, because \( \langle \delta_k, a_k, \delta_{k+1} \rangle \in \tilde{T}(D) \) and \( \delta_k \) is a subset of every state \( s \in S_k \), by the definition of the approximation, we have

\[
\delta_{k+1} \subseteq \bigcap_{s \in S_{k+1}} s
\]
As a result, we can conclude the inductive step.

We have showed that (122) and (123) hold. As a result we have for each \( \delta \in \Delta \), \( \widehat{\Phi}^P(\alpha, \text{ext}(\delta)) \neq \bot \) and \( \delta_{n+1} \subseteq s \) for each \( s \in \widehat{\Phi}^P(\alpha, \text{ext}(\delta)) \). Since \( \varphi \) is true in \( \delta_{k+1} \), it also true in \( \widehat{\Phi}^P(\alpha, \text{ext}(\delta)) \). On the other hand observe that

\[
\widehat{\Phi}^P(\alpha, \text{ext}(\Delta)) = \bigcup_{\delta \in \Delta} \widehat{\Phi}^P(\alpha, \text{ext}(\delta))
\]

Hence, \( \varphi \) is also true in \( \widehat{\Phi}^P(\alpha, \text{ext}(\Delta)) \). By the definition of \( \models^P \), this implies that

\[(D, \Delta) \models^P \varphi \text{ after } \alpha \]

Proof done.

D.2 Proofs of the Results in Section 4.3

D.2.1 Proof of Lemma 4.1

Consider a state \( s' \in \Phi(a, s) \). Let \( \sigma = de(a, s) \cup (s \cap s') \). First of all, we will show that for \( i \geq 0 \), we have

\[
\Lambda^i(\sigma) \subseteq ph^i(a, \delta)
\]

by induction on \( i \) (where that the operator \( \Lambda \) is defined by (120)).

1. **Base case: \( i = 0 \).** We need to show that

\[
\sigma \subseteq ph^0(a, \delta) = (pe(a, \delta) \cup \{l \mid \neg l \not\in \delta\}) \setminus \neg E(a, \delta)
\]

As \( E(a, \delta) \subseteq s' \) we have

\[
\neg E(a, \delta) \cap s' = \emptyset
\]
Furthermore, as \( \sigma \subseteq s' \), it follows that
\[
\neg E(a, \delta) \cap \sigma = \emptyset
\]

Therefore, to prove (125), it suffices to show that
\[
\sigma \subseteq pe(a, \delta) \cup \{l \mid \neg l \notin \delta\}
\]

As \( \delta \subseteq s \), whatever holds in \( s \) certainly possibly holds in \( \delta \), i.e., \( s \subseteq \{l \mid \neg l \notin \delta\} \).

Furthermore, we know that \( de(a, s) \subseteq pe(a, \delta) \). Hence, we have
\[
\sigma = de(a, s) \cup (s \cap s') \subseteq de(a, s) \cup s \subseteq pe(a, \delta) \cup \{l \mid \neg l \notin \delta\}
\]

Therefore, the base case is true.

2. **Inductive step:** Suppose (124) holds for \( i \leq k \). We need to show that it also holds for \( i = k + 1 \).

Consider a fluent literal \( l \) in \( \Lambda^{k+1}(\sigma) \). We need to show that \( l \in ph^{k+1}(a, \delta) \).

If \( l \in \Lambda^k(\sigma) \) then by the inductive hypothesis we have \( l \in ph^k(a, \delta) \subseteq ph^{k+1}(a, \delta) \).

Now assume that \( l \notin \Lambda^k(\sigma) \). By the definition of \( \Lambda \), it follows that there exists a static causal law

\[ l \text{ if } \psi \]

in \( \mathcal{D} \) such that \( \psi \subseteq \Lambda^k(\sigma) \). By the inductive hypothesis, it follows that
\[
\psi \subseteq ph^k(a, \delta)
\]

(126)
As $\psi \subseteq \Lambda^k(\sigma) \subseteq \text{Cl}_D(\sigma) = s'$ and $E(a, \delta) \subseteq s'$, $\psi$ possibly holds in $E(a, \delta)$, i.e.,

$$-\psi \cap E(a, \delta) = \emptyset \quad (127)$$

Furthermore, as $l \in \Lambda^{k+1}(\sigma) \subseteq s'$ and $E(a, \delta) \subseteq s'$, $l$ possibly holds in $E(a, \delta)$, i.e.,

$$l \not\in -E(a, \delta) \quad (128)$$

From (126)–(128) and by the definition of $ph^{k+1}(a, \delta)$, we can derive that $l \in ph^{k+1}(a, \delta)$.

We have showed that (124) holds. Accordingly, we have

$$s' = \text{Cl}_D(\sigma) = \sigma_D^{\text{limit}} = \bigcup_{i=0}^{\infty} \Lambda^i(\sigma) \subseteq \bigcup_{i=0}^{\infty} ph^i(a, \delta) = ph(a, \delta)$$

(Note that the second equality is due to Lemma D.1). Therefore, the lemma holds.

D.2.2 Proof of Theorem 4.2

Let $\langle \delta, a, \delta' \rangle$ be a transition in $T^{ph}(D)$. By the definition of $T^{ph}(D)$ we have

1. $a$ is executable in $\delta$,

2. $\delta' = \Phi^{ph}(a, \delta)$

Let $s$ be a state in $\text{ext}(\delta)$ and $s'$ be a state in $\Phi(a, s)$. As $\delta \subseteq s$, $a$ is also executable in $s$. Therefore, to prove that $T^{ph}(D)$ is an approximation of $D$, it suffices to show that $\Phi^{ph}(a, \delta) \subseteq s'$. 

263
By Lemma 4.1, we have

\[ s' \subseteq ph(a, \delta) \]

Hence,

\[ \{ l \mid l \notin \neg ph(a, \delta) \} \subseteq \{ l \mid l \notin \neg s' \} = s' \]  
(note that \( s' \) is consistent and complete)

According to Corollary D.1, we have

\[ \Phi^{ph}(a, \delta) = Cl_\mathcal{D}(\{ l \mid l \notin \neg ph(a, \delta) \}) \subseteq Cl_\mathcal{D}(s') = s' \]

Proof done.

D.2.3 Proof of Proposition 4.1

It is easy to see that the function \( CLOSURE(\mathcal{D}, \sigma) \) is a straightforward computation of \( \sigma_\mathcal{D}^{lim} \) (Equations (120) and (121)). Hence, by Lemma D.1, we have \( CLOSURE(\mathcal{D}, \sigma) = \sigma_\mathcal{D}^{lim} = Cl_\mathcal{D}(\sigma) \).

Back to the proposition. Because \( CLOSURE(\mathcal{D}, \sigma) \) correctly implements \( Cl_\mathcal{D}(\sigma) \), it is easy to see that the variables \( E \) and \( pe \) after Line 9 take the values \( E(a, \delta) \) and \( pe(a, \delta) \) respectively. Hence, the value of the variable \( ph \) after Line 10 is \( ph^0(a, \delta) \).

Furthermore, it is not difficult to show that the value of \( ph \) after the \textbf{repeat} loop (Lines 11–16) is the same as \( ph(a, \delta) \). As a result, the value returned by the algorithm (Line 17) is \( \Phi^{ph}(a, \delta) \) (see Definition 4.5). That is, the proposition holds.
D.2.4 Proof of Lemma 4.3

Consider an arbitrary state \( s \in \text{ext}(\delta) \) and a state \( s' \in \Phi(a, s) \). Since \( \delta \) is valid and \( \mathcal{D} \) is consistent, such \( s \) and \( s' \) always exist.

First of all, let us show that for any partial state \( \delta' \subseteq s' \), we have

\[
\delta' \subseteq \{ l \mid l \notin \neg \text{ph}(a, \delta, \delta') \}
\]

(129)

From the construction of \( \text{ph}(a, \delta, \delta') \) we have that \( \neg \delta' \cap \neg \text{ph}(a, \delta, \delta') = \emptyset \), i.e.,

\[
\delta' \cap \neg \text{ph}(a, \delta, \delta') = \emptyset
\]

Therefore for every fluent literal \( l \), if \( l \in \delta' \) then \( l \notin \neg \text{ph}(a, \delta, \delta') \), which implies that \( l \) belongs to the right side of the set inclusion in (129). This means that (129) holds.

It is easy to see that to prove the lemma, it suffices to show that for \( i \geq 0 \)

\[
\delta_i^{ph} \subseteq \delta_{i+1}^{ph} \subseteq s'
\]

(130)

1. **Base case**: \( i = 0 \). Then, we have

\[
\delta_0^{ph} = E(a, \delta)
\]

On the other hand, by Observation (4.2), \( E(a, \delta) \subseteq s' \). Hence, by (129), we have that

\[
\delta_0^{ph} \subseteq \{ l \mid l \notin \neg \text{ph}(a, \delta, \delta_0^{ph}) \} \subseteq \text{Cl}_\mathcal{D}(\{ l \mid l \notin \neg \text{ph}(a, \delta, \delta_0^{ph}) \}) = \delta_1^{ph}
\]

That is, the first set inclusion “\( \subseteq \)” of (130) holds.
The second set inclusion of (130) holds because

\[ \delta_{ph}^i = \Phi_{ph}^i(a, \delta) \subseteq s' \quad \text{(by Theorem 4.2)} \]

The base case is thus true.

2. **Inductive step:** Suppose (130) holds for \( i < k \) (\( k > 0 \)). We need to show that it also holds for \( i = k \). By the inductive hypothesis, we have

\[ \delta_{ph}^k \subseteq s' \]

and thus by (129), we have

\[ \delta_{ph}^k \subseteq \{l \mid l \notin \neg ph(a, \delta, \delta_{ph}^k)\} \subseteq Cl_D(\{l \mid l \notin \neg ph(a, \delta, \delta_{ph}^k)\}) = \delta_{ph}^{k+1} \] \hspace{1cm} (131)

and by Lemma 4.2 we have

\[ s' \subseteq ph(a, \delta, \delta_{ph}^k) \]

The latter implies that

\[ \{l \mid l \notin \neg ph(a, \delta, \delta_{ph}^k)\} \subseteq \{l \mid l \notin \neg s'\} = s' \]

Hence, by Corollary (D.1) we have

\[ \delta_{ph}^{k+1} = Cl_D(\{l \mid l \notin \neg ph(a, \delta, \delta_{ph}^k)\}) \subseteq Cl_D(s') = s' \] \hspace{1cm} (132)

From (131) and (132) we can conclude the inductive step.
D.3 Proofs of the Results in Section 4.4

D.3.1 Proof of Lemma 4.4

Let $\sigma$ denote $\text{de}(a, s) \cup (s \cap s')$. We first show that, for every $i \geq 1$, (see (120) for the definition of $\Lambda$):

$$\Lambda^i(\sigma) \setminus \Lambda^{i-1}(\sigma) \subseteq \text{pc}^i(a, \delta)$$ (133)

by induction on $i$.

1. **Base case:** $i = 1$. Let $l$ be a fluent literal in $\Lambda^1(\sigma) \setminus \Lambda^0(\sigma)$. We need to prove that $l \in \text{pc}^1(a, \delta)$.

   By the definition of $\Lambda$, it follows that

   $$l \notin \Lambda^0(\sigma) = \sigma$$ (134)

   $$l \in \Lambda^1(\sigma) \subseteq s'$$ (135)

   and, in addition, there exists a static causal law

   $$l \text{ if } \psi$$

   in $D$ such that

   $$\psi \subseteq \Lambda^0(\sigma) = \sigma$$ (136)

   By (134), we have $l \notin (s \cap s')$. By (135), we have $l \in s'$. Accordingly, we have $l \notin s$. On the other hand, because $\delta \subseteq s$, we have

   $$l \notin \delta$$ (137)
Furthermore, by (136) we have $\psi \subseteq s'$ since $\sigma \subseteq s'$. Because of the completeness of $s'$, if follows that $\neg \psi \cap s' = \emptyset$. As $E(a, \delta) \subseteq s'$ (Observation 4.2), this implies that

$$\neg \psi \cap E(a, \delta) = \emptyset$$

(138)

We now show that $\psi \not\subseteq s$. Suppose otherwise, that is, $\psi \subseteq s$. Because $s$ satisfies all static causal laws in $\mathcal{D}$, we have $l \in s$. By (135), it follows that $l \in (s \cap s') \subseteq \sigma$ and this is a contradiction to (134). Hence, we have

$$\psi \not\subseteq s$$

(139)

On the other hand, by (136), we have

$$\psi \subseteq \sigma = de(a, s) \cup (s \cap s') \subseteq de(a, s) \cup s$$

(140)

It follows from (139) and (140) that there exists a fluent literal $l' \in \psi$ such that $l' \in de(a, s)$ but $l' \not\in s$. This implies

$$\psi \cap (de(a, s) \setminus s) \neq \emptyset$$

In addition, it is easy to see that $de(a, s) \setminus s \subseteq de(a, s) \setminus \delta \subseteq pe(a, \delta) \setminus \delta = pc^0(a, \delta)$. Therefore, we have

$$\psi \cap pc^0(a, \delta) \neq \emptyset$$

(141)

From (137), (138), and (141), and by the definition of $pc^1(a, \delta)$, we can conclude that $l \in pc^1(a, \delta)$. The base case is thus true.
2. **Inductive Step:** Assume that (133) is true for all \( i \leq k \). We need to prove that it is also true for \( i = k + 1 \). Let \( l \) be a fluent literal in \( \Lambda^{k+1}(\sigma) \setminus \Lambda^{k}(\sigma) \). We will show that \( l \in pc^{k+1}(a, \delta) \).

By the definition of \( \Lambda \), there exists a static causal law

\[
\textbf{if } \psi
\]

in \( \mathcal{D} \) such that

\[
\psi \subseteq \Lambda^{k}(\sigma) \subseteq s'
\]  

(142)

Because \( \psi \subseteq s' \), we have \( -\psi \cap s' = \emptyset \). In addition, we know that \( E(a, \delta) \) is a subset of \( s' \). As a result, we have

\[
-\psi \cap E(a, \delta) = \emptyset
\]  

(143)

It is easy to see that \( \psi \nsubseteq \Lambda^{k-1}(\sigma) \) for if otherwise then, by the definition of \( \Lambda \), \( l \) must be in \( \Lambda^{k}(\sigma) \), which contradicts to \( l \in \Lambda^{k+1}(\sigma) \setminus \Lambda^{k}(\sigma) \). As \( \psi \nsubseteq \Lambda^{k}(\sigma) \), it follows that there exists \( l' \in \psi \) such that \( l' \in \Lambda^{k}(\sigma) \setminus \Lambda^{k-1}(\sigma) \). By the inductive hypothesis, we have \( l' \in \Lambda^{k}(\sigma) \setminus \Lambda^{k-1}(\sigma) \subseteq pc^{k}(a, \delta) \), which implies that

\[
\psi \cap pc^{k}(a, \delta) \neq \emptyset
\]  

(144)

Because \( l \nsubseteq \Lambda^{k}(\sigma) \), we have \( l \nsubseteq \sigma \). As a result, \( l \nsubseteq (s \cap s') \). On the other hand, since \( l \in \Lambda^{k+1}(\sigma) \subseteq s' \), it follows that \( l \nsubseteq s \). Thus, we have

\[
l \nsubseteq \delta
\]  

(145)
From (143) – (145), and by the definition of $pc^{k+1}(a, \delta)$, it follows that $l \in pc^{k+1}(a, \delta)$. So the inductive step is proved.

We have showed that (133) holds. Hence, we have

\[
\Lambda^i(\sigma) \setminus \sigma = \bigcup_{j=1}^{i}(\Lambda^j(\sigma) \setminus \Lambda^{j-1}(\sigma)) \subseteq \bigcup_{j=1}^{i}pc^j(\sigma, \delta) = pc^i(\sigma, \delta)
\]

(note that the second equality holds because $pc^1(\sigma, \delta) \subseteq pc^2(\sigma, \delta) \subseteq \cdots \subseteq pc^i(\sigma, \delta)$).

As a result, we have

\[
\bigcup_{i=0}^{\infty}(\Lambda^i(\sigma) \setminus \sigma) \subseteq \bigcup_{i=0}^{\infty}pc^i(\sigma, \delta)
\]

By Lemma D.1 and by the definition of $pc(\sigma, \delta)$, this implies that

\[(s' \setminus \sigma) \subseteq pc(\sigma, \delta).
\]

The lemma is thus true.

D.3.2 Proof of Theorem 4.4

Let $\langle \delta, a, \delta' \rangle$ be a transition in $T^{pc}(D)$. By the definition of $T^{pc}(D)$ we have

1. $a$ is executable in $\delta$,

2. $\delta' = \Phi^{pc}(a, \delta) = Cl_D(E(a, \delta) \cup (\delta \setminus \neg pc(a, \delta)))$

Let $s$ be a state in $ext(\delta)$ and $s'$ be a state in $\Phi(a, s)$. By the definition of $\Phi$, we have

\[s' = Cl_D(de(a, s) \cup (s \cap s'))\]
As $\delta \subseteq s$, $a$ is also executable in $s$. Therefore, to prove that $T^{pc}(D)$ is an approximation of $D$, it suffices to show that $\Phi^{pc}(a, \delta) \subseteq s'$.

As $E(a, s) = Cl_D(de(a, s))$, it follows from Lemma D.2 that

$$s' = Cl_D(E(a, s) \cup (s \cap s'))$$

Since $E(a, \delta) \subseteq E(a, s)$, to prove $\delta' \subseteq s'$, we only need to show that

$$\delta \setminus \neg pc(a, \delta) \subseteq s \cap s'$$

Let $l$ be an arbitrary fluent literal in the left hand side. We need to show that $l \in s \cap s'$.

Since $l \in \delta \setminus \neg pc(a, \delta)$, we have $l \in \delta \subseteq s$ and $l \not\in \neg pc(a, \delta)$. The latter implies that $\neg l \not\in pc(a, \delta)$. By Lemma 4.4, it follows that

$$\neg l \not\in s' \setminus (de(a, s) \cup (s \cap s'))$$  \hspace{1cm} (146)

If $\neg l \not\in s'$ then because $s'$ is complete, we have $l \in s'$ and thus the proof is done. Now consider the case that $\neg l \in s'$. By (146), it follows that

$$\neg l \in de(a, s) \cup (s \cap s')$$  \hspace{1cm} (147)

Since $l \in s$, we have $\neg l \not\in s$. As a result,

$$\neg l \not\in (s \cap s')$$

Together with (147), this means that $\neg l \in de(a, s)$. On the other hand, because $\neg l \not\in pc(a, \delta)$ and $pc^0(a, \delta) \subseteq pc(a, \delta)$, we have

$$\neg l \not\in pc^0(a, \delta) = pc(a, \delta) \setminus \delta$$
Furthermore it is easy to see that

$$-l \notin \delta$$

as $l \in \delta$.

Accordingly, we have

$$-l \notin pe(a, \delta)$$

which contradicts to

$$-l \in de(a, s)$$

as $de(a, s) \subseteq pe(a, \delta)$. Hence, this case never occurs.

Proof done.

**D.3.3 Proof of Proposition 4.4**

It is easy to see that at Line 9 of the algorithm, the value of the variable $pc$ is $pc^0(a, \delta)$ and the value of the variable $E$ is $E(a, \delta)$. After the repeat loop (Line 15), the variable $pc$ takes the value $pc(a, \delta)$. As a result, the value returned by the algorithm is $\Phi^pc(a, \delta)$. Hence, the proposition holds.

**D.3.4 Proof of Lemma 4.6**

By the definition of $\delta^pc_i$, it is easy to see that the sequence $(\delta^pc_i)_{i=0}^\infty$ is monotonically non-decreasing. Therefore to prove the lemma, it suffices to show that for $i \geq 0$ and

$$\delta^pc_i \subseteq s'$$

(148)
Let us prove this by induction on $i$.

1. **Base case:** $i = 0$. Trivial holds because

   $$
   \delta_{0}^{pc} = E(a, \delta) \subseteq s'
   $$

2. **Inductive step:** Suppose (148) holds for $i \leq k$. We need to show that it holds for $i = k + 1$, i.e.,

   $$
   \delta_{k+1}^{pc} \subseteq s'
   $$

   The proof of this result is very similar to the proof of

   $$
   \Phi^{pc}(a, \delta) \subseteq s'
   $$

   in Theorem 4.4 (Section D.3.2) except that $E(a, s)$ is replaced with $\delta_{k}^{pc}$, $pc(a, \delta)$ is replaced with $pc(a, \delta, \delta_{k}^{pc})$ and observe that by the inductive hypothesis we have $\delta_{k}^{pc} \subseteq s'$.

### D.4 Proofs of Results in Section 4.5

#### D.4.1 Proof of Theorem 4.6

Clearly to prove the theorem, it is sufficient to show that for any action $a$ and valid partial state $\delta$ such that $a$ is executable in $\delta$, we have

$$
\Phi^{pl+}(a, \delta) = \Phi^{pl}(a, \delta) = \Phi^{0}(a, \delta)
$$

(149)

$$
\Phi^{pc+}(a, \delta) = \Phi^{pc}(a, \delta) = \Phi^{0}(a, \delta)
$$

(150)
1. *Proof of (149).* Since $D$ contains no static causal law we have

$$E(a, \delta) = de(a, \delta)$$

$$ph(a, \delta) = ph^0(a, \delta) = (pe(a, \delta) \cup \{l \mid \neg l \notin \delta\}) \setminus \neg de(a, \delta)$$

$$\neg ph(a, \delta) = (-pe(a, \delta) \cup \{l \mid l \notin \delta\}) \setminus de(a, \delta)$$

$$= (-pe(a, \delta) \cup (L \setminus \delta)) \setminus de(a, \delta)$$

Hence,

$$\{l \mid l \notin \neg ph(a, \delta)\} = \{l \mid l \notin (-pe(a, \delta) \cup (L \setminus \delta)) \setminus de(a, \delta)\}$$

$$= \{l \mid l \notin (-pe(a, \delta) \cup (L \setminus \delta)) \} \cup \{l \mid l \in de(a, \delta)\}$$

(as $\{l \mid l \notin X \setminus Y\} = \{l \mid l \notin X\} \cup \{l \mid l \in Y\}$)

$$= \{l \mid l \notin -pe(a, \delta) \land l \notin (L \setminus \delta)\} \cup de(a, \delta)$$

$$= \{l \mid l \notin -pe(a, \delta) \land l \in \delta\} \cup de(a, \delta)$$

$$= (\delta \setminus -pe(a, \delta)) \cup de(a, \delta)$$

By the definition of $\Phi^{ph}$ and $\Phi^0$ we have

$$\delta^{ph}_{1} = \Phi^{ph}(a, \delta) = \{l \mid l \notin -ph(a, \delta)\} = (\delta \setminus -pe(a, \delta)) \cup de(a, \delta) = \Phi^0(a, \delta)$$

(151)

It is not difficult to show that

$$ph(a, \delta, \delta^{ph}_{1}) = ph(a, \delta)$$

(152)
Hence, we have

$$
\delta_{2}^{ph} = \delta_{1}^{ph}
$$

On the other hand, it follows from the construction of $\delta_{i}^{ph}$ that if $\delta_{i}^{ph} = \delta_{i+1}^{ph}$ then for every $k \geq i$ we have $\delta_{k}^{ph} = \delta_{i}^{ph}$. Therefore, for $k \geq 1$ we have

$$
\delta_{k}^{ph} = \delta_{1}^{ph}
$$

Thus, we have

$$
\Phi^{ph+}(a, \delta) = \delta_{1}^{ph} = \Phi^{ph}(a, \delta) = \Phi^{0}(a, \delta)
$$

2. Proof of (150). Since $\mathcal{D}$ contains no static causal law, we have

$$
pc(a, \delta) = pc^{0}(a, \delta) = pe(a, \delta) \setminus \delta
$$

As a result, we have

$$
\Phi^{pc}(a, \delta) = de(a, \delta) \cup (\delta \setminus pc(a, \delta))
$$

$$
= de(a, \delta) \cup (\delta \setminus (pc(a, \delta) \setminus \delta))
$$

$$
= de(a, \delta) \cup (\delta \setminus (\neg pc(a, \delta) \setminus \delta))
$$

$$
= de(a, \delta) \cup (\delta \setminus \neg pc(a, \delta))
$$

$$
(\text{as } \delta \cap \neg \delta = \emptyset)
$$

$$
= \Phi^{0}(a, \delta)
$$

On the other hand, it is easy to see that as $\mathcal{D}$ contains no static causal laws we have

$$
\Phi^{pc+}(a, \delta) = \Phi^{pc}(a, \delta)
$$
Hence, we have

\[ \Phi^{pc+}(a, \delta) = \Phi^{pc}(a, \delta) = \Phi^0(a, \delta) \]

Proof done.

D.4.2 Proof of Proposition 4.8

First of all let us extend the result in Lemma C.2 (presented in the very beginning of Section C) to simple domain descriptions.

**Lemma D.3** Let \( D \) be a simple domain description. Let \( s \) be a state, \( \delta \subseteq s \) be a partial state, and \( a \) be an action that is executable in \( \delta \). Let \( s' \) be a successor state of \( s \) and let \( \delta' = \Phi^A(a, \delta) \). Then, for each fluent literal \( l \in s' \setminus \delta' \), there exists a fluent literal \( l_1 \) in \( s \setminus \delta \) such that \( l \triangleright l_1 \).

**Proof.** Consider a fluent literal \( l \in s' \setminus \delta' \). If \( l \in s \setminus \delta \) then the proposition is trivial because by definition, \( l \) depends on itself and thus we can take \( l_1 = l \in s \setminus \delta \) to have \( l \triangleright l_1 \). Now, consider the case that \( l \notin s \setminus \delta \). There are two possibilities

1. \( l \notin s \), or
2. \( l \in \delta \).

Let us consider each possibility in turn.

1. \( l \notin s \). As \( l \in s' \setminus \delta' \), we have \( l \in s' \). It follows from Observation 4.1 that one of the following two cases occurs
(a) \( \mathcal{D} \) contains a dynamic causal law

\[ e \text{ causes } l \text{ if } \psi \]

such that \( e \in a \) and \( \psi \) holds in \( s \).

If \( \psi \) holds in \( \delta \) then \( l \) is a direct effect of \( a \) in \( \delta \), i.e., \( l \in \text{de}(a, \delta) \subseteq E(a, \delta) \).

By Definitions 4.5 and 4.7 we have \( l \in \delta' \). This contradicts to \( l \in s' \setminus \delta' \).

Because \( \psi \) holds in \( s \) and does not hold in \( \delta \), we have

\[ \psi \subseteq s \quad \text{and} \quad \psi \not\subseteq \delta \]

As a result, there exists a literal \( l_1 \in \psi \) such that \( l_1 \in s \setminus \delta \). By the definition of \( \blacktriangle \) (Definition 4.10), we have \( l \blacktriangle l_1 \).

(b) \( \mathcal{D} \) contains a dynamic causal law

\[ e \text{ causes } g_1 \text{ if } \psi \]

and a sequence of static causal laws \([g_2 \text{ if } g_1], [g_3 \text{ if } l_2], \ldots, [g_n \text{ if } g_{n-1}], [l \text{ if } g_n] \]

such that \( e \in a \) and \( \psi \) holds in \( s \).

If \( \psi \) holds in \( \delta \) then \( g_1 \in \text{de}(a, \delta) \subseteq E(a, \delta) \) as \( E(a, \delta) \) is closed under the set of static causal laws in \( \mathcal{D} \). Hence, \( l \in \delta' \) as \( E(a, \delta) \subseteq \delta' \), which contradicts to the assumption \( l \in s' \setminus \delta' \).

So, we have \( \psi \) does not hold in \( \delta \). Similarly to previous case, this implies that there exists a fluent literal \( l_1 \in \psi \) such that \( l_1 \in s \setminus \delta \). Hence, we
have
\[ l \uparrow g_n \uparrow g_{n-1} \uparrow \ldots \uparrow g_1 \uparrow l_1 \]

2. \( l \in \delta \).

(a) \( \delta' = \Phi^{ph}(a, \delta) \). In this case, we have
\[ \delta' = Cl_D(\{l \mid l \not\in \neg ph(a, \delta)\}) \]

Since \( l \not\in \delta' \), this implies that \( l \in \neg ph(a, \delta) \), i.e., \( \neg l \in ph(a, \delta) \). By the definition of \( ph(a, \delta) \) (Equation (20)), there are three possible cases

i. \( \neg l \in pe(a, \delta) \), i.e., \( D \) contains a dynamic causal law

\[ e \text{ causes } \neg l \text{ if } \psi \]

such that \( e \in a \) and \( \psi \) possibly holds in \( \delta \). The latter implies that
\[ \neg \psi \cap \delta = \emptyset \quad (153) \]

As \( l \in s' \) and \( s' \) is a state, we have \( \neg l \not\in s' \). Hence, \( \neg l \not\in de(a, s) \) as \( de(a, s) \subseteq s' \). By definition of \( e(a, s) \), it follows that \( \psi \) does not hold in \( s \), i.e.,
\[ \neg \psi \cap s \neq \emptyset \quad (154) \]

From (153) and (154), it follows that there exists a fluent literal \( l_1 \in \neg \psi \) such that \( l_1 \in s \setminus \delta \). Because

\[ e \text{ causes } \neg l \text{ if } \psi \]
belongs to $\mathcal{D}$, this implies that $\neg l \bowtie \neg l_1$. By the definition of dependencies, $l \bowtie l_1 \in s \setminus \delta$.

ii. $\neg l \in \delta$. Because $l \in \delta$, and $\delta$ is consistent, this case never happens.

iii. $\mathcal{D}$ contains a sequence of static causal laws

$$[g_2 \text{ if } g_1], [g_3 \text{ if } g_2], \ldots, [g_n \text{ if } g_{n-1}], [\neg l \text{ if } g_n]$$

such that $g_1 \in (pe(a, \delta) \cup \{l \mid \neg l \not\in \delta\}) \setminus \neg E(a, \delta))$. Consider the following cases:

A. $g_1 \in pe(a, \delta)$. By definition, this means that $\mathcal{D}$ contains a dynamic causal law

$$e \text{ causes } g_1 \text{ if } \psi$$

such that $e \in a$ and $\psi$ possibly holds in $\delta$.

As $\psi$ possibly holds in $\delta$, we have

$$\neg \psi \cap \delta = \emptyset \quad (155)$$

On the other hand, it is easy to see that $\psi$ does not hold in $s$ as if otherwise, we would have $\neg l \in s'$, which contradicts to the assumption $l \in s'$. Therefore, we have

$$\neg \psi \cap s \neq \emptyset \quad (156)$$

By (155) and (156), there exists a literal $l_1 \in s \setminus \delta$ such that $\neg l_1 \in \psi$. 279
It is easy to see that

\[-l \leftarrow g_n \leftarrow g_{n-1} \leftarrow \ldots g_1 \leftarrow \neg l_1\]

Hence, we have \(l \leftarrow l_1\).

B. \(g_1 \in \{l \mid \neg l \notin \delta\}\) and \(g_1 \notin \neg E(a, \delta)\).

It is easy to see that \(g_1\) does not hold in \(s\) as if otherwise, we would have \(-l \in s'\), which contradicts to \(l \in s'\). Because \(s\) is a state, it follows that \(-g_1 \in s\).

As \(-g_1 \notin \delta\) and \(-g_1 \in s\), we have \(-g_1 \in s \setminus \delta\). On the other hand, by the definition of dependencies, we have \(l \leftarrow \neg g_1\). Accordingly, we can select \(l_1 = \neg g_1 \in s \setminus \delta\) to have \(l \leftarrow l_1\).

(b) \(\delta' = \Phi^{pc}(a, \delta) = Cl_D(E(a, \delta) \cup (\delta \setminus \neg pc(a, \delta)))\). As \(l \notin \delta'\), it follows that \(l \notin (\delta \setminus \neg pc(a, \delta))\). As \(l \in \delta\), this implies that \(l \in \neg pc(a, \delta)\), i.e., \(-l \in pc(a, \delta)\). By the definition of \(pc(a, \delta)\) (Equation (30)), either \(-l \in pc(a, \delta) \setminus \delta\) or there exists a sequence of static causal laws

\[[g_2 \text{ if } g_1], [g_3 \text{ if } g_2], \ldots, [g_n \text{ if } g_{n-1}], [\neg l \text{ if } g_n]\]

such that \(g_1 \in pc(a, \delta) \setminus \delta\).

If the first case occurs then \(D\) must contain a dynamic causal law

\(e \text{ causes } \neg l \text{ if } \psi\)

280
such that $e \in a$ and $\psi$ possibly holds in $\delta$, i.e.,

$$\neg \psi \cap \delta = \emptyset$$

On the other hand, $\psi$ cannot hold in $s$, i.e.,

$$\neg \psi \cap s \neq \emptyset$$

as if otherwise, we would have $\neg l \in s'$ which contradicts to $l \in s'$. Accordingly, there exists a literal $l_1 \in \neg \psi$ such that $l_1 \in s \setminus \delta$. By definition, we also have $l \bowtie l_1$.

Now consider the second case. As $g_1 \in pe(a, \delta) \setminus \delta$, $D$ contains a dynamic causal law

$$e \text{ causes } g_1 \text{ if } \psi$$

such that $\psi$ possibly holds in $\delta$, i.e.,

$$\neg \psi \cap \delta = \emptyset$$

On the other hand, it is easy to see that $\psi$ does not hold in $s$, i.e.,

$$\neg \psi \cap s \neq \emptyset$$

as if otherwise, $g_1$, and thus, $g_2, \ldots, g_n$, and $\neg l$ all belong to $s'$ which contradicts to $l \in s'$. Because $\neg \psi \cap \delta = \emptyset$ and $\neg \psi \cap s \neq \emptyset$, there exists a fluent literal $l_1 \in \neg \psi$ such that $l_1 \in s \setminus \delta$. It is easy to see that

$$\neg l \bowtie g_n \bowtie \ldots \bowtie g_1 \bowtie \neg l_1$$
As $\neg l \bowtie \neg l_1$, we have $l \bowtie l_1 \in s \setminus \delta$.

So, in both cases ($\delta' = \Phi^{ph}(a, \delta)$ or $\delta' = \Phi^{pc}(a, \delta)$), we can find a literal $l_1 \in s \setminus \delta$ such that $l \bowtie l_1$.

Proof done.

The proof of this proposition is similar to the proof of Proposition 3.2 but makes use of the result in Lemma D.3 instead of the result in Lemma C.2.

D.4.3 Proof of Theorem 4.8

Membership follows from Propositions 4.2, 4.3, 4.5, and 4.6. Hardness follows from the fact that the approximations coincide with the 0-approximation [126] on situations without static causal laws (Theorem 4.6), and from Theorem 2.3.

D.5 Proofs of Results in Section 4.6

D.5.1 Proof of Theorem 4.9

The proof is primarily based on the splitting theorems (see Section 2.4.4) for disjunctive logic programs. As the rule (54) of $\pi_h(P)$ contains a cardinality constraint which is specific to $\text{sm}o\text{d}e\text{ls}$, the first step in the proof is to convert this rule into

\begin{align*}
\text{occ}(E, T) \mid \neg\text{occ}(E, T) & \leftarrow \text{not goal}(T) \\
& \leftarrow \text{not occ}(A, T), \text{not goal}(T)
\end{align*}

(157) \quad (158)
Let $\pi_h(P)$ denotes the new program. Notice that this conversion does not affect the consistency of the program: if $\pi_h(P)$ has an answer set then so does $\pi_h(P)$ and vice versa. Furthermore, each answer set for $\pi_h(P)$ corresponds to an answer set for $\pi_h(P)$ and vice versa and the only difference between the two answer sets is that the latter may contain extra literals of the form $\neg occ(E,T)$.

Then, we divide $\pi_h(P)$ into two disjoint subprograms:

1. $\pi^c_h(P)$ consisting of constraints of $\pi_h(P)$ and
2. $\pi^*_{h}(P)$ consisting of normal logic programming rules of $\pi_h(P)$.

We now introduce some notations that will be used in this proof. For a logic program $\Pi$, by $lit(\Pi)$ we denote the set of atoms in $\Pi$. If $Z$ is a splitting set for $\Pi$ and $\Sigma$ is a set of atoms then by $b_Z(\Pi)$ and $e_Z(\Pi \setminus b_Z(\Pi), \Sigma)$, we denote the bottom part of $\Pi$ with respect to $Z$ and the partial evaluation of the top part with respect to $(Z, \Sigma)$ (see Section 2.4.4 for more information about these notations).

Let $V$ be the following set of atoms

$$\{\text{fluent}(f), \text{literal}(f), \text{literal}(\neg f), \text{contrary}(f, \neg f), \text{contrary}(\neg f, f) \mid f \in F\} \cup$$
$$\{\text{action}(e) \mid e \in A\} \cup \{\text{time}(t) \mid t \in \{1..h\}\} \cup \{\text{time1}(t) \mid t \in \{1..h+1\}\}$$

For $1 \leq i \leq h + 1$ let $A_i$ be the following set of atoms

$$\{\text{occ}(e,i) \mid e \in A\} \cup \{\text{holds}(l,i), \text{ef}(l,i), \text{ph}(l,i) \mid l \in L\} \cup \{\text{goal}(i)\}$$
and let

\[ A = \bigcup_{i=1}^{h+1} A_i \]  

(161)

Furthermore, for a set of atoms \( \Sigma \subseteq A \) and a set of predicate symbols \( X \), by \( \Sigma^X \) we denote the set of atoms in \( \Sigma \) whose predicate symbols are in \( X \), and by \( \delta_i(\Sigma) \) we denote the set of fluent literals \( \{ l \mid \text{holds}(l, i) \in \Sigma \} \).

Now let \( S \) be an answer set for \( \pi_h(\mathcal{P}) \). Let \( \bar{S} \) be the answer set of \( \pi_h(\mathcal{P}) \) that corresponds to \( S \). By the definition of an answer set for a program with constraints, \( \bar{S} \) is also an answer set for \( \pi_h^*(\mathcal{P}) \) and \( \bar{S} \) does not violate any constraint in \( \pi_h^*(\mathcal{P}) \). Furthermore \( S \) and \( \bar{S} \) differ from each other only in the appearance of literals of the form \( \neg \text{occ}(e, i) \).

It is easy to see that the set \( V \) (Equation (159)) is a splitting set for \( \pi_h^*(\mathcal{P}) \). Furthermore, the bottom part \( b_V(\pi_h^*(\mathcal{P})) \) is a positive program and has only one answer set

\[ X_0 = V \]  

(162)

The partial evaluation of the top part of \( \pi_h^*(\mathcal{P}) \) with respect to \( X_0 \),

\[ \pi_0 = e_V(\pi_h^*(\mathcal{P}) \setminus b_V(\pi_h^*(\mathcal{P})), X_0) \]

is the following set of rules (the condition for each rule follows that rule and by default \( t \) ranges over \( 1..h \), \( l \) ranges over the set of fluent literals \( L \), and \( e \) ranges over the set of elementary actions \( A \) unless otherwise specified explicitly):
\[
\text{holds}(l, 1) \leftarrow (l \in \delta)
\]

\[
\text{ef}(l, t) \leftarrow \text{occ}(e, t), \text{holds}(\psi, t)
\]

\[
([ e \text{ causes } l \text{ if } \psi ] \in \mathcal{D})
\]

\[
\text{ph}(l, t) \leftarrow \text{occ}(e, t), \text{not ef}(\neg l, t), \text{not holds}(\neg \psi, t)
\]

\[
([ e \text{ causes } l \text{ if } \psi ] \in \mathcal{D})
\]

\[
\text{ef}(l, t) \leftarrow \text{ef}(\psi, t)
\]

\[
([ l \text{ if } \psi ] \in \mathcal{D})
\]

\[
\text{ph}(l, t) \leftarrow \text{not ef}(\neg l, t), \text{ph}(\psi, t), \text{not ef}(\neg \psi, t)
\]

\[
([ l \text{ if } \psi ] \in \mathcal{D})
\]

\[
\text{holds}(l, t) \leftarrow \text{holds}(\psi, t)
\]

\[
([ l \text{ if } \psi ] \in \mathcal{D}, 1 \leq t \leq h + 1)
\]

\[
\text{goal}(t) \leftarrow \text{holds}(\mathcal{G}, t)
\]

\[
(1 \leq t \leq h + 1)
\]

\[
\text{goal}(t + 1) \leftarrow \text{goal}(t)
\]

\[
\text{ph}(l, t) \leftarrow \text{not holds}(\neg l, t), \text{not ef}(\neg l, t)
\]

\[
\text{holds}(l, t+1) \leftarrow \text{not ph}(\neg l, t)
\]

\[
\text{occ}(e, t) \mid \text{¬occ}(e, t) \leftarrow \text{not goal}(t)
\]
By the splitting set theorem, there exists an answer set $S_0$ for $\pi_0$ such that

$$S = S_0 \cup X_0$$  \hspace{1cm} (174)$$

where $X_0 = V$ is the answer set for $b_V(\pi^*_h(P))$ (see (162)).

The program $\pi^*_h(P)$ consists of the following constraints:

$$\leftarrow \text{occ}(b, t), \text{not holds}(\neg \psi, t)$$  \hspace{1cm} (175)$$

$$\leftarrow [\text{impossible } b \text{ if } \psi \in D]$$

$$\leftarrow \text{not goal}(h + 1)$$  \hspace{1cm} (176)$$

$$\leftarrow \text{not occ}(A, t), \text{not goal}(t)$$  \hspace{1cm} (177)$$

Let $U_i$ be the set of atoms in $\pi_0$ whose time parameter is less than or equal to $i$, i.e.,

$$U_i = \bigcup_{j=1}^{i} A_j$$  \hspace{1cm} (178)$$

It is easy to see that the sequence $\langle U_i \rangle_{i=1}^{h+1}$ is a splitting sequence for $\pi_0$. By the splitting sequence theorem, since $S_0$ is an answer set for $\pi_0$, there must be a sequence of sets of literals $\langle X_i \rangle_{i=1}^{h+1}$ such that $X_i \subseteq U_i \setminus U_{i-1}$, and

- $S_0 = \bigcup_{i=1}^{h+1} X_i$

- $X_1$ is an answer set for

$$\pi_1 = b_{U_1}(\pi_0)$$  \hspace{1cm} (179)$$

- for every $1 < i \leq h + 1$, $X_i$ is an answer set for

$$\pi_i = e_{U_i}(b_{U_i}(\pi_0) \setminus b_{U_{i-1}}(\pi_0), \bigcup_{1 \leq t \leq i-1} X_t)$$  \hspace{1cm} (180)$$
We can check that the program $\pi_i$ is as follows. If $i = 1$ then

$$
\pi_1 = \{(163)-(169), (171), (173) \mid t = 1\}
$$

(181)
If $i > 1$, then $\pi_i$ consists of the following rules

\begin{align*}
e f(l, i) & \leftarrow oc(e, i), \text{holds}(\psi, i) & (182) \\
(\left[ e \text{ causes } l \text{ if } \psi \right] \in \mathcal{D}) \\
ph(l, i) & \leftarrow oc(e, i), not ef(-l, i), not \text{holds}(-\psi, i) & (183) \\
(\left[ e \text{ causes } l \text{ if } \psi \right] \in \mathcal{D}) \\
e f(l, i) & \leftarrow ef(\psi, i) & (184) \\
(\left[ l \text{ if } \psi \right] \in \mathcal{D}) \\
ph(l, i) & \leftarrow not ef(-l, i), ph(\psi, i), not ef(-\psi, i) & (185) \\
(\left[ l \text{ if } \psi \right] \in \mathcal{D}) \\
\text{holds}(l, i) & \leftarrow \text{holds}(\psi, i) & (186) \\
(\left[ l \text{ if } \psi \right] \in \mathcal{D}) \\
goal(i) & \leftarrow \text{holds}(G, i) & (187) \\
goal(i) & \leftarrow (goal(i - 1) \in X_{i-1}) & (188) \\
ph(l, i) & \leftarrow not \text{holds}(-l, i), not ef(-l, i) & (189) \\
\text{holds}(l, i) & \leftarrow (ph(-l, i) \notin X_{i-1}) & (190) \\
oc(e, i) | \neg oc(e, i) & \leftarrow not \text{goal}(i) & (191)
\end{align*}
and $\pi_{h+1}$ is the set of rules

$$holds(l, h+1) \leftarrow holds(\psi, h+1) \quad (192)$$

$$([ l \text{ if } \psi ] \in \mathcal{D})$$

$$goal(h+1) \leftarrow holds(G, h+1) \quad (193)$$

$$goal(h+1) \leftarrow \quad (194)$$

$$(goal(h) \in X_h)$$

$$holds(l, h+1) \leftarrow \quad (195)$$

$$(ph(-l, h) \not\in X_h)$$

The relationship between logic programs $\pi_h(P), \pi_h(P), \pi^c_h(P), \pi^*_h(P)$, and $\pi_i$’s and the relationship between their answer sets are shown graphically in Figure D.1.

It is easy to see that

$$\delta_i(S) = \delta_i(\bar{S}) = \delta_i(S_0) = \delta_i(X_i) \quad (196)$$

Therefore, from now on, we will use $\delta_i$ to refer to these set of fluent literals.

We have the following lemma about the correctness of a code fragment that encodes the closure of a set of literals.

**Lemma D.4** Let $i$ be an integer greater than 0, and $x$ be a binary predicate. For any set $\sigma$ of literals, the following program

$$x(l, i) \leftarrow \quad (l \in \sigma)$$

$$x(l, i) \leftarrow x(\psi, i) \quad ([ l \text{ if } \psi ] \in \mathcal{D})$$

has the unique answer set $\{x(l, i) \mid l \in Cl_{\mathcal{D}}(\sigma)\}$. 

289
π_h(\mathcal{P})[S]

convert to

\bar{\pi}_h(\mathcal{P})[\bar{S}]

constraints
disjunctive logic program

\pi^c(\mathcal{P})

\pi^*_h(\mathcal{P})[\bar{S} = S_0 \cup X_0]

use splitting set \( V \) (Eq. (159))

\gamma_V(\pi^*_h(\mathcal{P}))[X_0 = V]

\pi_0[S_0 = \bigcup_{i=1}^{h+1} X_i]

use splitting sequence \( \langle U_i \rangle_{i=1}^{h+1} \) (Eq. (178))

\pi_1[X_1] \quad \pi_2[X_2] \quad \ldots \quad \pi_{h+1}[X_{h+1}]

Figure D.1: The relationship between subprograms of \( \pi_h(\mathcal{P}) \)

**Proof.** By the definition of a model of a positive program, it is easy to see that the above program has the unique answer set \( \{x(l, i) \mid l \in \sigma_{D}^{\text{init}}\} = \{x(l, i) \mid l \in Cl_D(\sigma)\} \) (see Lemma D.1).

Given the answer set \( \bar{S} \), for any integer \( 1 \leq i \leq h \), let

\[ a_i = \{e \mid \text{occ}(e, i) \in \bar{S}\} \]  

(197)

By the constraint (175), it is easy to see that if \( a_i \neq \emptyset \) then it is executable in \( \delta_i \) (a formal proof for this can be seen in the proof of Lemma D.6). Furthermore, we have the following result.

**Lemma D.5** If \( a_i \neq \emptyset \) then (recall that \( X_i \) is the answer set for \( \pi_i \))
1. \( ef(l, i) \in X_i \iff l \in E(a_i, \delta_i), \) and

2. \( ph(l, i) \in X_i \iff l \in ph(a_i, \delta_i). \)

**Proof.** Let us split \( \pi_i \) (rules (182)–(191) with \( t = 1 \) if \( i > 1 \), or rules (163)-(169), (171), and (173) with \( t = 1 \) if \( i = 1 \)) using the splitting set \( Z_1 = A_i^{\{holds, occ, \neg occ, goal\}}. \)

By the splitting set theorem, we have

\[
X_i = M \cup N
\] (198)

where \( M \) is an answer set for \( b_{Z_1}(\pi_i) \) and \( N \) is an answer set for \( \Pi_1 = e_{Z_1}(\pi_i \setminus b_{Z_1}(\pi_i), M) \) – the program with the following rules

- \( ef(l, i) \leftarrow \)
  
  \( ([ e \text{ causes } l \text{ if } \psi ] \in D, occ(e, i) \in M, holds(\psi, i) \subseteq M) \)

- \( ph(l, i) \leftarrow not ef(\neg l, i) \)
  
  \( ([ e \text{ causes } l \text{ if } \psi ] \in D, occ(e, i) \in M, holds(\neg \psi, i) \cap M = \emptyset) \)

- \( ef(l, i) \leftarrow ef(\psi, i) \)
  
  \( ([ l \text{ if } \psi ] \in \mathcal{D}) \)

- \( ph(l, i) \leftarrow not ef(l, i), ph(\psi, i), not ef(\neg \psi, i) \)
  
  \( ([ l \text{ if } \psi ] \in \mathcal{D}) \)

- \( ph(l, i) \leftarrow not ef(\neg l, i) \)
  
  \( (\text{holds}(\neg l, i) \not\in M) \)
Note that $\delta_i(M) = \delta_i(X_i) = \delta_i$ as $N \cap Z_1 = \emptyset$. Hence, the conditions for the first two rules can be written as

$$ l \in de(a_i, \delta_i) $$

and

$$ l \in pe(a_i, \delta_i) $$

respectively.

If we further split $\Pi_1$ using $Z_2 = A_i^{\{ef\}}$ then by Lemma D.4 the bottom part has the only answer set

$$ N_1 = \{ef(l, i) \mid l \in Cl_D(de(a_i, \delta_i))\} = \{ef(l, i) \mid l \in E(a_i, \delta_i)\} $$

and the partial evaluation of the top part with respect to $N_1$ is

$$ ph(l, i) \leftarrow (l \in pe(a_i, \delta_i), ef(-l, i) \not\in N_1) $$

$$ ph(l, i) \leftarrow ph(\psi, i) $$

$$ ([ l if \psi ] \in D, ef(-l, i) \not\in N_1, ef(-\psi, i) \cap N_1 = \emptyset) $$

$$ ph(l, i) \leftarrow (holds(-l, i) \not\in M, ef(-l, i) \not\in N_1) $$

Observe the conditions for the above rules are equivalent to

$$ l \in pe(a_i, \delta_i) \setminus \neg E(a_i, \delta_i) $$
\[ l \text{ if } \psi \in \mathcal{D}, l \not\in \neg E(a_i, \delta_i), \neg \psi \cap E(a_i, \delta_i) = \emptyset \]

\[ l \in \{ l \mid \neg l \not\in \delta_i \} \setminus \neg E(a_i, \delta_i) \]

respectively. Similarly to Lemma D.4, with the observation that the first and the third conditions above can be rewritten as

\[ l \in ph^0(a_i, \delta_i) \]

we can easily show that this program has the only answer set

\[ N_2 = \{ ph(l, i) \mid l \in ph(a_i, \delta_i) \} \]

By the splitting theorem we have

\[ N = N_1 \cup N_2 = \{ ef(l, i) \mid l \in E(a_i, \delta_i) \} \cup \{ ph(l, i) \mid l \in ph(a_i, \delta_i) \} \quad (199) \]

Because \( M \) does not contain any atom of the form \( ef(l, i) \) or \( ph(l, i) \), it follows from (198) and (199) that the lemma holds.

\[ \square \]

The following lemma shows that \( \pi_h(P) \) correctly implements the transition function \( \Phi^{ph} \).

**Lemma D.6** We have the following results (see (196) and (197) for the definitions of \( \delta_i \) and \( a_i \)).

1. \( \delta_1 = \delta \) (recall that \( \delta \) is the initial partial state of the planning problem \( P \)),

293
2. for \(1 \leq i \leq h\), if \(a_i \neq \emptyset\) then \(a_i\) is executable in \(\delta_i\) and

\[\Phi_{\text{ph}}(a, \delta_i) = \delta_{i+1}\]

**Proof.**

1. First of all, let us prove that \(\delta_1 = \delta\). Consider the program \(\pi_1\) (see (181)): Then \(Z_1 = A_1^{\{\text{holds}\}}\) is a splitting set for \(\pi_1\). The bottom part, \(b_{Z_1}(\pi_1)\), consists of the following rules

\[
\text{holds}(l, 1) \leftarrow \\
\quad (l \in \delta)
\]

\[
\text{holds}(l, 1) \leftarrow \text{holds}(\psi, 1) \\
\quad ([l \text{ if } \psi] \in D)
\]

By Lemma D.4, this program has only one answer set

\[M = \{\text{holds}(l, 1) \mid l \in \text{Cl}_D(\delta)\} = \{\text{holds}(l, 1) \mid l \in \delta\}\]

as \(\text{Cl}_D(\delta) = \delta\)

As a result, we have

\[\delta_1(\bar{S}) = \delta_1(X_1) = \delta_1(M) = \delta\]

2. Let us prove Item 2. Suppose \(a_i \neq \emptyset\). Since \(\bar{S}\) satisfies the constraint (175), there is no impossibility condition

\[\text{impossible } b \text{ if } \psi\]
in $\mathcal{D}$ such that $b \subseteq a_i$ and $\text{holds}(\neg \psi, i) \cap S = \emptyset$ (or equivalently $\psi$ possibly holds in $\delta_i$). This means that $a$ is executable in $\delta_i$.

Now consider the program $\pi_{i+1}$ and let $Z_1 = A_{i+1}^{\text{holds}}$. Then, $Z_1$ is a splitting set for $\pi_{i+1}$. Hence, by the splitting set theorem, $X_{i+1} = M \cup N$, where $M \subseteq Z_1$ is an answer set for $\Pi_1 = b_{Z_1}(\pi_{i+1})$ and $N$ is an answer set for $\Pi_2 = e_{Z_1}(\pi_{i+1} \setminus b_{Z_1}(\pi_{i+1}), M)$.

Observe that no matter whether $i < h$ or $i = h$, $\Pi_1$ is the following set of rules

\[
\begin{align*}
\text{holds}(l, i + 1) & \leftarrow \text{holds}(\psi, i + 1) \\
& \quad \left(\{l \text{ if } \psi\} \in \mathcal{D}\right) \\
\text{holds}(l, i + 1) & \leftarrow (\text{ph}(\neg l, i) \not\in X_i)
\end{align*}
\]

By Lemma D.5, the condition for the third rule is equivalent to

\[l \not\in \neg \text{ph}(a_i, \delta_i)\]

As a result, by Lemma D.4, this program has the only answer set

\[M = \{\text{holds}(l, i + 1) \mid l \in Cl_\mathcal{D}([\{l \mid l \not\in \neg \text{ph}(a_i, \delta_i)\})\} = \{\text{holds}(l, i + 1) \mid l \in \Phi^{\text{ph}}(a_i, \delta_i)\}\]

Therefore, we have

\[\delta_{i+1} = \delta_{i+1}(M) = \Phi^{\text{ph}}(a_i, \delta_i)\]
Now let us prove the theorem. From the discussion in Section 4.6.1, we know that \( \alpha_h(S) = \alpha_h(\bar{S}) = \langle a_1, \ldots, a_k \rangle \) for some \( 1 \leq k \leq h \) such that \( a_i \neq \emptyset \) for \( 1 \leq i \leq k \) and \( a_{k+1} = \emptyset \). From this, and by Items 1 & 2 of Lemma D.6, we have that

\[
\hat{\Phi}^{ph}(\alpha_h(S), \delta) = \hat{\Phi}^{ph}(\alpha_h(S), \delta_1) = \delta_{k+1}
\]

Since \( a_{k+1} = \emptyset \), there is no elementary action \( e \) such that \( occ(e, k+1) \in \bar{S} \). If \( k < h \) then by constraint (177), it follows that \( goal(k+1) \in \bar{S} \). On the other hand, by constraint (176), we have \( goal(h+1) \in \bar{S} \). So, in the both cases, we have \( goal(k+1) \in \bar{S} \).

Now, observe that there are only two rules that have the head \( goal(k+1) \): (169) with \( t = k + 1 \) and (170) with \( t = k \). Furthermore, since \( a_k \neq \emptyset \) and the rule (173) with \( t = k \) is the only rule of \( \pi_0 \) that has atoms of the form \( occ(e, k) \) as the head, it follows that \( goal(k) \notin \bar{S} \). That is, the body of the rule (170) with \( t = k \), \( goal(k) \), does not hold in \( \bar{S} \). As a result, the body of rule (169) with \( t = k + 1 \) must hold in \( \bar{S} \), i.e, \( holds(G, k+1) \subseteq \bar{S} \). This implies that \( G \) holds in \( \delta_{k+1} \). Accordingly, we can conclude that \( \alpha_h(S) \) is a solution of \( \langle D, \delta, G \rangle \) with respect to the \( T^{ph}(D) \) approximation.

D.5.2 Proof of Theorem 4.10

We will prove this theorem by construction. Let \( \text{reduct}(\alpha) = \langle a_1, a_2, \ldots, a_k \rangle \) and let \( h = k \). Since \( \text{reduct}(\alpha) \) is a solution of \( P = \langle D, \delta, G \rangle \) with respect to the \( T^{ph}(D) \) approximation, by definition, there exists a sequence of partial states \( \langle \delta_i \rangle_{i=1}^{n+1} \)
such that
\[ \delta_1 = \delta \]
\[ \delta_{i+1} = \Phi_{ph}(a_i, \delta_i) \]
and \( G \) holds in \( \delta_{h+1} \) but does not hold in \( \delta_i \) for \( 1 \leq i \leq h \).

For each \( 1 \leq i \leq h \), construct a set of literals \( Y_i \) as follows

\[ Y_i = \{ \text{holds}(l, i) \mid l \in \delta_i \} \cup \{ \text{ef}(l, i) \mid l \in E(a_i, \delta_i) \} \cup \{ \text{ph}(l, i) \mid l \in \text{ph}(a_i, \delta_i) \} \cup \{ \text{occ}(e, i) \mid e \in a_i \} \cup \{ \neg \text{occ}(e, i) \mid e \not\in a_i \} \]

and let
\[ Y_{h+1} = \{ \text{holds}(l, h+1) \mid l \in \delta_{h+1} \} \cup \{ \text{goal}(h+1) \} \]

and let \( S'_0 = \bigcup_{i=1}^{h+1} Y_i \).

We have the following lemma.

**Lemma D.7** For \( 1 \leq i \leq h \), let \( M = Y_i^{\{\text{holds,occ,}\neg\text{occ,goal}\}} \) and let \( \Pi(Y_i) \) be the following
program

\[ ef(l, i) \leftarrow (\text{[e causes l if } \psi ] \in D, \text{occ}(e, i) \in M, \text{holds}(\psi, i) \subseteq M) \]

\[ ph(l, i) \leftarrow \neg ef(\neg l, i) \]

\[ ph(l, i) \leftarrow \neg ef(\neg l, i), ph(\psi, i), \neg ef(\neg \psi, i) \]

\[ ef(l, i) \leftarrow ef(\psi, i) \]

\[ ph(l, i) \leftarrow \neg ef(\neg l, i), \neg \text{holds}(\neg l, i) \notin M \]

\[ \neg \text{holds}(\neg l, i) \notin M \]

Then \( N = Y_i^{\{ef, ph\}} \) is the only answer set for \( \Pi(Y_i) \).

**Proof.** It is easy to see that this program is stratified and thus it is consistent and has a unique answer set. However, it follows from the proof of Lemma D.5 that \( N = Y_i^{\{ef, ph\}} \) is an answer set for \( \Pi(Y_i) \). As a result, it is also the only answer set for \( \Pi(Y_i) \).

\( \square \)

For \( 1 \leq i \leq h+1 \), let \( \pi'_i \) be the program defined in the same way as the program \( \pi_i \) in the proof of Theorem 4.9 (Section D.5.1, Equations (179) and (180)) except that every occurrence of \( X_t \) is replaced with \( Y_t \). We have the following lemma.
Lemma D.8 For $1 \leq i \leq h + 1$, $Y_i$ is an answer set for $\pi_i'$.

Proof. Consider three cases $i = 1$, $1 < i \leq h$ and $i = h + 1$.

1. $i = 1$: Let $Z_1 = A_1^{\{\text{holds,occ,} \neg \text{occ,goal}\}}$. If we use $Z_1$ to split $\pi_1'$ then $b_{Z_1}(\pi_1')$ is the following set of rules

\[
\begin{align*}
\text{holds}(l, 1) & \leftarrow \\
& (l \in \delta) \\
\text{holds}(l, 1) & \leftarrow \text{holds}(\psi, 1) \\
& ([l \text{ if } \psi] \in \mathcal{D}) \\
\text{goal}(1) & \leftarrow \text{holds}(G, 1) \\
\text{occ}(e, 1) | \neg \text{occ}(e, 1) & \leftarrow \text{not goal}(1)
\end{align*}
\]

It is easy to see that $M = Y_1^{\{\text{holds,occ,} \neg \text{occ,goal}\}}$ is an answer set for this program.

The evaluation of the top part, $e_{Z_1}(\pi_1' \setminus b_{Z_1}(\pi_1'), M)$ is the program $\Pi(Y_1)$ that is defined in Lemma D.7. According to this lemma, this program has $N = Y_1^{\{\text{ef,ph}\}}$ as an answer set. As a result, $Y_1 = M \cup N$ is an answer set for $\pi_1'$.

2. $1 < i \leq h$: Using the splitting set $Z_2 = A_i^{\{\text{holds,occ,} \neg \text{occ,goal}\}}$ to split $\pi_i'$, we have that the bottom part $\Pi_1 = b_{Z_2}(\pi_i')$ is the following set of rules (note that as $Y_{i-1}$
does not contain $\text{goal}(i - 1)$, the rule of the form (188) does not exist in $\pi'_i$:

$$
\text{goal}(i) \leftarrow \text{holds}(\mathcal{G}, i)
$$

$$
\text{holds}(l, i) \leftarrow \text{holds}(\psi, i)
$$

$$
([ l \text{ if } \psi ] \in \mathcal{D})
$$

$$
\text{occ}(e, i) | \neg \text{occ}(e, i) \leftarrow \text{not \ goal}(i)
$$

If we further split $\prod_1$ using the splitting set $\mathcal{Z}_3 = \mathcal{A}^{\{\text{holds}\}}$ then by Lemma D.4 and by the construction of $Y_{i-1}$ the bottom part $\prod_2 = b_{\mathcal{Z}_3}(\prod_1)$ has only one answer set

$$
M_1 = \{ \text{holds}(l, i) | l \in \text{Cl}_\mathcal{D}(\{ l | l \not\in \neg \text{ph}(a_{i-1}, \delta_{i-1}) \}) \}
$$

$$
= \{ \text{holds}(l, i) | l \in \delta_i \} = Y_i^{\{\text{holds}\}}
$$

Because $\mathcal{G}$ does not hold in $\delta_i$, from the construction of $Y_i$, we have $\text{holds}(\mathcal{G}) \not\subseteq Y_i^{\{\text{holds}\}} = M_1$. As a result, the evaluation of the top part of $\prod_1$ with respect to $M_1, \Pi_3 = e_{\mathcal{Z}_3}(\Pi_1 \setminus b_{\mathcal{Z}_3}(\Pi_1), M_1)$ is the following

$$
\text{occ}(e, i) | \neg \text{occ}(e, i) \leftarrow \text{not \ goal}(i)
$$

which has $M_2 = Y_i^{\{\text{occ,\neg occ}\}}$ as an answer set.

Therefore, $M = M_1 \cup M_2 = Y_i^{\{\text{holds,occ,\neg occ}\}}$ is an answer set for $\prod_1$. Now consider the evaluation of the top part of $\pi'_i$ with respect to $M, \Pi_4 = e_{\mathcal{Z}_1}(\prod'_i \setminus$
\( b_{\pi_1}(\pi_i', M) \). It is easy to see that \( \Pi_4 = \Pi(Y_i) \), which, by Lemma D.7, has the answer set \( N = Y_i^{\{\text{ef,ph}\}} \). Accordingly, \( M \cup N = Y_i \) (note that \( Y_i \) does not contain \( \text{goal}(i) \)) is an answer set for \( \pi_i' \).

3. \( i = h + 1 \): Similar to the above cases but note that \( G \) holds in \( \delta_{h+1} \).

We have showed that \( Y_i \) is an answer set for \( \pi_i' \). By the splitting sequence theorem, it follows that \( S'_0 = \bigcup_{i=1}^{h+1} Y_i \) is an answer set for \( \pi_0 \). Hence,

\[ \bar{S}' = S'_0 \cup X_0 \]

where \( X_0 \) is defined by (162), is an answer set for \( \pi_h^*(\mathcal{P}) \). On the other hand, it is not difficult to see that \( \bar{S}' \) satisfies the constraints of \( \pi_h^*(\mathcal{P}) \). As a result, \( \bar{S}' \) is an answer for \( \pi_h(\mathcal{P}) \). Let \( S \) be the set of literals obtained from \( \bar{S}' \) by removing literals of the form \( \neg \text{occ}(e, i) \). Then, \( S \) is an answer set for \( \pi_h(\mathcal{P}) \). On the other hand, from the construction of \( Y_i \), we can see that \( \alpha(S) = \text{reduct}(\alpha) \). Accordingly, the theorem holds.
This appendix contains the encoding of a planning instance $P$ for the bomb-in-the-toilet domain from Example 2.1. The first subsection describes the input planning problem. The next subsection presents the corresponding logic program $\pi_h(P)$. The last two subsections list the outputs of smodels for $\pi_h(P)$ when run with parameters $h = 2$.

### E.1 Input Domain

```
fluent(armed(P)) :- pkg(P). % window open
fluent(clogged(T)) :- toilet(T). % window closed but not locked
fluent(safe). % window is closed and locked

action(dunk(P,T)) :-
    pkg(P), toilet(T).
action(flush(T)) :-
    toilet(T).

% impossibility conditions
impossible([dunk(P,T),flush(T)],[]) :-
    pkg(P), toilet(T).
impossible([dunk(P1,T),dunk(P2,T)],[]) :-
    pkg(P1), pkg(P2), toilet(T), P1 \= P2.
impossible([dunk(P,T1),dunk(P,T2)],[]) :-
    pkg(P), toilet(T1), toilet(T2), T1 \= T2.
impossible([dunk(P,T)],[clogged(T)]) :-
    pkg(P), toilet(T).

% dynamic causal laws
causes(dunk(P,T),neg(armed(P)),[]) :-
    action(dunk(P,T)).
causes(dunk(P,T),clogged(T),[]) :-
    action(dunk(P,T)).
```
causes(flush(T),neg(clogged(T)),[]) :-
    action(flush(T)).

% static causal law
caused(safe,L) :-
    findall(neg(armed(P)),pkg(P),L).

% initial state
pkg(p1). pkg(p2). toilet(t1). toilet(t2).
% initially(neg(clogged(1))) :- toilet(T).

% goal
goal(safe).

E.2  The Program  $\pi_h(P)$

% Usage:
% lparse -c h=<height> | smodels
#domain fluent(F).
#domain literal(L,L1).
#domain time(T).
#domain time1(T1).
#domain action(A).

% Input parameters
time(1..h).
time1(1..h+1).

% Action declarations
action(dunk(p1,t1)).
action(dunk(p1,t2)).
action(dunk(p2,t1)).
action(dunk(p2,t2)).
action(flush(t1)).
action(flush(t2)).

% Fluent declarations
fluent(armed(p1)).
fluent(armed(p2)).
fluent(clogged(t1)).
fluent(clogged(t2)).
fluent(safe).

% DOMAIN INDEPENDENT RULES
% Auxiliary Rules
literal(F).
literal(neg(F)).

contrary(F,neg(F)).
contrary(neg(F),F).

% Goal representation
goal(T+1) :-
    goal(T).
:- not goal(h+1).

% Inertial rule
ph(L,T) :-
    contrary(L,L1),
    not holds(L1,T),
    not ef(L1,T).
holds(L,T+1) :-
    contrary(L,L1),
    not ph(L1,T).

% Rules for generating action occurrences
1{occ(X,T):action(X)} :-
    not goal(T).

% DOMAIN DEPENDENT RULES
% Impossibility conditions
:- occ(dunk(p1,t1),T),
   occ(flush(t1),T).
:- occ(dunk(p1,t2),T),
    occ(flush(t2),T).
:- occ(dunk(p2,t1),T),
    occ(flush(t1),T).
:- occ(dunk(p2,t2),T),
    occ(flush(t2),T).
:- occ(dunk(p1,t1),T),
    occ(dunk(p2,t1),T).
:- occ(dunk(p1,t2),T),
    occ(dunk(p2,t2),T).
:- occ(dunk(p2,t1),T),
    occ(dunk(p1,t1),T).
:- occ(dunk(p2,t2),T),
    occ(dunk(p1,t2),T).
:- occ(dunk(p1,t1),T),
    not holds(neg(clogged(t1)),T).
:- occ(dunk(p1,t2),T),
    not holds(neg(clogged(t2)),T).
:- occ(dunk(p2,t1),T),
    not holds(neg(clogged(t1)),T).
:- occ(dunk(p2,t2),T),
    not holds(neg(clogged(t2)),T).

% Static causal laws
holds(safe,T1) :-
    holds(neg(armed(p1)),T1),
    holds(neg(armed(p2)),T1).

ef(safe,T) :-
    ef(neg(armed(p1)),T),
    ef(neg(armed(p2)),T).

ph(safe,T) :-
    not ef(neg(safe),T),
    ph(neg(armed(p1)),T),
    ph(neg(armed(p2)),T),
    not ef(armed(p1),T),
    not ef(armed(p2),T).

% Effects of actions
ef(neg(armed(p1)),T) :-
    occ(dunk(p1,t1),T).
ph(neg(armed(p1)),T) :-
  occ(dunk(p1,t1),T),
  not ef(armed(p1),T).

ef(neg(armed(p1)),T) :-
  occ(dunk(p1,t2),T).

ph(neg(armed(p1)),T) :-
  occ(dunk(p1,t2),T),
  not ef(armed(p1),T).

ef(neg(armed(p2)),T) :-
  occ(dunk(p2,t1),T).

ph(neg(armed(p2)),T) :-
  occ(dunk(p2,t1),T),
  not ef(armed(p2),T).

ef(neg(armed(p2)),T) :-
  occ(dunk(p2,t2),T).

ph(neg(armed(p2)),T) :-
  occ(dunk(p2,t2),T),
  not ef(armed(p2),T).

ef(clogged(t1),T) :-
  occ(dunk(p1,t1),T).

ph(clogged(t1),T) :-
  occ(dunk(p1,t1),T),
  not ef(neg(clogged(t1)),T).

ef(clogged(t2),T) :-
  occ(dunk(p1,t2),T).

ph(clogged(t2),T) :-
  occ(dunk(p1,t2),T),
  not ef(neg(clogged(t2)),T).

ef(clogged(t1),T) :-
  occ(dunk(p2,t1),T).

ph(clogged(t1),T) :-
  occ(dunk(p2,t1),T),
  not ef(neg(clogged(t1)),T).

ef(clogged(t2),T) :-
  occ(dunk(p2,t2),T).

ph(clogged(t2),T) :-
  occ(dunk(p2,t2),T),
  not ef(neg(clogged(t2)),T).

ef(neg(clogged(t1)),T) :-
  occ(flush(t1),T).

ph(neg(clogged(t1)),T) :-
  occ(flush(t1),T),
  not ef(clogged(t1),T).
ef(neg(clogged(t2)),T) :-
    occ(flush(t2),T).
ph(neg(clogged(t2)),T) :-
    occ(flush(t2),T),
    not ef(clogged(t2),T).

% INITIAL STATE

% GOAL SPECIFICATION

goal(T1) :-
    holds(safe,T1).

% HIDE THE FOLLOWING ATOMS

E.3 The output of smodels

$ lparse -c h=2 bomb.smo | smodels
smodels version 2.28. Reading...done
Answer: 1
Stable Model: occ(flush(t2),1) occ(flush(t1),1)
occ(dunk(p2,t1),2) occ(dunk(p1,t2),2)
True
Duration: 0.008
Number of choice points: 1
Number of wrong choices: 0
Number of atoms: 124
Number of rules: 159
Number of picked atoms: 21
Number of forced atoms: 9
Number of truth assignments: 217
Size of searchspace (removed): 4 (0)
APPENDIX F

PROOFS OF CHAPTER 5

This appendix contains the proofs for the propositions and theorems presented in Chapter 5. Recall that we assume that the body of each static law (64) is not empty and $G \neq \emptyset$ for every planning instance $\langle D, \Delta, G \rangle$.

F.1 Proofs of the Results in Section 5.2

F.1.1 Proof of Proposition 5.1

Since $\delta$ is valid, there exists a state $s$ such that $\delta \subseteq s$. Furthermore, because $\Phi_A S(a, \delta) \neq \bot$, we have that $a$ is executable in $\delta$, and thus, executable in $s$ as well. Since we assume that $D$ is consistent, it follows that there exists at least one possible successor state of $s$, say $s'$ (i.e., $s' \in \Phi(a, s)$).

If $a$ is a non-sensing action then $\Phi_A(a, \delta)$ is a valid partial state as it follows implicitly from Theorems 4.2, 4.3, 4.4, and 4.5 that $\Phi_A(a, \delta) \subseteq s'$. As $\Phi_A(a, \delta)$ belongs to $\Phi_A S(a, \delta)$, we can conclude the proposition.

Now suppose $a$ is a sensing action. Since we assume that in every state of the world, exactly one fluent literal in $\theta$ holds, there exists a fluent literal $g \in \theta$ such that $g$ holds in $s$ and for all $g' \in \theta \setminus \{g\}$, $g'$ does not hold in $s$. Accordingly, we have $\delta \cup \{g\} \subseteq s$. By Corollary D.1, we have

$$\delta' = Cl_D(\delta \cup \{g\}) \subseteq Cl_D(s) = s$$
Hence, $\delta'$ is consistent and valid. By the definition of the $\Phi^A_S$ function, we have $\delta' \in \Phi^A_S(a, \delta)$ and thus we can conclude the proposition.

**F.1.2 Proof of Proposition 5.2**

The proof will make use of the result in Proposition 5.1. We will prove the proposition by using structural induction on $p$.

1. $\alpha = \langle \rangle$. Trivial.

2. $\alpha = \langle a, \beta \rangle$, where $\beta$ is a conditional plan and $a$ is a non-sensing action.

   Assume that the statement of the proposition is true for $\beta$. We need to prove that it is also true for $\alpha$.

   As $\hat{\Phi}^A_S(\alpha, \delta) \neq \bot$, it follows from the definition of $\hat{\Phi}^A_S(\alpha, \delta)$ that $\Phi^A_S(a, \delta) \neq \bot$.

   On the other hand, since $\delta$ is valid, it follows from Proposition 5.1 that $\Phi^A_S(a, \delta) = \{\delta'\}$ for some valid partial state $\delta'$.

   By the inductive hypothesis, we have $\hat{\Phi}^A_S(\alpha, \delta) = \hat{\Phi}^A_S(\beta, \delta')$ contains at least one valid partial state.

3. $\alpha = \langle a, \text{cases}(\{g_j \rightarrow \beta_j\}_{j=1}^n) \rangle$, where $a$ is a sensing action that senses $g_1, \ldots, g_n$.

   Assume that the proposition is true for $\beta_j$'s. We need to prove that it is also true for $\alpha$.

   Because $\hat{\Phi}^A_S(\alpha, \delta) \neq \bot$, we have $\Phi^A_S(a, \delta) \neq \bot$. As $\delta$ is valid, by Proposition 5.1, $\Phi^A_S(a, \delta)$ contains at least one valid partial state $\delta'$. 

309
By the definition of the $\Phi^A_S$ function for sensing actions, we know that $\delta' = Cl_D(\delta \cup \{g_k\})$ for some $1 \leq k \leq n$. It is easy to see that $\hat{\Phi}^A_S(\beta_k, \delta') \neq \bot$ for if otherwise, $\hat{\Phi}^A_A(\alpha, \delta') = \bot$. Hence, by the inductive hypothesis, we have $\hat{\Phi}^A_S(\beta_k, \delta')$ contains at least one valid partial state.

By the definition of the $\hat{\Phi}^A_S$ function, we have $\hat{\Phi}^A_S(\beta_k, \delta') \subseteq \hat{\Phi}^A_A(\alpha, \delta)$. Thus, $\hat{\Phi}^A_A(\alpha, \delta)$ contains at least one valid partial state.

F.1.3 Proof of Proposition 5.3

From results in Chapter 4 we know that if $a$ is a non-sensing action then computing $\Phi^A(a, \delta)$, thus $\Phi^A_S(a, \delta)$ as well, can be performed in polynomial time. On the other hand, we know that the closure of a set of fluent literals can also be computed in polynomial time. As a result, if $a$ is a sensing action then $\Phi^A_S(a, \delta)$ can also be computed in polynomial time.

Hence, the proposition is true.

F.1.4 Proof of Theorem 5.1

The proof is similar to the proof of Theorem 3 in [13] which states that the conditional planning problem with respect to the 0-approximation in [126] is NP-complete. Membership follows from Corollary 5.1. Hardness follows from the fact that the approximations proposed in this chapter coincide with the 0-approximation in [126], i.e, the conditional planning problem considered in this chapter coincides with the planning problem with limited-sensing in [13] which is NP-complete. By the restriction
principle, we conclude that the conditional planning problem of our interest is also NP-complete.

F.2 Proofs of Results in Section 5.4

This section contain proofs related to the correctness of $\pi_{h,w}(P)$. We will use similar notations as in the proofs of results in Chapter 4 (see Appendix D). Furthermore, given a program $\Pi$, by $\text{lit}(\Pi)$ we mean the set of literals in $\Pi$.

Lemma F.1 Let $\Pi$ be a logic program. Then, we have the following results.

1. Suppose $\Pi$ can be divided into two disjoint subprograms $\Pi_1$ and $\Pi_2$, i.e., $\Pi = \Pi_1 \cup \Pi_2$ and $\text{lit}(\Pi_1) \cap \text{lit}(\Pi_2) = \emptyset$. Then $S$ is an answer set for $\Pi$ if and only if there exist two sets $S_1$ and $S_2$ of atoms such that $S = S_1 \cup S_2$ and $S_1$ and $S_2$ are answer sets for $\Pi_1$ and $\Pi_2$ respectively.

2. The result in Item 1 can be generalized to $n$ disjoint subprograms, where $n$ is an arbitrary integer.

Proof. The first item can easily proved by using the splitting set $Z = \text{lit}(\Pi_1)$. The second item immediately follows from this result.

Next, it is easy to see that the following lemma, which is similar to Lemma D.4, holds.
Lemma F.2 Let \( i \) and \( k \) be two integers greater than 0, and \( x \) be a 3-ary predicate. For any set \( \sigma \) of fluent literals, the following program

\[
x(l, i, k) \leftarrow (l \in \sigma)
\]

\[
x(l, i, k) \leftarrow x(\psi, i, k) \quad ([l \text{ if } \psi] \in \mathcal{D})
\]

has the unique answer set \( \{x(l, i, k) \mid l \in \text{Cl}_\mathcal{D}(\sigma)\} \).

Note that we are making the following assumptions: (a) for every knowledge producing law

\[ c \text{ determines } \theta \]

in \( \mathcal{D} \), \( \theta \) contains at least two elements; and (b) for every static causal law

\[ l \text{ if } \psi \]

in \( \mathcal{D} \), \( \psi \) is not an empty set. These assumptions are important for the correctness of the proofs that are given below.

F.2.1 Proof of Theorem 5.2

Suppose we are given a planning instance \( P = (\mathcal{D}, \delta, \mathcal{G}) \) and the program \( \pi_{h,w}(P) \) \((h \geq 1 \text{ and } w \geq 1)\) returns an answer set \( S \). The proof, which is primarily based on the splitting set and splitting sequence theorems described in Section 2.4.4, is organized as follows. We will use Lemmas F.1 and F.2 (presented in the beginning of this section) to prove some properties of the program \( \pi_{h,w}(P) \) (Lemmas F.3, F.4 & F.5). Based on these results, we will prove the correctness of \( \pi_{h,w}(P) \) in implementing the
\( \Phi^\text{pc}_S \) and \( \Phi^\text{pc}_S \) functions (Lemmas F.6 & F.7). Theorem 5.2 can be derived directly from Lemma F.7.

Similarly as with CPASP, in the proof of the soundness of ASCP, we first translate the program \( \pi_{h,w}(P) \) into an “equivalent” program without choice rules by replacing the rules of the form (78) with

\[
br(g, t, p, p) | \ldots
\]

\[
| br(g, t, p, w) \leftarrow occ(c, t, p)
\]

(\( [\; c \text{ determines } \theta \;] \in D, g \in \theta \))

\[
\leftarrow occ(c, t, p), br(g, t, p, p_1), br(g, t, p, p_2)
\]

(\( [\; c \text{ determines } \theta \;] \in D, g \in \theta, p \leq p_1 < p_2 \))

and the rules of the form (94) with

\[
occ(E, T, P) | \neg occ(E, T, P) \leftarrow used(T, P), not\ goal(T, P)
\]

\[
\leftarrow \neg occ(A, T, P), used(T, P), not\ goal(T, P)
\]

This translation does not affect the consistency of the program as well as the presence of positive literals, which are of our interest, in answer sets. As a result, we will not distinguish between the original program and the new program and still use the same notation \( \pi_{h,w}(P) \) to refer to both of them. Furthermore, we will ignore the literals of the form \( occ(\ldots) \) in the answer sets for the new program. Next, we divide \( \pi_{h,w}(P) \) into two programs \( \pi^*_{h,w}(P) \) and \( \pi^\text{c}_{h,w}(P) \) which consist of normal logic program rules and
constraints in $\pi_{h,w}(P)$ respectively. Then we use the splitting set theorem to remove from $\pi^*_h, w(P)$ auxiliary atoms such as fluent(...), literal(...), time(...), path(...), etc. The resulting program, denoted by $\pi_0$, consists of “main” atoms only. We then use the splitting sequence theorem to split $\pi_0$ into a set of programs $\pi_i$’s each of which, intuitively, corresponds to a “cut” of $\pi_0$ at a specific time step. Finally, each $\pi_i$ is further split into disjoint subprograms $\pi^{k_i}_i$’s corresponding cuts of $\pi_i$ at paths.

For $1 \leq i \leq h + 1$ and $1 \leq k \leq w$, let $A_{i,k}$ be the set of atoms of the form $occ(e, i, k)$, $used(i, k)$, $goal(i, k)$, $holds(l, i, k)$, $br(g, i, k, k') (k' \geq k)$, $ef(l, i, k)$, $pc(l, i, k)$, i.e.,

$$A_{i,k} = \{occ(e, i, k) \mid e \in A\} \cup$$

$$\{holds(l, i, k), ef(l, i, k), pc(l, i, k) \mid l \text{ is a fluent literal}\} \cup$$

$$\{br(g, i, k, k') \mid g \text{ is a sensed literal, } k \leq k' \leq w\} \cup$$

$$\{used(i, k), goal(i, k)\}$$

(204)

and let

$$A_i = \bigcup_{k=1}^{w} A_{i,k}, \quad A = \bigcup_{i=1}^{h+1} A_i$$

(205)

For a set of atoms $\Sigma \subseteq A$ and a set of predicate symbols $X$, by $\Sigma^X$ we denote the set of atoms in $\Sigma$ whose predicate symbols are in $X$ and by $\delta_{i,k}(\Sigma)$, we mean $\{l \mid holds(l, i, k) \in \Sigma\}$. Let $V$ be the set of atoms in $\pi_{h,w}(P)$ whose parameter list does not contain
either the time or path variable. Specifically, $V$ is the following set of atoms

$$\{\text{fluent}(f), \text{literal}(f), \text{literal}(\neg f), \text{contrary}(f, \neg f), \text{contrary}(\neg f, f) \mid f \in F\}$$

$$\cup \{\text{sensed}(g) \mid \exists [c \text{ determines } \theta] \in \mathcal{D}.g \in \theta\}$$

$$\cup \{\text{nonsensing}(e) \mid \text{is a non-sensing action}\}$$

$$\cup \{\text{sensing}(e) \mid \text{is a sensing action}\} \cup \{\text{action}(e) \mid e \in A\}$$

$$\cup \{\text{time}(t) \mid t \in \{1..h\}\} \cup \{\text{time1}(t) \mid t \in \{1..h + 1\}\}$$

$$\cup \{\text{path}(p) \mid p \in \{1..w\}\}$$

(206)

It is easy to see that $V$ is a splitting set for $\pi_{h,w}^*(P)$. Furthermore, the bottom part $b_V(\pi_{h,w}^*(P))$ is a positive program and has only one answer set $X_0 = V$. The partial evaluation of the top part of $\pi_{h,w}^*(P)$ with respect to $X_0$,

$$\pi_0 = e_V(\pi_{h,w}^*(P) \setminus b_V(\pi_{h,w}^*(P)), X_0),$$

is the following set of rules (the condition for each rule follows that rule, where, unless otherwise stated explicitly, $t$ ranges over $1 \ldots h$; $p$ ranges over $1 \ldots w$; $f$ ranges over the set of fluents; $l$ ranges over the set of fluent literals; $g$ ranges over the set of sensed literals; and $e$ ranges over the set of elementary actions):
holds(l, 1, 1) \leftarrow \hspace{1cm} (l \in \delta) \hspace{1cm} (207)

e f(l, t, p) \leftarrow \text{occ}(e, t, p), \text{holds}(\psi, t, p) \hspace{1cm} (208)

\text{[[ e causes l if } \psi \text{ ]} \in \mathcal{D})

\text{pc}(l, t, p) \leftarrow \text{occ}(e, t, p), \text{not holds}(l, t, p), \text{not holds}(\neg \psi, t, p) \hspace{1cm} (209)

\text{[[ e causes l if } \psi \text{ ]} \in \mathcal{D})

br(g, t, p, p) \mid \ldots \mid \text{br}(g, t, p, w) \leftarrow \text{occ}(c, t, p) \hspace{1cm} (210)

\text{[[ c determines } \theta \text{ ]} \in \mathcal{D}, g \in \theta)

\text{pc}(l, t, p) \leftarrow \text{not holds}(l, t, p), \text{pc}(l', t, p), \text{not ef}(\neg \psi, t, p) \hspace{1cm} (211)

\text{[[ l if } \psi \text{ ]} \in \mathcal{D}, l' \in \psi)

e f(l, t, p) \leftarrow \text{ef}(\psi, t, p) \hspace{1cm} (212)

\text{[[ l if } \psi \text{ ]} \in \mathcal{D})

\text{holds}(l, t, p) \leftarrow \text{holds}(\psi, t, p) \hspace{1cm} (213)

\text{[[ l if } \psi \text{ ]} \in \mathcal{D}, 1 \leq t \leq h + 1)
\[
\text{goal}(t, p) \leftarrow \text{holds}(G, t, p)
\]
\[
(1 \leq t \leq h + 1)
\]
\[
\text{goal}(t, p) \leftarrow \text{holds}(f, t, p), \text{holds}(\neg f, t, p)
\]
\[
(1 \leq t \leq h + 1)
\]
\[
\text{holds}(l, t+1, p) \leftarrow ef(l, t, p)
\]
\[
(216)
\]
\[
\text{holds}(l, t+1, p) \leftarrow h(l, t, p), \text{not pc}(\neg l, t, p)
\]
\[
(217)
\]
\[
\text{used}(t+1, p) \leftarrow br(g, t, p_1, p)
\]
\[
(p_1 < p)
\]
\[
(218)
\]
\[
\text{holds}(g, t+1, p) \leftarrow br(g, t, p_1, p)
\]
\[
(219)
\]
\[
(\text{holds}(l, t+1, p) \leftarrow br(g, t, p_1, p), \text{holds}(l, t, p_1))
\]
\[
(p_1 \leq p)
\]
\[
(220)
\]
\[
\text{occ}(e, t, p) | \neg \text{occ}(e, t, p) \leftarrow \text{used}(t, p), \text{not goal}(t, p)
\]
\[
(221)
\]
\[
\text{used}(1, 1) \leftarrow
\]
\[
(222)
\]
\[
\text{used}(t+1, p) \leftarrow \text{used}(t, p)
\]
\[
(223)
\]
And $\pi_{h,w}^c(P)$ is the following collection of constraints

\[ \leftarrow \ \text{occ}(b, t, p), \text{not holds}(\neg \psi, t, p) \] (224)

(\text{impossible } b \text{ if } \psi) \in D

\[ \leftarrow \ \text{occ}(c, t, p), \text{not br}(\theta, t, p, p) \] (225)

(\text{c determines } \theta) \in D

\[ \leftarrow \ \text{occ}(c, t, p), \text{br}(g, t, p, p_1), \text{br}(g, t, p, p_2) \] (226)

(\text{c determines } \theta \in D, g \in \theta, p \leq p_1 < p_2)

\[ \leftarrow \ \text{occ}(c, t, p), \text{holds}(g, t, p) \] (227)

(\text{c determines } \theta \in D, g \in \theta)

\[ \leftarrow \ \text{used}(h+1, p), \text{not goal}(h+1, p) \] (228)

\[ \leftarrow \ \text{br}(g_1, t, p_1, p), \text{br}(g_2, t, p_2, p) \] (229)

(p_1 < p_2 < p)

\[ \leftarrow \ \text{br}(g_1, t, p_1, p), \text{br}(g_2, t, p_1, p) \] (230)

(g_1 \neq g_2, p_1 \leq p)

\[ \leftarrow \ \text{br}(g, t, p_1, p), \text{used}(t, p) \] (231)

(p_1 < p)

\[ \leftarrow \ \text{not occ}(A, T, P), \text{used}(T, P), \text{not goal}(T, P) \] (232)

\[ \leftarrow \ \text{occ}(se, t, p), \text{occ}(e, t, p) \] (233)

(se \neq e)
By the splitting set theorem, as $S$ is the answer set for $\pi_{h,w}(D)$, there exists an answer set $S_0$ for $\pi_0$ such that

$$S = S_0 \cup X_0$$  \hspace{1cm} (234)

Let $U_i$ be the set of atoms in $\pi_0$ whose time parameter is less than or equal to $i$, i.e.,

$$U_i = \bigcup_{j=1}^{i} A_j$$  \hspace{1cm} (235)

It is easy to see that the sequence $\langle U_i \rangle_{i=1}^{h+1}$ is a splitting sequence for $\pi_0$. By the splitting sequence theorem, since $S_0$ is an answer set for $\pi_0$, there must be a sequence of sets of literals $\langle X_i \rangle_{i=1}^{h+1}$ such that $X_i \subseteq U_i \setminus U_{i-1}$, and

- $S_0 = \bigcup_{i=1}^{h+1} X_i$

- $X_1$ is an answer set for

$$\pi_1 = b_{U_1}(\pi_0)$$  \hspace{1cm} (236)

- for every $1 < i \leq h + 1$, $X_i$ is an answer set for

$$\pi_i = e_{U_i}(b_{U_i}(\pi_0) \setminus b_{U_{i-1}}(\pi_0), \bigcup_{1 \leq t \leq i-1} X_t)$$  \hspace{1cm} (237)
Given a set of atoms $\Sigma$, consider rules of the following forms:

$$holds(l, 1, 1) \leftarrow$$  \hspace{0.5cm} (238)

$$ \quad (l \in \delta)$$

$$ef(l, t, p) \leftarrow occ(e, t, p), holds(\psi, t, p)$$  \hspace{0.5cm} (239)

$$\quad ([ \text{e causes } l \text{ if } \psi ] \in \mathcal{D})$$

$$pc(l, t, p) \leftarrow occ(e, t, p), not holds(l, t, p),$$

$$not holds(\neg \psi, t, p)$$  \hspace{0.5cm} (240)

$$\quad ([ \text{e causes } l \text{ if } \psi ] \in \mathcal{D})$$

$$br(g, t, k, p) \mid \ldots$$

$$br(g, t, k, w) \leftarrow occ(c, t, p)$$  \hspace{0.5cm} (241)

$$\quad ([ \text{c determines } \theta ] \in \mathcal{D}, g \in \theta)$$

$$pc(l, t, p) \leftarrow not holds(l, t, p), pc(l', t, p), not ef(\neg \psi, t, p)$$  \hspace{0.5cm} (242)

$$\quad ([ l \text{ if } \psi ] \in \mathcal{D}, l' \in \psi)$$

$$ef(l, t, p) \leftarrow ef(\psi, t, p)$$  \hspace{0.5cm} (243)

$$\quad ([ l \text{ if } \psi ] \in \mathcal{D})$$

$$holds(l, t, p) \leftarrow holds(\psi, t, p)$$  \hspace{0.5cm} (244)

$$\quad ([ l \text{ if } \psi ] \in \mathcal{D})$$

$$goal(t, p) \leftarrow holds(\mathcal{G}, t, p)$$  \hspace{0.5cm} (245)

$$\quad holds(f, t, p), holds(\neg f, t, p)$$  \hspace{0.5cm} (246)
holds(l, t, p) ← 
(ef(l, t−1, p) ∈ Σ)  

holds(l, t, p) ← 
(holds(l, t−1, p) ∈ Σ, pc(¬l, t−1, p) ∉ Σ)  

used(t, p) ← 
(∃⟨g, p’⟩.p’ < p ∧ br(g, t−1, p’, p) ∈ Σ)  

holds(g, t, p) ← 
(∃⟨g, p’⟩.p’ ≤ p ∧ br(g, t−1, p’, p) ∈ Σ)  

holds(l, t, p) ← 
∃⟨g, p’⟩.p’ < p ∧ br(g, t−1, p’, p) ∈ Σ ∧ 
holds(l, t−1, p’) ∈ Σ)  

occ(e, t, p) | ¬occ(e, t, p) ← used(t, p), not goal(t, p)  

used(1, 1) ← 
used(t, p) ← 
(used(t−1, p) ∈ Σ)  

Then for each i ∈ {1, . . . , h + 1}, πi can be divided into w disjoint subprograms 

πi,k, 1 ≤ k ≤ w, where πi,k is defined as follows 

$$
π_i^k = \begin{cases} 
{(238) - (246), (252) - (253)} & | t = 1, p = 1 
\text{ if } i = 1, k = 1 \\
{(239) - (246), (252)} & | t = 1, p = k 
\text{ if } i = 1, k > 1 \\
{(239) - (254)} & | t = i, p = k, \Sigma = X_{i-1} 
\text{ if } 1 < i \leq h \\
{(244) - (251), (254)} & | t = h + 1, p = k, \Sigma = X_{h} 
\text{ otherwise} 
\end{cases}
$$
Let $X_{i,k}$ denote $X_i \cap A_{i,k}$. From Lemma F.1, it follows that $X_{i,k}$ is an answer set for $\pi^k_i$. Hence, we have

$$\delta_{i,k}(S) = \delta_{i,k}(S_0) = \delta_{i,k}(X_i) = \delta_{i,k}(X_{i,k})$$

Due to this fact, from now on, we will use $\delta_{i,k}$ to refer to either $\delta_{i,k}(S)$, $\delta_{i,k}(S_0)$, $\delta_{i,k}(X_i)$, or $\delta_{i,k}(X_{i,k})$.

The relationship among subprograms of $\pi_{h,w}(P)$ and the relationship among their answer sets are depicted in Figure F.1. We have the following lemma.

**Lemma F.3** For $1 \leq i \leq h + 1$ and $1 \leq k \leq w$, 

322
1. If \( \text{used}(i, k) \notin S \) then \( S \) does not contain either \( \text{holds}(l, i, k) \), \( \text{ef}(l, i, k) \), or \( \text{br}(g, i, k, k) \);

2. If \( \text{used}(h + 1, k) \in S \) and \( \delta_{h+1,k} \) is consistent then \( G \) is true in \( \delta_{h+1,k} \).

**Proof.**

1. We will use induction on \( i \) to prove this item.

   a. **Base case:** \( i = 1 \). Let \( k \) be an integer such that \( \text{used}(1, k) \notin S \). Clearly we have \( k > 1 \). We can easily show that if \( k > 1 \) then the only answer set of the program \( \pi_k^1 \) (see (255)) is the empty set (note that we are assuming \( G \neq \emptyset \) and the body of every static causal law is non-empty). Hence, \( X_{1,k} \), and thus \( S \) as well do not contain any of the above atoms, i.e., the base case is true.

   b. **Inductive step:** Assume that Item 1 is true for \( i \leq j - 1 \), where \( j > 1 \). We will prove that it is also true for \( i = j \). Let \( k \) be an integer such that \( \text{used}(j, k) \notin S \).

   Clearly, to prove Item 1 we only need to prove that atoms of the forms \( \text{ef}(l, j, k) \), \( \text{holds}(l, j, k) \), \( \text{br}(g, j, k, k) \) do not belong to \( X_{j,k} \). Consider the program \( \pi_j^k \). We know that \( X_{j,k} \) is an answer set for \( \pi_j^k \).

   Because of rule (254), we have \( \text{used}(j - 1, k) \notin X_{j-1} \). From (249), it follows that \( \text{br}(g, j - 1, k', k) \notin X_{j-1} \) for every pair \( \langle g, k' \rangle \) such that \( k' < k \).
In addition, by the inductive hypothesis, we have that for any $l$ and $g$, $ef(l, j-1, k)$, $holds(l, j-1, k)$, and $br(g, j-1, k, k)$ are not in $X_{j-1}$. As a result, rules of the forms (247)-(251) do not exist in $\pi^k_j$. If we split $\pi^k_j$ by the set $Z = A_{j,k}^{\{\text{holds,ef,br,occ,\neg occ,used,goal}\}}$ then $b_Z(\pi^k_j)$ is the set of rules of the forms (with $t = i$ and $p = k$)

i. (239), (241), (243)-(245), (252) if $i \leq h$

ii. (244)-(245) if $i = h + 1$

It is not difficult to show that this program has the empty set as its only answer set (recall that we are assuming $G \neq \emptyset$ and the body of every static causal law is non-empty). Hence, we can conclude the inductive step.

2. It is obvious because of the rules (214), (215) and the constraint (228).

$$\square$$

For $1 \leq i \leq h$ and $1 \leq k \leq w$, let

$$a_{i,k} = \{e \mid \text{occ}(e, i, k) \in S\}$$ (256)

Then, we have the following lemma.

**Lemma F.4** If $a_{i,k} \neq \emptyset$ then

1. $a_{i,k}$ is an action, i.e., it is either a non-sensing action, or a sensing action, and

2. $a_{i,k}$ is executable in $\delta_{i,k}$.
Proof. The first item holds by constraint (233). The second item holds by constraint (224).

Lemma F.5 For $1 \leq i \leq h$ and $1 \leq k \leq w$

1. if $a_{i,k}$ is a non-sensing action then
   
   a. $ef(l, i, k) \in S$ iff $l \in E(a_{i,k}, \delta_{i,k})$
   
   b. $pc(l, i, k) \in S$ iff $l \in pc(a_{i,k}, \delta_{i,k})$
   
   c. $\neg \exists (g, k'). br(g, i, k', k) \in S$

2. if $a_{i,k}$ is a sensing action associated with the knowledge-producing law of the form

   \[ a_{i,k} \text{ determines } g_1, \ldots, g_n \]

   in $D$ then there exist $n$ distinct integers $k_1, \ldots, k_n$ greater than or equal to $k$ such that

   a. \[ X_{i,k}^{br} = \{ br(g_j, i, k, k_j) \mid j \in \{1, \ldots, n\} \}, \]

   b. $g_j$ does not hold in $\delta_{i,k}$.

   c. if $k_j > k$ then $S$ does not contain any atom of the form holds($l, i, k_j$), and

3. if $a_{i,k} = \emptyset$ then

   a. $\forall l. pc(l, i, k) \notin S \land ef(l, i, k) \notin S$
b. \( \forall(g, k').br(g, i, k, k') \not\in S \)

**Proof.** Let us split \( \pi_i^k \) by the set \( Z_1 = A_{i,k}^{\{used,goal,occ,\neg occ,holds\}} \). By the splitting set theorem, \( X_{i,k} = M \cup N \) where \( M \) is an answer set for \( b_{Z_1}(\pi_i^k) \) and \( N \) is an answer set for \( \Pi_1 = e_{Z_1}(\pi_i^k \setminus b_{Z_1}(\pi_i^k), M) \), which consists of the following rules

\[
e f(l, i, k) \leftarrow (\text{occ}(e, i, k) \in M, [ \text{e causes } l \text{ if } \psi ] \in D, \text{holds}(\psi, i, k) \subseteq M) \quad (257)
\]

\[
\text{pc}(l, i, k) \leftarrow (\text{occ}(e, i, k) \in M, \text{holds}(l, i, k) \not\in M, [ \text{e causes } l \text{ if } \psi ] \in D, \text{holds}(\neg \psi, i, k) \cap M = \emptyset) \quad (258)
\]

\[
br(g, i, k, k) | \ldots \quad (259)
\]

\[
br(g, i, k, w) \leftarrow (\text{occ}(c, i, k) \in M, [ \text{c determines } \theta ] \in D, g \in \theta) \quad (260)
\]

\[
\text{pc}(l, i, k) \leftarrow \text{pc}(l', i, k), \text{not ef}(\neg \psi, i, k) \quad (260)
\]

\[
([ l \text{ if } \psi ] \in D, \text{holds}(l, i, k) \not\in M, l' \in \psi) \quad (261)
\]

\[
e f(l, i, k) \leftarrow e f(\psi, i, k) \quad (261)
\]

\[
([ l \text{ if } \psi ] \in D)
\]

By the splitting set theorem, we have

\[ \delta_{i,k}(M) = \delta_{i,k} \]
1. Since $a_{i,k}$ is a non-sensing action, by Lemma F.4, we know that there exists no sensing action $c$ such that $\text{occ}(c, i, k) \in S$. This means that rules of form (259) does not exist. Therefore, $\Pi_1$ can be rewritten to

\[
e f(l, i, k) \leftarrow ([e \text{ causes } l \text{ if } \psi] \in D, \text{holds}(\psi, i, k) \subseteq M)\]

\[
pc(l, i, k) \leftarrow
\]

\[(\text{holds}(l, i, k) \not\in M, [e \text{ causes } l \text{ if } \psi] \in D, \text{holds}(\neg\psi, i, k) \cap M = \emptyset)\]

\[
pc(l, i, k) \leftarrow pc(l', i, k), \text{not } ef(\neg\psi, i, k)
\]

\[(\text{[l if } \psi] \in D, \text{holds}(l, i, k) \not\in M, l' \in \psi)\]

\[
e f(l, i, k) \leftarrow ef(\psi, i, k)
\]

\[(\text{[l if } \psi] \in D)\]

If we continue splitting the above program using $Z_2 = A_{i,k}^{\{e\}}$ then by Lemma F.2, the bottom part has the only answer set

\[
\{ef(l, i, k) \mid l \in E(a_{i,k}, \delta_{i,k})\}
\]

and the evaluation of the top part has the only answer set

\[
\{pc(l, i, k) \mid l \in pc(a_{i,k}, \delta_{i,k})\}
\]

As $M$ does not contain any atom of the form $ef(l, i, k)$ or $pc(l, i, k)$, we can conclude Items (a) and (b).
We now show that \( \neg \exists \langle g, k' \rangle. br(g, i, k', k) \in S \). Suppose otherwise, i.e., there exists \( g \) and \( k' \) such that \( br(g, i, k', k) \in S \). Notice that only rule (221) with \( t = i \) and \( p = k \) has \( occ(e, i, k) \) for some elementary action \( e \) in its head. Hence, its body must be satisfied by \( S \). That implies \( used(i, k) \in S \).

On the other hand, since only rules of the form (210) with \( p = k' \) may have \( br(g, i, k', k) \) in their heads, there is a sensing action \( c \) such that \( occ(c, i, k') \in S \) and in addition, \( k' \leq k \). As the sets of non-sensing actions and sensing actions are disjoint from each other, we have \( c \notin a_{i,k} \). From Lemma F.4, it follows that \( k' \neq k \), implying \( k' < k \).

Accordingly, we have \( used(i, k) \in S, br(g, i, k', k) \in S \) and \( k' < k \). Constraint (231) with \( t = i, p = k \), and \( p_1 = k' \) is thus violated. Thus, Item (c) holds.

2. Observe that in this case, rules of the forms (257) and (258) do not exist. There-
fore, \( \Pi_1 \) is the following set of rules

\[
\begin{align*}
\text{br}(g_1, i, k, k) & \mid \ldots \\
\text{br}(g_1, i, k, w) & \leftarrow \\
\ldots & \ldots \ldots \\
\text{br}(g_n, i, k, k) & \mid \ldots \\
\text{br}(g_n, i, k, w) & \leftarrow \\
\text{pc}(l, i, k) & \leftarrow \text{pc}(l', i, k), \text{not } \text{ef}(\neg \psi, i, k) \\
& \quad \left( ([ \text{if } \psi ] \in \mathcal{D}, \text{holds}(l, i, k) \notin M, l' \in \psi) \right) \\
\text{ef}(l, i, k) & \leftarrow \text{ef}(\psi, i, k) \\
& \quad \left( ([ \text{if } \psi ] \in \mathcal{D}) \right)
\end{align*}
\]

By further splitting the above program using the set \( A^\{\text{ef,pc}\}_{i,k} \), we will see that the bottom part has the empty set as its only answer set (recall that we are assuming that the body of every static causal law is not empty). Therefore, the answer set for the above program is also the answer set for the following program and vice
versa.

\[ br(g_1, i, k, k) \mid \ldots \]
\[ br(g_1, i, k, w) \leftarrow \]
\[ \ldots \ldots \ldots \]
\[ br(g_n, i, k, k) \mid \ldots \]
\[ br(g_n, i, k, w) \leftarrow \]

Thus, there exist \( n \) integers \( k_1, \ldots, k_n \) greater than or equal to \( k \) such that

\[ N = \bigcup_{j=1}^{n} \{br(g_j, i, k, k_j)\} \]

It is easy to see that \( X^{\{br\}}_{i,k} = N^{\{br\}} \). In addition, by constraints of the form (230), \( k_j \)'s must be distinct. Thus, Items (a) is true.

Item (b) can be drawn from constraints of the form (227).

Assume \( k_j > k \). Because of constraints of the form (231), we have \( used(i, k_j) \notin S \). From Lemma F.3, it follows that \( S \) does not contain any atoms of the form \( holds(l, i, k_j) \). Item (c) is thus true.

3. It is easy to see that if \( a_{i,k} = \emptyset \) then \( \Pi_1 \) is the following set of rules

\[ pc(l, i, k) \leftarrow pc(l', i, k), not \ ef(\neg\psi, i, k) \]
\[ \text{if } [l \ if \ \psi] \in D, holds(l, i, k) \notin M, l' \in \psi \]
\[ ef(l, i, k) \leftarrow ef(\psi, i, k) \]
\[ \text{if } [l \ if \ \psi] \in D \]

330
which has an empty set as its only answer set. Items (a)-(b) follow from this.

The following lemma shows that the program $\pi_{h,w}(P)$ correctly implements the transition function $\Phi_{pc}^s$.

**Lemma F.6** For $1 \leq i \leq h$ and $1 \leq k \leq w$, the following results hold.

1. If $a_{i,k}$ is a non-sensing action then

   $$\Phi_{pc}^s(a_{i,k}, \delta_{i,k}) = \begin{cases} \emptyset & \text{if } \delta_{i+1,k} \text{ is inconsistent} \\ \{\delta_{i+1,k}\} & \text{otherwise} \end{cases}$$

2. If $a_{i,k}$ is a sensing action associated with a knowledge-producing law of the form $a_{i,k}$ determines $g_1, \ldots, g_n$ in $D$ then there exist $n$ integers $\{k_1, \ldots, k_n\}$ such that

   $$\Phi_{pc}^s(a_{i,k}, \delta_{i,k}) = \{\delta_{i+1,k_j} \mid 1 \leq j \leq n, \delta_{i+1,k_j} \text{ is consistent}\},$$

   and for each $j$, $g_j$ holds in $\delta_{i+1,k_j}$

3. If $a_{i,k} = \emptyset$ then

   $\delta_{i+1,k} = \delta_{i,k}$

**Proof.**

1. Observe that $Z_1 = A_{i+1,k}^{\text{holds}}$ is a splitting set for $\pi_{i+1}^k$. Hence, by the splitting set theorem, $X_{i+1,k} = M \cup N$, where $M \subseteq Z_1$ is an answer set for $\Pi_1 = b_{Z_1}(\pi_{i+1}^k)$ and $N$ is an answer set for $\Pi_2 = c_{Z_1}(\pi_{i+1}^k \setminus \Pi_1, M)$. 331
Notice that by Lemma F.5, rules (250)-(251) for $t = i + 1, p = k$ do not exist.

Thus, $\Pi_1$ is the following set of rules:

$$holds(l, i+1, k) \leftarrow holds(\psi, i+1, k)$$

$$([l \text{ if } \psi] \in D)$$

$$holds(l, i+1, k) \leftarrow$$

$$ef(l, i, k) \in X_i$$

$$holds(l, i+1, k) \leftarrow$$

$$(holds(l, i, k) \in X_i, pc(\neg l, i, k) \notin X_i)$$

Also by Lemma F.5, the conditions for the second and third rules can be written as $(l \in E(a_{i,k}, \delta_{i,k}))$ and $(l \in \delta_{i,k}, \neg l \notin pc(a_{i,k}, \delta_{i,k}))$ respectively. Thus, by Lemma F.2, $\Pi_1$ has the unique answer set

$$M = \{ holds(l, i+1, k) \mid l \in Cl_D(a_{i,k}, \delta_{i,k}) \}$$

On the other hand, by Lemma F.4, $a_{i,k}$ is executable in $\delta_{i,k}$. By the definition of $\Phi^pc_S$, it follows that

$$\Phi^pc_S(a_{i,k}, \delta_{i,k}) = \begin{cases} \emptyset & \text{if } \delta_{i+1,k} \text{ is inconsistent} \\ \{\delta_{i+1,k}\} & \text{otherwise} \end{cases}$$

2. By Lemma F.5, for each $j \in \{1 \ldots n\}$, there exists $k_j \geq k$ such that $br(g_j, i, k, k_j) \in X_i$. It is easy to see that $Z_2 = A^{(holds)}_{i+1,k_j}$ is a splitting set for $\pi_{i+1}^{k_j}$. Considering cases $k_j = k$ and $k_j > k$ in turn and observe that $holds(l, i, k_j) \notin S$ if $k_j > k$,
we will see that in both cases $b_{Z_2}(\pi_{i+1}^{k_j})$ is the following set of rules:

$$
holds(l, i+1, k_j) \leftarrow holds(\psi, i+1, k_j)
$$

$$
\quad (\lfloor l \text{ if } \psi \rfloor \in D)
$$

$$
holds(l, i+1, k_j) \leftarrow
$$

$$
\quad (holds(l, i, k) \in X_i)
$$

$$
holds(g_j, i+1, k_j) \leftarrow
$$

By Lemma F.2, the only answer set for the above program is

$$
M = \{holds(l, i + 1, k_j) \mid l \in Cl_D(\delta_{i,k} \cup \{g_j\})\}
$$

On the other hand, by Lemma F.4, $a_{i,k}$ is executable in $\delta_{i,k}$ and by Lemma F.5, $g_j$ does not hold in $\delta_{i,k}$. Thus, according to the definition of the transition function, we have

$$
\Phi_{pc}^{S}(a, \delta_{i,k}) = \{Cl_D(\delta_{i,k} \cup \{g_j\}) \mid 1 \leq j \leq n, Cl_D(\delta_{i,k} \cup \{g_j\}) \text{ is consistent}\}
$$

Hence, we have

$$
\Phi_{pc}^{S}(a, \delta_{i,k}) = \{\delta_{i+1,k_j}(M) \mid 1 \leq j \leq n, \delta_{i+1,k_j}(M) \text{ is consistent}\} =
$$

$$
\{\delta_{i+1,k_j} \mid 1 \leq j \leq n, \delta_{i+1,k_j} \text{ is consistent}\}
$$

and obviously, $g_j$ holds in $\delta_{i+1,k_j}$. 

333
3. Similar to the first case, $Z_1$ is a splitting set for $\pi_{i+1}^k$. It is easy to see that the program $b_{Z_1}(\pi_{i+1}^k)$ is the following set of rules (observe Lemma F.5):

$$holds(l, i+1, k) \leftarrow holds(\psi, i+1, k)$$

$$([ l \text{ if } \psi ] \in \mathcal{D})$$

$$holds(l, i+1, k) \leftarrow$$

$$(holds(l, i, k) \in X_i)$$

By Lemma F.2 the only answer set for this program is

$$M = \{holds(l, i + 1, k) \mid l \in \delta_{i,k}\}$$

Thus, we have

$$\delta_{i+1,k} = \delta_{i+1,k}(M) = \delta_{i,k}$$

The following lemma shows that $\pi_{h,w}(\mathcal{P})$ correctly implements the extended transition function.

**Lemma F.7** The following results hold.

1. $\delta_{1,1} = \delta$.

2. For every pair of integers $1 \leq i \leq h+1$, $1 \leq k \leq w$, if $used(i, k) \in S$ then

   a) $p^k_i(S)$ is a conditional plan
b) furthermore, if \( \delta_{i,k} \) is consistent then \( \mathcal{G} \) is true in \( \mathcal{G}_{S}^{pc}(p_{1}^{k}(S), \delta_{i,k}) \).

Proof.

1. \( Z_{1} = A_{1,1}^{\{\text{holds}\}} \) is a splitting set for \( \pi_{1} \). The bottom part, \( b_{Z_{1}}(\pi_{1}) \), consists of the following rules:

\[
\text{holds}(l, 1, 1) \leftarrow \\
\{ l \in \delta \}
\]
\[
\text{holds}(l, 1, 1) \leftarrow \text{holds} (\psi, 1, 1) \\
\{ [ l \text{ if } \psi ] \in D \}
\]

By Lemma F.2, the only answer set for the above program is

\[
M = \{ \text{holds}(l, 1, 1) \mid l \in \text{Cl}_{D}(\delta) \} = \{ \text{holds}(l, 1, 1) \mid l \in \delta \}
\]

Thus,

\[
\delta_{1,1} = \delta_{1,1}(M) = \delta
\]

2. We now prove Item 2 by induction on parameter \( i \).

   a. Base case: \( i = h+1 \). Let \( k \) be an arbitrary integer between 1 and \( w \) such that \( used(i, k) \in S \). Clearly \( p_{1}^{k}(S) = [\] \) is a conditional plan.

   Now suppose that \( \delta_{i,k} \) is consistent. According to the definition of the extended transition function, we have

\[
\mathcal{G}_{S}^{pc}(o_{1}^{k}(S), \delta_{i,k}) = \mathcal{G}_{S}^{pc}(\langle \rangle, \delta_{i,k}) = \{ \delta_{i,k} \}
\]
On the other hand, by Lemma F.3, we have that $G$ is true in $\delta_{i,k}$. Thus, Item 2 is true for $i = h+1$.

b. **Inductive step:** Assume that Item 2 is true for all $h + 1 \geq i > t$. We will show that it is true for $i = t$. Let $k$ be an integer between 1 and $w$ such that $used(t, k) \in S$. Consider three possibilities (recall that $a_{t,k}$ is defined by (256)):

i. $a_{t,k}$ is a non-sensing action.

By the definition of $\alpha^k_t(S)$, we have $\alpha^k_t(S) = \langle a_{t,k}, \alpha^k_{t+1}(S) \rangle$. In addition, by rule (223) we have $used(t+1, k) \in S$. Thus, according to the inductive hypothesis, $\alpha^k_{t+1}(S)$ is a conditional plan. Accordingly, $\alpha^k_t(S)$ is also a conditional plan.

Now suppose that $\delta_{t,k}$ is consistent. Consider two cases

* $\delta_{t+1,k}$ is consistent. We have

$$\hat{\Phi}^pe_S(\alpha^k_t(S), \delta_{t,k}) = \hat{\Phi}^pe_S(\langle a_{t,k}, \alpha^k_{t+1}(S) \rangle, \delta_{t,k}) = \hat{\Phi}^pe_S(\alpha^k_{t+1}(S), \delta_{t+1,k})$$

(by Lemma F.6 and by the definition of the extended transition function).

On the other hand, according to the inductive hypothesis, $G$ is true in $\hat{\Phi}^pe_S(\alpha^k_{t+1}(S), \delta_{t+1,k})$. Hence, the inductive step is proved.

* $\delta_{t+1,k}$ is inconsistent. By Lemma F.6, we have $\hat{\Phi}^pe_S(\alpha^k_t(S), \delta_{t,k}) = \emptyset$. Thus, the inductive step is proved.
ii. \( a_{t,k} \) is a sensing action with the following knowledge-producing law

\[
a_{t,k} \text{ determines } g_1, \ldots, g_n
\]

By Lemma F.5 there exist exactly \( n \) integers \( k_1, \ldots, k_n \) greater than \( k \) such that \( br(g_j, t, k, k_j) \in S \) for \( 1 \leq j \leq n \). This implies that \( used(t+1, k_j) \in S \) (see rules (218) and (223)). Thus, by the definition of \( \alpha^k_t(S) \), we have \( \alpha^k_t(S) = \langle a_{t,k}, \text{cases}(\{g_j \rightarrow p_{t+1}^{k_j}(S)\}_{j=1}^n) \rangle \). On the other hand, we know by the inductive hypothesis that \( p_{t+1}^{k_j}(S) \) is a conditional plan for \( 1 \leq j \leq n \). As a result, \( \alpha^k_t(S) \) is also a conditional plan.

Suppose \( \delta_{t,k} \) is consistent. Let \( J = \{ j \mid \delta_{t+1,k_j} \text{ is consistent} \} \). By Lemma F.6, we have

\[
\Phi_{pc}^S(a_{t,k}, \delta_{t,k}) = \{ \delta_{t+1,k_j} \mid j \in J \}
\]

and \( g_j \) holds in \( \delta_{t+1,k_j} \) for every \( 1 \leq j \leq n \). Hence, by the definition of \( \hat{\Phi}_{pc}^S \), we have

\[
\hat{\Phi}_{pc}^S(\alpha^k_t(S), \delta_{t,k}) = \bigcup_{j \in J} \hat{\Phi}_{pc}^S(\alpha^k_{t+1}(S), \delta_{t+1,k_j})
\]

According to the inductive hypothesis, \( G \) is true in \( \hat{\Phi}_{pc}^S(\alpha^k_{t+1}(S), \delta_{t+1,k_j}) \),

where \( j \in J \). This implies that \( G \) is also true in \( \hat{\Phi}_{pc}^S(\alpha^k_t(S), \delta_{t,k}) \).

iii. \( a_{t,k} = \emptyset \).
According to the definition of $p^k_t(S)$, $p^k_t(S) = \langle \rangle$, which by definition, is also a conditional plan.

It is easy to see that $goal(t, k) \in S$, which means that either $\delta_{t,k}$ is inconsistent or $\mathcal{G}$ is true in $\delta_{t,k}$ (see rules (214), (215), and (221)). Now suppose that $\delta_{t,k}$ is consistent. This implies that $\mathcal{G}$ is true in $\delta_{t,k}$. We have

$$\Phi_{S}^{pc}(\alpha^k_t(S), \delta_{t,k}) = \Phi_{S}^{pc}(\langle \rangle, \delta_{t,k}) = \{\delta_{t,k}\}$$

Thus, the inductive step is proved.

\[\square\]

Theorem 5.2 immediately follows from Lemma F.7.

F.2.2 Proof of Proposition 5.4

First, we prove the following lemma.

**Lemma F.8** Let $\mathcal{P} = \langle \mathcal{D}, \delta, \mathcal{G} \rangle$ be a planning instance, $\delta$ be a partial state and $\alpha$ be a plan. If $\mathcal{G}$ is true in $\Phi_{S}^{pc}(\alpha, \delta)$ then $\mathcal{G}$ is also true in $\Phi_{S}^{pc}(\text{reduct}_\delta(\alpha), \delta)$.

**Proof.** Let us prove the lemma by structural induction on $p$.

1. $\alpha = \langle \rangle$.

   The proof is trivial since $\text{reduct}_\delta(\alpha) = \alpha = \langle \rangle$.

2. Assume that $\alpha = \langle a, \beta \rangle$, where $\beta$ is a conditional plan and $a$ is a non-sensing action and the lemma is true for $\beta$. 

338
Suppose $G$ is true in $\hat{\Phi}_{S}^{pc}(\alpha, \delta)$. We need to show that it is also true in $\hat{\Phi}_{S}^{pc}(\text{reduct}_{\delta}(\alpha), \delta)$.

If $G$ is true in $\delta$ then it is easy to see that it is also true in

$$\hat{\Phi}_{S}^{pc}(\text{reduct}_{\delta}(\alpha), \delta) = \hat{\Phi}_{S}^{pc}(()), \delta) = \{\delta\}$$

Now consider the case that $G$ is not true in $\delta$ (i.e., either it is false or unknown in $\delta$).

Since $\hat{\Phi}_{S}^{pc}(a, \delta) \neq \perp$, we have $\Phi_{S}^{pc}(a, \delta) \neq \perp$. Therefore, $\Phi_{S}^{pc}(a, \delta) = \{\delta'\}$ for some $\delta'$. Hence, by the definition of reduct, we have

$$\text{reduct}_{\delta}(\alpha) = (a, \text{reduct}_{\delta'}(\beta))$$

Thus,

$$\hat{\Phi}_{S}^{pc}(\text{reduct}_{\delta}(\alpha), \delta) = \hat{\Phi}_{S}^{pc}(\text{reduct}_{\delta'}(\beta), \delta')$$

On the other hand, we have

$$\hat{\Phi}_{S}^{pc}(\alpha, \delta) = \hat{\Phi}_{S}^{pc}(\beta, \delta')$$

Because $G$ is true in $\hat{\Phi}_{S}^{pc}(\alpha, \delta)$, $G$ is also true in $\hat{\Phi}_{S}^{pc}(\beta, \delta')$. By inductive hypothesis, we have $G$ is true in $\hat{\Phi}_{S}^{pc}(\text{reduct}_{\delta'}(\beta), \delta')$. Hence, $G$ is true in

$$\hat{\Phi}_{S}^{pc}(\text{reduct}_{\delta}(\alpha), \delta)$$

3. Assume that $\alpha = (a, \text{cases}(\{g_{j} \rightarrow \beta_{j}\}_{j=1}^{n}))$, where $a$ is a sensing action that senses $g_{1}, \ldots, g_{n}$, and the lemma holds for $\beta_{j}$'s.
Suppose \( G \) is true in \( \hat{\Phi}^p_c(\alpha, \delta) \). We need to show that it is also true in \( \hat{\Phi}^p_c(\text{reduct}, \delta, \alpha) \).

If \( G \) is true in \( \delta \) then it is also true in

\[
\hat{\Phi}^p_c(\text{reduct}, \delta, \alpha) = \hat{\Phi}^p_c(\delta, \delta) = \{\delta\}
\]

Now consider the case that \( G \) is not true in \( \delta \). There are two possibilities.

a) There exists \( g_k \) such that \( g_k \in \delta \).

By the definition of \( \text{reduct} \), we have

\[
\text{reduct}_\delta(\alpha) = \text{reduct}_\delta(\beta_k)
\]

By the definition of the \( \hat{\Phi}^A \) function, it is easy to see that

\[
\hat{\Phi}^p_c(\alpha, \delta) = \hat{\Phi}^p_c(\beta_k, \delta)
\]

Since \( G \) is true in \( \hat{\Phi}^p_c(\alpha, \delta) \), it is true in \( \hat{\Phi}^p_c(\beta_k, \delta) \). By the inductive hypothesis, it follows that \( G \) is true in \( \hat{\Phi}^p_c(\text{reduct}_\delta(\beta_k), \delta) \). Hence, \( G \) is true in

\[
\hat{\Phi}^p_c(\text{reduct}_\delta(\alpha), \delta)
\]

a) For every \( 1 \leq j \leq n \), \( g_j \not\in \delta \).

For every \( 1 \leq j \leq n \), let \( \delta_j = Cl_D(\delta \cup \{g_j\}) \). Let \( J = \{j \mid \delta_j \text{ is consistent}\} \).

It is easy to see that

\[
\hat{\Phi}^p_c(\alpha, \delta) = \bigcup_{j \in J} \hat{\Phi}^p_c(\beta_j, \delta_j)
\]
Hence, it follows that $G$ is also true in $\Phi_{pc}^S(\beta_j, \delta_j)$ for every $j \in J$.

On the other hand, by the definition of reduct, we have

$$\text{reduct}_\delta(\alpha) = \langle a, \text{cases}(\{g_j \mapsto \beta_j'\}_{j=1}^n) \rangle$$

where

$$\beta_j' = \begin{cases} \langle \rangle & \text{if } j \not\in J \\ \text{reduct}_{\delta_j}(\beta_j) & \text{otherwise} \end{cases}$$

Thus,

$$\Phi_{pc}^S(\text{reduct}_\delta(\alpha), \delta) = \bigcup_{j \in J} \Phi_{pc}^S(\beta_j', \delta_j) = \bigcup_{j \in J} \Phi_{pc}^S(\text{reduct}_{\delta_j}(\beta_j), \delta_j)$$

By the inductive hypothesis, for every $j \in J$, as $G$ is true in $\Phi_{pc}^S(\beta_j, \delta_j)$, we have that $G$ is also true in $\Phi_{pc}^S(\text{reduct}_{\delta_j}(\beta_j), \delta_j)$. As a result, $G$ is true in $\Phi_{pc}^S(\text{reduct}_\delta(\alpha), \delta)$.

We now prove Proposition 5.4. Let $\alpha$ be a solution of $\mathcal{P}$ with respect to $T_{pc}^{PC}(D)$. From the construction of reduct, it is easy to see that $\text{reduct}_\delta(\alpha)$ is unique. By Lemma F.8, we have that $G$ is true in $\Phi_{pc}^S(\text{reduct}_\delta(\alpha), \delta)$. Thus, $\text{reduct}_\delta(\alpha)$ is also a solution of $\mathcal{P}$ with respect to $T_{pc}^{PC}(D)$.

So, we can conclude the proposition.

**F.2.3 Proof of Theorem 5.3**

The idea of the proof is as follows. Let $\beta$ be $\text{reduct}_\delta(\alpha)$, where $\delta$ is the initial partial state of $\mathcal{P}$, and let $T_\beta$ be the labeled tree for $\beta$ numbered according to the
principles described in Section 5.4. Let $h$ and $w$ denote the height and width of $T_\beta$ respectively. For $1 \leq i \leq h + 1$, $1 \leq k \leq w$, we define $\delta_{i,k}$ to be the partial state at node $(i, k)^1$ of $T_\beta$ if such a node exists and $\bot$ otherwise. Based on $T_\beta$ and $\delta_{i,k}$, we construct the set $Y_{i,k}$ of atoms that hold at node $(i, k)$. Then we prove that the union of these sets, denoted by $S'_0$, is an answer set for $\pi_0$ (rules (207)-(223)) by showing that each $Y_{i,k}$ is an answer set for a part of $\pi_0$, denoted by $\pi'^{i,k}$. Furthermore, a set $S'$ can be constructed from $S'_0$ in such a way that it is an answer set for $\pi_{h,w}^c (P)$ and does not violate any constraints in $\pi_{h,w}^c$ (rules (225)-(233)). As such, it is an answer set for $\pi_{h,w} (P)$ (more accurately it is an answer set for the program $\pi_{h,w} (P)$ with choice rules are converted into normal logic program rules as described in Section F.2.1). Moreover, $\beta = \alpha_1^1 (S')$. 

Given the numbered tree $T_\beta$, by $\langle a, i, k \rangle$ we mean the node labeled with $a$ and numbered with $(i, k)$ in $T_\beta$; by $\langle g, i, k, k' \rangle \in T_\beta$ we mean the link, whose label is $g$, between the nodes $(i, k)$ and $(i + 1, k')$ in $T_\beta$.

For $1 \leq i \leq h + 1$, $1 \leq k \leq w$, we define the partial state $\delta_{i,k}$ as follows.

i. if $i = 1$

$$\delta_{i,k} = \begin{cases} \delta & \text{if } k = 1 \\ \bot & \text{if } k > 1 \end{cases}$$

(262)

ii. if $i > 1$

$$\delta_{i,k} = \begin{cases} \text{Cl}_D(\text{E}(a, \delta_{i-1,k}) \cup (\delta_{i-1,k} \setminus \text{pc}(a, \delta_{i-1,k}))) & \text{if } \langle a, i-1, k \rangle \in T_\beta \text{ for a non-sensing action } a \\ \text{Cl}_D(\delta_{i-1,k'} \cup \{g\}) & \text{if } \langle g, i-1, k', k \rangle \in T_\beta \\ \delta_{i-1,k} & \text{otherwise} \end{cases}$$

(263)

\(^1\text{That is, the node numbered with } (i, k) \text{ in } T_\beta \)
Note that given \((i, k)\), there exists at most one action \(a\) such that \(\langle a, i-1, k \rangle \in T_\beta\), and furthermore, at most one pair \(\langle g, k' \rangle\) such that \(\langle g, i-1, k', k \rangle \in T_\beta\). In addition, the conditions in Equation (263) do not overlap each other. Thus, \(\delta_{i,k}\) is uniquely defined for \(1 \leq i \leq h + 1\) and \(1 \leq k \leq w\). In what follows, the undefined situation \(\perp\) can sometimes be thought of as \(\emptyset\), depending the context in which it is used.

Let us construct the set \(Y_{i,k}\) of atoms based on \(\delta_{i,k}\) as follows.

1. \(used(1, 1) \in Y_{1,1}\)

2. If \(l \in \delta_{i,k}\) then \(holds(l, i, k) \in Y_{i,k}\).

3. If \(\langle g, i, k, k' \rangle \in T_\beta\) for some \(g, k'\) then \(br(g, i, k, k') \in Y_{i,k}\).

4. If \(\langle a, i, k \rangle \in T_\beta\) and \(l \in E(a, \delta_{i,k})\) for some non-sensing action \(a\) then \(ef(l, i, k) \in Y_{i,k}\).

5. If \(\langle a, i, k \rangle \in T_\beta\) and \(l \in pc(a, \delta_{i,k})\) for some non-sensing action \(a\) then \(pc(l, i, k) \in Y_{i,k}\).

6. For \(i > 1\), if either
   
   (a) \(used(i-1, k) \in Y_{i-1,k}\); or
   
   (b) there exists \(\langle g, k' \rangle\) s.t. \(\langle g, i-1, k', k \rangle \in Y_{i-1,k'}\)

   then \(used(i, k) \in Y_{i,k}\).

7. If \(G\) is true in \(\delta_{i,k}\) or \(\delta_{i,k}\) is inconsistent then \(goal(i, k) \in Y_{i,k}\).
8. If there exists an action $a$ such that $\langle a, i, k \rangle \in T_\beta$ then

$$\text{occ}(e, i, k) \in Y_{i,k}$$

for all $e \in a$, and

$$\neg\text{occ}(e, i, k) \in Y_{i,k}$$

for all $e \notin a$.

9. If there exists no action $a$ such that $\langle a, i, k \rangle \in T_\beta$, $\text{used}(i, k) \in Y_{i,k}$, and $\text{goal}(i, k) \notin Y_{i,k}$ then

$$\neg\text{occ}(e, i, k) \in Y_{i,k}$$

for all elementary action $e \in A$.

10. Nothing else in $Y_{i,k}$.

Clearly, $Y_{i,k}$’s are uniquely defined. Furthermore, they are disjoint from each other. Let

$$Y_i = \bigcup_{k=1}^{w} Y_{i,k} \text{ and } S_0' = \bigcup_{i=1}^{h+1} Y_i$$

**Lemma E.9** For $1 \leq i \leq h$ and $1 \leq k \leq w$, let $M = Y_{i,k}^{\{\text{holds, goal, used, occ, } \neg\text{occ}\}}$ and let $\Pi$
be the following program:

\[
\begin{align*}
\text{ef}(l, i, k) & \leftarrow \\
& (\text{occ}(e, i, k) \in M, [e \text{ causes } l \text{ if } \psi] \in D, \text{holds}(\psi, i, k) \subseteq M) \\
\text{pc}(l, i, k) & \leftarrow \\
& (\text{occ}(e, i, k) \in M, [e \text{ causes } l \text{ if } \psi] \in D, \\
& \text{holds}(l, i, k) \not\in M, \text{holds}(\neg \psi, i, k) \cap M = \emptyset) \\
\text{br}(g, i, k, k) & \mid \ldots \\
\text{br}(g, i, k, w) & \leftarrow \\
& (\text{occ}(c, i, k) \in M, [c \text{ determines } \theta] \in D, g \in \theta) \\
\text{pc}(l, i, k) & \leftarrow \text{pc}(l', i, k), \text{not } ef(\neg \psi, i, k) \\
& ([l \text{ if } \psi] \in D, \text{holds}(l, i, k) \not\in M, l' \in \psi) \\
\text{ef}(l, i, k) & \leftarrow \text{ef}(\psi, i, k) \\
& ([l \text{ if } \psi] \in D)
\end{align*}
\]

Then, \(N = \text{A}^{e, pc, br}_{i,k} \) is an answer set for \(\Pi\).

**Proof.** Given \((i, k)\), there are three cases that may happen at node \((i, k)\).

- there exists a non-sensing action \(a\) such that \(\langle a, i, k \rangle \in T_\beta\);
- there exists a sensing action \(a\) such that \(\langle a, i, k \rangle \in T_\beta\);
- \(\langle a, i, k \rangle \not\in T_\beta\) for every action \(a\)
Let us consider each of those in turn.

1. $\langle a, i, k \rangle \in T_\beta$ for some non-sensing action $a$.

From the construction of $Y_{i,k}$, we know that $a = \{ e ~|~ \text{occ}(e, i, k) \in M \}$.

Furthermore, due to the fact that $N$ does not contain any atom of the form $\text{holds}(l, i, k)$, we have $\text{holds}(l, i, k) \in M$ iff $\text{holds}(l, i, k) \in Y_{i,k}$. That means $\text{holds}(l, i, k) \in M$ iff $l \in \delta_{i,k}$.

Hence, $\Pi$ can be rewritten to:

$$
\begin{align*}
\text{ef}(l, i, k) & \leftarrow \\
& (l \in E(a, \delta_{i,k})) \\
\text{pc}(l, i, k) & \leftarrow \\
& (l \in \text{pc}^0(a, \delta_{i,k})) \\
\text{pc}(l, i, k) & \leftarrow \text{pc}(l', i, k), \text{not} \text{ef}(\neg \psi, i, k) \\
& ([l \text{ if } \psi] \in D, l \notin \delta_{i,k}, l' \in \psi) \\
\text{ef}(l, i, k) & \leftarrow \text{ef}(\psi, i, k) \\
& ([l \text{ if } \psi] \in D)
\end{align*}
$$

As have been seen in the proof of Theorem 5.2 (see the proof of Lemma F.5, Item 1), the only answer set for this program is

$$
\{ \text{ef}(l, i, k) ~|~ l \in E(a, \delta_{i,k}) \} \cup \{ \text{pc}(l, i, k) ~|~ l \in \text{pc}(a, \delta_{i,k}) \} = N.
$$

2. $\langle c, i, k \rangle \in T_\beta$ for some sensing action $c$
Since \( \text{occ}(e, i, k) \not\in M \) for every elementary action \( e \neq c \), the program \( \Pi \) is the following set of rules.

\[
\begin{align*}
\text{br}(g, i, k, k) & \mid \ldots \\
\text{br}(g, i, k, w) & \leftarrow \\
& (\text{occ}(c, i, k) \in M, [c \text{ determines } \theta] \in D, g \in \theta)
\end{align*}
\]

\[
\begin{align*}
\text{pc}(l, i, k) & \leftarrow \text{pc}(l', i, k), \text{not } \text{ef}(\neg \psi, i, k) \\
& ([l \text{ if } \psi] \in D, \text{holds}(l, i, k) \not\in M, l' \in \psi)
\end{align*}
\]

\[
\begin{align*}
\text{ef}(l, i, k) & \leftarrow \text{ef}(\psi, i, k) \\
& ([l \text{ if } \psi] \in D)
\end{align*}
\]

It is easy that an answer set for \( \Pi \) is also an answer set for

\[
\begin{align*}
\text{br}(g, i, k, k) & \mid \ldots \\
\text{br}(g, i, k, w) & \leftarrow \\
& (\text{occ}(a, i, k) \in M, [c \text{ determines } \theta] \in D, g \in \theta)
\end{align*}
\]

and vice versa. On the other hand,

\[
N = Y_{i,k}^{\{\text{ef}, \text{pc}, \text{br}\}} = \{\text{br}(g, i, k, k') \mid \langle g, i, k, k' \rangle \in T_\beta\}
\]

is an answer set for the latter program. As a result, \( N \) is also an answer set for \( \Pi \).

3. \( \langle a, i, k \rangle \not\in T_\beta \) for every action \( a \).
In this case, the first three rules of $\Pi$ do not exist because $occ(e, i, k) \notin M$ for every elementary action $e$. Thus, $\Pi$ consists of the last two rules only. It is easy to see that the program has the empty set as its only answer set. On the other hand, from the construction of $Y_{i, k}$, we have $Y_{i, k}^{\{ef, pc, br\}} = \emptyset$. Accordingly, $Y_{i, k}^{\{ef, pc, br\}}$ is an answer set for $\Pi$.

The proof is done. $\square$

**Lemma F.10** For $1 \leq i \leq h + 1$, $1 \leq k \leq w$, $Y_{i, k}$ is an answer set for $\pi_{i}^{k}$, where $\pi_{i}^{k}$ is defined in the same way as $\pi_{i}^{k}$ except that we replace every occurrence of $X$ in Equation (255) by $Y$.

**Proof.** Let us consider in turn two cases $i = 1$ and $i > 1$.

1. $i = 1$. It is easy to see that the only answer set for $\pi_{i}^{k}$, where $k > 1$, is

$$Y_{1, k} = \emptyset$$

by using the splitting set $A_{1, k}^{\{\text{holds, occ, } \neg \text{occ, br, } \text{used, ef, pc}\}}$ (see (204) for the definition of $A_{i, k}$) and observe that the bottom part has the empty set as its only answer set and the evaluation of the top part has $\emptyset$ as its only answer set.

We now prove that $Y_{1, 1}$ is an answer set for $\pi_{1}^{1}$ which consists of the rules of the forms (238)-(246), (252)-(253) where $t = 1$ and $p = 1$. If we use the set $Z_{1} = A_{1, 1}^{\{\text{holds, occ, } \neg \text{occ, goal, used}\}}$ to split $\pi_{1}^{1}$ then $b_{Z_{1}}(\pi_{1}^{1})$ is

$$\{(238), (244) - (246), (252), (253) \mid t = 1, p = 1\}$$
From the definition of $Y_{1,1}$, we can easily show that $M = Y_{1,1}^{\{\text{holds, occ, } \neg \text{ occ}, \text{goal, used}\}}$ is an answer set for $b_{Z_1}(\pi^1_1)$. Furthermore, we have

$$\delta_{1,1}(M) = \delta_{1,1}(Y_{1,1}) = \delta_{1,1}$$

The evaluation of the top part, $\Pi_1 = e_{Z_1}(\pi^1_1 \setminus b_{Z_1}(\pi^1_1), M)$, is the following set of rules

$$ef(l, 1, 1) \iff (\text{occ}(e, 1, 1) \in M, [e \text{ causes } l \text{ if } \psi] \in D,$$

$$\text{holds}(\psi, 1, 1) \subseteq M)$$

$$pc(l, 1, 1) \iff (\text{occ}(e, 1, 1) \in M, [e \text{ causes } l \text{ if } \psi] \in D,$$

$$\text{holds}(l, 1, 1) \not\in M, \text{holds}(\neg \psi, 1, 1) \cap M = \emptyset)$$

$$br(g, 1, 1, k) \mid \ldots$$

$$br(g, 1, 1, w) \iff ([e \text{ determines } \theta] \in D, g \in \theta, \text{occ}(c, 1, 1) \in M)$$

$$pc(l, 1, 1) \iff pc(l', 1, 1), \not ef(\neg \psi, 1, 1)$$

$$([l \text{ if } \psi] \in D, l' \in \psi, \text{holds}(l, 1, 1) \not\in M)$$

$$ef(l, 1, 1) \iff ef(\psi, 1, 1)$$

$$([l \text{ if } \psi] \in D)$$
By Lemma F.9, \( N = Y_{1,1}^{\{ef,pc,br\}} \) is an answer set for \( \Pi_1 \). As a result, \( Y_{1,1} = M \cup N \) is an answer set for \( \pi'_{1} \).

2. \( 1 < i \leq h + 1 \).

Using the splitting set \( Z_2 = A_{i,k}^{\{holds,occ,goal,used\}} \) to split \( \pi'^k \), we have that the bottom part \( \Pi_2 = b_{Z_2}(\pi'^k_i) \) consists of rules of the forms

\[
\begin{align*}
\bullet & \text{ (244)-(252), and (254) if } i \leq h \\
\bullet & \text{ (244)-(251), and (254) if } i = h + 1
\end{align*}
\]

We now prove that \( M = Y_{i,k}^{\{holds,occ,goal,used\}} \) is an answer set for \( \Pi_2 \). Let us further split \( \Pi_2 \) by the set \( Z_3 = A_{i,k}^{\{holds\}} \). Then, the bottom part \( b_{Z_3}(\Pi_2) \) consists of rules of the forms (244), (247)-(248), (250)-(251) only.

Consider three cases

a. \textit{there exists a elementary non-sensing action }e\textit{ such that }occ(e, i−1, k) \in Y_i−1.\]

Let

\[
a = \{ e \mid occ(e, i−1, k) \in Y_i−1 \}
\]

From the construction of \( Y_{i,k} \)'s, it is easy to see that there exists no \( \langle g, k' \rangle \) such that \( br(g, i−1, k', k) \in Y_i−1 \). Thus, \( b_{Z_3}(\Pi_2) \) contains rules of the forms (244), (247)-(248) only. On the other hand, we have

\[
e f(l, i−1, k) \in Y_i−1 \text{ iff } l \in E(a, \delta_{i−1,k})
\]
pc(−l, i−1, k) ⊈ Y_{i−1} iff −l ⊈ pc(a, δ_{i−1,k})

Hence, $b_{Z_3} (\Pi_2)$ is the following collection of rules:

\[
holds(l, i, k) \leftarrow holds(\psi, i, k)
\]

\[
holds(l, i, k) \leftarrow ([ l \text{ if } \psi ] \in \mathcal{D})
\]

\[
holds(l, i, k) \leftarrow (l \in E(a, \delta_{i−1,k}))
\]

\[
holds(l, i, k) \leftarrow (l \in \delta_{i−1,k}, \neg l \notin \delta_{i−1,k})
\]

By Lemma F.2, it has the only answer set

\[
\{ holds(l, i, k) \mid l \in \text{Cl}_{\mathcal{D}}(E(a, \delta_{i−1,k}) \cup (\delta_{i−1,k} \setminus pc(a, \delta_{i−1,k}))) \} = Y'_{i,k}^{\text{holds}}
\]

b. $\exists (g, k'). \text{br}(g, i−1, k', k) \in Y_{i−1}$.

From the construction of $Y'_{i,k}$'s, such $(g, k')$ is unique and in addition $k' \leq k$.

Thus, $b_{Z_3} (\Pi_2)$ is

\[
holds(l, i, k) \leftarrow holds(\psi, i, k)
\]

\[
holds(l, i, k) \leftarrow ([ l \text{ if } \psi ] \in \mathcal{D})
\]

\[
holds(l, i, k) \leftarrow ((l \in \delta_{i−1,k}) \lor (k' < k \land l \in \delta_{i−1,k'}))
\]

\[
holds(g, i, k) \leftarrow
\]
or equivalently,

\[ \text{holds}(l, i, k) \leftarrow \text{holds}(\psi, i, k) \]

\[ ([ l \text{ if } \psi ] \in \mathcal{D}) \]

\[ \text{holds}(l, i, k) \leftarrow \]

\[ (l \in \delta_{i-1,k'} \cup \{g\}) \]

since if \( k' < k \) then \( \delta_{i-1,k} = \emptyset \). By Lemma F.2, this program has the only answer set

\[ \{\text{holds}(l, i, k) \mid l \in Cl_\mathcal{D}(\delta_{i-1,k'} \cup \{g\})\} = \{\text{holds}(l, i, k) \mid l \in \delta_{i,k}\} \]

Hence, \( Y_{i,k}^{\{\text{holds}\}} \) is the only answer set for \( b_{Z_3}(\Pi_2) \).

c. \( \text{occ}(e, i-1, k) \not\in Y_{i-1} \) for every elementary, non-sensing action \( e \) and,

\( \text{br}(g, i-1, k', k) \not\in Y_{i-1} \) for all pair \( (g, k') \).

From the construction of \( Y_{i,k} \)'s, it follows that \( ef(l, i-1, k) \not\in Y_{i-1} \) and

\( pc(l, i-1, k) \not\in Y_{i-1} \) for every \( l \). Hence, \( b_{Z_3}(\Pi_2) \) is the following set of rules

\[ \text{holds}(l, i, k) \leftarrow \text{holds}(\psi, i, k) \]

\[ ([ l \text{ if } \psi ] \in \mathcal{D}) \]

\[ \text{holds}(l, i, k) \leftarrow \]

\[ (l \in \delta_{i-1,k}) \]

whose only answer set is

\[ \{\text{holds}(l, i, k) \mid l \in \delta_{i-1,k}\} = \{\text{holds}(l, i, k) \mid l \in \delta_{i,k}\} = Y_{i,k}^{\{\text{holds}\}} \]
So, in all three cases, we have $Y_{i,k}^{\text{holds}}$ is an answer set for $b_{Z_3}(\Pi_2)$.

Hence, $\Pi_3 = e_{Z_3}(\Pi_2 \setminus b_{Z_3}(\Pi_2), Y_{i,k}^{\text{holds}})$ is the following set of rules:

\[
\begin{align*}
\text{used}(i, k) & \leftarrow \\
& \quad (\exists (g, k'). k' < k, br(g, i-1, k', k) \in Y_{i-1}) \\
\text{goal}(i, k) & \leftarrow \\
& \quad (\mathcal{G} \subseteq \delta_{i,k}) \\
\text{goal}(i, k) & \leftarrow \\
& \quad (\delta_{i,k} \text{ is inconsistent}) \\
\text{occ}(e, i, k) \mid \neg \text{occ}(e, i, k) & \leftarrow \text{used}(i, k), \neg \text{goal}(i, k) \\
& \quad (e \in A) \\
\text{used}(i, k) & \leftarrow \\
& \quad (\text{used}(i-1, k) \in Y_{i-1})
\end{align*}
\]

It is easy to see that $Y_{i,k}^{\text{used,goal,occ,\neg occ}}$ is an answer set for $\Pi_3$. Accordingly, we have $M = Y_{i,k}^{\text{holds,used,goal,occ,\neg occ}}$ is an answer set for $\Pi_2$. 

353
\[ \Pi_4 = e_{Z_2}(\pi'^k_i \setminus \Pi_2, M) \] is thus the following set of rules:

\[
ef(l, i, k) \leftarrow (\text{occ}(e, i, k) \in M, [e \text{ causes } l \text{ if } \psi] \in \mathcal{D},
\]
\[
\text{holds}(\psi, i, k) \subseteq M)
\]

\[
\text{pc}(l, i, k) \leftarrow (\text{occ}(e, i, k) \in M, [e \text{ causes } l \text{ if } \psi] \in \mathcal{D},
\]
\[
\text{holds}(l, i, k) \notin M, \text{holds}(\neg\psi, i, k) \cap M = \emptyset)
\]

\[
br(g, i, k, k) | \ldots
\]

\[
br(g, i, k, w) \leftarrow (\text{occ}(c, i, k) \in M, [c \text{ determines } \theta] \in \mathcal{D}, g \in \theta)
\]

\[
\text{pc}(l, i, k) \leftarrow \text{pc}(l', i, k), \neg \text{ef}(\neg\psi, i, k)
\]
\[
([l \text{ if } \psi] \in \mathcal{D}, \text{holds}(l, i, k) \notin M, l' \in \psi)
\]

\[
ef(l, i, k) \leftarrow \text{ef}(\psi, i, k)
\]
\[
([l \text{ if } \psi] \in \mathcal{D})
\]

By Lemma F.9, \(N = Y^{\{\text{ef,pc,br}\}}_{i,k}\) is an answer set for \(\Pi_4\).

As a result, \(Y_{i,k} = M \cup N\) is an answer set for \(\pi'^k_i\).

Lemma F.11 We have
1. \( S' = \bigcup_{i=1}^{h+1} Y_i \cup X_0 \) is an answer set for \( \pi_{h,w}(D) \), where \( X_0 = V \) is defined in (206).

2. \( \alpha_1^1(S') = \beta \) where \( \beta = \text{reduct}_\delta(\alpha) \).

Proof.

1. Since \( Y_{i,k} \) is an answer set for \( \pi^k_i \) and \( \pi^r_i \)'s are disjoint from each other, we have \( Y_i \) is an answer set for \( \pi'_i \), where \( \pi'_i \) is defined in the same way as \( \pi_i \) except that every occurrence of \( X \) in Equations (236) and (237) is replaced with \( Y \). From the splitting sequence theorem, it follows that \( S'_0 = \bigcup_{i=1}^{h+1} Y_i \) is an answer set for \( \pi_0 \). Thus, \( S' \) is an answer set for \( \pi_{h,w}(P) \).

On the other hand, it is not difficult to show that \( S' \) satisfies all constraints in \( \pi_{h,w}^c(P) \) based on the following observations.

- If \( \text{occ}(c, i, k) \in Y_{i,k} \) for some sensing action \( c \) with the knowledge-producing law

\[
  c \text{ determines } \theta
\]

then there exists \( g \) in \( \theta \) such that \( \text{br}(g, i, k, k) \in Y_{i,k} \). Furthermore, for every \( g' \in \theta, g' \) does not in \( \delta_{i,k} \). The latter property holds because that \( \beta \) does not contain an action that senses an already known-to-be-true fluent literal.

In addition, there is no other elementary action \( e \) such that \( \text{occ}(e, i, k) \in Y_{i,k} \).
• If $used(h + 1, k) \in Y_{h+1,k}$ then $G$ is true in $\delta_{h+1,k}$.

• $\delta_{i,k}$ is either $\bot$ or a partial state. This means that $Y_{i,k}^{\{\text{holds}\}}$ does not contain two atoms of the forms $\text{holds}(l, i, k)$ and $\text{holds}(l', i, k)$, where $l$ and $l'$ are contrary fluent literals.

• No two branches come to the same node $(i, k)$.

• If $used(i, k) \in Y_i$ then $br(g, i, k', k) \not\in Y_i$ for any pair $(g, k')$, $k' \neq k$.

• If $(a, i, k) \in T_\beta$ then $a$ must be executable in $\delta_{i,k}$.

Accordingly, we have $S$ is an answer set for $\pi_{h,w}(P)$.

2. Immediate from the construction of $Y_{i,k}$.

\[ \square \]

Theorem 5.3 follows directly from this lemma.
APPENDIX G

A SAMPLE ENCODING OF ASCP

This appendix contains the encoding of a planning instance \( \mathcal{P} \) for the domain in Example 5.2. The first subsection describes the input planning problem. The next subsection presents the corresponding logic program \( \pi_{h,w}(\mathcal{P}) \). The last two subsections list the outputs of \textit{smodels} and \textit{cmodels} for \( \pi_{h,w}(\mathcal{P}) \) when run with parameters \( h = 2 \) and \( w = 3 \).

G.1 Input Domain

```
% Usage:
% lparse -c h=<height> -c w=<width> | smodels

% Input parameters
% time(1..h).
% time1(1..h+1).
% path(1..w).
```

% Action declarations
```
nonsensing(push_up).
nonsensing(push_down).
```

357
nonsensing(flip_lock).
sensing(check).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fluent declarations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fluent(open).
fluent(closed).
fluent(locked).
sense(open).
sense(closed).
sense(locked).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DOMAIN INDEPENDENT RULES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Auxiliary Rules
literal(F).
literal(neg(F)).
contrary(F,neg(F)).
contrary(neg(F),F).

action(NE).
action(SE).

used(1,1).
used(T+1,P) :-
   used(T,P).

% Goal representation
:- used(h+1,P),
   not goal(h+1,P).

% Inertial rules for non-sensing actions
holds(L,T+1,P) :-
   ef(L,T+1,P).

holds(L,T+1,P) :-
   holds(L,T,P),
   contrary(L,L1),
   not pc(L1,T+1,P).

358
% Inertial rules for sensing actions
new_br(P,P1) :-
P <= P1.

% Cannot branch to the same path
:- P1 < P2,
P2 < P,
br(G1,T,P1,P),
br(G2,T,P2,P).

:- G1 != G2,
P <= P1,
br(G1,T,P,P1),
br(G2,T,P,P1).

:- P1 < P,
br(G,T,P1,P),
used(T,P).

used(T+1,P) :-
P1 < P,
br(G,T,P1,P).

holds(G,T+1,P) :-
P1 <= P,
br(G,T,P1,P).

holds(L,T+1,P) :-
P1 < P,
br(G,T,P1,P),
holds(L,T,P1).

% Rules for generating action occurrences
1{occ(X,T,P):action(X)}1 :-
    used(T,P),
    not goal(T,P).

% DOMAIN DEPENDENT RULES

% Impossibility conditions
:- occ(push_up,T,P),
    not holds(closed,T,P).
:- occ(push_down,T,P),
    not holds(open,T,P).
:- occ(flip_lock,T,P),
    not holds(neg(open),T,P).
:- occ(push_up,T,P),
    occ(push_down,T,P).
:- occ(push_up,T,P),
    occ(flip_lock,T,P).
:- occ(flip_lock,T,P),
    occ(push_down,T,P).

% Static laws
holds(neg(open),T1,P) :-
    holds(closed,T1,P).

ef(neg(open),T+1,P) :-
    ef(closed,T+1,P).

pc(neg(open),T+1,P) :-
    pc(closed,T+1,P),
    not holds(neg(open),T,P),
    not ef(neg(closed),T+1,P).

holds(neg(open),T1,P) :-
    holds(locked,T1,P).

ef(neg(open),T+1,P) :-
    ef(locked,T+1,P).

pc(neg(open),T+1,P) :-
    pc(locked,T+1,P),
    not holds(neg(open),T,P),
    not ef(neg(locked),T+1,P).

holds(open,T1,P) :-
    holds(neg(closed),T1,P),
    holds(neg(locked),T1,P).

ef(open,T+1,P) :-
    ef(neg(closed),T+1,P),
    ef(neg(locked),T+1,P).

pc(open,T+1,P) :-

360
pc(neg(closed), T+1, P),
not holds(open, T, P),
not ef(closed, T+1, P),
not ef(locked, T+1, P).

pc(open, T+1, P) :-
    pc(neg(locked), T+1, P),
    not holds(open, T, P),
    not ef(closed, T+1, P),
    not ef(locked, T+1, P).

holds(neg(closed), T1, P) :-
    holds(open, T1, P).

ef(neg(closed), T+1, P) :-
    ef(open, T+1, P).

pc(neg(closed), T+1, P) :-
    pc(open, T+1, P),
    not holds(neg(closed), T, P),
    not ef(neg(open), T+1, P).

holds(neg(closed), T1, P) :-
    holds(locked, T1, P).

ef(neg(closed), T+1, P) :-
    ef(locked, T+1, P).

pc(neg(closed), T+1, P) :-
    pc(locked, T+1, P),
    not holds(neg(closed), T, P),
    not ef(neg(locked), T+1, P).

holds(closed, T1, P) :-
    holds(neg(open), T1, P),
    holds(neg(locked), T1, P).

ef(closed, T+1, P) :-
    ef(neg(open), T+1, P),
    ef(neg(locked), T+1, P).

pc(closed, T+1, P) :-
    pc(neg(open), T+1, P),
    not holds(closed, T, P),
not ef(open,T+1,P),
not ef(locked,T+1,P).

pc(closed,T+1,P) :-
    pc(neg(locked),T+1,P),
    not holds(closed,T,P),
    not ef(open,T+1,P),
    not ef(locked,T+1,P).

holds(neg(locked),T1,P) :-
    holds(open,T1,P).

ef(neg(locked),T+1,P) :-
    ef(open,T+1,P).

pc(neg(locked),T+1,P) :-
    pc(open,T+1,P),
    not holds(neg(locked),T,P),
    not ef(neg(open),T+1,P).

holds(neg(locked),T1,P) :-
    holds(closed,T1,P).

ef(neg(locked),T+1,P) :-
    ef(closed,T+1,P).

pc(neg(locked),T+1,P) :-
    pc(closed,T+1,P),
    not holds(neg(locked),T,P),
    not ef(neg(closed),T+1,P).

holds(locked,T1,P) :-
    holds(neg(open),T1,P),
    holds(neg(closed),T1,P).

ef(locked,T+1,P) :-
    ef(neg(open),T+1,P),
    ef(neg(closed),T+1,P).

pc(locked,T+1,P) :-
    pc(neg(open),T+1,P),
    not holds(locked,T,P),
    not ef(open,T+1,P),
    not ef(closed,T+1,P).
pc(locked,T+1,P) :-
    pc(neg(closed),T+1,P),
    not holds(locked,T,P),
    not ef(open,T+1,P),
    not ef(closed,T+1,P).

% Effects of non-sensing actions
ef(closed,T+1,P) :-
    occ(push_down,T,P).
pc(closed,T+1,P) :-
    occ(push_down,T,P).
ef(open,T+1,P) :-
    occ(push_up,T,P).
pc(open,T+1,P) :-
    occ(push_up,T,P).
ef(locked,T+1,P) :-
    occ(flip_lock,T,P),
    holds(closed,T,P).
pc(locked,T+1,P) :-
    occ(flip_lock,T,P),
    not holds(neg(closed),T,P).
ef(closed,T+1,P) :-
    occ(flip_lock,T,P),
    holds(locked,T,P).
pc(closed,T+1,P) :-
    occ(flip_lock,T,P),
    not holds(neg(locked),T,P).

% Effects of sensing actions
:- occ(check,T,P),
    not br(open,T,P,P),
    not br(closed,T,P,P),
    not br(locked,T,P,P).
1{br(open,T,P,X):new_br(P,X)}1 :-
    occ(check,T,P).
1{br(closed,T,P,X):new_br(P,X)}1 :-
    occ(check,T,P).
1{br(locked,T,P,X):new_br(P,X)}1 :-
    occ(check,T,P).
:- occ(check,T,P),
    holds(open,T,P).
:- occ(check,T,P),
holds(closed,T,P).
:- occ(check,T,P),
    holds(locked,T,P).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INITIAL STATE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
holds(neg(open),1,1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GOAL SPECIFICATION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
goal(T1,P) :-
    holds(locked,T1,P).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% HIDE THE FOLLOWING ATOMS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
hide.
show occ(A,T,P).
show br(G,T,P,P1).

G.2 The Output of smodels

$ lparse -c h=2 -c w=3 security_robot.smo | smodels
smodels version 2.28. Reading...done
Answer: 1
Stable Model: br(open,1,1,2) occ(check,1,1) br(closed,1,1,1)
    br(locked,1,1,3) occ(flip_lock,2,1)

True
Duration: 0.032
Number of choice points: 2
Number of wrong choices: 0
Number of atoms: 293
Number of rules: 813
Number of picked atoms: 209
Number of forced atoms: 30
Number of truth assignments: 3260
Size of searchspace (removed): 12 (57)
G.3 The Output of \texttt{cmodels}

\$ lparse -c h=2 -c w=3 \texttt{security\_robot.smo} | \texttt{cmodels}

cmodels

cmodels version 3.01 Reading...done
Program is not tight.
Calling SAT solver mChaff...
Answer: 1
Answer set: \texttt{br(open,1,1,3) occ(check,1,1) br(closed,1,1,1)}
\texttt{br(locked,1,1,2) occ(flip\_lock,2,1)}
Number of Loop Formulas 6
REFERENCES


369


