IMPROVING EFFICIENCY OF SOLVING COMPUTATIONAL PROBLEMS WITH ASP

by

A. RICARDO MORALES, B.S.

A Ph.D. DISSERTATION

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty of Texas Tech University in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Approved

Dr. Michael Gelfond
Committee Chairman

Dr. Richard Watson

Dr. Yuan-Lin Zhang

Fred Hartmeister
Dean of the Graduate School

December, 2010
A mi madre,
que su tarea sea cumplida.

In memory of Dr. Daniel Cooke,
a great inspiration.
ACKNOWLEDGEMENTS

During the many years that took this dissertation to reach its conclusion, many people have contributed directly or indirectly to my academic, research and personal life. Before I start mentioning them I apologize beforehand if I miss somebody.

I have had the great privilege and fortune to work under the guidance of Dr. Michael Gelfond. The experience of working with him has forever changed my life and made me the person I am now. His teachings have been alongside me since my college freshman years. I never imagined then that I would learn so much from him, and now I know that I will never stop learning from him. I greatly thank his help, guidance, support, and specially his patience to deal with me every time I tripped during this long journey. He gave me the confidence to continue every time I thought of giving up.

A special thanks to my committee members Dr. Richard Watson and Dr. Yuanlin Zhang. Countless hours were spend discussing the work in this dissertation, and their help was invaluable for slowly improving it into this finished work.

I thank the Computer Science department at Texas Tech university for the education I received here, for the years of financial support and for giving me the chance of being a lecturer. I really enjoyed the teaching experience.

During my stay at Texas Tech, many professors have been instrumental in shaping my education. I would like to specially thank Dr. Hector Hernandez, Dr. Eric Sinzinger, Dr. Larry Pyeatt, Dr. Gopal Lakhani and Dr. Nelson Rushton. I will also like thank Dr. Daniel Cooke. I will miss his laugh. In his memory I find great inspiration. I thank Misty Digby for always helping me navigate the departmental paperwork, and Derek Birkenfeld and David Hensley for their invaluable IT support. Other important professors in my life not at Texas Tech I would like to thank include Dr. Alessandro Provetti, Dr. Enrico Pontelli, Dr. Chitta Baral,

A special thanks to Dr. Son Cao Tran and Dr. Tu Phan. A large part of this work is thanks to their great collaboration, help and inspiration.
The Knowledge Representation Lab at Texas Tech University provided a sense of community and peer support. Together we form a group of people in parallel journeys. I thank all the members, past and present, for all the fun times and insightful discussions, and may our friendship continue into our later careers. Veena, Marcello, Jared, Justin, Yana, Daniella, Weijun, Forrest, Deb: Thanks a lot!

I wish to specially thank my lab-mate Sandeep Chintabathina for his company and friendship. Sandeep; it has been great to have shared with you this journey from the beginning, and very happy that now we finish this phase of life together. Your friendship is a treasure I keep close to my heart.

I may not have been able to finish this dissertation without the support of the Office of International Affairs at Texas Tech University. The part-time job they provided kept me financially afloat, their flexibility gave me the precious time I needed, and their kindness I will remember forever.

During my long stay in Lubbock I had a long list of roommates. Sitapathy, Raghavendra, Khwaja, Lohit, Prassana and Pankaj, Manoj, Sathya, Gautham and Jagganath. Thank you all for sharing the roof, the bread and the dirty dishes with me. Thanks also to Gamba, for his simple but invaluable company.

I also want to thank Tanecia, Eswar, Nachiappan, Kotti, Arunoday and Avinash for their friendship, company, moral support, and the many discussions that extended well into the night. Life would have been lot less colorful without them. Some of you may stay here in Texas Tech after I leave. I promise to keep in touch.

Other people who have touched my student life while in Texas Tech include Shridhar, Pratamesh, Shivaji, Vivek, Jagadish, Nitin, Rajkumar, Krishna Prasad, Sathya Marisetti, Sujit, Krishnan, Angus, Ken Tsai, Roger, . Thank you all for your friendship.

I also want to thank my friends from before coming to Texas Tech: Ritesh, Vijay, Chakravarthy, Mitesh, Luis Tari, Juan Loera and Pablo Loera, for their understanding and for staying in touch, even if it meant asking for many years “when are you going to graduate?”
A token of appreciation to Navi, Pad-kun, Alastor, Altair, Russell, Tachikoma and the blue sofa in the KR-Lab; the most valuable peace of furniture ever.

Finally, my greatest and deepest thanks to my mother Lucy and my brother Jabel. They continuously supported me morally, financially and emotionally during every stage of this PhD, during the good times and specially the bad times. I would not have made it without you. ¡GRACIAS!
TABLE OF CONTENTS

Acknowledgements ........................................................ iii
Abstract ................................................................. x
List of Tables ............................................................. xii
List of Figures ........................................................... xiii
1. INTRODUCTION ...................................................... 1
   1.1 The ASP Paradigm ................................................. 1
   1.2 Goals of this research .......................................... 2
      1.2.1 Approximating Action Theories ......................... 2
      1.2.2 System LUNA ................................................ 3
   1.3 Summary .......................................................... 4
2. BACKGROUND .......................................................... 6
   2.1 Answer Set Programming ........................................ 6
      2.1.1 Syntax ...................................................... 6
      2.1.2 Semantics .................................................. 7
   2.2 ASP Solvers ..................................................... 9
3. APPROXIMATING ACTION THEORIES ................................. 10
   3.1 The $\mathcal{AL}$ action language ............................... 10
      3.1.1 Syntax ..................................................... 10
      3.1.2 Semantics .................................................. 12
   3.2 The Conformant Planning Problem ............................. 16
   3.3 Approximations of $\mathcal{AL}$ action theories ................ 18
   3.4 Approximation $T_{\Pi}(\mathcal{D})$ ............................. 19
      3.4.1 The A-Prolog program $\pi(\mathcal{D})$ .................... 19
      3.4.2 The Transition diagram $T_{\Pi}(\mathcal{D})$ ................ 22
   3.5 An ASP Conformant Planner .................................... 23
      3.5.1 Properties of $\pi(\mathcal{P}, n)$ ......................... 24
      3.5.2 A Sufficient Condition for Completeness of CPASP ...... 26
3.6 An Extension for Problems with Disjunctive Initial State . . 28
3.7 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
  3.7.1 Evaluation of CPasp . . . . . . . . . . . . . . . . . . . . 31
    3.7.1.1 Planning Systems . . . . . . . . . . . . . . . . . . . 31
    3.7.1.2 Benchmarks . . . . . . . . . . . . . . . . . . . . . . 32
    3.7.1.3 Experimental Results . . . . . . . . . . . . . . . . . 34
  3.7.2 Evaluation of CPasp_m . . . . . . . . . . . . . . . . . . . 40
4. IMPROVING SYSTEM ACSOLVER - SYSTEM LUNA . . . . . . . . 45
  4.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
    4.1.1 The Language AC(C) . . . . . . . . . . . . . . . . . . . 46
      4.1.1.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . 46
      4.1.1.2 Semantics . . . . . . . . . . . . . . . . . . . . . . 48
    4.1.2 The Solver ACsolver . . . . . . . . . . . . . . . . . . . . 48
  4.2 An Algorithmic Framework . . . . . . . . . . . . . . . . . . . 49
    4.2.1 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . 49
      4.2.1.1 The Upper Closure . . . . . . . . . . . . . . . . . . 50
    4.2.2 AC(C) problems and solutions . . . . . . . . . . . . . . 51
    4.2.3 Model of computation . . . . . . . . . . . . . . . . . . . 51
    4.2.4 Constructive negation . . . . . . . . . . . . . . . . . . . 52
    4.2.5 Paths of computation . . . . . . . . . . . . . . . . . . . 54
    4.2.6 The simple class of programs . . . . . . . . . . . . . . . 54
    4.2.7 Properties of the model of computation . . . . . . . . . 55
    4.2.8 General Algorithm . . . . . . . . . . . . . . . . . . . . . 55
      4.2.8.1 Heuristics . . . . . . . . . . . . . . . . . . . . . . . 56
  4.3 An Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
    4.3.1 The input class of programs . . . . . . . . . . . . . . . . 57
    4.3.2 Simplified Graph . . . . . . . . . . . . . . . . . . . . . 58
    4.3.3 An algorithm for solving AC(C) problems . . . . . . . . . 58
    4.3.4 Computing Simplifications: The simp1() function . . . . 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3 Related work on integrating ASP and constraint solving techniques</td>
<td>108</td>
</tr>
<tr>
<td>8. CONCLUSIONS AND FUTURE WORK</td>
<td>110</td>
</tr>
<tr>
<td>8.1 Conclusions</td>
<td>110</td>
</tr>
<tr>
<td>8.2 Future work</td>
<td>111</td>
</tr>
<tr>
<td>Bibliography</td>
<td>113</td>
</tr>
<tr>
<td>Appendix: Programs</td>
<td>122</td>
</tr>
<tr>
<td>8.3 Programs from Chapter 4</td>
<td>122</td>
</tr>
<tr>
<td>8.3.1 House-Flowers domain</td>
<td>122</td>
</tr>
<tr>
<td>8.3.2 Ball drop, problem from action description $H$</td>
<td>123</td>
</tr>
<tr>
<td>8.4 Programs from Chapter 5</td>
<td>132</td>
</tr>
<tr>
<td>8.4.1 Bomb in the toilet with timers</td>
<td>132</td>
</tr>
<tr>
<td>8.4.2 Forrest bus rides</td>
<td>137</td>
</tr>
<tr>
<td>8.4.3 Airport pickups</td>
<td>142</td>
</tr>
</tbody>
</table>
ABSTRACT

Answer Set Programming (ASP) is a declarative programming paradigm for knowledge representation and reasoning. It can be used to represent and reason with recursive definitions, defaults and their exceptions, causal relations, beliefs, and various forms of incomplete information. ASP has been successfully used and applied to solve a variety of difficult (primarily NP-Hard) problems; however, sometimes finding solutions can be computationally expensive. Under the ASP methodology, a computational problem is encoded into a logic program that represents information relevant to the problem to be solved. The answer set semantics assigns to a program a collection of answer sets. Finding solutions to a problem is reduced to computing answer sets of the logic program that represent the problem.

The goal of this research is to improve the efficiency of solving computational problems with ASP by: improving problem representation techniques and improving existing answer set solvers. We illustrate our approaches by solving the problem of Conformant Planning.

First, we developed new representation techniques for the Conformant Planning problem. Second, we identified and implemented algorithmic improvements on the prototype solver ACsolver.

Conformant planning is a class of planning problems in which the planning agent is given incomplete information about the initial state of the planning domain and is asked to find a plan that will achieve a given goal regardless of the uncertainty present in the initial situation. Our proposed encoding methodology is based on the use of an approximation transition diagram, by which the resulting logic program is capable of correctly reasoning about the effects of actions in the presence of incomplete information. We show that our methodology is sound and under some circumstances complete, and that it extends the applicability and efficiency of ASP to this class of problems.

In the second part of the dissertation we concentrate on improving the prototype
solver \textit{ACsolver}. \textit{ACsolver} is a solver for the language \textit{AC(\mathbb{C})}, an extension to ASP that combines ASP with Constraint Logic Programming (CLP). \textit{ACsolver}'s design allows it to solve problems that involve real number constraints. Such problems are beyond the reach of conventional ASP solvers. We investigated the algorithmic and implementation properties of \textit{ACsolver}, and develop an algorithmic framework to design solvers of \textit{AC(\mathbb{C})} that overcome the limitations of \textit{ACsolver}'s algorithm. We implemented an algorithm from this framework over \textit{ACsolver}, to build an improved solver, \textit{LUNA}, and demonstrated the improvements in efficiency experimentally.

Furthermore, we apply both research goals to solve instances of conformant planning problems with numerical constraints.
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Conformant Turkey domain</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Lost Cleaner Domain</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>House-Flowers Domain</td>
<td>67</td>
</tr>
<tr>
<td>4.2</td>
<td>$\mathcal{H}$ Domains</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>BTC w/ Timers</td>
<td>72</td>
</tr>
<tr>
<td>5.2</td>
<td>Forrest</td>
<td>72</td>
</tr>
<tr>
<td>5.3</td>
<td>Airport Pickups</td>
<td>73</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

3.1 $D_{\text{bomb}}$, the bomb in the toilet theory ........................................ 12
3.2 BTC Domain ................................................................. 36
3.3 Gaspipipe domain ........................................................ 38
3.4 Conformant Turkey Domain .............................................. 42
3.5 Lost Cleaner Domain ...................................................... 44
4.1 House-Flowers Domain ................................................... 68
4.2 $\mathcal{H}$ Domains .......................................................... 69
5.1 Bomb in the toilet with timers Domain .................................... 73
5.2 Forrest Bus Drives Domain ............................................... 74
5.3 Airport Pickups Domain ................................................... 75
CHAPTER 1
INTRODUCTION

Answer Set Programming (ASP) is a *declarative* programming paradigm, oriented towards the tasks of knowledge representation and reasoning. As programs of a declarative language, programs of ASP are seen as collections of statements that represent relevant information about the problem to be solved. (as opposed to a program in an *imperative* language, where programs represent ordered sequences of instructions for a computer to perform). In order to find a solution to the problem encoded by a declarative program, an *inference engine* is used to process the program and compute the *models* or *consequences* of such program as defined by the semantics of the language. Then, solutions to the encoded problem can be extracted from the models of the program.

1.1 The ASP Paradigm

The ASP paradigm was introduced by Michael Gelfond and Vladimir Lifschitz in 1988 [38] and has its roots in research on non-monotonic logic and the semantics of negation in Logic Programming. ASP can be used to represent and reason about recursive definitions, defaults and their exceptions, causal relations, beliefs, intention, obligations, authorization policies, various forms of incomplete information, and is specially suited for knowledge-intensive applications.

Programming methodology based on the use of ASP was originally advocated by W. Marek and M Truszczynsky [60] and by I. Niemela [71]. Under the ASP methodology the ASP Programmer takes a given problem $P$ and:

- Encodes information relevant to the problem in a ASP Program $\Pi$.
- Reduces $P$ to a query $Q$ requiring computation of the model(s) of $\Pi$.
- Uses an inference engine, (a system that implements a collection of reasoning algorithms) to solve $Q$. 


ASP has been used effectively in a variety of programming tasks: as a decision support system for the space shuttle [72], in product configuration [85], and even in the areas of bio-informatics [10] zoology [16] and linguistics [17]. ASP is well suited for tackling difficult problems, usually of the NP-Hard complexity level. In such applications a common issue is that, as the size of the problems grow, finding solutions becomes computationally more and more expensive. In order to extend the applicability of ASP, there is a need to improve the efficiency of solving problems.

1.2 Goals of this research

The goal of this research is to improve the efficiency of solving computational problems with ASP. This goal is pursued in two ways:

- **Improving encoding techniques.** We develop a new technique for solving the *Conformant Planning* problem based on approximating action theories, which improves the efficiency and extends the applicability of ASP in this problem domain. We implement a planner using this technique and show that it is sound and, under certain circumstances, complete.

- **Improving existing solvers.** We study and identify algorithmic design improvements over the prototype solver *ACsolver*. A new solver, *LUNA*, is implemented following these designs, and we demonstrate experimentally the improvements in efficiency of our solver.

1.2.1 Approximating Action Theories

*Action theories* are formal models used to describe dynamic domains and reason about actions and their effects. Usually this is done by defining a transition diagram whose nodes represent possible states of the domain and arcs that correspond to actions that take the domain from one state to another. In this work, we concentrate on the transitions diagrams that can be described using the action description language *ACL* [9].
Action theories can be used for multiple purposes including temporal projection, planning and diagnosis. Planning problems are reduced to finding a path (a sequence of actions) that, from a given initial state, leads to a final state that satisfies some goal condition. The conformant planning problem is the problem of finding a sequence of actions which achieves a goal from any possible initial state of the world, given incomplete information about this initial state.

In this work, we describe a methodology for the design and implementation of conformant planners. This methodology is rooted in the notion of approximations. Given an action theory $D$ and its corresponding transition diagram $T(D)$, an approximation of $D$ is a transition diagram whose nodes are partial states of $T(D)$ and whose arcs are "sound" with respect to the transitions of $T(D)$. Using approximations allows us to reason about the effect of actions under incomplete information and find solutions to conformant planning problems. Furthermore, using approximations can substantially improve the efficiency of ASP based planners, allowing us to solve problems that previous planners could not solve. However, a drawback of this approach is the possible incompleteness of our approximation based planners, i.e. there are solvable problems for which our planner may not find a solution.

1.2.2 System LUNA

Given the kind of difficult problems ASP programmers usually tackle, there is always a motivation to find improvements and solve problems previously beyond reach. An area in which a lack of efficiency limits the applicability of ASP are situations in which a program of ASP contains variables ranging over large domains. This is specially the case in problems dealing with numerical computations and that require reasoning about constraints over integer or real numbers. This problem was addressed by the prototype solver ACsolver.

ACsolver is a solver for the language $\mathcal{AC}(\mathcal{C})$ [70], a declarative programming language for knowledge representation that extends the syntax and semantics of ASP.
ACsolver combines ASP solving methods with Constraint Logic Programming (CLP) techniques. The $\mathcal{AC}(\mathcal{C})$ extensions are often capable of solving problems which are impossible to solve with more traditional ASP solving techniques. Further investigation of the language and the development of more efficient solvers is desired for extending the domain of applicability of the $\mathcal{AC}(\mathcal{C})$ approach to ASP.

In this dissertation, we present work to improve the prototype solver ACsolver, and implement these improvements into a new solver: LUNA. Even though ACsolver existed, it was not thoroughly tested. As part of this work we also test system LUNA to solve problems more reliably.

We show the efficiency gains achieved in LUNA and how it allows us to solve problems that were not solvable before. In the process of developing the improved system, a new algorithmic framework for finding solutions to $\mathcal{AC}(\mathcal{C})$ programs was developed. This framework can be used for the development of multiple different algorithms and opens a broad avenue for further research in improving the efficiency of $\mathcal{AC}(\mathcal{C})$ solvers.

1.3 Summary

The contributions of this dissertation are itemized as follows:

- A logic programming based approximation of $\mathcal{AL}$ domain descriptions suitable for solving conformant planning problems when the initial state is represented by a set of fluent literals

- An ASP conformant planner based on this approximation.
  - planner CPasp.

- An extension to CPasp to deal with initial situations that are described using disjunction.

- Experimental results on the performance of CPasp.

- An algorithmic framework to find models of programs of the $\mathcal{AC}(\mathcal{C})$ language.
• An implementation of an instance of this framework over the prototype system $ACsolver$.

  – System $LUNA$.

• Experimental results of the performance gains of system $LUNA$.

• Examples of $\mathcal{AC}(\mathcal{C})$ programs to solve instances of conformant planning problems with numerical constraints, a kind of problems not attempted before.

This dissertation is organized as follows: Chapter 2 covers the necessary background on the ASP Paradigm, its language, syntax and semantics. Chapter 3 presents action theories, the conformant planning problem and introduces the notion of approximation. A methodology for building conformant planners based on the concept of approximating action theories is then presented. Theorems for the soundness and completeness of such planners are also presented. Chapter 4 introduces the language $\mathcal{AC}(\mathcal{C})$ and the prototype solver $ACsolver$. An algorithmic framework for finding solutions to programs of $\mathcal{AC}(\mathcal{C})$ is then shown. Then, efficiency results of the improved solver $LUNA$ are presented. Chapter 5 brings together the different aspects of this dissertation and presents programs to solve conformant planning problems with numerical constraints, problems which are not solvable by previously described methods. Chapter 6 presents proofs for the theorems in Chapter 3. Chapter 7 briefly describes related work to this research. Finally, Chapter 8 gives conclusions and future work.
CHAPTER 2
BACKGROUND

This chapter provides an introduction to the Answer Set Programming paradigm.

2.1 Answer Set Programming

Answer Set Programming (ASP) is a form of declarative programming oriented towards the tasks of knowledge representation and solving difficult search problems \cite{38} \cite{56}. ASP has its roots in research in non-monotonic logic and in the semantics of negation in logic programming \cite{42} \cite{39}. Programs of ASP are usually written in the Answer Set Prolog (A-Prolog) language. A-Prolog is a powerful language for knowledge representation and reasoning. It can be used to represent and reason about a variety of common-sense constructs like defaults, causal properties of actions and fluents, beliefs and intentions, various types of incomplete information and recursive definitions. The language is elaboration tolerant and has strong mathematical foundations and well developed syntax and semantics.

2.1.1 Syntax

The syntax of A-Prolog is determined by a typed signature $\Sigma$ consisting of types and properly typed object constants, function symbols and predicate symbols. A term of $\Sigma$ is either a constant or an expression $f(t_1, \ldots, t_n)$ where $f$ is a function symbol of arity $n$, and $t_1, \ldots, t_n$ are terms of the proper types. An atom is of the form $p(t_1, \ldots, t_n)$ where $p$ is an $n$-ary predicate symbol, and $t_1, \ldots, t_n$ are terms of the proper types. A literal is either an atom $\alpha$ or its negation $\neg\alpha$. Literals $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called complimentary. We will denote by $\bar{l}$ the literal complimentary to literal $l$.

Definition 1 (Rules of A-Prolog). A rule of A-Prolog is an expression of the form

\[ l_0 \text{ or } \ldots \text{ or } l_k \leftarrow l_{k+1}, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n. \quad (2.1) \]
where l’s are literals of \( \Sigma \).

The connective or is called \textit{epistemic disjunction} and the connective not is called \textit{default negation}. Intuitively the above rule states in english: \textit{If you believe in} \( l_{k+1}, \ldots, l_m \) \textit{and have no reason to believe in any of} \( l_{m+1}, \ldots, l_n \), \textit{then believe one of} \( l_0, \ldots, l_k \). Let \( r \) be a rule as defined above. By \( \text{head}(r) \) we will denote the set of literals \( \{l_0, \ldots, l_k\} \). \( \text{body}(r) \) will denote the set \( \{l_{k+1}, \ldots, l_m, l_{m+1}, \ldots, l_n\} \). In a similar way \( \text{pos}(r) \) will be used to denote \( \{l_{k+1}, \ldots, l_m\} \) and \( \text{neg}(r) \) will be used to denote \( \{l_{m+1}, \ldots, l_n\} \).

A rule whose body is empty will be called a \textit{fact}, while a body with no head, where \( k = 0 \), will be called a \textit{constraint}.

**Definition 2 (Programs of A-Prolog ).** A program of A-Prolog is a pair \( \langle \Sigma, \Pi \rangle \) where \( \Sigma \) is a signature and \( \Pi \) is a collection of rules over \( \Sigma \).

Whenever possible, we will identify a program by its collection of rules \( \Pi \), and use \( \Sigma(\Pi) \) to refer to its signature. Rules containing variables (written in capital letters) will be viewed as shorthand of their \textit{ground instantiations}, the result of replacing the variables by all the terms of the corresponding type.

**Example 1 (Program of A-Prolog ).**

\[
\begin{align*}
p(a) & \leftarrow \text{not } q(a). \quad (2.2) \\
p(b) & \leftarrow \text{not } q(b). \quad (2.3) \\
q(a). \quad (2.4)
\end{align*}
\]

**2.1.2 Semantics**

The semantics of programs of A-Prolog are given by the Answer Set Semantics \[38\] \[42\]. The Answer Set Semantics assigns to a logic program \( \Pi \) a collection of (answer sets) – consistent sets of literals from \( \Sigma(\Pi) \) – that correspond to a possible set of beliefs. Intuitively, these beliefs follow the \textit{rationality principle} which states that \textit{one shall not believe anything one is not forced to believe}. The definition of
answer sets will be given in two parts, first for programs without default negation and then for programs with default negation, but before we shall introduce some terminology.

A *partial interpretation* is a consistent set of literals from a signature $\Sigma$. We say that a partial interpretation $S$ *satisfies* a rule without default negation $r$ if $\text{head}(r) \cup S \neq \emptyset$ whenever $\text{pos}(r) \subseteq S$.

**Definition 3** (Answer Set - part one). A partial interpretation $S$ is an *Answer Set* of a program without default negation $\Pi$ if $S$ is minimal (by set-theoretic inclusion) among the partial interpretations of $\Sigma(\Pi)$ satisfying every rule of $\Pi$.

Programs made of rules whose bodies lack default negation and whose heads have at most one literal are called *definite* programs. It has been shown that definite programs have at most one answer set [73] [33].

Before stating the second part of the semantics, we will introduce the concept of *reduct*, which will allow us to connect the definition of Answer Sets of arbitrary programs with the definition of Answer Sets for programs without default negation.

**Definition 4** (Answer Set - part two). The reduct of a program with default negation $\Pi$ relative to a partial interpretation $S$, denoted $\Pi^S$, is the set of rules

$$l_0 \text{ or } \ldots \text{ or } l_k \leftarrow l_{k+1}, \ldots, l_m.$$ 

for all rules (2.1) in $\Pi$ such that $\{l_{m+1}, \ldots, l_n\} \cup S \neq \emptyset$.

Notice that by definition, the reduct of a program does not contain default negation.

**Definition 5** (Answer Set - part two). A partial interpretation $S$ is an *Answer Set* of an A-Prolog program $\Pi$ if $S$ is an answer set of $\Pi^S$.

**Example 2** (Answer Set). Let $\Pi$ be the A-Prolog program from Example 1 and let $S = \{q(A), p(b)\}$. It is easy to verify that the reduct of $\Pi$ with respect to $S$, $\Pi^S$
is

\[ p(b). \quad (2.5) \]
\[ q(a). \quad (2.6) \]

Notice that \( S \) is an answer set of the definite program \( \Pi^S \), therefore \( S \) is an answer set of \( \Pi \).

2.2 ASP Solvers

There exists a variety of implemented ASP solvers, systems to compute answer sets of ASP programs. The list includes Smodels [77], DLV [20], Surya [66], Cmodels [51], ASSAT [58], and Clasp [35]. The design of many of these solvers involve enhancements of the Davis-Putnam-Loveland procedure [26], similar to the one used in SAT solvers [47].

The efficiency provided by these solvers has found many applications, ranging from building decision support systems for the space shuttle control [72], encoding planning problems [27], and checking medical invoices for health insurance companies [12], to solving problems involving product configuration [85], wire-routing [34] data-integration [50], bio-informatics [10], zoology [16], and linguistics [17].
CHAPTER 3
APPROXIMATING ACTION THEORIES

In this chapter we present our work on approximating action theories. First, we review the action description language \( \mathcal{AL} \) used for modeling dynamic domains as transition diagrams. Nodes of such diagrams represent states of the domain and arcs correspond to actions that take the domain from one state to another. Then we review the concept of a conformant planning problem and its solutions. Afterwards, we present the notion of approximation of a transition diagram of a dynamic system developed in [79] [82] [83] [86]. Then, we define a particular approximation, first introduced in [82], using a logic program of ASP and show how this approximation (and the defining program) can be used to build an ASP based conformant planner. Theorems about the soundness and completeness of this planner follow. Then, we extended the conformant planner to allow it to work with conformant planning problems with more complex descriptions of the initial situation. Finally, we present experimental results on the efficiency of our conformant planners.

3.1 The \( \mathcal{AL} \) action language

3.1.1 Syntax

The signature \( \Sigma \) of an \( \mathcal{AL} \) action theory consists of two disjoints sets of symbols: a set \( F \) of fluents and a set \( A \) of elementary actions. An action is a non-empty set of elementary actions. A fluent literal (or literal for short) is either a fluent \( f \) or its negation \( \neg f \). If \( l \) is a literal, by \( \neg l \) we denote the fluent literal complementary to \( l \), i.e., \( \neg (f) = \neg f \) and \( \neg (\neg f) = f \). An \( \mathcal{AL} \) action theory is a collection of statements of the forms:

\[
\begin{align*}
e & \text{ causes } l \text{ if } p & (3.1) \\
l & \text{ if } p & (3.2) \\
impossible & \text{ a if } p & (3.3)
\end{align*}
\]
where \( e \) is an elementary action, \( a \) is an action, and \( p \) is a set of fluent literals from the signature \( \Sigma \). The set \( p \) is called the \textit{preconditions} of the corresponding statement. When \( p \) is empty, the if part of the statement is omitted. Intuitively, statement (3.1), called a \textit{dynamic causal law}, says that, if \( e \) is executed in a state satisfying \( p \) then \( l \) will hold in any resulting state. Statement (3.2), called a \textit{static causal law}, says that any state satisfying \( p \) must satisfy \( l \). Statement (3.3) is an \textit{impossibility condition}, and says that action \( a \) cannot be performed in any state satisfying \( p \).

To illustrate the syntax of \( \mathcal{AL} \), let us consider an instance of (a variant of) the bomb in the toilet domain [45].

Example 3. Suppose there are two packages \( p_1 \) and \( p_2 \) and two toilets \( t_1 \) and \( t_2 \). Each of the packages may contain a bomb which can be disarmed by dunking the package in a toilet. Dunking a package into a toilet also clogs the toilet. Flushing a toilet unclogs it. We are safe only if both packages are disarmed.

Figure 3.1 shows an action theory of \( \mathcal{AL} \), denoted by \( D_{\text{bomb}} \), that describes the domain\(^1\).

\[ \text{Figure 3.1 shows an action theory of } \mathcal{AL}, \text{ denoted by } D_{\text{bomb}}, \text{ that describes the domain} \]

There are four impossibility statements in the action theory. The first one says that “it is impossible to dunk a package into a toilet 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 statements 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 clogged toilet”.

In addition to the impossibility statements, the action theory also includes statements to describe the effects of actions \textit{dunk} and \textit{flush} and the relationship be-

\(^1\text{Note that in the description of an action theory, we often use typed variables. A statement with variables are understood as a shorthand for the collection of its ground instances.} \)
Meta variables:
- $p_i$'s stand for packages, $i \in \{1, 2\}, p_1 \neq p_2$
- $t_j$'s stand for toilets, $j \in \{1, 2\}, t_1 \neq t_2$.

Fluents:
- $\text{armed}(p_i)$: package $p_i$ contains the bomb
- $\text{clogged}(t_j)$: toilet $t_j$ is clogged
- $\text{safe}$: all the bombs are disarmed

Actions:
- $\text{dunk}(p_i, t_j)$: dunk package $p_i$ into toilet $t_j$
- $\text{flush}(t_j)$: flush toilet $t_j$

Action theory:
- $\text{impossible}\ \{\text{dunk}(p_i, t_j), \text{flush}(t_j)\}$
- $\text{impossible}\ \{\text{dunk}(p_i, t_j), \text{dunk}(p_2, t_j)\}$
- $\text{impossible}\ \{\text{dunk}(p_i, t_1), \text{dunk}(p_i, t_2)\}$
- $\text{impossible}\ \text{dunk}(p_i, t_j)$ if $\text{clogged}(t_j)$
- $\text{dunk}(p_i, t_j)$ causes $\neg \text{armed}(p_i)$
- $\text{dunk}(p_i, t_j)$ causes $\text{clogged}(t_j)$
- $\text{flush}(t_j)$ causes $\neg \text{clogged}(t_j)$
- $\text{safe}$ if $\neg \text{armed}(1), \neg \text{armed}(2)$

Figure 3.1. $D_{\text{bomb}}$, the bomb in the toilet theory

tween fluent $\text{safe}$ and fluents $\text{armed}$'s.

3.1.2 Semantics

Intuitively, an action theory of $\mathcal{AL}$ describes a transition diagrams whose nodes correspond to possible states of the domain and whose arcs correspond to actions that take the domain from one state to another. The semantics of $\mathcal{AL}$ provide a precise definition of such diagram. Before continuing we will introduce some useful terminology and notation.

Given an action theory $D$, a set of literals $s$ is *consistent* if it does not contain two complimentary literals. We say $s$ is *complete* if for every fluent $f$ either $f$ or $\neg f$ belong to $s$. A set of fluent literals $s$ is *closed* under a static causal law (3.2) if $l \in s$ whenever $p \subseteq s$. For a set of literals $s$, by $\text{Cn}_D(s)$, we denote the smallest set of literals that includes $s$ and is closed under the set of static causal laws of $D$.  

---
A state $\sigma$ is a complete and consistent sets of literals closed under the set of static causal laws of $\mathcal{D}$. An action $b$ is said to be prohibited in state $\sigma$ if $\mathcal{D}$ contains an impossibility condition (3.3) such that $p \subseteq \sigma$ and $a \subseteq b$. For an action $a$ and a state $\sigma$, $E(a, \sigma)$ stands for the set of all literals $l$ such that $\mathcal{D}$ contains a law (3.1) with $p \subseteq \sigma$ and $e \in a$. Elements of $E(a, \sigma)$ are called direct effects of the execution of $a$ in $\sigma$.

An action theory $\mathcal{D}$ describes a transition diagram $T(\mathcal{D})$. The nodes of $T(\mathcal{D})$ are states of $\mathcal{D}$ and arcs are labeled with actions. The transitions of $T(\mathcal{D})$ are defined as follows.

**Definition 6 ([62]).** For an action $a$ and two states $\sigma$ and $\sigma'$, a transition $\langle \sigma, a, \sigma' \rangle \in T(\mathcal{D})$ iff $a$ is not prohibited in $\sigma$ and

$$\sigma' = Cn_\mathcal{D}(E(a, \sigma) \cup (\sigma \cap \sigma'))$$

(3.4)

Intuitively, if a transition $\langle \sigma, a, \sigma' \rangle$ belongs to $T(\mathcal{D})$ this means that after the execution of action $a$ in $\sigma$ the system will move to state $\sigma'$. Such $\sigma'$ is often referred to as a possible successor state of $\sigma$ as the result of the execution of $a$. When $\sigma$ and $a$ are clear from the context, we simply say that $\sigma'$ is a possible successor state.

The following example illustrates the notion of transition and successor state.

**Example 4.** Consider the action theory $\mathcal{D}_{\text{bomb}}$ from Example 3. Let

$$\sigma_0 = \{\text{armed}(1), \text{armed}(2), \neg\text{clogged}(1), \neg\text{clogged}(2), \neg\text{safe}\}$$

and

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

Then

$$\sigma_1 = \{\neg\text{armed}(1), \neg\text{armed}(2), \text{clogged}(1), \text{clogged}(2), \text{safe}\}$$
is the unique successor state of $s_0$, i.e., $\langle s_0, a, s_1 \rangle \in T(D_{bomb})$, because

$$\text{Cl}_{D_{bomb}}(\text{de}(a, s_0) \cup (s_0 \cap s_1)) =$$

$$\text{Cl}_{D_{bomb}}((-\text{armed}(1), -\text{armed}(2), \text{clogged}(1), \text{clogged}(2)) \cup \emptyset) =$$

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

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

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

and both $\neg\text{armed}(1)$ and $\neg\text{armed}(2)$ hold in $\sigma$.

Now let

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

Then,

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

is the unique successor state of $s_0$, i.e., $\langle s_0, b, s_2 \rangle \in T(D_{bomb})$. 

In general, even if an action $a$ is not prohibited in a state $\sigma$, a successor state may not exist. The following definition captures this formally.

**Definition 7 (Consistent Action Theory).** An action theory $D$ is *consistent* if for any state $\sigma$ and action $a$ which is not prohibited in $\sigma$ there exists at least one possible successor state $\sigma'$. i.e. $\langle \sigma, a, \sigma' \rangle \in T(D)$. 

**Example 5 (Consistent and Inconsistent Action Theories).** Consider the following action theory:

$$D_0 = \left\{ \begin{array}{ll} e \text{ causes } f \text{ if } g \\ e \text{ causes } \neg f \text{ if } h \end{array} \right.$$
We have that $e$ is executable in $\sigma = \{f, g, h\}$. If $\sigma'$ is a successor state of $\sigma$ then it is easy to see that both $f$ and $\neg f$ belong to $\sigma'$. This is a contradiction because $s$ is consistent. Hence, there exists no successor state for $\sigma$. According to the above definition, this implies that $D_0$ is inconsistent.

However, if we add to $D_0$ the following impossibility condition

$$\text{impossible } e \text{ if } g, h$$

then the action theory will become consistent because $e$ cannot be executed in any state in which both $g$ and $h$ holds. Hence, at most one of the above dynamic causal laws takes effect and the inconsistency in the previous case would not occur. \[\square\]

In this work, we consider only consistent action theories. It is also important to notice that action theories of $\mathcal{AL}$ can be non-deterministic. For an action $a$ and state $\sigma$ there may be more than one successor state.

**Definition 8 (Deterministic Action Theory).** An action theory $D$ is deterministic if for any state $\sigma$ and action $a$, there exists at most one state $\sigma'$ such that $\langle \sigma, a, \sigma' \rangle \in T(D)$.

It is easy to see that if an action theory $D$ does not contain any static causal laws then it is deterministic. In the presence of static causal laws, an action theory, however, may be non-deterministic. The following example shows such an action theory.

**Example 6 (Non-Deterministic Action Theory).** Consider the following action theory:

$$D_1 = \begin{cases} 
  e \text{ causes } f \\
  g \text{ if } f, \neg h \\
  h \text{ if } f, \neg g
\end{cases}$$

Let $\sigma_0 = \{\neg f, \neg g, \neg h\}$. We can verify that $\langle \sigma_0, e, \sigma_1 \rangle \in T(D_1)$ and $\langle \sigma_0, e, \sigma_2 \rangle \in T(D_1)$ where $\sigma_1 = \{f, h, \neg g\}$ and $\sigma_2 = \{f, g, \neg h\}$. Hence, by definition, $D_1$ is non-deterministic. \[\square\]
We now introduce some terminology used to describe properties of a transition\ diagram $T(D)$. By convention, from now on we will use letters $\alpha$ and $\sigma$ (possibly indexed) to denote actions and states respectively. A \textit{model} $M$ of a chain of events $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$ is an alternate sequence of states and actions $\langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ s.t. $\langle \sigma_i, a_i, \sigma_{i+1} \rangle \in T(D)$ for $0 \leq i < n$. $\sigma_0$ is referred to as the \textit{initial state} of $M$ and $\sigma_n$ is referred to as the \textit{final state} of $M$.

**Definition 9 (Entailment).** We say that a model $M = \langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ \textit{entails} a set of literals $s$, written as $M \models s$, if $s \subseteq \sigma_n$. Furthermore, we sometimes write $\langle \sigma_0, \alpha, \sigma_n \rangle \in T(D)$ to denote that there is a model of $\alpha$ whose initial state and final state are $\sigma_0$ and $\sigma_n$, respectively.

An action $\alpha$ is \textit{executable} in a state $\sigma$ if there is a state $\sigma'$ s.t. $\langle \sigma, \alpha, \sigma' \rangle \in T(D)$. A chain of events $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$ is executable in $\sigma$ if either (i) $n = 0$ (i.e., $\alpha$ is an empty chain of events), or (ii) $a_0$ is executable in $\sigma$ and for every $\sigma'$ s.t. $\langle \sigma, a, \sigma' \rangle \in T(D)$, $\langle a_1, \ldots, a_{n-1} \rangle$ is executable in $\sigma'$.

### 3.2 The Conformant Planning Problem

In many of real world applications, an agent may not have complete knowledge of the current state of the domain. Instead, its knowledge is limited to a set of literals that might be incomplete. Under such circumstances it is not possible for the agent to directly reason about the effects of actions using the transition diagram of the domain. It is, however, important for the agent to be able to reason about the effects of actions in the presence of incomplete information about the world. The conformant planning problem is the problem of finding a sequence of actions which achieves a goal from any possible initial state of the world, given information about the initial states and the possible effects of actions. The conformant planning problem as discussed here has been previously discussed in [15, 18, 19, 21, 22, 23, 29, 32, 75, 78]. An ASP approach for this problem is discussed in [81, 82, 83]. In what is left of this section, we will now formally describe the conformant planning problem in the context of action theories of $\mathcal{AL}$.
Let $\mathcal{D}$ be an action theory of $\mathcal{A}L$. A set $s$ of literals is a partial state if it is a subset of some state and it is closed under the set of static causal laws of $\mathcal{D}$. From now on, we will use the (possible indexed) symbols $\sigma$ to denote a state and $s$ to denote a partial state. For a partial state $s$, a completion of $s$ is a state $\sigma$ such that $s \subseteq \sigma$ and by $\text{comp}(s)$ we denote the set of all completions of $s$. For a set of partial states $S$ by $\text{comp}(S)$ we denote the set $\bigcup_{s \in S} \text{comp}(s)$.

**Definition 10 (Conformant Planning Problem).** A conformant planning problem $P$ is a tuple $\langle \mathcal{D}, \Gamma, G \rangle$ where $\Gamma$ is a set of possible initial states and $G$ is a set of literals.

A solution of a conformant planning problem is defined next.

**Definition 11 (Solution to a Conformant Planning problem).** A chain of events $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$ is a solution or a conformant plan of a conformant planning problem $P = \langle \mathcal{D}, \Gamma, G \rangle$ if, for every $\sigma \in \Gamma$, $\alpha$ is executable in $\sigma$ and for every model $M$ of $\alpha$ with initial state $\sigma$, $M \models G$.

Let us illustrate these definitions using the bomb in the toilet example.

**Example 7.** Consider the action theory $\mathcal{D}_{\text{bomb}}$ and let $\Gamma = \text{comp}(\emptyset)$ and $G = \{\text{safe}\}$. Then, $P_{\text{bomb}} = \langle \mathcal{D}, \Gamma, G \rangle$ is a planning problem.

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 two solutions of $P_{\text{bomb}}$. The first one is a sequential plan, whereas the second one is a parallel plan.

\[\square\]
For the first part of this discussion, we will limit ourselves to conformant planning problems where the set of possible initial states $\Gamma$ can be described by a single partial state $s^0$, i.e. $\Gamma = \text{comp}(s^0)$

### 3.3 Approximations of $\mathcal{AL}$ action theories

Our approach to solving conformant planning problems is based on using approximations of action theories [79, 82, 83, 86] in order to be able to reason with the incomplete information present in the problem. In this section, we present the definition of approximation first shown in [83] and discuss how an approximation can be used to find solutions of a conformant planning problem.

**Definition 12 (Approximation [83]).** A transition diagram $T'(\mathcal{D})$ is an approximation of $\mathcal{D}$ if the following conditions are satisfied:

1. Nodes of $T'(\mathcal{D})$ are partial states of $T(\mathcal{D})$ and arcs of $T'(\mathcal{D})$ are labeled by actions.

2. If $\langle s, a, s' \rangle \in T'(\mathcal{D})$ then for every $\sigma \in \text{comp}(s)$,

   (a) $a$ is executable in $\sigma$, and

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

Intuitively, the first condition states that an approximation $T'(\mathcal{D})$ is a transition diagram between partial states while the second condition requires $T'(\mathcal{D})$ to be sound with respect to $T(\mathcal{D})$.

It is not difficult to see that according to definition 12 the following observation holds.

**Observation 1 ([83]).** Let $T'(\mathcal{D})$ be an approximation of $\mathcal{D}$. Then, for every chain of events $\alpha$, if $\langle s, \alpha, s' \rangle \in T'(\mathcal{D})$, then for every $\sigma \in \text{comp}(s)$:

1. $\alpha$ is executable in $\sigma$.

2. $s' \subseteq \sigma'$ for every $\sigma'$ such that $\langle \sigma, \alpha, \sigma' \rangle \in T(\mathcal{D})$. 

18
The following relationship between conformant plans and paths of an approximation of $\mathcal{D}$ follows from Observation 1.

**Observation 2.** Let $T'(\mathcal{D})$ be an approximation of $\mathcal{D}$. If $(s, \alpha, s') \in T'(\mathcal{D})$ and $G \subseteq s'$ then $\alpha$ is a solution of the conformant planning problem $\langle \mathcal{D}, \text{comp}(s), G \rangle$.

Observation 2 shows us a way to solve a conformant planning problem $\langle \mathcal{D}, \Gamma, G \rangle$: look for a path in an approximation that leads to a partial state satisfying the goal.

### 3.4 Approximation $T^{ip}(\mathcal{D})$

In this section, we define an approximation of $T(\mathcal{D})$ called $T^{ip}(\mathcal{D})$. The transitions of $T^{ip}(\mathcal{D})$ will be defined by an *A-Prolog* program $\pi(\mathcal{D})$.

#### 3.4.1 The *A-Prolog* program $\pi(\mathcal{D})$

The signature of $\pi(\mathcal{D})$ includes terms corresponding to fluent literals and actions of $\mathcal{D}$, as well as non-negative integers used to represent time steps. We often write $\pi(\mathcal{D}, n)$ to denote the restriction of the program $\pi(\mathcal{D})$ to time steps between 0 and $n$. Atoms of $\pi(\mathcal{D})$ are formed by the following (typed) predicate symbols:

- $\text{fluent}(F)$ is true if $F$ is a fluent.
- $\text{literal}(L)$ is true if $L$ is a fluent literal.
- $\text{contrary}(L, L')$ is true if $L$ and $L'$ are contrary fluent literals.
- $\text{h}(L, T)$ is true if the fluent literal $L$ holds at time-step $T$.
- $\text{o}(E, T)$ is true if the elementary action $E$ occurs at time-step $T$.
- $\text{de}(L, T)$ is true if literal $L$ is a direct effect of an action that occurs at time step $T - 1$; and
- $\text{ph}(L, T)$ is true if literal $L$ possibly holds at time $T$. 

19
In our representation, letters $T$, $F$, $L$, $A$, and $E$ (possibly indexed) are used to represent types timestep, fluent, fluent literal, action, and elementary action correspondingly. Similarly letters $t$, $f$, $l$, $a$, and $e$ will be used to represent the corresponding constants. Moreover, we also use some shorthands: if $a$ is an action then $o(a, t) = \{o(e, t) : e \in a\}$. If $p$ is a set of literals, and $F$ is either $h$, $de$, or $ph$, then $F(p, t) = \{F(l, t) : l \in p\}$ and $not\ F(p, t) = \{not\ F(l, t) : l \in p\}$. For a literal $l$, by $\bar{l}$ we denote its complement, i.e. $\bar{\bar{l}} = \bar{l}$ and $\bar{not\ l} = l$. For a set of literal $p$, $\bar{p} = \{\bar{l} : l \in p\}$.

The rules $^2$ of $\pi(D)$ are build from the rules of $D$ as follow:

1. Dynamic causal laws: for each statement of the form (3.2) in $D$, the rules:

$$h(l, T + 1) \leftarrow o(e, T), h(p, T). \tag{3.5}$$

$$de(l, T + 1) \leftarrow o(e, T), h(p, T). \tag{3.6}$$

belong to $\pi(D)$. The first rule states that if the elementary action $e$ occurs at time step $T$ and the precondition $p$ holds at that time step then $l$ holds afterward. The second rule says that if $e$ occurs at time step $T$ and the precondition $p$ holds then the fluent literal $l$ is a direct effect of $e$. Since the agent’s knowledge about the state of the world at the time step $T$ might be incomplete, we also add to $\pi(D)$ the rule to define what possibly holds after the execution of an action $e$ at the time step $T$:

$$ph(l, T + 1) \leftarrow o(e, T), not\ h(\bar{p}, T). \tag{3.7}$$

2. Static causal laws: for each statement of the form (3.2) in $D$, $\pi(D)$ contains the rules:

$$h(l, T) \leftarrow h(p, T) \tag{3.8}$$

$$ph(l, T) \leftarrow ph(p, T) \tag{3.9}$$

$^2$In each rule, predicates defining sorts of variables are omitted for brevity.
The first rule states that if the precondition \( p \) holds at \( T \) then so does \( l \). The second rule states that if \( p \) possibly holds at \( T \) then so does \( l \).

3. **Executability conditions:** for each statement of the form (3.3) in \( D \), \( \pi(D) \) contains the following rule:

\[
\leftarrow o(a,T), \text{not } h(\neg p,T)
\]

(3.10)

This rule states that if the precondition \( p \) possibly holds at time step \( T \) then the action \( a \) cannot occur at that time step.

4. **Inertia:** \( \pi(D) \) contains the following rules.

\[
h(L,T) \leftarrow \text{not ph}(\neg L,T).
\]

(3.11)

\[
\text{ph}(L,T+T) \leftarrow \text{not } h(\neg L,T), \text{not } de(\neg L,T).
\]

(3.12)

The first rule encodes the inertial law and solves the frame problem [63] and says that \( L \) holds at the time moment \( T \) if its negation does not possibly hold at \( T \). The second rule completes the definition of the predicate \( \text{ph} \). It defines what possibly holds by inertia: a fluent literal possibly holds if:

(a) its negation does not hold at the previous time step; and

(b) its negation is not a direct effect of an action occurring in the previous time step.

5. **Auxiliary rules:** \( \pi(D) \) also contains the following rules:

\[
\leftarrow h(F,T), h(\neg F,T)
\]

(3.13)

\[
\text{literal}(F) \leftarrow \text{fluent}(F)
\]

(3.14)

\[
\text{literal}(\neg F) \leftarrow \text{fluent}(F)
\]

(3.15)

\[
\text{contrary}(F,\neg F) \leftarrow \text{fluent}(F)
\]

(3.16)

\[
\text{contrary}(\neg F,F) \leftarrow \text{fluent}(F)
\]

(3.17)
The first constraint impedes two contrary fluent literals from holding at the same time. The last four rules are used to define fluent literals and complementary fluent literals.

For an action \( a \) and a partial state \( s \), let

\[
\Pi(D, a, s) = \pi(D, 1) \cup h(s, 0, 0) \cup o(a, 0).
\]  

(3.18)

Then, the program \( \Pi(D, a, s) \) has the following property.

**Proposition 1.** Let \( D \) be an action description of \( \mathcal{AL} \), \( s \) be a partial state and \( a \) be an action. If \( \Pi(D, a, s) \) is consistent then it has a unique answer set \( B \) and \( s' = \{ l | h(l, 1) \in B \} \) is a partial state.

**Proof.** See Appendix 6.2.1. \( \square \)

### 3.4.2 The Transition diagram \( T^{lp}(D) \)

We now define the transition diagram \( T^{lp}(D) \) based on the program \( \pi(D) \) as follows:

**Definition 13.** Let \( T^{lp}(D) \) be a transition diagram such that

1. Nodes of \( T^{lp}(D) \) are partial states of \( D \) and arcs of \( T^{lp}(D) \) are actions of \( D \).
2. \( (s, a, s') \in T^{lp}(D) \) iff \( \Pi(D, a, s) \) is consistent and \( s' = \{ l | h(l, 1) \in A \} \) where \( A \) is the answer set of \( \Pi(D, a, s) \).

The next theorem states that \( T^{lp}(D) \) is indeed an approximation of \( T(D) \).

**Theorem 1.** If \( D \) is consistent then \( T^{lp}(D) \) is a deterministic approximation of \( T(D) \).

**Proof.** See Appendix 6.2.2. \( \square \)

It is important at this point to remark that the approximation shown here is applicable for action languages whose semantics can be specified by a transition diagram, and not tied specifically to action language \( \mathcal{AL} \).
3.5 An ASP Conformant Planner

Let $P = \langle D, \text{comp}(s^0), G \rangle$ be a conformant planning problem. Since $T^{lp}(D)$ is an approximation of $D$, we can use the program $\pi(D)$ to find solutions of $P$.

In this section we will show how we can use $\pi(D)$ to build an ASP conformant planner to compute solution of $P$. Let us denote by $\pi(P, n)$ such planner. $\pi(P, n)$ takes as input a planning problem $P$ and an integer $n$. The answer sets of $\pi(P, n)$ contains solutions to $P$ of length $n$.

The signature of $\pi(P, n)$ includes the signature of $\pi(D)$. Rules of $\pi(P, n)$ contain the rules of $\pi(D)$ and the following rules:

1. **Rules encoding the initial state**: we add to $\pi(P, n)$ the following facts to describe the initial partial state

   $$h(s^0, 0). \quad (3.19)$$

   Note that the above is a shorthand for the set of facts $\{h(l, 0) \mid l \in s^0\}$.

2. **Rules encoding the goal**: for each $l \in G$, $\pi(P, n)$ contains the constraint:

   $$\leftarrow \neg h(l, n) \quad (3.20)$$

   This set of constraints makes sure that every fluent literal in $s^0$ holds in the final state.

3. **Rules to generate actions**: $\pi(P, n)$ contains the following rules for generating action occurrences:

   $$o(E, T) \lor \neg o(E, T) \leftarrow T < n. \quad (3.21)$$

   $$\leftarrow \neg o(A, T), T < n \quad (3.22)$$

   (Recall that $A$ is the set of all actions.) These rules are to generate at least
one action occurrence at any time $T < n^3$.

3.5.1 Properties of $\pi(P, n)$

The following theorem shows that we can use $\pi(P, n)$ to find solutions of $P$.

Theorem 2 (Soundness). Let $A$ be an answer set of $\pi(P, n)$ and let $a_i = \{e \mid o(e, i) \in A\} \ (0 \leq i < n)$. Then, $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$ is a solution of $P$.

Proof. See Appendix 6.3. $\square$

It follows from theorem 2 that answer sets of $\pi(P, n)$ correspond to solutions to $P$ of length $n$. If what is required is to find minimal plans, an strategy if to attempt to find answer sets to the program $\pi(P, n)$ for values of $n = 0, 1, 2, \ldots$ until an answer set is found. (i.e. first $n$ such that $\pi(P, n)$ is consistent). The chain of events represented in this answer set will be a minimal solution of $P$. This strategy was implemented in the ASP conformant planner CPASP$^4$.

Although theorem 6.3 tells us that $\pi(P, n)$ is sound, the program is not complete in the sense that there are conformant planning problems that have solutions of length $n$ for which $\pi(P, n)$ returns no answer set. A precise definition of completeness and incompleteness if given below.

Definition 14 (Completeness and Incompleteness of CPASP). Let $P$ be a planning problem. We say that CPASP is complete with respect to $P$ if either (i) $P$ does not have a solution; or (ii) $P$ has a solution and there exists an integer $n$ such that $\pi(P, n)$ is consistent. Otherwise, we say that CPASP is incomplete with respect to $P$.

The following examples illustrates this issue.

$^3$An alternative for this set of rules is a choice rule

$$1\{o(E, T) : \text{action}(E)\} \leftarrow T < \text{length}$$

which were introduced in [77]. If we wish to find a sequential plan, the only thing needed to do is to change the left side of the rule to $1\{o(E, T) : \text{action}(E)\}1$.

$^4$CPASP stands for Conformant Planning using Answer Set Programming.
Example 8. Consider the action theory \( D_2 \) with two dynamic causal laws

\[
e \text{ causes } f \text{ if } g
\]
\[
e \text{ causes } f \text{ if } \neg g
\]

Let \( P_2 = \langle D_2, \text{comp}(\emptyset), \{f\} \rangle \). Clearly \( \langle e \rangle \) is a solution of \( P_2 \) because either \( g \) or \( \neg g \) is true in any state belonging to \( \text{comp}(\emptyset) \) and thus one of the above dynamic causal laws would take effect when \( e \) is performed. Yet, it is easy to verify that this solution cannot be generated by CPasp due to the fact that for every \( n \), \( \pi(P_2, n) \) does not have any answer set (Constraint (3.20) is not satisfied).

The next example shows that not only conditional effects but also static causal laws can cause CPasp to be incomplete.

Example 9. Consider the action theory \( D_3 \):

\[
e \text{ causes } f
\]
\[
g \text{ if } f, h
\]
\[
g \text{ if } f, \neg h
\]

It is not difficult to check that \( \langle e \rangle \) is a solution to the planning problem \( P_3 = \langle D_3, \text{comp}([\neg f, \neg g]), \{g\} \rangle \) since \( e \) causes \( f \) to be true and the two static causal laws make \( g \) become true whenever \( f \) is true.

Now suppose that the program \( \pi(P_3, 1) \) has an answer set \( C \). Then, because only rule (3.6) has \( \text{dc}(\ldots, \ldots) \) as its head, we have \( \text{dc}(f, 1) \) is the only atom of the form \( \text{dc}(\ldots, \ldots) \) in \( C \). By rule (3.7), this implies that \( \text{ph}(f, 1) \in A \). By rule (3.12), \( \text{ph}(\neg g, 1) \), \( \text{ph}(h, 1) \), \( \text{ph}(\neg h, 1) \) all belong to \( C \). By rule (3.11), because \( \text{ph}(\neg g, 1) \in C \), \( h(g, 1) \) cannot belong to \( C \). As a result, constraint (3.20) is not satisfied. This is a contradiction because \( C \) must satisfies all the constraints of \( \pi(P_3, 1) \).

Hence, \( \pi(P_3, 1) \) does not have any answer set. In fact, we can verify that for any integer \( n \), \( \pi(P_3, n) \) does not have an answer set. This implies that CPasp cannot find a solution of \( P_3 \)
At this point, it is natural to question the applicability of CPasp. It is therefore worth mentioning that CPasp can solve all benchmark problems that we have encountered so far with non-disjunctive description of the initial state.

3.5.2 A Sufficient Condition for Completeness of CPasp.

In this section, we analyze the completeness of CPasp. In particular we define a class of planning problems called *simple planning problems* for which our planning strategy is complete. First we shall introduce some terminology.

**Definition 15 (Simple Action Theories).** A static causal law is *simple* if its precondition contains at most one fluent literal. An action theory $\mathcal{D}$ is *simple* if each of its static causal laws is simple.

In what follows, we further characterize the conditions under which $\pi(\mathcal{P}, n)$ is incomplete. In order to do so, we define a notion of dependencies between literals and actions [81].

**Definition 16 (Dependencies between literals).** A literal $l$ *depends on* a literal $g$, written as $l \triangleleft g$, whenever one of the following conditions holds.

1. $l = g$.
2. $\mathcal{D}$ contains a law $[a \text{ causes } l \text{ if } p]$ such that $g \in p$.
3. $\mathcal{D}$ contains a law $[l \text{ if } p]$ such that $g \in p$.
4. There exists a literal $h$ such that $l \triangleleft h$ and $h \triangleleft g$.
5. $l \triangleleft \overline{g}$.

Notice that the above dependency relationship between fluent literals is reflexive, transitive but not symmetric. The next definition describes the dependence between actions and fluent literals.
Definition 17 (Dependencies between actions and fluent Literals). An action \( b \) depends on a literal \( l \), written as \( b \triangleleft l \), if either

1. \( D \) contains an impossibility condition (3.3) such that \( a \subseteq b \) and \( \bar{l} \in p \), or

2. there exists a literal \( g \) such that \( b \triangleleft g \) and \( g \triangleleft l \).

For a set of fluent literals \( w \), we write \( l \triangleleft w \) to denote that \( l \triangleleft g \) for some \( g \in w \), and \( l \ntriangleleft w \) to denote that there exists no \( g \in w \) such that \( l \triangleleft g \).

We now define the notion of reducibility.

Definition 18. Let \( \Delta \) be a set of states, \( s \) be a partial state, and \( w \) be a set of fluent literals. We say that \( \Delta \) is reducible to \( s \) wrt \( w \), denoted by \( \Delta \gg w s \) if

1. \( s \) is a subset of every state \( \sigma \) in \( \Delta \),

2. for each fluent literal \( l \in w \), \( \Delta \) contains a state \( \sigma \) such that \( l \ntriangleleft (\sigma \setminus s) \), and

3. for each action \( a \), \( \Delta \) contains a state \( s \) such that \( a \ntriangleleft (\sigma \setminus s) \).

We now define a class of conformant planning problems, called simple planning problems, as follows:

Definition 19 (Simple Planning Problems). A planning problem \( \langle D, \text{comp}(s^0), G \rangle \) is simple if

1. \( D \) is simple, and

2. \( \text{comp}(s^0) \gg_G s^0 \).

As promised, the following theorem states that for the class of simple planning problems the planner \( \text{CPasp} \) is complete.

Theorem 3. Let \( \mathcal{P} = \langle D, \text{comp}(s^0), G \rangle \) be a planning problem. If \( \mathcal{P} \) is simple then \( \text{CPasp} \) is complete with respect to \( \mathcal{P} \).

Proof. See Appendix 6.5.5. \( \square \)

It is important to remark that the proofs of the completeness results rely on an establish theoretical foundation and techniques developed by the logic programming community.
3.6 An Extension for Problems with Disjunctive Initial State

The approach shown in the previous section depends on the possible initial states of a conformant planning problem to be modeled as the completions of a single partial state $s$. And indeed, it is shown in [82, 83] that many conformant planning problems in the literature can be successfully solved this way. In this section we extend the conformant planner described in the previous section to allow it to work with more complex definitions of the initial states. In particular, definitions of the initial states that involve disjunctive information.

With a more careful examination of approximations we can conclude the following observation, which can be viewed as a generalization of Observation 2.

**Observation 3.** Let $\mathcal{P} = \langle \mathcal{D}, \Gamma, G \rangle$ be a planning problem, $\mathcal{T}'(\mathcal{D})$ be an approximation, and $\alpha$ be a chain of events. Let $S$ be a set of partial states s.t. $\text{comp}(S) = \Gamma$. If for every $s \in S$, there exists a partial state $s'$ s.t. $\langle s, \alpha, s' \rangle \in \mathcal{T}'(\mathcal{D})$ and $G \subseteq s'$ then $\alpha$ is a solution of $\mathcal{P}$.

This observation extends the applicability of approximations to conformant planning problems with a disjunctive description of the initial state of the domain. In particular, instead of viewing the set of possible initial states $\Gamma$ as the completions of a single initial partial state $s$ as before, we view it as the completions of a set of initial partial states $S$. Then, to find a conformant plan within an approximation, we look for a chain of events $\alpha$ s.t. every possible path of $\alpha$ starting from a partial state in $S$ always leads to a goal partial state.

The framework presented in the previous section can be naturally extended to find solutions of a disjunctive conformant planning problem $\mathcal{P}$. First, we extend the program $\pi(\mathcal{D})$ to deal with explicit disjunctive information about the initial state. This extended program will be denoted $\tilde{\pi}(\mathcal{D})$. Following the framework, we then construct a planning program $\tilde{\pi}(\mathcal{P}, \mathcal{N})$ so that every answer set of $\tilde{\pi}(\mathcal{P}, \mathcal{N})$ represents a solution to $\mathcal{P}$.

The basic idea is to attach a distinct label to every initial partial state considered. In order to build $\tilde{\pi}(\mathcal{D})$, some of the predicates from $\pi(\mathcal{D})$ are modified to contain
a third parameter $K$ to indicate the initial partial state considered. The modified predicates are:

- $h(L, T, K)$ is true if literal $l$ holds at time step $t$ in a path starting from the initial partial state labeled with $k$;

- $de(L, T, K)$ is true if literal $l$ is a direct effect of an action that occurs at $(t - 1, k)$; and

- $ph(L, T, K)$ is true if literal $l$ possibly holds at time $t$ in a path with the initial partial state labeled with $k$.

The rules of $\pi(D)$ are obtained from the rules of $\pi(D)$ by replacing predicates $h(L, T)$, $de(L, T)$ and $ph(L, T)$ with the new predicates $h(L, T, K)$, $de(L, T, K)$ and $ph(L, T, K)$ respectively. For example, the rule (3.5) will become

$$h(l, T + 1, K) \leftarrow o(e, T), h(p, T, K).$$

Similarly, the rule (3.7) becomes

$$ph(l, T + 1, K) \leftarrow o(e, T), \neg h(\neg p, T, K), \neg de(\neg l, T + 1, K).$$

As before, we use the notation $\pi(D, n)$ to denote the restriction of $\pi(D)$ to the time step to take value between 0 and $n$.

Let $P = (D, \Gamma, G)$ be a conformant planning problem. Let $S = \{s_0, s_1, .., s_m\}$ be a set of partial states s.t. $\Gamma = \text{comp}(S)$ (hereafter we refer to such $S$ as a set of initial partial states of $P$). Let the subindex of each element of $S$ act as its label. We construct the program $\pi(P, n)$ based on $S$ as follows. $\pi(P, n)$ contains the rules of $\pi(D, n)$ and the following rules.

1. **Rules encoding the initial state:** For every $s_i \in S$, $\pi(P, n)$ contains the rule

$$h(s_i, 0, i). \quad (3.23)$$
2. Rules encoding the goal: To guarantee that the goal is satisfied in the final state of every path, we add to $\tilde{\pi}(\mathcal{P}, n)$ the following rules:

$$\{ \leftarrow \text{not } h(l, n, K) : l \in G \}.$$ 

3. Rules to generate actions: To generate action occurrences, we add to $\tilde{\pi}(\mathcal{P}, n)$ the following choice rule [77]:

$$1 \{ o(E, T) : \text{action}(E) \} \leftarrow T < n \quad (3.24)$$

The following proposition establishes the relationship between $\pi(\mathcal{P}, n)$ and $\tilde{\pi}(\mathcal{P}, n)$.

Proposition 2. Let $\mathcal{P} = \langle \mathcal{D}, \text{comp}([s^0]), G \rangle$ and $\mathcal{P}' = \langle \mathcal{D}, \text{comp}(s^0), G \rangle$. A set of atoms $A$ is an answer set of $\tilde{\pi}(\mathcal{P}, n)$ iff there exists an answer set $B$ of $\pi(\mathcal{P}', n)$ such that

1. for every fluent literal $l$, $h(l, i, 0) \in A$ iff $h(l, i) \in B$, and

2. for every elementary action $e$, $o(e, i, 0) \in A$ iff $o(e, i) \in B$.

Theorem 4. Let $A$ be an answer set of $\tilde{\pi}(\mathcal{P}, n)$. For $0 \leq i < n$ let $a_i = \{ e : o(e, i) \in A \}$. Then, $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$ is a solution of $\mathcal{P}$. \hfill $\Box$

The proof of theorem 4 is similar to the proof of Theorem 2.

Bases on the previous work on the completeness of $\pi(\mathcal{P}, n)$, we have the following result about the completeness of $\tilde{\pi}(\mathcal{P}, n)$.

Theorem 5. Let $\alpha = \langle a_1, \ldots, a_{n-1} \rangle$ be a solution of $\mathcal{P} = \langle \mathcal{D}, \Gamma, G \rangle$. If $\mathcal{D}$ is simple and $\tilde{\pi}(\mathcal{P}, n)$ is constructed based on a set of initial partial states $S$ s.t. $\Gamma \gg G S$ then $\tilde{\pi}(\mathcal{P}, n)$ is consistent and has an answer set $A$ s.t. $a_i = \{ e : o(e, i) \in A \}$. \hfill $\Box$

As done before, we build a conformant planner from program $\tilde{\pi}(\mathcal{P}, n)$ by the following algorithm: sequentially run program $\tilde{\pi}(\mathcal{P}, n)$ with $n = 1, 2, \ldots$, until it

\footnote{If we wish to find a sequential plan, the only thing needed to do is to change the left side of the rule to $1 \{ o(E, T) : \text{action}(E) \}$.}
returns an answer set. We have implemented this algorithm in a planning system called CPASPm.

We can verify that for the planning problem \( P_2 \) in Example 8, if we construct \( \pi(P, n) \) based on \( S_1 = \{ \emptyset \} \) then we would miss the solution \( \langle a \rangle \). However, if we construct it based on the set of partial states \( S_2 = \{ \{ f \}, \{ \neg f \} \} \) then \( \pi(P, n) \) would return an answer set corresponding to the solution \( \langle a \rangle \).

3.7 Experiments

In this section, we will present our experimental evaluation of the performance of CPASP and CPASPm. The platform used for testing is a 2.4 GHz CPU, 768MB RAM machine, running Slackware 10.0 operating system. Every experiment had a time limit of 30 minutes.

3.7.1 Evaluation of CPASP

For the experiments in this section, the platform used for testing is a 2.4 GHz CPU, 768MB RAM machine, running Slackware 10.0 operating system. Every experiment had a time limit of 30 minutes.

3.7.1.1 Planning Systems

We compared CPASP with three other conformant planners: CMBP [22], DLV\(^k\) [29], and C-PLAN [21]; We selected these planners because they are in spirit similar to 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). A brief overview of these systems is given below.

- CMBP (Conformant Model Based Planner): CMBP is a conformant planner developed by Cimatti and Roveri [22]. CMBP employs BDD (Binary Decision Diagram) techniques to represent planning domains and search for solutions. CMBP allows non-deterministic domains with uncertainty in both the initial state and action effects. However, it does not have the capability of generating

- **DLV$^K$**: DLV$^K$ is a declarative, logic-programming-based planning system built on top of the DLV system http://www.dbai.tuwien.ac.at/proj/dlv/). Its input language $K$ is a logic-based planning language described in [29]. The version used for testing was downloaded from http://www.dbai.tuwien.ac.at/proj/dlv/K/. DLV$^K$ is capable of generating both concurrent and conformant plans.

- **C-Plan**: C-Plan is a SAT-based conformant planner. C-Plan works using a generate-and-test method: candidate plans are generated and then verified to be solutions. The input language is the action language CCalc [46]. C-Plan is designed for generating concurrent plans.

3.7.1.2 Benchmarks

We prepared two test suites. The first test suite contains sequential, conformant planning benchmarks and the second contains concurrent, conformant planning benchmarks. Many benchmarks are taken from the literature (e.g. [15, 21, 29]) and some were developed for the international planning competition (e.g. [28]). To test the performance of our systems in domains with static causal laws, we developed two simple domains which are rich in static causal laws.

Sequential Benchmarks. The sequential benchmark test suite includes the following domains:

- **BMT(m,n)**: This domain is a variant of the well-known Bomb-In-the-Toilet domain [64]. It can be briefly described as follows: We have received an alarm that there might be a bomb in a lavatory. There are $m$ suspicious packages and $n$ toilets. A package can be disarmed by dunking it into a toilet. The goal is to have the bomb defused.
• **BMTC**(m,n): This domain is similar to BMT. However, we assume that dunking a package into a toilet will clog the toilet; flushing the toilet will make it unclogged.

• **RING**(n) [23]: In this domain, the agent can move around a building with \( n \)-room arranged in a ring in a cyclic fashion (either clockwise or counterclockwise) Each room has a window which can be closed and/or locked. A window can only be locked if it is closed. Initially, neither the location of the agent nor the states (open/locked) 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. Note that this domain contains disjunctive information in the initial state.

• **Domino**(n): We have \( n \) dominoes 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 dominoes are unknown. The goal is to have the rightmost domino fall.

• **Gaspipe**(n): The objective of this domain is 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 end of the pipeline and the burner is on the right-most end. The pipe line is made of sections connected with each other by valves. The pipe sections can be either pressured by the tank or un-pressured. Opening a valve causes the section on its right side to be pressured if the section to its left is pressured. Moreover, for safety reasons, 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. These domain presents 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 be 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 in the initial situation is that the states of the valves are unknown. A possible conformant plan will be to close all valves except the first one (that is, the one that directly connects to the gas tank) from right to left and then opening them from left to right.

- **Cleaner(n,p):** This domain is a modified version of the Ring domain in which instead of locking windows, the goal of the agent is to clean multiple objects located in every room (there are p objects in each room). Initially, the agent is in the first room and does not know whether or not any of the objects are cleaned.

**Concurrent Benchmarks.** There are four domains in this test suite, namely BT$^p$, BTC$^p$, Gaspipes$^p$ and Cleaner$^p$. The BT$^p$ and 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 Gaspipes$^p$ domain is a modification of the Gaspipes 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, Cleaner$^p$, is a modified version of the Cleaner domain where we allow the robot to concurrently clean multiple objects in the same room.

3.7.1.3 **Experimental Results**

We ran CPASP on using the answer set solvers SModels [77] and CModels [51] and observed that CModels yielded better performance in general. The running times of CPASP reported here were obtained using CModels. The timing results for the sequential benchmarks is shown in Tables 1 & 2, and the results for the concurrent benchmarks is shown in Table 3. We did not test C-Plan on the sequential planning benchmarks since it is supposed to be used only for concurrent
In these tables, times are shown in seconds; The “PL” column shows the length of the plan found by the planner. “-” denotes that the planner did not return a solution within the time limit for some reasons: e.g., out of time and out of memory. “NA” denotes that the problem was not run with the planner. Since both \( DLV^K \) and \( CPasp \) require as an input parameter the length of a plan to search for, we ran them inside of a loop in which we incrementally increase the plan length to search for, starting from 1\(^7\), until a plan is found. Notice that in this way \( CPasp \) is not only finding conformant plans but minimum conformant plans with respect to the defined approximation. For example, \( C-Plan \) and \( CPasp \) took 0.72 seconds and 2.72 seconds, respectively, to find a solution for the BTC(6,2) problem. Nevertheless, \( CPasp \)'s solution is shorter.

<table>
<thead>
<tr>
<th>Problem</th>
<th>CMBP PL</th>
<th>CMBP Time</th>
<th>DLV^K PL</th>
<th>DLV^K Time</th>
<th>CPasp PL</th>
<th>CPasp Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT(2,2)</td>
<td>2</td>
<td>0.03</td>
<td>2</td>
<td>0.04</td>
<td>2</td>
<td>0.20</td>
</tr>
<tr>
<td>BT(4,2)</td>
<td>4</td>
<td>0.17</td>
<td>4</td>
<td>0.55</td>
<td>4</td>
<td>0.41</td>
</tr>
<tr>
<td>BT(6,2)</td>
<td>6</td>
<td>0.21</td>
<td>6</td>
<td>216.55</td>
<td>6</td>
<td>0.77</td>
</tr>
<tr>
<td>BT(8,4)</td>
<td>8</td>
<td>0.63</td>
<td>-</td>
<td>8</td>
<td>6.73</td>
<td></td>
</tr>
<tr>
<td>BT(10,4)</td>
<td>10</td>
<td>1.5</td>
<td>-</td>
<td>10</td>
<td>890.06</td>
<td></td>
</tr>
<tr>
<td>BTC(2,2)</td>
<td>2</td>
<td>0.16</td>
<td>2</td>
<td>0.12</td>
<td>2</td>
<td>0.22</td>
</tr>
<tr>
<td>BTC(4,2)</td>
<td>6</td>
<td>0.26</td>
<td>6</td>
<td>72.44</td>
<td>6</td>
<td>0.71</td>
</tr>
<tr>
<td>BTC(6,2)</td>
<td>10</td>
<td>0.74</td>
<td>-</td>
<td>8</td>
<td>2.72</td>
<td></td>
</tr>
<tr>
<td>BTC(8,4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>BTC(10,4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Ring(2)</td>
<td>5</td>
<td>0.06</td>
<td>5</td>
<td>0.10</td>
<td>5</td>
<td>0.52</td>
</tr>
<tr>
<td>Ring(4)</td>
<td>11</td>
<td>0.10</td>
<td>11</td>
<td>2.14</td>
<td>11</td>
<td>2.98</td>
</tr>
<tr>
<td>Ring(6)</td>
<td>17</td>
<td>0.48</td>
<td>-</td>
<td>17</td>
<td>44.43</td>
<td></td>
</tr>
<tr>
<td>Ring(8)</td>
<td>-</td>
<td>-</td>
<td>23</td>
<td>1424.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ring(10)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Sequential Benchmarks: Bomb & Ring domains

\(^6\)The authors told us that \( C-Plan \) was not intended for searching sequential plans

\(^7\)We did not start from 0 because none of the benchmarks has a plan of length 0
As it can be seen in Table 1, in the BT and BTC domains CMBP outperforms both $DLV^K$ and CPASP on most problem instances. In general CPASP has better performance than $DLV^K$ on these domains. As an example, $DLV^K$ took more than three minutes to solve BT(6,2), while it took only 0.77 seconds for CPASP to solve the same problem. In addition, within the time limit, CPASP was able to solve more problems than $DLV^K$. In the Ring domain, although outperformed by both CMBP and $DLV^K$ in some small instances, CPASP is the only planner that was able to solve Ring(8).
<table>
<thead>
<tr>
<th>Problem</th>
<th>CMBP</th>
<th>DLV&lt;sup&gt;k&lt;/sup&gt;</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PL</td>
<td>Time</td>
<td>PL</td>
</tr>
<tr>
<td>Domino(100)</td>
<td>1</td>
<td>0.26</td>
<td>1</td>
</tr>
<tr>
<td>Domino(200)</td>
<td>1</td>
<td>1.79</td>
<td>1</td>
</tr>
<tr>
<td>Domino(500)</td>
<td>1</td>
<td>7.92</td>
<td>1</td>
</tr>
<tr>
<td>Domino(1000)</td>
<td>1</td>
<td>13.2</td>
<td>1</td>
</tr>
<tr>
<td>Domino(2000)</td>
<td>1</td>
<td>66.6</td>
<td>1</td>
</tr>
<tr>
<td>Domino(5000)</td>
<td>1</td>
<td>559.46</td>
<td>1</td>
</tr>
<tr>
<td>Domino(10000)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Gaspip(3)</td>
<td>NA</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Gaspip(5)</td>
<td>NA</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Gaspip(7)</td>
<td>NA</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Gaspip(9)</td>
<td>NA</td>
<td>-</td>
<td>17</td>
</tr>
<tr>
<td>Gaspip(11)</td>
<td>NA</td>
<td>-</td>
<td>21</td>
</tr>
<tr>
<td>Cleaner(2,2)</td>
<td>5</td>
<td>0.1</td>
<td>5</td>
</tr>
<tr>
<td>Cleaner(2,5)</td>
<td>11</td>
<td>0.61</td>
<td>11</td>
</tr>
<tr>
<td>Cleaner(2,10)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cleaner(4,2)</td>
<td>11</td>
<td>0.13</td>
<td>11</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>17</td>
<td>4.1</td>
<td>17</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>

Table 2. Sequential Benchmarks: Domino, Gaspip & Cleaner domains

CPASP works well with domains rich in static causal laws like Domino and Gaspip. In the Domino domain, CPASP outperforms all the other planners in most of instances. It took only 2.41 seconds to solve Domino(2000), while both DLV<sup>k</sup> and CMBP took more than one minute. In fact CPASP could scale up very well to larger instances, e.g., Domino(10000). In the Gaspip domain, CPASP also outperforms DLV<sup>k</sup>: it was able to solve all the problem instances while DLV<sup>k</sup> was able
to solve only the first three problem instances\textsuperscript{8}.

The Cleaner domain turns out to be hard for the planners: they could solve very small instances only. In this domain, CPASp is outperformed by CMBP. To solve the Cleaner(6,2), CMBP took only 4.1 seconds while CPASp took more than 3 minutes. However, CPASp performs better than DLV\textsuperscript{K} in general: DLV\textsuperscript{K} reported a timeout with the problem Cleaner(6,2).

We have seen that CPASp can be competitive with CMBP and DLV\textsuperscript{K} on the sequential benchmarks. Let us move our attention now to the concurrent benchmarks. As can be seen from Table 3, CPASp outperforms both DLV\textsuperscript{K} and C-PLAN on most instances of the BT\textsuperscript{P}, BTC\textsuperscript{P}, and Gaspipe\textsuperscript{P} domains. Furthermore, CPASp is the only planner that was able to solve all the instances in the test suite. In the Cleaner

\textsuperscript{8}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.
domain, C-PLAN is the best. To solve the Cleaner\(^p\)(6,10) problem, C-PLAN took only 0.35 seconds, whereas DLV\(^k\) reported a timeout and CPASP needed 3.73 seconds.

<table>
<thead>
<tr>
<th>Problem</th>
<th>C-PLAN</th>
<th>DLV(^k)</th>
<th>CPASP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PL</td>
<td>Time</td>
<td>PL</td>
</tr>
<tr>
<td>BT(^p)(2,2)</td>
<td>1</td>
<td>0.07</td>
<td>1</td>
</tr>
<tr>
<td>BT(^p)(4,2)</td>
<td>2</td>
<td>0.05</td>
<td>2</td>
</tr>
<tr>
<td>BT(^p)(6,2)</td>
<td>3</td>
<td>1.81</td>
<td>3</td>
</tr>
<tr>
<td>BT(^p)(8,4)</td>
<td>2</td>
<td>4.32</td>
<td>2</td>
</tr>
<tr>
<td>BT(^p)(10,4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BTC(^p)(2,2)</td>
<td>1</td>
<td>0.05</td>
<td>1</td>
</tr>
<tr>
<td>BTC(^p)(4,2)</td>
<td>3</td>
<td>0.07</td>
<td>3</td>
</tr>
<tr>
<td>BTC(^p)(6,2)</td>
<td>5</td>
<td>7.51</td>
<td>5</td>
</tr>
<tr>
<td>BTC(^p)(8,4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BTC(^p)(10,4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Gaspipe(^p)(3)</td>
<td>-</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Gaspipe(^p)(5)</td>
<td>-</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Gaspipe(^p)(7)</td>
<td>-</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Gaspipe(^p)(9)</td>
<td>-</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Gaspipe(^p)(11)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cleaner(^p)(2,2)</td>
<td>3</td>
<td>0.05</td>
<td>3</td>
</tr>
<tr>
<td>Cleaner(^p)(2,5)</td>
<td>3</td>
<td>0.12</td>
<td>3</td>
</tr>
<tr>
<td>Cleaner(^p)(2,10)</td>
<td>3</td>
<td>0.06</td>
<td>3</td>
</tr>
<tr>
<td>Cleaner(^p)(4,2)</td>
<td>7</td>
<td>0.06</td>
<td>7</td>
</tr>
<tr>
<td>Cleaner(^p)(4,5)</td>
<td>7</td>
<td>0.09</td>
<td>7</td>
</tr>
<tr>
<td>Cleaner(^p)(4,10)</td>
<td>7</td>
<td>0.13</td>
<td>7</td>
</tr>
<tr>
<td>Cleaner(^p)(6,2)</td>
<td>11</td>
<td>0.11</td>
<td>11</td>
</tr>
<tr>
<td>Cleaner(^p)(6,5)</td>
<td>11</td>
<td>0.19</td>
<td>11</td>
</tr>
<tr>
<td>Cleaner(^p)(6,10)</td>
<td>11</td>
<td>0.35</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3. Concurrent Benchmarks: Bomb, Gaspipe & Cleaner domains
3.7.2 Evaluation of CPAspm

We prepared a test suite of problems which involves both concurrency and complex initial situations. Some of this problems are simple modifications to conformation benchmarks in the literature. The test suite consists of the problems Conformant Turkey and Lost Cleaner.

- **Conformant Turkey** is a modification of the classic Yale shooting problem. In our problem, there are $n$ guns, but only one of which is loaded. Guns can be shot concurrently, with the exception of gun #1 and #2. The objective is to kill the turkey.

- **Lost Cleaner** is a modification of the ring [23]. The agent is in one of $n$ rooms arranged in a ring fashion. Each room has $m$ objects that must be cleaned by the agent. The agent can move to the next room in a clockwise or counter-clockwise fashion and at a time it can clean objects in a room concurrently. Initially, the agent does not know its initial location nor the current status (cleaned or dirty) of the objects.

Experiments were conducted in a 1.4GHz Pentium 4 machine running Linux, and each planner was given 30 minutes to complete each instance of each problem. The results are shown in Tables 1 and 2. In each table, times are in seconds, and PL is the number of steps of a shortest solution of the corresponding problem.

As we can see from Table 3.1, CPAspm outperforms both DLV$^k$ and C-PLAN in the Conformant Turkey domain. Solutions for these domain all involve 3 steps, with one step consisting of firing multiple guns concurrently. In this domain, exploiting concurrency is key to obtaining a short plan.

In the Lost Cleaner domain, shown in Table 5.3, only CPAspm is able to solve all problem instances within the time limit. In this domain, the length of a solution ranges from 3 to 19 steps and it is determined by the number of rooms in the ring. As the number of rooms in the problem increases, the number of possible initial situations increases. One of the advantages of using approximations is that it is
Table 3.1. Conformant Turkey domain

possible to reason about the domain without having to consider all possible worlds. As the complexity of the problem increases, only CPASP\textsubscript{m} is able to scale. In this problem C-PLAN behaved erratically, failing some simple problem instances and then solving more complicated ones. In general, the use of approximations allows CPASP\textsubscript{m} to exploit concurrent actions when searching for a solution, and to scale better than the other concurrent planners as the number of possible initial states and the length of the solution increases.
Figure 3.4. Conformant Turkey Domain
<table>
<thead>
<tr>
<th>No. Rooms</th>
<th>No. Obj.</th>
<th>PL</th>
<th>DLV^x</th>
<th>C-PAN</th>
<th>CPASP_{m}</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0.068</td>
<td>0.063</td>
<td>0.201</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>0.092</td>
<td>-</td>
<td>0.233</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>3</td>
<td>0.143</td>
<td>-</td>
<td>0.288</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>7</td>
<td>0.483</td>
<td>425.048</td>
<td>1.219</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>2.648</td>
<td>-</td>
<td>1.897</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>7</td>
<td>441.485</td>
<td>-</td>
<td>3.015</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>11</td>
<td>35.901</td>
<td>-</td>
<td>7.190</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>12.414</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>22.503</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>51.147</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>76.707</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>142.563</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>19</td>
<td>-</td>
<td>-</td>
<td>225.405</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>19</td>
<td>-</td>
<td>-</td>
<td>403.443</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>19</td>
<td>-</td>
<td>-</td>
<td>642.319</td>
</tr>
</tbody>
</table>

Table 3.2. Lost Cleaner Domain
Figure 3.5. Lost Cleaner Domain
CHAPTER 4
IMPROVING SYSTEM ACSOLVER - SYSTEM LUNA

In this chapter we first review the knowledge representation language $\mathcal{AC}(\mathcal{C})$, which extends the syntax and semantics of ASP. $\mathcal{AC}(\mathcal{C})$ was first presented in [70], as part of research on integrating answer set programming and constraint logic programming (CLP). A solver for programs of $\mathcal{AC}(\mathcal{C})$ called ACSolver was developed as part of the same research effort.

Afterwards we present an algorithmic framework for computing answer sets of $\mathcal{AC}(\mathcal{C})$ programs. Given an input program, the framework describes a search graph whose nodes are states of computation an algorithm will follow while building answer sets for the input program. Transitions between states of computation are specified declaratively, allowing multiple possible algorithms and different search strategies to be built over the framework.

Later, we describe an algorithm based on this framework, which forms the basis of the implementation of system LUNA. The algorithm used in LUNA features efficiency improvements over the algorithm of the previous prototype solver ACSolver.

Finally, we present experimental results on the efficiency of system LUNA.

4.1 Background

Grounding is the process of instantiating the variables of an ASP program by ground terms. The first step that traditional ASP solvers take in the search for answer sets is to ground a program. Current ASP solvers use sophisticated grounding mechanisms and intelligent grounding optimization techniques. However, the ground instantiation of a program can be huge, specially if the program contains variables that range over large numerical domains. A program with a huge ground instantiation causes both memory and time problems that overwhelm even the most sophisticated ASP solvers. This problem is recognized as an important problem by the ASP community. A partial solution is suggested in [11, 68], where an extension
to ASP and a corresponding reasoning mechanism is proposed to partially avoid the grounding of variables ranging over large domains. In this work, we start from the solution proposed in [70] and [67], in which a new language, $\mathcal{AC}(\mathcal{C})$, and a corresponding computing method is presented.

The language $\mathcal{AC}(\mathcal{C})$ is an extension of ASP that combines ASP with Constraint Logic Programming (CLP). Solvers for $\mathcal{AC}(\mathcal{C})$ programs would use CLP techniques to avoid grounding variables ranging over large domains. The $\mathcal{AC}(\mathcal{C})$ based approach often allows to solve problems which are impossible with traditional ASP solving methods.

4.1.1 The Language $\mathcal{AC}(\mathcal{C})$

4.1.1.1 Syntax

$\mathcal{AC}(\mathcal{C})$ is a typed language. Its programs are defined over a sorted signature $\Sigma$, consisting of sorts, and properly typed predicate symbols, function symbols, and variables. By a sort we mean a non-empty countable collection of strings over some fixed alphabet. Strings of a sort $S$ will be referred as object constants of $S$. Each variable takes on values of a unique sort. A term of $\Sigma$ is either a constant, a variable, or an expression $f(t_1, \ldots, t_n)$ where $f$ is a function symbol of arity $n$, and $t_1, \ldots, t_n$ are terms of the proper sorts. The sort of a function $f : S_1 \times \cdots \times S_n \rightarrow S$ is $S$. An atom is of the form $p(t_1, \ldots, t_n)$ where $p$ is an $n$-ary predicate symbol, and $t_1, \ldots, t_n$ are terms of the proper sorts. A literal is either an atom $a$ or its negation $\neg a$. Sorts of $\mathcal{AC}(\mathcal{C})$ will be partitioned into regular and constraint. Intuitively, a constraint sort is a large (often numerical) set with primitive constraint relations (e.g. $\leq$).

The partitioning of sorts induces a natural partition of predicates, literals, and programs of $\mathcal{AC}(\mathcal{C})$.

- **regular** predicates denote relations among objects of regular sorts;
- **constraint** predicates denote primitive numerical relations on constraint sorts;
• \textit{defined} predicates are defined in terms of constraint, regular or defined predicates.

• \textit{mixed} predicates denote relations between objects which belong to regular sorts and those who belong to constraint sorts.

Literals formed by regular predicates are called \textit{regular}. Similarly for constraint, \textit{defined} and \textit{mixed} literals.

A rule of \textit{AC} is an expression of the form

\[ l_0 \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n \]

where \( l \)'s are literals of \( \Sigma \).

A program \( \Pi \) of \textit{AC}(\mathcal{C}) consists of definitions of sorts of a signature \( \Sigma \), declarations of variables, and a collection of rules over a signature \( \Sigma \).

The rules of \( \Pi \) will be partitioned into a \textit{regular part}, \( \Pi_R \), consisting of rules built from regular literals, a \textit{defined part}, \( \Pi_D \), consisting of rules whose heads are defined literals, and a \textit{middle part}, \( \Pi_M \), consisting of all other rules of \( \Pi \). Elements of \( \Pi_R, \Pi_D \) and \( \Pi_M \) will be called \textit{regular rules}, \textit{defined rules} and \textit{middle rules} respectively. Intuitively, this partitioning of the program will be used by the inference algorithms. In particular the regular part will be processed by an ASP reasoning mechanism, and the defined part will be processed by a CLP mechanism, with the middle rules serving as a bridge between the two. A \textit{AC} programmer will therefore have some control over the inference engines associated with the language. Furthermore, a programmer will be able to ensure that not all variables of the program will be grounded and that the engine will properly combine ASP and CLP algorithms. This control however will be achieved by purely declarative means.
4.1.1.2 Semantics

Let $R$ be a rule of an $AC(c)$ program $\Pi$ with signature $\Sigma$. A ground instance of $R$ is obtained from $R$ by

1. replacing variables of $R$ by ground terms from the respective sorts; and
2. replacing all numerical terms by their values.

An ASP program $\text{ground}(\Pi)$ consisting of all ground instances of all rules in $\Pi$ is called the ground instantiation of $\Pi$.

Definition 20 (Partial interpretation). A consistent set $S$ of ground literals over the signature $\Sigma$ is called a partial interpretation of an $AC(c)$ program $\Pi$ if it satisfies the following conditions:

1. A constraint literal $l \in S$ iff $l$ is true under the intended interpretation of its symbols;
2. For every mixed predicate $m(\overline{X_r}, \overline{Y_c})$ and every ground instantiation $\overline{t_r}$ of $\overline{X_r}$, there is a unique ground instantiation $\overline{t_c}$ of $\overline{Y_c}$ such that $m(\overline{t_r}, \overline{t_c}) \in S$.

Definition 21 (Answer sets of $AC(c)$ programs). A partial interpretation $S$ of the signature $\Sigma$ of an $AC(c)$ program $\Pi$ is called an answer set of $\Pi$ if there is a set $M$ of ground mixed literals of $\Sigma$ such that $S$ is an answer set of the ASP program $\text{ground}(\Pi) \cup M$.

4.1.2 The Solver $ACsolver$

$ACsolver$ is the name of an algorithm and the first prototype system for finding answer sets of $AC(c)$ programs. It was first presented in [67] and [70]. The algorithm combines ASP solving methods with constraint satisfaction techniques and SLDNF resolution. As a prototype system, it suffers from reliability and efficiency problems.

With regards to efficiency, the main drawback of $ACsolver$ lies in the way the ASP solving methods and the CLP techniques are combined. Internally, $ACsolver$
is divided into two modules: an ASP module that builds the regular part of an answer set, and a CLP module that answers queries from the ASP module about the defined part of the program. The problem lies in that answers to these queries are not computed incrementally. This is, the answer to a query is not used to compute the answer to the next query, and computational work is repeated.

The algorithmic framework presented in this dissertation addresses this issue by allowing different algorithms build over this framework different degrees of coupling between the computation of the regular part of the answer set and the computation of solutions to queries, and therefore different degrees of incrementally.

4.2 An Algorithmic Framework

4.2.1 Preliminaries

In order to precisely define our algorithmic framework, we need to introduce the following definitions.

Definition 22 (Query). A query is a set of defined and constraint e-literals (primitive constraints).

A ground set $S$ of literals satisfies a query $Q$ if there is a substitution $\gamma$ of variables of $\Sigma$ by ground terms such that the result, $\gamma(Q)$, of this substitution is a subset of $S$. We will refer to $\gamma$ as a solution of $Q$ with respect to $S$. A set of primitive constraints $C$ will be called a constraint solution of a query $Q$ if $C$ is consistent and every solution $\gamma$ of $C$ is a solution of $Q$.

Definition 23 (Forced Query). We say that a query $Q$ is forced by a set of ground regular e-literals $B$ with respect to a logic program $\Pi$ whenever $\Pi$ has an answer set containing $B$ iff $\Pi$ has an answer set containing $B$ and satisfying $Q$.

Definition 24 (Simplification). A logic program $\Pi'$ is called a simplification of a logic program $\Pi$ with respect to a set of ground regular e-literals $B$ if:

1. $\Pi'$ contains no literals from $B$. 

49
2. For every set \( M \) of regular e-literals containing \( B \), \( M \) is an answer set of \( \Pi \) iff 
\( M \setminus B \) is an answer set of \( \Pi' \).

**Definition 25 (Consequences).** \( B' \) is a set of consequences of a set of ground regular e-literals \( B \) with respect to a logic program \( \Pi \) if:

1. \( B \subseteq B' \).
2. \( M \) is an answer set of \( \Pi \) containing \( B \) iff \( M \) is an answer set of \( \Pi \) containing \( B' \).

### 4.2.1.1 The Upper Closure

**Definition 26.** Given a set of regular e-literals \( B \) and program \( \Pi \), by \( \alpha(\Pi, B) \), we will denote the program obtained from \( \Pi \) by:

1. Removing all rules in \( \Pi \) whose bodies are falsified by \( B \).
2. Removing all rules in \( \Pi \) whose head is false in \( B \).
3. Removing all constraint, defined and bridge literals from the bodies of rules in \( \Pi \).

**Definition 27 (Upper Closure).** The upper closure of a program \( \Pi \) with respect to a set of regular e-literals \( B \), denoted \( \text{up}(\Pi, B) \), is the deductive closure of \( \alpha(\Pi, B) \).

**Claim.** Let \( \Pi \) be a program and \( B \) be a set of regular e-literals. Then:

- \( \text{up}(\Pi, B) \) is unique,
- If \( M \) is an answer set of \( \Pi \) containing \( B \) then the regular part of \( M \) is a subset of \( \text{up}(\Pi, B) \).
4.2.2 \( \mathcal{AC}(C) \) problems and solutions

Now we precisely define the problem to be solved by an \( \mathcal{AC}(C) \) solver.

**Definition 28 (Problem of \( \mathcal{AC}(C) \)).** A problem of \( \mathcal{AC}(C) \) is a triple \( (\Pi, A, K) \) where \( \Pi \) is a logic program, \( A \) is a set of ground regular e-literals and \( K \) is a query.

**Definition 29 (Solution to a problem of \( \mathcal{AC}(C) \)).** A pair \( (B, C) \), where \( B \) is a set of ground regular e-literals and \( C \) is a consistent set of primitive constraints, is called a solution to a problem \( (\Pi, A, K) \) if there exists an answer set \( W \) of \( \Pi \) where \( B \) is the regular part of \( W \), \( B \) contains \( A \), and any solution \( \gamma \) of \( C \) is a solution of \( K \) with respect to \( W \).

4.2.3 Model of computation

The following definitions describe the set of possible states of computation that an \( \mathcal{AC}(C) \) solver build over this framework may follow during the computation of a solution to a problem of \( \mathcal{AC}(C) \). The set of all possible states of computation is often called the *search space* of the problem.

**Definition 30 (State of computation).** A state of computation is a 4-tuple \( \langle \pi_0, ..., \pi_j | B_0, ..., B_k | Q_0, ..., Q_n | C \rangle \) where \( \pi_0, ..., \pi_j \) is a sequence of logic programs, \( B_0, ..., B_k \) is a sequence of sets of ground regular e-literals, \( Q_0, ..., Q_n \) is a sequence of individual queries, and \( C \) is a set of primitive constraints. \( Q_0, ..., Q_n \) will be referred as the *query component* and \( C \) will be referred to as the *constraint store*. The empty sequence will be identified by nil.

**Definition 31 (Transition).** Let \( S = \langle \pi_0, ..., \pi_j | B_0, ..., B_k | Q_0, ..., Q_n | C \rangle \) be a state of computation. Transitions from \( S \) to another state \( S' \) are defined as follows:

- *(guess-transition)*

  \( \langle S, l, S' \rangle \) is a transition, where \( l \) is a regular e-literal in the signature of \( \pi_k \) and

  \[ S' = \langle \pi_0, ..., \pi_j, \pi_{j+1} | B_0, ..., B_{k+1}, B_{k+1} | Q_0, ..., Q_n, | C \rangle \] if \( B_{k+1} = \{l\} \) and \( \pi_{j+1} \) is a simplification of \( \pi_j \) w.r.t. \( B_{k+1} \).

51
\[ \langle S, B_{k+1}, S' \rangle \] is a transition, where \( B_{k+1} \) is a set of regular e-literals and \( S' = \langle \pi_0, ..., \pi_j, \pi_{j+1} | B_0, ..., B_k, B_{k+1} | Q_0, ..., Q_n | C \rangle \) if \( A \) is a set of consequences of \( B_k \) w.r.t. \( \pi_k, B_{k+1} = A \cup \{ \text{not} l | l \notin \text{up}(A, \pi_k) \} \), and \( \pi_{j+1} \) is a simplification of \( \pi_j \) w.r.t. \( B_{k+1} \).

\[ \langle S, K, S' \rangle \] is a transition, where \( K \) is a query, and \( S' = \langle \pi_0, ..., \pi_j, \pi_{j+1} | B_0, ..., B_k | Q_0, ..., Q_n, Q_{n+1}, ..., Q_m | C \rangle \) if \( K = Q_{n+1}, ..., Q_m \) is the query forced by \( B_k \) w.r.t \( \pi_j \), and \( \pi_{j+1} \) is a simplification of \( \pi_j \) w.r.t. \( K \).

\[ \langle S, c, S' \rangle \] is a transition, where \( c \) is a primitive constraint such that \( Q_0 = c \) and \( S' = \langle \pi_0, ..., \pi_j | B_0, ..., B_k | Q_1, ..., Q_n | C' \rangle \) if \( C' \) is a (possible simplified) set of constraints equivalent to \( C \cup \{ c \} \) and \( C' \) is solvable (has a solution).

\[ \langle S, r, S' \rangle \] is a transition, where \( r \) is a rule in \( \pi_k \) such that the head of \( r \) can be unified with \( Q_0 \) and \( S' = \langle \pi_0, ..., \pi_j | B_0, ..., B_k | \chi_1 = t_1, ..., \chi_s = t_s, b_1, ..., b_m, Q_1, ..., Q_n | C \rangle \) if \( \chi_1 = t_1, ..., \chi_s = t_s \) is the most general unifier of \( Q_0 \) with the head of \( r \), and \( b_1, ..., b_m \) is the body of \( r \).

\[ \langle S, l, S' \rangle \] is a transition, where \( l \) is a regular e-literal such that \( Q_0 = l \) and \( S' = \langle \pi_0, ..., \pi_j, \pi_{j+1} | B_0, ..., B_k, B_{k+1} | Q_1, ..., Q_n | C \rangle \) if \( B_{k+1} = l \), for any \( i, 0 \leq i \leq k \), not \( l \notin B_i \), and \( \pi_{j+1} \) is a simplification of \( \pi_j \) w.r.t. \( B_{k+1} \).

Notice that a state cannot simultaneously have constraint-, resolution-, and feedback-transitions.

### 4.2.4 Constructive negation

The presence of negation in the bodies of defined rules leads to the occurrence of negative queries. We expand the definition of derivation to account for negative
queries using constructive negation.

**Definition 32 (Frontier).** Let $\Pi$ be a program, $Q$ be a query, and let $r_1, \ldots, r_n$ be the rules of $\Pi$ whose heads unify with $Q$. The frontier of $Q$ with respect to $\Pi$ to a program $S$ is the formula

$$(U_1 \land B_1) \lor \ldots \lor (U_n \land B_n)$$

where each $U_i$ is the most general unifier of $Q$ with the head of $r_i$, and $B_i$ is the body of $r_i$.

**Definition 33 (negation expansion formula).** Let $\neg Q$ be a negative query, the *negation expansion formula* of $\neg Q$ with respect to program $\Pi$ is obtained as follows:

1. Let $(U_1 \land B_1) \lor \ldots \lor (U_n \land B_n)$ be the frontier of $Q$ with respect to $\Pi$. Notice that the following logical equivalence holds:

$$Q \equiv (U_1 \land B_1) \lor \ldots \lor (U_n \land B_n)$$

2. Negating both sides produces:

$$\neg Q \equiv \neg(U_1 \land B_1) \land \ldots \land \neg(U_n \land B_n)$$

3. Obtaining the disjunctive normal form of the right hand side yields

$$\neg Q \equiv (C'_1 \land A'_1 \land N_1) \lor \ldots \lor (C'_m \land A'_m \land N_m) \quad (4.1)$$

where each $C'_i$ is a conjunction of constraints, $A'_i$ is a conjunction of regular e-literals and $N_m$ is a conjunction of negative queries.

We will call the right hand side of formula 4.1 the *negation expansion formula* of $\neg Q$ with respect to $\Pi$. 

53
Definition 34 (negative-transition). Let $S = \langle \pi_0, ..., \pi_j \mid B_0, ..., B_k \mid Q_0, ..., Q_n \mid C \rangle$ be a state of computation, $Q_0 = \neg K$ be a negative query, and $(C_1 \land A_1 \land N_1) \lor ... \lor (C_m \land A_m \land N_m)$ be the negation expansion formula of $\neg K$ w.r.t $\pi_j$.

For each $i, 0 \leq i \leq m$ there is a transition $\langle S, N_i, S' \rangle$, where $S' = \langle \pi_0, ..., \pi_j, \pi_{j+1} \mid B_0, ..., B_k, B_{k+1} \mid N_i, Q_1, ..., Q_n \mid C' \rangle$ if $B_{k+1} = A_i$, $\bigcup_{n=0}^{k+1} B_n$ is consistent, $\pi_{j+1}$ is a simplification of $\pi_j$ with respect to $B_{k+1}$, and $C'$ is a (possible simplified) set of constraints equivalent to $C \cup \{C_i\}$ and $C'$ is solvable (has a solution).

4.2.5 Paths of computation

A path of computation is a sequence of states $\langle S_0, S_1, ... \rangle$ such that there is a transition between $S_i$ and $S_{i+1}$, $i \geq 0$.

Definition 35 (success state). The state $S = \langle \pi_0, ..., \pi_k \mid B_0, ..., B_k \mid \text{nil} \mid C \rangle$ is a success state if $\pi_k$ has no occurrences of regular literals. A path of computation is successful if it is finite and its last element is a success state.

Definition 36 (Initial State). Given a problem $P = \langle \Pi, A, K \rangle$. Let $\pi_0$ be a simplification of $\Pi$ with respect to $A$. The initial state of computation of $P$ is $\langle \pi_0 \mid A \mid K \mid \emptyset \rangle$.

A successful path of a problem $P$ is a finite path of computation whose first state is the initial state of $P$ and whose last state is a success state.

4.2.6 The simple class of programs

The model of computation described before has certain limitations. It only works for a certain class of programs. Here we describe such class.

Given $\Pi$, a program of $\mathcal{AC}(\mathcal{C})$, we say that a literal $l$ depends on literal $l'$ if:

- There is a rule in $\Pi$ whose head is $l$ and whose body contains $l'$.

- There exists a literal $l''$ such that $l$ depends in $l''$ and $l''$ depends on $l'$.

Definition 37 (Simple Program). A program $\Pi$ of $\mathcal{AC}(\mathcal{C})$ is called simple if it has the following properties:
4.2.7 Properties of the model of computation

Claim (Soundness). Let $S$ be the initial state of a simple problem $P$. If there is a successful path of computation from $S$ to a success state $S' = \langle \pi_0, \ldots, \pi_k | B_0, \ldots, B_k | \nil | C \rangle$ then $\langle B, C \rangle$ is a solution to $P$, where $B$ is the union of all $B_i, 0 \leq i \leq k$.

Claim (Weak Completeness). If a simple problem $P$ has a solution then there is a successful path of computation from the initial state of computation of $P$.

4.2.8 General Algorithm

Given $P$, a problem of $\mathcal{AC}(C)$, the above described model of computation can be used to describe a search graph where nodes are states of computation and where finding a solution to $P$ is reduced to finding a path from the initial state of $P$ to a success state.

The following is a general algorithm that could be used to search for a solution to an input problem $P$.

\begin{verbatim}
AC-Solver(problem P = <Pi, A, K> )
BEGIN

   S := initial state of P

   IF S is a success state
      RETURN S

   L := set of transitions possible from S
   push <S, nil, L> into stack

END
\end{verbatim}
WHILE stack is not empty
    <S, a, L> := pop stack
    IF L is not empty
        select and remove an element T from L
        push <S, a, L> into stack
    S' := successor of S after T
    IF S' is a success state
        RETURN S'
    ELSE
        L' := set of transitions possible from S'
        push <S', T, L'> into stack
    RETURN false
END

Selection of the transition to be followed could be accomplished in many different ways. Some transitions could be eliminated, and/or a heuristic function could be used to select the most desirable transition.

4.2.8.1 Heuristics

The following are possible heuristic functions that could be use to guide the search of a solution:

- Use some stochastic approach to choose which transition to explore.
- As queries are finite, record how much time each query takes to be solved, and use this information to decide to solve or not a query when encountered again.
• Use the number of rules with a query in the head to estimate the toughness of a query.

• Look at the query component only after so many successful guesses.

• Look at the syntactical requirements of the query component. If variables in the elements of the query component are unrelated, then the query component could be assumed to be easily satisfiable.

• Use standard heuristics from SAT or different answer set solvers to pick most appropriate literal to guess.

4.3 An Algorithm

In this section, we describe a particular approach for exploring the search graph of an $\mathcal{AC}(\mathcal{C})$ problem $P$. For this approach, we will limit ourselves to a particular subclass of simple programs called the input class. Also, we will limit our search to a simplified graph designed to limit a state’s transitions while preserving solutions.

4.3.1 The input class of programs

Definition 38 (Input Program). We say that $\Pi$, a simple program of $\mathcal{AC}(\mathcal{C})$, is an input program if:

1. $\Pi$ contains no regular variables.

2. $\Pi$ contains no negation $\neg$.

3. Every mixed atom of $\Pi$ has a form $m(\overline{t_r}, \overline{X})$ where $\overline{X}$ is a list of constraint variables.

4. Any atom that occurs in the head of a middle rule does not occurs in the head of any other rule.

5. Negated mixed atoms are not allowed in the middle rules.
6. Every middle rule in \( \Pi \) contains at most one occurrence of a defined atom and no occurrences of constraint atoms.

4.3.2 Simplified Graph

We define the simplified search graph of a problem \( P \) as follows: For every node in the search graph of \( P \):

1. If a state has a constraint-, resolution-, feedback- or negative-transition, remove every other transition.
2. else, If a state has a expand-transition, remove every other transition.
3. else, If a state has a query-transition, remove every other transition.
4. else, pick an undefined literal \( l \) and remove all but the guess-transitions labeled by \( l \) and not \( l \).

Claim. For every successful node \( S \) in the search graph of a problem \( P \), there exists a path from the initial state of \( P \) to \( S \) in the simplified search graph of \( P \).

In the simplified graph, we will call states with constraint-, resolution-, feedback- or negative-transitions blue states, while states with only guess transitions will be called red states.

**Definition 39 (heir).** If \( S \) is a blue state, we say that \( S' \), a descendant of \( S \), is a heir of \( S \) if \( S' \) is a red state or a success state. If \( S \) is a red state. We say that \( S' \), a descendant of \( S \), is a heir of \( S \) if \( S' \) is a red, blue or success state.

Notice that a red state can have at most two closest heirs. Also, the initial state of a problem \( P \) is either a blue state, or has a unique closest red state descendant.

4.3.3 An algorithm for solving \( \mathcal{AC}(\mathcal{C}) \) problems

The following algorithm develops on the previously presented Generic algorithm. A stack is used to keep track of the path being explored. The stack stores a sequence
of red and blue states. For each state in the stack extra information is stored along side it in the stack. For a blue state, the stack stores a set $L$ of heirs already explored. For a red state, the stack stores a set $L$ of outgoing transitions yet to be explored.

AC-Solver-RB(problem $P = <P_i, A, K>$ )
BEGIN
  $S :=$ initial state of $P$

  IF $S$ is a success state
  RETURN $S$

  IF $S$ is not a blue state
  $S' :=$ closest red state descendant of $S$
  $L :=$ set of outgoing choice-transitions possible from $S'$

  push $<S, nil, L>$ into stack

  WHILE stack is not empty
  $<S, a, L> :=$ pop stack
  $S' :=$ nil;

  IF $S$ is a red state AND $L$ is not empty
  remove an element $T$ from $L$
  $S' :=$ heir of $S$ after $T$

  IF $S$ is a blue state
  $S' :=$ hair of $S$ not in $L$
  $L :=$ $L$ union $S'$
IF NOT S' = nil
    push <S, a, L> into stack

IF S' is a success state
    RETURN S'
ELSE
    L' := set of outgoing choice-transitions possible from S'
    push <S', T, L'> into stack

RETURN false
END

In order to refine the algorithm we first need to define a collection of auxiliary functions.

The following sections describe auxiliary functions that will be used in the computation of heirs of states.

4.3.4 Computing Simplifications: The simpl() function.

The function simpl simplifies a program given a set of ground e-literals. Given an r-ground program \( \Pi \) and a set of regular e-literals \( B \), \( \text{simpl}(\Pi, B) \) is the program obtained from \( \Pi \) by:

1. Removing all rules whose head belongs to \( B \).
2. Removing all rules falsified by \( B \).
3. Removing all other occurrences of e-literals from \( B \).

4.3.5 Computing Consequences: The lc() function.

Let \( \Pi \) be an r-ground program and \( B \) be a set of ground regular e-literals. We define \( \text{lc}_0(\Pi, B) \) as the set of ground extended literals that is minimal (set theoretic) and satisfies the following conditions:
1. if the body of a rule $r$ is a subset of $B$, then the head of $r$ belongs to $\text{lc}_0(\Pi, B)$.

2. If an $r$-atom $h$ is not in the head of any rule of $\Pi$ then $\text{not } h \in \text{lc}_0(\Pi, B)$.

3. If rule $r$ is the only rule with head $h$ and $h \in B$, then all regular e-literals in the body of $r$ belong to $\text{lc}_0(\Pi, B)$.

4. If $r$ is a regular rule with head $h$, $\text{not } h \in B$, and all literals in the body of $r$ except $l$ belong to $B$, then $\text{not } l \in \text{lc}_0(\Pi, B)$.

Definition 40 (function $\text{lc()}$). Given a program $\Pi$ and a set of ground regular e-literals $B$, $\text{lc}(\Pi, B)$ is the set of ground extended r-literals defined by the following algorithm:

FUNCTION $\text{lc}(\Pi, B)$
BEGIN
    $A := \text{lc}_0(\Pi, B)$ union $B$
    IF $A$ is consistent
        RETURN $A$
    RETURN false
END

4.3.6 Computing a forced query: The $\text{query()}$ function.

The function $\text{query}$ takes as input an input program $\Pi$ and a ground set of regular e-literals $B$ and returns the query forced by $B$ with respect to $\Pi$.

Before defining the function we will introduce some terminology.

Definition 41 (Active middle rules). Given a program $\Pi$ we define the active middle rules of $\Pi$, $\text{amr}(\Pi)$, as the middle rules of $\Pi$ that contain no regular e-literals in the body.

Definition 42 (Query of a rule). For every rule $R$ in $\text{amr}(\Pi)$ we define the query of an active rule, $q(R)$, as follows:
• If the head of \( R \) is in \( B \) and the body of \( R \) contains defined e-literal \( l \) then \( q(R) = \{ l \} \).

• If the head of \( R \) is in \( B \) and \( R \) contains no defined e-literals then \( q(R) = \{ \text{true} \} \).

• If the head of \( R \) is not in \( B \) and the body of \( R \) contains defined e-literal \( l \) then \( q(R) = \{ \neg l \} \).

• If the head of \( R \) is not in \( B \) and \( R \) contains no defined e-literals then \( q(R) = \{ \text{false} \} \).

Definition 43 (Function query(\( \Pi, B \))). Function query(\( \Pi, B \)) returns

\[ \{ q(R) : R \in \text{amr}(\Pi) \} \]

4.3.7 The init_state() function.

This function takes as input a problem \( P \) and returns the initial state of \( P \).

FUNCTION init_red( \( P = < Pi, A, K > \) )
BEGIN

\( Pi' := \text{simpl}(Pi, A) \)

return \( < Pi', A, K, 0 > \)
END

4.3.8 The init_red() function.

This function takes as input a non blue initial state \( S \) and computes the closest red state descendant of \( S \).

FUNCTION init_red( \( S = < Pi, B, Q, C > \) )
BEGIN
\( B' := \text{lc}(B) \)
B' := B' union {'not l' | 'l' not in up(B', Pi) }

Q' := query(Pi, B')

Pi' := simpl(Pi, B')

return < Pi', B', Q', C >
END

4.3.9 The red_heir() function.

This function takes as input a red state S and an outgoing choice-transition of S and returns the closest heir descendant of S.

FUNCTION red_heir(S = < Pi, B, Q, C >)
BEGIN
    B' := lc(B)
    B' := B' union {'not l' | 'l' not in up(B', Pi) }

    Q' := query(Pi, B')

    Pi' := simpl(Pi, B')

    return < Pi', B', Q', C >
END

4.3.10 The blue_heir() function.

This function takes as input a blue state S and a set L of heirs of S and returns an heir of S not in L.
4.3.11 The Algorithm Revisited

AC-Solver-RB(problem P = <Pi, A, K> )
BEGIN

S := init_state(P)

IF S is a success state
    RETURN S

IF S is not a blue state
    S' := init_red(S)
    L := pick(S')

push <S, nil, L> into stack

WHILE stack is not empty
    <S, a, L> := pop stack
    S' := nil;

    IF S is a red state AND L is not empty
        remove an element T from L
        S' := red_heir(S, T)

    IF S is a blue state
        S' := blue_heir(S, L)
        L := L union S'

    IF NOT S' = nil
        push <S, a, L> into stack
IF \( S' \) is a success state  
    RETURN \( S' \)  
ELSE  
    \( L' := \text{pick}(S') \)  
    push \( <S', T, L'> \) into stack  
RETURN false  
END

4.4 System \textsc{luna}

The above algorithm was implemented over the prototype \textit{ACsolver} to produce system \textsc{luna}. System \textsc{luna} differs from \textit{ACsolver} in the main search function used, and in the way in which queries are processed. In the original \textit{ACsolver} every time a choice-transition is taken a forced query is computed and solved. In contrast, system \textsc{luna} features some incrementally. Answers to queries are stored and reused, avoiding solving the same query repeatedly.

4.5 Experimental Results

We compared systems \textit{ACsolver} and \textsc{luna} in a problem designed to provide different workloads for both the ASP and CLP mechanisms inside each solver. The expectation was that in problems with low CLP workload the difference between both solvers would be minimal, while in problems with higher CLP workload system \textsc{luna} would show increasingly better performance.

Example 10 (House-Flowers Domain). Bob has a collection of houses under mortgage. Each house also has a number of flowerbeds. For each house, Bob wants to know what would be the monthly payment on its mortgage and how many flowers, from 1 to 5, to put in each flowerbed. For the sake of simplicity, each house has the same mortgage value conditions.
The input parameters for an instance of this problem are the number of houses and the number of flowerbeds per house. A solution to a problem instance includes the monthly mortgage payment of the house(s) and a possible allocation of flowers to flowerbeds (A problem instance with one house and one flowerbed will therefore have 5 solutions). In each instance, computation of mortgage payments is done by a CLP mechanism, as such computation requires the use of real numbers. As the number of houses increases, more CLP computations are required to solve the problem. Similarly, as the number of flowerbeds increases, the ASP mechanism needs to make more computations.

Experiments were run on a MAC OSX MacBook Pro with a 2.4 Ghz processor and 2 GB of RAM.

The results are shown in Table 4.5. The table shows the time in milliseconds it took each solver to solve each problem instance. In problem instances with only one house (where the workload of the CLP mechanism is low) LUNA is slightly better than ACsolver, and the difference does not change much as the number of flowerbeds increases. In contrast, in problem instances with 10 houses, (with high CLP workload), LUNA is significantly faster than ACsolver and the advantage greatly increases as the number of flowerbeds increases.

Besides the somewhat synthetic problem shown above, we also tested the solvers in a small collection of problems from [25]. These problems are implementations of scheduling problems modeled in the hybrid action description language H. Implementing these problems requires the use of variables that range over numerical domains and are a perfect example of the intended applications of AC(€) solvers. The results of these tests can be seen in table 4.5. Times are in milliseconds as before. Although the problem sample is small, it is still possible to appreciate that LUNA offers an efficiency advantage over ACsolver.
<table>
<thead>
<tr>
<th>No. Houses</th>
<th>No. Flowerbeds</th>
<th>ACsolver</th>
<th>LUNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>99.33</td>
<td>106.67</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>149.67</td>
<td>108.33</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>164</td>
<td>113.67</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>168</td>
<td>112.33</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>193.33</td>
<td>123.33</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>234.33</td>
<td>144.67</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>298.67</td>
<td>167</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>473.33</td>
<td>153</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>734.67</td>
<td>191.33</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>1033</td>
<td>251.67</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>1358.33</td>
<td>347.67</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>2131.67</td>
<td>628</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>566.33</td>
<td>162</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>924.67</td>
<td>182</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1562.67</td>
<td>277.67</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>2313.33</td>
<td>439.33</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>3073.67</td>
<td>639.67</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
<td>4812.67</td>
<td>1283</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>1693</td>
<td>262.67</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>2858</td>
<td>321.33</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>5260</td>
<td>631.33</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>7877.33</td>
<td>1162.67</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>10650</td>
<td>1939.33</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>33418.33</td>
<td>4176</td>
</tr>
</tbody>
</table>

Table 4.1. House-Flowers Domain

<table>
<thead>
<tr>
<th>No. Houses</th>
<th>AC solver</th>
<th>LUNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanks</td>
<td>277</td>
<td>245</td>
</tr>
<tr>
<td>Timer</td>
<td>209</td>
<td>179</td>
</tr>
<tr>
<td>Ball Drop (planning)</td>
<td>179</td>
<td>174</td>
</tr>
<tr>
<td>Ball Drop (projection)</td>
<td>178</td>
<td>159</td>
</tr>
</tbody>
</table>

Table 4.2. ℋ Domains
Figure 4.1. House-Flowers Domain
Figure 4.2. $H$ Domains
CHAPTER 5
CONFORMANT PLANNING WITH NUMERICAL CONSTRAINTS

In this section, we put together the work from previous chapters.

The definition of the $T^{lp}$ approximation presented in Chapter 3 can be modified to use language $AC(C)$ instead of ASP. We used such modification to build a conformant planner, similar to CPASP$_m$, that runs over LUNA or ACsolver.

We then used this planner to solve a collection of conformant planning and scheduling problems with numerical constraints. Such problems do not appear in the literature of both conformant planning and ASP planning as they require solving constraints over real numbers.

This collection of problems provide us with a set of application oriented benchmarks to compare the efficiency of LUNA and ACsolver.

5.1 Problems

1. Bomb in the Toilet with Timers
   We extend the classic "bomb in the toilet with flushing" domain by giving each bomb a timer set to explode in 240 seconds. Flushing a toilet takes 30 seconds to complete. A solution to this problem involves not only a sequence of actions but that each action is scheduled at a time that warranties safety.

2. Forrest Bus Rides
   Forrest needs to travel to Canada by bus. Such trip requires him to travel through a sequence of cities. In each city, Forrest needs to take a bus that takes him to the next city in the sequence. Each leg of the trip has an expected duration of 80 minutes but a bus can arrive on time or up to 50 minutes late. Fortunately, from each city there is a bus departing every hour to the next city. Which bus tickets should Forrest book such that he does not misses any connecting ride? A conformant plan involves expecting every bus to be late, and booking bus-tickets accordingly.
3. Airport Pickups

Adam, Betty, and Gopal are new students to Texas Tech, arriving to Lubbock via airport. Adam is bringing 3 bags, weighting 5.34kg, 17.561kg and 10.24kg. Berry is bringing two bags, weighting 19.0kg and 10.01kg. Gopal is bringing 4 bags, weighting 3.14159kg, 25.01kg, 24.9kg, and 33.333kg.

There are 3 vehicles available to pick up the new students, a compact car, a van and a truck. The compact car can carry up to 30kg of weight. The van can carry up to 40kg of weight. The truck can carry up to 100kg of weight.

In order to pick up the new students, each vehicle must be driven to the airport, however, this is not possible if any of the vehicles does not have enough fuel. A vehicle can be driven to the gas station and fueled.

Initially, all vehicles are at campus, and it is not know if any of them has enough fuel.

The objective is to find a conformant plan to have all 3 vehicles reach the airport, and have each vehicle pick a student such that the vehicle can load the student’s bags.

A conformant plan for this problem involves driving each vehicle to the gas station, fueling them, driving them to the airport, and picking a student such that the vehicle can load the student’s bags. This problem combines the need for planning using approximations in order to deal with the unknowns and real number constraints to choose appropriate vehicle to pick each student.

5.2 Experimental Results

We used our $A\mathcal{C}(C)$ based conformant planner to solve multiple instances of each problem and evaluated the performance of $ACsolver$ and LUNA. The platform used for testing was a 2.4GHz, 2GB RAM, intel MacBook Pro running OSX 10.6 operating system.

The results of the $Bomb in the Toilet with Timers$ domain can be seen in table 5.2. Surprisingly, such a slight modification of the classic domain provides a good
Table 5.1. BTC w/ Timers

<table>
<thead>
<tr>
<th>No. Packages</th>
<th>ACsolver</th>
<th>LUNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>208.33</td>
<td>169.00</td>
</tr>
<tr>
<td>4</td>
<td>366.00</td>
<td>214.67</td>
</tr>
<tr>
<td>5</td>
<td>621.00</td>
<td>290.33</td>
</tr>
<tr>
<td>6</td>
<td>771.00</td>
<td>380.67</td>
</tr>
<tr>
<td>7</td>
<td>1538.67</td>
<td>538.33</td>
</tr>
<tr>
<td>8</td>
<td>2491.67</td>
<td>689.33</td>
</tr>
<tr>
<td>9</td>
<td>3608.67</td>
<td>962.00</td>
</tr>
</tbody>
</table>

Table 5.2. Forrest

<table>
<thead>
<tr>
<th>No. Cities</th>
<th>ACsolver</th>
<th>LUNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>367</td>
<td>262</td>
</tr>
<tr>
<td>5</td>
<td>406</td>
<td>370</td>
</tr>
<tr>
<td>7</td>
<td>527</td>
<td>460</td>
</tr>
<tr>
<td>9</td>
<td>693</td>
<td>529</td>
</tr>
<tr>
<td>11</td>
<td>743</td>
<td>594</td>
</tr>
<tr>
<td>13</td>
<td>926</td>
<td>785</td>
</tr>
</tbody>
</table>

example of the performance gains of system LUNA. As the number of packages increases, and with it the length of the plan needed, the performance advantage of LUNA becomes more prominent.

The results of the Forrest Bus Rides problem is shown in table 5.2. In this example, the advantage of LUNA over ACsolver is small, and as the number of cities increases, the performance of both solvers seem to degrade at roughly the same rate.

Table 5.2 shows the results for the Airport Pickups domain. Once again, solver LUNA shows a small speed advantage for the smaller instances of this domain, but a greater one for the most difficult instance.
Figure 5.1. Bomb in the toilet with timers Domain

<table>
<thead>
<tr>
<th>No. Pickups</th>
<th>ACsolver</th>
<th>LUNA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>208</td>
<td>208</td>
</tr>
<tr>
<td>3</td>
<td>646</td>
<td>620</td>
</tr>
<tr>
<td>4</td>
<td>1005</td>
<td>935</td>
</tr>
<tr>
<td>5</td>
<td>15218</td>
<td>14753</td>
</tr>
<tr>
<td>6</td>
<td>49650</td>
<td>42487</td>
</tr>
</tbody>
</table>

Table 5.3. Airport Pickups
Figure 5.2. Forrest Bus Drives Domain
Figure 5.3. Airport Pickups Domain
CHAPTER 6
PROOFS

6.1 Preliminaries

6.1.1 A Logic Programming Representation of $T(D)$

We now describe a logic program, called $lp(D)$, which can be used to compute the transitions in $T(D)$. $lp(D)$ consists of rules for reasoning about the effects of actions. Among these rules, the inertia rule encodes the solution to the frame problem, first discussed by John McCarthy and Pat Hayes in their landmark paper on reasoning about actions and changes [63].

The signature of $lp(D)$ includes terms corresponding to fluent literals and actions of $D$, as well as non-negative integers used to represent time steps. We often write $lp(D, n)$ to denote the restriction of the program $lp(D)$ to time steps between 0 and $n$. Atoms of $lp(D)$ are formed by the following (sorted) predicate symbols:

- $fluent(F)$ is true if $F$ is a fluent;
- $literal(L)$ is true if $L$ is a fluent literal;
- $contrary(L, L')$ is true if $L$ and $L'$ are contrary fluent literals;
- $h(L, T)$ is true if the fluent literal $L$ holds at time-step $T$; and
- $o(E, T)$ is true if the elementary action $E$ occurs at time-step $T$.

In our representation, letters $T$, $F$, $L$, $A$, and $E$ (possibly indexed) (resp. $t$, $f$, $l$, $a$, and $e$) are used to represent variables (resp. constants) of sorts time step, fluent, fluent literal, action, and elementary action correspondingly. Moreover, we also use some shorthands: if $a$ is an action then $o(a, T) = \{o(e, T) \mid e \in a\}$. For a set of fluent literals $\gamma$, $h(\gamma, T) = \{h(l, T) \mid l \in \gamma\}$, $not\ h(\gamma, T) = \{not\ h(l, T) \mid l \in \gamma\}$, and $\neg \gamma = \{-l \mid l \in \gamma\}$. The set of rules of $lp(D)$ is divided into the following five subsets:
1. *Dynamic causal laws:* for each statement of the form (3.1) in $D$, the rule:

\[
    h(l, T) \leftarrow o(e, T-1), h(p, T-1), T > 0
\]  

(6.1)

belongs to $lp(D)$. This rule states that if the elementary action $e$ occurs at time step $T$ and the precondition $p$ holds at that time step then $l$ holds afterward.

2. *Static causal laws:* for each statement of the form (3.2) in $D$, $lp(D)$ contains the rule:

\[
    h(l, T) \leftarrow h(p, T)
\]  

(6.2)

This rule states that if $p$ holds at $T$ then so does $l$.

3. *Executability conditions:* for each statement of the form (3.3) in $D$, $lp(D)$ contains the following rule:

\[
    o(a, T), not h(¬p, T)
\]  

(6.3)

This rule states that if the precondition $p$ possibly holds at time step $T$ then the action $a$ cannot occur at that time step.

4. *Inertia:* $lp(D)$ contains the following rule which solves the frame problem [63]:

\[
    h(L, T) \leftarrow h(L, T-1), not h(¬L, T), T > 0
\]  

(6.4)

This rule says that a fluent literal $L$ holds at time step $T$ if it holds at the previous time step and its negation does not hold at $T$. 

77
5. **Auxiliary rules:** \( lp(D) \) also contains the following rules:

\[
\begin{align*}
\text{literal}(F) & \leftarrow \text{fluent}(F) \quad (6.6) \\
\text{literal}(\neg F) & \leftarrow \text{fluent}(F) \quad (6.7) \\
\text{contrary}(F, \neg F) & \leftarrow \text{fluent}(F) \quad (6.8) \\
\text{contrary}(\neg F, F) & \leftarrow \text{fluent}(F) \quad (6.9)
\end{align*}
\]

The first constraint impedes two contrary fluent literals from holding at the same time. The last four rules are used to define fluent literals and complementary fluent literals.

For an action \( a \) and a state \( \sigma \), let

\[
\Phi(a, \sigma) = lp(D, 1) \cup h(\sigma, 0) \cup o(a, 0)
\]  

(6.10)

The next theorem states that the program \( lp(D) \) correctly implements \( T(D) \).

**Theorem 6.** [80, 88] Let \( \sigma \) be a state and \( a \) be an action. Then \( \langle \sigma, a, \sigma' \rangle \in T(D) \) iff there exists an answer set \( A \) of \( \Phi(a, \sigma) \) such that \( s' = \{ l | h(l, 1) \in A \} \).

**Proof.** See [80, 88].

### 6.2 Proofs of Proposition 1 and Theorems 1 & 2

Suppose an action theory \( D \) is given. Let \( \sigma \) be a state, \( s \) be a partial state and \( a \) be an action. From Theorem 6, we have the following result.

**Lemma 1.** Let \( A \) be an answer set of \( \Phi(a, \sigma) \) and let \( \sigma' = \{ l | h(l, 1) \in A \} \). Then \( \sigma' \) is a state and \( \langle \sigma, a, \sigma' \rangle \in T(D) \).

Since this section mainly deals with programs \( \Phi(a, \sigma) \) and \( \Pi(a, s) \) (defined by (3.18)), to simplify the proofs, we remove from the programs atoms and rules that are of no interest.
First, let $\Phi_0(a, \sigma)$ (resp. $\Pi_0(a, s)$) denote the program obtained from $\Phi(a, \sigma)$ (resp. $\Pi(a, s)$) by removing its constraints.

**Lemma 2.** $\Pi_0(a, s)$ is a stratified program.

**Proof.** We need to find a function $\lambda$ which maps atoms in $\Pi_0(a, s)$ into non-negative integers such that for every $r$ in $\Pi_0(a, s)$ with the head $y$,

- $\lambda(y) \geq \lambda(x)$ for every atom $x$ such that $x$ appears in the body of $r$; and
- $\lambda(y) > \lambda(x)$ for every atom $x$ such that not $x$ appears in the body of $r$.

It is easy to check that the following function $\lambda$ satisfies the above property.

- $\lambda(\text{de}(l, 1)) = 1$;
- $\lambda(\text{ph}(l, 1)) = 2$;
- $\lambda(\text{h}(l, 1)) = 3$; and
- $\lambda(\text{at}) = 0$ for any other atom $\text{at}$, e.g., $\lambda(\text{h}(l, 0)) = 0$ for every fluent literal $l$.

**Lemma 3.** The program $\Pi_0(a, s)$ is consistent and has a unique answer set.

**Proof.** It follows from Lemma 2 and [38].

Let $X$ be the set of atoms of the forms fluent(F), literal(L) and contrary(L, L'). Clearly $X$ is a splitting set [54] of both $\Phi_0(a, \sigma)$ and $\Pi_0(a, s)$. It is easy to see that the bottom parts of $\Phi_0(a, \sigma)$ and $\Pi_0(a, s)$ wrt $X$ are positive programs and have only one answer set

$$
\mathcal{U} = \{\text{fluent}(F), \text{literal}(F), \text{literal}(\neg F), \\
\text{contrary}(F, \neg F), \text{contrary}(\neg F, F) \mid F \in \mathcal{F}\}
$$ (6.11)

Let $\Phi_1(a, \sigma)$ and $\Pi_1(a, s)$ denote the evaluation of the top parts of $\Phi_0(a, \sigma)$ and $\Pi_0(a, s)$ wrt $\mathcal{U}$. The rules of these programs are listed below (the condition for each rule follows the rule).
• $\Phi_1(a, \sigma)$ contains the following rules:

\[
\begin{align*}
    h(l, 1) & \leftarrow o(e, 0), h(p, 0) \quad (6.12) \\
    & \quad ([e \text{ causes } l \text{ if } p] \in D) \\
    h(l, 0) & \leftarrow h(p, 0) \quad (6.13) \quad ([l \text{ if } p] \in D) \\
    h(l, 1) & \leftarrow h(p, 1) \quad (6.14) \quad ([l \text{ if } p] \in D) \\
    h(L, 1) & \leftarrow h(L, 0), \neg h(-L, 1) \quad (6.15) \\
    h(\sigma, 0) & \leftarrow \quad (6.16) \\
    h(a, 0) & \leftarrow \quad (6.17)
\end{align*}
\]

• $\Pi_1(a, s)$ contains the following rules:
\begin{align*}
\text{h(l,1)} &\leftarrow \text{o(e,0), h(p,0)} \\
&\quad ([e \text{ causes l if } p] \in \mathcal{D}) \tag{6.18} \\
\text{de(l,1)} &\leftarrow \text{o(e,0), h(p,0)} \\
&\quad ([e \text{ causes l if } p] \in \mathcal{D}) \tag{6.19} \\
\text{ph(l,1)} &\leftarrow \text{o(e,0), not } h(\neg p,0), \text{not de(} \neg l,1) \\
&\quad ([e \text{ causes l if } p] \in \mathcal{D}) \tag{6.20} \\
\text{ph(L,1)} &\leftarrow \text{not } h(\neg L,0), \text{not de(} \neg L,1) \tag{6.21} \\
\text{h(l,0)} &\leftarrow \text{h(p,0)} \\
&\quad ([l \text{ if } p] \in \mathcal{D}) \tag{6.22} \\
\text{h(l,1)} &\leftarrow \text{h(p,1)} \\
&\quad ([l \text{ if } p] \in \mathcal{D}) \tag{6.23} \\
\text{ph(l,1)} &\leftarrow \text{ph(p,1)} \\
&\quad ([l \text{ if } p] \in \mathcal{D}) \tag{6.24} \\
\text{h(L,1)} &\leftarrow \text{not ph(} \neg L,1) \tag{6.25} \\
\text{h(s,0)} &\leftarrow \tag{6.26} \\
\text{h(a,0)} &\leftarrow \tag{6.27}
\end{align*}

Let \( Y \) be the set of atoms of the form \( o(e,0), h(l,0), \) or \( ph(l,0). \) Then it is easy to see that \( Y \) is a splitting set of both \( \Phi_1(a,\sigma) \) and \( \Pi_1(a,s). \)

Because \( \sigma \) is a state, the bottom part of \( \Phi_1(a,\sigma) \) wrt \( Y \) has the unique answer set

\[
V = o(a,0) \cup h(\sigma,0) \tag{6.28}
\]
Hence, the evaluation of the top part of $\Phi_1(a, \sigma)$ wrt $V$ is the following set of rules

\[
\begin{align*}
h(l, 1) & \leftarrow \quad (l \text{ is a direct effect of } a \text{ in } \sigma) \\
h(l, 1) & \leftarrow h(p, 1) \quad (l \text{ if } p \in D) \\
h(L, 1) & \leftarrow \text{not } h(\neg L, 1) \quad (L \text{ holds in } \sigma)
\end{align*}
\]

(6.29) (6.30) (6.31)

Let $\Phi_2(a, \sigma)$ denote this program.

Lemma 4. $A$ is an answer set of $\Phi(a, \sigma)$ iff there exists an answer set $A_1$ of $\Phi_2(a, \sigma)$ such that

\[
A = U \cup V \cup A_1
\]

where $U$ and $V$ are defined by (6.11) and (6.28). Furthermore, we have

\[
h(l, 1) \in A \text{ iff } h(l, 1) \in A_1
\]

Proof. By the splitting set theorem [54].

Similarly, because $s$ is a partial state, the bottom part of $\Pi_1(a, s)$ has the unique answer set

\[
W = o(a, 0) \cup h(s, 0)
\]

(6.32)
The evaluation of the top part of $\Pi_1(a, s)$ wrt $W$ is the following set of rules

\[
\begin{align*}
  h(l, 1) & \leftarrow \quad (6.33) \\
  & \quad (l \text{ is a direct effect of } a \text{ in } s) \\
  \text{dc}(l, 1) & \leftarrow \quad (6.34) \\
  & \quad (l \text{ is a direct effect of } a \text{ in } s) \\
  \text{ph}(l, 1) & \leftarrow \text{not dc}(\neg l, 1) \quad (6.35) \\
  & \quad (l \text{ is a possible direct effect of } a \text{ in } s) \\
  \text{ph}(L, 1) & \leftarrow \text{not dc}(\neg L, 1) \quad (6.36) \\
  & \quad (L \text{ possibly holds in } s) \\
  h(l, 1) & \leftarrow h(p, 1) \quad (6.37) \\
  & \quad ([l \text{ if } p] \in D) \\
  \text{ph}(l, 1) & \leftarrow \text{ph}(p, 1) \quad (6.38) \\
  & \quad ([l \text{ if } p] \in D) \\
  h(L, 1) & \leftarrow \text{not ph}(\neg L, 1) \quad (6.39)
\end{align*}
\]

Let us denote this program by $\Pi_2(a, s)$.

**Lemma 5.** $B$ is an answer set of $\Pi_0(a, s)$ iff there exists an answer set $B_0$ of $\Pi_2(a, s)$ such that

\[
B = U \cup W \cup B_0 \quad (6.40)
\]

where $U$ and $W$ are defined by (6.11) and (6.32). Furthermore, we have

\[
h(l, 1) \in B \text{ iff } h(l, 1) \in B_0 \quad (6.41)
\]

**Proof.** It follows from the splitting set theorem. \qed

Let us further split the program $\Pi_2(a, s)$ by using the splitting set consisting of atoms of the form $\text{dc}(l, 1)$. The bottom part of $\Pi_2(a, s)$ contains only rules of the
form (6.34) and thus it has the only answer set

\[ \{ \text{de}(l, 1) \mid l \text{ is a direct effect of } a \} \]

Hence, the evaluation of the top part of \( \Pi_2(a, s) \) wrt this answer set, denoted by \( \Pi_3(a, s) \), contains the following rules:

\[
\begin{align*}
\text{h}(l, 1) & \leftarrow (6.42) \\
& (l \text{ is a direct effect of } a \text{ in } s) \\
\text{ph}(l, 1) & \leftarrow (6.43) \\
& (l \text{ is a possible direct effect of } a \text{ in } s, \neg l \text{ is not a direct effect of } a) \\
\text{ph}(L, 1) & \leftarrow (6.44) \\
& (L \text{ possibly holds in } s, \neg L \text{ is not a direct effect of } a) \\
\text{h}(l, 1) & \leftarrow \text{h}(p, 1) (6.46) \\
& ([l \text{ if } p] \in \mathcal{D}) \\
\text{ph}(l, 1) & \leftarrow \text{ph}(p, 1) (6.47) \\
& ([l \text{ if } p] \in \mathcal{D}) \\
\text{h}(L, 1) & \leftarrow \neg \text{ph}(\neg L, 1) (6.48)
\end{align*}
\]

**Lemma 6.** B is an answer set of \( \Pi_0(a, s) \) iff there exists an answer set \( B_1 \) of \( \Pi_3(a, s) \) such that

\[ B = U \cup W \cup \{ \text{de}(l, 1) \mid l \text{ is a direct effect of } a \text{ in } s \} \cup B_1 \quad (6.49) \]

Furthermore, we have

\[ \text{h}(l, 1) \in B \text{ iff } \text{h}(l, 1) \in B_1 \quad (6.50) \]

**Proof.** It follows from Lemma 5 and the splitting set theorem. \( \square \)

For a set of atoms \( \mathcal{X} \), let \( \Phi_2^X(a, \sigma) \) (resp. \( \Pi_3^X(a, s) \)) denote the reduct of \( \Phi_2(a, \sigma) \)
(resp. $\Pi_3(a, s)$) wrt $X$. That is, $\Phi_2^X(a, \sigma)$ is the following set of rules:

$$h(l, 1) \leftarrow$$  
(l is a direct effect of $a$ in $\sigma$)  
(6.51)

$$h(l, 1) \leftarrow h(p, 1)$$  
([l if $p$] $\in \mathcal{D}$)  
(6.52)

$$h(L, 1) \leftarrow$$  
(L holds in $\sigma$, $h(\neg L, 1) \not\in X$)  
(6.53)

and $\Pi_3^X(a, s)$ is the following set of rules:

$$h(l, 1) \leftarrow$$  
(l is a direct effect of $a$ in $s$)  
(6.54)

$$ph(l, 1) \leftarrow$$  
(l is a possible direct effect of $a$ in $s$, $\neg l$ is not a direct effect of $a$)  
(6.55)

$$ph(L, 1) \leftarrow$$  
(L possibly holds in $s$, $\neg L$ is not a direct effect of $a$)  
(6.56)

$$h(l, 1) \leftarrow h(p, 1)$$  
([l if $p$] $\in \mathcal{D}$)  
(6.57)

$$ph(l, 1) \leftarrow ph(p, 1)$$  
([l if $p$] $\in \mathcal{D}$)  
(6.58)

$$h(L, 1) \leftarrow$$  
($ph(\neg L, 1) \not\in X$)  
(6.59)

Let us prove the following lemma

Lemma 7. Let $A_1$ be an answer set of $\Phi_2(a, \sigma)$ and $B_1$ be an answer set of $\Pi_3(a, s)$. If $s \subseteq \sigma$ then for any fluent literal $l$, $ph(l, 1) \in B_1$ implies that $h(l, 1) \in A_1$.

Proof. Suppose $s \subseteq \sigma$. Let $P = \Phi_2^{A_1}(a, \sigma)$ and $Q = \Pi_3^{B_1}(a, s)$. Because $A_1$ and
\( B_1 \) are answer sets of \( \Phi_2(a, \sigma) \) and \( \Pi_3(a, s) \) respectively, we have

\[
A_1 = \bigcup_i T^i_P(\emptyset) \\
B_1 = \bigcup_i T^i_Q(\emptyset)
\]

(6.60) (6.61)

where \( T_P \) and \( T_Q \) are the immediate consequence operators of programs \( P \) and \( Q \) respectively.

First of all, we show that for any integer \( i \geq 0 \) the following result holds

\[
h(l, 1) \in T^i_P(\emptyset) \Rightarrow ph(l, 1) \in T^i_Q(\emptyset)
\]

(6.62)

Let us prove by induction on \( i \).

1. Base case: \( i = 0 \). (6.62) holds because \( T^0_P(\emptyset) = T^0_Q(\emptyset) = \emptyset \).

2. Inductive step: suppose (6.62) holds for \( i = k \), we need to show that it also holds for \( i = k + 1 \).

Let \( l \) be a fluent literal such that \( h(l, 1) \in T^{k+1}_P(\emptyset) \). By the definition of \( T^{k+1}_P(\emptyset) \), there are three cases

(a) \( h(l, 1) \) holds in \( T^{k+1}_P(\emptyset) \) by rule (6.51). This means that \( l \) is a direct effect of \( a \) in \( \sigma \). Observe that a direct effect of \( a \) in \( \sigma \) is always a possible direct effect of \( a \) in \( s \). Furthermore, because \( D \) is consistent, \( \neg l \) is not a direct effect of \( a \) in \( \sigma \). Hence, \( \neg l \) is not a direct effect of \( a \) in \( s \). As a result, rule (6.55) belongs to \( Q \), which implies that \( ph(l, 1) \in T^{k+1}_Q(\emptyset) \).

(b) \( h(l, 1) \) holds in \( T^{k+1}_P(\emptyset) \) by rule (6.52). This means that there exists a static causal law (3.2) such that

\[
h(p, 1) \subseteq T^k_P(\emptyset)
\]
By the inductive hypothesis, we have

\[ \text{ph}(p, 1) \subseteq T_Q^k(\emptyset) \]

By rule (6.58), this implies that \( \text{ph}(l, 1) \in T_Q^{k+1}(\emptyset) \).

(c) \( \text{h}(l, 1) \) holds in \( T_{p}^{k+1}(\emptyset) \) by rule (6.53). This means that \( l \) holds in \( \sigma \) and \( \text{h}(\neg l, 1) \not\in A_1 \).

Because \( l \) holds in \( \sigma \), \( l \) possibly holds in \( s \). On the other hand, \( \text{h}(\neg l, 1) \not\in A_1 \) implies that \( \neg l \) is not a direct effect of \( a \) in \( \sigma \) (because of rule (6.51)). Hence, \( \neg l \) is not a direct effect of \( a \) in \( s \). Accordingly, \( Q \) contains the rule of the form (6.56) with \( L = l \). Thus, it follows that \( \text{ph}(l, 1) \in T_Q^{k+1}(\emptyset) \).

As a result, (6.62) holds. The lemma directly follows from (6.62), (6.60), and (6.61).

Lemma 8. Let \( A \) be an answer set of \( \Phi(a, \sigma) \) and \( B \) be an answer set of \( \Pi_0(a, s) \). If \( s \subseteq \sigma \) then for any fluent literal \( l \), \( \text{h}(l, 1) \in B \) implies \( \text{h}(l, 1) \in A \).

Proof. By Lemmas 4 and 6, the programs \( \Phi_2(a, \sigma) \) and \( \Pi_3(a, \sigma) \) have answer sets \( A_1 \) and \( B_1 \), respectively, such that

\[ A = U \cup V \cup A_1 \]

and

\[ B = U \cup W \cup \{ \text{de}(l, 1) \mid l \text{ is a direct effect of } a \text{ in } s \} \cup B_1 \]

Let \( P \) and \( Q \) be programs defined in Lemma 7. First of all, we will show that

\[ \text{h}(l, 1) \in T_Q^i(\emptyset) \Rightarrow \text{h}(l, 1) \in A_1 \]

for \( i \geq 0 \) by using induction on \( i \).

1. Base case: trivial because there exist no fluent literal \( l \) such that \( l \in T_Q^i(\emptyset) = \emptyset \).
2. Inductive step: suppose (6.63) holds for \( i \leq k \).

Let \( l \) be a fluent literal such that \( h(l, 1) \in T_{Q}^{k+1}(\emptyset) \). We need to show that \( h(l, 1) \in A_1 \). Consider the following cases

(a) \( h(l, 1) \in T_{Q}^{k+1}(\emptyset) \) by rule (6.54). This means that \( l \) is a direct effect of \( a \) in \( s \). On the other hand, a direct effect of \( a \) in \( s \) is also a direct effect of \( a \) in \( \sigma \). As a result, \( l \) is a direct effect of \( a \) in \( \sigma \). By rule (6.29), this implies that \( h(l, 1) \in A_1 \).

(b) \( h(l, 1) \in T_{Q}^{k+1}(\emptyset) \) by rule (6.57). This implies that \( h(p, 1) \subseteq T_{Q}^{k}(\emptyset) \). By the inductive hypothesis, we have \( h(p, 1) \subseteq A_1 \). As a result, by rule (6.30), it follows that \( h(l, 1) \in A_1 \).

(c) \( h(l, 1) \in T_{Q}^{k+1}(\emptyset) \) by rule (6.59). This implies that \( ph(\neg l, 1) \not\subseteq B_1 \). It follows from Lemma 7 that \( h(\neg l, 1) \not\subseteq A_1 \), i.e.,

\[
h(\neg l, 1) \not\subseteq A \tag{6.64}
\]

Let \( \sigma' = \{ g \mid h(g, 1) \in A \} \). From (6.64), we have \( \neg l \not\subseteq \sigma' \). On the other hand, by Lemma 1, \( \sigma' \) is a state, which implies that either \( l \) or \( \neg l \) belongs to \( \sigma' \). Accordingly, we have \( l \in \sigma' \). From this, it follows that \( h(l, 1) \in A \) and thus \( h(l, 1) \in A_1 \).

From (6.63) and (6.61), we have

\[
h(l, 1) \in B_1 \Rightarrow h(l, 1) \in A_1
\]

On the other hand, by Lemmas 4 and 6, we have

\[
h(l, 1) \in A_1 \text{ iff } h(l, 1) \in A
\]

and

\[
h(l, 1) \in B_1 \text{ iff } h(l, 1) \in B
\]

88
Consequently, we can conclude that the lemma holds.

\[ \square \]

Lemma 9. \( \Pi(a, s) \) is consistent iff \( a \) is safe in \( s \).

Proof. Suppose \( \Pi(a, s) \) is consistent. Let \( B \) denote the answer set of \( \Pi(a, s) \). According to the definition of an answer set of a program with constraints, \( B \) is an answer set of \( \Pi_0(a, s) \) and \( B \) does not violate constraints (6.3) and (6.5). This implies that there exists no impossibility condition

\[ \text{impossible } b \text{ if } p \]

such that \( b \subseteq a \) and \( p \) possibly holds in \( s \). By definition, this means \( a \) is safe in \( s \).

Now suppose that \( a \) is safe in \( s \). By Lemma 3, \( \Pi_0(a, s) \) has a unique answer set \( B \). We will show that \( B \) is also an answer set of \( \Pi(a, s) \) by showing that it satisfies constraints (6.3) and (6.5):

1. Constraint (6.3): Trivial because \( a \) is safe in \( s \).

2. Constraint (6.5): Since \( s \) is a partial state, there exists a state \( \sigma \) such that \( s \subseteq \sigma \). As \( a \) is safe in \( s \), it is executable in \( \sigma \). By Theorem 1, it follows that the program \( \Phi(a, \sigma) \) has an answer set \( A \) and this answer set satisfies constraint (6.5). By Lemma 8, it follows that \( B \) also satisfies constraint (6.5).

\[ \square \]
Lemma 10. If $\Pi(a, s)$ is consistent then the only answer set of $\Pi(a, s)$ is the answer set of $\Pi_0(a, s)$.

Proof. The lemma follows from Lemma 3 and from the fact that $\Pi(a, s)$ differs from the program $\Pi_0(a, s)$ in two constraints (6.3) and (6.5) only. □

6.2.1 Proof of Proposition 1

Suppose $\Pi(a, s)$ is consistent. Lemma 10 implies that the only answer set of $\Pi(a, s)$ is the answer set of $\Pi_0(a, s)$. Let

$$s' = \{l \mid h(l, 1) \in B\}$$

To complete the proof, we need to show that $s'$ is a partial state. First of all, observe that $s'$ satisfies all the static causal laws of $D$ because of constraint (3.8). So, we only need to show that there exists a state $\sigma'$ such that $s' \subseteq \sigma'$.

Since $s$ is a partial state there exists a state $\sigma$ such that $s \subseteq \sigma$. Because $\Pi(a, s)$ is consistent, by Lemma 9, $a$ is safe in $s$. Thus, it is executable in $s$. Since we assume $D$ is consistent, there must be a state $\sigma'$ such that $\langle \sigma, a, \sigma' \rangle \in T(D)$. By Theorem 6, this implies that the program $\Phi(a, \sigma)$ has an answer set $A$ such that $\sigma' = \{l \mid h(l, 1) \in A\}$. By Lemma 8, it is easy to see that $s' \subseteq \sigma'$.

6.2.2 Proof of Theorem 1

Let $\langle s, a, s' \rangle$ be a transition in $T^{lp}(D)$. It follows from Definition 13 that the program $\Pi(a, s)$ is consistent and has an answer set $B$ such that $s' = \{l \mid h(l, 1) \in B\}$. Note that by Lemma 10, such an answer set $B$ is unique and it is also the answer set of $\Pi_0(a, s)$.

First, let us show that $T^{lp}(D)$ is an approximation of $T(D)$. Clearly, to prove that, it suffices to show that for every $\sigma \in \text{comp}(s)$,

1. $a$ is executable in $\sigma$

2. for every state $\sigma'$ such that $\langle \sigma, a, \sigma' \rangle \in T(D)$, $s' \subseteq \sigma'$.
Consider a state $\sigma \in \text{comp}(s)$. By Lemma 9, $a$ is safe in $s$. Because $s \subseteq \sigma$, $a$ is executable in $\sigma$. Now let $\sigma'$ be a state such that $\langle \sigma, a, \sigma' \rangle \in T(D)$. By Theorem 6, this implies that there exists an answer set $A$ of $\Phi(a, \sigma)$ such that $\sigma' = \{ l \mid h(l, 1) \in A \}$. By Lemma 8, we have $s' = \{ l \mid h(l, 1) \in B \} \subseteq \{ l \mid h(l, 1) \in A \} = \sigma'$.

We have showed that $T^p(D)$ is an approximation of $T(D)$. The determinism of $T^p(D)$ follows directly from the fact that $B$ is unique.

### 6.3 Proof of Theorem 2

This section contains the proof of Theorem 2. We assume that a planning problem $P = \langle D, \text{comp}(s^0), s^f \rangle$ is given. For the sake of simplicity of the proof, similarly to the previous section, we will begin with a simplification of the program $\pi(P, n)$.

Let $\pi_0(P, n)$ be the program obtained from $\pi(P, n)$ by removing constraints (6.3) and (6.5). Let $X$ be the set of atoms of the forms fluent($F$), literal($L$) and contrary($L, L'$). Then, $X$ is a splitting set of $\pi_0(P, n)$. The bottom part of $\pi_0(P, n)$ is a positive program and has a unique answer set $U$ defined by (6.11). Let $\pi_1(P, n)$ denote the evaluation of the top part of $\pi_0(P, n)$ wrt $U$.

**Lemma 11.** A set of atoms $C$ is an answer set of $\pi_0(P, n)$ iff $C = C_1 \cup U$ where $C_1$ is an answer set of $\pi_1(P, n)$.

**Proof.** Follows from the splitting set theorem. \[\square\]

For an integer $0 \leq i \leq n$, let $X_i$ denote the set of atoms whose time parameters are less than or equal to $i$. Then, it is easy to see that the sequence $\langle X_i \rangle_{i=0}^n$ is a splitting sequence [54] of $\pi_1(P, n)$.

**Lemma 12.** A set of atoms $C_1$ is an answer set of $\pi_1(P, n)$ iff there is a sequence of sets of atoms $\langle D_i \rangle_{i=0}^n$ such that the following conditions are satisfied.

1. $D_0$ is an answer set of $\mu_0 = b_{X_0}(\pi_1(P, n))$ \[ (6.65) \]
2. For every $1 \leq i \leq n$, $D_i$ is an answer set of

$$\mu_i = e_{X_i}(b_{X_i}(\pi_1(P, n)) \setminus b_{X_{i-1}}(\pi_1(P, n)), \bigcup_{1 \leq j \leq i-1} D_j) \tag{6.66}$$

3.

$$C_1 = \bigcup_{i=0}^{n} D_i \tag{6.67}$$

where $b_{X}(P)$ denote the bottom part of a program $P$ wrt $X$ and $e_{X}(Q, V)$ denote the evaluation of a program $Q$ relative to $V$.

Proof. Follows from the splitting sequence theorem [54].

Now suppose that $C$ is an answer set of $\pi(P, n)$. By definition $C$ is also an answer set of $\pi_0(P, n)$ and $C$ does not violate any constraint of $\pi(P, n)$. By Lemma 11, the program $\pi_1(P, n)$ has an answer set $C_1$ such that $C = C_1 \cup U$. By Lemma 12, it follows that there exists a sequence of sets of atoms $\langle D_i \rangle_{i=1}^{n}$ that satisfies (6.65)–(6.67). Let $s_i = \{l \mid h(l, i) \in D_i\}$ and let $a_i = \{l \mid o(e, i) \in D_i\}$. It is easy to see that $\mu_0$ is the following set of rules

$$h(l, 0) \leftarrow h(p, 0) \quad \tag{6.68}$$

$$[[l \text{ if } p] \in \mathcal{D}]$$

$$h(s^0, 0) \leftarrow \quad \tag{6.69}$$

$$o(E, 0) \lor \neg o(E, 0) \leftarrow \quad \tag{6.70}$$
and for \( i \geq 1 \), \( \mu_i \) is the following set of rules

\[
\begin{align*}
\text{h}(l, i) & \leftarrow \quad \text{(l is a direct effect of } a_{i-1} \text{ in } s_{i-1}) \quad (6.71) \\
\text{de}(l, i) & \leftarrow \quad \text{(l is a direct effect of } a_{i-1} \text{ in } s_{i-1}) \quad (6.72) \\
\text{ph}(l, i) & \leftarrow \text{not } \text{de}(\neg l, i) \quad \text{(6.73)} \\
\text{ph}(L, i) & \leftarrow \text{not } \text{de}(\neg L, i) \quad (6.74) \\
\text{h}(L, i) & \leftarrow \text{not } \text{ph}(\neg L, i) \quad (6.75) \\
\text{h}(l, i) & \leftarrow \text{h}(p, i) \quad \text{([l if } p \text{] } \in \mathcal{D}) \quad (6.76) \\
\text{ph}(l, i) & \leftarrow \text{ph}(p, i) \quad \text{([l if } p \text{] } \in \mathcal{D}) \quad (6.77) \\
o(E, 0) \lor \neg o(E, 0) & \leftarrow \quad (6.78)
\end{align*}
\]

Lemma 13. If \( s_{i-1} \) is a partial state then \( \langle s_{i-1}, a_{i-1}, s_i \rangle \in T^{lp}(\mathcal{D}) \)

Proof. Suppose \( s_{i-1} \) is a partial state. To prove that \( \langle s_{i-1}, a_{i-1}, s_i \rangle \in T^{lp}(\mathcal{D}) \) we need to show that \( \Pi(a_{i-1}, s_{i-1}) \) is consistent and its only answer set \( B \) satisfies

\[ \{l \mid \text{h}(l, 1) \in B\} = s_i \]

First, observe that because \( C \) is an answer set of \( \pi(P, n) \), its satisfies the constraint \( (6.3) \). As a result, \( a_{i-1} \) is safe in \( s_{i-1} \). By Lemma 9, this implies that \( \Pi(a_{i-1}, s_{i-1}) \) is consistent and thus, by Proposition 1, it has a unique answer set \( B \). By Lemma 5, the program \( \Pi_2(a_{i-1}, s_{i-1}) \) (rules \( (6.33)-(6.39) \) with \( a = a_{i-1} \) and \( s = s_{i-1} \)) has
an answer set $B_0$ such that

$$\{l \mid h(l, 1) \in B\} = \{l \mid h(l, 1) \in B_0\}$$

Furthermore, such $B_0$ is unique because $\Pi_2(a_{i-1}, s_{i-1})$ is stratified.

Observe that the program $\Pi_2(a_{i-1}, s_{i-1})$ is the same as $\mu_i$ except that the time parameter of predicates in the former is 1 while it is $i$ in the latter. Hence, we have

$$\{l \mid h(l, 1) \in B_0\} = s_i$$

That is, the lemma holds. \hfill \square

Let us go back to the proof of Theorem 2. It is easy to see that $s_0 = s^0$ and thus it is a partial state. By Lemma 13, it easy to see that for all $1 \leq i \leq n$ we have $\langle s_{i-1}, a_{i-1}, s_i \rangle \in T^{lp}(D)$.

Hence, we have $\langle s^0, \alpha, s_n \rangle \in T^{lp}(D)$. On the other hand, because $C$ satisfies constraint (3.20), we have $s^f \subseteq s_n$. Accordingly, $\alpha$ is a solution of $P$. Thus, the theorem is proved.

### 6.4 Preliminaries for the proof of Theorem 3

For convenience, given an action theory $D$, for a set $\Gamma$ of states and action $a$, by $\text{Res}(a, \Gamma)$ we denote the set of possible successor states of states in $\Gamma$ after the execution of $a$, i.e.,

$$\text{Res}(a, \Gamma) = \{\sigma' \mid \sigma \in \Gamma, \langle \sigma, a, \sigma' \rangle \in T(D)\}$$

For a chain of events $\alpha$, by $\text{Res}(\alpha, \Gamma)$ we denote the set of possible states reachable from some state in $\Gamma$ after the execution of $\alpha$, i.e.,

$$\text{Res}(\alpha, \Gamma) = \{\sigma' \mid \sigma \in \Gamma, \langle \sigma, \alpha, \sigma' \rangle \in T(D)\}$$

**Proposition 3.** Let $D$ be a simple action theory. Let $\langle \sigma, a, \sigma' \rangle \in T(D)$ and
\[ \langle s, a, s' \rangle \in T^1p(D). \] If \( s \subseteq \sigma \) then for every fluent literal \( l \in \sigma' \setminus s' \), we have \( l \triangleleft (\sigma \setminus s) \).

**Proof.** See section 6.5.1.

**Proposition 4.** Let \( D \) be a simple action theory, \( S \) be a set of states, \( \delta \) be a partial state, and \( \sigma \) be a set of fluent literals such that \( S \gg_{\sigma} \delta \). Then,

\[
\bigcup_{s \in S} s \cap \sigma = \delta \cap \sigma
\]

**Proof.** See section 6.5.2.

**Proposition 5.** Let \( D \) be a simple action theory, \( \Gamma \) be a set of states, \( s \) be a partial state, and \( L \) be a set of fluent literals such that \( \Gamma \gg_{\sigma} s \). For any action \( a \), if \( a \) is executable in \( \Gamma \) then

1. \( a \) is safe in \( s \),
2. \( \text{Res}(a, \Gamma) \gg_{L} s' \) where \( \langle s, a, s' \rangle \in T^1p(D) \).

**Proof.** See section 6.5.3.

The second property can be extended to a chain of events \( \alpha \) as follows.

**Proposition 6.** Let \( D \) be a simple action theory, \( \Gamma \) be a set of states, \( s \) be a partial state, and \( L \) be a set of fluent literals such that \( \Gamma \gg_{L} s \). For any chain of events \( \alpha \), if \( \alpha \) is safe in \( \Gamma \) then

1. \( \alpha \) is safe in \( s \)
2. \( \text{Res}(a, \Gamma) \gg_{L} s' \) where \( \langle s, a, s' \rangle \in T^1p(D) \).

**Proof.** See section 6.5.4.
6.5 Proofs of Propositions 3–6 and Theorem 3

This section contains the proofs of Propositions 3–6 and Theorem 3. We assume
that a simple planning problem \( P \) is given. In addition, for simplicity, we assume
that the body of each static causal law of \( D \) has exactly one fluent literal as with
some minor changes, the proofs in this section can be applied to simple action
theories with arbitrary simple static causal laws, including those with an empty
body. To make the proofs easy to follow, let us define some notions.

**Definition 1.** Let \( a \) be an action and \( \sigma \) be a state. A fluent literal \( l \) is called an
*effect* of \( a \) in \( \sigma \) if either

1. \( l \) is a direct effect of \( a \) in \( \sigma \); or
2. \( D \) contains a static causal law
   
   \[ l \text{ if } g \]

   such that \( g \) is an effect of \( a \) in \( \sigma \).

**Definition 2.** Let \( a \) be an action and \( s \) be a state. A fluent literal \( l \) is called a
*possible effect* of \( a \) in \( s \) iff either

1. \( l \) is a possible direct effect of \( a \) in \( s \); or
2. \( l \) possibly holds by inertia, i.e., \( l \) possibly holds in \( s \) and \( \neg l \) is not a direct
   effect of \( a \) in \( s \); or
3. \( D \) contains a static causal law
   
   \[ l \text{ if } g \]

   such that \( g \) is a possible effect of \( a \) in \( s \).

The proofs in this section will make use of programs \( \Phi_2(a, \sigma) \) (rules (6.29)–(6.31))
and \( \Pi_3(a, s) \) (rules (6.42)–(6.48)) and some results from sections 6.2 & 6.3.

Let \( \sigma \) and \( \sigma' \) be states, \( s \) and \( s' \) be partial states and \( a \) be an action such that
\( \langle \sigma, a, \sigma' \rangle \in T(D) \) and \( \langle s, a, s' \rangle \in T^{lp}(D) \). By Theorems 6 & 1, and Definition 13, the
programs $\Phi(a, \sigma)$ and $\Pi(a, s)$ have answer sets $A$ and $B$, respectively, such that

$$\sigma' = \{ l \mid h(l, 1) \in A \} \quad (6.79)$$

and

$$s' = \{ l \mid h(l, 1) \in B \} \quad (6.80)$$

By Lemmas 4 and 6, this implies that the programs $\Phi_2(a, s)$ and $\Pi_3(a, s)$ have answer sets $A_1$ and $B_1$, respectively such that

$$A = U \cup V \cup A_1 \quad (6.81)$$

$$B = U \cup W \cup \{ de(l, 1) \mid l \text{ is a direct effect of } a \text{ in } s \} \cup B_1 \quad (6.82)$$

where $U$, $V$, and $W$ are defined by (6.11) and (6.28), and (6.32). Furthermore, we have

$$h(l, 1) \in A \text{ iff } h(l, 1) \in A_1 \quad (6.83)$$

$$h(l, 1) \in B \text{ iff } h(l, 1) \in B_1 \quad (6.84)$$

Let $P$ and $Q$ be the reducts of $\Phi_2(a, \sigma)$ and $\Pi_3(a, s)$ wrt $A_1$ and $B_1$, respectively. That is, $P$ is the set of rules (6.51)–(6.53) where $X = A_1$, and $Q$ is the set of rules (6.54)–(6.59) where $X = B_1$.

Lemma 14. If $g \notin \sigma$ and $h(g, 1) \in A_1$ then $g$ is an effect of $a$ in $\sigma$.

Proof. Let $i \geq 0$ be an arbitrary integer. Because $A_1 = \bigcup_i T_i^b(\emptyset)$, to prove the lemma, it suffices to show that if $g \notin \sigma$ and $h(g, 1) \in T_i^b(\emptyset)$ then $g$ is an effect of $a$ in $\sigma$. We prove this by induction on $i$.

1. Base case: $i = 0$. Trivial because there exists no fluent literal $g$ such that $h(g, 1) \in T_0^b(\emptyset) = \emptyset$.

2. Inductive step: Suppose the lemma holds for $i \leq k$. We will show that it also holds for $i = k + 1$. Let $g$ be a fluent literal such that
Because \( g \) does not hold in \( \sigma \), the rule (6.53) with \( L = g \) does not belong to \( P \). As a result, there are two possibilities for \( h(g, 1) \in T^{k+1}_p(\emptyset) \):

(a) \( g \) is a direct effect of \( a \) in \( \sigma \). By Definition 1, \( g \) is an effect of \( a \) in \( \sigma \).

(b) \( D \) contains a static causal law

\[
g \text{ if } h
\]

such that

\[
h(h, 1) \in T^k_p(\emptyset)
\]

It is easy to see that

\[
h \not\in \sigma
\]

because if otherwise, we would have \( g \in \sigma \) (note that \( \sigma \) is a state and thus it satisfies static causal law (6.85)).

From (6.86) and (6.87) and by the inductive hypothesis, it follows that \( h \) is an effect of \( a \) in \( \sigma \). Hence, by Definition 1, \( g \) is also an effect of \( a \) in \( \sigma \).

\[ \square \]

**Lemma 15.** If \( ph(g, 1) \in B_1 \) then \( g \) is a possible effect of \( a \) in \( s \).

**Proof.** Let \( i \geq 0 \) be an arbitrary integer. Clearly, to prove the lemma, it suffices to show that

\[
\text{if } ph(g, 1) \in T^1_q(\emptyset) \text{ then } g \text{ is a possible effect of } a \text{ in } s
\]

(6.88)

Let us prove (6.88) by induction on \( i \).
1. Base case: \( i = 0 \). Trivial because there is no \( i \) such that \( \text{ph}(g, 1) \in T_Q^0(\emptyset) = \emptyset \).

2. Inductive step: Suppose (6.88) is true for \( i \leq k \). We will show that it is also true for \( i = k + 1 \).

Let \( g \) be a fluent literal such that \( \text{ph}(g, 1) \in T_Q^{k+1}(\emptyset) \). Recall that \( Q \) is the set of rules of the form (6.54)–(6.59) where \( X = B_1 \). Hence, there are three possibilities for \( \text{ph}(g, 1) \in T_Q^{k+1}(\emptyset) \).

(a) \( g \) is a possible direct effect of \( a \) in \( s \) and \( \neg g \) is not a direct effect of \( a \) in \( s \). By definition, \( g \) is also a possible effect of \( a \) in \( s \).

(b) \( g \) possibly holds in \( s \) and \( \neg g \) is not a direct effect of \( a \) in \( s \). By definition, in this case \( g \) is also a possible effect of \( a \) in \( s \).

(c) \( D \) contains a static causal law

\[
g \text{ if } h
\]

such that \( \text{ph}(h, 1) \in T_Q^k(\emptyset) \). By the inductive hypothesis, \( h \) is a possible effect of \( a \) in \( s \). Hence, by Definition 2, \( g \) is also a possible effect of \( a \) in \( s \).

So, (6.88) is true. The lemma follows directly from this result. \( \square \)

6.5.1 Proof of Proposition 3

Suppose \( s \subseteq \sigma \) and \( l \) be a fluent literal in \( \sigma' \setminus s' \). We need to show that there exists a fluent literal \( g \in \sigma \setminus s \) such that \( l \triangleleft g \).

If \( l \in \sigma \setminus s \) then the proposition is trivial because by definition, \( l \) depends on itself and thus we can take \( g = l \in \sigma \setminus s \) to have \( l \triangleleft g \). Now, consider the case that \( l \not\in \sigma \setminus s \). There are two possibilities

1. \( l \not\in \sigma \), or
2. \( l \in s \).
Let us consider each possibility in turn.

1. \( l \not\in \sigma \). As \( l \in \sigma' \setminus s' \), we have \( l \in \sigma' \). This implies that \( h(l, 1) \in A_1 \). According to Lemma 14, \( l \) is an effect of \( a \) in \( \sigma \). It follows from Definition 1, that one of the following two cases occurs

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

   \[ \begin{align*}
   e & \text{ causes } l \text{ if } p \\
   \text{such that } e & \in a \text{ and } p \text{ holds in } \sigma.
   \end{align*} \]

   If \( p \) holds in \( s \) then \( l \) is a direct effect of \( a \) in \( s \). By rule (6.54), we have \( h(l, 1) \in B_1 \), i.e., \( l \in s' \). This contradicts to \( l \in \sigma' \setminus s' \).

   Because \( p \) holds in \( \sigma \) and does not hold in \( s \), we have

   \[ \begin{align*}
   p & \subseteq \sigma \quad \text{and} \quad p \not\subseteq s
   \end{align*} \]

   As a result, there exists a fluent literal \( g \in p \) such that \( g \in \sigma \setminus s \). By the definition of dependencies (Definition 16), we have \( l \triangleleft g \).

   (b) \( D \) contains a dynamic causal law \( e \) causes \( l_0 \) if \( p \) and a sequence of static causal laws \([l_1 \text{ if } l_0], [l_2 \text{ if } l_1], \ldots, [l_n \text{ if } l_{n-1}], [l \text{ if } l_n] \) such that \( e \in a \) and \( p \) holds in \( \sigma \).

   If \( p \) holds in \( s \) then by rule (6.54), we have \( l_0 \in s' \); on the other hand, because \( s' \) is closed under the static causal laws of \( D \), it follows that \( l \in s' \); this contradicts to the assumption \( l \in \sigma' \setminus s' \).

   So, we have \( p \) does not hold in \( s \). Similarly to previous case, this implies that there exists a fluent literal \( g \in p \) such that \( g \in \sigma \setminus s \). Hence, we have

   \[ l \triangleleft l_n \triangleleft l_{n-1} \triangleleft \ldots \triangleleft l_0 \triangleleft g \]

2. \( l \in s \).
First, we will show that $\phi h(\neg l, 1) \in B_1$. Since $l \not\in s'$, we have $h(l, 1) \not\in B_1$. By rule (6.59), it follows that $\phi h(\neg l, 1) \in B_1$.

According to Lemma 15, $\phi h(\neg l, 1) \in B_1$ implies that $\neg l$ is a possible effect of $a$ in $s$. By the definition of a possible effect (Definition 2), we have the following three cases

(a) $\neg l$ is a possible direct effect of $a$ in $s$. That is, $D$ contains a dynamic causal law

$$e \text{ causes } \neg l \text{ if } p$$

such that $e \in a$ and $p$ possibly holds in $s$. This implies that

$$\neg p \cap s = \emptyset \quad (6.89)$$

As $l \in \sigma'$ and $\sigma'$ is a state, we have $\neg l \not\in \sigma'$. This means that $h(\neg l, 1) \not\in A$. By rule (6.51), it follows that $\neg l$ is not a direct effect of $a$ in $\sigma$. Hence, $p$ does not hold in $\sigma$, i.e.,

$$\neg p \cap \sigma \neq \emptyset \quad (6.90)$$

From (6.89) and (6.90), it follows that there exists a fluent literal $g \in \neg p$ such that $g \in \sigma \setminus s$. Because $[e \text{ causes } \neg l \text{ if } p]$ belongs to $D$, this implies that $\neg l \triangleleft \neg g$. By the definition of dependencies, it follows that $l \triangleleft g$.

(b) $l$ does not hold in $s$ and $l$ is not a direct effect of $a$. Because $l \in s$, this case never happens.

(c) $D$ contains a sequence of static causal laws

$$[l_1 \text{ if } l_0], [l_2 \text{ if } l_1], \ldots, [l_n \text{ if } l_{n-1}], [\neg l \text{ if } l_n]$$

such that $l_0$ is a possible effect of $a$ in $s$ or $l_0$ possibly holds by inertia.

i. $l_0$ is a possible direct effect of $a$. By definition, this means that $D$
contains a dynamic causal law

\[ e \text{ causes } l_0 \text{ if } p \]

such that \( e \in a \) and \( p \) possibly holds in \( s \).

As \( p \) possibly holds in \( s \), we have

\[ \neg p \cap s = \emptyset \quad (6.91) \]

On the other hand, it is easy to see that \( p \) does not hold in \( \sigma \) as if otherwise, we would have \( \neg l \in \sigma' \), which contradicts to the assumption \( l \in \sigma' \). Therefore, we have

\[ \neg p \cap \sigma = \emptyset \quad (6.92) \]

By (6.91) and (6.92), there exists a fluent literal \( g \in \sigma \setminus s \) such that \( \neg g \in p \). It is easy to see that

\[ \neg l \triangleleft l_n \triangleleft l_{n-1} \triangleleft \ldots \triangleleft l_0 \triangleleft \neg g \]

Hence, we have \( l \triangleleft g \).

ii. \( l_0 \) possibly holds by inertia. This means that \( l_0 \) possibly holds in \( s \) and \( \neg l \) is not a direct effect of \( a \) in \( s \).

It is easy to see that \( l_0 \) does not hold in \( \sigma \) as if otherwise, we would have \( \neg l \in \sigma' \), which is impossible due to \( l \in \sigma' \). Because \( \sigma \) is a state, it follows that \( \neg l_0 \in \sigma \).

As \( l_0 \) possibly holds in \( s \) and \( \neg l_0 \in \sigma \), we have \( \neg l_0 \in \sigma \setminus s \). On the other hand, by the definition of dependencies, we have \( l \triangleleft \neg l_0 \).

Accordingly, we can select \( g = \neg l_0 \in \sigma \setminus s \) to have \( l \triangleleft g \).
6.5.2 Proof of Proposition 4

By the definition of $\gg_L$ (Definition 18), $s$ is a subset of every state $\sigma$ in $\Gamma$. Hence, the right hand side of the equation of the proposition is a subset of the left hand side. Therefore, to prove the lemma, it is sufficient to show that

$$ \bigcap_{\sigma \in \Gamma} \sigma \cap L \subseteq s \cap L $$

Suppose otherwise, that is, there exists a fluent literal $l$ such that $l \in \bigcap_{\sigma \in \Gamma} \sigma \cap L$ but $l \not\in s \cap L$. This implies that (i) $l \in L$, and (ii) $l \in \sigma \setminus s$ for every $\sigma \in \Gamma$. The latter implies that $l \triangleleft (\sigma \setminus s)$ for every $\sigma \in \Gamma$. By the definition of $\gg$, $\Gamma \gg_L s$. This is a contradiction.

6.5.3 Proof of Proposition 5

1. Assume that $a$ is not safe in $s$. That means there exists an impossibility condition

   impossible $b$ if $p$

such that $b \subseteq a$ and $p$ possibly holds in $s$, i.e.,

$$ \neg p \cap s = \emptyset $$ (6.93)

By the definition of $\gg$, $\Gamma \gg_L s$ implies that there exists a state $\sigma \in \Gamma$ such that $b \not\triangleleft (\sigma \setminus s)$. Because $a$ is executable in $\sigma$, $p$ does not hold in $\sigma$, i.e., $p \not\subseteq \sigma$. Because $\sigma$ is a complete set of fluent literals, it follows that

$$ \neg p \cap \sigma \neq \emptyset $$ (6.94)

By (6.93) and (6.94), there exists a fluent literal $l \in (\sigma \setminus s)$ such that $l \in \neg p$. By the definition of dependencies, we have $b \triangleleft l$ and this is a contradiction because $b \not\triangleleft (\sigma \setminus s)$.

2. Let $\Gamma' = \text{Res}(a, \Gamma)$.
Consider an arbitrary state $\sigma \in \Gamma$. Because $a$ is executable in $\Gamma$, it follows from the previous item that $a$ is safe in $s$. By Lemma 9, Proposition 1, and the definition of $T^\text{lp}(D)$, it follows that there exists a (unique) partial state $s'$ such that $\langle s, a, s' \rangle \in T^\text{lp}(D)$. We need to show that $\Gamma' \gg_L s'$.

Suppose otherwise, that is, $\Gamma' \not\gg_L s'$. Then, there are two possible cases (note that because $s \subseteq \sigma$ for every $\sigma \in \Gamma$, by Theorem 1, $s' \subseteq \sigma'$ for every $s' \in \Gamma'$):

(a) there exists a fluent literal $l \in L$ such that $l \triangleleft (\sigma' \setminus s')$ for every $\sigma' \in \Gamma'$.

Let $l$ be such a fluent literal. Consider an arbitrary state $\sigma \in \Gamma$ and let $\langle \sigma, a, \sigma' \rangle$ be a transition in $T(D)$. Furthermore, let $g \in \sigma' \setminus s'$ such that $l \triangleleft g$. By Proposition 3, because $g \in \sigma' \setminus s'$, there must be a fluent literal $h \in \sigma \setminus s$ such that $g \triangleleft h$. Because of the transitivity of $\triangleleft$, we have $l \triangleleft h$.

This implies that

$$l \triangleleft (\sigma \setminus s) \quad (6.95)$$

Because $\sigma$ can be any arbitrary state in $\Gamma$, (6.95) implies that $\Gamma' \not\gg_L s$. This is a contradiction.

(b) there exists an action $b$ such that $b \triangleleft (\sigma' \setminus s')$ for every $\sigma' \in \Gamma'$.

Consider an arbitrary state $\sigma \in \Gamma$ and let $\langle \sigma, b, \sigma' \rangle$ be a transition in $T(D)$. Furthermore, let $l \in \sigma' \setminus s'$ be a fluent literal such that $b \triangleleft l$.

By Proposition 3, because $l \in \sigma' \setminus s'$, there exists a fluent literal $g$ in $\sigma \setminus s$ such that $l \triangleleft g$. By the definition of dependencies, it follows that $b \triangleleft g \in (\sigma \setminus s)$. Hence, we have

$$b \triangleleft (\sigma \setminus s) \quad (6.96)$$

Because $\sigma$ can be any arbitrary state in $\Gamma$, (6.96) implies that $\Gamma' \not\gg_L s$. This is a contradiction.
6.5.4 Proof of Proposition 6

Let \( \alpha = \langle a_0, a_1, \ldots, a_{n-1} \rangle \). For \( i \geq 0 \), let \( \alpha[i] \) denote the chain of the initial \( i \) events of \( \alpha \), i.e., \( \alpha[i] = \langle a_0, a_1, \ldots, a_{i-1} \rangle \). We will prove the proposition by induction on the length \( n \) of \( \alpha \).

1. Base case: \( n = 0 \).

   Item 1 is trivial. Item 2 is true because \( \text{Res}(\alpha, \Gamma) = \Gamma \gg_L s \), \( \langle s, \langle \rangle, s \rangle \in T^{1p}(D) \) and \( T^{1p}(D) \) is deterministic.

2. Inductive Step: Suppose the proposition is true for \( n \leq k \). We need to show that it is true for \( n = k + 1 \).

   Let \( \Gamma_i = \text{Res}(\alpha[i], \Gamma) \) and let \( s_i \) be the partial state such that \( \langle s, \alpha[i], s_i \rangle \in T^{1p}(D) \). Clearly to prove the inductive step, we only need to show that

   (a) \( a_k \) is safe in \( s_k \), and
   (b) \( \Gamma_{k+1} \gg_L s_{k+1} \)

   By the inductive hypothesis, we have \( \Gamma_k \gg_L s_k \). By Proposition 5, it follows that \( a_k \) is safe in \( s_k \) and \( \Gamma_{k+1} \gg_L s_{k+1} \).

6.5.5 Proof of Theorem 3

This theorem follows directly from Proposition 6 and the definition of a simple planning problem (Definition 19). If \( \mathcal{P} \) has no solution then it is trivial. Suppose that \( \mathcal{P} \) has a solution, say, \( \alpha = \langle a_0, \ldots, a_{n-1} \rangle \). We will show that \( \pi(\mathcal{P}, n) \) is consistent.

Because \( \alpha \) is a solution of \( \mathcal{P} \), there exists a sequence of sets of states \( \langle \Gamma \rangle_{i=0}^n \) such that

1. \( \Gamma_0 = \text{comp}(s^0) \)
2. \( \Gamma_i = \text{Res}(a_{i-1}, S_{i-1}) \) for \( i \geq 1 \)
3. \( s^f \subseteq \sigma \) for every \( \sigma \in \Gamma_n \).
According to Proposition (6), there exists a sequence of partial states $\langle s_i \rangle^n_{i=0}$ such that $s_0 = s^0$ and $\Gamma_n \gg_s s_n$. By Proposition 4, we have $s^f \subseteq s_n$.

Let us construct a sequence of sets of atoms $D_i$ as follows:

$$D_0 = h(s^0, 0) \cup o(a_0, 0) \cup \neg o(A \setminus a_0, 0)$$

for $1 \leq i \leq n - 1$,

$$D_i = h(l, s_i) \cup o(a_i, i) \cup \neg o(A \setminus a_i, i)$$

$$\{de(l, i) | l \in de(a_{i-1}, s_{i-1})\} \cup \{ph(l, i) | l \in ph(a_{i-1}, s_{i-1})\}$$

and

$$D_n = \{h(l, n) | l \in s_n\} \cup$$

$$\{de(l, n) | l \in de(a_{n-1}, s_{n-1})\} \cup \{ph(l, n) | l \in ph(a_{n-1}, s_{n-1})\}$$

It is easy to see that for $0 \leq i \leq n$, $D_i$ is an answer set of $\mu_i$ (defined previously in section 6.3). By Lemma 12, $C_1 = \bigcup^n_{i=0} D_i$ is an answer set of $\pi_1(P, n)$. As a result, by Lemma 11, $C = C_1 \cup U$ is an answer set of $\pi_0(P, n)$.

We will show that $C$ is also an answer set of $\pi(P, n)$. Because $\pi(P, n)$ is the program $\pi_0(P, n)$ with additional constraints (6.3), (6.5), (3.20), and (3.22), all we need to do is to show that $C$ does not violate any of these constraints. For (6.3) and (6.5), it is trivial because $s_i$ is a partial state and $\langle s_{i-1}, a_{i-1}, s_i \rangle \in T^{lp}(D)$. Constraint (3.20) is satisfied by $C$ because $s^f \subseteq s_n$. Constraint (3.22) is satisfied because $a_i$ is an action, i.e., a non-empty set of elementary actions.

Hence, $C$ is an answer set of $\pi(P, n)$. This means that the program $\pi(P, n)$ is consistent. As a result, the theorem holds.
CHAPTER 7
RELATED WORK

In this section we describe related work in the fields of conformant planning, the use of approximations for conformant planning, and integration of ASP and constraint solving techniques.

7.1 Related Work on Conformant Planning

The conformant planning problem, as investigated in this work, has been discussed in [15, 18, 19, 21, 22, 23, 29, 32, 75, 78]. Several purpose-built conformant planning systems have arise from this work. Some of these systems, the ones closer in spirit to our system, were discussed in section 3.7.1.1. Other specialized systems include CFF [15], KACMBP [23], POND [19] and \( t_0 \) [74].

CFF (Conformant FF) extends the classical planner FF [48] to deal with uncertainty in the initial state. CFF uses a CNF formula to describe the initial belief state. A belief state \( s \) is represented by the initial state formula plus the sequence of actions that lead to \( s \). Reasoning is done by checking the satisfiability of CNF formulae. The input language of CFF is a subset of the PDDL language plus a minor extension to allow the user to specify the initial state as a CNF formula. CFF supports sequential conformant planning but does not supports concurrent and conditional conformant planning.

KACMBP is an extension of CMBP that uses techniques from Symbolic Model Checking to search in belief state space. KACMBP is guided by a heuristic function which is derived based on knowledge associated with a belief state. KACMBP is designed for both sequential and concurrent planning.

POND extends the planning graph algorithm [13], to deal with sensing actions and to support conformant planning. Its input language is a subset of PDDL.

\( t_0 \) is a conformant planning system that solves a problem \( P \) by translating it into a classical planning problem \( P' \).
Systems CFF, POND and $t_0$ does not support the use of static causal laws.

7.2 Related work to approximations

Another approximation is presented in [86]. The approximation in [86] is based on the idea of computing what can possibly change after an action is executed. The approximation $T^{lp}$, presented in this work, sometimes entails some conclusions that could not be derived using the approximation in [86] and vice versa.

Son and Tu [82, 83] [87] developed a heuristic search based planner, called CPA, using the $T^{lp}$ approximation. CPA uses a simple heuristic to direct search but its performance is comparable to state-of-the-art conformant planners. We believe that the good performance of CPA lies in the use of the approximation. In a more recent development, Tu worked to improve the search heuristic of CPA and entered an improved version of CPA into the International Planning Competition. The improved version of CPA won the best price in the conformant planning category.

7.3 Related work on integrating ASP and constraint solving techniques

There exists other efforts to integrate Answer Set Programming with constraint solving techniques besides the $\mathcal{AC}(C)$ approach presented here. $ADsolver$ [69], developed alongside $\mathcal{AC}(C)$, was the first tightly coupled solver that integrated ASP with constraint solving. Two other very promising approaches are EZCSP [4, 5] and Clingcon [36].

$ADsolver$ integrates ASP with difference constraints, and operates on the language $\mathcal{AC}_0$. In $\mathcal{AC}_0$, all the constraints are of the form $X - Y > K$, and they appear only on the bodies of rules without heads. The implementation of $ADsolver$ shares many components with $ACsolver$.

EZCSP is a loose coupling of ASP and the SICSTUS Prolog constraint solver. EZCSP extends the language of ASP with special predicates used to represent con-

---

1In the IPPC'08 planning competition, an improved version of CPA won the first price over $t_0$ (see http://ipcc-2008.loria.fr/wiki/index.php/Results)
straints such that each answer set of a program represents a constraint satisfaction problem. After answer sets are computed a constraint solver is used to solve the corresponding constraint solvers. Answer sets are considered valid only if the constraint problem they encode has a solution. EZCSP loose coupling allows it to be easily adapted to be used with different off-the-shelf constraint solvers.

Clingcon is a tight coupling of ASP and the constraint solver gecode. Clingcon extends the language of ASP with constraint atoms which are each associated with a constraint. During the computation of answer sets of Clingcon programs, the constraint solver gecode is used to verify that the constraints associated with the constraint atoms collected so far (considered true) still have a solution. Clingcon could potentially use different constraint solvers. Clingcon is built over Clingo, one of the most sophisticated answer set solvers, featuring back-jumping and conflict-driven learning techniques and its performance is state-of-the-art.
CHAPTER 8
CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

This dissertation investigates techniques for improving efficiency of solving computational problems in ASP using the conformant planning problem as the illustrating example. In particular, we investigated the use of Approximations of action theories to improve the efficiency of ASP based conformant planners, and developed an algorithmic framework to improve the efficiency of solvers of the $AC(C)$ language, an extension to ASP that integrates ASP with Constraint Logic Programming.

The contribution of this work is enumerated as follows:

1. We defined the notion of an approximation of action theories and proposed a deterministic approximation that can be computed efficiently by a program of ASP.
   - The proposed approximation is suitable for solving conformant planning problems when the initial state is represented by a set of fluent literals.

2. We develop a logic programming based planner using the proposed approximation called CPasp.
   - CPasp can generate both parallel and sequential plans.

3. We extended CPasp to handle initial situations that are described using disjunction.

4. We presented performance results over several benchmark domains as well as over newly invented domains against other comparable conformant planners.

5. We also provide a sufficient condition for the completeness of the approximation. The condition is applicable to simple action theories whose static causal laws are of a special form.
• It is worth to note that we found no problem in the literature that fails this condition.

6. We introduced an algorithmic framework to design programs for computing answer sets for the knowledge representation language \( \mathcal{A}\mathcal{C}(\mathcal{C}) \), a language that extends the syntax and semantics of ASP.

7. Presented an implementation of an instance of this framework over the prototype solver \( ACsolver \).

• system LUNA

8. We presented results on the efficiency improvements achieved by system LUNA in several problem domains.

9. Finally, we applied approximations and system LUNA to solve instances of conformant planning problems with numerical constraints.

8.2 Future work

The approximation presented in this work is not the only one possible. The search for more powerful approximations may continue. Different approximations may provide a different balance between completeness and computational complexity. The theoretical results presented here could also form the foundation for results for newer approximations.

We provide a sufficient condition for the completeness of the approximation. The condition is applicable to simple action theories whose static causal laws are of certain form. We do believe that this condition can be extended to cover a broader class of action theories.

The proposed algorithmic framework for building \( \mathcal{A}\mathcal{C}(\mathcal{C}) \) solvers can be used for the development of multiple reasoning algorithms and opens a wide avenue of research in better and more efficient algorithms. Different approaches to computing transitions between states of computation and different searching strategies of the
search space could be developed with the framework serving as a tool for showing algorithm correctness.

The definition of the language $\mathcal{AC}(C)$ could still be improved into a more elegant form. Also, the framework could also be extended to be used in the integration of Answer Set Programming with other constraint solving techniques.

The main ideas presented in this dissertation, the use of approximations and the $\mathcal{AC}(C)$ approach to answer set programming, are not limited in application to the conformant planning problem. Approximations could be used in any problem in which the main mathematical model is a transition diagram. $\mathcal{AC}(C)$, being able to operate on programs with variables that range over large domains, opens the door to areas of application previously beyond the reach of traditional ASP systems. Still, there is lots of work possible in making these systems more efficient and more reliable.
BIBLIOGRAPHY


8.3 Programs from Chapter 4.

8.3.1 House-Flowers domain.

% Comparison #00
% -c const=n
% acsolver -c hh=5 -c vv=5 comp00-trees.acc
%

h(0..hh).
v(0..vv).

field(X,Y) :- h(X), v(Y).
trees(1..5).

house(Y) :- v(Y).

1 { trees_at(X,Y,T) : trees(T) } 1 :- field(X,Y), h(X), v(Y).

#csort time(0..36000).
#mixed time_to_pay(v,time).

pay(Y) <~ house(Y), time_to_pay(Y,TT), mortgage(TT), v(Y).
:- not pay(Y), v(Y).

{: %start of defined part

% Standard mortgage relationship between:
% P: Principal
% T: Life of loan in months
% I: Fixed (but compounded) monthly interest rate
% B: Outstanding balance at the end
% M: Monthly payment
mg(P, T, I, B, MP) <-
  T = 1,
  B = P + (P*I - MP).
mg(P, T, I, B, MP) <-
  T > 1,
  mg(P*(1 + I) - MP, T - 1, I, B, MP).

mortgage(TT) <- printf("[#CLP(R)] Mortgage!!\n"),
  mg(999999, 720, 0.01, 0, TT).

go <-
  ztime, mg(999999, 360, 0.01, 0, M), ctime(T),
  printf("Time = %, M = %\n", [T, M]).

:} % end if defined part

#hide.
#show field(X,Y).
#show house(X,Y).
#show trees_at(X,Y,Z).

8.3.2 Ball drop, problem from action description H.

%% 9/8/2010
%% Brick drop example in the old AC syntax
%% This will be a translation into current runnable version of AC.
%% After great difficulty able to run to it with correct results
%% Avg time is 3.6 sec (user) + 21.07 (sys) = 25.5 (real)

%% Brick drop example in the latest syntax of H.
%% drop causes falling
%% catch causes -falling
%% impossible drop if falling
%% impossible catch if -falling
%% impossible drop if height(end)=0
%% impossible catch if height(end)=0
%% drop causes cur_ht=X if height(end)=X
%% catch causes cur_ht=X if height(end)=X
%% drop causes time_changed= T0 if end=T0
%% catch causes time_changed=T0 if end=t0
%% height = \lambda T. X- 4.9*(T-T0)^2 if cur_ht=X,
%% time_changed=T0,
%% falling.
%% height = \lambda T. X if cur_ht=X,
%% -falling.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Regular sorts
%%---------------
%% Sorts

%% Boolean values

b_val(true).
b_val(false).

#domain b_val(B;B1).

%% actions

action(drop).
action(catch).

#domain action(A).
%% Processes

fluent(falling).
range(falling,B).

%% Indices of the trajectory

const n=2.
step(0..n).
#domain step(I;I1;I2).

next(I,I+1):- I<n.

%% Constraint sorts
%%-------------------

%% Defining time sort

#csort time(0..60).

#domain time(T;T1;T2).

%% Defining meters

#csort meters(0..500).

#domain meters(X;X1).

%% Declaration of mixed predicates
%%-----------------------------------

#mixed start(step,time).

#mixed end(step,time).

%% mixed predicates for cur_ht and time_changed

#mixed ch(step,meters).
#mixed tc(step,time).

#mixed height(step,meters).  \%\% This will denote height at the 
\%\% end of each step

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\%\% Domain Independent axioms
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

\%\% Axioms to define start and end times of a step

\%\% Start time of step 0 is 0
\<\~ start(0,T),
\T!\=0.

\%\% Start time of every step is defined in terms of
\%\% end time of previous step
\<\~ start(I1,T1),
next(I,I1),
occurs(A,I),
end(I,T),
\T!\=T1.

\%\% Uniqueness axiom
\%\% A fluent has a unique value in every step

\~\text{val}(Y1,P,I) :- \text{val}(Y2,P,I),
\Y1!\=Y2,
\text{fluent}(P),
\text{range}(P,Y1),
\text{range}(P,Y2).

\%\% Inertia axiom

\text{val}(Y,P,I1) :- \text{val}(Y,P,I),

126
not -val(Y,P,I1),
next(I,I1),
I<n,
fluent(P),
range(P,Y).

ab(I) :- occurs(drop,I).
ab(I) :- occurs(catch,I).

-ab(I) :- not ab(I).

%%% Inertia for cur_ht and time_changed

<~ ch(I,X),
   next(I,I1),
   ch(I1,X1),
   I<n,
   -ab(I),       %% not ab(I) is giving problems
   X!= X1.

<~ tc(I,T),
   tc(I1,T1),
   next(I,I1),
   I<n,
   -ab(I),
   T != T1.

%%% Height is well defined so no need for inertia.

%%%--------------------------------------------------------------
%%% Domain Dependent axioms
%%%--------------------------------------------------------------

%%% 1. Effects of "drop" and "catch"

%%% 1a. falling

%%% drop causes falling
\begin{verbatim}
val(true,falling,I+1) :- occurs(drop,I).

%%% impossible drop if falling
-occurs(drop,I) :- val(true,falling,I).

%%% catch causes -falling
val(false,falling,I+1) :- occurs(catch,I).

%%% impossible catch if -falling
-occurs(catch,I) :- val(false,falling,I).

%%%******************************************************************************
%%% 1b. cur_ht
%%% drop causes cur_ht=X if height(end)=X

%%% These are middle rules
<~ occurs(drop,I),
    height(I,X),
    next(I,I1),
    I<n,
    ch(I1,X1),
    X1!= X.

%%% catch causes cur_ht if height(end) = X

<~ occurs(catch,I),
    height(I,X),
    next(I,I1),
    I<n,
    ch(I1,X1),
    X1!= X.

%%%******************************************************************************
%%% 1c. time_changed
\end{verbatim}
%% drop causes time_changed= T0 if end=T0

%% Middle rules

<~ occurs(drop,I),
    end(I,T),
    next(I,I1),
    tc(I1,T1),
    T!=T1.

%% catch causes time_changed=T0 if end=t0

<~ occurs(catch,I),
    end(I,T),
    next(I,I1),
    tc(I1,T1),
    T!=T1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% 2. Definition of height

%% We use middle rules to define height at end of each step

%% height = \lambda T. X- 4.9*(T-T0)^2 if cur_ht=X,
%% time_changed=T0,
%% falling.

compute_ht(I)<~ val(true,falling,I),
    ch(I,X),
    tc(I,T),
    end(I,T1),
    height(I,X1),
    d(X,X1,T,T1).

%% Forcing to compute height

- compute_ht(I):-not compute_ht(I).

:- - compute_ht(I),
val(true, falling, I).

%% Having trouble writing this equation as part of above rule
%% X_1 != X - 4.9*(T_1-T)*(T_1-T).

%% so introduced the defined predicate d(X, X_1, T, T_1)

%% height = \lambda T. X if cur_ht=X,
%% -falling.
<~ val(false, falling, I),
    ch(I, X),
    height(I, X_1),
    X_1!=X.

{: 
%% max(x,y) works!

d(X, X_1, T, T_1) <- X_1 = max(0, X - 4.9*(T_1-T)*(T_1-T)).

:}

%% Executability conditions involving "height"

%% Impossibility condition - "impossible drop if height(end)=0"
%% (middle rule)

%% "drop" cannot take place in any state in which the height
%% at the end of the state is 0.
<~ occurs(drop, I),   %% This works!
    height(I, X),
    X==0.

%% Impossibility condition - "impossible catch if height(end)=0"
%% (middle rule)
%% "catch" cannot take place in any state in which the height at
%% the end of the state is 0.

<~ occurs(catch, I), %% This works!
    height(I, X),
    X==0.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Initial situation

val(true, falling, 0). %% Ball is falling

<~ ch(0, X), %% cur_ht=500
    X!=500.

<~ tc(0, T),
    T!=0. %% time_changed=0

%% Sequence of actions

occurs(catch, 0).

<~ end(0, T),
    T!=2.

occurs(drop, 1).
<~ end(1, T),
    T!=5.

occurs(catch, 2). % (Could get rid of this action)
<~ end(2, T),
    T!=10.

% occurs(drop, 2).
%<~ end(2, T),
%    T!=10.

% occurs(catch, 3).
8.4 Programs from Chapter 5.

8.4.1 Bomb in the toilet with timers.

%-------------------------------------------------------------
% % Bomb in The Toilet Problem with Timers!
% % % Multiple Packages, Bomb, Multiple Toilets, Clogging
% % % Using CPasp_m encoding. (As in the IJCAI Paper)
% % % Timers! bomb will explode in 300 seconds!
% % Flushing takes 30 seconds to complete
% %-------------------------------------------------------------
% % % External Constants:
% % % nnn : no of timesteps
% % ppp : no of packages
% % eee : no of toilets
% % % acsolver -c nnn=3 -c ppp=2 -c eee=1 bmtc-timer.ac
% %

% --------------
% % Declarations
% % --------------
#domain timestep(S,S1,S2).
#domain fluent(F,F1).
#domain literal(L;L1;L2;L3).
#domain label(K).

#domain pkg(P;P0;P1).
#domain toilet(E;E1;E2).

% ----------------
% Rules, Fluents, Literals, Actions, time-steps
% ----------------
timestep(0..nnn).
nextstep(S,S+1) :- timestep(S+1).

label(1).
literal(F).
literal(neg(F)).

fluent(armed(P)).
fluent(clogged(E)).

action(dunk(P,E)).
action(flush(E)).

% -------------------------------------------------------
% Objects
% -------------------------------------------------------
pkg(1..ppp).
toilet(1..eee).

% -------------------------------------------------------
% Causal Law Encoding
% -------------------------------------------------------
%(1) Dumping a package disarms the package
h(neg(armed(P)),S+1,K) :- o(dunk(P,E),S).
dc(neg(armed(P)),S+1,K) :- o(dunk(P,E),S).
ph(neg(armed(P)),S+1,K) :- o(dunk(P,E),S).

%(2) Dumping a package cloggs the toilet.
%
% dump(P,E) causes clogged(E)  
%  
\[ h(\text{clogged}(E), S+1, K) :- o(\text{dunk}(P, E), S). \]  
\[ dc(\text{clogged}(E), S+1, K) :- o(\text{dunk}(P, E), S). \]  
\[ ph(\text{clogged}(E), S+1, K) :- o(\text{dunk}(P, E), S). \]  

% (3) Flushing a toilet disarms the toilet  
%  
% flush(E) causes neg(clogged(E))  
%  
\[ h(\text{neg}(\text{clogged}(E)), S+1, K) :- o(\text{flush}(E), S). \]  
\[ dc(\text{neg}(\text{clogged}(E)), S+1, K) :- o(\text{flush}(E), S). \]  
\[ ph(\text{neg}(\text{clogged}(E)), S+1, K) :- o(\text{flush}(E), S). \]  

%(4) impossible to dump into a cloged toilet  
%  
% impossible dump(P,E) if cloged(E)  
%  
\[ :- o(\text{dunk}(P, E), S), \text{not} \ h(\text{neg}(\text{clogged}(E)), S, K). \]  

%--------  
% Imposibilities For Parallel Plans.  

%(5) impossible to dump to a toilet and flush the toilet at the same time  
\[ :- o(\text{dunk}(P, E), S), o(\text{flush}(E), S). \]  

%(6) impossible to dump two packages into the same toilet at the same time.  
\[ :- 2\{o(\text{dunk}(P, E), S) : \text{pkg}(P)\}. \]  

%(7) impossible to dump same package to two toilets at the same time  
\[ :- 2\{o(\text{dunk}(P, E), S) : \text{toilet}(E)\}. \]  

%---------------------------------------------  
% Control Sentences  
%---------------------------------------------  
% impossibilities  
%impossible to dump a package that is already disarmed  
\[ :- o(\text{dunk}(P, E), S), h(\text{neg}(\text{armed}(P)), S, K). \]
% impossible to flush a toilet that is not clogged
% NOTE: here \( -h(\text{clogged}) = eq = not h(\text{clogged}) \)
:- o(flush(E),S), not h(clogged(E),S,K).
%
% Don't flush a toilet if there are toilets available.
:- o(flush(E1),S), not h(clogged(E2),S,K), E1 != E2.

% -------------------------------------------------------
% Domain Independent and Inertia Axioms
% -------------------------------------------------------

:- h(F,S,K), h(neg(F),S,K).

%%% Inertia
ph(F,S+1,K) :- not h(neg(F),S,K), not dc(neg(F),S+1,K), S < nnn.
ph(neg(F),S+1,K) :- not h(F,S,K), not dc(F, S+1,K), S < nnn.

h(F,S,K) :- not ph(neg(F),S,K), S > 0.
h(neg(F),S,K):- not ph(F,S,K), S > 0.

% -------------------------------------------------------
% Initial Situation
% -------------------------------------------------------
% all toilets are free..
h(neg(clogged(X)),0,K) :- toilet(X).

% ---------------
% Goal
% ---------------
goal(neg(armed(P))).

% ---------------
% Goal & Concurrent Planner
% ---------------

1{o(A, S) : action(A)} :- S < nnn.
%----------------------- goal
% may_fail
:- goal(L), not h(L, nnn, K).
%:- success

% %-----------------------------------------------
% % Timing Constraints
% %-----------------------------------------------
% Time
% Goal must be achieved in less than 300 seconds
#csort time(0..240).

#mixed at_time(timestep,time).

% % times should be assigned sequentially
<~ nextstep(S1, S2), at_time(S1, T1), at_time(S2, T2), gt(T1 - T2, 0).

% % flushing a toilet takes 30 seconds.
<~ o(flush(E1),S1), nextstep(S1,S2), at_time(S1,T1), at_time(S2,T2),
   gt(T1 - T2, -30).

% Goal must be achieved in less than 300 seconds

% %-----------------------------------------------
% % Format Operators...
% %-----------------------------------------------

hide.
show o(X,Y).
%show h(X,Y).
%show may_fail(X).
%show ab(X,Y).
show goal.
8.4.2 Forrest bus rides.

%
% acsolver -c nnn=4 -c ccc=4 forrest.ac
%
% -------------------------------------------------------------
%
% Forrest’s bus trip.
%
%-------------------------------------------------------------
%
% External Constants:
%
% nnn : no of timesteps
% ccc : no of stops in the trip
%
% ----------------
% Declarations
% ----------------
#domain timestep(S;S1;S2).
#domain fluent(F;F1).
#domain literal(L;L1;L2;L3).
#domain stops(C;C0;C1).

% -------------------------------------------------------
% Objects
% -------------------------------------------------------
timestep(0..nnn).
nextstep(S,S+1) :- timestep(S+1).

stops(0..ccc).

label(1).
literal(F).
literal(neg(F)).

% ----------------
% Rules, Fluents, Literals, Actions, time-steps
% ----------------

fluent(at(C)).
fluent(ontime_bus_to(C)).

action(take_bus_at(C)).

% -------------------------------------------------------
% Causal Law Encoding
% -------------------------------------------------------

%(1) taking a bus takes forest to the next city
h(at(C+1),S+1) :- o(take_bus_at(C),S).

%(1) taking a bus removes forest from a city
h(neg(at(C)),S+1) :- o(take_bus_at(C),S).

%(3) impossible to take a bus if not at that city.
:- o(take_bus_at(C),S), not h(at(C),S).

%-----------------------------------------------------------
% Control Sentences
%-----------------------------------------------------------

%-----------------------------------------------------------
% Domain Independent and Inertia Axioms
%-----------------------------------------------------------

:- h(F,S,K), h(neg(F),S,K).

%%% Inertia

%%%ph(F,S+1,K) :- not h(neg(F),S,K), not dc(neg(F),S+1,K), S < nnn.
%%%ph(neg(F),S+1,K) :- not h(F,S,K), not dc(F, S+1,K), S < nnn.
\%h(F,S,K) :- not ph(neg(F),S,K), S > 0.
\%h(neg(F),S,K):- not ph(F,S,K), S > 0.

%%%%%% Inertia
h(F,S+1) :- h(F,S), not h(neg(F),S+1), S > 0, S < nnn.
h(neg(F),S+1) :- h(neg(F),S), not h(F,S+1), S > 0, S < nnn.

% Initial Situation
% -----------------------------------------------
% all toilets are free..
h(at(0),0).

% Test Sample Action sequence
% -----------------------------------------------
o(take_bus_at(0),0).
o(take_bus_at(1),1).
o(take_bus_at(2),2).

% Goal
% -----------------------------------------------
goal(at(ccc)).

% Goal & Concurrent Planner
% -----------------------------------------------
1{o(A, S) : action(A)}1 :- S < nnn.

%----------------------- goal
% may_fail
:- goal(L), not h(L, nnn).
%:- success
% -----------------------------------
% Timing Constraints
% -----------------------------------

% Time
% Time

csort time(0..10000).

#mixed at_time(timestep,time).

% times should be assigned sequentially
% <- nextstep(S1, S2), at_time(S1, T1), at_time(S2, T2), gt(T1 - T2, 0).

% Buses must be booked from the time-table.
booked(S) <- o(take_bus_at(C),S), at_time(S, T), bus_available(T), timestep(S).
:- not booked(S), timestep(S), S < nnn.

% Buses must be booked sequentially
<- nextstep(S1, S2), booked(S1), booked(S2), at_time(S1, T1), at_time(S2, T2),
  bef(T2, T1).

% A bus takes 80 minutes to go to the next city
<- nextstep(S1, S2), booked(S1), booked(S2), at_time(S1, T1), at_time(S2, T2),
  gt(T1 - T2, -80).

% A bus takes at most 130 minutes if it is late.
late_at(S) :- not h(ontime_bus_to(C),S).
<- nextstep(S1, S2),
  booked(S1), booked(S2), late_at(S1),
  at_time(S1, T1), at_time(S2, T2), gt(T1 - T2, -130).

{: %start of defined part

bus_at(X) <- X = 60.
bus_at(X) <- X = 120.
bus_at(X) <- X = 180.
bus_at(X) <- X = 240.
bus_at(X) <- X = 300.
bus_at(X) <- X = 360.
bis_at(X) <- X = 420.
bis_at(X) <- X = 480.
bis_at(X) <- X = 540.
bis_at(X) <- X = 600.
bis_at(X) <- X = 660.
bis_at(X) <- X = 720.
bis_at(X) <- X = 780.
bis_at(X) <- X = 840.
bis_at(X) <- X = 900.
bis_at(X) <- X = 960.
bis_at(X) <- X = 1020.
bis_at(X) <- X = 1080.
bis_at(X) <- X = 1140.
bis_at(X) <- X = 1200.
bis_at(X) <- X = 1260.
bis_at(X) <- X = 1320.
bis_at(X) <- X = 1380.
bis_at(X) <- X = 1440.
bis_at(X) <- X = 1500.
bis_at(X) <- X = 1560.
bis_at(X) <- X = 1620.
bis_at(X) <- X = 1680.
bis_at(X) <- X = 1740.
bis_at(X) <- X = 1800.
bis_at(X) <- X = 1860.
bis_at(X) <- X = 1920.
bis_at(X) <- X = 1980.
bis_at(X) <- X = 2040.
bis_at(X) <- X = 2100.
bis_at(X) <- X = 2160.
bis_at(X) <- X = 2220.
bis_at(X) <- X = 2280.
bis_at(X) <- X = 2340.
bis_at(X) <- X = 2400.

bus_available(X) <- printf("[CLP(R)]!!\n"), bus_at(X).

bef(X2, X1) <- X2 <= X1.
8.4.3 Airport pickups.

% acsolver -c nnn=4 airpick.ac
% %-----------------------------------------------
% % Airport Pickups
% %-----------------------------------------------
% % Declarations
% %-------------------
#domain timestep(S;S1;S2).
#domain fluent(F;F1).
#domain literal(L;L1;L2;L3).
#domain label(K).

#domain person(P;P0;P1).
#domain veh(E;E1;E2).
#domain location(R;R0;R1;R2).

% -------------------------------------------------------
% Objects
% -------------------------------------------------------
person(alpha0).
person(beta0).
person(gamma0).
veh(car0).
veh(van0).
veh(truck0).

location(campus).
location(gas_station).
location(airport).

% ----------------
% Rules, Fluents, Literals, Actions, time-steps
% ----------------
timestep(0..nnn).
nextstep(S,S+1) :- timestep(S+1).

label(1).
literal(F).
literal(neg(F)).

fluent(at(E,R)).
fluent(fueled(E)).
fluent(picked(P)).
fluent(loaded(E)).

action(drive_to(E,R)).
action(fuel(E)).
action(pick(E,P)).

% -------------------------------------------------------
% Causal Law Encoding
% -------------------------------------------------------
%(1a) Driving to a location takes a vehicle to that location
\[h(\text{at}(E,R),S+1,K) :- o(\text{drive}_\text{to}(E,R),S).\]
\[\text{dc}(\text{at}(E,R),S+1,K) :- o(\text{drive}_\text{to}(E,R),S).\]
\[\text{ph}(\text{at}(E,R),S+1,K) :- o(\text{drive}_\text{to}(E,R),S).\]

%(1b) Driving to a location makes a vehicle not at any other location
\[h(\neg(\text{at}(E,R1)),S+1,K) :- o(\text{drive}_\text{to}(E,R0),S), R1 \neq R0.\]
\[\text{dc}(\neg(\text{at}(E,R1)),S+1,K) :- o(\text{drive}_\text{to}(E,R0),S), R1 \neq R0.\]
\[\text{ph}(\neg(\text{at}(E,R1)),S+1,K) :- o(\text{drive}_\text{to}(E,R0),S), R1 \neq R0.\]

%(2a) Fueling fuels the car
\[h(\text{fueled}(E),S+1,K) :- o(\text{fuel}(E),S).\]
\[\text{dc}(\text{fueled}(E),S+1,K) :- o(\text{fuel}(E),S).\]
\[\text{ph}(\text{fueled}(E),S+1,K) :- o(\text{fuel}(E),S).\]

%(2b) impossible to fuel if not at gas station
\[- o(\text{fuel}(E),S), \text{not } h(\text{at}(E,\text{gas}\_\text{station}),S,K).\]

%(3) impossible to drive to airport if not fueled
\[- o(\text{drive}_\text{to}(E,\text{airport}),S), \text{not } h(\text{fueled}(E),S,K).\]

%(4a) Picking picks a person
\[h(\text{picked}(P),S+1,K) :- o(\text{pick}(E,P),S).\]
\[\text{dc}(\text{picked}(P),S+1,K) :- o(\text{pick}(E,P),S).\]
\[\text{ph}(\text{picked}(P),S+1,K) :- o(\text{pick}(E,P),S).\]

%(4b) impossible pick a person with a vehicle if vehicle is not at airport.
\[- o(\text{pick}(E,P),S), \text{not } h(\text{at}(E,\text{airport}),S,K).\]

%(5a) Picking loads a vehicle
\[h(\text{loaded}(E),S+1,K) :- o(\text{pick}(E,P),S).\]
\[\text{dc}(\text{loaded}(E),S+1,K) :- o(\text{pick}(E,P),S).\]
\[\text{ph}(\text{loaded}(E),S+1,K) :- o(\text{pick}(E,P),S).\]

%(4b) impossible pick a person with a vehicle that is loaded.
\[- o(\text{pick}(E,P),S), h(\text{loaded}(E),S,K).\]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Impossibilities For Parallel Plans.

%(5) impossible to fuel and drive at the same time
:- o(fuel(E),S), o(drive_to(E,R),S).

%(5) impossible to pick and drive at the same time
:- o(pick(E,P),S), o(drive_to(E,R),S).

%(6) impossible to pick two people at the same time.
:- 2{o(pick(E,PX),S) : person(PX)}.

%-----------------------------------------------------------
% Control Sentences
%-----------------------------------------------------------
% impossibilities
% impossible to pick a person who is already picked
:- o(pick(E,P),S), h(picked(P),S,K).

%-----------------------------------------------------------
% Domain Independent and Inertia Axioms
%-----------------------------------------------------------
:- h(F,S,K), h(neg(F),S,K).

%%% Inertia
ph(F,S+1,K) :- not h(neg(F),S,K), not dc(neg(F),S+1,K), S < nnn.
ph(neg(F),S+1,K) :- not h(F,S,K), not dc(F, S+1,K), S < nnn.

h(F,S,K) :- not ph(neg(F),S,K), S > 0.
h(neg(F),S,K):- not ph(F,S,K), S > 0.

%-------------------------------------------
% Initial Situation
%-------------------------------------------
\% all vehicles are at campus.
h(at(X,campus),0,K) :- veh(X).

\% ----------------
\% Goal
\% ----------------
goal(picked(P)).

\% ----------------
\% Goal & Concurrent Planner
\% ----------------
1\{o(A, S) : action(A)\} :- S < nnn.

\%----------------------- goal
\% may_fail
: - goal(L), not h(L, nnn, K).
\%: - success

\% ------------------------- Test Sequence
\% -------------------------
\%o(drive_to(car,gas_station),0).
\%o(fuel(car),1).
\%o(drive_to(car,airport),2).
\%o(pick(car,alpha),3).

\% ------------------------- Real Number Constraints
\% -------------------------
#csort real(0..500).

#mixed weight_load(veh,real).

\%
\% Vehicles are loaded with a weight.
\% weighted(E) `<~ o(pick(E,P),S), weight_load(E,WW), total_w(P,WW).
\% :- not weighted(E), loaded(E).
% Impossible for a car to weight more than its capacity
% <~ weighted(E), weight_load(E,WW), over_capacity(E,WW).
%

% Vehicles are loaded with a weight.
weighted(E) <~ o(pick(E,P),S), weight_load(E,WW), wload(E,P,WW).
:- not weighted(E), h(loaded(E),S,K).

{:: %start of defined part

capacity(car0,30.0).
capacity(van0,40.0).
capacity(truck0,100.0).

bags(alpha0,[5.34, 17.561 ,10.24]).
bags(beta0,[19.0, 10.01]).
bags(gamma0,[3.14159, 25.01, 24.9, 33.333]).

sum([], 0).
sum([H |T], H + S) <- sum(T, S).

wload(E,P,WW) <-
   printf("[#CLP(R)] wload( % , % , WW )!!\n",[E, P]),
bags(P, L),
   sum(L,WW),
capacity(E,C),
   WW <= C.

} % end of defined part

% -------------------------------
% Format Operators...
% -------------------------------

hh(F,S,K) :- h(F,S,K).

#hide.
#show o(X,Y).
% #show h(X,Y,Z).
% #show hh(X,Y,K).
% #show may_fail(X).

147
% #show ab(X,Y).
#show goal.
#show loaded(E,S,K).

.